emRun

A small, efficient C runtime library

User Guide & Reference Manual

Document: UM12007
Software Version: 4.22.0
Revision: 0
Date: November 2, 2022

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2003-2022 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	`support@segger.com`[*]
Internet:	*www.segger.com*

[*]By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: November 2, 2022

Software	Revision	Date	By	Description
4.22.0	0	221101	PC	Updated to latest software version.
4.20.0	0	221007	PC	Updated to latest software version.
4.18.0	0	220613	PC	Chapter “C library API” • Added `memset_s()`, `memcpy_s()`, `memmove_s()`. • Added `strnlen_s()`, `strcpy_s()`. • Added `abort_handler_s()`, `ignore_handler_s()`. • Added `set_constraint_handler_s()`. Chapter “Compiler support API” • Added support for atomic datatype operations.
4.16.0	0	220429	PC	Chapter “C library API” • Added `printf_l()`, `sprintf_l()`, `snprintf_l()`, `vprintf_l()`. • Added `vsprintf_l()`, `vsnprintf_l()`, `fprintf_l()`, `vfprintf_l()`. • Added `asprintf_l()`, `vasprintf_l()`. • Added `scanf_l()`, `sscanf_l()`, `vscanf_l()`, `vsscanf_l()`. • Added `fscanf_l()`, `vfscanf_l()`.
4.14.1	0	220426	PC	Updated to latest software version.
4.14.0	0	220413	PC	Chapter “Runtime support” • Added section “Thread safety”. Chapter “C library API” • Added section “<signal.h>”. Chapter “Compiler support API” • Added section “Environment support”. Chapter “GNU library API” • Added section “Complex arithmetic”.
4.12.1	0	220321	PC	Updated to latest software version.
4.12.0	0	220317	PC	Updated to latest software version.
4.10.0	0	220120	PC	Chapter “C library API” • Added `aligned_alloc()`. • Added `asprintf()`, `vasprintf()`. • Added `uselocale()`. • Added `isascii()`, `isascii_l()`. • Added `wcstol()`, `wcstoll()`, `wcstoul()`, `wcstoull()`. • Added `wcstof()`, `wcstod()`, `wcstold()`. • Added `mbsnrtowcs()`, `mbnsrtowcs_l()`. • Added `wcsnrtombs()`, `wcsnrtombs_l()`. • Added `strcoll()`, `wcscoll()`. • Added `strxfrm()`, `wcsxfrm()`. Chapter “Compiling emRum” • Removed section “Heap size”. Chapter “Runtime support” • Added section “Dynamic storage and the heap”. Chapter “External function interface” • Added `__SEGGER_RTL_init_heap()`.
3.10.1	0	211223	PC	Chapter “C library API” • Added `__multi3()`. Chapter “External function interface” • Added `__SEGGER_RTL_X_file_bufsize()`.
3.10.0	0	211208	PC	Chapter “C library API” • Added `__divti3()`, `__modti3()`. • Added `__udivti3()`, `__umodti3()`. • Added `strnlen_s()`. • Added `fopen()`, `freopen()`, `fclose()`. • Added `fread()`, `fwrite()`. • Added `feof()`, `ferror()`, `clearerr()`. • Added `fprintf()`, `vfprintf()`, `fscanf()`, `vfscanf()`. • Added `fputc()`, `fgetc()`, `fputs()`, `fgets()`. • Added `fsetpos()`, `fgetpos()`, `fseek()`, `ftell()`, `rewind()`. • Added `rename()`, `remove()`, `perror()`.

Software	Revision	Date	By	Description
				Chapter “External function interface” • Added __SEGGER_RTL_X_file_open(). • Added __SEGGER_RTL_X_file_error(). • Added __SEGGER_RTL_X_file_end(). • Added __SEGGER_RTL_X_file_stat(). • Added __SEGGER_RTL_X_file_flush(). • Added __SEGGER_RTL_X_file_getpos(). • Added __SEGGER_RTL_X_file_seek(). • Added __SEGGER_RTL_X_file_clrerr(). • Added __SEGGER_RTL_X_file_close(). • Added __SEGGER_RTL_X_file_rename(). • Added __SEGGER_RTL_X_file_remove(). • Added __SEGGER_RTL_X_file_tmpnam(). • Added __SEGGER_RTL_X_file_tmpfile().
2.30.0	0	211122	PC	Chapter “Compiling emRun” • Added section for half-precision float configuration. Chapter “GNU library API” • Added __fixhfsi(), __fixhfdi(). • Added __fixunshfsi(), __fixunshfdi(). • Added __floatsihf(), __floatdihf(). • Added __floatunsihf(), __floatundihf(). • Added __extendhfsf2(), __extendhfdf2(), __extendhftf2(). • Added __trunctfhf2(), __truncdfhf2(), __truncsfhf2(). • Added __eqhf2(), __nehf2(). • Added __lthf2(), __gthf2(). • Added __lehf2(), __getf2().
2.28.2	0	211120	PC	Chapter “Compiling emRun” • Added __SEGGER_RTL_NO_BUILTIN. Chapter “C library API” • Added sincos(), sincosf(), sincosl().
2.28.1	0	211102	PC	Updated to latest software version.
2.28.0	0	211029	PC	Section “Configuring for RISC-V” • Added stack alignment configuration. Chapter “C library API” • Added __popcountsi2(), __popcountdi2(). • Added __paritysi2(), __paritydi2().
2.26.1	0	210922	PC	Updated to latest software version.
2.26.0	0	210920	PC	Chapter “C library API” • Added section “<fenv.h>”. Chapter “Compiling emRun” • Expanded section “General configuration”. • Added section “Configuring for Arm”. • Added section “Configuring for RISC-V”. Section “Benchmarks” • Added. Section “Input and output” • Expanded and rewritten.
2.24.0	0	210811	PC	Chapter “C library API” • Added strtold().
2.22.0	0	210803	PC	Updated to latest software version.
2.20.0	0	210714	PC	Chapter “C library API” • Added sinl(), cosl(), tanl(). • Added asinl(), acosl(), atanl(), atan2l(). • Added sinhl(), coshl(), tanhl(). • Added asinhl(), acoshl(), atanhl(). • Added logl(), log2l(), log10l(), logbl(), log1pl(), ilogbl(). • Added expl(), exp2l(), exp10l(), expm1l(). • Added sqrtl(), cbrtl(), rsqrtl(), hypotl(), powl(). • Added fminl(), fmaxl(), fdiml(), fabsl(). • Added tgammal(), lgammal(), erfl(), erfcl(). • Added ceill(), floorl(), truncl(). • Added rintl(), lrintl(), llrintl(). • Added roundl(), lroundl(), llroundl(). • Added fmodl(), remainderl(), remquol(). • Added modfl(), frexpl(), ldexpl(), scalbnl(), scalblnl(). • Added nearbyintl(), fmal(). • Added copysignl(), nextafterl(), nexttowardl(), nanl(). Chapter “GNU library API”

Software	Revision	Date	By	Description
				• Added `__addtf3()`, `__subtf3()`, `__multf3()`, `__divtf3()`. • Added `__fixtfsi()`, `__fixtfdi()`, `__fixunstfsi()`. • Added `__fixunstfdi()`, `__floatsitf()`, `__floatditf()`. • Added `__floatunsitf()`, `__floatunditf()`, `__extendsftf2()`. • Added `__extenddftf2()`, `__trunctfdf2()`, `__trunctfsf2()`. • Added `__eqtf2()`, `__netf2()`, `__lttf2()`, `__letf2()`, `__gttf2()`. • Added `__getf2()`.
2.4.2	0	210225	PC	Chapter “C library API” • Added `trunc()`, `truncf()`. • Added `scalbln()`, `scalblnf()`.
2.4.0	0	201101	PC	Chapter “C library API” • Added `stpcpy()` and `stpncpy()`. • Added `nan()` and `nanf()`. • Added `copysign()` and `copysignf()`. • Added `nextafter()` and `nextafterf()`. • Added `nexttoward()` and `nexttowardf()`. • Added `remainder()` and `remainderf()`. • Added `remquo()` and `remquof()`. • Added `lgamma()` and `lgammaf()`. • Added `tgamma()` and `tgammaf()`. • Added `erf()` and `erff()`. • Added `erfc()` and `erfcf()`. • Added `csin()` and `csinf()`. • Added `ccos()` and `ccosf()`. • Added `ctan()` and `ctanf()`. • Added `casin()` and `casinf()`. • Added `cacos()` and `cacosf()`. • Added `catan()` and `catanf()`. • Added `csinh()` and `csinhf()`. • Added `ccosh()` and `ccoshf()`. • Added `ctanh()` and `ctanhf()`. • Added `casinh()` and `casinhf()`. • Added `cacosh()` and `cacoshf()`. • Added `catanh()` and `catanhf()`. • Added `clog()` and `clogf()`. • Added `cexp()` and `cexpf()`. • Added `cabs()` and `cabsf()`. • Added `carg()` and `cargf()`. • Added `creal()` and `crealf()`. • Added `cimag()` and `cimagf()`. • Added `cproj()` and `cprojf()`. • Added `conj()` and `conjf()`.
2.12	0	191220	PC	Chapter “C library API” • Added `expm1f()`.
2.10	0	190307	PC	Release version.
1.00	0	190204	PC	Internal version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
`User Input`	Text entered at the keyboard by a user in a session transcript.
`Secret Input`	Text entered at the keyboard by a user, but not echoed (e.g. password entry), in a session transcript.
Reference	Reference to chapters, sections, tables and figures.
Emphasis	Very important sections.
SEGGER home page	A hyperlink to an external document or web site.

Table of contents

© 2003-2022 SEGGER Microcontroller GmbH

Chapter 1

Introduction

This section presents an overview of emRun, its structure, and its capabilities.

1.1 What is emRun?

emRun is an optimized C library for Arm and RISC-V processors.

1.2 Features

emRun is written in standard ANSI C and Arm assembly language and can run on any Arm or RISC-V CPU. Here’s a list summarising the main features of emRun:

- Clean ISO/ANSI C source code.
- Fast assembly language floating point support.
- Conforms to standard runtime ABIs for the Arm and RISC-V architectures.
- Simple configuration.
- Royalty free.

1.3 Recommended project structure

We recommend keeping emRun separate from your application files. It is good practice to keep all the program files (including the header files) together in the `LIB` subdirectory of your project's root directory. This practice has the advantage of being very easy to update to newer versions of emRun by simply replacing the `LIB` directory. Your application files can be stored anywhere.

Note

When updating to a newer emRun version: as files may have been added, moved or deleted, the project directories may need to be updated accordingly.

1.4 Package content

emRun is provided in source code and contains everything needed. The following table shows the content of the emRun Package:

Directory	Description
`Doc`	emRun documentation.
`Src`	emRun source code.

1.4.1 Include directories

You should make sure that the system include path contains the following directory:

- `Src`

Note

Always make sure that you have only one version of each file!

It is frequently a major problem when updating to a new version of emRun if you have old files included and therefore mix different versions. If you keep emRun in the directories as suggested (and only in these), this type of problem cannot occur. When updating to a newer version, you should be able to keep your configuration files and leave them unchanged. For safety reasons, we recommend backing up (or at least renaming) the `LIB` directories before to updating.

Chapter 2

Compiling emRun

2.1 User-facing source files

The standard C library is exposed to the user by a set of header files that provide an interface to the library. In addition, there must be additional "invisible" functions added to provide C language support, such as software floating point and integer mathematics, that the C compiler calls.

The user-facing interface files are:

File	Description
`<assert.h>`	Assertion macros.
`<complex.h>`	Complex number functions.
`<ctype.h>`	Character classification functions.
`<errno.h>`	Access to errno.
`<fenv.h>`	Floating-point environment functions.
`<float.h>`	Parameterization of floating types.
`<inttypes.h>`	Parameterization of formatting of integer types.
`<iso646.h>`	Alternative spelling of C operators.
`<limits.h>`	Minima and maxima of floating and integer types.
`<locale.h>`	Functions for internationalizing software.
`<math.h>`	Mathematical functions.
`<setjmp.h>`	Non-local jumps.
`<signal.h>`	Signals and interrupts.
`<stdbool.h>`	Boolean type and values.
`<stddef.h>`	Standard definitions such as `NULL`.
`<stdint.h>`	Specification of fixed-size integer types.
`<stdio.h>`	Formatted input and output functions.
`<stdlib.h>`	Standardized common library functions.
`<string.h>`	String and memory functions.
`<time.h>`	Time and date functions.
`<wchar.h>`	Wide character functions.
`<wctype.h>`	Wide character classification functions.
`<xlocale.h>`	Extended POSIX.1 locale functions.

In addition some private header files are required:

File	Description
`__SEGGER_RTL.h`	General definitions used when compiling the library.
`__SEGGER_RTL_Conf.h`	Specific configuration of the library.
`__SEGGER_RTL_ConfDefaults.h`	Default configuration of the library.

2.2 Implementation source files

emRun is delivered in a small number of files that must be added to your project before building:

File	Description
`atomicops.c`	Support for atomic operations.
`codesets.c`	Support for code pages used in locales.
`config.c`	Support for configuration checks.
`compilersmops_arm.s`	Support for compiler-generated helpers and builtins (ARM).
`compilersmops_rv.s`	Support for compiler-generated helpers and builtins (RISC-V).
`convops.c`	Support for conversion between binary and printable strings.
`errno.c`	Support for `errno` in a tasking environment.
`errno_arm.c`	Support for `errno` in an AEABI environment (ARM).
`execops.c`	Support for execution-control functions e.g. `atexit()`.
`execops_arm.c`	Support for execution-control functions in an AEABI environment (ARM).
`fenvops.c`	Support for floating-point environment functions e.g. `feraiseexcept()`.
`fileops.c`	Support for file-based I/O operations e.g. `fputs`.
`floatasmops_arm.s`	Support for low-level floating point functions (ARM).
`floatasmops_rv.s`	Support for low-level floating point functions (RISC-V).
`floatops.c`	Support for high-level floating point functions.
`heapops.c`	Support for generic dynamic storage functions e.g. `malloc()`.
`heapops_minimal.c`	Support for allocate-only dynamic storage management.
`heapops_basic.c`	Support for low-overhead dynamic storage management.
`heapops_realtime.c`	Support for real-time O(1) dynamic storage management.
`intops.c`	Support for high-level integer functions e.g. `ldiv()`.
`intasmops_arm.s`	Support for low-level integer functions (ARM).
`intasmops_rv.s`	Support for low-level integer functions (RISC-V).
`jumpasmops_arm.s`	Support for nonlocal 'goto' functions e.g. `longjmp` (ARM).
`jumpasmops_rv.s`	Support for nonlocal 'goto' functions e.g. `longjmp` (RISC-V).
`locales.c`	Support for various locales.
`mbops.c`	Support for multi-byte functions e.g. `mbtowc()`.
`prinops.c`	Support for formatting functions e.g. `sprintf()`.
`scanops.c`	Support for formatted input functions e.g. `scanf()`.
`sortops.c`	Support for searching and sorting functions e.g. `qsort()`.
`strasmops_arm.s`	Support for fast string and memory functions e.g. `strcpy()` (ARM).
`strasmops_rv.s`	Support for fast string and memory functions e.g. `strcpy()` (RISC-V).
`strops.c`	Support for string and memory functions e.g. `strcat()`.
`timeops.c`	Support for time operations e.g. `mktime()`.
`timeops_x.c`	Support for low-level time operations e.g. `__SEGGER_RTL_gettimeofday()}`.
`utilops.c`	Support for common functions used in emRun.
`wconvops.c`	Support for conversion between binary and wide strings.

File	Description
`wprinops.c`	Support for wide formatted output functions e.g. `wprintf()`.
`wscanops.c`	Support for wide formatted input functions e.g. `wscanf()`.
`wstrops.c`	Support for wide string functions e.g. `wcscpy()`.

Additionally, example I/O implementations are provided, only one of which must be compiled into your application or library when using emRun:

File	Description
`fileops_semi.c`	Support for complete I/O interface using SEGGER semihosting.
`prinops_rtt.c`	Support for character I/O using SEGGER RTT.
`prinops_semi.c`	Support for character I/O using SEGGER semihosting.
`prinops_uart.c`	Support for character I/O using a UART.

A customized version of the SEGGER real-time heap is provided:

File	Description
`__SEGGER_RTL_RTHEAP.h`	Real-time heap interface.
`__SEGGER_RTL_RTHEAP_Conf.h`	Real-time heap configuration.
`__SEGGER_RTL_RTHEAP_ConfDefaults.h`	Real-time heap configuration defaults.
`__SEGGER_RTL_RTHEAP.c`	Real-time heap implementation.

2.3 General configuration

All source files should be added to the project and the following preprocessor symbols set correctly to select the particular variant of the library:

The configuration of emRun is defined by the content of `__SEGGER_RTL_Conf.h` which is included by all C and assembly language source files. The example configuration files that ship with emRun are described in the following sections.

The following preprocessor symbol definitions affect how the library is compiled and the features that are implemented:

Symbol	Description
`__SEGGER_RTL_OPTIMIZE`	Prefer size-optimized or speed-optimized code.
`__SEGGER_RTL_FORMAT_INT_WIDTH`	Support for `int`, `long`, and `long long` in `printf()` and `scanf()` functions.
`__SEGGER_RTL_FORMAT_FLOAT_WIDTH`	Support `float` in `printf()` and `scanf()` functions.
`__SEGGER_RTL_FORMAT_WIDTH_PRECISION`	Support width and precision in `printf()` and `scanf()` functions.
`__SEGGER_RTL_FORMAT_CHAR_CLASS`	Support character classes in `scanf()` functions.
`__SEGGER_RTL_FORMAT_WCHAR`	Support wide character output in `printf()` and `scanf()` functions.
`__SEGGER_RTL_STDOUT_BUFFER_LEN`	Configuration of buffer capacity for standard output stream.
`__SEGGER_RTL_ATEXIT_COUNT`	The maximum number of registered `atexit()` functions.
`__SEGGER_RTL_SCALED_INTEGER`	Selection of scaled-integer floating-point algorithms.
`__SEGGER_RTL_NO_BUILTIN`	Prevent optimizations that cause incorrect code generation when compiling at high optimization levels.

2.3.1 Source-level optimization

Default

```
#ifndef   __SEGGER_RTL_OPTIMIZE
  #define __SEGGER_RTL_OPTIMIZE                  0
#endif
```

Description

Define the preprocessor symbol `__SEGGER_RTL_OPTIMIZE` to select size-optimized implementations for both C and assembly language code.

If this preprocessor symbol is undefined (the default) the library is configured to select balanced implementations.

Value	Description
-2	Favor size at the expense of speed.
-1	Favor size over speed.
0	Balanced.
+1	Favor speed over size.
+2	Favor speed at the expense of size.

2.3.2 Integer I/O capability selection

Default

```
#define __WIDTH_INT                                0
#define __WIDTH_LONG                               1
#define __WIDTH_LONG_LONG                          2

#ifndef   __SEGGER_RTL_FORMAT_INT_WIDTH
  #define __SEGGER_RTL_FORMAT_INT_WIDTH            __WIDTH_LONG_LONG
#endif
```

Description

To select the level of `printf()` and `scanf()` support, set this preprocessor symbol as follows:

Value	Description
0	Support only int, do not support long or long long.
1	Support int and long, do not support long long.
2	Support int, long, and long long.

2.3.3 Floating I/O capability selection

Default

```
#define __WIDTH_NONE                          0
#define __WIDTH_FLOAT                         1
#define __WIDTH_DOUBLE                        2

#ifndef   __SEGGER_RTL_FORMAT_FLOAT_WIDTH
  #define __SEGGER_RTL_FORMAT_FLOAT_WIDTH       __WIDTH_DOUBLE
#endif
```

Description

Set this preprocessor symbol to include floating-point support in `printf()` and `scanf()` as follows:

Value	Description
0	Eliminate all formatted floating point support.
1	Support output of float values, no doubles.
2	Support output of float, double, and long double values.

2.3.4 Wide character I/O support

Default

```
#ifndef   __SEGGER_RTL_FORMAT_WCHAR
  #define __SEGGER_RTL_FORMAT_WCHAR            1
#endif
```

Description

Set this preprocessor symbol to include wide character support in `printf()` and `scanf()` as follows:

Value	Description
0	Eliminate all wide character support.
1	Support formatted input and output of wide characters.

2.3.5 Input character class support selection

Default

```
#ifndef   __SEGGER_RTL_FORMAT_CHAR_CLASS
  #define __SEGGER_RTL_FORMAT_CHAR_CLASS        1
#endif
```

Description

Set this preprocessor symbol to include character class support in `scanf()` as follows:

Value	Description
0	Eliminate all character class support.
1	Support formatted input with character classes.

2.3.6 Width and precision specification selection

Default

```
#ifndef   __SEGGER_RTL_FORMAT_WIDTH_PRECISION
  #define __SEGGER_RTL_FORMAT_WIDTH_PRECISION   1
#endif
```

Description

Set this preprocessor symbol to include width and precision support in `printf()` and `scanf()` as follows:

Value	Description
0	Eliminate all width and precision support.
1	Support formatted input and output with width and precision.

2.3.7 Standard output stream buffering

Default

```
#ifndef   __SEGGER_RTL_STDOUT_BUFFER_LEN
  #define __SEGGER_RTL_STDOUT_BUFFER_LEN        64
#endif
```

Description

Set this preprocessor symbol to set the internal size of the formatting buffer, in characters, used when printing to the standard output stream. By default it is 64.

2.3.8 Registration of exit cleanup functions

Default

```
#ifndef   __SEGGER_RTL_ATEXIT_COUNT
  #define __SEGGER_RTL_ATEXIT_COUNT             1
#endif
```

Description

Set this preprocessor symbol to the maximum number of registered `atexit()` functions to support. The registered functions can be executed when `main())` returns by calling `__SEGGER_RTL_execute_at_exit_fns()`, typically as part of the startup code.

2.3.9 Scaled-integer algorithm selection

Default

```
#ifndef   __SEGGER_RTL_SCALED_INTEGER
  #define __SEGGER_RTL_SCALED_INTEGER          0
#endif
```

Description

Define the preprocessor symbol `__SEGGER_RTL_SCALED_INTEGER` to select scaled-intger algorithms over standard floating-point algorithms.

Value	Description
0	Algorithms use C-language floating-point arithmetic.
1	IEEE single-precision functions use scaled integer arithmetic if there is a scaled-integer implementation of the function.
+2	IEEE single-precision and double-precision functions use scaled integer arithmetic if there is a scaled-integer implementation of the function.

Note that selecting scaled-integer arithmetic does not reduce the range or accuracy of the function as seen by the user. Scaled-integer arithmetic runs quickly on integer-only processors and delivers results that are correctly rounded in more cases as 31 bits or 63 bits of precision are retained internally whereas using IEEE aritmetic retains only 24 or 53 bits of precision.

Scaled-integer algorithms are faster than standard algorithms using the floating-point emulator, but can be significantly larger depending upon compiler optimization settings.

2.3.10 Optimization prevention

Default

None; this must be specifically configured for compiler and architecture. The defaults for Arm and RISC-V are:

```
#if defined(__clang__)
  #define __SEGGER_RTL_NO_BUILTIN
#elif defined(__GNUC__)
  #define __SEGGER_RTL_NO_BUILTIN \
    __attribute__((optimize("-fno-tree-loop-distribute-patterns")))
#endif
```

Description

Define the preprocessor symbol `__SEGGER_RTL_NO_BUILTIN` to prevent GCC from applying incorrect optimizations at high optimization levels.

Specifically, at high optimization GCC will:

- Replace a repeated-fill loop with a call to `memset()`.
- Replace a repeated-copy loop with a call to `memcpy()`.

This definition prevents GCC from identifying a loop copy in the implementation of `memcpy()` and replacing it with a call to `memcpy()`, thereby introducing infinite recursion.

GCC has been observed to make the following transformations:

- Replace `malloc()` immediately followed by `memset()` to zero with a call to `calloc()`.
- Replace `sin()` and `cos()` of the same value with a call to `sincos()`.

Clang has been observed to make the following transformations:

- Replace `exp(10, x)` with a call to `exp10(x)`.

Unfortunately it is not possible to prevent these optimizations using a per-function optimization attribute. These optimizations *may* be disabled by using the GCC command-line option `-fno-builtins` or `-ffreestanding`, but you are advised to check the subject compiler for adherence.

To prevent the transformation of `malloc()` followed by `memset()`, emRun works around this by a volatile store to the allocated memory (if successfully allocated with nonzero size).

To prevent user programs from suffering optimization of `sin()` and `cos()` to `sincos()`, an implementations of POSIX.1 `sincos()`, `sincosf()`, and `sincosl()` are provided. The implementation of the `sincos()` family does not suffer this misoptimization as emRun does not directly call the `sin()` and `cos()` functions.

To prevent user programs from suffering optimization of `exp(10, x)`, implementations of `exp10()`, `exp10f()`, and `exp10l()` are provided. The implementation of the `exp10()` family does not suffer this misoptimization as emRun does not directly call the `exp()` functions.

2.4 Configuring for Arm

This section provides a walkthrough of the library configuration supplied in `__SEGGER_RTL_Arm_Conf.h` for Arm processors.

The library is configured for execution on Arm targets by querying the environment. The example configuration assumes that the compiler supports the *Arm C Language Extensions* (ACLE) standard.

In many cases the library can can be configured automatically. For ARM the default configuration of the library is derived from these preprocessor symbols:

Symbol	Description
Compiler identification	
`__GNUC__`	Compiler is GNU C.
`__clang__`	Compiler is Clang.
Target instruction set	
`__thumb__`	Target the Thumb instruction set (as opposed to ARM).
`__thumb2__`	Target the Thumb-2 instruction set.
ACLE definitions	
`__ARM_ARCH`	Arm target architecture version.
`__ARM_ARCH_PROFILE`	Arm architecture profile, if applicable.
`__ARM_ARCH_ISA_ARM`	Processor implements AArch32 instruction set.
`__ARM_ARCH_ISA_THUMB`	Processor implements Thumb instruction set.
`__ARM_BIG_ENDIAN`	Byte order is big endian.
`__ARM_PCS`	Functions use standard Arm PCS calling convention.
`__ARM_PCS_VFP`	Functions use Arm VFP calling convention.
`__ARM_FP`	Arm floating-point hardware availability.
`__ARM_FEATURE_CLZ`	Indicates existence of CLZ instruction.
`__ARM_FEATURE_IDIV`	Indicates existence of integer division instructions.

2.4.1 Target instruction set

Default

```
#define __SEGGER_RTL_ISA_T16                       0
#define __SEGGER_RTL_ISA_T32                       1
#define __SEGGER_RTL_ISA_ARM                       2

#if defined(__thumb__) && !defined(__thumb2__)
  #define __SEGGER_RTL_TARGET_ISA                  __SEGGER_RTL_ISA_T16
#elif defined(__thumb2__)
  #define __SEGGER_RTL_TARGET_ISA                  __SEGGER_RTL_ISA_T32
#else
  #define __SEGGER_RTL_TARGET_ISA                  __SEGGER_RTL_ISA_ARM
#endif
```

Description

These definitions are used by assembly language files to check the instruction set being compiled for. The preprocessor symbol `__thumb__` is defined when compiling for cores that support 16-bit Thumb instructions but not Thumb-2 instructions; the preprocessor symbol `__thumb2__` is defined when compiling for cores that support the 32-bit Thumb-2 instructions. If neither of these symbols is defined, the core supports the AArch32 Arm instruction set.

2.4.2 Arm AEABI

Default

```
#if defined(__GNUC__) || defined(__clang__)
  #define __SEGGER_RTL_INCLUDE_AEABI_API        2
#endif
```

Description

Implementation of the ARM AEABI functions are required by all AEABI-conforming C compilers. This definition can be set to 1, in which case C-coded generic implementations of AEABI functions are compiled into the library; or it can be set to 2, in which case assembly-coded implementations are compiled into the library and is the preferred option.

2.4.3 Processor byte order

Default

```
#if defined(__ARM_BIG_ENDIAN) && (__ARM_BIG_ENDIAN == 1)
  #define __SEGGER_RTL_BYTE_ORDER               (+1)
#else
  #define __SEGGER_RTL_BYTE_ORDER               (-1)
#endif
```

Description

The ACLE symbol `__ARM_BIG_ENDIAN` is queried to determine whether the target core runs in litte-endian or big-endian mode and configures the library for that byte ordering.

2.4.4 Maximal data type alignment

Default

```
#define __SEGGER_RTL_MAX_ALIGN                      8
```

Description

This sets the maximal type alignment required for any type. For 64-bit double data loaded by LDRD or VLDR, it is best to align data on 64-bit boundaries.

2.4.5 ABI type set

Default

```
#define __SEGGER_RTL_TYPESET                        32
```

Description

All Arm targets use a 32-bit ILP32 ABI, and this is not configurable otherwise for the library.

2.4.6 Static branch probability

Default

```
#if defined(__GNUC__) || defined(__clang__)
  #define __SEGGER_RTL_UNLIKELY(X)              __builtin_expect((X), 0)
#endif
```

Description

The preprocessor macro `__SEGGER_RTL_UNLIKELY` is configured to indicate that the expression `X` is unlikely to occur. This enables the compiler to use this information to configure the condition of branch instructions to place exceptional code off the hot trace and not incur branch penalties for the likely execution path.

This definition is specific to the GNU and Clang compilers; configure this to whatever your compiler supports or, if not supported at all, leave `__SEGGER_RTL_UNLIKELY` undefined.

2.4.7 Thread-local storage

Default

```
#if defined(__GNUC__) || defined(__clang__)
  #define __SEGGER_RTL_THREAD                      __thread
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_THREAD` can be defined to the storage class specifier for thread-local data, if your compiler supports thread-local storage. For Arm processors, thread-local storage is accessed using the `__aeabi_read_tp` function which is dependent upon the target operating system and whether an operating system is present.

The library has a number of file-scope and external variables that benefit from thread-local storage, such as the implementation of `errno`.

If your compiler does not support thread-local storage class specifiers or your target does not run an operating system, leave `__SEGGER_RTL_THREAD` undefined.

2.4.8 Function inlining control

Default

```
#if (defined(__GNUC__) || defined(__clang__))
  #ifndef    __SEGGER_RTL_NEVER_INLINE
    #if defined(__clang__)
      #define __SEGGER_RTL_NEVER_INLINE   __attribute__((__noinline__))
    #else
      #define __SEGGER_RTL_NEVER_INLINE
  __attribute__((__noinline__, __noclone__))
    #endif
  #endif
  //
  #ifndef    __SEGGER_RTL_ALWAYS_INLINE
    #define __SEGGER_RTL_ALWAYS_INLINE
  __inline__ __attribute__((__always_inline__))
  #endif
  //
  #ifndef    __SEGGER_RTL_REQUEST_INLINE
    #define __SEGGER_RTL_REQUEST_INLINE   __inline__
  #endif
  //
#endif
```

Description

The preprocessor symbols `__SEGGER_RTL_NEVER_INLINE`, `__SEGGER_RTL_ALWAYS_INLINE`, and `__SEGGER_RTL_REQUEST_INLINE` are configured indicate to the compiler the benefit of inlining.

`__SEGGER_RTL_NEVER_INLINE` should be configured to disable inlining of a function in all cases.

`__SEGGER_RTL_ALWAYS_INLINE` should be configured to encourage inlining of a function in all cases.

`__SEGGER_RTL_REQUEST_INLINE` should be configured to indicate that a function benefits from inlining but it is not essential to inline this function. Typically this is used to inline a function when compiling to maximize execution speed and not inline a function when compiling to minimize code size.

The above definitions work for the GNU and clang compilers when targeting Arm. If your compiler is different, configure thsse symbols to suit.

2.4.9 Public API indication

Default

```
#if defined(__GNUC__) || defined(__clang__)
  #define __SEGGER_RTL_PUBLIC_API               __attribute__((__weak__))
#endif
```

Description

Every function in the library that forms part of the API is labeled using __SEGGER_RTL_PUBLIC_API. For GCC and Clang compilers, all API entry points are defined as weak ELF symbols. You can customize this for your particular compiler or, if compiling the library as part of your project, you can leave this undefined in order to have strong definitions of each library symbol.

2.4.10 Floating-point ABI

Default

```
#if defined(__ARM_PCS_VFP) && (__ARM_PCS_VFP == 1)
  //
  // PCS uses hardware registers for passing parameters.  For VFP
  // with only single-precision operations, parameters are still
  // passed in floating registers.
  //
  #define __SEGGER_RTL_FP_ABI                   2
  //
#elif defined(__ARM_PCS) && (__ARM_PCS == 1)
  //
  // PCS is standard integer PCS.
  //
  #define __SEGGER_RTL_FP_ABI                   0
  //
#else
  #error Unable to determine floating-point ABI used
#endif
```

Description

Configuration of the floating-point ABI in use is determined from the ACLE symbols `__ARM_PCS_VFP` and `__ARM_PCS`.

`__SEGGER_RTL_FP_ABI` must be set to 0 if `float` and `double` parameters are passed using integer registes, to 1 if `float` parameters are passed using floating registers and `double` parameters are passed using integer registers, and to 2 if both `float` and `double` parameters are passed using floating registers.

The ACLE symbol `__ARM_PCS_VFP` being set to 1 indicates that floating-point arguments are passed using floating-point registers; the ACLE symbol `__ARM_PCS` being set to 1 indicates that floating-point arguments are passed in integer registers. From these definitions, `__SEGGER_RTL_FP_ABI` is set appropriately.

Note that for cores that have only single-precision (32-bit) floating-point, double precision (64-bit) arguments are passed in two single-precision floating-point registers and *not* in integer registers.

2.4.11 Floating-point hardware

Default

```
#if defined(__ARM_FP) && (__ARM_FP & 0x08)
  #define __SEGGER_RTL_FP_HW                    2
#elif defined(__ARM_FP) && (__ARM_FP & 0x04)
  #define __SEGGER_RTL_FP_HW                    1
#else
  #define __SEGGER_RTL_FP_HW                    0
#endif

// Clang gets __ARM_FP wrong for the T16 target ISA indicating
// that floating-point instructions exist in this ISA -- which
// they don't.  Patch that definition up here.
#if __ARM_ARCH_ISA_THUMB == 1
  #undef   __SEGGER_RTL_FP_HW
  #define __SEGGER_RTL_FP_HW                    0
  #undef   __SEGGER_RTL_FP_ABI
  #define __SEGGER_RTL_FP_ABI                   0
#endif
```

Description

Floating-point hardware support is configured separately from the floating-point calling convention. Even if floating-point parameters are passed in integer registers, it is still possible that floating-point instructions operate on those parameters in the called function.

The ACLE symbol __ARM_FP is queried to determine the target core's floating-point ability and set __SEGGER_RTL_FP_HW appropriately.

__SEGGER_RTL_FP_HW is set to 0 to indicate that no floating-point hardware exists, to 1 to indicate that hardware exists to support `float` arithmetic, and to 2 to to indicate that hardware exists to support `double` arithmetic.

Unfortunately, a fix-up is required for Clang when tageting the 16-bit Thumb instruction set.

2.4.12 Half-precision floating-point type

Default

```
#define __SEGGER_RTL_FLOAT16                           _Float16
```

Description

The GNU and clang compilers support 16-bit floating-point data in IEEE format. This configures the emRun type that implements 16-bit floating-point. Some compilers use `__fp16` as type name, but `_Float16` is the standard C name for such a type.

2.4.13 Multiply-subtract instruction availability

Default

```
#if (__ARM_ARCH >= 6) && (__SEGGER_RTL_TARGET_ISA != __SEGGER_RTL_ISA_T16)
  #define __SEGGER_RTL_CORE_HAS_MLS              1
#else
  #define __SEGGER_RTL_CORE_HAS_MLS              0
#endif
```

Description

Assembly-language source files use the preprocessor symbol `__SEGGER_RTL_CORE_HAS_MLS` to conditionally assemble `MLS` instructions. The ACLE symbol `__ARM_ARCH` is queried to determine whether the target architecture offers a `MLS` instruction and then `__SEGGER_RTL_TARGET_ISA` is checked to ensure that it is offered in the selected instruction set.

2.4.14 Long multiply instruction availability

Default

```
#if __SEGGER_RTL_TARGET_ISA == __SEGGER_RTL_ISA_T16
  //
  // T16 ISA has no extended multiplication at all.
  //
  #define __SEGGER_RTL_CORE_HAS_EXT_MUL         0
  //
#elif __ARM_ARCH >= 6
  //
  // ARMv6 and above have no restrictions on their input
  // and output registers, so assembly-level inserts with
  // constraints to guide the compiler are acceptable.
  //
  #define __SEGGER_RTL_CORE_HAS_EXT_MUL         1
  //
#elif (__ARM_ARCH == 5) && defined(__clang__)
  //
  // Take Arm at its word and disable restrictions on input
  // and output registers.
  //
  #define __SEGGER_RTL_CORE_HAS_EXT_MUL         1
  //
#else
  //
  // ARMv5TE and lower have restrictions on their input
  // and output registers, therefore do not enable extended
  // multiply inserts.
  //
  #define __SEGGER_RTL_CORE_HAS_EXT_MUL         0
  //
#endif
```

Description

Assembly-language source files use the preprocessor symbol `__SEGGER_RTL_CORE_HAS_EXT_MUL` to conditionally compile and assemble long-multiply instructions. This symbol must be set to 1 to indicate that long multiply instructions are supported in the target instruction set, and to zero otherwise.

In the ARM Architecture Reference Manual, DDI 01001, Arm states the following for the SMULL and UMULL instructions:

> **Note**
>
> "Specifying the same register for either RdHi and Rm, or RdLo and Rm, was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results."

Unfortunately, the GNU assembler enforces this restriction which means that assembly-level long-multiply inserts will not work for ARMv4 and ARMv5 even though there is no indication that they fail in practice. For the `clang` compiler, no such restriction is enforced.

The default configuration is deliberately conservative; you may configure this differently for your specific compiler, assembler, and target processor.

2.4.15 Count-leading-zeros instruction availability

Default

```
#if defined(__ARM_FEATURE_CLZ) && (__ARM_FEATURE_CLZ == 1)
  #define __SEGGER_RTL_CORE_HAS_CLZ             1
#else
  #define __SEGGER_RTL_CORE_HAS_CLZ             0
#endif

#if __SEGGER_RTL_CORE_HAS_CLZ
  //
  // For ACLE-conforming C compilers that declare the architecture or
  // profile has a CLZ instruction, use that CLZ instruction.
  //
  #define __SEGGER_RTL_CLZ_U32(X)               __builtin_clz(X)
#endif

// Clang gets __ARM_FEATURE_CLZ wrong for v8M.Baseline, indicating
// that CLZ is available in this ISA  -- which it isn't.  Patch that
// definition up here.
#if (__ARM_ARCH == 8) && (__SEGGER_RTL_TARGET_ISA == __SEGGER_RTL_ISA_T16)
  #undef   __SEGGER_RTL_CORE_HAS_CLZ
  #define __SEGGER_RTL_CORE_HAS_CLZ             0
#endif

// GCC gets __ARM_FEATURE_CLZ wrong for v5TE compiling for Thumb,
// indicating that CLZ is available in this ISA -- which it isn't.
// Patch that definition up here.
#if (__ARM_ARCH == 5) && (__SEGGER_RTL_TARGET_ISA == __SEGGER_RTL_ISA_T16)
  #undef   __SEGGER_RTL_CORE_HAS_CLZ
  #define __SEGGER_RTL_CORE_HAS_CLZ             0
#endif
```

Description

The library benefits from the availability of a count-leading-zero instruction. The ACLE symbol __ARM_FEATURE_CLZ is set to 1 to indicate that the target architecture provides a CLZ instruction. This definition works for ACLE-conforming compilers.

The preprocessor symbol __SEGGER_RTL_CLZ_U32 is defined to expand to a way to use the CLZ instruction when the core is known to have one.

Unfortunately, although GNU and Clang compilers conform to the ACLE, they disagree on the availability of the CLZ instruction and provide an incorrect definition of __ARM_FEATURE_CLZ for some architectures. Therefore the fixups above are applied for these known cases.

2.4.16 SIMD media instruction availability

Default

```
#if defined(__ARM_ARCH) && (__ARM_ARCH >= 6) && (__SEGGER_RTL_TARGET_ISA !
= __SEGGER_RTL_ISA_T32)
  #define __SEGGER_RTL_CORE_HAS_MEDIA           1
#else
  #define __SEGGER_RTL_CORE_HAS_MEDIA           0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_MEDIA` must be set to 1 if the target instruction set has the DSP media instructions, and 0 otherwise.

The library uses the media instructions to accelerate string processing functions such as `strlen()` and `strcmp()`.

2.4.17 Bit-reverse instruction availability

Default

```
#if defined(__ARM_ARCH) && (__ARM_ARCH >= 7)
  #define __SEGGER_RTL_CORE_HAS_REV             1
#else
  #define __SEGGER_RTL_CORE_HAS_REV             0
#endif
```

Description

The preprocessor symbol __SEGGER_RTL_CORE_HAS_REV must be set to 1 if the target instruction set offers the REV instruction, and 0 otherwise.

2.4.18 And/subtract-word instruction availability

Default

```
#if (__ARM_ARCH >= 7) && (__SEGGER_RTL_TARGET_ISA == __SEGGER_RTL_ISA_T32)
  #define __SEGGER_RTL_CORE_HAS_ADDW_SUBW       1
  // ARMv8A/R only has ADDW in Thumb mode
#else
  #define __SEGGER_RTL_CORE_HAS_ADDW_SUBW       0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_ADDW_SUBW` must be set to 1 if the target instruction set offers the ADDW and SUBW instructions, and 0 otherwise.

2.4.19 Move-word instruction availability

Default

```
#if __ARM_ARCH >= 7
  #define __SEGGER_RTL_CORE_HAS_MOVW_MOVT       1
#else
  #define __SEGGER_RTL_CORE_HAS_MOVW_MOVT       0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_MOVW_MOVT` must be set to 1 if the target instruction set offers the MOVW and MOVT instructions, and 0 otherwise.

2.4.20 Integer-divide instruction availability

Default

```
#if defined(__ARM_FEATURE_IDIV) && __ARM_FEATURE_IDIV
  #define __SEGGER_RTL_CORE_HAS_IDIV            1
#else
  #define __SEGGER_RTL_CORE_HAS_IDIV            0
#endif

// Unfortunately the ACLE specifies "__ARM_FEATURE_IDIV is defined to
 1 if the target
// has hardware support for
 32-bit integer division in all available instruction sets."
// For v7R, there is typically no divide in the Arm instruction set but there is
// support for divide in the Thumb instruction set, so provide an exception here
// when targeting v7R in Thumb mode.
#if (__ARM_ARCH_PROFILE == 'R') && (__SEGGER_RTL_TARGET_ISA == __SEGGER_RTL_ISA_T32)
  #undef  __SEGGER_RTL_CORE_HAS_IDIV
  #define __SEGGER_RTL_CORE_HAS_IDIV            1
#endif
```

Description

The preprocessor symbol __SEGGER_RTL_CORE_HAS_IDIV must be set to 1 if the target instruction set offers integer divide instructions, and 0 otherwise. Note the ACLE inquiry above, if not adjusted for the specific v7R instruction set, leads to suboptimal code.

2.4.21 Zero-branch instruction availability

Default

```
#if (__ARM_ARCH >= 7) && (__SEGGER_RTL_TARGET_ISA != __SEGGER_RTL_ISA_ARM)
  #define __SEGGER_RTL_CORE_HAS_CBZ_CBNZ        1
#else
  #define __SEGGER_RTL_CORE_HAS_CBZ_CBNZ        0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_CBZ_CBNZ` must be set to 1 if the target architecture offers CBZ and CBNZ instructions, and to 0 otherwise.

2.4.22 Table-branch instruction availability

Default

```
#if (__ARM_ARCH >= 7) && (__SEGGER_RTL_TARGET_ISA == __SEGGER_RTL_ISA_T32)
  #define __SEGGER_RTL_CORE_HAS_TBB_TBH         1
#else
  #define __SEGGER_RTL_CORE_HAS_TBB_TBH         0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_TBB_TBH` must be set to 1 if the target architecture offers TBB and TBH instructions, and to 0 otherwise.

2.4.23 Sign/zero-extension instruction availability

Default

```
#if __ARM_ARCH >= 6
  #define __SEGGER_RTL_CORE_HAS_UXT_SXT         1
#else
  #define __SEGGER_RTL_CORE_HAS_UXT_SXT         0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_UXT_SXT` must be set to 1 if the target architecture offers UXT and SXT instructions, and to 0 otherwise.

2.4.24 Bitfield instruction availability

Default

```
#if (__SEGGER_RTL_TARGET_ISA == __SEGGER_RTL_ISA_T32) || (__ARM_ARCH >= 7)
  #define __SEGGER_RTL_CORE_HAS_BFC_BFI_BFX     1
#else
  #define __SEGGER_RTL_CORE_HAS_BFC_BFI_BFX     0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_BFC_BFI_BFX` must be set to 1 if the target architecture offers BFC, BFI, and BFX instructions, and to 0 otherwise.

2.4.25 BLX-to-register instruction availability

Default

```
#if __ARM_ARCH >= 5
  #define __SEGGER_RTL_CORE_HAS_BLX_REG         1
#else
  #define __SEGGER_RTL_CORE_HAS_BLX_REG         0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_BLX_REG` must be set to 1 if the target architecture offers BLX using a register, and to 0 otherwise.

2.4.26 Long shift-count availability

Default

```
#if (__ARM_ARCH >= 6) && (__SEGGER_RTL_TARGET_ISA == __SEGGER_RTL_ISA_T32)
  #define __SEGGER_RTL_CORE_HAS_LONG_SHIFT      1
#else
  #define __SEGGER_RTL_CORE_HAS_LONG_SHIFT      0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_LONG_SHIFT` must be set to `1` if the target architecture offers correct shifting of registers when the bitcount is greater than 32.

2.5 Configuring for RISC-V

This section provides a walkthrough of the library configuration supplied in `__SEGGER_RTL_RISCV_Conf.h` for RV32 processors.

The library is configured for execution on RISC-V targets by querying the environment. The example configuration assumes that the compiler supports the preprocessor symbols definied for the RISC-V architecture as follows:

Symbol	Description
`__riscv`	Target is RISC-V.
`__riscv_abi_rve`	Target RV32E base instruction set.
`__riscv_compressed`	Target has C extension.
`__riscv_float_abi_soft`	Target has neither F nor D extension.
`__riscv_float_abi_single`	Target has F extension.
`__riscv_float_abi_double`	Target has D and F extensions.
`__riscv_mul`	Target has M extension.
`__riscv_muldiv`	Target has M extension with divide support.
`__riscv_div`	Target has M extension with divide support.
`__riscv_dsp`	Target has P (packed SIMD) extension.
`__riscv_zba`	Target has Zba (shift-add) extension.
`__riscv_zbb`	Target has Zbb (CLZ, negated logic) extension.
`__riscv_zbs`	Target has Zbs (bt manipulation) extension.
`__riscv_xlen`	Register width.
`__riscv_flen`	Floating-point register width.
`__nds_v5`	Andes Performance Extension support.

2.5.1 Base instruction set architecture

Default

```
#if defined(__riscv_abi_rve)
  #define __SEGGER_RTL_CORE_HAS_ISA_RVE                1
#else
  #define __SEGGER_RTL_CORE_HAS_ISA_RVE                0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_ISA_RVE` must be set to 1 if the base instruction set is RV32E and to 0 if the base instruction set is RV32I.

2.5.2 GNU libgcc API

Default

```
#if defined(__GNUC__) || defined(__clang__)
  #if __riscv_xlen == 32
    #define __SEGGER_RTL_INCLUDE_GNU_API  2
  #else
    #define __SEGGER_RTL_INCLUDE_GNU_API  1
  #endif
#endif
```

Description

The GNU and clang compilers both use the standard GNU libgcc API for runtime services. The following settings to select the GNU libgcc API are supported:

Setting	Description
0	GNU libgcc API is eliminated.
1	GNU libgcc API uses all C-coded functions.
2	GNU libgcc API uses a combination of C-coded functions and assembly language acceleration functions.

Note: Assembly-language acceleration is only supported for RV32E and RV32I architectures.

2.5.3 GNU libgcc 16-bit float API

Default

```
#define __SEGGER_RTL_INCLUDE_GNU_FP16_API       1
```

Description

The GNU and clang compilers support 16-bit floating-point data in IEEE format. This configures emRun support for GCC on RISC-V.

The following settings to select the GNU libgcc API are supported:

Setting	Description
0	GNU libgcc 16-bit float API is eliminated.
1	GNU libgcc 16-bit float API is present.

Note that `__SEGGER_RTL_FLOAT16` must also be configured if runtime support for 16-bit floating-point types is configured.

2.5.4 Half-precision floating-point type

Default

```
#define __SEGGER_RTL_FLOAT16                          _Float16
```

Description

The GNU and clang compilers support 16-bit floating-point data in IEEE format. This configures the emRun type that implements 16-bit floating-point. Some compilers use `__fp16` as type name, but `_Float16` is the standard C name for such a type.

2.5.5 ABI type set

Default

```
#define __SEGGER_RTL_TYPESET                              32
```

Description

All RV32 targets use a 32-bit ILP32 ABI, and this is not configurable otherwise for the library.

2.5.6 Processor byte order

Default

```
#define __SEGGER_RTL_BYTE_ORDER                  (-1)
```

Description

Only little-endian RISC-V processors are supported at this time, and this preprocessor symbol cannot be configured any other way.

2.5.7 Minimum stack alignment

Default

```
#ifndef   __SEGGER_RTL_STACK_ALIGN
  #define __SEGGER_RTL_STACK_ALIGN              16
#endif
```

Description

The compiler provides correct stack alignment for the RISC-V ABI selected for compilation. However, assembly language files must also know the intended stack alignment of the system and ensure that alignment constraints are respected.

At the time of writing, there is an ongoing discussion in the RISC-V community as to the minimum stack alignment for RV32I and RV32E ABIs. As such, this definition is conservative and works for both RV32I and RV32E.

2.5.8 Static branch probability

Default

```
#if defined(__GNUC__) || defined(__clang__)
  #define __SEGGER_RTL_UNLIKELY(X)            __builtin_expect((X), 0)
#endif
```

Description

The preprocessor macro `__SEGGER_RTL_UNLIKELY` is configured to indicate that the expression `X` is unlikely to occur. This enables the compiler to use this information to configure the condition of branch instructions to place exceptional code off the hot trace and not incur branch penalties for the likely execution path.

This definition is specific to the GNU and Clang compilers; configure this to whatever your compiler supports or, if not supported at all, leave `__SEGGER_RTL_UNLIKELY` undefined.

2.5.9 Thread-local storage

Default

```
#if defined(__GNUC__) || defined(__clang__)
  #define __SEGGER_RTL_THREAD                     __thread
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_THREAD` can be defined to the storage class specifier for thread-local data, if your compiler supports thread-local storage. There is no standard embedded ABI for RISC-V processors, but for now thread-local storage is accessed using the `tp` register and is upon the target operating system and whether an operating system is present.

The library has a number of file-scope and external variables that benefit from thread-local storage, such as the implementation of `errno`.

If your compiler does not support thread-local storage class specifiers or your target does not run an operating system, leave `__SEGGER_RTL_THREAD` undefined.

2.5.10 Function inlining control

Default

```
#if (defined(__GNUC__) || defined(__clang__)) && (__SEGGER_RTL_CONFIG_CODE_COVERAGE == 0)
  #ifndef   __SEGGER_RTL_NEVER_INLINE
    #if defined(__clang__)
      #define __SEGGER_RTL_NEVER_INLINE    __attribute__((__noinline__))
    #else
      #define __SEGGER_RTL_NEVER_INLINE
  __attribute__((__noinline__, __noclone__))
    #endif
  #endif
  //
  #ifndef   __SEGGER_RTL_ALWAYS_INLINE
    #define __SEGGER_RTL_ALWAYS_INLINE
  __inline__ __attribute__((__always_inline__))
  #endif
  //
  #ifndef   __SEGGER_RTL_REQUEST_INLINE
    #define __SEGGER_RTL_REQUEST_INLINE    __inline__
  #endif
  //
#endif
```

Description

The preprocessor symbols __SEGGER_RTL_NEVER_INLINE, __SEGGER_RTL_ALWAYS_INLINE, and __SEGGER_RTL_REQUEST_INLINE are configured indicate to the compiler the benefit of inlining.

__SEGGER_RTL_NEVER_INLINE should be configured to disable inlining of a function in all cases.

__SEGGER_RTL_ALWAYS_INLINE should be configured to encourage inlining of a function in all cases.

__SEGGER_RTL_REQUEST_INLINE should be configured to indicate that a function benefits from inlining but it is not essential to inline this function. Typically this is used to inline a function when compiling to maximize execution speed and not inline a function when compiling to minimize code size.

The above definitions work for the GNU and clang compilers when targeting Arm. If your compiler is different, configure thsse symbols to suit.

2.5.11 Public API indication

Default

```
#if defined(__GNUC__) || defined(__clang__)
  #define __SEGGER_RTL_PUBLIC_API               __attribute__((__weak__))
#endif
```

Description

Every function in the library that forms part of the API is labeled using `__SEGGER_RTL_PUBLIC_API`. For GCC and Clang compilers, all API entry points are defined as weak ELF symbols. You can customize this for your particular compiler or, if compiling the library as part of your project, you can leave this undefined in order to have strong definitions of each library symbol.

2.5.12 Floating-point ABI

Default

```
#if defined(__riscv_float_abi_soft)
  #define __SEGGER_RTL_FP_ABI                      0
#elif defined(__riscv_float_abi_single)
  #define __SEGGER_RTL_FP_ABI                      1
#elif defined(__riscv_float_abi_double)
  #define __SEGGER_RTL_FP_ABI                      2
#else
  #error Cannot determine RISC-V floating-point ABI
#endif
```

Description

Configuration of the floating-point ABI in use is determined from the compiler-provided symbols `__riscv_float_abi_soft`, `__riscv_float_abi_single`, and `__riscv_float_abi_double`.

`__SEGGER_RTL_FP_ABI` must be set to 0 if `float` and `double` parameters are passed using integer registes, to 1 if `float` parameters are passed using floating registers and `double` parameters are passed using integer registers, and to 2 if both `float` and `double` parameters are passed using floating registers.

2.5.13 Floating-point hardware

Default

```
#if defined(__riscv_flen) && (__riscv_flen == 64)
  #define __SEGGER_RTL_FP_HW                      2
#elif defined(__riscv_flen) && (__riscv_flen == 32)
  #define __SEGGER_RTL_FP_HW                      1
#else
  #define __SEGGER_RTL_FP_HW                      0
#endif
```

Description

Floating-point hardware support is configured separately from the floating-point calling convention. Even if floating-point parameters are passed in integer registers, it is still possible that floating-point instructions operate on those parameters in the called function.

The ACLE symbol `__ARM_FP` is queried to determine the target core's floating-point ability and set `__SEGGER_RTL_FP_HW` appropriately.

`__SEGGER_RTL_FP_HW` is set to 0 to indicate that no floating-point hardware exists, to 1 to indicate that hardware exists to support `float` arithmetic, and to 2 to to indicate that hardware exists to support `double` arithmetic.

Unfortunately, a fix-up is required:

```
// Clang gets __ARM_FP wrong for the T16 target ISA indicating
// that floating-point instructions exist in this ISA -- which
// they don't.  Patch that definition up here.
#if __ARM_ARCH_ISA_THUMB == 1
  #undef  __SEGGER_RTL_FP_HW
  #define __SEGGER_RTL_FP_HW                      0
  #undef  __SEGGER_RTL_FP_ABI
  #define __SEGGER_RTL_FP_ABI                     0
#endif
```

2.5.14 SIMD instruction set extension availability

Default

```
#if defined(__riscv_dsp)
  #define __SEGGER_RTL_CORE_HAS_ISA_SIMD               1
#else
  #define __SEGGER_RTL_CORE_HAS_ISA_SIMD               0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_ISA_SIMD` must be set to 1 if the RISC-V P (packed SIMD) instruction set extension is present, and 0 otherwise.

The assembly-language integer and floating-point implementations benefit significantly in terms of reduced code size and increased execution speed with this instruction set extension.

2.5.15 Andes Performance Extension availability

Default

```
#if defined(__nds_v5)
  #define __SEGGER_RTL_CORE_HAS_ISA_ANDES_V5           1
#else
  #define __SEGGER_RTL_CORE_HAS_ISA_ANDES_V5           0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_ISA_ANDES_V5` must be set to 1 if the Andes Performance Extension is present, and 0 otherwise.

The assembly-language integer and floating-point implementations benefit in terms of reduced code size and increased execution speed with this instruction set extension.

2.5.16 Multiply instruction availability

Default

```
#if defined(__riscv_mul)
  #define __SEGGER_RTL_CORE_HAS_MUL_MULH               1
#else
  #define __SEGGER_RTL_CORE_HAS_MUL_MULH               0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_MUL_MULH` must be set to 1 if the MUL and MULH instructions are present, and 0 otherwise.

The assembly-language integer and floating-point implementations benefit in terms of reduced code size and increased execution speed with the presence of these instructions.

2.5.17 Divide instruction availability

Default

```
#if defined(__riscv_div)
  #define __SEGGER_RTL_CORE_HAS_DIV                        1
#else
  #define __SEGGER_RTL_CORE_HAS_DIV                        0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_DIV` must be set to 1 if the DIV, DIVU, REM, and REMU instructions are present, and 0 otherwise.

The assembly-language integer and floating-point implementations benefit in terms of reduced code size and increased execution speed with the presence of these instructions.

2.5.18 Count-leading-zeros instruction availability

Default

```
#if  defined(__riscv_zbb)
  #define __SEGGER_RTL_CORE_HAS_CLZ                    1
#else
  #define __SEGGER_RTL_CORE_HAS_CLZ                    0
#endif

#if defined(__riscv_dsp)
  #define __SEGGER_RTL_CORE_HAS_CLZ32                  1
#else
  #define __SEGGER_RTL_CORE_HAS_CLZ32                  0
#endif
```

Description

The preprocessor symbol __SEGGER_RTL_CORE_HAS_CLZ must be set to 1 if the CLZ instruction from the RISC-V bit-manipulation extension is present, and 0 otherwise.

The preprocessor symbol __SEGGER_RTL_CORE_HAS_CLZ32 must be set to 1 if the SIMD CLZ32 instruction is present, and 0 otherwise.

The assembly-language integer and floating-point implementations benefit in terms of reduced code size and increased execution speed with the presence of these instructions.

The preprocessor symbol __SEGGER_RTL_CLZ_U32 is defined to expand to a way to use the CLZ instruction when the core is known to have one:

```
#if __SEGGER_RTL_CORE_HAS_CLZ || __SEGGER_RTL_CORE_HAS_CLZ32
  #define __SEGGER_RTL_CLZ_U32(X)        __builtin_clz(X)
#endif
```

2.5.19 Negated-logic instruction availability

Default

```
#if defined(__riscv_zbb)
  #define __SEGGER_RTL_CORE_HAS_ANDN_ORN_XORN          1
#else
  #define __SEGGER_RTL_CORE_HAS_ANDN_ORN_XORN          0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_ANDN_ORN_XORN` must be set to 1 if the ANDN, ORN, and XORN instructions from the RISC-V bit-manipulation extension are present, and 0 otherwise.

The assembly-language integer and floating-point implementations benefit in terms of reduced code size and increased execution speed with the presence of these instructions.

2.5.20 Bitfield instruction availability

Default

```
#if defined(__riscv_zbs)
  #define __SEGGER_RTL_CORE_HAS_BSET_BCLR_BINV_BEXT    1
#else
  #define __SEGGER_RTL_CORE_HAS_BSET_BCLR_BINV_BEXT    0
#endif
```

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_BSET_BCLR_BINV_BEXT` must be set to 1 if the BSET, BCLR, BINV, and BEXT instructions from the RISC-V bit-manipulation extension are present, and 0 otherwise.

The assembly-language integer and floating-point implementations benefit in terms of reduced code size and increased execution speed with the presence of these instructions.

2.5.21 Shift-and-add instruction availability

Default

```
#if defined(__riscv_zba)
  #define __SEGGER_RTL_CORE_HAS_SHxADD                 1
#else
  #define __SEGGER_RTL_CORE_HAS_SHxADD                 0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_SHxADD` must be set to 1 if the SH1ADD, SH2ADD, and SH3ADD instructions from the RISC-V bit-manipulation extension are present, and 0 otherwise.

The assembly-language integer and floating-point implementations benefit in terms of reduced code size and increased execution speed with the presence of these instructions.

2.5.22 Divide-remainder macro-op fusion availability

Default

```
#ifndef   __SEGGER_RTL_CORE_HAS_FUSED_DIVREM
  #define __SEGGER_RTL_CORE_HAS_FUSED_DIVREM           0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_CORE_HAS_FUSED_DIVREM` can be set to 1 if the target supports macro-op fusion of DIV and REM instructions, and 0 otherwise.

As of the time of writing, SEGGER have not seen a core with macro-op fusion of division with remainder and define this to zero unconditionally.

2.5.23 Branch-free code preference

Default

```
#ifndef   __SEGGER_RTL_PREFER_BRANCH_FREE_CODE
  #define __SEGGER_RTL_PREFER_BRANCH_FREE_CODE         0
#endif
```

Description

The preprocessor symbol `__SEGGER_RTL_PREFER_BRANCH_FREE_CODE` must be set to 1 to select branch-free code sequences in preference to branching code sequences.

Whether a target benefits from branch-free code depends upon branch penalties for mispredicted branches and how often these occur in practice. By default this is set to zero, assuming that the branch predictor is more often correct than incorrect, and also reducing overall code size.

For high-performance cores, it may be advantageous to compile using branch-free code.

Chapter 3

Runtime support

This section describes how to set up the execution environment for the C library.

3.1 Getting to main() and then exit()

Before entering `main()` the execution environment must be set up such that the C standard library will function correctly.

This section does not describe the compiler or linker support for placing code and data into memory, how to configure any RAM, or how to zero memory required for zero-initialized data. For this, please refer to your toolset compiler and linker documentation.

Nor does this section document how to call constructors and destructors in the correct order. Again, refer to your toolset manuals.

3.1.1 At-exit function support

After returning from `main()` or by calling `exit()`, any registered `atexit` functions must be called to close down. To do this, call `__SEGGER_RTL_execute_at_exit_fns()` from the runtime startup immdiately after the call to `main()`.

3.1.2 Locale name buffer

For ANSI-correct correct functioning `setlocale()`, `__SEGGER_RTL_set_locale_name_buffer()` must be used. If `__SEGGER_RTL_set_locale_name_buffer()` is not used to set a name buffer, `setlocale()` will still set the locale but will return `NULL` rather than the previous locale.

Please refer to *setlocale* on page 262 for further information.

3.2 Dynamic storage and the heap

emRun provides three heap implementations which you may choose from:

- A real-time heap where allocation and deallocation have O(1) performance, provided in `heapops_realtime.c` and `__SEGGER_RTL_RTHEAP.c`.
- A low-overhead best-fit heap where allocation and deallocation have very little internal fragmentation, provided in `heapops_basic.c`. This implementation has no support for `aligned_alloc()`.
- An allocate-only heap where deallocation and reallocation are not implemented, provided in `heapops_minimal.c`. This implementation only supports `malloc()` and `calloc()`.

3.2.1 Multithreaded protection for the heap

Heap functions (allocation, reallocation, deallocation) can be protected from reentrancy in a multithreaded environment by implementing lock and unlock functions. By default, these functions do nothing and memory allocation functions are not protected.

See *__SEGGER_RTL_X_heap_lock* on page 1133 and *__SEGGER_RTL_X_heap_unlock* on page 1134.

3.2.2 Setting up the heap

Whichever heap implementation is chosen, the dynamic memory managed by the heap must be initialized by calling `__SEGGER_RTL_init_heap()` passing the base address of the managed area and its size in bytes.

This initialization is typically carried out as part of system startup, before any constructors are called.

3.3 Input and output

The way characters and strings are printed and scanned can be configured in multiple ways. This section describes how a generic implementation works, how to optimize input and output for other technologies such as SEGGER RTT and SEGGER semihosting, and how to optimized for UART-style I/O.

3.3.1 Standard input and output

Standard input and output are performed using the low-level functions `__SEGGER_RTL_X_file_read()` and `__SEGGER_RTL_X_file_write()`, These functions are defined in the file `__SEGGER_RTL.h` as follows:

```
int __SEGGER_RTL_X_file_read  (__SEGGER_RTL_FILE *stream, char *s, unsigned len);
int __SEGGER_RTL_X_file_write (__SEGGER_RTL_FILE *stream, const char *s, unsigned len);
```

The type `__SEGGER_RTL_FILE` and its corresponding standard C version `FILE` are defined opaqely by `__SEGGER_RTL.h` as:

```
typedef struct __SEGGER_RTL_FILE_IMPL __SEGGER_RTL_FILE;
typedef struct __SEGGER_RTL_FILE_IMPL FILE;
```

This leaves the exact structure of a `FILE` and the implementation of file I/O to the library integrator. The following are sample implementations for SEGGER RTT, SEGGER Semihosting, and a version that supports only output to a UART.

3.3.2 Using SEGGER RTT for I/O

Complete listing

```
/*********************************************************************
*                    (c) SEGGER Microcontroller GmbH                 *
*                         The Embedded Experts                       *
*                            www.segger.com                          *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "__SEGGER_RTL_Int.h"
#include "stdio.h"
#include "RTT/SEGGER_RTT.h"

/*********************************************************************
*
*       Local types
*
**********************************************************************
*/

struct __SEGGER_RTL_FILE_impl {
  int handle;
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static FILE __SEGGER_RTL_stdin_file  = { 0 };  // stdin reads from RTT buffer #0
static FILE __SEGGER_RTL_stdout_file = { 0 };  // stdout writes to RTT buffer #0
static FILE __SEGGER_RTL_stderr_file = { 0 };  // stdout writes to RTT buffer #0
static int  __SEGGER_RTL_stdin_ungot = EOF;

/*********************************************************************
*
*       Public data
*
**********************************************************************
*/

FILE *stdin  = &__SEGGER_RTL_stdin_file;
FILE *stdout = &__SEGGER_RTL_stdout_file;
FILE *stderr = &__SEGGER_RTL_stderr_file;

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       __SEGGER_RTL_stdin_getc()
*
*  Function description
*    Get character from standard input.
*
*  Return value
*    Character received.
```

```
*
*  Additional information
*    This function never fails to deliver a character.
*/
static char __SEGGER_RTL_stdin_getc(void) {
  int  r;
  char c;
  //
  if (__SEGGER_RTL_stdin_ungot != EOF) {
    c = __SEGGER_RTL_stdin_ungot;
    __SEGGER_RTL_stdin_ungot = EOF;
  } else {
    do {
      r = SEGGER_RTT_Read(stdin->handle, &c, 1);
    } while (r == 0);
  }
  //
  return c;
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       __SEGGER_RTL_X_file_stat()
*
*  Function description
*    Get file status.
*
*  Parameters
*    stream - Pointer to file.
*
*  Additional information
*    Low-overhead test to determine if stream is valid.  If stream
*    is a valid pointer and the stream is open, this function must
*    succeed.  If stream is a valid pointer and the stream is closed,
*    this function must fail.
*
*    The implementation may optionally determine whether stream is
*    a valid pointer: this may not always be possible and is not
*    required, but may assist debugging when clients provide wild
*    pointers.
*
*  Return value
*    <  0 - Failure, stream is not a valid file.
*    >= 0 - Success, stream is a valid file.
*/
int __SEGGER_RTL_X_file_stat(__SEGGER_RTL_FILE *stream) {
  if (stream == stdin || stream == stdout || stream == stderr) {
    return 0;
  } else {
    return EOF;
  }
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_bufsize()
*
*  Function description
*    Get stream buffer size.
*
*  Parameters
*    stream - Pointer to file.
*
*  Additional information
*    Returns the number of characters to use for buffered I/O on
*    the file stream.  The I/O buffer is allocated on the stack
*    for the duration of the I/O call, therefore this value should
*    not be set arbitrarily large.
*
```

```
*    For unbuffered I/O, return 1.
*
*  Return value
*    Nonzero number of characters to use for buffered I/O; for
*    unbuffered I/O, return 1.
*/
int __SEGGER_RTL_X_file_bufsize(__SEGGER_RTL_FILE *stream) {
  //
  __SEGGER_RTL_USE_PARA(stream);
  //
  return 64;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_read()
*
*  Function description
*    Read data from file.
*
*  Parameters
*    stream - Pointer to file to read from.
*    s      - Pointer to object that receives the input.
*    len    - Number of characters to read from file.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
*
*  Additional information
*    Reading from any stream other than stdin results in an error.
*/
int __SEGGER_RTL_X_file_read(__SEGGER_RTL_FILE * stream,
                             char              * s,
                             unsigned            len) {
  int c;
  //
  if (stream == stdin) {
    c = 0;
    while (len > 0) {
      *s++ = __SEGGER_RTL_stdin_getc();
      --len;
      ++c;
    }
  } else {
    c = EOF;
  }
  //
  return c;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_flush()
*
*  Function description
*    Flush unwritten data to file.
*
*  Parameters
*    stream - Pointer to file.
*
*  Return value
*    <  0 - Failure, file cannot be flushed or was not successfully flushed.
*    == 0 - Success, unwritten data is flushed.
*/
int __SEGGER_RTL_X_file_flush(__SEGGER_RTL_FILE *stream) {
  //
  __SEGGER_RTL_USE_PARA(stream);
  //
  return 0;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_write()
*
```

```
*  Function description
*    Write data to file.
*
*  Parameters
*    stream - Pointer to file to write to.
*    s      - Pointer to object to write to file.
*    len    - Number of characters to write to the file.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
*
*  Additional information
*    stdout is directed to RTT buffer #0; stderr is directed to RTT buffer #1;
*    writing to any stream other than stdout or stderr results in an error
*/
int __SEGGER_RTL_X_file_write(__SEGGER_RTL_FILE *stream, const char *s, unsigned len) {
  return SEGGER_RTT_Write(stream->handle, s, len);
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_unget()
*
*  Function description
*    Push character back to stream.
*
*  Parameters
*    stream - Pointer to file to push back to.
*    c      - Character to push back.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
*
*  Additional information
*    Push-back is only supported for standard input, and
*    only a single-character pushback buffer is implemented.
*/
int __SEGGER_RTL_X_file_unget(__SEGGER_RTL_FILE *stream, int c) {
  if (stream == stdin) {
    if (c != EOF && __SEGGER_RTL_stdin_ungot == EOF) {
      __SEGGER_RTL_stdin_ungot = c;
    } else {
      c = EOF;
    }
  } else {
    c = EOF;
  }
  //
  return c;
}

/*************************** End of file ****************************/
```

3.3.3 Using SEGGER semihosting for I/O

Complete listing

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "__SEGGER_RTL_Int.h"
#include "stdio.h"
#include "SEMIHOST/SEGGER_SEMIHOST.h"

/*********************************************************************
*
*       Local types
*
**********************************************************************
*/

struct __SEGGER_RTL_FILE_impl {
  int handle;
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static FILE __SEGGER_RTL_stdin_file  = { SEGGER_SEMIHOST_STDIN  };
static FILE __SEGGER_RTL_stdout_file = { SEGGER_SEMIHOST_STDOUT };
static FILE __SEGGER_RTL_stderr_file = { SEGGER_SEMIHOST_ERROUT };
static int  __SEGGER_RTL_stdin_ungot = EOF;

/*********************************************************************
*
*       Public data
*
**********************************************************************
*/

FILE *stdin  = &__SEGGER_RTL_stdin_file;
FILE *stdout = &__SEGGER_RTL_stdout_file;
FILE *stderr = &__SEGGER_RTL_stderr_file;

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       __SEGGER_RTL_X_file_stat()
*
*  Function description
*    Get file status.
*
*  Parameters
*    stream - Pointer to file.
```

```
*
*  Additional information
*    Low-overhead test to determine if stream is valid.  If stream
*    is a valid pointer and the stream is open, this function must
*    succeed.  If stream is a valid pointer and the stream is closed,
*    this function must fail.
*
*    The implementation may optionally determine whether stream is
*    a valid pointer: this may not always be possible and is not
*    required, but may assist debugging when clients provide wild
*    pointers.
*
*  Return value
*    <  0 - Failure, stream is not a valid file.
*    >= 0 - Success, stream is a valid file.
*/
int __SEGGER_RTL_X_file_stat(__SEGGER_RTL_FILE *stream) {
  if (stream == stdin || stream == stdout || stream == stderr) {
    return 0;
  } else {
    return EOF;
  }
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_bufsize()
*
*  Function description
*    Get stream buffer size.
*
*  Parameters
*    stream - Pointer to file.
*
*  Additional information
*    Returns the number of characters to use for buffered I/O on
*    the file stream.  The I/O buffer is allocated on the stack
*    for the duration of the I/O call, therefore this value should
*    not be set arbitrarily large.
*
*    For unbuffered I/O, return 1.
*
*  Return value
*    Nonzero number of characters to use for buffered I/O; for
*    unbuffered I/O, return 1.
*/
int __SEGGER_RTL_X_file_bufsize(__SEGGER_RTL_FILE *stream) {
  return 64;
}

/*********************************************************************
*
*       __SEGGER_RTL_stdin_getc()
*
*  Function description
*    Get character from standard input.
*
*  Return value
*    >= 0   - Character read.
*    == EOF - End of stream or error reading.
*
*  Additional information
*    This function never fails to deliver a character.
*/
static int __SEGGER_RTL_stdin_getc(void) {
  int  r;
  char c;
  //
  if (__SEGGER_RTL_stdin_ungot != EOF) {
    c = __SEGGER_RTL_stdin_ungot;
    __SEGGER_RTL_stdin_ungot = EOF;
    r = 0;
  } else {
    r = SEGGER_SEMIHOST_ReadC();
  }
  //
  return r < 0 ? EOF : c;
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       __SEGGER_RTL_X_file_read()
*
*  Function description
*    Read data from file.
*
*  Parameters
*    stream - Pointer to file to read from.
*    s      - Pointer to object that receives the input.
*    len    - Number of characters to read from file.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
*
*  Additional information
*    Reading from any stream other than stdin results in an error.
*/
int __SEGGER_RTL_X_file_read(__SEGGER_RTL_FILE * stream,
                             char              * s,
                             unsigned            len) {
  int c;
  //
  if (stream == stdin) {
    c = 0;
    while (len > 0) {
      *s++ = __SEGGER_RTL_stdin_getc();
      --len;
    }
  } else {
    c = SEGGER_SEMIHOST_Read(stream->handle, s, len);
  }
  //
  return c;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_write()
*
*  Function description
*    Write data to file.
*
*  Parameters
*    stream - Pointer to file to write to.
*    s      - Pointer to object to write to file.
*    len    - Number of characters to write to the file.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
*/
int __SEGGER_RTL_X_file_write(__SEGGER_RTL_FILE *stream, const char *s, unsigned len) {
  int r;
  //
  r = SEGGER_SEMIHOST_Write(stream->handle, s, len);
  if (r < 0) {
    r = EOF;
  }
  //
  return r;
}

/*********************************************************************
*
```

```
*       __SEGGER_RTL_X_file_unget()
*
*  Function description
*    Push character back to stream.
*
*  Parameters
*    stream - Pointer to stream to push back to.
*    c      - Character to push back.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
*
*  Additional information
*    Push-back is only supported for standard input, and
*    only a single-character pushback buffer is implemented.
*/
int __SEGGER_RTL_X_file_unget(__SEGGER_RTL_FILE *stream, int c) {
  if (stream == stdin) {
    if (c != EOF && __SEGGER_RTL_stdin_ungot == EOF) {
      __SEGGER_RTL_stdin_ungot = c;
    } else {
      c = EOF;
    }
  } else {
    c = EOF;
  }
  //
  return c;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_flush()
*
*  Function description
*    Flush unwritten data to file.
*
*  Parameters
*    stream - Pointer to file.
*
*  Return value
*    <  0 - Failure, file cannot be flushed or was not successfully flushed.
*    == 0 - Success, unwritten data is flushed.
*/
int __SEGGER_RTL_X_file_flush(__SEGGER_RTL_FILE *stream) {
  return 0;
}

/*************************** End of file ****************************/
```

3.3.4 Using a UART for I/O

Complete listing

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "__SEGGER_RTL_Int.h"
#include "stdio.h"

/*********************************************************************
*
*       Local types
*
**********************************************************************
*/

struct __SEGGER_RTL_FILE_impl {
  int handle;  // At least one field required (but unused) to ensure
               // the three file descriptors have unique addresses.
};

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

#ifdef __cplusplus
extern "C"
#endif
int metal_tty_putc(int c);  // UART output function

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static FILE __SEGGER_RTL_stdin  = { 0 };
static FILE __SEGGER_RTL_stdout = { 1 };
static FILE __SEGGER_RTL_stderr = { 2 };

/*********************************************************************
*
*       Public data
*
**********************************************************************
*/

FILE *stdin  = &__SEGGER_RTL_stdin;
FILE *stdout = &__SEGGER_RTL_stdout;
FILE *stderr = &__SEGGER_RTL_stderr;

/*********************************************************************
*
*       Public code
*
**********************************************************************
```

```
*/

/*********************************************************************
*
*       __SEGGER_RTL_X_file_stat()
*
*  Function description
*    Get file status.
*
*  Parameters
*    stream - Pointer to file.
*
*  Additional information
*    Low-overhead test to determine if stream is valid.  If stream
*    is a valid pointer and the stream is open, this function must
*    succeed.  If stream is a valid pointer and the stream is closed,
*    this function must fail.
*
*    The implementation may optionally determine whether stream is
*    a valid pointer: this may not always be possible and is not
*    required, but may assist debugging when clients provide wild
*    pointers.
*
*  Return value
*    <  0 - Failure, stream is not a valid file.
*    >= 0 - Success, stream is a valid file.
*/
int __SEGGER_RTL_X_file_stat(__SEGGER_RTL_FILE *stream) {
  if (stream == stdin || stream == stdout || stream == stderr) {
    return 0;
  } else {
    return EOF;
  }
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_bufsize()
*
*  Function description
*    Get stream buffer size.
*
*  Parameters
*    stream - Pointer to file.
*
*  Additional information
*    Returns the number of characters to use for buffered I/O on
*    the file stream.  The I/O buffer is allocated on the stack
*    for the duration of the I/O call, therefore this value should
*    not be set arbitrarily large.
*
*    For unbuffered I/O, return 1.
*
*  Return value
*    Nonzero number of characters to use for buffered I/O; for
*    unbuffered I/O, return 1.
*/
int __SEGGER_RTL_X_file_bufsize(__SEGGER_RTL_FILE *stream) {
  return 1;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_read()
*
*  Function description
*    Read data from file.
*
*  Parameters
*    stream - Pointer to file to read from.
*    s      - Pointer to object that receives the input.
*    len    - Number of characters to read from file.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
```

```
*
*  Additional information
*    As input from the UART is not supported, this function always fails.
*/
int __SEGGER_RTL_X_file_read(__SEGGER_RTL_FILE * stream,
                             char              * s,
                             unsigned            len) {
  return EOF;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_write()
*
*  Function description
*    Write data to file.
*
*  Parameters
*    stream - Pointer to file to write to.
*    s      - Pointer to object to write to file.
*    len    - Number of characters to write to the file.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
*
*  Additional information
*    Writing to any file other than stdout or stderr results in an error.
*/
int __SEGGER_RTL_X_file_write(__SEGGER_RTL_FILE *stream, const char *s, unsigned len) {
  int r;
  //
  if (stream == stdout || stream == stderr) {
    r = len;
    while (len > 0) {
      metal_tty_putc(*s++);
      --len;
    }
  } else {
    r = EOF;
  }
  //
  return r;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_unget()
*
*  Function description
*    Push character back to stream.
*
*  Parameters
*    stream - Pointer to file to push back to.
*    c      - Character to push back.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Failure.
*
*  Additional information
*    As input from the UART is not supported, this function always fails.
*/
int __SEGGER_RTL_X_file_unget(__SEGGER_RTL_FILE *stream, int c) {
  return EOF;
}

/*********************************************************************
*
*       __SEGGER_RTL_X_file_flush()
*
*  Function description
*    Flush unwritten data to file.
*
*  Parameters
*    stream - Pointer to file.
```

```
*
*  Return value
*    <  0 - Failure, file cannot be flushed or was not successfully flushed.
*    == 0 - Success, unwritten data is flushed.
*/
int __SEGGER_RTL_X_file_flush(__SEGGER_RTL_FILE *stream) {
  return 0;
}

/*************************** End of file ****************************/
```

3.4 Thread safety

Functions in emRun are written with varying levels of thread-safe operation. Some functions are inherently re-entrant and thread-safe, some are thread-safe if configured to be so, and some are never thread-safe.

The following section desfribe the various ways that the execution environment for a C or C++ program can be configured.

No threading

In this case there are no separate threads of execution save for interrupt and exception handlers. In this case, emRun will not be required to support thread-local storage and the `__SEGGER_RTL_THREAD` macro can be defined to be empty and the heap-lock and heap-unlock functions can be empty.

It is the user's responsibility to ensure there is no conflict in the use of shared data between mainline code and interrupt-handling code.

In this scenario, all functions are inherently thread-safe as there is no threading.

Threading with no RTOS thread-local support

In this case there are separate threads of execution but only a single instance of emRun private data. As such, any function that manipulates emRun private data, directly or indirectly, is thread-unsafe.

Although emRun can be configured this way, it is highly likely that cross-contamination of emRun private data will occur. For instance, `errno` will be shared between all threads and code such as the following is prone to failure:

```
errno = 0;
d = strtod(sInput, NULL);
if (errno != 0) { ... }
```

At first glance, the above code looks entirely reasonable. However, in this configuration a thread could be scheduled between setting and reading `errno`, potentially corrupting the value of `errno` for the original thread. Such errors are very hard to track down.

In this configuration, there can be no guarantee made regarding thread-safety of emRun and the "Thread safety" section in each function desciption must be ignored.

Threading with RTOS thread-local support

In this case there are separate threads of execution with each thread receiving its own copy of emRun private data. As such, any function that manipulates private data, directly or indirectly, is thread-safe.

In contrast to the previous configuration, each thread receives its own private copy of `errno` and cross-contamination of emRun runtime data will not occur inside emRun functions.

3.4.1 Functions that are re-entrant and thread-safe

Functions that only take scalar data (chars, integers, reals) and do not read global state are both re-entrant and thread-safe. For instance, `sin()` is thread-safe as the floating-point environment is per-thread and `sin()` does not use any global state variables.

Other functions, such as `strcat()`, are re-entrant and thread-safe only if the objects they operate on are not shared between threads. For instance, it is not possible for two or more threads to use `strcat()` to concatenate data into a single array shared between the two threads, such as appending to some in-memory error or trace log.

3.4.2 Functions that are thread-safe if configured

Per-thread global data in emRun is declared using the `__SEGGER_RTL_THREAD` macro; see *Thread-local storage* on page 55.

errno

The errno macro is thread-safe if both emRun and the underlying RTOS is configured to support thread-local data.

If you have not configured per-thread storage or the RTOS does not support thread-local storage, there will be a single instance of emRun private data shared between all threads and therefore any function mentioned above, or any function that potentially sets errno, directly or indirectly, will write a single instance of it and will not be thread-safe.

String and multi-byte functions

The following functions are thread-safe if both emRun and the underlying RTOS is configured to support thread-local data.

- `strtok()`
- `wcsrtombs()`
- `wctomb()`
- `wcrtomb()`
- `mbrlen()`
- `mbrtowc()`
- `mbtowc()`
- `mbsrtowcs()`

If you have not configured per-thread storage or the RTOS does not support thread-local storage, there will be a single instance of emRun private data shared between all threads and therefore any function mentioned above, or any function that potentially sets uses these directly or indirectly, will write a single instance of emRun private data and will not be thread-safe.

Note that it is well understood that functions maintaining global state are undesirable from a program design and multi-threading perspective. This has been recognized by industry standards bodies, such as The Open Group, and this has led to the introduction of "restartable" functions in, for instance, the POSIX.1 standard. emRun implements restartable functions that appear in POSIX.1, such as `strtok_r()`.

Restartable functions are preferable to multi-threading-enabled versions of the standard functions because they do not introduce a per-thread overhead (where threads that do not use e.g. `strtok()` still pay to have thread-local state reserved for it) and also because access to thread-local data is more expensive than accessing data provided as an additional parameter to the function.

Locale-aware functions

All functions that use or set a locale are thread-safe if both emRun and the underlying RTOS is configured to support thread-local data. This includes all character type and conversion functions, multibyte functions, and locale maipulation funtions.

Heap functions

Heap functions are thread-safe if and only if the heap-lock and heap-unlock functions `__SEGGER_RTL_X_heap_lock()` and `__SEGGER_RTL_X_heap_unlock()` are present and prevent simultaneous use of the shared heap. These two functions ensure that the heap is in use by a single execution context only. If these functions are not provided, the heap is unprotected and is not thread-safe.

3.4.3 Functions that are never thread-safe

All I/O functions that work on streams are never thread safe. A design goal of the C library is to be efficient and, as such, it is not possible to share files and streams between threads.

Should this be required, the user is responsible for using an appropriate locking mechanism outside of emRun to ensure no stream is simultaneously in use by two or more threads.

3.5 Atomic datatype support

Athrough compilers will lay down instructions for data declared `_Atomic`, some C-level operations will not be able to be achieved atomically.

To support this, emRun provides support for both GCC-defined and Clang-defined atomic support functions which are implemented in terms of three C functions that the user must provide:

- `SEGGER_RTL_X_atomic_lock()`
- `SEGGER_RTL_X_atomic_unlock()`
- `SEGGER_RTL_X_atomic_synchronize()`

Chapter 4

C library API

This section describes the C library ABI.

Conformance section

Where a conformance section is present, it defined the conformance of the function to a particular standards.

The non-C standards are:

- **POSIX.1-2001**: The function or object is defined by POSIX.1-2001, and is defined in later POSIX.1 versions, unless otherwise indicated.
- **POSIX.1-2007**: The function or object is defined by POSIX.1-2007, and is defined in later POSIX.1 versions, unless otherwise indicated.
- **POSIX.1-2008**: The function or object is defined by POSIX.1-2008, and is defined in later POSIX.1 versions, unless otherwise indicated.

Thread-safety sections

Where applicable, thread-safety relating to a multi-threaded system is described using the following:

- **Unsafe**: This function is never safe to use in a multi-threaded environment and requires callers to ensure only a single thread of execution uses this function.
- **Safe [if configured]**: This function is safe to use in a multi-threaded system only if emRun is configured to be thread-safe in co-operation with the underlying RTOS. Typically this relates to the heap and any function that uses per-thread (thread-local) data as described in previous sections.
- **Safe**: This function is always safe to use in a multi-threaded system. Typically this relates to state-free functions such as `sin()` and `div()`. This function is also safe to use between threads if the objects pointed to by any user-supplied pointers are in use by a single thread only. Typically this relates to functions such as `strcat()` which are thread-safe only if the objects passed into the function are not shared between threads.
- **Not applicable**: Thread-safety does not apply to this function as it is not intended for execution in a threading environment. Typically this relates to runtime functions that initialize to deinitialize the runtime system.

4.1 <assert.h>

4.1.1 Assertion functions

Function	Description
`assert`	Place assertion.

4.1.1.1 assert

Description

Place assertion.

Definition

```
#define assert(e)    ...
```

Additional information

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 as:

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

If `NDEBUG` is not defined as a macro name at the point in the source file where `<assert.h>` is included, the `assert()` macro expands to a void expression that calls `__SEGGER_RTL_X_assert()`.

When such an assert is executed and `e` is false, `assert()` calls the function `__SEGGER_RTL_X_assert()` with information about the particular call that failed: the text of the argument, the name of the source file, and the source line number. These are the stringized expression and the values of the preprocessing macros `__FILE__` and `__LINE__`.

Notes

The `assert()` macro is redefined according to the current state of `NDEBUG` each time that `<assert.h>` is included.

4.2 <complex.h>

emRun provides complex math library functions, including all of those required by ISO C99. These functions are implemented to balance performance with correctness. Because producing the correctly rounded result may be prohibitively expensive, these functions are designed to efficiently produce a close approximation to the correctly rounded result. In most cases, the result produced is within +/-1 ulp of the correctly rounded result, though there may be cases where there is greater inaccuracy.

4.2.1 Manipulation functions

Function	Description
`cabs()`	Compute magnitude, double complex.
`cabsf()`	Compute magnitude, float complex.
`cabsl()`	Compute magnitude, long double complex.
`carg()`	Compute phase, double complex.
`cargf()`	Compute phase, float complex.
`cargl()`	Compute phase, long double complex.
`cimag()`	Imaginary part, double complex.
`cimagf()`	Imaginary part, float complex.
`cimagl()`	Imaginary part, long double complex.
`creal()`	Real part, double complex.
`crealf()`	Real part, float complex.
`creall()`	Real part, long double complex.
`cproj()`	Project, double complex.
`cprojf()`	Project, float complex.
`cprojl()`	Project, long double complex.
`conj()`	Conjugate, double complex.
`conjf()`	Conjugate, float complex.
`conjl()`	Conjugate, long double complex.

4.2.1.1 cabs()

Description

Compute magnitude, double complex.

Prototype

```
double cabs(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

The magnitude of x, |x|.

Thread safety

Safe.

4.2.1.2 cabsf()

Description

Compute magnitude, float complex.

Prototype

```
float cabsf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

The magnitude of x, |x|.

Thread safety

Safe.

4.2.1.3 cabsl()

Description

Compute magnitude, long double complex.

Prototype

```
long double cabsl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

The magnitude of x, |x|.

Thread safety

Safe.

4.2.1.4 carg()

Description

Compute phase, double complex.

Prototype

```
double carg(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute phase of.

Return value

The phase of x.

Thread safety

Safe.

4.2.1.5 cargf()

Description

Compute phase, float complex.

Prototype

```
float cargf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute phase of.

Return value

The phase of x.

Thread safety

Safe.

4.2.1.6 cargl()

Description

Compute phase, long double complex.

Prototype

```
long double cargl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute phase of.

Return value

The phase of x.

Thread safety

Safe.

4.2.1.7 cimag()

Description

Imaginary part, double complex.

Prototype

```
double cimag(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

The imaginary part of the complex value.

Thread safety

Safe.

4.2.1.8 cimagf()

Description

Imaginary part, float complex.

Prototype

```
float cimagf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

The imaginary part of the complex value.

Thread safety

Safe.

4.2.1.9 cimagl()

Description

Imaginary part, long double complex.

Prototype

```
long double cimagl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

The imaginary part of the complex value.

Thread safety

Safe.

4.2.1.10 creal()

Description

Real part, double complex.

Prototype

```
double creal(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

The real part of the complex value.

Thread safety

Safe.

4.2.1.11 crealf()

Description

Real part, float complex.

Prototype

```
float crealf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

The real part of the complex value.

Thread safety

Safe.

4.2.1.12 creall()

Description

Real part, long double complex.

Prototype

```
long double creall(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

The real part of the complex value.

Thread safety

Safe.

4.2.1.13 cproj()

Description

Project, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX cproj(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to project.

Return value

The projection of x to the Reimann sphere.

Additional information

x projects to x, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If x has an infinite part, then cproj(x) is be equivalent to:

- INFINITY + I * copysign(0.0, cimag(x))

Thread safety

Safe.

4.2.1.14 cprojf()

Description

Project, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX cprojf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to project.

Return value

The projection of x to the Reimann sphere.

Additional information

x projects to x, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If x has an infinite part, then cproj(x) is be equivalent to:

- INFINITY + I * copysign(0.0, cimag(x))

Thread safety

Safe.

4.2.1.15 cprojl()

Description

Project, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX cprojl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to project.

Return value

The projection of x to the Reimann sphere.

Additional information

x projects to x, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If x has an infinite part, then cproj(x) is be equivalent to:

- INFINITY + I * copysignl(0.0, cimagl(x))

Thread safety

Safe.

4.2.1.16 conj()

Description

Conjugate, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX conj(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to conjugate.

Return value

The complex conjugate of x.

Thread safety

Safe.

4.2.1.17 conjf()

Description

Conjugate, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX conjf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to conjugate.

Return value

The complex conjugate of x.

Thread safety

Safe.

4.2.1.18 conjl()

Description

Conjugate, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX conjl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to conjugate.

Return value

The complex conjugate of x.

Thread safety

Safe.

4.2.2 Trigonometric functions

Function	Description
`csin()`	Compute sine, double complex.
`csinf()`	Compute sine, float complex.
`csinl()`	Compute sine, long double complex.
`ccos()`	Compute cosine, double complex.
`ccosf()`	Compute cosine, float complex.
`ccosl()`	Compute cosine, long double complex.
`ctan()`	Compute tangent, double complex.
`ctanf()`	Compute tangent, float complex.
`ctanl()`	Compute tangent, long double complex.
`casin()`	Compute inverse sine, double complex.
`casinf()`	Compute inverse sine, float complex.
`casinl()`	Compute inverse sine, long double complex.
`cacos()`	Compute inverse cosine, double complex.
`cacosf()`	Compute inverse cosine, float complex.
`cacosl()`	Compute inverse cosine, long double complex.
`catan()`	Compute inverse tangent, double complex.
`catanf()`	Compute inverse tangent, float complex.
`catanl()`	Compute inverse tangent, long double complex.

4.2.2.1 csin()

Description

Compute sine, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX csin(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute sine of.

Return value

The sine of x.

Thread safety

Safe.

4.2.2.2 csinf()

Description

Compute sine, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX csinf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute sine of.

Return value

The sine of x.

Thread safety

Safe.

4.2.2.3 csinl()

Description

Compute sine, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX csinl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute sine of.

Return value

The sine of x.

Thread safety

Safe.

4.2.2.4 ccos()

Description

Compute cosine, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX ccos(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute cosine of.

Return value

The cosine of x.

Thread safety

Safe.

4.2.2.5 ccosf()

Description

Compute cosine, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX ccosf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute cosine of.

Return value

The cosine of x.

Thread safety

Safe.

4.2.2.6 ccosl()

Description

Compute cosine, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX ccosl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute cosine of.

Return value

The cosine of x.

Thread safety

Safe.

4.2.2.7 ctan()

Description

Compute tangent, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX ctan(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute tangent of.

Return value

The tangent of x.

Thread safety

Safe.

4.2.2.8 ctanf()

Description

Compute tangent, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX ctanf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute tangent of.

Return value

The tangent of x.

Thread safety

Safe.

4.2.2.9 ctanl()

Description

Compute tangent, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX ctanl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute tangent of.

Return value

The tangent of x.

Thread safety

Safe.

4.2.2.10 casin()

Description

Compute inverse sine, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX casin(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

Inverse sine of x.

Notes

casin(z) = -i casinh(i.z)

Thread safety

Safe.

4.2.2.11 casinf()

Description

Compute inverse sine, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX casinf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

Inverse sine of x.

Notes

casin(z) = -i casinh(i.z)

Thread safety

Safe.

4.2.2.12 casinl()

Description

Compute inverse sine, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX casinl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

Inverse sine of x.

Notes

casinl(z) = -i casinhl(i.z)

Thread safety

Safe.

4.2.2.13 cacos()

Description

Compute inverse cosine, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX cacos(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse cosine of.

Return value

The inverse cosine of x.

Thread safety

Safe.

4.2.2.14 cacosf()

Description

Compute inverse cosine, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX cacosf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse cosine of.

Return value

The inverse cosine of x.

Thread safety

Safe.

4.2.2.15 cacosl()

Description

Compute inverse cosine, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX cacosl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse cosine of.

Return value

The inverse cosine of x.

Thread safety

Safe.

4.2.2.16 catan()

Description

Compute inverse tangent, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX catan(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

Inverse tangent of x.

Notes

catan(z) = -i catanh(i.z)

Thread safety

Safe.

4.2.2.17 catanf()

Description

Compute inverse tangent, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX catanf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

Inverse tangent of x.

Notes

catan(z) = -i catanh(i.z)

Thread safety

Safe.

4.2.2.18 catanl()

Description

Compute inverse tangent, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX catanl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Argument.

Return value

Inverse tangent of x.

Notes

catanl(z) = -i catanhl(i.z)

Thread safety

Safe.

4.2.3 Hyperbolic functions

Function	Description
`csinh()`	Compute hyperbolic sine, double complex.
`csinhf()`	Compute hyperbolic sine, float complex.
`csinhl()`	Compute hyperbolic sine, long double complex.
`ccosh()`	Compute hyperbolic cosine, double complex.
`ccoshf()`	Compute hyperbolic cosine, float complex.
`ccoshl()`	Compute hyperbolic cosine, long double complex.
`ctanh()`	Compute hyperbolic tangent, double complex.
`ctanhf()`	Compute hyperbolic tangent, float complex.
`ctanhl()`	Compute hyperbolic tangent, long double complex.
`casinh()`	Compute inverse hyperbolic sine, double complex.
`casinhf()`	Compute inverse hyperbolic sine, float complex.
`casinhl()`	Compute inverse hyperbolic sine, long double complex.
`cacosh()`	Compute inverse hyperbolic cosine, double complex.
`cacoshf()`	Compute inverse hyperbolic cosine, float complex.
`cacoshl()`	Compute inverse hyperbolic cosine, long double complex.
`catanh()`	Compute inverse hyperbolic tangent, double complex.
`catanhf()`	Compute inverse hyperbolic tangent, float complex.
`catanhl()`	Compute inverse hyperbolic tangent, long double complex.

4.2.3.1 csinh()

Description

Compute hyperbolic sine, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX csinh(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic sine of.

Return value

The hyperbolic sine of x according to the following table:

Argument	csinh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + ∞*i*	±0 + NaN*i*, sign of real part unspecified
+0 + NaN*i*	±0 + NaN*i*, sign of real part unspecified
a + ∞*i*	NaN + NaN*i*, for positive finite a
a + NaN*i*	NaN + NaN*i*, for finite nonzero a
+∞ + 0*i*	+∞ + 0*i*
+∞ + b*i*	+∞×cos(b) + +∞×sin(b).i for positive finite b
+∞ + ∞*i*	±∞ + NaN*i*, sign of real part unspecified
+∞ + NaN*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- csinh(conj(z)) = conj(csinh(z)).

For arguments with a negative real component, use the equality:

- csinh(-z) = -csinh(z).

Thread safety

Safe.

4.2.3.2 csinhf()

Description

Compute hyperbolic sine, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX csinhf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic sine of.

Return value

The hyperbolic sine of x according to the following table:

Argument	csinh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + ∞*i*	±0 + NaN*i*, sign of real part unspecified
+0 + NaN*i*	±0 + NaN*i*, sign of real part unspecified
a + ∞*i*	NaN + NaN*i*, for positive finite a
a + NaN*i*	NaN + NaN*i*, for finite nonzero a
+∞ + 0*i*	+∞ + 0*i*
+∞ + b*i*	+∞×cos(b) + +∞×sin(b).i for positive finite b
+∞ + ∞*i*	±∞ + NaN*i*, sign of real part unspecified
+∞ + NaN*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- csinh(conj(z)) = conj(csinh(z)).

For arguments with a negative real component, use the equality:

- csinh(-z) = -csinh(z).

Thread safety

Safe.

4.2.3.3 csinhl()

Description

Compute hyperbolic sine, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX csinhl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic sine of.

Return value

The hyperbolic sine of x according to the following table:

Argument	csinh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + ∞*i*	±0 + NaN*i*, sign of real part unspecified
+0 + NaN*i*	±0 + NaN*i*, sign of real part unspecified
a + ∞*i*	NaN + NaN*i*, for positive finite a
a + NaN*i*	NaN + NaN*i*, for finite nonzero a
+∞ + 0*i*	+∞ + 0*i*
+∞ + b*i*	+∞×cos(b) + +∞×sin(b).i for positive finite b
+∞ + ∞*i*	±∞ + NaN*i*, sign of real part unspecified
+∞ + NaN*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- csinh(conj(z)) = conj(csinh(z)).

For arguments with a negative real component, use the equality:

- csinh(-z) = -csinh(z).

Thread safety

Safe.

4.2.3.4 ccosh()

Description

Compute hyperbolic cosine, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX ccosh(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic cosine of.

Return value

The hyperbolic cosine of x according to the following table:

Argument	ccosh(Argument)
+0 + 0*i*	+1 + 0*i*
+0 + ∞*i*	NaN + ±0*i*, sign of imaginary part unspecified
+0 + NaN*i*	NaN + ±0*i*, sign of imaginary part unspecified
a + ∞*i*	NaN + NaN*i*, for finite nonzero a
a + NaN*i*	NaN + NaN*i*, for finite nonzero a
+∞ + 0*i*	+∞ + 0*i*
+∞ + b*i*	+∞×cos(b) + Inf×sin(b).i for finite nonzero b
+∞ + ∞*i*	+∞ + NaN*i*
+∞ + NaN*i*	+∞ + NaN*i*
NaN + 0*i*	NaN + ±0*i*, sign of imaginary part unspecified
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- ccosh(conj(z)) = conj(ccosh(z)).

Thread safety

Safe.

4.2.3.5 ccoshf()

Description

Compute hyperbolic cosine, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX ccoshf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic cosine of.

Return value

The hyperbolic cosine of x according to the following table:

Argument	ccosh(Argument)
+0 + 0*i*	+1 + 0*i*
+0 + ∞*i*	NaN + ±0*i*, sign of imaginary part unspecified
+0 + NaN*i*	NaN + ±0*i*, sign of imaginary part unspecified
a + ∞*i*	NaN + NaN*i*, for finite nonzero a
a + NaN*i*	NaN + NaN*i*, for finite nonzero a
+∞ + 0*i*	+∞ + 0*i*
+∞ + b*i*	+∞×cos(b) + Inf×sin(b).i for finite nonzero b
+∞ + ∞*i*	+∞ + NaN*i*
+∞ + NaN*i*	+∞ + NaN*i*
NaN + 0*i*	NaN + ±0*i*, sign of imaginary part unspecified
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- ccosh(conj(z)) = conj(ccosh(z)).

Thread safety

Safe.

4.2.3.6 ccoshl()

Description

Compute hyperbolic cosine, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX ccoshl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic cosine of.

Return value

The hyperbolic cosine of x according to the following table:

Argument	ccosh(Argument)
+0 + 0*i*	+1 + 0*i*
+0 + ∞*i*	NaN + ±0*i*, sign of imaginary part unspecified
+0 + NaN*i*	NaN + ±0*i*, sign of imaginary part unspecified
a + ∞*i*	NaN + NaN*i*, for finite nonzero a
a + NaN*i*	NaN + NaN*i*, for finite nonzero a
+∞ + 0*i*	+∞ + 0*i*
+∞ + b*i*	+∞×cos(b) + Inf×sin(b).i for finite nonzero b
+∞ + ∞*i*	+∞ + NaN*i*
+∞ + NaN*i*	+∞ + NaN*i*
NaN + 0*i*	NaN + ±0*i*, sign of imaginary part unspecified
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- ccosh(conj(z)) = conj(ccosh(z)).

Thread safety

Safe.

4.2.3.7 ctanh()

Description

Compute hyperbolic tangent, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX ctanh(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic tangent of.

Return value

The hyperbolic tangent of x according to the following table:

Argument	ctanh(Argument)
+0 + 0*i*	+0 + 0*i*
a + ∞*i*	NaN + NaN*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
+∞ + b*i*	+1 + sin(2b)×0*i* for positive-signed finite b
+∞ + ∞*i*	+1 + ±0*i*, sign of imaginary part unspecified
+∞ + NaN*i*	+1 + ±0*i*, sign of imaginary part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- ctanh(conj(z)) = conj(ctanh(z)).

For arguments with a negative real component, use the equality:

- ctanh(-z) = -ctanh(z).

Thread safety

Safe.

4.2.3.8 ctanhf()

Description

Compute hyperbolic tangent, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX ctanhf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic tangent of.

Return value

The hyperbolic tangent of x according to the following table:

Argument	ctanh(Argument)
+0 + 0*i*	+0 + 0*i*
a + ∞*i*	NaN + NaN*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
+∞ + b*i*	+1 + sin(2b)×0*i* for positive-signed finite b
+∞ + ∞*i*	+1 + ±0*i*, sign of imaginary part unspecified
+∞ + NaN*i*	+1 + ±0*i*, sign of imaginary part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- ctanhf(conj(z)) = conj(ctanhf(z)).

For arguments with a negative real component, use the equality:

- ctanhf(-z) = -ctanhf(z).

Thread safety

Safe.

4.2.3.9 ctanhl()

Description

Compute hyperbolic tangent, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX ctanhl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic tangent of.

Return value

The hyperbolic tangent of x according to the following table:

Argument	ctanh(Argument)
+0 + 0*i*	+0 + 0*i*
a + ∞*i*	NaN + NaN*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
+∞ + b*i*	+1 + sin(2b)×0*i* for positive-signed finite b
+∞ + ∞*i*	+1 + ±0*i*, sign of imaginary part unspecified
+∞ + NaN*i*	+1 + ±0*i*, sign of imaginary part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for all nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- ctanh(conj(z)) = conj(ctanh(z)).

For arguments with a negative real component, use the equality:

- ctanh(-z) = -ctanh(z).

Thread safety

Safe.

4.2.3.10 casinh()

Description

Compute inverse hyperbolic sine, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX casinh(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic sineof.

Return value

The inverse hyperbolic sine of x according to the following table:

Argument	casinh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + ∞*i*	+∞ + ½π*i*
a + NaN*i*	NaN + NaN*i*
+∞ + b*i*	+∞ + 0*i*, for positive-signed b
+∞ + ∞*i*	+Pi + 0*i*
+∞ + NaN*i*	+∞ + NaN*i*
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for finite nonzero b
NaN + ∞*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- casinh(conj(z)) = conj(casinh(z)).

For arguments with a negative real component, use the equality:

- casinh(-z) = -casinh(z).

Thread safety

Safe.

4.2.3.11 casinhf()

Description

Compute inverse hyperbolic sine, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX casinhf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic sineof.

Return value

The inverse hyperbolic sine of x according to the following table:

Argument	casinh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + ∞*i*	+∞ + ½π*i*
a + NaN*i*	NaN + NaN*i*
+∞ + b*i*	+∞ + 0*i*, for positive-signed b
+∞ + ∞*i*	+Pi + 0*i*
+∞ + NaN*i*	+∞ + NaN*i*
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for finite nonzero b
NaN + ∞*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- casinh(conj(z)) = conj(casinh(z)).

For arguments with a negative real component, use the equality:

- casinh(-z) = -casinh(z).

Thread safety

Safe.

4.2.3.12 casinhl()

Description

Compute inverse hyperbolic sine, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX casinhl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic sineof.

Return value

The inverse hyperbolic sine of x according to the following table:

Argument	casinh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + ∞*i*	+∞ + ½π*i*
a + NaN*i*	NaN + NaN*i*
+∞ + b*i*	+∞ + 0*i*, for positive-signed b
+∞ + ∞*i*	+Pi + 0*i*
+∞ + NaN*i*	+∞ + NaN*i*
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for finite nonzero b
NaN + ∞*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- casinh(conj(z)) = conj(casinh(z)).

For arguments with a negative real component, use the equality:

- casinh(-z) = -casinh(z).

Thread safety

Safe.

4.2.3.13 cacosh()

Description

Compute inverse hyperbolic cosine, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX cacosh(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic cosine of.

Return value

The inverse hyperbolic cosine of x according to the following table:

Argument	cacosh(Argument)
±0 + 0*i*	+0 + 0*i*
a + ∞*i*	+∞ + ½π*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
-∞ + b*i*	+∞ + π*i*, for positive-signed finite b
+∞ + b*i*	+∞ + 0*i*, for positive-signed finite b
-∞ + ∞*i*	±∞ + ¾π*i*
+∞ + ∞*i*	±∞ + ¼π*i*
±∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + NaN*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- cacosh(conj(z)) = conj(cacosh(z)).

Thread safety

Safe.

4.2.3.14 cacoshf()

Description

Compute inverse hyperbolic cosine, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX cacoshf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic cosine of.

Return value

The inverse hyperbolic cosine of x according to the following table:

Argument	cacosh(Argument)
±0 + 0*i*	+0 + 0*i*
a + ∞*i*	+∞ + ½π*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
-∞ + b*i*	+∞ + π*i*, for positive-signed finite b
+∞ + b*i*	+∞ + 0*i*, for positive-signed finite b
-∞ + ∞*i*	±∞ + ¾π*i*
+∞ + ∞*i*	±∞ + ¼π*i*
±∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + NaN*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- cacosh(conj(z)) = conj(cacosh(z)).

Thread safety

Safe.

4.2.3.15 cacoshl()

Description

Compute inverse hyperbolic cosine, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX cacoshl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic cosine of.

Return value

The inverse hyperbolic cosine of x according to the following table:

Argument	cacosh(Argument)
±0 + 0*i*	+0 + 0*i*
a + ∞*i*	+∞ + ½π*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
-∞ + b*i*	+∞ + π*i*, for positive-signed finite b
+∞ + b*i*	+∞ + 0*i*, for positive-signed finite b
-∞ + ∞*i*	±∞ + ¾π*i*
+∞ + ∞*i*	±∞ + ¼π*i*
±∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + NaN*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- cacosh(conj(z)) = conj(cacosh(z)).

Thread safety

Safe.

4.2.3.16 catanh()

Description

Compute inverse hyperbolic tangent, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX catanh(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic tangent of.

Return value

The inverse hyperbolic tangent of x according to the following table:

Argument	catanh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + NaN*i*	+0 + NaN*i*
+1 + 0*i*	+∞ + 0*i*
a + ∞*i*	+0 + ½π*i* for positive-signed a
a + NaN*i*	NaN + NaN*i*, for nonzero finite a
+∞ + b*i*	+0 + ½π*i* for positive-signed b
+∞ + ∞*i*	+0 + ½π*i*
+∞ + NaN*i*	+0 + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- catanh(conj(z)) = conj(catanh(z)).

For arguments with a negative real component, use the equality:

- catanh(-z) = -catanh(z).

Thread safety

Safe.

4.2.3.17 catanhf()

Description

Compute inverse hyperbolic tangent, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX catanhf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic tangent of.

Return value

The inverse hyperbolic tangent of x according to the following table:

Argument	catanh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + NaN*i*	+0 + NaN*i*
+1 + 0*i*	+∞ + 0*i*
a + ∞*i*	+0 + ½π*i* for positive-signed a
a + NaN*i*	NaN + NaN*i*, for nonzero finite a
+∞ + b*i*	+0 + ½π*i* for positive-signed b
+∞ + ∞*i*	+0 + ½π*i*
+∞ + NaN*i*	+0 + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- catanh(conj(z)) = conj(catanh(z)).

For arguments with a negative real component, use the equality:

- catanh(-z) = -catanh(z).

Thread safety

Safe.

4.2.3.18 catanhl()

Description

Compute inverse hyperbolic tangent, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX catanhl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic tangent of.

Return value

The inverse hyperbolic tangent of x according to the following table:

Argument	catanh(Argument)
+0 + 0*i*	+0 + 0*i*
+0 + NaN*i*	+0 + NaN*i*
+1 + 0*i*	+∞ + 0*i*
a + ∞*i*	+0 + ½π*i* for positive-signed a
a + NaN*i*	NaN + NaN*i*, for nonzero finite a
+∞ + b*i*	+0 + ½π*i* for positive-signed b
+∞ + ∞*i*	+0 + ½π*i*
+∞ + NaN*i*	+0 + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- catanh(conj(z)) = conj(catanh(z)).

For arguments with a negative real component, use the equality:

- catanh(-z) = -catanh(z).

Thread safety

Safe.

4.2.4 Power and absolute value

Function	Description
`cabs()`	Compute magnitude, double complex.
`cabsf()`	Compute magnitude, float complex.
`cabsl()`	Compute magnitude, long double complex.
`cpow()`	Power, double complex.
`cpowf()`	Power, float complex.
`cpowl()`	Power, long double complex.
`csqrt()`	Square root, double complex.
`csqrtf()`	Square root, float complex.
`csqrtl()`	Square root, long double complex.

4.2.4.1 cabs()

Description

Compute magnitude, double complex.

Prototype

```
double cabs(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

The magnitude of x, |x|.

Thread safety

Safe.

4.2.4.2 cabsf()

Description

Compute magnitude, float complex.

Prototype

```
float cabsf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

The magnitude of x, |x|.

Thread safety

Safe.

4.2.4.3 cabsl()

Description

Compute magnitude, long double complex.

Prototype

```
long double cabsl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

The magnitude of x, |x|.

Thread safety

Safe.

4.2.4.4 cpow()

Description

Power, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX cpow(__SEGGER_RTL_FLOAT64_C_COMPLEX x,
                                    __SEGGER_RTL_FLOAT64_C_COMPLEX y);
```

Parameters

Parameter	Description
x	Base.
y	Power.

Return value

Return x raised to the power of y.

Thread safety

Safe.

4.2.4.5 cpowf()

Description

Power, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX cpowf(__SEGGER_RTL_FLOAT32_C_COMPLEX x,
                                     __SEGGER_RTL_FLOAT32_C_COMPLEX y);
```

Parameters

Parameter	Description
x	Base.
y	Power.

Return value

Return x raised to the power of y.

Thread safety

Safe.

4.2.4.6 cpowl()

Description

Power, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX cpowl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x,
                                     __SEGGER_RTL_LDOUBLE_C_COMPLEX y);
```

Parameters

Parameter	Description
x	Base.
y	Power.

Return value

Return x raised to the power of y.

Thread safety

Safe.

4.2.4.7 csqrt()

Description

Square root, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX csqrt(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute squate root of.

Return value

The square root of x according to the following table:

Argument	csqrt(Argument)
±0 + 0*i*	+0 + 0i
a + ∞*i*	+∞ + ∞*i*, for all a
a + NaN*i*	+NaN + NaN*i*, for finite a
-∞ + b*i*	+0 + ∞*i* for finite positive-signed b
+∞ + b*i*	+∞ + 0*i*, for finite positive-signed b
+∞ + ∞*i*	+∞ + ¼π*i*
-∞ + NaN*i*	+NaN + +/∞*i*, sign of imaginary part unspecified
+∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + ∞*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- csqrt(conj(z)) = conj(csqrt(z)).

Thread safety

Safe.

4.2.4.8 csqrtf()

Description

Square root, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX csqrtf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute squate root of.

Return value

The square root of x according to the following table:

Argument	csqrt(Argument)
±0 + 0*i*	+0 + 0i
a + ∞*i*	+∞ + ∞*i*, for all a
a + NaN*i*	+NaN + NaN*i*, for finite a
-∞ + b*i*	+0 + ∞*i* for finite positive-signed b
+∞ + b*i*	+∞ + 0*i*, for finite positive-signed b
+∞ + ∞*i*	+∞ + ¼π*i*
-∞ + NaN*i*	+NaN + +/∞*i*, sign of imaginary part unspecified
+∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + ∞*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- csqrt(conj(z)) = conj(csqrt(z)).

Thread safety

Safe.

4.2.4.9 csqrtl()

Description

Square root, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX csqrtl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute squate root of.

Return value

The square root of x according to the following table:

Argument	csqrt(Argument)
±0 + 0*i*	+0 + 0i
a + ∞*i*	+∞ + ∞*i*, for all a
a + NaN*i*	+NaN + NaN*i*, for finite a
-∞ + b*i*	+0 + ∞*i* for finite positive-signed b
+∞ + b*i*	+∞ + 0*i*, for finite positive-signed b
+∞ + ∞*i*	+∞ + ¼π*i*
-∞ + NaN*i*	+NaN + +/∞*i*, sign of imaginary part unspecified
+∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + ∞*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- csqrt(conj(z)) = conj(csqrt(z)).

Thread safety

Safe.

4.2.5 Exponential and logarithm functions

Function	Description
`clog()`	Compute natural logarithm, double complex.
`clogf()`	Compute natural logarithm, float complex.
`clogl()`	Compute natural logarithm, long double complex.
`cexp()`	Compute base-e exponential, double complex.
`cexpf()`	Compute base-e exponential, float complex.
`cexpl()`	Compute base-e exponential, long double complex.

4.2.5.1 clog()

Description

Compute natural logarithm, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX clog(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

The natural logarithm of x according to the following table:

Argument	clog(Argument)
-0 + 0*i*	-∞ + π*i*
+0 + 0*i*	-∞ + 0*i*
a + ∞*i*	+∞ + ½π*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
-∞ + b*i*	+∞ + π*i*, for finite positive b
+∞ + b*i*	+∞ + 0*i*, for finite positive b
-∞ + ∞*i*	+∞ + ¾π*i*
+∞ + ∞*i*	+∞ + ¼π*i*
±∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + NaN*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- clog(conj(z)) = conj(clog(z)).

Thread safety

Safe.

4.2.5.2 clogf()

Description

Compute natural logarithm, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX clogf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

The natural logarithm of x according to the following table:

Argument	clog(Argument)
-0 + 0*i*	-∞ + π*i*
+0 + 0*i*	-∞ + 0*i*
a + ∞*i*	+∞ + ½π*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
-∞ + b*i*	+∞ + π*i*, for finite positive b
+∞ + b*i*	+∞ + 0*i*, for finite positive b
-∞ + ∞*i*	+∞ + ¾π*i*
+∞ + ∞*i*	+∞ + ¼π*i*
±∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + NaN*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- clog(conj(z)) = conj(clog(z)).

Thread safety

Safe.

4.2.5.3 clogl()

Description

Compute natural logarithm, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX clogl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

The natural logarithm of x according to the following table:

Argument	clog(Argument)
-0 + 0*i*	-∞ + π*i*
+0 + 0*i*	-∞ + 0*i*
a + ∞*i*	+∞ + ½π*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
-∞ + b*i*	+∞ + π*i*, for finite positive b
+∞ + b*i*	+∞ + 0*i*, for finite positive b
-∞ + ∞*i*	+∞ + ¾π*i*
+∞ + ∞*i*	+∞ + ¼π*i*
±∞ + NaN*i*	+∞ + NaN*i*
NaN + b*i*	NaN + NaN*i*, for finite b
NaN + ∞*i*	+∞ + NaN*i*
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality:

- clog(conj(z)) = conj(clog(z)).

Thread safety

Safe.

4.2.5.4 cexp()

Description

Compute base-e exponential, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX cexp(__SEGGER_RTL_FLOAT64_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute exponential of.

Return value

The base-e exponential of x=a+b*i* according to the following table:

Argument	cexp(Argument)
-/-0 + 0*i*	+1 + 0*i*
a + ∞*i*	NaN + NaN*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
+∞ + 0*i*	+∞ + 0*i*, for finite positive b
-∞ + b*i*	+0 cis(b) for finite b
+∞ + b*i*	+∞ cis(b) for finite nonzero b
-∞ + ∞*i*	±∞ + ±0*i*, signs unspecified
+∞ + ∞*i*	±∞ + i.NaN, sign of real part unspecified
-∞ + NaN*i*	±0 + ±0*i*, signs unspecified
+∞ + NaN*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality

- cexp(conj(x)) = conj(cexp(x)).

Thread safety

Safe.

4.2.5.5 cexpf()

Description

Compute base-e exponential, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX cexpf(__SEGGER_RTL_FLOAT32_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute exponential of.

Return value

The base-e exponential of x=a+b*i* according to the following table:

Argument	cexp(Argument)
-/-0 + 0*i*	+1 + 0*i*
a + ∞*i*	NaN + NaN*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
+∞ + 0*i*	+∞ + 0*i*, for finite positive b
-∞ + b*i*	+0 cis(b) for finite b
+∞ + b*i*	+∞ cis(b) for finite nonzero b
-∞ + ∞*i*	±∞ + ±0*i*, signs unspecified
+∞ + ∞*i*	±∞ + i.NaN, sign of real part unspecified
-∞ + NaN*i*	±0 + ±0*i*, signs unspecified
+∞ + NaN*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality

- cexp(conj(x)) = conj(cexp(x)).

Thread safety

Safe.

4.2.5.6 cexpl()

Description

Compute base-e exponential, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX cexpl(__SEGGER_RTL_LDOUBLE_C_COMPLEX x);
```

Parameters

Parameter	Description
x	Value to compute exponential of.

Return value

The base-e exponential of x=a+b*i* according to the following table:

Argument	cexp(Argument)
-/-0 + 0*i*	+1 + 0*i*
a + ∞*i*	NaN + NaN*i*, for finite a
a + NaN*i*	NaN + NaN*i*, for finite a
+∞ + 0*i*	+∞ + 0*i*, for finite positive b
-∞ + b*i*	+0 cis(b) for finite b
+∞ + b*i*	+∞ cis(b) for finite nonzero b
-∞ + ∞*i*	±∞ + ±0*i*, signs unspecified
+∞ + ∞*i*	±∞ + i.NaN, sign of real part unspecified
-∞ + NaN*i*	±0 + ±0*i*, signs unspecified
+∞ + NaN*i*	±∞ + NaN*i*, sign of real part unspecified
NaN + 0*i*	NaN + 0*i*
NaN + b*i*	NaN + NaN*i*, for nonzero b
NaN + NaN*i*	NaN + NaN*i*

For arguments with a negative imaginary component, use the equality

- cexp(conj(x)) = conj(cexp(x)).

Thread safety

Safe.

4.3 <ctype.h>

4.3.1 Classification functions

Function	Description
isascii()	Is character a 7-bit ASCII code?
isascii_l()	Is character a 7-bit ASCII code, per locale (POSIX.
iscntrl()	Is character a control?
iscntrl_l()	Is character a control, per locale? (POSIX.1).
isblank()	Is character a blank?
isblank_l()	Is character a blank, per locale? (POSIX.1).
isspace()	Is character a whitespace character?
isspace_l()	Is character a whitespace character, per locale? (POSIX.1).
ispunct()	Is character a punctuation mark?
ispunct_l()	Is character a punctuation mark, per locale? (POSIX.1).
isdigit()	Is character a decimal digit?
isdigit_l()	Is character a decimal digit, per locale? (POSIX.
isxdigit()	Is character a hexadecimal digit?
isxdigit_l()	Is character a hexadecimal digit, per locale? (POSIX.1).
isalpha()	Is character alphabetic?
isalpha_l()	Is character alphabetic, per locale? (POSIX.1).
isalnum()	Is character alphanumeric?
isalnum_l()	Is character alphanumeric, per locale? (POSIX.1).
isupper()	Is character an uppercase letter?
isupper_l()	Is character an uppercase letter, per locale? (POSIX.1).
islower()	Is character a lowercase letter?
islower_l()	Is character a lowercase letter, per locale? (POSIX.1).
isprint()	Is character printable?
isprint_l()	Is character printable, per locale? (POSIX.1).
isgraph()	Is character any printing character?
isgraph_l()	Is character any printing character, per locale? (POSIX.1).

4.3.1.1 isascii()

Description

Is character a 7-bit ASCII code?

Prototype

```
int isascii(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument has an ASCII code between 0 and 127 in the current locale.

Thread safety

Safe.

4.3.1.2 isascii_l()

Description

Is character a 7-bit ASCII code, per locale (POSIX.1)?

Prototype

```
int isascii_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument has an ASCII code between 0 and 127 in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.3 iscntrl()

Description

Is character a control?

Prototype

```
int iscntrl(int c);
```

Parameters

Parameter	Description
`c`	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument `c` is a control character in the current locale.

Thread safety

Safe [if configured].

4.3.1.4 iscntrl_l()

Description

Is character a control, per locale? (POSIX.1).

Prototype

```
int iscntrl_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a control character in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.5 isblank()

Description

Is character a blank?

Prototype

```
int isblank(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is either a space character or tab character in the current locale.

Thread safety

Safe [if configured].

4.3.1.6 isblank_l()

Description

Is character a blank, per locale? (POSIX.1).

Prototype

```
int isblank_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is either a space character or the tab character in locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.7 isspace()

Description

Is character a whitespace character?

Prototype

```
int isspace(int c);
```

Parameters

Parameter	Description
`c`	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument `c` is a standard white-space character in the current locale. The standard white-space characters are space, form feed, new-line, carriage return, horizontal tab, and vertical tab.

Thread safety

Safe [if configured].

4.3.1.8 isspace_l()

Description

Is character a whitespace character, per locale? (POSIX.1).

Prototype

```
int isspace_l(int c,
                locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a standard white-space character in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.9 ispunct()

Description

Is character a punctuation mark?

Prototype

```
int ispunct(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) for every printing character for which neither `isspace()` nor `isalnum()` is true in the current locale.

Thread safety

Safe [if configured].

4.3.1.10 ispunct_l()

Description

Is character a punctuation mark, per locale? (POSIX.1).

Prototype

```
int ispunct_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) for every printing character for which neither `isspace()` nor `isalnum()` is true in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.11 isdigit()

Description

Is character a decimal digit?

Prototype

```
int isdigit(int c);
```

Parameters

Parameter	Description
`c`	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument `c` is a digit in the current locale.

Thread safety

Safe [if configured].

4.3.1.12 isdigit_l()

Description

Is character a decimal digit, per locale? (POSIX.1)

Prototype

```
int isdigit_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a digit in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.13 isxdigit()

Description

Is character a hexadecimal digit?

Prototype

```
int isxdigit(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is a hexadecimal digit in the current locale.

Thread safety

Safe [if configured].

4.3.1.14 isxdigit_l()

Description

Is character a hexadecimal digit, per locale? (POSIX.1).

Prototype

```
int isxdigit_l(int c,
                   locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a hexadecimal digit in the current locale.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.15 isalpha()

Description

Is character alphabetic?

Prototype

```
int isalpha(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns true if the character c is alphabetic in the current locale. That is, any character for which `isupper()` or `islower()` returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, or `isspace()` is true.

In the C locale, `isalpha()` returns nonzero (true) if and only if `isupper()` or `islower()` return true for value of the argument c.

Thread safety

Safe [if configured].

4.3.1.16 isalpha_l()

Description

Is character alphabetic, per locale? (POSIX.1).

Prototype

```
int isalpha_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns true if the character c is alphabetic in the locale loc. That is, any character for which isupper() or islower() returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of iscntrl_l(), isdigit_l(), ispunct_l(), or isspace_l() is true in the locale loc.

In the C locale, isalpha_l() returns nonzero (true) if and only if isupper_l() or islower_l() return true for value of the argument c.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.17 isalnum()

Description

Is character alphanumeric?

Prototype

```
int isalnum(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is an alphabetic or numeric character in the current locale.

Thread safety

Safe [if configured].

4.3.1.18 isalnum_l()

Description

Is character alphanumeric, per locale? (POSIX.1).

Prototype

```
int isalnum_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is an alphabetic or numeric character in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.19 isupper()

Description

Is character an uppercase letter?

Prototype

```
int isupper(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is an uppercase letter in the current locale.

Thread safety

Safe [if configured].

4.3.1.20 isupper_l()

Description

Is character an uppercase letter, per locale? (POSIX.1).

Prototype

```
int isupper_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is an uppercase letter in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.21 islower()

Description

Is character a lowercase letter?

Prototype

```
int islower(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is a lowercase letter in the current locale.

Thread safety

Safe [if configured].

4.3.1.22 islower_l()

Description

Is character a lowercase letter, per locale? (POSIX.1).

Prototype

```
int islower_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a lowercase letter in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.23 isprint()

Description

Is character printable?

Prototype

```
int isprint(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is any printing character including space in the current locale.

Thread safety

Safe [if configured].

4.3.1.24 isprint_l()

Description

Is character printable, per locale? (POSIX.1).

Prototype

```
int isprint_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is any printing character including space in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.1.25 isgraph()

Description

Is character any printing character?

Prototype

```
int isgraph(int c);
```

Parameters

Parameter	Description
c	Character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is any printing character except space in the current locale.

Thread safety

Safe [if configured].

4.3.1.26 isgraph_l()

Description

Is character any printing character, per locale? (POSIX.1).

Prototype

```
int isgraph_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is any printing character except space in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.2 Conversion functions

Function	Description
`toupper()`	Convert lowercase character to uppercase.
`toupper_l()`	Convert lowercase character to uppercase, per locale (POSIX.1).
`tolower()`	Convert uppercase character to lowercase.
`tolower_l()`	Convert uppercase character to lowercase, per locale (POSIX.1).

4.3.2.1 toupper()

Description

Convert lowercase character to uppercase.

Prototype

```
int toupper(int c);
```

Parameters

Parameter	Description
`c`	Character to convert.

Return value

Converted character.

Additional information

Converts a lowercase letter to a corresponding uppercase letter.

If the argument `c` is a character for which `islower()` is true and there are one or more corresponding characters, as specified by the current locale, for which `isupper()` is true, `toupper()` returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

Notes

Even though `islower()` can return true for some characters, `toupper()` may return that lowercase character unchanged as there are no corresponding uppercase characters in the locale.

Thread safety

Safe [if configured].

4.3.2.2 toupper_l()

Description

Convert lowercase character to uppercase, per locale (POSIX.1).

Prototype

```
int toupper_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to convert.
loc	Locale used to convert c.

Return value

Converted character.

Additional information

Converts a lowercase letter to a corresponding uppercase letter in locale loc. If the argument c is a character for which islower_l() is true in locale loc, tolower_l() returns the corresponding uppercase letter; otherwise, the argument is returned unchanged.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.3.2.3 tolower()

Description

Convert uppercase character to lowercase.

Prototype

```
int tolower(int c);
```

Parameters

Parameter	Description
`c`	Character to convert.

Return value

Converted character.

Additional information

Converts an uppercase letter to a corresponding lowercase letter.

If the argument `c` is a character for which `isupper()` is true and there are one or more corresponding characters, as specified by the current locale, for which `islower()` is true, the `tolower()` function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

Notes

Even though `isupper()` can return true for some characters, `tolower()` may return that uppercase character unchanged as there are no corresponding lowercase characters in the locale.

Thread safety

Safe [if configured].

4.3.2.4 tolower_l()

Description

Convert uppercase character to lowercase, per locale (POSIX.1).

Prototype

```
int tolower_l(int c,
              locale_t loc);
```

Parameters

Parameter	Description
c	Character to convert.
loc	Locale used to convert c.

Return value

Converted character.

Additional information

Converts an uppercase letter to a corresponding lowercase letter in locale loc. If the argument is a character for which isupper_l() is true in locale loc, tolower_l() returns the corresponding lowercase letter; otherwise, the argument is returned unchanged.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.4 <errno.h>

4.4.1 Errors

Object	Description
`errno`	Macro returning the current error.
`errno_t`	Type describing errors (C11).

4.4.1.1 Error names

Description

Symbolic error names for raised errors.

Definition

```
#define EHEAP     0x04
#define ENOMEM    0x05
#define EINVAL    0x06
#define ESPIPE    0x07
#define EAGAIN    0x08
#define ECHILD    0x09
#define EMLINK    0x0A
#define ENOENT    0x0B
#define EDOM      (__aeabi_EDOM)
#define EILSEQ    (__aeabi_EILSEQ)
#define ERANGE    (__aeabi_ERANGE)
```

Symbols

Definition	Description
EDOM	Internal use.
EILSEQ	Internal use.
ERANGE	Internal use.
EHEAP	Heap is corrupt (emRun)
ENOMEM	Out of memory (POSIX.1-2001)
EINVAL	Invalid parameter (POSIX.1-2001)
ESPIPE	Invalid seek (POSIX.1-2001)
EAGAIN	Resource unavailable, try again (POSIX.1-2001)
ECHILD	No child processes (POSIX.1-2001)
EMLINK	Too many links (POSIX.1-2001)
ENOENT	No such file or directory (POSIX.1-2001) Modify for AEABI compliance
EDOM	Internal use.
EILSEQ	Internal use.
ERANGE	Internal use.

4.4.1.2 errno

Description

Macro returning the current error.

Definition

```
#define errno    (*__SEGGER_RTL_X_errno_addr())
```

Additional information

The value in `errno` is significant only when the return value of the call indicated an error. A function that succeeds is allowed to change `errno`. The value of `errno` is never set to zero by a library function.

4.4.1.3 errno_t

Description

Type describing errors (C11).

Type definition

```
typedef int errno_t;
```

Additional information

The macro `__STDC_WANT_LIB_EXT1__` must be set to 1 before including <errno.h> to access this type.

This type is used by the C11/C18 bounds-checking functions.

Conformance

ISO/IEC 9899:2011 (C11).

4.5 <fenv.h>

4.5.1 Floating-point exceptions

Function	Description
`feclearexcept()`	Clear floating-point exceptions.
`feraiseexcept()`	Raise floating-point exceptions.
`fegetexceptflag()`	Get floating-point exceptions.
`fesetexceptflag()`	Set floating-point exceptions.
`fetestexcept()`	Test floating-point exceptions.

4.5.1.1 feclearexcept()

Description

Clear floating-point exceptions.

Prototype

```
int feclearexcept(int excepts);
```

Parameters

Parameter	Description
excepts	Bitmask of floating-point exceptions to clear.

Return value

= 0	Floating-point exceptions successfully cleared.
≠ 0	Floating-point exceptions not cleared or not supported.

Additional information

This function attempts to clear the floating-point exceptions indicated by excepts.

Notes

This function has no return value in ISO C (1999) and an integer return value in ISO C (2008).

Thread safety

Safe [if configured].

4.5.1.2 feraiseexcept()

Description

Raise floating-point exceptions.

Prototype

```
int feraiseexcept(int excepts);
```

Parameters

Parameter	Description
`excepts`	Bitmask of floating-point exceptions to raise.

Return value

= 0	All floating-point exceptions successfully raised.
≠ 0	Floating-point exceptions not successuly raised or not supported.

Additional information

This function attempts to raise the floating-point exceptions indicated by excepts.

Notes

This function has no return value in ISO C (1999) and an integer return value in ISO C (2008).

Thread safety

Safe [if configured].

4.5.1.3 fegetexceptflag()

Description

Get floating-point exceptions.

Prototype

```
int fegetexceptflag(fexcept_t * flagp,
                    int         excepts);
```

Parameters

Parameter	Description
`flagp`	Pointer to object that receives the floating-point exception state.
`excepts`	Bitmask of floating-point exceptions to store.

Return value

= 0	Floating-point exceptions correctly stored.
≠ 0	Floating-point exceptions not correctly stored.

Additional information

This function attempts to save the floating-point exceptions indicated by `excepts` to the object pointed to by flagp.

Thread safety

Safe [if configured].

See also

`fesetexceptflag()`.

4.5.1.4 fesetexceptflag()

Description

Set floating-point exceptions.

Prototype

```
int fesetexceptflag(const fexcept_t * flagp,
                          int         excepts);
```

Parameters

Parameter	Description
`flagp`	Pointer to object containing a previously-stored floating-point exception state.
`excepts`	Bitmask of floating-point exceptions to restore.

Return value

= 0	Floating-point exceptions correctly restored.
≠ 0	Floating-point exceptions not correctly restored.

Additional information

This function attempts to restore the floating-point exceptions indicated by `excepts` from the object pointed to by `flagp`. The exceptions to restore as indicated by `excepts` must have at least been specified when storing the exceptions using `fegetexceptflag()`.

Thread safety

Safe [if configured].

See also

`fegetexceptflag()`.

4.5.1.5 fetestexcept()

Description

Test floating-point exceptions.

Prototype

```
int fetestexcept(int excepts);
```

Parameters

Parameter	Description
`excepts`	Bitmask of floating-point exceptions to test.

Return value

The bitmask of all floating-point exceptions that are currently set and are specified in excepts.

Additional information

This function determines which of the floating-point exceptions indicated by `excepts` are currently set.

Thread safety

Safe [if configured].

4.5.2 Floating-point rounding mode

Function	Description
`fegetround()`	Get floating-point rounding mode.
`fesetround()`	Set floating-point rounding mode.

4.5.2.1 fegetround()

Description

Get floating-point rounding mode.

Prototype

```
int fegetround(void);
```

Return value

≥ 0	Current floating-point rounding mode.
< 0	Floating-point rounding mode cannot be determined.

Additional information

This function attempts to read the current floating-point rounding mode.

Thread safety

Safe [if configured].

See also

`fesetround()`.

4.5.2.2 fesetround()

Description

Set floating-point rounding mode.

Prototype

```
int fesetround(int round);
```

Parameters

Parameter	Description
`round`	Rounding mode to set.

Return value

= 0	Current floating-point rounding mode is set to `round`.
≠ 0	Requested floating-point rounding mode cannot be set.

Additional information

This function attempts to set the current floating-point rounding mode to round.

Thread safety

Safe [if configured].

See also

`fegetround()`.

4.5.3 Floating-point environment

Function	Description
`fegetenv()`	Get floating-point environment.
`fesetenv()`	Set floating-point environment.
`feupdateenv()`	Restore floating-point environment and reraise exceptions.
`feholdexcept()`	Save floating-point environment and set non-stop mode.

4.5.3.1 fegetenv()

Description

Get floating-point environment.

Prototype

```
int fegetenv(fenv_t * envp);
```

Parameters

Parameter	Description
envp	Pointer to object that receives the floating-point environment.

Return value

= 0	Current floating-point environment successfully stored.
≠ 0	Floating-point environment cannot be stored.

Additional information

This function attempts to store the current floating-point environment to the object pointed to by envp.

Notes

This function has no return value in ISO C (1999) and an integer return value in ISO C (2008).

Thread safety

Safe [if configured].

See also

`fesetenv()`.

4.5.3.2 fesetenv()

Description

Set floating-point environment.

Prototype

```
int fesetenv(const fenv_t * envp);
```

Parameters

Parameter	Description
`envp`	Pointer to object containing previously-stored floating-point environment.

Return value

= 0	Current floating-point environment successfully restored.
≠ 0	Floating-point environment cannot be restored.

Additional information

This function attempts to restore the floating-point environment from the object pointed to by envp.

Notes

This function has no return value in ISO C (1999) and an integer return value in ISO C (2008).

Thread safety

Safe [if configured].

See also

`fegetenv()`.

4.5.3.3 feupdateenv()

Description

Restore floating-point environment and reraise exceptions.

Prototype

```
int feupdateenv(const fenv_t * envp);
```

Parameters

Parameter	Description
envp	Pointer to object containing previously-stored floating-point environment.

Return value

= 0	Environment restored and exceptions raised successfully.
≠ 0	Failed to restore environment and raise exceptions.

Additional information

This function attempts to save the currently raised floating-point exceptions, restore the floating-point environment from the object pointed to by envp, and raise the saved exceptions.

Notes

This function has no return value in ISO C (1999) and an integer return value in ISO C (2008).

Thread safety

Safe [if configured].

4.5.3.4 feholdexcept()

Description

Save floating-point environment and set non-stop mode.

Prototype

```
int feholdexcept(fenv_t * envp);
```

Parameters

Parameter	Description
envp	Pointer to object that receives the floating-point environment.

Return value

= 0	Environment stored and non-stop mode set successfully.
≠ 0	Failed to store environment or set non-stop mode.

Additional information

This function function saves the current floating-point environment to the object pointed to by envp, clears the floating-point status flags, and then installs a non-stop mode for all floating-point exceptions

Thread safety

Safe [if configured].

4.6 <float.h>

4.6.1 Floating-point constants

4.6.1.1 Common parameters

Description

Applies to single-precision and double-precision formats.

Definition

```
#define FLT_ROUNDS       1
#define FLT_EVAL_METHOD  0
#define FLT_RADIX        2
#define DECIMAL_DIG      17
```

Symbols

Definition	Description
FLT_ROUNDS	Rounding mode of floating-point addition is round to nearest.
FLT_EVAL_METHOD	All operations and constants are evaluated to the range and precision of the type.
FLT_RADIX	Radix of the exponent representation.
DECIMAL_DIG	Number of decimal digits that can be rounded to a floating-point number without change to the value.

4.6.1.2 Float parameters

Description

IEEE 32-bit single-precision floating format parameters.

Definition

```
#define FLT_MANT_DIG      24
#define FLT_EPSILON       1.19209290E-07f
#define FLT_DIG           6
#define FLT_MIN_EXP       -125
#define FLT_MIN           1.17549435E-38f
#define FLT_MIN_10_EXP    -37
#define FLT_MAX_EXP       +128
#define FLT_MAX           3.40282347E+38f
#define FLT_MAX_10_EXP    +38
```

Symbols

Definition	Description
FLT_MANT_DIG	Number of base FLT_RADIX digits in the mantissa part of a float.
FLT_EPSILON	Minimum positive number such that 1.0f + FLT_EPSILON ≠ 1.0f.
FLT_DIG	Number of decimal digits of precision of a float.
FLT_MIN_EXP	Minimum value of base FLT_RADIX in the exponent part of a float.
FLT_MIN	Minimum value of a float.
FLT_MIN_10_EXP	Minimum value in base 10 of the exponent part of a float.
FLT_MAX_EXP	Maximum value of base FLT_RADIX in the exponent part of a float.
FLT_MAX	Maximum value of a float.
FLT_MAX_10_EXP	Maximum value in base 10 of the exponent part of a float.

4.6.1.3 Double parameters

Description

IEEE 64-bit double-precision floating format parameters.

Definition

```
#define DBL_MANT_DIG       53
#define DBL_EPSILON        2.2204460492503131E-16
#define DBL_DIG            15
#define DBL_MIN_EXP        -1021
#define DBL_MIN            2.2250738585072014E-308
#define DBL_MIN_10_EXP     -307
#define DBL_MAX_EXP        +1024
#define DBL_MAX            1.7976931348623157E+308
#define DBL_MAX_10_EXP     +308
```

Symbols

Definition	Description
DBL_MANT_DIG	Number of base DBL_RADIX digits in the mantissa part of a double.
DBL_EPSILON	Minimum positive number such that 1.0 + DBL_EPSILON ≠ 1.0.
DBL_DIG	Number of decimal digits of precision of a double.
DBL_MIN_EXP	Minimum value of base DBL_RADIX in the exponent part of a double.
DBL_MIN	Minimum value of a double.
DBL_MIN_10_EXP	Minimum value in base 10 of the exponent part of a double.
DBL_MAX_EXP	Maximum value of base DBL_RADIX in the exponent part of a double.
DBL_MAX	Maximum value of a double.
DBL_MAX_10_EXP	Maximum value in base 10 of the exponent part of a double.

4.7 <iso646.h>

The header `<iso646.h>` defines macros that expand to the corresponding tokens to ease writing C programs with keyboards that do not have keys for frequently-used operators.

4.7.1 Macros

4.7.1.1 Replacement macros

Description

Standard replacement macros.

Definition

```
#define and       &&
#define and_eq    &=
#define bitand    &
#define bitor     |
#define compl     ~
#define not       !
#define not_eq    !=
#define or        ||
#define or_eq     |=
#define xor       ^
#define xor_eq    ^=
```

4.8 <limits.h>

4.8.1 Minima and maxima

4.8.1.1 Character minima and maxima

Description

Minimum and maximum values for character types.

Definition

```
#define CHAR_BIT      8
#define CHAR_MIN      0
#define CHAR_MAX      255
#define SCHAR_MAX     127
#define SCHAR_MIN     (-128)
#define UCHAR_MAX     255
```

Symbols

Definition	Description
CHAR_BIT	Number of bits for smallest object that is not a bit-field (byte).
CHAR_MIN	Minimum value of a plain character.
CHAR_MAX	Maximum value of a plain character.
SCHAR_MAX	Maximum value of a signed character.
SCHAR_MIN	Minimum value of a signed character.
UCHAR_MAX	Maximum value of an unsigned character.

4.8.1.2 Short integer minima and maxima

Description

Minimum and maximum values for short integer types.

Definition

```
#define SHRT_MIN     (-32767 - 1)
#define SHRT_MAX     32767
#define USHRT_MAX    65535
```

Symbols

Definition	Description
SHRT_MIN	Minimum value of a short integer.
SHRT_MAX	Maximum value of a short integer.
USHRT_MAX	Maximum value of an unsigned short integer.

4.8.1.3 Integer minima and maxima

Description

Minimum and maximum values for integer types.

Definition

```
#define INT_MIN     (-2147483647 - 1)
#define INT_MAX     2147483647
#define UINT_MAX    4294967295u
```

Symbols

Definition	Description
INT_MIN	Minimum value of an integer.
INT_MAX	Maximum value of an integer.
UINT_MAX	Maximum value of an unsigned integer.

4.8.1.4 Long integer minima and maxima (32-bit)

Description

Minimum and maximum values for long integer types.

Definition

```
#define LONG_MIN     (-2147483647L - 1)
#define LONG_MAX     2147483647L
#define ULONG_MAX    4294967295uL
```

Symbols

Definition	Description
LONG_MIN	Maximum value of a long integer.
LONG_MAX	Minimum value of a long integer.
ULONG_MAX	Maximum value of an unsigned long integer.

4.8.1.5 Long integer minima and maxima (64-bit)

Description

Minimum and maximum values for long integer types.

Definition

```
#define LONG_MIN     (-9223372036854775807L - 1)
#define LONG_MAX     9223372036854775807L
#define ULONG_MAX    18446744073709551615uL
```

Symbols

Definition	Description
LONG_MIN	Minimum value of a long integer.
LONG_MAX	Maximum value of a long integer.
ULONG_MAX	Maximum value of an unsigned long integer.

4.8.1.6 Long long integer minima and maxima

Description

Minimum and maximum values for long integer types.

Definition

```
#define LLONG_MIN     (-9223372036854775807LL - 1)
#define LLONG_MAX     9223372036854775807LL
#define ULLONG_MAX    18446744073709551615uLL
```

Symbols

Definition	Description
LLONG_MIN	Minimum value of a long long integer.
LLONG_MAX	Maximum value of a long long integer.
ULLONG_MAX	Maximum value of an unsigned long long integer.

4.8.1.7 Multibyte characters

Description

Maximum number of bytes in a multi-byte character.

Definition

```
#define MB_LEN_MAX     4
```

Symbols

Definition	Description
`MB_LEN_MAX`	Maximum

Additional information

The maximum number of bytes in a multi-byte character for any supported locale. Unicode (ISO 10646) characters between `0x000000` and `0x10FFFF` inclusive are supported which convert to a maximum of four bytes in the UTF-8 encoding.

4.9 <locale.h>

4.9.1 Data types

4.9.1.1 __SEGGER_RTL_lconv

Type definition

```
typedef struct {
  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   frac_digits;
  char   p_cs_precedes;
  char   p_sep_by_space;
  char   n_cs_precedes;
  char   n_sep_by_space;
  char   p_sign_posn;
  char   n_sign_posn;
  char   int_p_cs_precedes;
  char   int_n_cs_precedes;
  char   int_p_sep_by_space;
  char   int_n_sep_by_space;
  char   int_p_sign_posn;
  char   int_n_sign_posn;
} __SEGGER_RTL_lconv;
```

Structure members

Member	Description
decimal_point	Decimal point separator.
thousands_sep	Separators used to delimit groups of digits to the left of the decimal point for non-monetary quantities.
grouping	Specifies the amount of digits that form each of the groups to be separated by thousands_sep separator for non-monetary quantities.
int_curr_symbol	International currency symbol.
currency_symbol	Local currency symbol.
mon_decimal_point	Decimal-point separator used for monetary quantities.
mon_thousands_sep	Separators used to delimit groups of digits to the left of the decimal point for monetary quantities.
mon_grouping	Specifies the amount of digits that form each of the groups to be separated by mon_thousands_sep separator for monetary quantities.
positive_sign	Sign to be used for nonnegative (positive or zero) monetary quantities.
negative_sign	Sign to be used for negative monetary quantities.
int_frac_digits	Amount of fractional digits to the right of the decimal point for monetary quantities in the international format.

Member	Description
frac_digits	Amount of fractional digits to the right of the decimal point for monetary quantities in the local format.
p_cs_precedes	Whether the currency symbol should precede nonnegative (positive or zero) monetary quantities.
p_sep_by_space	Whether a space should appear between the currency symbol and nonnegative (positive or zero) monetary quantities.
n_cs_precedes	Whether the currency symbol should precede negative monetary quantities.
n_sep_by_space	Whether a space should appear between the currency symbol and negative monetary quantities.
p_sign_posn	Position of the sign for nonnegative (positive or zero) monetary quantities.
n_sign_posn	Position of the sign for negative monetary quantities.
int_p_cs_precedes	Whether int_curr_symbol precedes or succeeds the value for a nonnegative internationally formatted monetary quantity.
int_n_cs_precedes	Whether int_curr_symbol precedes or succeeds the value for a negative internationally formatted monetary quantity.
int_p_sep_by_space	Value indicating the separation of the int_curr_symbol, the sign string, and the value for a nonnegative internationally formatted monetary quantity.
int_n_sep_by_space	Value indicating the separation of the int_curr_symbol, the sign string, and the value for a negative internationally formatted monetary quantity.
int_p_sign_posn	Value indicating the positioning of the positive_sign for a nonnegative internationally formatted monetary quantity.
int_n_sign_posn	Value indicating the positioning of the positive_sign for a negative internationally formatted monetary quantity.

4.9.2 Locale management

Function	Description
`setlocale()`	Set locale.
`localeconv()`	Get current locale data.

4.9.2.1 setlocale()

Description

Set locale.

Prototype

```
char *setlocale(      int    category,
                const char * loc);
```

Parameters

Parameter	Description
category	Category of locale to set, see below.
loc	Pointer to name of locale to set or, if NULL, the current locale.

Return value

Returns the name of the current locale if a locale name buffer has been set using `__SEGGER_RTL_set_locale_name_buffer()`, else returns NULL.

Additional information

For ISO-correct operation, a local name buffer needs to be set using `__SEGGER_RTL_set_locale_name_buffer()` when the name of the current or global locale can be encoded. In many cases the previous locale's name is not required, yet would take static storage on a global or per-thread basis. In order to avoid this, the standard operation of `setlocale()` in this library is to return `NULL` and not require any static data. If the previous locale's name is required, at runtime startup or before calling `setlocale()`, use `__SEGGER_RTL_set_locale_name_buffer()` to set the address of the object to use where the locale name can be encoded. To make this thread-safe, the object where the locale name is stored must be local to the thread.

The `category` parameter can have the following values:

Value	Description
LC_ALL	Entire locale.
LC_COLLATE	Affects `strcoll()` and `strxfrm()`.
LC_CTYPE	Affects character handling.
LC_MONETARY	Affects monetary formatting information.
LC_NUMERIC	Affects decimal-point character in I/O and string formatting operations.
LC_TIME	Affects `strftime()`.

Thread safety

Safe [if configured].

4.9.2.2 localeconv()

Description

Get current locale data.

Prototype

```
localeconv(void);
```

Return value

Returns a pointer to a structure of type lconv with the corresponding values for the current locale filled in.

Thread safety

Safe [if configured].

4.10 <math.h>

4.10.1 Exponential and logarithm functions

Function	Description
sqrt()	Compute square root, double.
sqrtf()	Compute square root, float.
sqrtl()	Compute square root, long double.
cbrt()	Compute cube root, double.
cbrtf()	Compute cube root, float.
cbrtl()	Compute cube root, long double.
rsqrt()	Compute reciprocal square root, double.
rsqrtf()	Compute reciprocal square root, float.
rsqrtl()	Compute reciprocal square root, long double.
exp()	Compute base-e exponential, double.
expf()	Compute base-e exponential, float.
expl()	Compute base-e exponential, long double.
expm1()	Compute base-e exponential, modified, double.
expm1f()	Compute base-e exponential, modified, float.
expm1l()	Compute base-e exponential, modified, long double.
exp2()	Compute base-2 exponential, double.
exp2f()	Compute base-2 exponential, float.
exp2l()	Compute base-2 exponential, long double.
exp10()	Compute base-10 exponential, double.
exp10f()	Compute base-10 exponential, float.
exp10l()	Compute base-10 exponential, long double.
frexp()	Split to significand and exponent, double.
frexpf()	Split to significand and exponent, float.
frexpl()	Split to significand and exponent, long double.
hypot()	Compute magnitude of complex, double.
hypotf()	Compute magnitude of complex, float.
hypotl()	Compute magnitude of complex, long double.
log()	Compute natural logarithm, double.
logf()	Compute natural logarithm, float.
logl()	Compute natural logarithm, long double.
log2()	Compute base-2 logarithm, double.
log2f()	Compute base-2 logarithm, float.
log2l()	Compute base-2 logarithm, long double.
log10()	Compute common logarithm, double.
log10f()	Compute common logarithm, float.
log10l()	Compute common logarithm, long double.
logb()	Radix-indpendent exponent, double.
logbf()	Radix-indpendent exponent, float.
logbl()	Radix-indpendent exponent, long double.

Function	Description
`ilogb()`	Radix-independent exponent, double.
`ilogbf()`	Radix-independent exponent, float.
`ilogbl()`	Radix-independent exponent, long double.
`log1p()`	Compute natural logarithm plus one, double.
`log1pf()`	Compute natural logarithm plus one, float.
`log1pl()`	Compute natural logarithm plus one, long double.
`ldexp()`	Scale by power of two, double.
`ldexpf()`	Scale by power of two, float.
`ldexpl()`	Scale by power of two, long double.
`pow()`	Raise to power, double.
`powf()`	Raise to power, float.
`powl()`	Raise to power, long double.
`scalbn()`	Scale, double.
`scalbnf()`	Scale, float.
`scalbnl()`	Scale, long double.
`scalbln()`	Scale, double.
`scalblnf()`	Scale, float.
`scalblnl()`	Scale, long double.

4.10.1.1 sqrt()

Description

Compute square root, double.

Prototype

```
double sqrt(double x);
```

Parameters

Parameter	Description
x	Value to compute square root of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- If x < 0, return NaN.
- Else, return square root of x.

Additional information

`sqrt()` computes the nonnegative square root of x. C90 and C99 require that a domain error occurs if the argument is less than zero, `sqrt()` deviates and always uses IEC 60559 semantics.

Thread safety

Safe.

4.10.1.2 sqrtf()

Description

Compute square root, float.

Prototype

```
float sqrtf(float x);
```

Parameters

Parameter	Description
x	Value to compute square root of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- If x < 0, return NaN.
- Else, return square root of x.

Additional information

sqrt() computes the nonnegative square root of x. C90 and C99 require that a domain error occurs if the argument is less than zero, sqrt() deviates and always uses IEC 60559 semantics.

Thread safety

Safe.

4.10.1.3 sqrtl()

Description

Compute square root, long double.

Prototype

```
long double sqrtl(long double x);
```

Parameters

Parameter	Description
x	Value to compute square root of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- If x < 0, return NaN.
- Else, return square root of x.

Additional information

`sqrtl()` computes the nonnegative square root of x. C90 and C99 require that a domain error occurs if the argument is less than zero, `sqrtl()` deviates and always uses IEC 60559 semantics.

Thread safety

Safe.

4.10.1.4 cbrt()

Description

Compute cube root, double.

Prototype

```
double cbrt(double x);
```

Parameters

Parameter	Description
x	Value to compute cube root of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return cube root of x.

Thread safety

Safe.

4.10.1.5 cbrtf()

Description

Compute cube root, float.

Prototype

```
float cbrtf(float x);
```

Parameters

Parameter	Description
x	Value to compute cube root of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return cube root of x.

Thread safety

Safe.

4.10.1.6 cbrtl()

Description

Compute cube root, long double.

Prototype

```
long double cbrtl(long double x);
```

Parameters

Parameter	Description
x	Value to compute cube root of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return cube root of x.

Thread safety

Safe.

4.10.1.7 rsqrt()

Description

Compute reciprocal square root, double.

Prototype

```
double rsqrt(double x);
```

Parameters

Parameter	Description
x	Value to compute reciprocal square root of.

Return value

- If x is +/-zero, return +/-infinity.
- If x is positively infinite, return 0.
- If x is NaN, return x.
- If x < 0, return NaN.
- Else, return reciprocal square root of x.

Thread safety

Safe.

4.10.1.8 rsqrtf()

Description

Compute reciprocal square root, float.

Prototype

```
float rsqrtf(float x);
```

Parameters

Parameter	Description
x	Value to compute reciprocal square root of.

Return value

- If x is +/-zero, return +/-infinity.
- If x is positively infinite, return 0.
- If x is NaN, return x.
- If x < 0, return NaN.
- Else, return reciprocal square root of x.

Thread safety

Safe.

4.10.1.9 rsqrtl()

Description

Compute reciprocal square root, long double.

Prototype

```
long double rsqrtl(long double x);
```

Parameters

Parameter	Description
x	Value to compute reciprocal square root of.

Return value

- If x is +/-zero, return +/-infinity.
- If x is positively infinite, return 0.
- If x is NaN, return x.
- If x < 0, return NaN.
- Else, return reciprocal square root of x.

Thread safety

Safe.

4.10.1.10 exp()

Description

Compute base-e exponential, double.

Prototype

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

Parameters

Parameter	Description
x	Value to compute base-e exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.11 expf()

Description

Compute base-e exponential, float.

Prototype

```
float expf(float x);
```

Parameters

Parameter	Description
x	Value to compute base-e exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.12 expl()

Description

Compute base-e exponential, long double.

Prototype

```
long double expl(long double x);
```

Parameters

Parameter	Description
x	Value to compute base-e exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.13 expm1()

Description

Compute base-e exponential, modified, double.

Prototype

```
double expm1(double x);
```

Parameters

Parameter	Description
x	Value to compute exponential of.

Return value

- If x is NaN, return x.
- Else, return base-e exponential of x minus 1 (e**x - 1).

Thread safety

Safe.

4.10.1.14 expm1f()

Description

Compute base-e exponential, modified, float.

Prototype

```
float expm1f(float x);
```

Parameters

Parameter	Description
x	Value to compute exponential of.

Return value

- If x is NaN, return x.
- Else, return base-e exponential of x minus 1 (e**x - 1).

Thread safety

Safe.

4.10.1.15 expm1l()

Description

Compute base-e exponential, modified, long double.

Prototype

```
long double expm1l(long double x);
```

Parameters

Parameter	Description
x	Value to compute exponential of.

Return value

- If x is NaN, return x.
- Else, return base-e exponential of x minus 1 (e**x - 1).

Thread safety

Safe.

4.10.1.16 exp2()

Description

Compute base-2 exponential, double.

Prototype

```
double exp2(double x);
```

Parameters

Parameter	Description
x	Value to compute base-2 exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.17 exp2f()

Description

Compute base-2 exponential, float.

Prototype

```
float exp2f(float x);
```

Parameters

Parameter	Description
x	Value to compute base-e exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.18 exp2l()

Description

Compute base-2 exponential, long double.

Prototype

```
long double exp2l(long double x);
```

Parameters

Parameter	Description
x	Value to compute base-2 exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.19 exp10()

Description

Compute base-10 exponential, double.

Prototype

```
double exp10(double x);
```

Parameters

Parameter	Description
x	Value to compute base-e exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.20 exp10f()

Description

Compute base-10 exponential, float.

Prototype

```
float exp10f(float x);
```

Parameters

Parameter	Description
x	Value to compute base-e exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.21 exp10l()

Description

Compute base-10 exponential, long double.

Prototype

```
long double exp10l(long double x);
```

Parameters

Parameter	Description
x	Value to compute base-e exponential of.

Return value

- If x is NaN, return x.
- If x is positively infinite, return x.
- If x is negatively infinite, return 0.
- Else, return base-e exponential of x.

Thread safety

Safe.

4.10.1.22 frexp()

Description

Split to significand and exponent, double.

Prototype

```
double frexp(double   x,
             int    * exp);
```

Parameters

Parameter	Description
x	Floating value to operate on.
exp	Pointer to integer receiving the power-of-two exponent of x.

Return value

- If x is zero, infinite or NaN, return x and store zero into the integer pointed to by exp.
- Else, return the value f, such that f has a magnitude in the interval [0.5, 1) and x equals f * pow(2, *exp)

Additional information

Breaks a floating-point number into a normalized fraction and an integral power of two.

Thread safety

Safe.

4.10.1.23 frexpf()

Description

Split to significand and exponent, float.

Prototype

```
float frexpf(float   x,
             int   * exp);
```

Parameters

Parameter	Description
x	Floating value to operate on.
exp	Pointer to integer receiving the power-of-two exponent of x.

Return value

- If x is zero, infinite or NaN, return x and store zero into the integer pointed to by exp.
- Else, return the value f, such that f has a magnitude in the interval [0.5, 1) and x equals f * pow(2, *exp)

Additional information

Breaks a floating-point number into a normalized fraction and an integral power of two.

Thread safety

Safe.

4.10.1.24 frexpl()

Description

Split to significand and exponent, long double.

Prototype

```
long double frexpl(long double   x,
                   int         * exp);
```

Parameters

Parameter	Description
x	Floating value to operate on.
exp	Pointer to integer receiving the power-of-two exponent of x.

Return value

- If x is zero, infinite or NaN, return x and store zero into the integer pointed to by exp.
- Else, return the value f, such that f has a magnitude in the interval [0.5, 1) and x equals f * pow(2, *exp)

Additional information

Breaks a floating-point number into a normalized fraction and an integral power of two.

Thread safety

Safe.

4.10.1.25 hypot()

Description

Compute magnitude of complex, double.

Prototype

```
double hypot(double x,
             double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x or y are infinite, return infinity.
- If x or y is NaN, return NaN.
- Else, return sqrt(x*x + y*y).

Additional information

Computes the square root of the sum of the squares of x and y without undue overflow or underflow. If x and y are the lengths of the sides of a right-angled triangle, then this computes the length of the hypotenuse.

Thread safety

Safe.

4.10.1.26 hypotf()

Description

Compute magnitude of complex, float.

Prototype

```
float hypotf(float x,
             float y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x or y are infinite, return infinity.
- If x or y is NaN, return NaN.
- Else, return sqrt(x*x + y*y).

Additional information

Computes the square root of the sum of the squares of x and y without undue overflow or underflow. If x and y are the lengths of the sides of a right-angled triangle, then this computes the length of the hypotenuse.

Thread safety

Safe.

4.10.1.27 hypotl()

Description

Compute magnitude of complex, long double.

Prototype

```
long double hypotl(long double x,
                   long double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x or y are infinite, return infinity.
- If x or y is NaN, return NaN.
- Else, return sqrtl(x*x + y*y).

Additional information

Computes the square root of the sum of the squares of x and y without undue overflow or underflow. If x and y are the lengths of the sides of a right-angled triangle, then this computes the length of the hypotenuse.

Thread safety

Safe.

4.10.1.28 log()

Description

Compute natural logarithm, double.

Prototype

```
double log(double x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return -∞.
- If x is +∞, return +∞.
- ELse, return base-e logarithm of x.

Thread safety

Safe.

4.10.1.29 logf()

Description

Compute natural logarithm, float.

Prototype

```
float logf(float x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-e logarithm of x.

Thread safety

Safe.

4.10.1.30 logl()

Description

Compute natural logarithm, long double.

Prototype

```
long double logl(long double x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return -∞.
- If x is +∞, return +∞.
- ELse, return base-e logarithm of x.

Thread safety

Safe.

4.10.1.31 log2()

Description

Compute base-2 logarithm, double.

Prototype

```
double log2(double x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-10 logarithm of x.

Thread safety

Safe.

4.10.1.32 log2f()

Description

Compute base-2 logarithm, float.

Prototype

```
float log2f(float x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-10 logarithm of x.

Thread safety

Safe.

4.10.1.33 log2l()

Description

Compute base-2 logarithm, long double.

Prototype

```
long double log2l(long double x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-10 logarithm of x.

Thread safety

Safe.

4.10.1.34 log10()

Description

Compute common logarithm, double.

Prototype

```
double log10(double x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-10 logarithm of x.

Thread safety

Safe.

4.10.1.35 log10f()

Description

Compute common logarithm, float.

Prototype

```
float log10f(float x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-10 logarithm of x.

Thread safety

Safe.

4.10.1.36 log10l()

Description

Compute common logarithm, long double.

Prototype

```
long double log10l(long double x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-10 logarithm of x.

Thread safety

Safe.

4.10.1.37 logb()

Description

Radix-indpendent exponent, double.

Prototype

```
double logb(double x);
```

Parameters

Parameter	Description
x	Floating value to operate on.

Return value

- If x is zero, return -∞.
- If x is infinite, return +∞.
- If x is NaN, return NaN.
- Else, return integer part of $\log_{\mathrm{FLTRADIX}}(x)$.

Additional information

Calculates the exponent of x, which is the integral part of the FLTRADIX-logarithm of x.

Thread safety

Safe.

4.10.1.38 logbf()

Description

Radix-indpendent exponent, float.

Prototype

```
float logbf(float x);
```

Parameters

Parameter	Description
x	Floating value to operate on.

Return value

- If x is zero, return -∞.
- If x is infinite, return +∞.
- If x is NaN, return NaN.
- Else, return integer part of $\log_{FLTRADIX}(x)$.

Additional information

Calculates the exponent of x, which is the integral part of the FLTRADIX-logarithm of x.

Thread safety

Safe.

4.10.1.39 logbl()

Description

Radix-indpendent exponent, long double.

Prototype

```
long double logbl(long double x);
```

Parameters

Parameter	Description
x	Floating value to operate on.

Return value

- If x is zero, return -∞.
- If x is infinite, return +∞.
- If x is NaN, return NaN.
- Else, return integer part of $\log_{\text{FLTRADIX}}(x)$.

Additional information

Calculates the exponent of x, which is the integral part of the FLTRADIX-logarithm of x.

Thread safety

Safe.

4.10.1.40 ilogb()

Description

Radix-independent exponent, double.

Prototype

```
int ilogb(double x);
```

Parameters

Parameter	Description
x	Floating value to operate on.

Return value

- If x is zero, return FP_ILOGB0.
- If x is NaN, return FP_ILOGBNAN.
- If x is infinite, return MAX_INT.
- Else, return integer part of $\log_{\text{FLTRADIX}}$(x).

Thread safety

Safe.

4.10.1.41 ilogbf()

Description

Radix-independent exponent, float.

Prototype

```
int ilogbf(float x);
```

Parameters

Parameter	Description
x	Floating value to operate on.

Return value

- If x is zero, return FP_ILOGB0.
- If x is NaN, return FP_ILOGBNAN.
- If x is infinite, return MAX_INT.
- Else, return integer part of $\log_{FLTRADIX}(x)$.

Thread safety

Safe.

4.10.1.42 ilogbl()

Description

Radix-independent exponent, long double.

Prototype

```
int ilogbl(long double x);
```

Parameters

Parameter	Description
x	Floating value to operate on.

Return value

- If x is zero, return `FP_ILOGB0`.
- If x is NaN, return `FP_ILOGBNAN`.
- If x is infinite, return `MAX_INT`.
- Else, return integer part of $\log_{\text{FLTRADIX}}$(x).

Thread safety

Safe.

4.10.1.43 log1p()

Description

Compute natural logarithm plus one, double.

Prototype

```
double log1p(double x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-e logarithm of x+1.

Thread safety

Safe.

4.10.1.44 log1pf()

Description

Compute natural logarithm plus one, float.

Prototype

```
float log1pf(float x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-e logarithm of x+1.

Thread safety

Safe.

4.10.1.45 log1pl()

Description

Compute natural logarithm plus one, long double.

Prototype

```
long double log1pl(long double x);
```

Parameters

Parameter	Description
x	Value to compute logarithm of.

Return value

- If x = NaN, return x.
- If x < 0, return NaN.
- If x = 0, return negative infinity.
- If x is positively infinite, return infinity.
- ELse, return base-e logarithm of x+1.

Thread safety

Safe.

4.10.1.46 ldexp()

Description

Scale by power of two, double.

Prototype

```
double ldexp(double x,
             int    n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of two to scale by.

Return value

- If x is ±0, return x;
- If x is ±∞, return x.
- If x is NaN, return x.
- Else, return x * 2 ^ n.

Additional information

Multiplies a floating-point number by an integral power of two.

Thread safety

Safe.

See also

`scalbn()`

4.10.1.47 ldexpf()

Description

Scale by power of two, float.

Prototype

```
float ldexpf(float x,
             int   n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of two to scale by.

Return value

- If x is zero, return x;
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x * 2^n.

Additional information

Multiplies a floating-point number by an integral power of two.

Thread safety

Safe.

See also

`scalbnf()`

4.10.1.48 ldexpl()

Description

Scale by power of two, long double.

Prototype

```
long double ldexpl(long double x,
                   int         n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of two to scale by.

Return value

- If x is ±0, return x;
- If x is ±∞, return x.
- If x is NaN, return x.
- Else, return x * 2 ^ n.

Additional information

Multiplies a floating-point number by an integral power of two.

Thread safety

Safe.

See also

scalbnl()

4.10.1.49 pow()

Description

Raise to power, double.

Prototype

```
double pow(double x,
           double y);
```

Parameters

Parameter	Description
x	Base.
y	Power.

Return value

Return x raised to the power y.

Thread safety

Safe.

4.10.1.50 powf()

Description

Raise to power, float.

Prototype

```
float powf(float x,
           float y);
```

Parameters

Parameter	Description
x	Base.
y	Power.

Return value

Return x raised to the power y.

Thread safety

Safe.

4.10.1.51 powl()

Description

Raise to power, long double.

Prototype

```
long double powl(long double x,
                 long double y);
```

Parameters

Parameter	Description
x	Base.
y	Power.

Return value

Return x raised to the power y.

Thread safety

Safe.

4.10.1.52 scalbn()

Description

Scale, double.

Prototype

```
double scalbn(double x,
              int    n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of DBL_RADIX to scale by.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x * DBL_RADIX ^ n.

Additional information

Multiplies a floating-point number by an integral power of DBL_RADIX.

As floating-point arithmetic conforms to IEC 60559, DBL_RADIX is 2 and scalbn() is (in this implementation) identical to ldexp().

Thread safety

Safe.

See also

ldexp()

4.10.1.53 scalbnf()

Description

Scale, float.

Prototype

```
float scalbnf(float x,
              int   n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of FLT_RADIX to scale by.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x * FLT_RADIX ^ n.

Additional information

Multiplies a floating-point number by an integral power of FLT_RADIX.

As floating-point arithmetic conforms to IEC 60559, FLT_RADIX is 2 and scalbnf() is (in this implementation) identical to ldexpf().

Thread safety

Safe.

See also

ldexpf()

4.10.1.54 scalbnl()

Description

Scale, long double.

Prototype

```
long double scalbnl(long double x,
                    int         n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of LDBL_RADIX to scale by.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x * LDBL_RADIX ^ n.

Additional information

Multiplies a floating-point number by an integral power of LDBL_RADIX.

As floating-point arithmetic conforms to IEC 60559, LDBL_RADIX is 2 and scalbnl() is (in this implementation) identical to ldexpl().

Thread safety

Safe.

See also

ldexpl()

4.10.1.55 scalbln()

Description

Scale, double.

Prototype

```
double scalbln(double x,
               long   n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of DBL_RADIX to scale by.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x * DBL_RADIX ^ n.

Additional information

Multiplies a floating-point number by an integral power of DBL_RADIX.

As floating-point arithmetic conforms to IEC 60559, DBL_RADIX is 2 and scalbln() is (in this implementation) identical to ldexp().

Thread safety

Safe.

See also

ldexp()

4.10.1.56 scalblnf()

Description

Scale, float.

Prototype

```
float scalblnf(float x,
               long  n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of FLT_RADIX to scale by.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x * FLT_RADIX ^ n.

Additional information

Multiplies a floating-point number by an integral power of FLT_RADIX.

As floating-point arithmetic conforms to IEC 60559, FLT_RADIX is 2 and scalbnf() is (in this implementation) identical to ldexpf().

Thread safety

Safe.

4.10.1.57 scalblnl()

Description

Scale, long double.

Prototype

```
long double scalblnl(long double x,
                     long        n);
```

Parameters

Parameter	Description
x	Value to scale.
n	Power of LDBL_RADIX to scale by.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x * LDBL_RADIX ^ n.

Additional information

Multiplies a floating-point number by an integral power of LDBL_RADIX.

As floating-point arithmetic conforms to IEC 60559, LDBL_RADIX is 2 and scalblnl() is (in this implementation) identical to ldexpl().

Thread safety

Safe.

See also

ldexpl()

4.10.2 Trigonometric functions

Function	Description
`sin()`	Calculate sine, double.
`sinf()`	Calculate sine, float.
`sinl()`	Calculate sine, long double.
`cos()`	Calculate cosine, double.
`cosf()`	Calculate cosine, float.
`cosl()`	Calculate cosine, long double.
`tan()`	Compute tangent, double.
`tanf()`	Compute tangent, float.
`tanl()`	Compute tangent, long double.
`sinh()`	Compute hyperbolic sine, double.
`sinhf()`	Compute hyperbolic sine, float.
`sinhl()`	Compute hyperbolic sine, long double.
`cosh()`	Compute hyperbolic cosine, double.
`coshf()`	Compute hyperbolic cosine, float.
`coshl()`	Compute hyperbolic cosine, long double.
`tanh()`	Compute hyperbolic tangent, double.
`tanhf()`	Compute hyperbolic tangent, float.
`tanhl()`	Compute hyperbolic tangent, long double.
`sincos()`	Calculate sine and cosine, double.
`sincosf()`	Calculate sine and cosine, float.
`sincosl()`	Calculate sine and cosine, long double.

4.10.2.1 sin()

Description

Calculate sine, double.

Prototype

```
double sin(double x);
```

Parameters

Parameter	Description
x	Angle to compute sine of, radians.

Return value

- If x is NaN, return x.
- If x is infinite, return NaN.
- Else, return circular sine of x.

Thread safety

Safe.

4.10.2.2 sinf()

Description

Calculate sine, float.

Prototype

```
float sinf(float x);
```

Parameters

Parameter	Description
x	Angle to compute sine of, radians.

Return value

- If x is NaN, return x.
- If x is infinite, return NaN.
- Else, return circular sine of x.

Thread safety

Safe.

4.10.2.3 sinl()

Description

Calculate sine, long double.

Prototype

```
long double sinl(long double x);
```

Parameters

Parameter	Description
x	Angle to compute sine of, radians.

Return value

- If x is NaN, return x.
- If x is infinite, return NaN.
- Else, return circular sine of x.

Thread safety

Safe.

4.10.2.4 cos()

Description

Calculate cosine, double.

Prototype

```
double cos(double x);
```

Parameters

Parameter	Description
x	Angle to compute cosine of, radians.

Return value

- If x is NaN, return x.
- If x is infinite, return NaN.
- Else, return circular cosine of x.

Thread safety

Safe.

4.10.2.5 cosf()

Description

Calculate cosine, float.

Prototype

```
float cosf(float x);
```

Parameters

Parameter	Description
x	Angle to compute cosine of, radians.

Return value

- If x is NaN, return x.
- If x is infinite, return NaN.
- Else, return circular cosine of x.

Thread safety

Safe.

4.10.2.6 cosl()

Description

Calculate cosine, long double.

Prototype

```
long double cosl(long double x);
```

Parameters

Parameter	Description
x	Angle to compute cosine of, radians.

Return value

- If x is NaN, return x.
- If x is infinite, return NaN.
- Else, return circular cosine of x.

Thread safety

Safe.

4.10.2.7 tan()

Description

Compute tangent, double.

Prototype

```
double tan(double x);
```

Parameters

Parameter	Description
x	Angle to compute tangent of, radians.

Return value

- If x is zero, return x.
- If x is infinite, return NaN.
- If x is NaN, return x.
- Else, return tangent of x.

Thread safety

Safe.

4.10.2.8 tanf()

Description

Compute tangent, float.

Prototype

```
float tanf(float x);
```

Parameters

Parameter	Description
x	Angle to compute tangent of, radians.

Return value

- If x is zero, return x.
- If x is infinite, return NaN.
- If x is NaN, return x.
- Else, return tangent of x.

Thread safety

Safe.

4.10.2.9 tanl()

Description

Compute tangent, long double.

Prototype

```
long double tanl(long double x);
```

Parameters

Parameter	Description
x	Angle to compute tangent of, radians.

Return value

- If x is zero, return x.
- If x is infinite, return NaN.
- If x is NaN, return x.
- Else, return tangent of x.

Thread safety

Safe.

4.10.2.10 sinh()

Description

Compute hyperbolic sine, double.

Prototype

```
double sinh(double x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic sine of.

Return value

- If x is NaN, return x.
- If x is infinite, return x.
- Else, return hyperbolic sine of x.

Thread safety

Safe.

4.10.2.11 sinhf()

Description

Compute hyperbolic sine, float.

Prototype

```
float sinhf(float x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic sine of.

Return value

- If x is NaN, return x.
- If x is infinite, return x.
- Else, return hyperbolic sine of x.

Thread safety

Safe.

4.10.2.12 sinhl()

Description

Compute hyperbolic sine, long double.

Prototype

```
long double sinhl(long double x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic sine of.

Return value

- If x is NaN, return x.
- If x is infinite, return x.
- Else, return hyperbolic sine of x.

Thread safety

Safe.

4.10.2.13 cosh()

Description

Compute hyperbolic cosine, double.

Prototype

```
double cosh(double x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic cosine of.

Return value

- If x is NaN, return x.
- If x is infinite, return +∞.
- Else, return hyperbolic cosine of x.

Thread safety

Safe.

4.10.2.14 coshf()

Description

Compute hyperbolic cosine, float.

Prototype

```
float coshf(float x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic cosine of.

Return value

- If x is NaN, return x.
- If x is infinite, return +∞.
- Else, return hyperbolic cosine of x.

Thread safety

Safe.

4.10.2.15 coshl()

Description

Compute hyperbolic cosine, long double.

Prototype

```
long double coshl(long double x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic cosine of.

Return value

- If x is NaN, return x.
- If x is infinite, return +∞.
- Else, return hyperbolic cosine of x.

Thread safety

Safe.

4.10.2.16 tanh()

Description

Compute hyperbolic tangent, double.

Prototype

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

Parameters

Parameter	Description
x	Value to compute hyperbolic tangent of.

Return value

- If x is NaN, return x.
- Else, return hyperbolic tangent of x.

Thread safety

Safe.

4.10.2.17 tanhf()

Description

Compute hyperbolic tangent, float.

Prototype

```
float tanhf(float x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic tangent of.

Return value

- If x is NaN, return x.
- Else, return hyperbolic tangent of x.

Thread safety

Safe.

4.10.2.18 tanhl()

Description

Compute hyperbolic tangent, long double.

Prototype

```
long double tanhl(long double x);
```

Parameters

Parameter	Description
x	Value to compute hyperbolic tangent of.

Return value

- If x is NaN, return x.
- Else, return hyperbolic tangent of x.

Thread safety

Safe.

4.10.2.19 sincos()

Description

Calculate sine and cosine, double.

Prototype

```
void sincos(double   x,
            double * pSin,
            double * pCos);
```

Parameters

Parameter	Description
x	Angle to compute sine and cosine of, radians.
pSin	Pointer to object that receives the sine of x.
pCos	Pointer to object that receives the cosine of x.

Thread safety

Safe.

4.10.2.20 sincosf()

Description

Calculate sine and cosine, float.

Prototype

```
void sincosf(float   x,
             float * pSin,
             float * pCos);
```

Parameters

Parameter	Description
`x`	Angle to compute sine and cosine of, radians.
`pSin`	Pointer to object that receives the sine of `x`.
`pCos`	Pointer to object that receives the cosine of `x`.

Thread safety

Safe.

4.10.2.21 sincosl()

Description

Calculate sine and cosine, long double.

Prototype

```
void sincosl(long double   x,
             long double * pSin,
             long double * pCos);
```

Parameters

Parameter	Description
x	Angle to compute sine and cosine of, radians.
pSin	Pointer to object that receives the sine of x.
pCos	Pointer to object that receives the cosine of x.

Thread safety

Safe.

4.10.3 Inverse trigonometric functions

Function	Description
asin()	Compute inverse sine, double.
asinf()	Compute inverse sine, float.
asinl()	Compute inverse sine, long double.
acos()	Compute inverse cosine, double.
acosf()	Compute inverse cosine, float.
acosl()	Compute inverse cosine, long double.
atan()	Compute inverse tangent, double.
atanf()	Compute inverse tangent, float.
atanl()	Compute inverse tangent, long double.
atan2()	Compute inverse tangent, with quadrant, double.
atan2f()	Compute inverse tangent, with quadrant, float.
atan2l()	Compute inverse tangent, with quadrant, long double.
asinh()	Compute inverse hyperbolic sine, double.
asinhf()	Compute inverse hyperbolic sine, float.
asinhl()	Compute inverse hyperbolic sine, long double.
acosh()	Compute inverse hyperbolic cosine, double.
acoshf()	Compute inverse hyperbolic cosine, float.
acoshl()	Compute inverse hyperbolic cosine, long double.
atanh()	Compute inverse hyperbolic tangent, double.
atanhf()	Compute inverse hyperbolic tangent, float.
atanhl()	Compute inverse hyperbolic tangent, long double.

4.10.3.1 asin()

Description

Compute inverse sine, double.

Prototype

```
double asin(double x);
```

Parameters

Parameter	Description
x	Value to compute inverse sine of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- Else, return inverse circular sine of x.

Additional information

Calculates the principal value, in radians, of the inverse circular sine of x. The principal value lies in the interval [-Pi/2, Pi/2] radians.

Thread safety

Safe.

4.10.3.2 asinf()

Description

Compute inverse sine, float.

Prototype

```
float asinf(float x);
```

Parameters

Parameter	Description
x	Value to compute inverse sine of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- Else, return inverse circular sine of x.

Additional information

Calculates the principal value, in radians, of the inverse circular sine of x. The principal value lies in the interval [-Pi/2, Pi/2] radians.

Thread safety

Safe.

4.10.3.3 asinl()

Description

Compute inverse sine, long double.

Prototype

```
long double asinl(long double x);
```

Parameters

Parameter	Description
x	Value to compute inverse sine of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- Else, return inverse circular sine of x.

Additional information

Calculates the principal value, in radians, of the inverse circular sine of x. The principal value lies in the interval [-Pi/2, Pi/2] radians.

Thread safety

Safe.

4.10.3.4 acos()

Description

Compute inverse cosine, double.

Prototype

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

Parameters

Parameter	Description
x	Value to compute inverse cosine of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- Else, return inverse circular cosine of x.

Additional information

Calculates the principal value, in radians, of the inverse circular cosine of x. The principal value lies in the interval [0, Pi] radians.

Thread safety

Safe.

4.10.3.5 acosf()

Description

Compute inverse cosine, float.

Prototype

```
float acosf(float x);
```

Parameters

Parameter	Description
x	Value to compute inverse cosine of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- Else, return inverse circular cosine of x.

Additional information

Calculates the principal value, in radians, of the inverse circular cosine of x. The principal value lies in the interval [0, Pi] radians.

Thread safety

Safe.

4.10.3.6 acosl()

Description

Compute inverse cosine, long double.

Prototype

```
long double acosl(long double x);
```

Parameters

Parameter	Description
x	Value to compute inverse cosine of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- Else, return inverse circular cosine of x.

Additional information

Calculates the principal value, in radians, of the inverse circular cosine of x. The principal value lies in the interval [0, Pi] radians.

Thread safety

Safe.

4.10.3.7 atan()

Description

Compute inverse tangent, double.

Prototype

```
double atan(double x);
```

Parameters

Parameter	Description
x	Value to compute inverse tangent of.

Return value

- If x is NaN, return x.
- Else, return inverse tangent of x.

Additional information

Calculates the principal value, in radians, of the inverse tangent of x. The principal value lies in the interval [-Pi/2, Pi/2] radians.

Thread safety

Safe.

4.10.3.8 atanf()

Description

Compute inverse tangent, float.

Prototype

```
float atanf(float x);
```

Parameters

Parameter	Description
x	Value to compute inverse tangent of.

Return value

- If x is NaN, return x.
- Else, return inverse tangent of x.

Additional information

Calculates the principal value, in radians, of the inverse tangent of x. The principal value lies in the interval [-Pi/2, Pi/2] radians.

Thread safety

Safe.

4.10.3.9 atanl()

Description

Compute inverse tangent, long double.

Prototype

```
long double atanl(long double x);
```

Parameters

Parameter	Description
x	Value to compute inverse tangent of.

Return value

- If x is NaN, return x.
- Else, return inverse tangent of x.

Additional information

Calculates the principal value, in radians, of the inverse tangent of x. The principal value lies in the interval [-Pi/2, Pi/2] radians.

Thread safety

Safe.

4.10.3.10 atan2()

Description

Compute inverse tangent, with quadrant, double.

Prototype

```
double atan2(double y,
             double x);
```

Parameters

Parameter	Description
y	Rise value of angle.
x	Run value of angle.

Return value

Inverse tangent of y/x.

Additional information

This calculates the value, in radians, of the inverse tangent of y divided by x using the signs of x and y to compute the quadrant of the return value. The principal value lies in the interval [-Pi, +Pi] radians.

Thread safety

Safe.

4.10.3.11 atan2f()

Description

Compute inverse tangent, with quadrant, float.

Prototype

```
float atan2f(float y,
             float x);
```

Parameters

Parameter	Description
y	Rise value of angle.
x	Run value of angle.

Return value

Inverse tangent of y/x.

Additional information

This calculates the value, in radians, of the inverse tangent of y divided by x using the signs of x and y to compute the quadrant of the return value. The principal value lies in the interval [-Pi, +Pi] radians.

Thread safety

Safe.

4.10.3.12 atan2l()

Description

Compute inverse tangent, with quadrant, long double.

Prototype

```
long double atan2l(long double y,
                   long double x);
```

Parameters

Parameter	Description
y	Rise value of angle.
x	Run value of angle.

Return value

Inverse tangent of y/x.

Additional information

This calculates the value, in radians, of the inverse tangent of y divided by x using the signs of x and y to compute the quadrant of the return value. The principal value lies in the interval [-Pi, +Pi] radians.

Thread safety

Safe.

4.10.3.13 asinh()

Description

Compute inverse hyperbolic sine, double.

Prototype

```
double asinh(double x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic sine of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return inverse hyperbolic sine of x.

Thread safety

Safe.

4.10.3.14 asinhf()

Description

Compute inverse hyperbolic sine, float.

Prototype

```
float asinhf(float x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic sine of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return inverse hyperbolic sine of x.

Additional information

Calculates the inverse hyperbolic sine of x.

Thread safety

Safe.

4.10.3.15 asinhl()

Description

Compute inverse hyperbolic sine, long double.

Prototype

```
long double asinhl(long double x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic sine of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return inverse hyperbolic sine of x.

Thread safety

Safe.

4.10.3.16 acosh()

Description

Compute inverse hyperbolic cosine, double.

Prototype

```
double acosh(double x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic cosine of.

Return value

- If x < 1, return NaN.
- If x is NaN, return x.
- Else, return non-negative inverse hyperbolic cosine of x.

Thread safety

Safe.

4.10.3.17 acoshf()

Description

Compute inverse hyperbolic cosine, float.

Prototype

```
float acoshf(float x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic cosine of.

Return value

- If x < 1, return NaN.
- If x is NaN, return x.
- Else, return non-negative inverse hyperbolic cosine of x.

Thread safety

Safe.

4.10.3.18 acoshl()

Description

Compute inverse hyperbolic cosine, long double.

Prototype

```
long double acoshl(long double x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic cosine of.

Return value

- If x < 1, return NaN.
- If x is NaN, return x.
- Else, return non-negative inverse hyperbolic cosine of x.

Thread safety

Safe.

4.10.3.19 atanh()

Description

Compute inverse hyperbolic tangent, double.

Prototype

```
double atanh(double x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic tangent of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- If x = +/-1, return +/-infinity.
- Else, return non-negative inverse hyperbolic tangent of x.

Thread safety

Safe.

4.10.3.20 atanhf()

Description

Compute inverse hyperbolic tangent, float.

Prototype

```
float atanhf(float x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic tangent of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- If x = +/-1, return +/-infinity.
- Else, return non-negative inverse hyperbolic tangent of x.

Thread safety

Safe.

4.10.3.21 atanhl()

Description

Compute inverse hyperbolic tangent, long double.

Prototype

```
long double atanhl(long double x);
```

Parameters

Parameter	Description
x	Value to compute inverse hyperbolic tangent of.

Return value

- If x is NaN, return x.
- If |x| > 1, return NaN.
- If x = +/-1, return +/-infinity.
- Else, return non-negative inverse hyperbolic tangent of x.

Thread safety

Safe.

4.10.4 Special functions

Function	Description
`erf()`	Error function, double.
`erff()`	Error function, float.
`erfl()`	Error function, long double.
`erfc()`	Complementary error function, double.
`erfcf()`	Complementary error function, float.
`erfcl()`	Complementary error function, long double.
`lgamma()`	Log-Gamma function, double.
`lgammaf()`	Log-Gamma function, float.
`lgammal()`	Log-Gamma function, long double.
`tgamma()`	Gamma function, double.
`tgammaf()`	Gamma function, float.
`tgammal()`	Gamma function, long double.

4.10.4.1 erf()

Description

Error function, double.

Prototype

```
double erf(double x);
```

Parameters

Parameter	Description
x	Argument.

Return value

erf(x).

Thread safety

Safe.

4.10.4.2 erff()

Description

Error function, float.

Prototype

```
float erff(float x);
```

Parameters

Parameter	Description
x	Argument.

Return value

erf(x).

Thread safety

Safe.

4.10.4.3 erfl()

Description

Error function, long double.

Prototype

```
long double erfl(long double x);
```

Parameters

Parameter	Description
x	Argument.

Return value

erf(x).

Thread safety

Safe.

4.10.4.4 erfc()

Description

Complementary error function, double.

Prototype

```
double erfc(double x);
```

Parameters

Parameter	Description
x	Argument.

Return value

erfc(x).

Thread safety

Safe.

4.10.4.5 erfcf()

Description

Complementary error function, float.

Prototype

```
float erfcf(float x);
```

Parameters

Parameter	Description
x	Argument.

Return value

erfc(x).

Thread safety

Safe.

4.10.4.6 erfcl()

Description

Complementary error function, long double.

Prototype

```
long double erfcl(long double x);
```

Parameters

Parameter	Description
x	Argument.

Return value

erfc(x).

Thread safety

Safe.

4.10.4.7 lgamma()

Description

Log-Gamma function, double.

Prototype

```
double lgamma(double x);
```

Parameters

Parameter	Description
x	Argument.

Return value

log(gamma(x)).

Thread safety

Safe.

4.10.4.8 lgammaf()

Description

Log-Gamma function, float.

Prototype

```
float lgammaf(float x);
```

Parameters

Parameter	Description
x	Argument.

Return value

log(gamma(x)).

Thread safety

Safe.

4.10.4.9 lgammal()

Description

Log-Gamma function, long double.

Prototype

```
long double lgammal(long double x);
```

Parameters

Parameter	Description
x	Argument.

Return value

log(gamma(x)).

Thread safety

Safe.

4.10.4.10 tgamma()

Description

Gamma function, double.

Prototype

```
double tgamma(double x);
```

Parameters

Parameter	Description
x	Argument.

Return value

gamma(x).

Thread safety

Safe.

4.10.4.11 tgammaf()

Description

Gamma function, float.

Prototype

```
float tgammaf(float x);
```

Parameters

Parameter	Description
x	Argument.

Return value

gamma(x).

Thread safety

Safe.

4.10.4.12 tgammal()

Description

Gamma function, long double.

Prototype

```
long double tgammal(long double x);
```

Parameters

Parameter	Description
x	Argument.

Return value

gamma(x).

Thread safety

Safe.

4.10.5 Rounding and remainder functions

Function	Description
ceil()	Compute smallest integer not less than, double.
ceilf()	Compute smallest integer not less than, float.
ceill()	Compute smallest integer not less than, long double.
floor()	Compute largest integer not greater than, double.
floorf()	Compute largest integer not greater than, float.
floorl()	Compute largest integer not greater than, long double.
trunc()	Truncate to integer, double.
truncf()	Truncate to integer, float.
truncl()	Truncate to integer, long double.
rint()	Round to nearest integer, double.
rintf()	Round to nearest integer, float.
rintl()	Round to nearest integer, long double.
lrint()	Round to nearest integer, double.
lrintf()	Round to nearest integer, float.
lrintl()	Round to nearest integer, long double.
llrint()	Round to nearest integer, double.
llrintf()	Round to nearest integer, float.
llrintl()	Round to nearest integer, long double.
round()	Round to nearest integer, double.
roundf()	Round to nearest integer, float.
roundl()	Round to nearest integer, long double.
lround()	Round to nearest integer, double.
lroundf()	Round to nearest integer, float.
lroundl()	Round to nearest integer, long double.
llround()	Round to nearest integer, double.
llroundf()	Round to nearest integer, float.
llroundl()	Round to nearest integer, long double.
nearbyint()	Round to nearest integer, double.
nearbyintf()	Round to nearest integer, float.
nearbyintl()	Round to nearest integer, long double.
fmod()	Compute remainder after division, double.
fmodf()	Compute remainder after division, float.
fmodl()	Compute remainder after division, long double.
modf()	Separate integer and fractional parts, double.
modff()	Separate integer and fractional parts, float.
modfl()	Separate integer and fractional parts, long double.
remainder()	Compute remainder after division, double.
remainderf()	Compute remainder after division, float.
remainderl()	Compute remainder after division, long double.
remquo()	Compute remainder after division, double.
remquof()	Compute remainder after division, float.
remquol()	Compute remainder after division, long double.

4.10.5.1 ceil()

Description

Compute smallest integer not less than, double.

Prototype

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

Parameters

Parameter	Description
x	Value to compute ceiling of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the smallest integer value not greater than x.

Thread safety

Safe.

4.10.5.2 ceilf()

Description

Compute smallest integer not less than, float.

Prototype

```
float ceilf(float x);
```

Parameters

Parameter	Description
x	Value to compute ceiling of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the smallest integer value not greater than x.

Thread safety

Safe.

4.10.5.3 ceill()

Description

Compute smallest integer not less than, long double.

Prototype

```
long double ceill(long double x);
```

Parameters

Parameter	Description
x	Value to compute ceiling of.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the smallest integer value not greater than x.

Thread safety

Safe.

4.10.5.4 floor()

Description

Compute largest integer not greater than, double.

Prototype

```
double floor(double x);
```

Parameters

Parameter	Description
x	Value to floor.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the largest integer value not greater than x.

Thread safety

Safe.

4.10.5.5 floorf()

Description

Compute largest integer not greater than, float.

Prototype

```
float floorf(float x);
```

Parameters

Parameter	Description
x	Value to floor.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the largest integer value not greater than x.

Thread safety

Safe.

4.10.5.6 floorl()

Description

Compute largest integer not greater than, long double.

Prototype

```
long double floorl(long double x);
```

Parameters

Parameter	Description
x	Value to floor.

Return value

- If x is zero, return x.
- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the largest integer value not greater than x.

Thread safety

Safe.

4.10.5.7 trunc()

Description

Truncate to integer, double.

Prototype

```
double trunc(double x);
```

Parameters

Parameter	Description
x	Value to truncate.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x with fractional part removed.

Thread safety

Safe.

4.10.5.8 truncf()

Description

Truncate to integer, float.

Prototype

```
float truncf(float x);
```

Parameters

Parameter	Description
x	Value to truncate.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x with fractional part removed.

Thread safety

Safe.

4.10.5.9 truncl()

Description

Truncate to integer, long double.

Prototype

```
long double truncl(long double x);
```

Parameters

Parameter	Description
x	Value to truncate.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return x with fractional part removed.

Thread safety

Safe.

4.10.5.10 rint()

Description

Round to nearest integer, double.

Prototype

```
double rint(double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.11 rintf()

Description

Round to nearest integer, float.

Prototype

```
float rintf(float x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.12 rintl()

Description

Round to nearest integer, long double.

Prototype

```
long double rintl(long double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.13 lrint()

Description

Round to nearest integer, double.

Prototype

```
long lrint(double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.14 lrintf()

Description

Round to nearest integer, float.

Prototype

```
long lrintf(float x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.15 lrintl()

Description

Round to nearest integer, long double.

Prototype

```
long lrintl(long double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.16 llrint()

Description

Round to nearest integer, double.

Prototype

```
long long llrint(double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.17 llrintf()

Description

Round to nearest integer, float.

Prototype

```
long long llrintf(float x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.18 llrintl()

Description

Round to nearest integer, long double.

Prototype

```
long long llrintl(long double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.19 round()

Description

Round to nearest integer, double.

Prototype

```
double round(double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x, ties away from zero.

Thread safety

Safe.

4.10.5.20 roundf()

Description

Round to nearest integer, float.

Prototype

```
float roundf(float x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x, ties away from zero.

Thread safety

Safe.

4.10.5.21 roundl()

Description

Round to nearest integer, long double.

Prototype

```
long double roundl(long double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x, ties away from zero.

Thread safety

Safe.

4.10.5.22 lround()

Description

Round to nearest integer, double.

Prototype

```
long lround(double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.23 lroundf()

Description

Round to nearest integer, float.

Prototype

```
long lroundf(float x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.24 lroundl()

Description

Round to nearest integer, long double.

Prototype

```
long lroundl(long double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.25 llround()

Description

Round to nearest integer, double.

Prototype

```
long long llround(double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.26 llroundf()

Description

Round to nearest integer, float.

Prototype

```
long long llroundf(float x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.27 llroundl()

Description

Round to nearest integer, long double.

Prototype

```
long long llroundl(long double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.28 nearbyint()

Description

Round to nearest integer, double.

Prototype

```
double nearbyint(double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.29 nearbyintf()

Description

Round to nearest integer, float.

Prototype

```
float nearbyintf(float x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.30 nearbyintl()

Description

Round to nearest integer, long double.

Prototype

```
long double nearbyintl(long double x);
```

Parameters

Parameter	Description
x	Value to compute nearest integer of.

Return value

- If x is infinite, return x.
- If x is NaN, return x.
- Else, return the nearest integer value to x.

Thread safety

Safe.

4.10.5.31 fmod()

Description

Compute remainder after division, double.

Prototype

```
double fmod(double x,
            double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.5.32 fmodf()

Description

Compute remainder after division, float.

Prototype

```
float fmodf(float x,
            float y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.5.33 fmodl()

Description

Compute remainder after division, long double.

Prototype

```
long double fmodl(long double x,
                  long double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.5.34 modf()

Description

Separate integer and fractional parts, double.

Prototype

```
double modf(double   x,
            double * iptr);
```

Parameters

Parameter	Description
x	Value to separate.
iptr	Pointer to object that receives the integral part of x.

Return value

The signed fractional part of x.

Additional information

Breaks x into integral and fractional parts, each of which has the same type and sign as x.

The integral part (in floating-point format) is stored in the object pointed to by iptr and modf() returns the signed fractional part of x.

Thread safety

Safe.

4.10.5.35 modff()

Description

Separate integer and fractional parts, float.

Prototype

```
float modff(float   x,
            float * iptr);
```

Parameters

Parameter	Description
x	Value to separate.
iptr	Pointer to object that receives the integral part of x.

Return value

The signed fractional part of x.

Additional information

Breaks x into integral and fractional parts, each of which has the same type and sign as x.

The integral part (in floating-point format) is stored in the object pointed to by iptr and modff() returns the signed fractional part of x.

Thread safety

Safe.

4.10.5.36 modfl()

Description

Separate integer and fractional parts, long double.

Prototype

```
long double modfl(long double   x,
                  long double * iptr);
```

Parameters

Parameter	Description
x	Value to separate.
iptr	Pointer to object that receives the integral part of x.

Return value

The signed fractional part of x.

Additional information

Breaks x into integral and fractional parts, each of which has the same type and sign as x.

The integral part (in floating-point format) is stored in the object pointed to by iptr and modf() returns the signed fractional part of x.

Thread safety

Safe.

4.10.5.37 remainder()

Description

Compute remainder after division, double.

Prototype

```
double remainder(double x,
                 double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.5.38 remainderf()

Description

Compute remainder after division, float.

Prototype

```
float remainderf(float x,
                 float y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.5.39 remainderl()

Description

Compute remainder after division, long double.

Prototype

```
long double remainderl(long double x,
                       long double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.5.40 remquo()

Description

Compute remainder after division, double.

Prototype

```
double remquo(double   x,
              double   y,
              int    * quo);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.
quo	Pointer to object that receives the integer part of x divided by y.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.5.41 remquof()

Description

Compute remainder after division, float.

Prototype

```
float remquof(float   x,
              float   y,
              int   * quo);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.
quo	Pointer to object that receives the integer part of x divided by y.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.5.42 remquol()

Description

Compute remainder after division, long double.

Prototype

```
long double remquol(long double   x,
                    long double   y,
                    int         * quo);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.
quo	Pointer to object that receives the integer part of x divided by y.

Return value

- If x is NaN, return NaN.
- If x is zero and y is nonzero, return x.
- If x is infinite, return NaN.
- If x is finite and y is infinite, return x.
- If y is NaN, return NaN.
- If y is zero, return NaN.
- Else, return remainder of x divided by y.

Additional information

Computes the floating-point remainder of x divided by y, i.e. the value x - i*y for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

Thread safety

Safe.

4.10.6 Absolute value functions

Function	Description
`fabs()`	Compute absolute value, double.
`fabsf()`	Compute absolute value, float.
`fabsl()`	Compute absolute value, long double.

4.10.6.1 fabs()

Description

Compute absolute value, double.

Prototype

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

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

- If x is NaN, return x.
- Else, absolute value of x.

Thread safety

Safe.

4.10.6.2 fabsf()

Description

Compute absolute value, float.

Prototype

```
float fabsf(float x);
```

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

- If x is NaN, return x.
- Else, absolute value of x.

Thread safety

Safe.

4.10.6.3 fabsl()

Description

Compute absolute value, long double.

Prototype

```
long double fabsl(long double x);
```

Parameters

Parameter	Description
x	Value to compute magnitude of.

Return value

- If x is NaN, return x.
- Else, absolute value of x.

Thread safety

Safe.

4.10.7 Fused multiply functions

Function	Description
`fma()`	Compute fused multiply-add, double.
`fmaf()`	Compute fused multiply-add, float.
`fmal()`	Compute fused multiply-add, long double.

4.10.7.1 fma()

Description

Compute fused multiply-add, double.

Prototype

```
double fma(double x,
           double y,
           double z);
```

Parameters

Parameter	Description
x	Multiplicand.
y	Multiplier.
z	Summand.

Return value

Return (x * y) + z.

Thread safety

Safe.

4.10.7.2 fmaf()

Description

Compute fused multiply-add, float.

Prototype

```
float fmaf(float x,
           float y,
           float z);
```

Parameters

Parameter	Description
x	Multiplier.
y	Multiplicand.
z	Summand.

Return value

Return (x * y) + z.

Thread safety

Safe.

4.10.7.3 fmal()

Description

Compute fused multiply-add, long double.

Prototype

```
long double fmal(long double x,
                 long double y,
                 long double z);
```

Parameters

Parameter	Description
x	Multiplicand.
y	Multiplier.
z	Summand.

Return value

Return (x * y) + z.

Thread safety

Safe.

4.10.8 Maximum, minimum, and positive difference functions

Function	Description
`fmin()`	Compute minimum, double.
`fminf()`	Compute minimum, float.
`fminl()`	Compute minimum, long double.
`fmax()`	Compute maximum, double.
`fmaxf()`	Compute maximum, float.
`fmaxl()`	Compute maximum, long double.
`fdim()`	Positive difference, double.
`fdimf()`	Positive difference, float.
`fdiml()`	Positive difference, long double.

4.10.8.1 fmin()

Description

Compute minimum, double.

Prototype

```
double fmin(double x,
            double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return y.
- If y is NaN, return x.
- Else, return minimum of x and y.

Thread safety

Safe.

4.10.8.2 fminf()

Description

Compute minimum, float.

Prototype

```
float fminf(float x,
            float y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return y.
- If y is NaN, return x.
- Else, return minimum of x and y.

Thread safety

Safe.

4.10.8.3 fminl()

Description

Compute minimum, long double.

Prototype

```
long double fminl(long double x,
                  long double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return y.
- If y is NaN, return x.
- Else, return minimum of x and y.

Thread safety

Safe.

4.10.8.4 fmax()

Description

Compute maximum, double.

Prototype

```
double fmax(double x,
            double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return y.
- If y is NaN, return x.
- Else, return maximum of x and y.

Thread safety

Safe.

4.10.8.5 fmaxf()

Description

Compute maximum, float.

Prototype

```
float fmaxf(float x,
            float y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return y.
- If y is NaN, return x.
- Else, return maximum of x and y.

Thread safety

Safe.

4.10.8.6 fmaxl()

Description

Compute maximum, long double.

Prototype

```
long double fmaxl(long double x,
                  long double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x is NaN, return y.
- If y is NaN, return x.
- Else, return maximum of x and y.

Thread safety

Safe.

4.10.8.7 fdim()

Description

Positive difference, double.

Prototype

```
double fdim(double x,
            double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x > y, x-y.
- Else, +0.

Thread safety

Safe.

4.10.8.8 fdimf()

Description

Positive difference, float.

Prototype

```
float fdimf(float x,
            float y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x > y, x-y.
- Else, +0.

Thread safety

Safe.

4.10.8.9 fdiml()

Description

Positive difference, long double.

Prototype

```
long double fdiml(long double x,
                  long double y);
```

Parameters

Parameter	Description
x	Value #1.
y	Value #2.

Return value

- If x > y, x-y.
- Else, +0.

Thread safety

Safe.

4.10.9 Miscellaneous functions

Function	Description
`nextafter()`	Next machine-floating value, double.
`nextafterf()`	Next machine-floating value, float.
`nextafterl()`	Next machine-floating value, long double.
`nexttoward()`	Next machine-floating value, double.
`nexttowardf()`	Next machine-floating value, float.
`nexttowardl()`	Next machine-floating value, long double.
`nan()`	Parse NaN, double.
`nanf()`	Parse NaN, float.
`nanl()`	Parse NaN, long double.
`copysign()`	Copy sign, double.
`copysignf()`	Copy sign, float.
`copysignl()`	Copy sign, long double.

4.10.9.1 nextafter()

Description

Next machine-floating value, double.

Prototype

```
double nextafter(double x,
                 double y);
```

Parameters

Parameter	Description
x	Value to step from.
y	Director to step in.

Return value

Next machine-floating value after x in direction of y.

Thread safety

Safe.

4.10.9.2 nextafterf()

Description

Next machine-floating value, float.

Prototype

```
float nextafterf(float x,
                 float y);
```

Parameters

Parameter	Description
x	Value to step from.
y	Director to step in.

Return value

Next machine-floating value after x in direction of y.

Thread safety

Safe.

4.10.9.3 nextafterl()

Description

Next machine-floating value, long double.

Prototype

```
long double nextafterl(long double x,
                       long double y);
```

Parameters

Parameter	Description
x	Value to step from.
y	Director to step in.

Return value

Next machine-floating value after x in direction of y.

Thread safety

Safe.

4.10.9.4 nexttoward()

Description

Next machine-floating value, double.

Prototype

```
double nexttoward(double      x,
                  long double y);
```

Parameters

Parameter	Description
x	Value to step from.
y	Direction to step in.

Return value

Next machine-floating value after x in direction of y.

Thread safety

Safe.

4.10.9.5 nexttowardf()

Description

Next machine-floating value, float.

Prototype

```
float nexttowardf(float       x,
                  long double y);
```

Parameters

Parameter	Description
x	Value to step from.
y	Direction to step in.

Return value

Next machine-floating value after x in direction of y.

Thread safety

Safe.

4.10.9.6 nexttowardl()

Description

Next machine-floating value, long double.

Prototype

```
long double nexttowardl(long double x,
                        long double y);
```

Parameters

Parameter	Description
x	Value to step from.
y	Direction to step in.

Return value

Next machine-floating value after x in direction of y.

Thread safety

Safe.

4.10.9.7 nan()

Description

Parse NaN, double.

Prototype

```
double nan(const char * tag);
```

Parameters

Parameter	Description
`tag`	NaN `tag`.

Return value

Quiet NaN formed from tag.

Thread safety

Safe.

4.10.9.8 nanf()

Description

Parse NaN, float.

Prototype

```
float nanf(const char * tag);
```

Parameters

Parameter	Description
`tag`	NaN `tag`.

Return value

Quiet NaN formed from tag.

Thread safety

Safe.

4.10.9.9 nanl()

Description

Parse NaN, long double.

Prototype

```
long double nanl(const char * tag);
```

Parameters

Parameter	Description
`tag`	NaN `tag`.

Return value

Quiet NaN formed from tag.

Thread safety

Safe.

4.10.9.10 copysign()

Description

Copy sign, double.

Prototype

```
double copysign(double x,
                double y);
```

Parameters

Parameter	Description
x	Floating value to inject sign into.
y	Floating value carrying the sign to inject.

Return value

x with the sign of y.

Thread safety

Safe.

4.10.9.11 copysignf()

Description

Copy sign, float.

Prototype

```
float copysignf(float x,
                float y);
```

Parameters

Parameter	Description
x	Floating value to inject sign into.
y	Floating value carrying the sign to inject.

Return value

x with the sign of y.

Thread safety

Safe.

4.10.9.12 copysignl()

Description

Copy sign, long double.

Prototype

```
long double copysignl(long double x,
                      long double y);
```

Parameters

Parameter	Description
x	Floating value to inject sign into.
y	Floating value carrying the sign to inject.

Return value

x with the sign of y.

Thread safety

Safe.

4.11 <setjmp.h>

Function	Description
`setjmp()`	Save calling environment for non-local jump.
`longjmp()`	Restores the saved environment.

4.11.1 Non-local flow control

4.11.1.1 setjmp()

Description

Save calling environment for non-local jump.

Prototype

```
int setjmp(jmp_buf buf);
```

Parameters

Parameter	Description
`buf`	Buffer to save context into.

Return value

On return from a direct invocation, returns the value zero. On return from a call to the `longjmp()` function, returns a nonzero value determined by the call to `longjmp()`.

Additional information

Saves its calling environment in env for later use by the `longjmp()` function.

The environment saved by a call to setjmp () consists of information sufficient for a call to the `longjmp()` function to return execution to the correct block and invocation of that block, were it called recursively.

Thread safety

Safe.

4.11.1.2 longjmp()

Description

Restores the saved environment.

Prototype

```
void longjmp(jmp_buf buf,
             int     val);
```

Parameters

Parameter	Description
buf	Buffer to restore context from.
val	Value to return to `setjmp()` call.

Additional information

Restores the environment saved by `setjmp()` in the corresponding env argument. If there has been no such invocation, or if the function containing the invocation of `setjmp()` has terminated execution in the interim, the behavior of `longjmp()` is undefined.

After `longjmp()` is completed, program execution continues as if the corresponding invocation of `setjmp()` had just returned the value specified by val.

Objects of automatic storage allocation that are local to the function containing the invocation of the corresponding `setjmp()` that do not have volatile-qualified type and have been changed between the `setjmp()` invocation and `longjmp()` call are indeterminate.

Notes

`longjmp()` cannot cause `setjmp()` to return the value 0; if `val` is 0, `setjmp()` returns the value 1.

Thread safety

Safe.

4.12 <signal.h>

Function	Description
signal()	Register signal function.
raise()	Raise a signal.

4.12.1 Exceptions

4.12.1.1 signal()

Description

Register signal function.

Prototype

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

Parameters

Parameter	Description
sig	Signal being registered.
func	Function to call when signal raised.

Return value

Previously-registered signal handler.

Thread safety

Safe.

4.12.1.2 raise()

Description

Raise a signal.

Prototype

```
int raise(int sig);
```

Parameters

Parameter	Description
sig	Signal to raise.

Return value

Zero if success.

Additional information

Signal handlers are executed in the context of the calling thread, if any. Signal handlers should not access or maniplate thread-local data.

Thread safety

Safe.

4.13 <stdbool.h>

4.13.1 Macros

4.13.1.1 bool

Description

Macros expanding to support the Boolean type.

Definition

```
#define bool      _Bool
#define true      1
#define false     0
```

Symbols

Definition	Description
`bool`	Underlying boolean type
`true`	Boolean `true` value
`false`	Boolean `false` value

4.14 <stddef.h>

4.14.1 Macros

4.14.1.1 NULL

Description

Null-pointer constant.

Definition

```
#define NULL    0
```

Symbols

Definition	Description
NULL	Null pointer

4.14.1.2 offsetof

Description

Calculate offset of member from start of structure.

Definition

```
#define offsetof(s,m)    __SEGGER_RTL_OFFSETOF(s, m)
```

Symbols

Definition	Description
offsetof(s,m)	Internal use.

4.14.2 Types

4.14.2.1 size_t

Description

Unsigned integral type returned by the sizeof operator.

Type definition

```
typedef __SEGGER_RTL_SIZE_T size_t;
```

4.14.2.2 ptrdiff_t

Description

Signed integral type of the result of subtracting two pointers.

Type definition

```
typedef __SEGGER_RTL_PTRDIFF_T ptrdiff_t;
```

4.14.2.3 wchar_t

Description

Integral type that can hold one wide character.

Type definition

```
typedef __SEGGER_RTL_WCHAR_T wchar_t;
```

4.15 <stdint.h>

4.15.1 Minima and maxima

4.15.1.1 Signed integer minima and maxima

Description

Minimum and maximum values for signed integer types.

Definition

```
#define INT8_MIN     (-128)
#define INT8_MAX     127
#define INT16_MIN    (-32767-1)
#define INT16_MAX    32767
#define INT32_MIN    (-2147483647L-1)
#define INT32_MAX    2147483647L
#define INT64_MIN    (-9223372036854775807LL-1)
#define INT64_MAX    9223372036854775807LL
```

Symbols

Definition	Description
INT8_MIN	Minimum value of int8_t
INT8_MAX	Maximum value of int8_t
INT16_MIN	Minimum value of int16_t
INT16_MAX	Maximum value of int16_t
INT32_MIN	Minimum value of int32_t
INT32_MAX	Maximum value of int32_t
INT64_MIN	Minimum value of int64_t
INT64_MAX	Maximum value of int64_t

4.15.1.2 Unsigned integer minima and maxima

Description

Minimum and maximum values for unsigned integer types.

Definition

```
#define UINT8_MAX     255
#define UINT16_MAX    65535
#define UINT32_MAX    4294967295UL
#define UINT64_MAX    18446744073709551615ULL
```

Symbols

Definition	Description
UINT8_MAX	Maximum value of uint8_t
UINT16_MAX	Maximum value of uint16_t
UINT32_MAX	Maximum value of uint32_t
UINT64_MAX	Maximum value of uint64_t

4.15.1.3 Maximal integer minima and maxima

Description

Minimum and maximum values for signed and unsigned maximal-integer types.

Definition

```
#define INTMAX_MIN      INT64_MIN
#define INTMAX_MAX      INT64_MAX
#define UINTMAX_MAX     UINT64_MAX
```

Symbols

Definition	Description
INTMAX_MIN	Minimum value of intmax_t
INTMAX_MAX	Maximum value of intmax_t
UINTMAX_MAX	Maximum value of uintmax_t

4.15.1.4 Least integer minima and maxima

Description

Minimum and maximum values for signed and unsigned least-integer types.

Definition

```
#define INT_LEAST8_MIN      INT8_MIN
#define INT_LEAST8_MAX      INT8_MAX
#define INT_LEAST16_MIN     INT16_MIN
#define INT_LEAST16_MAX     INT16_MAX
#define INT_LEAST32_MIN     INT32_MIN
#define INT_LEAST32_MAX     INT32_MAX
#define INT_LEAST64_MIN     INT64_MIN
#define INT_LEAST64_MAX     INT64_MAX
#define UINT_LEAST8_MAX     UINT8_MAX
#define UINT_LEAST16_MAX    UINT16_MAX
#define UINT_LEAST32_MAX    UINT32_MAX
#define UINT_LEAST64_MAX    UINT64_MAX
```

Symbols

Definition	Description
INT_LEAST8_MIN	Minimum value of int_least8_t
INT_LEAST8_MAX	Maximum value of int_least8_t
INT_LEAST16_MIN	Minimum value of int_least16_t
INT_LEAST16_MAX	Maximum value of int_least16_t
INT_LEAST32_MIN	Minimum value of int_least32_t
INT_LEAST32_MAX	Maximum value of int_least32_t
INT_LEAST64_MIN	Minimum value of int_least64_t
INT_LEAST64_MAX	Maximum value of int_least64_t
UINT_LEAST8_MAX	Maximum value of uint_least8_t
UINT_LEAST16_MAX	Maximum value of uint_least16_t
UINT_LEAST32_MAX	Maximum value of uint_least32_t
UINT_LEAST64_MAX	Maximum value of uint_least64_t

4.15.1.5 Fast integer minima and maxima

Description

Minimum and maximum values for signed and unsigned fast-integer types.

Definition

```
#define INT_FAST8_MIN      INT8_MIN
#define INT_FAST8_MAX      INT8_MAX
#define INT_FAST16_MIN     INT32_MIN
#define INT_FAST16_MAX     INT32_MAX
#define INT_FAST32_MIN     INT32_MIN
#define INT_FAST32_MAX     INT32_MAX
#define INT_FAST64_MIN     INT64_MIN
#define INT_FAST64_MAX     INT64_MAX
#define UINT_FAST8_MAX     UINT8_MAX
#define UINT_FAST16_MAX    UINT32_MAX
#define UINT_FAST32_MAX    UINT32_MAX
#define UINT_FAST64_MAX    UINT64_MAX
```

Symbols

Definition	Description
INT_FAST8_MIN	Minimum value of int_fast8_t
INT_FAST8_MAX	Maximum value of int_fast8_t
INT_FAST16_MIN	Minimum value of int_fast16_t
INT_FAST16_MAX	Maximum value of int_fast16_t
INT_FAST32_MIN	Minimum value of int_fast32_t
INT_FAST32_MAX	Maximum value of int_fast32_t
INT_FAST64_MIN	Minimum value of int_fast64_t
INT_FAST64_MAX	Maximum value of int_fast64_t
UINT_FAST8_MAX	Maximum value of uint_fast8_t
UINT_FAST16_MAX	Maximum value of uint_fast16_t
UINT_FAST32_MAX	Maximum value of uint_fast32_t
UINT_FAST64_MAX	Maximum value of uint_fast64_t

4.15.1.6 Pointer types minima and maxima

Description

Minimum and maximum values for pointer-related types.

Definition

```
#define PTRDIFF_MIN    INT64_MIN
#define PTRDIFF_MAX    INT64_MAX
#define SIZE_MAX       INT64_MAX
#define INTPTR_MIN     INT64_MIN
#define INTPTR_MAX     INT64_MAX
#define UINTPTR_MAX    UINT64_MAX
```

Symbols

Definition	Description
`PTRDIFF_MIN`	Minimum value of `ptrdiff_t`
`PTRDIFF_MAX`	Maximum value of `ptrdiff_t`
`SIZE_MAX`	Maximum value of `size_t`
`INTPTR_MIN`	Minimum value of `intptr_t`
`INTPTR_MAX`	Maximum value of `intptr_t`
`UINTPTR_MAX`	Maximum value of `uintptr_t`
`PTRDIFF_MIN`	Minimum value of `ptrdiff_t`
`PTRDIFF_MAX`	Maximum value of `ptrdiff_t`
`SIZE_MAX`	Maximum value of `size_t`
`INTPTR_MIN`	Minimum value of `intptr_t`
`INTPTR_MAX`	Maximum value of `intptr_t`
`UINTPTR_MAX`	Maximum value of `uintptr_t`

4.15.1.7 Wide integer minima and maxima

Description

Minimum and maximum values for the `wint_t` type.

Definition

```
#define WINT_MIN    (-2147483647L-1)
#define WINT_MAX    2147483647L
```

Symbols

Definition	Description
WINT_MIN	Minimum value of `wint_t`
WINT_MAX	Maximum value of `wint_t`

4.15.2 Constant construction macros

4.15.2.1 Signed integer construction macros

Description

Macros that create constants of type `intx_t`.

Definition

```
#define INT8_C(x)     (x)
#define INT16_C(x)    (x)
#define INT32_C(x)    (x)
#define INT64_C(x)    (x##LL)
```

Symbols

Definition	Description
`INT8_C(x)`	Create constant of type `int8_t`
`INT16_C(x)`	Create constant of type `int16_t`
`INT32_C(x)`	Create constant of type `int32_t`
`INT64_C(x)`	Create constant of type `int64_t`

4.15.2.2 Unsigned integer construction macros

Description

Macros that create constants of type `uintx_t`.

Definition

```
#define UINT8_C(x)     (x##u)
#define UINT16_C(x)    (x##u)
#define UINT32_C(x)    (x##u)
#define UINT64_C(x)    (x##uLL)
```

Symbols

Definition	Description
`UINT8_C(x)`	Create constant of type `uint8_t`
`UINT16_C(x)`	Create constant of type `uint16_t`
`UINT32_C(x)`	Create constant of type `uint32_t`
`UINT64_C(x)`	Create constant of type `uint64_t`

4.15.2.3 Maximal integer construction macros

Description

Macros that create constants of type `intmax_t` and `uintmax_t`.

Definition

```
#define INTMAX_C(x)     (x##LL)
#define UINTMAX_C(x)    (x##uLL)
```

Symbols

Definition	Description
`INTMAX_C(x)`	Create constant of type `intmax_t`
`UINTMAX_C(x)`	Create constant of type `uintmax_t`

4.16 <stdio.h>

4.16.1 Formatted output control strings

The functions in this section that accept a formatted output control string do so according to the specification that follows.

4.16.1.1 Composition

The format is composed of zero or more directives: ordinary characters (not `%`, which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each conversion specification is introduced by the character `%`. After the `%` the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional *minimum field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag has been given) to the field width. The field width takes the form of an asterisk `*` or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point character for `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of bytes to be written for `s` conversions. The precision takes the form of a period `.` followed either by an asterisk `*` or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, must appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

4.16.1.2 Flag characters

The flag characters and their meanings are:

Flag	Description
`-`	The result of the conversion is left-justified within the field. The default, if this flag is not specified, is that the result of the conversion is left-justified within the field.
`+`	The result of a signed conversion *always* begins with a plus or minus sign. The default, if this flag is not specified, is that it begins with a sign only when a negative value is converted.
space	If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and `+` flags both appear, the space flag is ignored.
`#`	The result is converted to an *alternative form*. For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both zero, a single `0` is printed). For `x` or `X` conversion, a nonzero result has `0x` or `0X` prefixed to it. For `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point

Flag	Description
	character appears in the result of these conversions only if a digit follows it.) For `g` and `F` conversions, trailing zeros are not removed from the result. As an extension, when used in `p` conversion, the results has `#` prefixed to it. For other conversions, the behavior is undefined.
`0`	For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.

4.16.1.3 Length modifiers

The length modifiers and their meanings are:

Flag	Description
`hh`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value will be converted to `signed char` or `unsigned char` before printing); or that a following `n` conversion specifier applies to a pointer to a `signed char` argument.
`h`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short int` or `unsigned short int` argument (the argument will have been promoted according to the integer promotions, but its value is converted to `short int` or `unsigned short int` before printing); or that a following `n` conversion specifier applies to a pointer to a `short int` argument.
`l`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long int` or `unsigned long int` argument; that a following `n` conversion specifier applies to a pointer to a `long int` argument; or has no effect on a following `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.
`ll`	Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long int` or `unsigned long long int` argument; that a following `n` conversion specifier applies to a pointer to a `long long int` argument.
`L`	Specifies that a following `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

4.16.1.4 Conversion specifiers

The conversion specifiers and their meanings are:

Flag	Description
`d`, `i`	The argument is converted to signed decimal in the style [-]*dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.
`o`, `u`, `x`, `X`	The unsigned argument is converted to unsigned octal for `o`, unsigned decimal for `u`, or unsigned hexadecimal notation for `x` or `X` in the style

Flag	Description
	dddd the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.
`f`, `F`	A double argument representing a floating-point number is converted to decimal notation in the style [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. A double argument representing an infinity is converted to `inf`. A double argument representing a NaN is converted to `nan`. The `F` conversion specifier produces `INF` or `NAN` instead of `inf` or `nan`, respectively.
`e`, `E`	A double argument representing a floating-point number is converted in the style [-]*d.ddd*e±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero. A double argument representing an infinity is converted to `inf`. A double argument representing a NaN is converted to `nan`. The `E` conversion specifier produces `INF` or `NAN` instead of `inf` or `nan`, respectively.
`g`, `G`	A double argument representing a floating-point number is converted in style `f` or `e` (or in style `F` or `e` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted; style `e` (or `E`) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the `#` flag is specified; a decimal-point character appears only if it is followed by a digit. A double argument representing an infinity is converted to `inf`. A double argument representing a NaN is converted to `nan`. The `G` conversion specifier produces `INF` or `NAN` instead of `inf` or `nan`, respectively.
`c`	The argument is converted to an `unsigned char`, and the resulting character is written.
`s`	The argument is be a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null character.
`p`	The argument is a pointer to `void`. The value of the pointer is converted in the same format as the `x` conversion specifier with a fixed precision of `2*sizeof(void *)`.
`n`	The argument is a pointer to a signed integer into which is *written* the number of characters written to the output stream so far by the

Flag	Description
	call to the formatting function. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.
`%`	A `%` character is written. No argument is converted.

Note that the C99 width modifier `l` used in conjunction with the `c` and `s` conversion specifiers is not supported and nor are the conversion specifiers `a` and `A`.

4.16.2 Formatted input control strings

The format is composed of zero or more directives: one or more white-space characters, an ordinary character (neither `%` nor a white-space character), or a conversion specification.

Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional nonzero decimal integer that specifies the maximum field width (in characters).
- An optional length modifier that specifies the size of the receiving object.
- A conversion specifier character that specifies the type of conversion to be applied.

The formatted input function executes each directive of the format in turn. If a directive fails, the function returns. Failures are described as input failures (because of the occurrence of an encoding error or the unavailability of input characters), or matching failures (because of inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

- Input white-space characters (as specified by the `isspace()` function) are skipped, unless the specification includes a `[`, `c`, or `n` specifier.
- An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.
- Except in the case of a `%` specifier, the input item (or, in the case of a %n directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

4.16.2.1 Length modifiers

The length modifiers and their meanings are:

Flag	Description
hh	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to signed char or pointer to unsigned char.
h	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to short int or unsigned short int.
l	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to long int or unsigned long int; that a following e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to double.
ll	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to long long int or unsigned long long int.
L	Specifies that a following e, E, f, F, g, or G conversion specifier applies to an argument with with type pointer to long double.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers j, z, and t are not supported.

4.16.2.2 Conversion specifiers

Flag	Description
d	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the strtol() function with the value 10 for the base argument. The corresponding argument must be a pointer to signed integer.
i	Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the strtol() function with the value zero for the base argument. The corresponding argument must be a pointer to signed integer.
o	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the strtol() function with the value 18 for the base argument. The corresponding argument must be a pointer to signed integer.
u	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the strtoul() function with the value 10 for the base argument. The corresponding argument must be a pointer to unsigned integer.
x	Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the strtoul() function with the value 16 for the base argument. The corresponding argument must be a pointer to unsigned integer.
e, f, g	Matches an optionally signed floating-point number whose format is the same as expected for the subject sequence of the strtod() function. The corresponding argument shall be a pointer to floating.
c	Matches a sequence of characters of exactly the number specified by the field width (one if no field width is present in the directive). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.
s	Matches a sequence of non-white-space characters The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

Flag	Description
`[`	Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket `]`. The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex `^`, in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or`[^]`, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a `-` character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, it is treated as a member of the scanset.
p	Reads a sequence output by the corresponding `%p` formatted output conversion. The corresponding argument must be a pointer to a pointer to `void`.
n	No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the formatted input function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the fscanf function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
`%`	Matches a single `%` character; no conversion or assignment occurs.

Note that the C99 width modifier `l` used in conjunction with the `c`, `s`, and `[` conversion specifiers is not supported and nor are the conversion specifiers `a` and `A`.

4.16.3 File functions

Function	Description
`fopen()`	Open file.
`freopen()`	Reopen file.
`fread()`	Read from file.
`fwrite()`	Write to file.
`fclose()`	Close file.
`feof()`	Test end-of-file indicator.
`ferror()`	Test error indicator.
`fflush()`	Flush file.
`clearerr()`	Clear error and end-of-file indicator on file.
`fsetpos()`	Set file position.
`fgetpos()`	Get file position.
`fseek()`	Set file position.
`ftell()`	Get file position.
`rewind()`	Rewind file.
`rename()`	Rename file.
`remove()`	Remove file.
`tmpnam()`	Generate name for temporary file.
`tmpfile()`	Generate temporary file.

4.16.3.1 fopen()

Description

Open file.

Prototype

```
FILE *fopen(const char * filename,
            const char * mode);
```

Parameters

Parameter	Description
`filename`	Pointer to zero-terminated file name.
`mode`	Pointer to zero-terminated file `mode`.

Return value

= `NULL`	File not opened.
≠ `NULL`	File opened.

Thread safety

Unsafe.

4.16.3.2 freopen()

Description

Reopen file.

Prototype

```
FILE *freopen(const char * filename,
              const char * mode,
                    FILE * stream);
```

Parameters

Parameter	Description
filename	Pointer to zero-terminated file name.
mode	Pointer to zero-terminated file mode.
stream	Pointer to file to reopen.

Return value

= NULL	File not reopened.
≠ NULL	File reopened.

Thread safety

Unsafe.

4.16.3.3 fread()

Description

Read from file.

Prototype

```
size_t fread(void    * ptr,
                       size_t size,
             size_t    nmemb,
             FILE    * stream);
```

Parameters

Parameter	Description
ptr	Pointer to object to write to.
size	Size of each element to read.
nmemb	Number of elements to read.
stream	Pointer to file to read from.

Return value

The number of elements successfully read, which may be less than nmemb if a read error or end-of-file is encountered.

Additional information

If size or nmemb is zero, fread() returns zero and the contents of the array and the state of the stream remain unchanged.

Thread safety

Unsafe.

4.16.3.4 fwrite()

Description

Write to file.

Prototype

```
size_t fwrite(const void   * ptr,
                             size_t size,
                    size_t   nmemb,
                    FILE   * stream);
```

Parameters

Parameter	Description
`ptr`	Pointer to data to write.
`size`	Size of each element to write.
`nmemb`	Number of elements to write.
`stream`	Pointer to file to write to.

Return value

The number of elements successfully written, which may be less than `nmemb` if a read error or end-of-file is encountered.

Additional information

If `size` or `nmemb` is zero, `fwrite()` returns zero and the contents of the array and the state of the `stream` remain unchanged.

Thread safety

Unsafe.

4.16.3.5 fclose()

Description

Close file.

Prototype

```
int fclose(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file to close.

Return value

0	File successfully closed.
EOF	File did not successfully close.

Thread safety

Unsafe.

4.16.3.6 feof()

Description

Test end-of-file indicator.

Prototype

```
int feof(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file to test.

Return value

= 0	No end-of-file on file.
≠ 0	End-of-file on file.

Thread safety

Unsafe.

4.16.3.7 ferror()

Description

Test error indicator.

Prototype

```
int ferror(FILE * stream);
```

Parameters

Parameter	Description
stream	Pointer to file to test.

Return value

= 0	No error on file.
≠ 0	Error on file.

Thread safety

Unsafe.

4.16.3.8 fflush()

Description

Flush file.

Prototype

```
int fflush(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file to flush, or `NULL`, indicating all files.

Return value

= 0	File (or all files) successfully flushed.
≠ EOF	Error flushing one or more files.

Additional information

If `stream` points to file in write or update mode where the most-recent operation was not input, any unwritten data for that file is delivered to the host environment to be written; otherwise, the behavior is undefined.

Thread safety

Unsafe.

4.16.3.9 clearerr()

Description

Clear error and end-of-file indicator on file.

Prototype

```
void clearerr(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file to clear indicators on.

Thread safety

Unsafe.

4.16.3.10 fsetpos()

Description

Set file position.

Prototype

```
int fsetpos(      FILE   * stream,
            const fpos_t * pos);
```

Parameters

Parameter	Description
stream	Pointer to file to position.
pos	Pointer to position.

Return value

= 0	Position set successfully.
≠ 0	Position not set successfully; errno set to ESPIPE.

Additional information

Sets the file position to `pos` which was previously retrieved using `fgetpos()`.

Thread safety

Unsafe.

4.16.3.11 fgetpos()

Description

Get file position.

Prototype

```
int fgetpos(FILE   * stream,
            fpos_t * pos);
```

Parameters

Parameter	Description
stream	Pointer to file to position.
pos	Pointer to object that receives the position.

Return value

= 0	Position retrieved successfully.
≠ 0	Position not retrieved successfully; errno set to ESPIPE.

Thread safety

Unsafe.

4.16.3.12 fseek()

Description

Set file position.

Prototype

```
int fseek(FILE * stream,
          long   offset,
          int    whence);
```

Parameters

Parameter	Description
`stream`	Pointer to file to position.
`offset`	Offset relative to anchor specified by `whence`.
`whence`	Where `offset` is relative to.

Return value

= 0	Position is set.
≠ 0	Position is not set.

Thread safety

Unsafe.

4.16.3.13 ftell()

Description

Get file position.

Prototype

```
long ftell(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file.

Return value

= 0	Position set successfully.
≠ 0	Position not set successfully; errno set to ESPIPE.

Additional information

Sets the file position to pos which was previously retrieved using `fgetpos()`.

Thread safety

Unsafe.

4.16.3.14 rewind()

Description

Rewind file.

Prototype

```
void rewind(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file to rewind.

Additional information

Sets the file position to start of file.

Thread safety

Unsafe.

4.16.3.15 rename()

Description

Rename file.

Prototype

```
int rename(const char * oldname,
           const char * newname);
```

Parameters

Parameter	Description
oldname	Pointer to string denoting old file name.
newname	Pointer to string denoting new file name.

Return value

= 0	Rename succeeded.
≠ 0	Rename failed.

Thread safety

Unsafe.

4.16.3.16 remove()

Description

Remove file.

Prototype

```
int remove(const char * filename);
```

Parameters

Parameter	Description
`filename`	Pointer to string denoting file name to remove.

Return value

= 0	Remove succeeded.
≠ 0	Remove failed.

Thread safety

Unsafe.

4.16.3.17 tmpnam()

Description

Generate name for temporary file.

Prototype

```
char *tmpnam(char * s);
```

Parameters

Parameter	Description
s	Pointer to object that receives the temporary file name, or NULL indicating that a (shared) internal buffer is used for the temporary name.

Return value

= NULL	Cannot generate a unique temporary name.
≠ NULL	Pointer to temporary name generated.

Thread safety

Unsafe.

4.16.3.18 tmpfile()

Description

Generate temporary file.

Prototype

```
FILE *tmpfile(void);
```

Return value

= NULL	Cannot generate a unique temporary file.
≠ NULL	Pointer to temporary file.

Thread safety

Unsafe.

4.16.4 Character and string I/O functions

Function	Description
getc()	Read character from stream.
fgetc()	Read character from file.
getchar()	Read character from standard input.
gets()	Read string from standard input.
fgets()	Read string from stream.
putc()	Write character to file.
fputc()	Write character to file.
putchar()	Write character to standard output.
puts()	Write string to standard output.
fputs()	Write string to standard output.
ungetc()	Push character back to file.

4.16.4.1 getc()

Description

Read character from stream.

Prototype

```
int getc(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file to read from.

Return value

If the `stream` is at end-of-file or a read error occurs, returns EOF, otherwise a nonnegative value.

Additional information

Reads a single character from a stream.

Thread safety

Unsafe.

4.16.4.2 fgetc()

Description

Read character from file.

Prototype

```
int fgetc(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file to read from.

Return value

If the end-of-file indicator for the `stream` is set, or if the `stream` is at end of file, the end-of-file indicator for the file is set and the fgetc function returns EOF. Otherwise, return the next character from the file pointed to by `stream`. If a read error occurs, the error indicator for the `stream` is set and return EOF.

Additional information

If the end-of-file indicator for the input `stream` pointed to by `stream` is not set and a next character is present, obtain that character as an unsigned char converted to an int and advance the associated file position.

Thread safety

Unsafe.

4.16.4.3 getchar()

Description

Read character from standard input.

Prototype

```
int getchar(void);
```

Return value

If the stream is at end-of-file or a read error occurs, returns EOF, otherwise a nonnegative value.

Additional information

Reads a single character from the standard input stream.

Thread safety

Unsafe.

4.16.4.4 gets()

Description

Read string from standard input.

Prototype

```
char *gets(char * s);
```

Parameters

Parameter	Description
s	Pointer to object that receives the string.

Return value

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 null pointer is return.

Additional information

This function reads characters from standard input into the array pointed to by s until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array.

Thread safety

Unsafe.

4.16.4.5 fgets()

Description

Read string from stream.

Prototype

```
char *fgets(char * s,
            int    n,
            FILE * stream);
```

Parameters

Parameter	Description
s	Pointer to object to write to.
n	Number of bytes to read.
stream	Pointer to file to read from.

Return value

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.

Additional information

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

Thread safety

Unsafe.

4.16.4.6 putc()

Description

Write character to file.

Prototype

```
int putc(int    c,
         FILE * stream);
```

Parameters

Parameter	Description
c	Character to write.
stream	Pointer to stream to write to.

Return value

If no error, the character written. If a write error occurs, returns EOF.

Additional information

Writes the character c to stream.

Thread safety

Unsafe.

4.16.4.7 fputc()

Description

Write character to file.

Prototype

```
int fputc(int    c,
          FILE * stream);
```

Parameters

Parameter	Description
c	Character to write.
stream	Pointer to file to write to.

Return value

If no error, the character written. If a write error occurs, returns EOF.

Additional information

Writes the character c to stream.

Thread safety

Unsafe.

4.16.4.8 putchar()

Description

Write character to standard output.

Prototype

```
int putchar(int c);
```

Parameters

Parameter	Description
c	Character to write.

Return value

If no error, the character written. If a write error occurs, returns EOF.

Additional information

Writes the character c to the standard output stream.

Thread safety

Unsafe.

4.16.4.9 puts()

Description

Write string to standard output.

Prototype

```
int puts(const char * s);
```

Parameters

Parameter	Description
s	Pointer to zero-terminated string.

Return value

Returns EOF if a write error occurs; otherwise it returns a nonnegative value.

Additional information

Writes the string pointed to by s to the standard output stream using putchar() and appends a newline character to the output. The terminating null character is not written.

Thread safety

Unsafe.

4.16.4.10 fputs()

Description

Write string to standard output.

Prototype

```
int fputs(const char * s,
                FILE * stream);
```

Parameters

Parameter	Description
s	Pointer to zero-terminated string.
stream	Pointer to file to write to.

Return value

Returns EOF if a write error occurs; otherwise returns a nonnegative value.

Additional information

Write the string pointed to by s to the file pointed to by stream. The terminating null character is not written.

Thread safety

Unsafe.

4.16.4.11 ungetc()

Description

Push character back to file.

Prototype

```
int ungetc(int    c,
           FILE * stream);
```

Parameters

Parameter	Description
c	Character to push back to file.
stream	File to push character to.

Return value

= EOF	Failed to push character back.
≠ EOF	The character pushed back to the file.

Additional information

This function pushes the character c back to the file stream so that it can be read again. If c is EOF, the function fails and EOF is returned. One character of pushback is guaranteed; if more than one character is pushed back without an intervening read, the pushback may fail.

Thread safety

Unsafe.

4.16.4.12 rewind()

Description

Rewind file.

Prototype

```
void rewind(FILE * stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file to rewind.

Additional information

Sets the file position to start of file.

Thread safety

Unsafe.

4.16.4.13 rename()

Description

Rename file.

Prototype

```
int rename(const char * oldname,
           const char * newname);
```

Parameters

Parameter	Description
oldname	Pointer to string denoting old file name.
newname	Pointer to string denoting new file name.

Return value

= 0	Rename succeeded.
≠ 0	Rename failed.

Thread safety

Unsafe.

4.16.4.14 remove()

Description

Remove file.

Prototype

```
int remove(const char * filename);
```

Parameters

Parameter	Description
`filename`	Pointer to string denoting file name to remove.

Return value

= 0	Remove succeeded.
≠ 0	Remove failed.

Thread safety

Unsafe.

4.16.5 Formatted input functions

Function	Description
`scanf()`	Formatted read from standard input.
`scanf_l()`	Formatted read from standard input, with locale.
`sscanf()`	Formatted read from string.
`sscanf_l()`	Formatted read from string, with locale.
`vscanf()`	Formatted read from standard input, variadic.
`vscanf_l()`	Formatted read from standard input, variadic, with locale.
`vsscanf()`	Formatted read from string, variadic.
`vsscanf_l()`	Formatted read from string, variadic, with local.
`fscanf()`	Formatted read from file.
`fscanf_l()`	Formatted read from file, with locale.
`vfscanf()`	Formatted read from file, variadic.
`vfscanf_l()`	Formatted read from file, variadic, with locale.

4.16.5.1 scanf()

Description

Formatted read from standard input.

Prototype

```
int scanf(const char * format,
                       ...);
```

Parameters

Parameter	Description
`format`	Pointer to zero-terminated `format` control string.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from standard input under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.5.2 scanf_l()

Description

Formatted read from standard input, with locale.

Prototype

```
int scanf_l(              locale_t loc,
            const char * format,
                          ...);
```

Parameters

Parameter	Description
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from standard input under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.5.3 sscanf()

Description

Formatted read from string.

Prototype

```
int sscanf(const char * s,
           const char * format,
                        ...);
```

Parameters

Parameter	Description
`s`	Pointer to string to read from.
`format`	Pointer to zero-terminated `format` control string.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the string `s` under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Safe [if configured].

4.16.5.4 sscanf_l()

Description

Formatted read from string, with locale.

Prototype

```
int sscanf_l(const char * s,
                          locale_t loc,
             const char * format,
                          ...);
```

Parameters

Parameter	Description
s	Pointer to string to read from.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the string s under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Safe.

4.16.5.5 vscanf()

Description

Formatted read from standard input, variadic.

Prototype

```
int vscanf(const char    * format,
                 va_list   arg);
```

Parameters

Parameter	Description
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the standard input stream under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling vscanf(), arg must be initialized by the va_start() macro (and possibly subsequent va_arg() calls). vscanf() does not invoke the va_end() macro.

If there are insufficient arguments for the format, the behavior is undefined.

Thread safety

Unsafe.

4.16.5.6 vscanf_l()

Description

Formatted read from standard input, variadic, with locale.

Prototype

```
int vscanf_l(                   locale_t loc,
             const char       * format,
                   va_list      arg);
```

Parameters

Parameter	Description
`loc`	Locale to use for conversion.
`format`	Pointer to zero-terminated `format` control string.
`arg`	Variable parameter list.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the standard input stream under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling `vscanf()`, `arg` must be initialized by the `va_start()` macro (and possibly subsequent `va_arg()` calls). `vscanf()` does not invoke the `va_end()` macro.

If there are insufficient arguments for the `format`, the behavior is undefined.

Thread safety

Unsafe.

4.16.5.7 vsscanf()

Description

Formatted read from string, variadic.

Prototype

```
int vsscanf(const char    * s,
            const char    * format,
                  va_list   arg);
```

Parameters

Parameter	Description
`s`	Pointer to string to read from.
`format`	Pointer to zero-terminated `format` control string.
`arg`	Variable parameter list.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the standard input stream under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling `vsscanf()`, `arg` must be initialized by the `va_start()` macro (and possibly subsequent `va_arg()` calls). `vsscanf()` does not invoke the `va_end()` macro.

If there are insufficient arguments for the `format`, the behavior is undefined.

Thread safety

Safe [if configured].

4.16.5.8 vsscanf_l()

Description

Formatted read from string, variadic, with local.

Prototype

```
int vsscanf_l(const char    * s,
                              locale_t loc,
              const char    * format,
                    va_list   arg);
```

Parameters

Parameter	Description
s	Pointer to string to read from.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the standard input stream under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling vsscanf(), arg must be initialized by the va_start() macro (and possibly subsequent va_arg() calls). vsscanf() does not invoke the va_end() macro.

If there are insufficient arguments for the format, the behavior is undefined.

Thread safety

Safe [if configured].

4.16.5.9 fscanf()

Description

Formatted read from file.

Prototype

```
int fscanf(      FILE * stream,
           const char * format,
                        ...);
```

Parameters

Parameter	Description
stream	Pointer to file to read from.
format	Pointer to zero-terminated format control string.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the file stream under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.5.10 fscanf_l()

Description

Formatted read from file, with locale.

Prototype

```
int fscanf_l(      FILE * stream,
                   locale_t loc,
             const char * format,
                   ...);
```

Parameters

Parameter	Description
stream	Pointer to file to read from.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the file stream under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.5.11 vfscanf()

Description

Formatted read from file, variadic.

Prototype

```
int vfscanf(      FILE    * stream,
            const char    * format,
                  va_list   arg);
```

Parameters

Parameter	Description
`stream`	Pointer to file to read from.
`format`	Pointer to zero-terminated `format` control string.
`arg`	Variable parameter list.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the file `stream` under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.5.12 vfscanf_l()

Description

Formatted read from file, variadic, with locale.

Prototype

```
int vfscanf_l(      FILE    * stream,
                              locale_t loc,
              const char    * format,
                    va_list   arg);
```

Parameters

Parameter	Description
stream	Pointer to file to read from.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

Returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Additional information

Reads input from the file stream under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.6 Formatted output functions

Function	Description
`printf()`	Formatted write to standard output.
`printf_l()`	Formatted write to standard output, with locale.
`sprintf()`	Formatted write to string.
`sprintf_l()`	Formatted write to string, with locale.
`snprintf()`	Formatted write to string, limit length.
`snprintf_l()`	Formatted write to string, limit length, with locale.
`vprintf()`	Formatted write to standard output, variadic.
`vprintf_l()`	Formatted write to standard output, variadic, with locale.
`vsprintf()`	Formatted write to string, variadic.
`vsprintf_l()`	Formatted write to string, variadic, with locale.
`vsnprintf()`	Formatted write to string, limit length, variadic.
`vsnprintf_l()`	Formatted write to string, limit length, variadic, with locale.
`fprintf()`	Formatted write to file.
`fprintf_l()`	Formatted write to file, with locale.
`vfprintf()`	Formatted write to file, variadic.
`vfprintf_l()`	Formatted write to file, variadic, with locale.
`asprintf()`	Print to newly allocated string.
`asprintf_l()`	Print to newly allocated string, with locale.
`vasprintf()`	Print to newly allocated string, variadic.
`vasprintf_l()`	Print to newly allocated string, variadic, with locale.

4.16.6.1 printf()

Description

Formatted write to standard output.

Prototype

```
int printf(const char * format,
                        ...);
```

Parameters

Parameter	Description
`format`	Pointer to zero-terminated `format` control string.

Return value

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

Additional information

Writes to the standard output stream under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.6.2 printf_l()

Description

Formatted write to standard output, with locale.

Prototype

```
int printf_l(             locale_t loc,
             const char * format,
                          ...);
```

Parameters

Parameter	Description
`loc`	Locale to use for conversion.
`format`	Pointer to zero-terminated `format` control string.

Return value

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

Additional information

Writes to the standard output stream under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.6.3 sprintf()

Description

Formatted write to string.

Prototype

```
int sprintf(      char * s,
            const char * format,
                         ...);
```

Parameters

Parameter	Description
s	Pointer to array that receives the formatted output.
format	Pointer to zero-terminated format control string.

Return value

Returns number of characters written to s (not counting the terminating null), or a negative value if an output or encoding error occurred.

Additional information

Writes to the string pointed to by s under control of the string pointed to by format that specifies how subsequent arguments are converted for output. A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

Thread safety

Safe [if configured].

4.16.6.4 sprintf_l()

Description

Formatted write to string, with locale.

Prototype

```
int sprintf_l(      char * s,
                    locale_t loc,
              const char * format,
                    ...);
```

Parameters

Parameter	Description
s	Pointer to array that receives the formatted output.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.

Return value

Returns number of characters written to s (not counting the terminating null), or a negative value if an output or encoding error occurred.

Additional information

Writes to the string pointed to by s under control of the string pointed to by format that specifies how subsequent arguments are converted for output. A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

Thread safety

Safe.

4.16.6.5 snprintf()

Description

Formatted write to string, limit length.

Prototype

```
int snprintf(      char   * s,
                   size_t   n,
             const char   * format,
                            ...);
```

Parameters

Parameter	Description
s	Pointer to array that receives the formatted output.
n	Maximum number of characters to write to the array pointed to by s.
format	Pointer to zero-terminated format control string.

Return value

Returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than n.

Additional information

Writes to the string pointed to by s under control of the string pointed to by format that specifies how subsequent arguments are converted for output.

If n is zero, nothing is written, and s can be a null pointer. Otherwise, output characters beyond count n-1 are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

Thread safety

Safe [if configured].

4.16.6.6 snprintf_l()

Description

Formatted write to string, limit length, with locale.

Prototype

```
int snprintf_l(      char   * s,
                     size_t   n,
                              locale_t loc,
               const char   * format,
                              ...);
```

Parameters

Parameter	Description
s	Pointer to array that receives the formatted output.
n	Maximum number of characters to write to the array pointed to by s.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.

Return value

Returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than n.

Additional information

Writes to the string pointed to by s under control of the string pointed to by format that specifies how subsequent arguments are converted for output.

If n is zero, nothing is written, and s can be a null pointer. Otherwise, output characters beyond count n-1 are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

Thread safety

Safe.

4.16.6.7 vprintf()

Description

Formatted write to standard output, variadic.

Prototype

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

Parameters

Parameter	Description
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

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

Additional information

Writes to the standard output stream using under control of the string pointed to by format that specifies how subsequent arguments are converted for output. Before calling vprintf(), arg must be initialized by the va_start macro (and possibly subsequent va_arg calls). vprintf() does not invoke the va_end macro.

Thread safety

Unsafe.

4.16.6.8 vprintf_l()

Description

Formatted write to standard output, variadic, with locale.

Prototype

```
int vprintf_l(                locale_t loc,
               const char    * format,
                     va_list   arg);
```

Parameters

Parameter	Description
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

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

Additional information

Writes to the standard output stream using under control of the string pointed to by format that specifies how subsequent arguments are converted for output. Before calling vprintf(), arg must be initialized by the va_start macro (and possibly subsequent va_arg calls). vprintf() does not invoke the va_end macro.

Thread safety

Unsafe.

4.16.6.9 vsprintf()

Description

Formatted write to string, variadic.

Prototype

```
int vsprintf(      char    * s,
             const char    * format,
                   va_list   arg);
```

Parameters

Parameter	Description
`s`	Pointer to array that receives the formatted output.
`format`	Pointer to zero-terminated `format` control string.
`arg`	Variable parameter list.

Return value

Returns number of characters written to `s` (not counting the terminating null), or a negative value if an output or encoding error occurred.

Additional information

Writes to the string pointed to by `s` under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output. A null character is written at the end of the characters written; it is not counted as part of the returned value.

Before calling `vsprintf()`, `arg` must be initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). `vsprintf()` does not invoke the `va_end` macro.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

Notes

This is equivalent to `sprintf()` with the variable argument list replaced by arg.

Thread safety

Safe [if configured].

4.16.6.10 vsprintf_l()

Description

Formatted write to string, variadic, with locale.

Prototype

```
int vsprintf_l(      char    * s,
                               locale_t loc,
               const char    * format,
                     va_list   arg);
```

Parameters

Parameter	Description
s	Pointer to array that receives the formatted output.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

Returns number of characters written to s (not counting the terminating null), or a negative value if an output or encoding error occurred.

Additional information

Writes to the string pointed to by s under control of the string pointed to by format that specifies how subsequent arguments are converted for output. A null character is written at the end of the characters written; it is not counted as part of the returned value.

Before calling vsprintf(), arg must be initialized by the va_start macro (and possibly subsequent va_arg calls). vsprintf() does not invoke the va_end macro.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

Notes

This is equivalent to sprintf() with the variable argument list replaced by arg.

Thread safety

Safe.

4.16.6.11 vsnprintf()

Description

Formatted write to string, limit length, variadic.

Prototype

```
int vsnprintf(       char     * s,
                     size_t     n,
               const char     * format,
                     va_list    arg);
```

Parameters

Parameter	Description
s	Pointer to array that receives the formatted output.
n	Maximum number of characters to write to the array pointed to by s.
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

Returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than n.

Additional information

Writes to the string pointed to by s under control of the string pointed to by format that specifies how subsequent arguments are converted for output. Before calling vsnprintf(), arg must be initialized by the va_start macro (and possibly subsequent va_arg() calls). vsnprintf() does not invoke the va_end macro.

If n is zero, nothing is written, and s can be a null pointer. Otherwise, output characters beyond count n-1 are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

Notes

This is equivalent to snprintf() with the variable argument list replaced by arg.

Thread safety

Safe [if configured].

4.16.6.12 vsnprintf_l()

Description

Formatted write to string, limit length, variadic, with locale.

Prototype

```
int vsnprintf_l(      char     * s,
                      size_t     n,
                                 locale_t loc,
                const char     * format,
                      va_list    arg);
```

Parameters

Parameter	Description
s	Pointer to array that receives the formatted output.
n	Maximum number of characters to write to the array pointed to by s.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

Returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than n.

Additional information

Writes to the string pointed to by s under control of the string pointed to by format that specifies how subsequent arguments are converted for output. Before calling vsnprintf(), arg must be initialized by the va_start macro (and possibly subsequent va_arg() calls). vsnprintf() does not invoke the va_end macro.

If n is zero, nothing is written, and s can be a null pointer. Otherwise, output characters beyond count n-1 are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

Notes

This is equivalent to snprintf() with the variable argument list replaced by arg.

Thread safety

Safe.

4.16.6.13 fprintf()

Description

Formatted write to file.

Prototype

```
int fprintf(      FILE * stream,
            const char * format,
                         ...);
```

Parameters

Parameter	Description
stream	Pointer to file to write to.
format	Pointer to zero-terminated format control string.

Return value

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

Additional information

Writes to the file stream under control of the string pointed to by format that specifies how subsequent arguments are converted for output.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.6.14 fprintf_l()

Description

Formatted write to file, with locale.

Prototype

```
int fprintf_l(      FILE * stream,
                           locale_t loc,
              const char * format,
                           ...);
```

Parameters

Parameter	Description
stream	Pointer to file to write to.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.

Return value

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

Additional information

Writes to the file stream under control of the string pointed to by format that specifies how subsequent arguments are converted for output.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Thread safety

Unsafe.

4.16.6.15 vfprintf()

Description

Formatted write to file, variadic.

Prototype

```
int vfprintf(      FILE    * stream,
             const char    * format,
                   va_list   arg);
```

Parameters

Parameter	Description
`stream`	Pointer to file to write to.
`format`	Pointer to zero-terminated `format` control string.
`arg`	Variable parameter list.

Return value

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

Additional information

Writes to the file `stream` using under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output. Before calling `vfprintf()`, `arg` must be initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). `vfprintf()` does not invoke the `va_end` macro.

Thread safety

Unsafe.

4.16.6.16 vfprintf_l()

Description

Formatted write to file, variadic, with locale.

Prototype

```
int vfprintf_l(      FILE    * stream,
                               locale_t loc,
               const char    * format,
                     va_list   arg);
```

Parameters

Parameter	Description
stream	Pointer to file to write to.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.
arg	Variable parameter list.

Return value

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

Additional information

Writes to the file stream using under control of the string pointed to by format that specifies how subsequent arguments are converted for output. Before calling vfprintf(), arg must be initialized by the va_start macro (and possibly subsequent va_arg calls). vfprintf() does not invoke the va_end macro.

Thread safety

Unsafe.

4.16.6.17 asprintf()

Description

Print to newly allocated string.

Prototype

```
int asprintf(      char ** strp,
             const char  * format,
                           ...);
```

Parameters

Parameter	Description
`strp`	Pointer to object that receives the pointer to the output string.
`format`	Pointer to zero-terminated `format` control string.

Return value

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

Additional information

Writes to a newly allocated string, using `malloc()` and `realloc()` if necessary, under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output.

The pointer to the newly allocated stirng is assigned to the object pointed to by `strp`. It is the client's responsibility to free this pointer.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Notes

Commonly found in Linux, BSD, and GNU C libraries.

Thread safety

Safe [if configured].

4.16.6.18 asprintf_l()

Description

Print to newly allocated string, with locale.

Prototype

```
int asprintf_l(      char ** strp,
                            locale_t loc,
               const char  * format,
                            ...);
```

Parameters

Parameter	Description
strp	Pointer to object that receives the pointer to the output string.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.

Return value

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

Additional information

Writes to a newly allocated string, using malloc() and realloc() if necessary, under control of the string pointed to by format that specifies how subsequent arguments are converted for output.

The pointer to the newly allocated stirng is assigned to the object pointed to by strp. It is the client’s responsibility to free this pointer.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Notes

Commonly found in Linux, BSD, and GNU C libraries.

Thread safety

Safe.

4.16.6.19 vasprintf()

Description

Print to newly allocated string, variadic.

Prototype

```
int vasprintf(      char    ** strp,
              const char     * format,
                    va_list    ap);
```

Parameters

Parameter	Description
`strp`	Pointer to object that receives the pointer to the output string.
`format`	Pointer to zero-terminated `format` control string.
`ap`	Variadic argument list.

Return value

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

Additional information

Writes to a newly allocated string, using `malloc()` and `realloc()` if necessary, under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output.

The pointer to the newly allocated stirng is assigned to the object pointed to by `strp`. It is the client's responsibility to free this pointer.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Notes

Commonly found in Linux, BSD, and GNU C libraries.

Thread safety

Safe [if configured].

4.16.6.20 vasprintf_l()

Description

Print to newly allocated string, variadic, with locale.

Prototype

```
int vasprintf_l(      char     ** strp,
                                 locale_t loc,
                const char      * format,
                      va_list     ap);
```

Parameters

Parameter	Description
strp	Pointer to object that receives the pointer to the output string.
loc	Locale to use for conversion.
format	Pointer to zero-terminated format control string.
ap	Variadic argument list.

Return value

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

Additional information

Writes to a newly allocated string, using `malloc()` and `realloc()` if necessary, under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output.

The pointer to the newly allocated stirng is assigned to the object pointed to by `strp`. It is the client's responsibility to free this pointer.

If there are insufficient arguments for the `format`, the behavior is undefined. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Notes

Commonly found in Linux, BSD, and GNU C libraries.

Thread safety

Safe.

4.16.7 Miscellaneous functions

Function	Description
`perror()`	Print error message to standard error stream.

4.16.7.1 perror()

Description

Print error message to standard error stream.

Prototype

```
void perror(const char * s);
```

Parameters

Parameter	Description
s	Pointer to string to prefix error with. May be `NULL`.

Thread safety

Unsafe.

4.17 <stdlib.h>

4.17.1 Process control functions

Function	Description
`atexit()`	Set function to be called on exit.
`abort()`	Abort execution.

4.17.1.1 atexit()

Description

Set function to be called on exit.

Prototype

```
int atexit(__SEGGER_RTL_exit_func fn);
```

Parameters

Parameter	Description
`fn`	Function to register.

Return value

= 0	Success registering function.
≠ 0	Did not register function.

Additional information

Registers function `fn` to be called when the application has exited. The functions registered with `atexit()` are executed in reverse order of their registration.

Thread safety

Unsafe.

4.17.1.2 abort()

Description

Abort execution.

Prototype

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

Additional information

Calls exit() with the exit status `EXIT_FAILURE`.

Thread safety

Not applicable.

4.17.2 Runtime constraint functions

Function	Description
`abort_handler_s()`	Abort on runtime constraint violation (C11).
`ignore_handler_s()`	Ignore runtime constraint violations (C11).
`set_constraint_handler_s()`	Set runtime constraint handler (C11).

4.17.2.1 abort_handler_s()

Description

Abort on runtime constraint violation (C11).

Prototype

```
void abort_handler_s(const char    * msg,
                           void    * ptr,
                           errno_t   error);
```

Parameters

Parameter	Description
msg	Pointer to a null-terminated character string describing the constraint violation.
ptr	A null pointer or a pointer to an implementation-defined object. This implementation always passes NULL.
error	If the function calling the handler has a return type declared as errno_t, the return value of the function is passed. Otherwise, a positive value of type errno_t is passed.

Additional information

The macro __STDC_WANT_LIB_EXT1__ must be set to 1 before including <stdlib.h> to access this function.

Conformance

ISO/IEC 9899:2011 (C11).

Thread safety

Not applicable.

4.17.2.2 ignore_handler_s()

Description

Ignore runtime constraint violations (C11).

Prototype

```
void ignore_handler_s(const char    * msg,
                            void    * ptr,
                            errno_t   error);
```

Parameters

Parameter	Description
msg	Pointer to a null-terminated character string describing the constraint violation.
ptr	A null pointer or a pointer to an implementation-defined object. This implementation always passes NULL.
error	If the function calling the handler has a return type declared as errno_t, the return value of the function is passed. Otherwise, a positive value of type errno_t is passed.

Additional information

The macro __STDC_WANT_LIB_EXT1__ must be set to 1 before including <stdlib.h> to access this function.

Conformance

ISO/IEC 9899:2011 (C11).

Thread safety

Safe.

4.17.2.3 set_constraint_handler_s()

Description

Set runtime constraint `handler` (C11).

Prototype

```
constraint_handler_t set_constraint_handler_s(constraint_handler_t handler);
```

Parameters

Parameter	Description
`handler`	Pointer to function that will handle runtime constraint violations. If `NULL`, the implementation-default abort `handler` is used.

Return value

The previously-set constraint `handler`. If the previous `handler` was registered by calling `set_constraint_handler()` with a null pointer argument, a pointer to the implementation-default `handler` is returned (not `NULL`).

Additional information

The macro `__STDC_WANT_LIB_EXT1__` must be set to 1 before including <stdlib.h> to access this function.

Conformance

ISO/IEC 9899:2011 (C11).

Thread safety

Unsafe.

4.17.3 Integer arithmetic functions

Function	Description
`abs()`	Calculate absolute value, int.
`labs()`	Calculate absolute value, long.
`llabs()`	Calculate absolute value, long long.
`div()`	Divide returning quotient and remainder, int.
`ldiv()`	Divide returning quotient and remainder, long.
`lldiv()`	Divide returning quotient and remainder, long long.

4.17.3.1 abs()

Description

Calculate absolute value, int.

Prototype

```
int abs(int Value);
```

Parameters

Parameter	Description
`Value`	Integer value.

Return value

The absolute value of the integer argument Value.

Thread safety

Safe.

4.17.3.2 labs()

Description

Calculate absolute value, long.

Prototype

```
long int labs(long int Value);
```

Parameters

Parameter	Description
`Value`	Long integer value.

Return value

The absolute value of the long integer argument Value.

Thread safety

Safe.

4.17.3.3 llabs()

Description

Calculate absolute value, long long.

Prototype

```
long long int llabs(long long int Value);
```

Parameters

Parameter	Description
`Value`	Long long integer value.

Return value

The absolute value of the long long integer argument Value.

Thread safety

Safe.

4.17.3.4 div()

Description

Divide returning quotient and remainder, int.

Prototype

```
div_t div(int Numer,
          int Denom);
```

Parameters

Parameter	Description
Numer	Numerator.
Denom	Demoninator.

Return value

Returns a structure of type div_t comprising both the quotient and the remainder. The structures contain the members quot (the quotient) and rem (the remainder), each of which has the same type as the arguments Numer and Denom. If either part of the result cannot be represented, the behavior is undefined.

Additional information

This computes Numer divided by Denom and Numer modulo Denom in a single operation.

Thread safety

Safe.

See also

div_t

4.17.3.5 ldiv()

Description

Divide returning quotient and remainder, long.

Prototype

```
ldiv_t ldiv(long Numer,
            long Denom);
```

Parameters

Parameter	Description
Numer	Numerator.
Denom	Demoninator.

Return value

Returns a structure of type `ldiv_t` comprising both the quotient and the remainder. The structures contain the members quot (the quotient) and rem (the remainder), each of which has the same type as the arguments `Numer` and `Denom`. If either part of the result cannot be represented, the behavior is undefined.

Additional information

This computes `Numer` divided by `Denom` and `Numer` modulo `Denom` in a single operation.

Thread safety

Safe.

See also

`ldiv_t`

4.17.3.6 lldiv()

Description

Divide returning quotient and remainder, long long.

Prototype

```
lldiv_t lldiv(long long Numer,
              long long Denom);
```

Parameters

Parameter	Description
Numer	Numerator.
Denom	Demoninator.

Return value

Returns a structure of type lldiv_t comprising both the quotient and the remainder. The structures contain the members quot (the quotient) and rem (the remainder), each of which has the same type as the arguments Numer and Denom. If either part of the result cannot be represented, the behavior is undefined.

Additional information

This computes Numer divided by Denom and Numer modulo Denom in a single operation.

Thread safety

Safe.

See also

lldiv_t

4.17.4 Pseudo-random sequence generation functions

Function	Description
`rand()`	Return next random number in sequence.
`srand()`	Set seed of random number sequence.

4.17.4.1 rand()

Description

Return next random number in sequence.

Prototype

```
int rand(void);
```

Return value

Returns the computed pseudo-random integer.

Additional information

This computes a sequence of pseudo-random integers in the range 0 to `RAND_MAX`.

Thread safety

Safe [if configured].

See also

`srand()`

4.17.4.2 srand()

Description

Set seed of random number sequence.

Prototype

```
void srand(unsigned s);
```

Parameters

Parameter	Description
`s`	New seed value for pseudo-random sequence.

Additional information

This uses the argument Seed as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand()`. If `srand()` is called with the same seed value, the same sequence of pseudo-random numbers is generated.

If `rand()` is called before any calls to `srand()` have been made, a sequence is generated as if `srand()` is first called with a seed value of `1`.

Thread safety

Safe [if configured].

See also

`rand()`

4.17.5 Memory allocation functions

Function	Description
`malloc()`	Allocate space for single object.
`aligned_alloc()`	Allocate space for aligned single object.
`calloc()`	Allocate space for multiple objects and zero them.
`realloc()`	Resize or allocate memory space.
`free()`	Free allocated memory for reuse.

4.17.5.1 malloc()

Description

Allocate space for single object.

Prototype

```
void *malloc(size_t sz);
```

Parameters

Parameter	Description
sz	Number of characters to allocate for the object.

Return value

Returns a null pointer if the space for the object cannot be allocated from free memory; if space for the object can be allocated, `malloc()` returns a pointer to the start of the allocated space.

Additional information

Allocates space for an object whose size is specified by `sz` and whose value is indeterminate.

Thread safety

Safe [if configured].

4.17.5.2 aligned_alloc()

Description

Allocate space for aligned single object.

Prototype

```
void *aligned_alloc(size_t align,
                    size_t sz);
```

Parameters

Parameter	Description
align	Alignment of object.
sz	Number of characters to allocate for the object.

Return value

Returns a null pointer if the space for the object cannot be allocated from free memory or the alignment cannot be satisfied; if space for the object can be allocated, aligned_alloc() returns a pointer to the start of the allocated, aligned space.

Additional information

Allocates space for an object whose size is specified by sz, whose alignment is align, and whose value is indeterminate.

Thread safety

Safe [if configured].

4.17.5.3 calloc()

Description

Allocate space for multiple objects and zero them.

Prototype

```
void *calloc(size_t nobj,
             size_t sz);
```

Parameters

Parameter	Description
nobj	Number of objects to allocate.
sz	Number of characters to allocate per object.

Return value

Returns a null pointer if the space for the object cannot be allocated from free memory; if space for the object can be allocated, `calloc()` returns a pointer to the start of the allocated space.

Additional information

Allocates space for an array of `nobj` objects, each of whose size is `sz`. The space is initialized to all zero bits.

Thread safety

Safe [if configured].

4.17.5.4 realloc()

Description

Resize or allocate memory space.

Prototype

```
void *realloc(void   * ptr,
              size_t   sz);
```

Parameters

Parameter	Description
`ptr`	Pointer to resize, or `NULL` to allocate.
`sz`	New size of object.

Return value

Returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

Additional information

Deallocates the old object pointed to by `ptr` and returns a pointer to a new object that has the size specified by `sz`. The contents of the new object is identical to that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

If `ptr` is a null pointer, `realloc()` behaves like `malloc()` for the specified size. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

If `ptr` does not match a pointer earlier returned by `calloc()`, `malloc()`, or `realloc()`, or if the space has been deallocated by a call to `free()` or `realloc()`, the behavior is undefined.

Thread safety

Safe [if configured].

4.17.5.5 free()

Description

Free allocated memory for reuse.

Prototype

```
void free(void * ptr);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to free.

Additional information

Causes the space pointed to by `ptr` to be deallocated, that is, made available for further allocation. If `ptr` is a null pointer, no action occurs.

If `ptr` does not match a pointer earlier returned by `calloc()`, `malloc()`, or `realloc()`, or if the space has been deallocated by a call to `free()` or `realloc()`, the behavior is undefined.

Thread safety

Safe [if configured].

4.17.6 Search and sort functions

Function	Description
`qsort()`	Sort array.
`bsearch()`	Search sorted array.

4.17.6.1 qsort()

Description

Sort array.

Prototype

```
void qsort(void    * base,
           size_t    nmemb,
           size_t    sz,
           int       ( *compare)(const void * elem1 , const void * elem2 ));
```

Parameters

Parameter	Description
`base`	Pointer to the start of the array.
`nmemb`	Number of array elements.
`sz`	Number of characters per array element.
`compare`	Pointer to element comparison function.

Additional information

Sorts the array pointed to by `base` using the `compare` function. The array should have `nmemb` elements of `sz` bytes. The `compare` function should return a negative value if the first parameter is less than the second parameter, zero if the parameters are equal, and a positive value if the first parameter is greater than the second parameter.

Thread safety

Safe.

4.17.6.2 bsearch()

Description

Search sorted array.

Prototype

```
void *bsearch
            (const void   * key,
             const void   * base,
                   size_t   nmemb,
                   size_t   sz,
                   int      ( *compare)(const void * elem1 , const void * elem2 ));
```

Parameters

Parameter	Description
key	Pointer to object to search for.
base	Pointer to the start of the array.
nmemb	Number of array elements.
sz	Number of characters per array element.
compare	Pointer to element comparison function.

Return value

= NULL	Key not found.
≠ NULL	Pointer to found object.

Additional information

Searches the array pointed to by base for the specified key and returns a pointer to the first entry that matches, or null if no match. The array should have nmemb elements of sz bytes and be sorted by the same algorithm as the compare function.

The compare function should return a negative value if the first parameter is less than second parameter, zero if the parameters are equal, and a positive value if the first parameter is greater than the second parameter.

Thread safety

Safe.

4.17.7 Number to string conversions

Function	Description
`itoa()`	Convert to string, int.
`ltoa()`	Convert to string, long.
`lltoa()`	Convert to string, long long.
`utoa()`	Convert to string, unsigned.
`ultoa()`	Convert to string, unsigned long.
`ulltoa()`	Convert to string, unsigned long long.

4.17.7.1 itoa()

Description

Convert to string, int.

Prototype

```
char *itoa(int    val,
           char * buf,
           int    radix);
```

Parameters

Parameter	Description
val	Value to convert.
buf	Pointer to array of characters that receives the string.
radix	Number base to use for conversion, 2 to 36.

Return value

Returns buf.

Additional information

Converts val to a string in base radix and places the result in buf which must be large enough to hold the output. If radix is greater than 36, the result is undefined.

If val is negative and radix is 10, the string has a leading minus sign (-); for all other values of radix, value is considered unsigned and never has a leading minus sign.

Notes

This is a non-standard function. Even though this function is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases.

Thread safety

Safe.

See also

ltoa(), lltoa(), utoa(), ultoa(), ulltoa()

4.17.7.2 ltoa()

Description

Convert to string, long.

Prototype

```
char *ltoa(long   val,
           char * buf,
           int    radix);
```

Parameters

Parameter	Description
val	Value to convert.
buf	Pointer to array of characters that receives the string.
radix	Number base to use for conversion, 2 to 36.

Return value

Returns buf.

Additional information

Converts val to a string in base radix and places the result in buf which must be large enough to hold the output. If radix is greater than 36, the result is undefined.

If val is negative and radix is 10, the string has a leading minus sign (-); for all other values of radix, value is considered unsigned and never has a leading minus sign.

Notes

This is a non-standard function. Even though this function is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases.

Thread safety

Safe.

See also

itoa(), lltoa(), utoa(), ultoa(), ulltoa()

4.17.7.3 lltoa()

Description

Convert to string, long long.

Prototype

```
char *lltoa(long long   val,
            char      * buf,
            int         radix);
```

Parameters

Parameter	Description
val	Value to convert.
buf	Pointer to array of characters that receives the string.
radix	Number base to use for conversion, 2 to 36.

Return value

Returns buf.

Additional information

Converts val to a string in base radix and places the result in buf which must be large enough to hold the output. If radix is greater than 36, the result is undefined.

If val is negative and radix is 10, the string has a leading minus sign (-); for all other values of radix, value is considered unsigned and never has a leading minus sign.

Notes

This is a non-standard function. Even though this function is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases.

Thread safety

Safe.

See also

itoa(), ltoa(), utoa(), ultoa(), ulltoa()

4.17.7.4 utoa()

Description

Convert to string, unsigned.

Prototype

```
char *utoa(unsigned   val,
           char     * buf,
           int        radix);
```

Parameters

Parameter	Description
`val`	Value to convert.
`buf`	Pointer to array of characters that receives the string.
`radix`	Number base to use for conversion, 2 to 36.

Return value

Returns buf.

Additional information

Converts `val` to a string in base `radix` and places the result in `buf` which must be large enough to hold the output. If `radix` is greater than 36, the result is undefined.

Notes

This is a non-standard function. Even though this function is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases.

Thread safety

Safe.

See also

`itoa()`, `ltoa()`, `lltoa()`, `ultoa()`, `ulltoa()`

4.17.7.5 ultoa()

Description

Convert to string, unsigned long.

Prototype

```
char *ultoa(unsigned long   val,
            char          * buf,
            int             radix);
```

Parameters

Parameter	Description
val	Value to convert.
buf	Pointer to array of characters that receives the string.
radix	Number base to use for conversion, 2 to 36.

Return value

Returns buf.

Additional information

Converts val to a string in base radix and places the result in buf which must be large enough to hold the output. If radix is greater than 36, the result is undefined.

Notes

This is a non-standard function. Even though this function is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases.

Thread safety

Safe.

See also

itoa(), ltoa(), lltoa(), ulltoa(), utoa()

4.17.7.6 ulltoa()

Description

Convert to string, unsigned long long.

Prototype

```
char *ulltoa(unsigned long long   val,
             char               * buf,
             int                  radix);
```

Parameters

Parameter	Description
val	Value to convert.
buf	Pointer to array of characters that receives the string.
radix	Number base to use for conversion, 2 to 36.

Return value

Returns buf.

Additional information

Converts val to a string in base radix and places the result in buf which must be large enough to hold the output. If radix is greater than 36, the result is undefined.

Notes

This is a non-standard function. Even though this function is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases.

Thread safety

Safe.

See also

itoa(), ltoa(), lltoa(), ultoa(), utoa()

4.17.8 String to number conversions

Function	Description
atoi()	Convert to number, int.
atol()	Convert to number, long.
atoll()	Convert to number, long long.
atof()	Convert to number, double.
strtol()	Convert to number, long.
strtoll()	Convert to number, long long.
strtoul()	Convert to number, unsigned long.
strtoull()	Convert to number, unsigned long long.
strtof()	Convert to number, float.
strtod()	Convert to number, double.
strtold()	Convert to number, long double.
wcstol()	Convert to number, long.
wcstoll()	Convert to number, long long.
wcstoul()	Convert to number, unsigned long.
wcstoull()	Convert to number, unsigned long long.
wcstof()	Convert to number, float.
wcstod()	Convert to number, double.
wcstold()	Convert to number, long double.

4.17.8.1 atoi()

Description

Convert to number, int.

Prototype

```
int atoi(const char * nptr);
```

Parameters

Parameter	Description
`nptr`	Pointer to string to convert from.

Return value

Returns the converted value, if any. If the value of the result cannot be represented, the behavior is undefined.

Additional information

Converts the initial portion of the string pointed to by `nptr` to an int representation.

`atoi()` does not affect the value of errno on an error.

Notes

Except for the behavior on error, `atoi()` is equivalent to (int)strtol(nptr, `NULL`, 10).

Thread safety

Safe.

See also

`strtol()`

4.17.8.2 atol()

Description

Convert to number, long.

Prototype

```
long int atol(const char * nptr);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.

Return value

Returns the converted value, if any. If the value of the result cannot be represented, the behavior is undefined.

Additional information

Converts the initial portion of the string pointed to by nptr to a long representation.

atol() does not affect the value of errno on an error.

Notes

Except for the behavior on error, atol() is equivalent to strtol(nptr, NULL, 10).

Thread safety

Safe.

See also

strtol()

4.17.8.3 atoll()

Description

Convert to number, long long.

Prototype

```
long long int atoll(const char * nptr);
```

Parameters

Parameter	Description
`nptr`	Pointer to string to convert from.

Return value

Returns the converted value, if any. If the value of the result cannot be represented, the behavior is undefined.

Additional information

Converts the initial portion of the string pointed to by `nptr` to a long-long representation.

`atoll()` does not affect the value of errno on an error.

Notes

Except for the behavior on error, `atoll()` is equivalent to strtoll(nptr, `NULL`, 10).

Thread safety

Safe.

See also

`strtoll()`

4.17.8.4 atof()

Description

Convert to number, double.

Prototype

```
double atof(const char * nptr);
```

Parameters

Parameter	Description
`nptr`	Pointer to string to convert from.

Return value

Returns the converted value, if any. If the value of the result cannot be represented, the behavior is undefined.

Additional information

Converts the initial portion of the string pointed to by `nptr` to an double representation.

`atof()` does not affect the value of errno on an error.

Notes

Except for the behavior on error, `atof()` is equivalent to (int)strtod(nptr, `NULL`).

Thread safety

Safe.

See also

`strtod()`

4.17.8.5 strtol()

Description

Convert to number, long.

Prototype

```
long strtol(const char  * nptr,
                  char ** endptr,
                  int     base);
```

Parameters

Parameter	Description
`nptr`	Pointer to string to convert from.
`endptr`	If nonnull, a pointer to object that receives the pointer to the first unconverted character.
`base`	Radix to use for conversion, 2 to 36.

Return value

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_MIN` or `LONG_MAX` is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by `nptr` to a long representation.

First, `strtol()` decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by `isspace()`, 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. `strtol()` then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of `base` is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by `base`. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted.

If the value of `base` is 16, the characters "`0x`" or "`0X`" may optionally precede the sequence of letters and digits, following the optional sign.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the `base` for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

4.17.8.6 strtoll()

Description

Convert to number, long long.

Prototype

```
long long strtoll(const char  * nptr,
                        char ** endptr,
                        int     base);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.
base	Radix to use for conversion, 2 to 36.

Return value

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, LLONG_MIN or LLONG_MAX is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to a long representation.

First, strtoll() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. strtoll() then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of base is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by base. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted.

If the value of base is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

4.17.8.7 strtoul()

Description

Convert to number, unsigned long.

Prototype

```
unsigned long strtoul(const char  * nptr,
                            char ** endptr,
                            int     base);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.
base	Radix to use for conversion, 2 to 36.

Return value

`strtoul()` 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 and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by `nptr` to a long int representation.

First, `strtoul()` decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by `isspace()`, 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. `strtoul()` then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of `base` is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by `base`. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted.

If the value of `base` is 16, the characters "`0x`" or "`0X`" may optionally precede the sequence of letters and digits, following the optional sign.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the `base` for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

4.17.8.8 strtoull()

Description

Convert to number, unsigned long long.

Prototype

```
unsigned long long strtoull(const char  * nptr,
                                  char ** endptr,
                                  int     base);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.
base	Radix to use for conversion, 2 to 36.

Return value

strtoull() 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, ULLONG_MAX is and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to a long int representation.

First, strtoull() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. strtoull() then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of base is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by base. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted.

If the value of base is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

4.17.8.9 strtof()

Description

Convert to number, float.

Prototype

```
float strtof(const char  * nptr,
                   char ** endptr);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.

Return value

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, HUGE_VALF is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by `nptr` to float representation.

First, `strtof()` decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by `isspace()`, 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. `strtof()` then attempts to convert the subject sequence to a floating-point number, and return the result.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

See also

`strtod()`

4.17.8.10 strtod()

Description

Convert to number, double.

Prototype

```
double strtod(const char  * nptr,
                    char ** endptr);
```

Parameters

Parameter	Description
`nptr`	Pointer to string to convert from.
`endptr`	If nonnull, a pointer to object that receives the pointer to the first unconverted character.

Return value

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, HUGE_VAL is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by `nptr` to double representation.

First, `strtod()` decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by `isspace()`, 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. `strtod()` then attempts to convert the subject sequence to a floating-point number, and return the result.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

See also

`strtof()`

4.17.8.11 strtold()

Description

Convert to number, long double.

Prototype

```
long double strtold(const char  * nptr,
                    char ** endptr);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.

Return value

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, HUGE_VAL is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to long double representation.

First, strtold() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. strtod() then attempts to convert the subject sequence to a floating-point number, and return the result.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Thread safety

Safe.

See also

strtod()

4.17.8.12 wcstol()

Description

Convert to number, long.

Prototype

```
long wcstol(const wchar_t  * nptr,
                  wchar_t ** endptr,
                  int        base);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.
base	Radix to use for conversion, 2 to 36.

Return value

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_MIN or LONG_MAX is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to a long representation.

First, wcstol() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. wcstol() then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of base is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by base. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted.

If the value of base is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

4.17.8.13 wcstoll()

Description

Convert to number, long long.

Prototype

```
long long wcstoll(const wchar_t  * nptr,
                        wchar_t ** endptr,
                        int        base);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.
base	Radix to use for conversion, 2 to 36.

Return value

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, LLONG_MIN or LLONG_MAX is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to a long representation.

First, wcstoll() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. wcstoll() then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of base is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by base. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted.

If the value of base is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

4.17.8.14 wcstoul()

Description

Convert to number, unsigned long.

Prototype

```
unsigned long wcstoul(const wchar_t  * nptr,
                            wchar_t ** endptr,
                            int         base);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.
base	Radix to use for conversion, 2 to 36.

Return value

wcstoul() 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 and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to a long int representation.

First, wcstoul() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. wcstoul() then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of base is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by base. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted.

If the value of base is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

4.17.8.15 wcstoull()

Description

Convert to number, unsigned long long.

Prototype

```
unsigned long long wcstoull(const wchar_t  * nptr,
                                  wchar_t ** endptr,
                                  int         base);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.
base	Radix to use for conversion, 2 to 36.

Return value

wcstoull() 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, ULLONG_MAX is and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to a long int representation.

First, wcstoull() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. wcstoull() then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of base is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by base. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted.

If the value of base is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

4.17.8.16 wcstof()

Description

Convert to number, float.

Prototype

```
float wcstof(const wchar_t  * nptr,
                   wchar_t ** endptr);
```

Parameters

Parameter	Description
`nptr`	Pointer to string to convert from.
`endptr`	If nonnull, a pointer to object that receives the pointer to the first unconverted character.

Return value

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, HUGE_VALF is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by `nptr` to float representation.

First, `wcstof()` decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by `isspace()`, 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. `wcstof()` then attempts to convert the subject sequence to a floating-point number, and return the result.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Thread safety

Safe.

See also

`wcstod()`

4.17.8.17 wcstod()

Description

Convert to number, double.

Prototype

```
double wcstod(const wchar_t  * nptr,
                    wchar_t ** endptr);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.

Return value

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, HUGE_VAL is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to double representation.

First, wcstod() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. wcstod() then attempts to convert the subject sequence to a floating-point number, and return the result.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Thread safety

Safe.

See also

wcstof()

4.17.8.18 wcstold()

Description

Convert to number, long double.

Prototype

```
long double wcstold(const wchar_t  * nptr,
                          wchar_t ** endptr);
```

Parameters

Parameter	Description
nptr	Pointer to string to convert from.
endptr	If nonnull, a pointer to object that receives the pointer to the first unconverted character.

Return value

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, HUGE_VAL is returned according to the sign of the value, if any, and the value of the macro ERANGE is stored in errno.

Additional information

Converts the initial portion of the string pointed to by nptr to long double representation.

First, wcstold() decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, as specified by isspace(), 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. wcstod() then attempts to convert the subject sequence to a floating-point number, and return the result.

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 entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Thread safety

Safe.

See also

wcstod()

4.17.9 Multi-byte/wide character functions

Function	Description
`btowc()`	Convert single-byte character to wide character.
`btowc_l()`	Convert single-byte character to wide character, per locale, (POSIX.1).
`mblen()`	Count number of bytes in multi-byte character.
`mblen_l()`	Count number of bytes in multi-byte character, per locale (POSIX.1).
`mbtowc()`	Convert multi-byte character to wide character.
`mbtowc_l()`	Convert multi-byte character to wide character, per locale (POSIX.1).
`mbstowcs()`	Convert multi-byte string to wide string.
`mbstowcs_l()`	Convert multi-byte string to wide string, per locale (POSIX.1).
`mbsrtowcs()`	Convert multi-byte string to wide character string, restartable.
`mbsrtowcs_l()`	Convert multi-byte string to wide character string, restartable, per locale (POSIX.1).
`mbsnrtowcs()`	Convert multi-byte string to wide character string, restartable, per locale (POSIX.1).
`mbsnrtowcs_l()`	Convert multi-byte string to wide character string, restartable, per locale (POSIX.1).
`wctomb()`	Convert wide character to multi-byte character.
`wctomb_l()`	Convert wide character to multi-byte character, per locale (POSIX.1).
`wcstombs()`	Convert wide string to multi-byte string.
`wcstombs_l()`	Convert wide string to multi-byte string.

4.17.9.1 btowc()

Description

Convert single-byte character to wide character.

Prototype

```
wint_t btowc(int c);
```

Parameters

Parameter	Description
c	Character to convert.

Return value

Returns WEOF if c has the value EOF or if c, converted to an unsigned char and in the current locale, does not constitute a valid single-byte character in the initial shift state.

Additional information

Determines whether c constitutes a valid single-byte character in the current locale. If c is a valid single-byte character, btowc() returns the wide character representation of that character.

Thread safety

Safe [if configured].

4.17.9.2 btowc_l()

Description

Convert single-byte character to wide character, per locale, (POSIX.1).

Prototype

```
wint_t btowc_l(int c,
                   locale_t loc);
```

Parameters

Parameter	Description
c	Character to convert.
loc	Locale used for conversion.

Return value

Returns WEOF if c has the value EOF or if c, converted to an unsigned char and in the locale loc, does not constitute a valid single-byte character in the initial shift state.

Additional information

Determines whether c constitutes a valid single-byte character in the locale loc. If c is a valid single-byte character, btowc_l() returns the wide character representation of that character.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.17.9.3 mblen()

Description

Count number of bytes in multi-byte character.

Prototype

```
int mblen(const char   * s,
                size_t   n);
```

Parameters

Parameter	Description
s	Pointer to multi-byte character.
n	Maximum number of bytes to examine.

Return value

If s is a null pointer, returns a nonzero or zero value, if multi-byte character encodings, respectively, do or do not have state-dependent encodings.

If s is not a null pointer, 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 valid multi-byte character), or returns -1 (if they do not form a valid multi-byte character).

Additional information

Determines the number of bytes contained in the multi-byte character pointed to by s in the current locale.

Except that the conversion state of the `mbtowc()` function is not affected, it is equivalent to

mbtowc(NULL, s, n);

Thread safety

Safe [if configured].

See also

`mblen_l()`, `mbtowc()`

4.17.9.4 mblen_l()

Description

Count number of bytes in multi-byte character, per locale (POSIX.1).

Prototype

```
int mblen_l(const char   * s,
                  size_t   n,
                           locale_t loc);
```

Parameters

Parameter	Description
s	Pointer to multi-byte character.
n	Maximum number of bytes to examine.
loc	Locale to use for conversion.

Return value

If s is a null pointer, returns a nonzero or zero value, if multi-byte character encodings, respectively, do or do not have state-dependent encodings in locale loc.

If s is not a null pointer, 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 valid multi-byte character), or returns -1 (if they do not form a valid multi-byte character).

Additional information

Determines the number of bytes contained in the multi-byte character pointed to by s in the locale loc.

Except that the conversion state of the `mbtowc()` function is not affected, it is equivalent to

`mbtowc_l(NULL, s, n, loc);`

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

See also

`mblen()`, `mbtowc()`

4.17.9.5 mbtowc()

Description

Convert multi-byte character to wide character.

Prototype

```
int mbtowc(      wchar_t * pwc,
           const char    * s,
                 size_t    n);
```

Parameters

Parameter	Description
pwc	Pointer to object that receives the wide character.
s	Pointer to multi-byte character string.
n	Maximum number of bytes that will be examined.

Return value

If s is a null pointer, mbtowc() returns a nonzero value if multi-byte character encodings are state-dependent in the current locale, and zero otherwise.

If s is not null and the object that s points to is a wide character null, mbtowc() returns 0.

If s is not null and the object that s points to forms a valid multi-byte character, mbtowc() returns the length in bytes of the multi-byte character.

If the object that mbtowc() points to does not form a valid multi-byte character within the first n characters, it returns -1.

Additional information

Converts a single multi-byte character to a wide character in the current locale. The wide character, if the multi-byte character string is converted correctly, is stored into the object pointed to by pwc.

Thread safety

Safe [if configured].

See also

mbtowc_l().

4.17.9.6 mbtowc_l()

Description

Convert multi-byte character to wide character, per locale (POSIX.1).

Prototype

```
int mbtowc_l(      wchar_t * pwc,
             const char    * s,
                   size_t    n,
                             locale_t loc);
```

Parameters

Parameter	Description
pwc	Pointer to object that receives the wide character.
s	Pointer to multi-byte character string.
n	Maximum number of bytes that will be examined.
loc	Locale used to convert the multi-byte character.

Return value

If s is a null pointer, mbtowc_l() returns a nonzero value if multi-byte character encodings are state-dependent in locale loc, and zero otherwise.

If s is not null and the object that s points to is a wide null character, mbtowc_l() returns 0.

If s is not null and the object that s points to forms a valid multi-byte character, mbtowc_l() returns the length in bytes of the multi-byte character.

If the object that mbtowc_l() points to does not form a valid multi-byte character within the first n characters, it returns -1.

Additional information

Converts a single multi-byte character to a wide character in the locale loc. The wide character, if the multi-byte character string is converted correctly, is stored into the object pointed to by pwc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

See also

mbtowc()

4.17.9.7 mbstowcs()

Description

Convert multi-byte string to wide string.

Prototype

```
size_t mbstowcs(      wchar_t * pwcs,
                const char    * s,
                      size_t    n);
```

Parameters

Parameter	Description
pwcs	Pointer to array that receives the wide character string.
s	Pointer to array that contains the multi-byte string.
n	Maximum number of wide characters to write into pwcs.

Return value

Returns -1 if an invalid multi-byte character is encountered, otherwise returns the number of array elements modified (if any), not including a terminating null wide character.

Additional information

Converts a sequence of multi-byte characters, in the current locale, that begins in the initial shift state from the array pointed to by s into a sequence of corresponding wide characters and stores not more than n wide characters into the array pointed to by pwcs.

No multi-byte characters that follow a null character (which is converted into a null wide character) will be examined or converted. Each multi-byte character is converted as if by a call to the mbtowc() function, except that the conversion state of the mbtowc() function is not affected.

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

Thread safety

Safe [if configured].

4.17.9.8 mbstowcs_l()

Description

Convert multi-byte string to wide string, per locale (POSIX.1).

Prototype

```
size_t mbstowcs_l(      wchar_t * pwcs,
                  const char    * s,
                        size_t    n,
                                  locale_t loc);
```

Parameters

Parameter	Description
pwcs	Pointer to array that receives the wide character string.
s	Pointer to array that contains the multi-byte string.
n	Maximum number of wide characters to write into pwcs.
loc	Locale to use for conversion.

Return value

Returns -1 if an invalid multi-byte character is encountered, otherwise returns the number of array elements modified (if any), not including a terminating null wide character.

Additional information

Converts a sequence of multi-byte characters, in the locale loc, that begins in the initial shift state from the array pointed to by s into a sequence of corresponding wide characters and stores not more than n wide characters into the array pointed to by pwcs.

No multi-byte characters that follow a null character (which is converted into a null wide character) will be examined or converted. Each multi-byte character is converted as if by a call to the `mbtowc()` function, except that the conversion state of the `mbtowc()` function is not affected.

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

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.17.9.9 mbsrtowcs()

Description

Convert multi-byte string to wide character string, restartable.

Prototype

```
size_t mbsrtowcs(      wchar_t    * dst,
                 const char      ** src,
                       size_t       len,
                       mbstate_t  * ps);
```

Parameters

Parameter	Description
dst	Pointer to object that receives the converted wide characters.
src	Pointer to pointer to multi-byte character string.
len	Maximum number of wide characters that will be written to dst.
ps	Pointer to multi-byte conversion state.

Return value

The number of wide characters written to dst (not including the eventual terminating null character).

Additional information

Converts a sequence of multi-byte characters, in the current locale, that begins in the conversion state described by the object pointed to by ps, from the array indirectly pointed to by src into a sequence of corresponding wide characters.

If dst is not a null pointer, the converted characters are stored into the array pointed to by dst. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multi-byte character, or (if dst is not a null pointer) when len wide characters have been stored into the array pointed to by dst. Each conversion takes place as if by a call to the mbrtowc() function.

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multi-byte character converted (if any). If conversion stopped due to reaching a terminating null character and if dst is not a null pointer, the resulting state described is the initial conversion state.

Thread safety

Safe [if configured].

See also

mbsrtowcs_l(), mbrtowc()

4.17.9.10 mbsrtowcs_l()

Description

Convert multi-byte string to wide character string, restartable, per locale (POSIX.1).

Prototype

```
size_t mbsrtowcs_l(      wchar_t     * dst,
                   const char       ** src,
                         size_t         len,
                         mbstate_t   * ps,
                                        locale_t loc);
```

Parameters

Parameter	Description
dst	Pointer to object that receives the converted wide characters.
src	Pointer to pointer to multi-byte character string.
len	Maximum number of wide characters that will be written to dst.
ps	Pointer to multi-byte conversion state.
loc	Locale used for conversion.

Return value

The number of wide characters written to dst (not including the eventual terminating null character).

Additional information

Converts a sequence of multi-byte characters, in the locale loc, that begins in the conversion state described by the object pointed to by ps, from the array indirectly pointed to by src into a sequence of corresponding wide characters.

If dst is not a null pointer, the converted characters are stored into the array pointed to by dst. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multi-byte character, or (if dst is not a null pointer) when len wide characters have been stored into the array pointed to by dst. Each conversion takes place as if by a call to the mbrtowc() function.

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multi-byte character converted (if any). If conversion stopped due to reaching a terminating null character and if dst is not a null pointer, the resulting state described is the initial conversion state.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

See also

mbsrtowcs(), mbrtowc()

4.17.9.11 mbsnrtowcs()

Description

Convert multi-byte string to wide character string, restartable, per locale (POSIX.1).

Prototype

```
size_t mbsnrtowcs(      wchar_t    * dst,
                  const char      ** src,
                        size_t        nmc,
                        size_t        len,
                        mbstate_t  * ps);
```

Parameters

Parameter	Description
dst	Pointer to object that receives the converted wide characters.
src	Pointer to pointer to multi-byte character string.
nmc	Maximum number of bytes to be read from src.
len	Maximum number of wide characters that will be written to dst.
ps	Pointer to multi-byte conversion state.

Return value

The number of wide characters written to dst (not including the eventual terminating null character).

Additional information

Converts a sequence of multi-byte characters, in the locale loc, that begins in the conversion state described by the object pointed to by ps, from the array indirectly pointed to by src into a sequence of corresponding wide characters.

If dst is not a null pointer, the converted characters are stored into the array pointed to by dst. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multi-byte character, or (if dst is not a null pointer) when len wide characters have been stored into the array pointed to by dst. Each conversion takes place as if by a call to the mbrtowc() function.

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multi-byte character converted (if any). If conversion stopped due to reaching a terminating null character and if dst is not a null pointer, the resulting state described is the initial conversion state.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe [if configured].

See also

mbsrtowcs(), mbrtowc()

4.17.9.12 mbsnrtowcs_l()

Description

Convert multi-byte string to wide character string, restartable, per locale (POSIX.1).

Prototype

```
size_t mbsnrtowcs_l(      wchar_t    * dst,
                    const char      ** src,
                          size_t        nmc,
                          size_t        len,
                          mbstate_t  * ps,
                                        locale_t loc);
```

Parameters

Parameter	Description
dst	Pointer to object that receives the converted wide characters.
src	Pointer to pointer to multi-byte character string.
nmc	Maximum number of bytes to be read from src.
len	Maximum number of wide characters that will be written to dst.
ps	Pointer to multi-byte conversion state.
loc	Locale used for conversion.

Return value

The number of wide characters written to dst (not including the eventual terminating null character).

Additional information

Converts a sequence of multi-byte characters, in the locale loc, that begins in the conversion state described by the object pointed to by ps, from the array indirectly pointed to by src into a sequence of corresponding wide characters.

If dst is not a null pointer, the converted characters are stored into the array pointed to by dst. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multi-byte character, or (if dst is not a null pointer) when len wide characters have been stored into the array pointed to by dst. Each conversion takes place as if by a call to the mbrtowc() function.

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multi-byte character converted (if any). If conversion stopped due to reaching a terminating null character and if dst is not a null pointer, the resulting state described is the initial conversion state.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

See also

`mbsrtowcs()`, `mbrtowc()`

4.17.9.13 wctomb()

Description

Convert wide character to multi-byte character.

Prototype

```
int wctomb(char * s,
           wchar_t wc);
```

Parameters

Parameter	Description
s	Pointer to array that receives the multi-byte character.
wc	Wide character to convert.

Return value

Returns the number of bytes stored in the array object. When wc is not a valid wide character, an encoding error occurs: `wctomb()` stores the value of the macro EILSEQ in errno and returns `(size_t)(-1)`; the conversion state is unspecified.

Additional information

If s is a null pointer, `wctomb()` is equivalent to the call wcrtomb(buf, 0, ps) where buf is an internal buffer.

If s is not a null pointer, `wctomb()` determines the number of bytes needed to represent the multi-byte character that corresponds to the wide character given by wc in the current locale, and stores the multi-byte character representation in the array whose first element is pointed to by s. At most `MB_CUR_MAX` bytes are stored. If wc is a null wide character, a null byte is stored; the resulting state described is the initial conversion state.

Thread safety

Safe [if configured].

4.17.9.14 wctomb_l()

Description

Convert wide character to multi-byte character, per locale (POSIX.1).

Prototype

```
int wctomb_l(char * s,
             wchar_t wc,
             locale_t loc);
```

Parameters

Parameter	Description
s	Pointer to array that receives the multi-byte character.
wc	Wide character to convert.
loc	Locale used for conversion.

Return value

Returns the number of bytes stored in the array object. When wc is not a valid wide character, an encoding error occurs: wctomb_l() stores the value of the macro EILSEQ in errno and returns (size_t)(-1); the conversion state is unspecified.

Additional information

If s is a null pointer, wctomb_l() is equivalent to the call wcrtomb_l(buf, 0, ps, loc) where buf is an internal buffer.

If s is not a null pointer, wctomb_l() determines the number of bytes needed to represent the multi-byte character that corresponds to the wide character given by wc in the locale loc, and stores the multi-byte character representation in the array whose first element is pointed to by s. At most MB_CUR_MAX bytes are stored. If wc is a null wide character, a null byte is stored; the resulting state described is the initial conversion state.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.17.9.15 wcstombs()

Description

Convert wide string to multi-byte string.

Prototype

```
size_t wcstombs(      char    * s,
                const wchar_t * pwcs,
                      size_t    n);
```

Parameters

Parameter	Description
s	Pointer to array that receives the multi-byte string.
pwcs	Pointer to wide character string to convert.
n	Maximum number of bytes to write into s.

Return value

If a wide character is encountered that does not correspond to a valid multibyte character in the current locale, returns (size_t)(-1). Otherwise, returns the number of bytes written, not including a terminating null character (if any).

Additional information

Converts a sequence of wide characters in the current locale from the array pointed to by pwcs into a sequence of corresponding 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 wide character is converted as if by a call to wctomb(), except that the conversion state of wctomb() is not affected.

Thread safety

Safe [if configured].

4.17.9.16 wcstombs_l()

Description

Convert wide string to multi-byte string.

Prototype

```
size_t wcstombs_l(      char    * s,
                  const wchar_t * pwcs,
                        size_t    n,
                                  locale_t loc);
```

Parameters

Parameter	Description
s	Pointer to array that receives the multi-byte string.
pwcs	Pointer to wide character string to convert.
n	Maximum number of bytes to write into s.
loc	Locale used for conversion.

Return value

If a wide character is encountered that does not correspond to a valid multibyte character in the locale loc, returns (size_t)(-1). Otherwise, returns the number of bytes written, not including a terminating null character (if any).

Additional information

Converts a sequence of wide characters in the locale loc from the array pointed to by pwcs into a sequence of corresponding 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 wide character is converted as if by a call to wctomb(), except that the conversion state of wctomb() is not affected.

Thread safety

Safe.

4.18 <string.h>

The header file `<string.h>` defines functions that operate on arrays that are interpreted as null-terminated strings.

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.

Where an argument declared as `size_t` *n* specifies the length of an array for a function, *n* can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function, pointer arguments must have valid values on a call with a zero size. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

4.18.1 Copying functions

Function	Description
`memset()`	Set memory to character.
`memset_s()`	Set memory to character, safe (C11).
`memcpy()`	Copy memory.
`memcpy_s()`	Copy memory, safe (C11).
`memccpy()`	Copy memory, specify terminator (POSIX.1).
`mempcpy()`	Copy memory (GNU).
`memmove()`	Copy memory, tolerate overlaps.
`memmove_s()`	Copy memory, tolerate overlaps, safe (C11).
`strcpy()`	Copy string.
`strcpy_s()`	Copy string, safe (C11).
`strncpy()`	Copy string, limit length.
`strlcpy()`	Copy string, limit length, always zero terminate (BSD).
`stpcpy()`	Copy string, return end.
`stpncpy()`	Copy string, limit length, return end.
`strcat()`	Concatenate strings.
`strncat()`	Concatenate strings, limit length.
`strlcat()`	Concatenate strings, limit length, always zero terminate (BSD).
`strdup()`	Duplicate string (POSIX.1).
`strndup()`	Duplicate string, limit length (POSIX.1).
`strxfrm()`	Transform strings.

4.18.1.1 memset()

Description

Set memory to character.

Prototype

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

Parameters

Parameter	Description
s	Pointer to destination object.
c	Character to write.
n	Number of characters to write in destination object.

Return value

Returns s.

Additional information

Copies the value of c (converted to an unsigned char) into each of the first n characters of the object pointed to by s.

Thread safety

Safe.

4.18.1.2 memset_s()

Description

Set memory to character, safe (C11).

Prototype

```
errno_t memset_s(void   * s,
                 size_t   smax,
                 int      c,
                 size_t   n);
```

Parameters

Parameter	Description
s	Pointer to destination object.
smax	Size of destination object in characters.
c	Character to write.
n	Number of characters to write in destination object.

Runtime constraints

- s shall not be a null pointer.
- Neither smax nor n shall be greater than RSIZE_MAX.
- n shall not be greater than s1max.

If there is a runtime-constraint violation, memset_s() stores c (converted to an unsigned char) into each of the first smax characters of the object pointed to by s, if s is not a null pointer and smax is not greater than RSIZE_MAX.

Return value

= 0	if runtime constraints are not violated.
≠ 0	if runtime constraints are violated.

Additional information

The macro __STDC_WANT_LIB_EXT1__ must be set to 1 before including <string.h> to access this 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.

Conformance

ISO/IEC 9899:2011 (C11).

Thread safety

Safe.

4.18.1.3 memcpy()

Description

Copy memory.

Prototype

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

Parameters

Parameter	Description
s1	Pointer to destination object.
s2	Pointer to source object.
n	Number of characters to copy.

Return value

Returns a pointer to the destination object.

Additional information

Copies n characters from the object pointed to by s2 into the object pointed to by s1. The behavior of memcpy() is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.18.1.4 memcpy_s()

Description

Copy memory, safe (C11).

Prototype

```
errno_t memcpy_s(       void    * s1,
                        size_t    s1max,
                  const void    * s2,
                        size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s1max	Size of destination object in characters.
s2	Pointer to source object.
n	Number of characters to copy.

Runtime constraints

- Neither s1 nor s2 shall be a null pointer.
- Neither s1max nor n shall be greater than RSIZE_MAX.
- n shall not be greater than s1max.
- Copying shall not take place between objects that overlap.

If there is a runtime-constraint violation, memcpy_s() stores zeros in the first s1max characters of the object pointed to by s1, if s1 is not a null pointer and s1max is not greater than RSIZE_MAX.

Return value

= 0	if runtime constraints are not violated.
≠ 0	if runtime constraints are violated.

Additional information

The macro __STDC_WANT_LIB_EXT1__ must be set to 1 before including <string.h> to access this function.

Copies n characters from the object pointed to by s2 into the object pointed to by s1.

Conformance

ISO/IEC 9899:2011 (C11).

Thread safety

Safe.

4.18.1.5 memccpy()

Description

Copy memory, specify terminator (POSIX.1).

Prototype

```
void *memccpy(      void   * s1,
              const void   * s2,
                    int      c,
                    size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s2	Pointer to source object.
c	Character that terminates copy.
n	Maximum number of characters to copy.

Return value

Returns a pointer to the character immediately following c in s1, or NULL if c was not found in the first n characters of s2.

Additional information

Copies at most n characters from the object pointed to by s2 into the object pointed to by s1. The copying stops as soon as n characters are copied or the character c is copied into the destination object pointed to by s1.

The behavior of memccpy() is undefined if copying takes place between objects that overlap.

Conformance

POSIX.1-2008.

Thread safety

Safe.

4.18.1.6 mempcpy()

Description

Copy memory (GNU).

Prototype

```
void *mempcpy(      void   * s1,
              const void   * s2,
                    size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s2	Pointer to source object.
n	Number of characters to copy.

Return value

Returns a pointer to the character immediately following the final character written into s1.

Additional information

Copies n characters from the object pointed to by s2 into the object pointed to by s1. The behavior of mempcpy() is undefined if copying takes place between objects that overlap.

Conformance

This is an extension found in GNU libc.

Thread safety

Safe.

4.18.1.7 memmove()

Description

Copy memory, tolerate overlaps.

Prototype

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

Parameters

Parameter	Description
s1	Pointer to destination object.
s2	Pointer to source object.
n	Number of characters to copy.

Return value

Returns the value of s1.

Additional information

Copies n characters from the object pointed to by s2 into the object pointed to by s1 ensuring that if s1 and s2 overlap, the copy works correctly. 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.

Thread safety

Safe.

4.18.1.8 memmove_s()

Description

Copy memory, tolerate overlaps, safe (C11).

Prototype

```
errno_t memmove_s(      void   * s1,
                        size_t   s1max,
                  const void   * s2,
                        size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s1max	Size of destination object in characters.
s2	Pointer to source object.
n	Number of characters to copy.

Runtime constraints

- Neither s1 nor s2 shall be a null pointer.
- Neither s1max nor n shall be greater than RSIZE_MAX.
- n shall not be greater than s1max.

If there is a runtime-constraint violation, memmove_s() stores zeros in the first s1max characters of the object pointed to by s1, if s1 is not a null pointer and s1max is not greater than RSIZE_MAX.

Return value

= 0	if runtime constraints are not violated.
≠ 0	if runtime constraints are violated.

Additional information

The macro __STDC_WANT_LIB_EXT1__ must be set to 1 before including <string.h> to access this function.

Copies n characters from the object pointed to by s2 into the object pointed to by s1 ensuring that if s1 and s2 overlap, the copy works correctly. 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.

Conformance

ISO/IEC 9899:2011 (C11).

Thread safety

Safe.

4.18.1.9 strcpy()

Description

Copy string.

Prototype

```
char *strcpy(      char * s1,
             const char * s2);
```

Parameters

Parameter	Description
s1	String to copy to.
s2	String to copy.

Return value

Returns the value of s1.

Additional information

Copies the string pointed to by s2 (including the terminating null character) into the array pointed to by s1. The behavior of strcpy() is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.18.1.10 strcpy_s()

Description

Copy string, safe (C11).

Prototype

```
errno_t strcpy_s(      char   * s1,
                       size_t   s1max,
                 const char   * s2);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s1max	Size of destination object in characters.
s2	Pointer to source object.

Runtime constraints

- Neither s1 nor s2 shall be a null pointer.
- s1max shall not be greater than RSIZE_MAX.
- s1max shall be greater than strnlen_s(s2, s1max).
- Copying shall not take place between objects that overlap.

If there is a runtime-constraint violation, strcpy_s() sets s1[0] to zero if s1 is nonnull and s1max is not greater than RSIZE_MAX.

Return value

= 0	if runtime constraints are not violated.
≠ 0	if runtime constraints are violated.

Additional information

The macro __STDC_WANT_LIB_EXT1__ must be set to 1 before including <string.h> to access this function.

Copies the string pointed to by s2 (including the terminating null character) into the array pointed to by s1.

Conformance

ISO/IEC 9899:2011 (C11).

Thread safety

Safe.

4.18.1.11 strncpy()

Description

Copy string, limit length.

Prototype

```
char *strncpy(      char   * s1,
              const char   * s2,
                    size_t   n);
```

Parameters

Parameter	Description
s1	String to copy to.
s2	String to copy.
n	Maximum number of characters to copy.

Return value

Returns the value of s1.

Additional information

Copies not more than n characters from the array pointed to by s2 to the array pointed to by s1. Characters that follow a null character in s2 are not copied. The behavior of strncpy() is undefined if copying takes place between objects that overlap. If the array pointed to by s2 is a string that is shorter than n characters, null characters are appended to the copy in the array pointed to by s1, until n characters in all have been written.

Notes

No null character is implicitly appended to the end of s1, so s1 will only be terminated by a null character if the length of the string pointed to by s2 is less than n.

Thread safety

Safe.

4.18.1.12 strlcpy()

Description

Copy string, limit length, always zero terminate (BSD).

Prototype

```
size_t strlcpy(      char   * s1,
               const char   * s2,
                     size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to string to copy to.
s2	Pointer to string to copy.
n	Maximum number of characters, including terminating null, in s1.

Return value

Returns the number of characters it tried to copy, which is the length of the string s2 or n, whichever is smaller.

Additional information

Copies up to n-1 characters from the string pointed to by s2 into the array pointed to by s1 and always terminates the result with a null character.

The behavior of strlcpy() is undefined if copying takes place between objects that overlap.

Conformance

Commonly found in BSD libraries and contrasts with strncpy() in that the resulting string is always terminated with a null character.

Thread safety

Safe.

4.18.1.13 stpcpy()

Description

Copy string, return end.

Prototype

```
char *stpcpy(      char * s1,
             const char * s2);
```

Parameters

Parameter	Description
s1	String to copy to.
s2	String to copy.

Return value

A pointer to the end of the string s1, i.e. the terminating null byte of the string s1, after s2 is copied to it.

Additional information

Copies the string pointed to by s2 (including the terminating null character) into the array pointed to by s1. The behavior of stpcpy() is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.18.1.14 stpncpy()

Description

Copy string, limit length, return end.

Prototype

```
char *stpncpy(      char   * s1,
              const char   * s2,
                    size_t   n);
```

Parameters

Parameter	Description
s1	String to copy to.
s2	String to copy.
n	Maximum number of characters to copy.

Return value

stpncpy() returns a pointer to the terminating null byte in s1 after it is copied to, or, if s1 is not null-terminated, s1+n.

Additional information

Copies not more than n characters from the array pointed to by s2 to the array pointed to by s1. Characters that follow a null character in s2 are not copied. The behavior of strncpy() is undefined if copying takes place between objects that overlap. If the array pointed to by s2 is a string that is shorter than n characters, null characters are appended to the copy in the array pointed to by s1, until n characters in all have been written.

Notes

No null character is implicitly appended to the end of s1, so s1 will only be terminated by a null character if the length of the string pointed to by s2 is less than n.

Thread safety

Safe.

4.18.1.15 strcat()

Description

Concatenate strings.

Prototype

```
char *strcat(      char * s1,
             const char * s2);
```

Parameters

Parameter	Description
s1	Zero-terminated string to append to.
s2	Zero-terminated string to append.

Return value

Returns the value of `s1`.

Additional information

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`. The behavior of `strcat()` is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.18.1.16 strncat()

Description

Concatenate strings, limit length.

Prototype

```
char *strncat(      char   * s1,
              const char   * s2,
                    size_t   n);
```

Parameters

Parameter	Description
s1	String to append to.
s2	String to append.
n	Maximum number of characters in s1.

Return value

Returns the value of s1.

Additional information

Appends not more than n characters from the array pointed to by s2 to the end of the string pointed to by s1. A null character in s1 and characters that follow it are not appended. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result.

The behavior of `strncat()` is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.18.1.17 strlcat()

Description

Concatenate strings, limit length, always zero terminate (BSD).

Prototype

```
size_t strlcat(      char   * s1,
               const char   * s2,
                     size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to string to append to.
s2	Pointer to string to append.
n	Maximum number of characters, including terminating null, in s1.

Return value

Returns the number of characters it tried to copy, which is the sum of the lengths of the strings s1 and s2 or n, whichever is smaller.

Additional information

Appends no more than n-strlen(s1}-1 characters pointed to by s2 into the array pointed to by s1 and always terminates the result with a null character if n is greater than zero. Both the strings s1 and s2 must be terminated with a null character on entry to strlcat() and a character position for the terminating null should be included in n.

The behavior of strlcat() is undefined if copying takes place between objects that overlap.

Conformance

Commonly found in BSD libraries.

Thread safety

Safe.

4.18.1.18 strdup()

Description

Duplicate string (POSIX.1).

Prototype

```
char *strdup(const char * s1);
```

Parameters

Parameter	Description
s1	Pointer to string to duplicate.

Return value

Returns a pointer to the new string or a null pointer if the new string cannot be created. The returned pointer can be passed to `free()`.

Additional information

Duplicates the string pointed to by s1 by using `malloc()` to allocate memory for a copy of s and then copyies s, including the terminating null, to that memory

Conformance

POSIX.1-2008 and SC22 TR 24731-2.

Thread safety

Safe.

4.18.1.19 strndup()

Description

Duplicate string, limit length (POSIX.1).

Prototype

```
char *strndup(const char   * s,
                     size_t   n);
```

Parameters

Parameter	Description
s	Pointer to string to duplicate.
n	Maximum number of characters to duplicate.

Return value

Returns a pointer to the new string or a null pointer if the new string cannot be created. The returned pointer can be passed to `free()`.

Additional information

Duplicates at most n characters from the the string pointed to by s by using `malloc()` to allocate memory for a copy of s.

If the length of string pointed to by s is greater than n characters, only n characters will be duplicated. If n is greater than the length of the string pointed to by s, all characters in the string are copied into the allocated array including the terminating null character.

Conformance

Conforms to POSIX.1-2008 and SC22 TR 24731-2.

Thread safety

Safe.

4.18.1.20 strxfrm()

Description

Transform strings.

Prototype

```
size_t strxfrm(       char    * s1,
                const char    * s2,
                      size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to destination array.
s2	Pointer to source string.
n	Maximum number of characters in the destination array.

Return value

Returns the length of the transformed string. If the value returned is n or more, the contents of the array pointed to by s1 are undefined.

Thread safety

Safe.

4.18.2 Comparison functions

Function	Description
`memcmp()`	Compare memory.
`strcmp()`	Compare strings.
`strncmp()`	Compare strings, limit length.
`strcasecmp()`	Compare strings, ignore case (POSIX.1).
`strncasecmp()`	Compare strings, ignore case, limit length (POSIX.1).
`strcoll()`	Collate strings.

4.18.2.1 memcmp()

Description

Compare memory.

Prototype

```
int memcmp(const void   * s1,
           const void   * s2,
                 size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to object #1.
s2	Pointer to object #2.
n	Number of characters to compare.

Return value

< 0	s1 is less than s2.
= 0	s1 is equal to s2.
> 0	s1 is greater than to s2.

Additional information

Compares the first n characters of the object pointed to by s1 to the first n characters of the object pointed to by s2. memcmp() returns an integer greater than, equal to, or less than zero as the object pointed to by s1 is greater than, equal to, or less than the object pointed to by s2.

Thread safety

Safe.

4.18.2.2 strcmp()

Description

Compare strings.

Prototype

```
int strcmp(const char * s1,
           const char * s2);
```

Parameters

Parameter	Description
s1	Pointer to string #1.
s2	Pointer to string #2.

Return value

Returns an integer greater than, equal to, or less than zero, if the null-terminated array pointed to by s1 is greater than, equal to, or less than the null-terminated array pointed to by s2.

Thread safety

Safe.

4.18.2.3 strncmp()

Description

Compare strings, limit length.

Prototype

```
int strncmp(const char   * s1,
            const char   * s2,
                  size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to string #1.
s2	Pointer to string #2.
n	Maximum number of characters to compare.

Return value

Returns an integer greater than, equal to, or less than zero, if 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.

Additional information

Compares not more than n characters from the array pointed to by s1 to the array pointed to by s2. Characters that follow a null character are not compared.

Thread safety

Safe.

4.18.2.4 strcasecmp()

Description

Compare strings, ignore case (POSIX.1).

Prototype

```
int strcasecmp(const char * s1,
               const char * s2);
```

Parameters

Parameter	Description
s1	Pointer to string #1.
s2	Pointer to string #2.

Return value

< 0	s1 is less than s2.
= 0	s1 is equal to s2.
> 0	s1 is greater than to s2.

Additional information

Compares the string pointed to by s1 to the string pointed to by s2 ignoring differences in case.

strcasecmp() returns an integer greater than, equal to, or less than zero if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.

Conformance

POSIX.1-2008.

Thread safety

Safe.

4.18.2.5 strncasecmp()

Description

Compare strings, ignore case, limit length (POSIX.1).

Prototype

```
int strncasecmp(const char   * s1,
                const char   * s2,
                      size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to string #1.
s2	Pointer to string #2.
n	Maximum number of characters to compare.

Return value

< 0	s1 is less than s2.
= 0	s1 is equal to s2.
> 0	s1 is greater than to s2.

Additional information

Compares not more than n characters from the array pointed to by s1 to the array pointed to by s2 ignoring differences in case. Characters that follow a null character are not compared.

strncasecmp() returns an integer greater than, equal to, or less than zero, if 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.

Conformance

POSIX.1-2008.

Thread safety

Safe.

4.18.2.6 strcoll()

Description

Collate strings.

Prototype

```
int strcoll(const char * s1,
            const char * s2);
```

Parameters

Parameter	Description
s1	Pointer to string #1.
s2	Pointer to string #2.

Return value

Returns an integer greater than, equal to, or less than zero, if the null-terminated array pointed to by s1 is greater than, equal to, or less than the null-terminated array pointed to by s2.

Thread safety

Safe.

4.18.3 Search functions

Function	Description
memchr()	Find character in memory, forward.
memrchr()	Find character in memory, reverse (BSD).
memmem()	Find memory in memory, forward (BSD).
strchr()	Find character within string, forward.
strnchr()	Find character within string, forward, limit length.
strrchr()	Find character within string, reverse.
strlen()	Calculate length of string.
strnlen()	Calculate length of string, limit length (POSIX.1).
strnlen_s()	Calculate length of string, limit length (C11).
strstr()	Find string within string, forward.
strnstr()	Find string within string, forward, limit length (BSD).
strcasestr()	Find string within string, forward, ignore case (BSD).
strncasestr()	Find string within string, forward, ignore case, limit length (BSD).
strpbrk()	Find first occurrence of characters within string.
strspn()	Compute size of string prefixed by a set of characters.
strcspn()	Compute size of string not prefixed by a set of characters.
strtok()	Break string into tokens.
strtok_r()	Break string into tokens, restartable (POSIX.1).
strsep()	Break string into tokens (BSD).

4.18.3.1 memchr()

Description

Find character in memory, forward.

Prototype

```
void *memchr(const void   * s,
                   int        c,
                   size_t     n);
```

Parameters

Parameter	Description
s	Pointer to object to search.
c	Character to search for.
n	Number of characters in object to search.

Return value

= NULL	c does not occur in the object.
≠ NULL	Pointer to the located character.

Additional information

Locates the first occurrence of c (converted to an unsigned char) in the initial n characters (each interpreted as unsigned char) of the object pointed to by s. Unlike `strchr()`, `memchr()` does not terminate a search when a null character is found in the object pointed to by s.

Thread safety

Safe.

4.18.3.2 memrchr()

Description

Find character in memory, reverse (BSD).

Prototype

```
void *memrchr(const void   * s,
                    int        c,
                    size_t     n);
```

Parameters

Parameter	Description
s	Pointer to object to search.
c	Character to search for.
n	Number of characters in object to search.

Return value

Returns a pointer to the located character, or a null pointer if c does not occur in the octet string.

Additional information

Locates the last occurrence of c (converted to a char) in the octet string pointed to by s.

Conformance

Commonly found in Linux and BSD C libraries.

Thread safety

Safe.

4.18.3.3 memmem()

Description

Find memory in memory, forward (BSD).

Prototype

```
void *memmem(const void   * s1,
                   size_t   n1,
             const void   * s2,
                   size_t   n2);
```

Parameters

Parameter	Description
s1	Pointer to object to search.
n1	Number of characters to search in s1.
s2	Pointer to object to search for.
n2	Number of characters to search from s2.

Return value

= NULL	(s2, n2) does not occur in (s1, n1).
≠ NULL	Pointer to the first occurrence of (s2, n2) in (s1, n1).

Additional information

Locates the first occurrence of the octet string s2 of length n2 in the octet string s1 of length n1.

Conformance

Commonly found in Linux and BSD C libraries.

Thread safety

Safe.

4.18.3.4 strchr()

Description

Find character within string, forward.

Prototype

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

Parameters

Parameter	Description
s	String to search.
c	Character to search for.

Return value

Returns a pointer to the located character, or a null pointer if c does not occur in the string.

Additional information

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.

Thread safety

Safe.

4.18.3.5 strnchr()

Description

Find character within string, forward, limit length.

Prototype

```
char *strnchr(const char   * s,
                    size_t   n,
                    int      c);
```

Parameters

Parameter	Description
s	String to search.
n	Number of characters to search.
c	Character to search for.

Return value

Returns a pointer to the located character, or a null pointer if c does not occur in the string.

Additional information

Searches not more than n characters to locate 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.

Thread safety

Safe.

4.18.3.6 strrchr()

Description

Find character within string, reverse.

Prototype

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

Parameters

Parameter	Description
s	String to search.
c	Character to search for.

Return value

Returns a pointer to the located character, or a null pointer if c does not occur in the string.

Additional information

Locates the last 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.

Thread safety

Safe.

4.18.3.7 strlen()

Description

Calculate length of string.

Prototype

```
size_t strlen(const char * s);
```

Parameters

Parameter	Description
s	Pointer to zero-terminated string.

Return value

Returns the length of the string pointed to by s, that is the number of characters that precede the terminating null character.

Thread safety

Safe.

4.18.3.8 strnlen()

Description

Calculate length of string, limit length (POSIX.1).

Prototype

```
size_t strnlen(const char   * s,
                     size_t   n);
```

Parameters

Parameter	Description
s	Pointer to string.
n	Maximum number of characters to examine.

Return value

Returns the length of the string pointed to by s, up to a maximum of n characters. `strnlen()` only examines the first n characters of the string s.

Conformance

POSIX.1-2008.

Thread safety

Safe.

4.18.3.9 strnlen_s()

Description

Calculate length of string, limit length (C11).

Prototype

```
size_t strnlen_s(const char   * s,
                       size_t   n);
```

Parameters

Parameter	Description
s	Pointer to string or NULL.
n	Maximum number of characters to examine.

Return value

If s is NULL, returns 0. If s is nonnull, returns the length of the string pointed to by s, up to a maximum of n characters. strnlen_s() only examines the first n characters of the string s.

Additional information

The macro __STDC_WANT_LIB_EXT1__ must be set to 1 before including <string.h> to access this function.

Conformance

ISO/IEC 9899:2011 (C11).

Thread safety

Safe.

4.18.3.10 strstr()

Description

Find string within string, forward.

Prototype

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

Parameters

Parameter	Description
s1	String to search.
s2	String to search for.

Return value

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, `strstr()` returns s1.

Additional information

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.

Thread safety

Safe.

4.18.3.11 strnstr()

Description

Find string within string, forward, limit length (BSD).

Prototype

```
char *strnstr(const char   * s1,
              const char   * s2,
                    size_t   n);
```

Parameters

Parameter	Description
s1	String to search.
s2	String to search for.
n	Maximum number of characters to search for.

Return value

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, `strnstr()` returns s1.

Additional information

Searches at most n characters to locate 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.

Conformance

Commonly found in Linux and BSD C libraries.

Thread safety

Safe.

4.18.3.12 strcasestr()

Description

Find string within string, forward, ignore case (BSD).

Prototype

```
char *strcasestr(const char * s1,
                 const char * s2);
```

Parameters

Parameter	Description
s1	String to search.
s2	String to search for.

Return value

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, returns s1.

Additional information

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 without regard to character case.

Conformance

This extension is commonly found in Linux and BSD C libraries.

Thread safety

Safe.

4.18.3.13 strncasestr()

Description

Find string within string, forward, ignore case, limit length (BSD).

Prototype

```
char *strncasestr(const char   * s1,
                  const char   * s2,
                        size_t   n);
```

Parameters

Parameter	Description
s1	String to search.
s2	String to search for.
n	Maximum number of characters to compare in s2.

Return value

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, returns s1.

Additional information

Searches at most n characters to locate 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 without regard to character case.

Conformance

This extension is commonly found in Linux and BSD C libraries.

Thread safety

Safe.

4.18.3.14 strpbrk()

Description

Find first occurrence of characters within string.

Prototype

```
char *strpbrk(const char * s1,
              const char * s2);
```

Parameters

Parameter	Description
s1	Pointer to string to search.
s2	Pointer to string to search for.

Return value

Returns a pointer to the first character, or a null pointer if no character from s2 occurs in s1.

Additional information

Locates the first occurrence in the string pointed to by s1 of any character from the string pointed to by s2.

Thread safety

Safe.

4.18.3.15 strspn()

Description

Compute size of string prefixed by a set of characters.

Prototype

```
size_t strspn(const char * s1,
              const char * s2);
```

Parameters

Parameter	Description
s1	Pointer to zero-terminated string to search.
s2	Pointer to zero-terminated acceptable-set string.

Return value

Returns the length of the string pointed to by s1 which consists entirely of characters from the string pointed to by s2

Additional information

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.

Thread safety

Safe.

4.18.3.16 strcspn()

Description

Compute size of string not prefixed by a set of characters.

Prototype

```
size_t strcspn(const char * s1,
               const char * s2);
```

Parameters

Parameter	Description
s1	Pointer to string to search.
s2	Pointer to string containing characters to skip.

Return value

Returns the length of the segment of string s1 prefixed by characters from s2.

Additional information

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.

Thread safety

Safe.

4.18.3.17 strtok()

Description

Break string into tokens.

Prototype

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

Parameters

Parameter	Description
s1	Pointer to zero-terminated string to parse.
s2	Pointer to zero-terminated set of separators.

Return value

`NULL` if no further tokens else a pointer to the next token.

Additional information

A sequence of calls to `strtok()` 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 a non-null first argument; subsequent calls in the sequence have a null first argument. The separator string pointed to by `s2` may be different from call to call.

The first call in the sequence searches the string pointed to by `s1` for the first character that is not contained in the current separator string pointed to by `s2`. If no such character is found, then there are no tokens in the string pointed to by `s1` and `strtok()` returns a null pointer. If such a character is found, it is the start of the first token.

`strtok()` then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by `s1`, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. `strtok()` saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Thread safety

Safe [if configured].

See also

`strsep()`, `strtok_r()`.

4.18.3.18 strtok_r()

Description

Break string into tokens, restartable (POSIX.1).

Prototype

```
char *strtok_r(      char  * s1,
               const char  * s2,
                     char ** lasts);
```

Parameters

Parameter	Description
s1	Pointer to zero-terminated string to parse.
s2	Pointer to zero-terminated set of separators.
lasts	Pointer to pointer to char that maintains parse state.

Return value

NULL if no further tokens else a pointer to the next token.

Additional information

`strtok_r()` is a restartable version of the function `strtok()` where the state is maintained in the object of type char * pointed to by s3.

Conformance

POSIX.1-2008.

Thread safety

Safe.

See also

`strtok()`

4.18.3.19 strsep()

Description

Break string into tokens (BSD).

Prototype

```
char *strsep(      char ** stringp,
             const char  * delim);
```

Parameters

Parameter	Description
stringp	Pointer to pointer to zero-terminated string.
delim	Pointer to delimiter set string.

Return value

See below.

Additional information

Locates, in the string referenced by *stringp, the first occurrence of any character in the string delim (or the terminating null character) and replaces it with a null character. The location of the next character after the delimiter character (or NULL, if the end of the string was reached) is stored in *stringp. The original value of *stringp is returned.

An empty field (that is, a character in the string delim occurs as the first character of *stringp) can be detected by comparing the location referenced by the returned pointer to the null wide character.

If *stringp is initially null, strsep() returns null.

Conformance

Commonly found in Linux and BSD C libraries.

Thread safety

Safe.

4.18.4 Miscellaneous functions

Function	Description
`strerror()`	Decode error code.

4.18.4.1 strerror()

Description

Decode error code.

Prototype

```
char *strerror(int num);
```

Parameters

Parameter	Description
`num`	Error number.

Return value

Returns a pointer to the message string. The program must not modify the returned message string. The message may be overwritten by a subsequent call to `strerror()`.

Additional information

Maps the number in `num` to a message string. Typically, the values for `num` come from errno, but `strerror()` can map any value of type int to a message.

Thread safety

Safe.

4.19 <time.h>

4.19.1 Operations

Function	Description
`mktime()`	Convert a struct tm to `time_t`.
`difftime()`	Calculate difference between two times.

© 2003-2022 SEGGER Microcontroller GmbH

4.19.1.1 mktime()

Description

Convert a struct tm to `time_t`.

Prototype

```
time_t mktime(tm * tp);
```

Parameters

Parameter	Description
`tp`	Pointer to time object.

Return value

Number of seconds since UTC 1 January 1970 of the validated object.

Additional information

Validates (and updates) the object pointed to by `tp` to ensure that the `tm_sec`, `tm_min`, `tm_hour`, and `tm_mon` fields are within the supported integer ranges and the `tm_mday`, `tm_mon` and `tm_year` fields are consistent. The validated object is converted to the number of seconds since UTC 1 January 1970 and returned.

Thread safety

Safe.

4.19.1.2 difftime()

Description

Calculate difference between two times.

Prototype

```
double difftime(time_t time2,
                time_t time1);
```

Parameters

Parameter	Description
`time2`	End time.
`time1`	Start time.

Return value

returns `time2`-`time1` as a double precision number.

Thread safety

Safe.

4.19.2 Conversion functions

Function	Description
`ctime()`	Convert `time_t` to a string.
`ctime_r()`	Convert `time_t` to a string, reentrant.
`asctime()`	Convert `time_t` to a string.
`asctime_r()`	Convert `time_t` to a string, reentrant.
`gmtime()`	Convert `time_t` to struct tm.
`gmtime_r()`	Convert `time_t` to struct tm, reentrant.
`localtime()`	Convert time to local time.
`localtime_r()`	Convert time to local time, reentrant.
`strftime()`	Convert time to a string.
`strftime_l()`	Convert time to a string.

4.19.2.1 ctime()

Description

Convert `time_t` to a string.

Prototype

```
char *ctime(const time_t * tp);
```

Parameters

Parameter	Description
`tp`	Pointer to time to convert.

Return value

Pointer to zero-terminated converted string.

Additional information

Converts the time pointed to by `tp` to a null-terminated string.

Notes

The returned string is held in a static buffer: this function is not thread safe.

Thread safety

Unsafe.

4.19.2.2 ctime_r()

Description

Convert `time_t` to a string, reentrant.

Prototype

```
char *ctime_r(const time_t * tp,
                    char   * buf);
```

Parameters

Parameter	Description
`tp`	Pointer to time to convert.
`buf`	Pointer to array of characters that receives the zero-terminated string; the array must be at least 26 characters in length.

Return value

Returns the value of buf.

Additional information

Converts the time pointed to by `tp` to a null-terminated string.

Notes

The returned string is held in a static buffer: this function is not thread safe.

Thread safety

Safe.

4.19.2.3 asctime()

Description

Convert `time_t` to a string.

Prototype

```
char *asctime(const tm * tp);
```

Parameters

Parameter	Description
`tp`	Pointer to time to convert.

Return value

Pointer to zero-terminated converted string.

Additional information

Converts the time pointed to by `tp` to a null-terminated string of the form `Sun Sep 16 01:03:52 1973`. The returned string is held in a static buffer.

Notes

The returned string is held in a static buffer: this function is not thread safe.

Thread safety

Unsafe.

4.19.2.4 asctime_r()

Description

Convert `time_t` to a string, reentrant.

Prototype

```
char *asctime_r(const tm   * tp,
                      char * buf);
```

Parameters

Parameter	Description
`tp`	Pointer to time to convert.
`buf`	Pointer to array of characters that receives the zero-terminated string; the array must be at least 26 characters in length.

Return value

Returns the value of buf.

Additional information

Converts the time pointed to by `tp` to a null-terminated string of the `Sun Sep 16 01:03:52 1973`. The converted string is written into the array pointed to by buf.

Thread safety

Safe.

4.19.2.5 gmtime()

Description

Convert `time_t` to struct tm.

Prototype

```
gmtime(const time_t * tp);
```

Parameters

Parameter	Description
`tp`	Pointer to time to convert.

Return value

Pointer to converted time.

Additional information

Converts the time pointed to by `tp` to a struct tm.

Notes

The returned pointer points to a static buffer: this function is not thread safe.

Thread safety

Unsafe.

4.19.2.6 gmtime_r()

Description

Convert time_t to struct tm, reentrant.

Prototype

```
gmtime_r(const time_t * tp,
                        tm *tm);
```

Parameters

Parameter	Description
tp	Pointer to time to convert.
tm	Pointer to object that receives the converted time.

Return value

Returns tm.

Additional information

Converts the time pointed to by tp to a struct tm.

Thread safety

Safe.

4.19.2.7 localtime()

Description

Convert time to local time.

Prototype

```
localtime(const time_t * tp);
```

Parameters

Parameter	Description
tp	Pointer to time to convert.

Return value

Pointer to a statically-allocated object holding the local time.

Additional information

Converts the time pointed to by tp to local time format.

Notes

The returned pointer points to a static object: this function is not thread safe.

Thread safety

Unsafe.

4.19.2.8 localtime_r()

Description

Convert time to local time, reentrant.

Prototype

```
localtime_r(const time_t * tp,
                           tm *tm);
```

Parameters

Parameter	Description
tp	Pointer to time to convert.
tm	Pointer to object that receives the converted local time.

Return value

Returns tm.

Additional information

Converts the time pointed to by tp to local time format and writes it to the object pointed to by tm.

Thread safety

Safe.

4.19.2.9 strftime()

Description

Convert time to a string.

Prototype

```
size_t strftime(      char   * s,
                      size_t   smax,
                const char   * fmt,
                const tm     * tp);
```

Parameters

Parameter	Description
s	Pointer to object that receives the converted string.
smax	Maximum number of characters written to the array pointed to by s.
fmt	Pointer to zero-terminated format control string.
tp	Pointer to time to convert.

Return value

Returns the name of the current locale.

Additional information

Formats the time pointed to by tp to a null-terminated string of maximum size smax-1 into the pointed to by *s based on the fmt format string. The format string consists of conversion specifications and ordinary characters. Conversion specifications start with a "%" character followed by an optional "#" character.

The following conversion specifications are supported:

Specification	Description
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%#c	Date and time formatted as "%A, %B %#d, %Y, %H:%M:%S" (Microsoft extension)
%C	Century number
%d	Day of month as a decimal number [01,31]
%#d	Day of month without leading zero [1,31]
%D	Date in the form %m/%d/%y (POSIX.1-2008 extension)
%e	Day of month [1,31], single digit preceded by space
%F	Date in the format %Y-%m-%d
%h	Abbreviated month name as %b
%H	Hour in 24-hour format [00,23]
%#H	Hour in 24-hour format without leading zeros [0,23]
%I	Hour in 12-hour format [01,12]
%#I	Hour in 12-hour format without leading zeros [1,12]
%j	Day of year as a decimal number [001,366]

Specification	Description
%#j	Day of year as a decimal number without leading zeros [1,366]
%k	Hour in 24-hour clock format [0,23] (POSIX.1-2008 extension)
%l	Hour in 12-hour clock format [0,12] (POSIX.1-2008 extension)
%m	Month as a decimal number [01,12]
%#m	Month as a decimal number without leading zeros [1,12]
%M	Minute as a decimal number [00,59]
%#M	Minute as a decimal number without leading zeros [0,59]
%n	Insert newline character (POSIX.1-2008 extension)
%p	Locale's a.m or p.m indicator for 12-hour clock
%r	Time as %I:%M:%s %p (POSIX.1-2008 extension)
%R	Time as %H:%M (POSIX.1-2008 extension)
%S	Second as a decimal number [00,59]
%t	Insert tab character (POSIX.1-2008 extension)
%T	Time as %H:%M:%S
%#S	Second as a decimal number without leading zeros [0,59]
%U	Week of year as a decimal number [00,53], Sunday is first day of the week
%#U	Week of year as a decimal number without leading zeros [0,53], Sunday is first day of the week
%w	Weekday as a decimal number [0,6], Sunday is 0
%W	Week number as a decimal number [00,53], Monday is first day of the week
%#W	Week number as a decimal number without leading zeros [0,53], Monday is first day of the week
%x	Locale's date representation
%#x	Locale's long date representation
%X	Locale's time representation
%y	Year without century, as a decimal number [00,99]
%#y	Year without century, as a decimal number without leading zeros [0,99]
%Y	Year with century, as decimal number
%z,%Z	Timezone name or abbreviation
%%	%

Thread safety

Safe [if configured].

4.19.2.10 strftime_l()

Description

Convert time to a string.

Prototype

```
size_t strftime_l(      char   * s,
                        size_t   smax,
                  const char   * fmt,
                  const tm     * tp,
                                 locale_t loc);
```

Parameters

Parameter	Description
s	Pointer to object that receives the converted string.
smax	Maximum number of characters written to the array pointed to by s.
fmt	Pointer to zero-terminated format control string.
tp	Pointer to time to convert.
loc	Locale to use for conversion.

Return value

Returns the name of the current locale.

Additional information

Formats the time pointed to by tp to a null-terminated string of maximum size smax-1 into the pointed to by *s based on the fmt format string and using the locale loc.

The format string consists of conversion specifications and ordinary characters. Conversion specifications start with a "%" character followed by an optional "#" character.

See strftime() for a description of the format conversion specifications.

Thread safety

Safe.

4.20 <wchar.h>

4.20.1 Copying functions

Function	Description
`wmemset()`	Set memory to wide character.
`wmemcpy()`	Copy memory.
`wmemccpy()`	Copy memory, specify terminator (POSIX.1).
`wmempcpy()`	Copy memory (GNU).
`wmemmove()`	Copy memory, tolerate overlaps.
`wcscpy()`	Copy string.
`wcsncpy()`	Copy string, limit length.
`wcslcpy()`	Copy string, limit length, always zero terminate (BSD).
`wcscat()`	Concatenate strings.
`wcsncat()`	Concatenate strings, limit length.
`wcslcat()`	Concatenate strings, limit length, always zero terminate (BSD).
`wcsdup()`	Duplicate string (POSIX.1).
`wcsndup()`	Duplicate string, limit length (GNU).
`wcsxfrm()`	Transform strings.

4.20.1.1 wmemset()

Description

Set memory to wide character.

Prototype

```
wchar_t *wmemset(wchar_t * s,
                 wchar_t   c,
                 size_t    n);
```

Parameters

Parameter	Description
s	Pointer to destination object.
c	Wide character to copy.
n	Length of destination object in wide characters.

Return value

Returns s.

Additional information

Copies the value of c into each of the first n wide characters of the object pointed to by s.

Thread safety

Safe.

4.20.1.2 wmemcpy()

Description

Copy memory.

Prototype

```
wchar_t *wmemcpy(      wchar_t * s1,
                 const wchar_t * s2,
                       size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s2	Pointer to source object.
n	Number of wide characters to copy.

Return value

Returns the value of s1.

Additional information

Copies n wide characters from the object pointed to by s2 into the object pointed to by s1. The behavior of wmemcpy() is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.20.1.3 wmemccpy()

Description

Copy memory, specify terminator (POSIX.1).

Prototype

```
wchar_t *wmemccpy(      wchar_t * s1,
                  const wchar_t * s2,
                        wchar_t   c,
                        size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s2	Pointer to source object.
c	Character that terminates copy.
n	Maximum number of characters to copy.

Return value

Returns a pointer to the wide character immediately following `c` in `s1`, or `NULL` if `c` was not found in the first `n` wide characters of s2.

Additional information

Copies at most `n` wide characters from the object pointed to by `s2` into the object pointed to by `s1`. The copying stops as soon as `n` wide characters are copied or the wide character `c` is copied into the destination object pointed to by s1.

The behavior of `wmemccpy()` is undefined if copying takes place between objects that overlap.

Notes

Conforms to POSIX.1-2008.

Thread safety

Safe.

4.20.1.4 wmempcpy()

Description

Copy memory (GNU).

Prototype

```
wchar_t *wmempcpy(      wchar_t * s1,
                  const wchar_t * s2,
                        size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s2	Pointer to source object.
n	Number of wide characters to copy.

Return value

Returns a pointer to the wide character immediately following the final wide character written into s1.

Additional information

Copies n wide characters from the object pointed to by s2 into the object pointed to by s1. The behavior of wmempcpy() is undefined if copying takes place between objects that overlap.

Notes

This is an extension found in GNU libc.

Thread safety

Safe.

4.20.1.5 wmemmove()

Description

Copy memory, tolerate overlaps.

Prototype

```
wchar_t *wmemmove(      wchar_t * s1,
                  const wchar_t * s2,
                        size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to destination object.
s2	Pointer to source object.
n	Number of wide characters to copy.

Return value

Returns the value of s1.

Additional information

Copies n wide characters from the object pointed to by s2 into the object pointed to by s1 ensuring that if s1 and s2 overlap, the copy works correctly. Copying takes place as if the n wide characters from the object pointed to by s2 are first copied into a temporary array of n wide characters that does not overlap the objects pointed to by s1 and s2, and then the n wide characters from the temporary array are copied into the object pointed to by s1.

Thread safety

Safe.

4.20.1.6 wcscpy()

Description

Copy string.

Prototype

```
wchar_t *wcscpy(      wchar_t * s1,
                const wchar_t * s2);
```

Parameters

Parameter	Description
s1	Pointer to wide string to copy to.
s2	Pointer to wide string to copy.

Return value

Returns the value of s1.

Additional information

Copies the wide string pointed to by s2 (including the terminating null wide character) into the array pointed to by s1. The behavior of wcscpy() is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.20.1.7 wcsncpy()

Description

Copy string, limit length.

Prototype

```
wchar_t *wcsncpy(      wchar_t * s1,
                 const wchar_t * s2,
                       size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to wide string to copy to.
s2	Pointer to wide string to copy.
n	Maximum number of wide characters to copy.

Return value

Returns the value of s1.

Additional information

Copies not more than n wide characters from the array pointed to by s2 to the array pointed to by s1. Wide characters that follow a null wide character in s2 are not copied. The behavior of `wcsncpy()` is undefined if copying takes place between objects that overlap. If the array pointed to by s2 is a wide string that is shorter than n wide characters, null wide characters are appended to the copy in the array pointed to by s1, until n characters in all have been written.

Notes

No wide null character is implicitly appended to the end of s1, so s1 will only be terminated by a wide null character if the length of the wide string pointed to by s2 is less than n.

Thread safety

Safe.

4.20.1.8 wcslcpy()

Description

Copy string, limit length, always zero terminate (BSD).

Prototype

```
size_t wcslcpy(      wchar_t * s1,
               const wchar_t * s2,
                     size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to wide string to copy to.
s2	Pointer to wide string to copy.
n	Maximum number of wide characters, including terminating null, in s1.

Return value

Returns the number of wide characters it tried to copy, which is the length of the wide string s2 or n, whichever is smaller.

Additional information

Copies up to n-1 wide characters from the wide string pointed to by s2 into the array pointed to by s1 and always terminates the result with a null character.

The behavior of strlcpy() is undefined if copying takes place between objects that overlap.

Notes

Commonly found in BSD libraries and contrasts with wcsncpy() in that the resulting string is always terminated with a null wide character.

Thread safety

Safe.

4.20.1.9 wcscat()

Description

Concatenate strings.

Prototype

```
wchar_t *wcscat(      wchar_t * s1,
                const wchar_t * s2);
```

Parameters

Parameter	Description
s1	Zero-terminated wide string to append to.
s2	Zero-terminated wide string to append.

Return value

Returns the value of `s1`.

Additional information

Appends a copy of the wide string pointed to by `s2` (including the terminating null wide character) to the end of the wide string pointed to by `s1`. The initial character of `s2` overwrites the null wide character at the end of `s1`. The behavior of `wcscat()` is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.20.1.10 wcsncat()

Description

Concatenate strings, limit length.

Prototype

```
wchar_t *wcsncat(      wchar_t * s1,
                 const wchar_t * s2,
                       size_t    n);
```

Parameters

Parameter	Description
s1	Wide string to append to.
s2	Wide string to append.
n	Maximum number of wide characters in s1.

Return value

Returns the value of s1.

Additional information

Appends not more than n wide characters from the array pointed to by s2 to the end of the wide string pointed to by s1. A null wide character in s1 and wide characters that follow it are not appended. The initial wide character of s2 overwrites the null wide character at the end of s1. A terminating wide null character is always appended to the result. The behavior of wcsncat() is undefined if copying takes place between objects that overlap.

Thread safety

Safe.

4.20.1.11 wcslcat()

Description

Concatenate strings, limit length, always zero terminate (BSD).

Prototype

```
size_t wcslcat(      char   * s1,
               const char   * s2,
                     size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to wide string to append to.
s2	Pointer to wide string to append.
n	Maximum number of characters, including terminating wide null, in s1.

Return value

Returns the number of wide characters it tried to copy, which is the sum of the lengths of the wide strings s1 and s2 or n, whichever is smaller.

Additional information

Appends no more than n-strlen(s1}-1 wide characters pointed to by s2 into the array pointed to by s1 and always terminates the result with a wide null character if n is greater than zero. Both the wide strings s1 and s2 must be terminated with a wide null character on entry to wcslcat() and a character position for the terminating wide null should be included in n.

The behavior of wcslcat() is undefined if copying takes place between objects that overlap.

Notes

Commonly found in BSD libraries.

Thread safety

Safe.

4.20.1.12 wcsdup()

Description

Duplicate string (POSIX.1).

Prototype

```
wchar_t *wcsdup(const wchar_t * s);
```

Parameters

Parameter	Description
s	Pointer to wide string to duplicate.

Return value

Returns a pointer to the new wide string or a null pointer if the new wide string cannot be created. The returned pointer can be passed to `free()`.

Additional information

Duplicates the wide string pointed to by `s` by using `malloc()` to allocate memory for a copy of `s` and then copies `s`, including the terminating null, to that memory

Notes

Conforms to POSIX.1-2008 and SC22 TR 24731-2.

Thread safety

Safe.

4.20.1.13 wcsndup()

Description

Duplicate string, limit length (GNU).

Prototype

```
wchar_t *wcsndup(const wchar_t * s,
                       size_t    n);
```

Parameters

Parameter	Description
s	Pointer to wide string to duplicate.
n	Maximum number of wide characters to duplicate.

Return value

Returns a pointer to the new wide string or a null pointer if the new wide string cannot be created. The returned pointer can be passed to `free()`.

Additional information

Duplicates at most n wide characters from the the string pointed to by s by using `malloc()` to allocate memory for a copy of s.

If the length of string pointed to by s is greater than n wide characters, only n wide characters will be duplicated. If n is greater than the length of the wide string pointed to by s, all characters in the string are copied into the allocated array including the terminating null character.

Notes

This is a GNU extension.

Thread safety

Safe.

4.20.1.14 wcsxfrm()

Description

Transform strings.

Prototype

```
size_t wcsxfrm(      wchar_t * s1,
               const wchar_t * s2,
                     size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to destination array.
s2	Pointer to source string.
n	Maximum number of characters in the destination array.

Return value

Returns the length of the transformed string. If the value returned is n or more, the contents of the array pointed to by s1 are undefined.

Thread safety

Safe.

4.20.2 Comparison functions

Function	Description
`wmemcmp()`	Compare memory.
`wcscmp()`	Compare strings.
`wcsncmp()`	Compare strings, limit length.
`wcscasecmp()`	Compare strings, ignore case (POSIX.1).
`wcsncasecmp()`	Compare strings, ignore case, limit length (POSIX.1).
`wcscoll()`	Collate strings.

4.20.2.1 wmemcmp()

Description

Compare memory.

Prototype

```
int wmemcmp(const wchar_t * s1,
            const wchar_t * s2,
                  size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to object #1.
s2	Pointer to object #2.
n	Number of wide characters to compare.

Return value

< 0	s1 is less than s2.
= 0	s1 is equal to s2.
> 0	s1 is greater than to s2.

Additional information

Compares the first n wide characters of the object pointed to by s1 to the first n wide characters of the object pointed to by s2. wmemcmp() returns an integer greater than, equal to, or less than zero as the object pointed to by s1 is greater than, equal to, or less than the object pointed to by s2.

Thread safety

Safe.

4.20.2.2 wcscmp()

Description

Compare strings.

Prototype

```
int wcscmp(const wchar_t * s1,
           const wchar_t * s2);
```

Parameters

Parameter	Description
s1	Pointer to wide string #1.
s2	Pointer to wide string #2.

Return value

Returns an integer greater than, equal to, or less than zero, if the null-terminated wide string pointed to by s1 is greater than, equal to, or less than the null-terminated wide string pointed to by s2.

Thread safety

Safe.

4.20.2.3 wcsncmp()

Description

Compare strings, limit length.

Prototype

```
int wcsncmp(const wchar_t * s1,
            const wchar_t * s2,
                  size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to wide string #1.
s2	Pointer to wide string #2.
n	Maximum number of wide characters to compare.

Return value

Returns an integer greater than, equal to, or less than zero, if 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.

Additional information

Compares not more than n wide characters from the array pointed to by s1 to the array pointed to by s2. Wide characters that follow a null wide character are not compared.

Thread safety

Safe.

4.20.2.4 wcscasecmp()

Description

Compare strings, ignore case (POSIX.1).

Prototype

```
int wcscasecmp(const char * s1,
               const char * s2);
```

Parameters

Parameter	Description
s1	Pointer to wide string #1.
s2	Pointer to wide string #2.

Return value

< 0	s1 is less than s2.
= 0	s1 is equal to s2.
> 0	s1 is greater than to s2.

Additional information

Compares the wide string pointed to by s1 to the wide string pointed to by s2 ignoring differences in case.

wcscasecmp() returns an integer greater than, equal to, or less than zero if the wide string pointed to by s1 is greater than, equal to, or less than the wide string pointed to by s2.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.20.2.5 wcsncasecmp()

Description

Compare strings, ignore case, limit length (POSIX.1).

Prototype

```
int wcsncasecmp(const char   * s1,
                const char   * s2,
                      size_t   n);
```

Parameters

Parameter	Description
s1	Pointer to wide string #1.
s2	Pointer to wide string #2.
n	Maximum number of wide characters to compare.

Return value

< 0	s1 is less than s2.
= 0	s1 is equal to s2.
> 0	s1 is greater than to s2.

Additional information

Compares not more than n wide characters from the array pointed to by s1 to the array pointed to by s2 ignoring differences in case. Characters that follow a wide null character are not compared.

strncasecmp() returns an integer greater than, equal to, or less than zero, if 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.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.20.2.6 wcscoll()

Description

Collate strings.

Prototype

```
int wcscoll(const wchar_t * s1,
            const wchar_t * s2);
```

Parameters

Parameter	Description
s1	Pointer to wide string #1.
s2	Pointer to wide string #2.

Return value

Returns an integer greater than, equal to, or less than zero, if the null-terminated wide string pointed to by s1 is greater than, equal to, or less than the null-terminated wide string pointed to by s2.

Thread safety

Safe.

4.20.3 Search functions

Function	Description
`wmemchr()`	Find character in memory, forward.
`wcschr()`	Find character within string, forward.
`wcsnchr()`	Find character within string, forward, limit length.
`wcsrchr()`	Find character within string, reverse.
`wcslen()`	Calculate length of string.
`wcsnlen()`	Calculate length of string, limit length (POSIX.1).
`wcsstr()`	Find string within string, forward.
`wcsnstr()`	Find string within string, forward, limit length (BSD).
`wcspbrk()`	Find first occurrence of characters within string.
`wcsspn()`	Compute size of string prefixed by a set of characters.
`wcscspn()`	Compute size of string not prefixed by a set of characters.
`wcstok()`	Break string into tokens.
`wcssep()`	Break string into tokens (BSD).

4.20.3.1 wmemchr()

Description

Find character in memory, forward.

Prototype

```
wchar_t *wmemchr(const wchar_t * s,
                       wchar_t   c,
                       size_t    n);
```

Parameters

Parameter	Description
s	Pointer to object to search.
c	Wide character to search for.
n	Number of wide characters in object to search.

Return value

= NULL	c does not occur in the object.
≠ NULL	Pointer to the located wide character.

Additional information

Locates the first occurrence of c in the initial n wide characters of the object pointed to by s. Unlike `wcschr()`, `wmemchr()` does not terminate a search when a null wide character is found in the object pointed to by s.

Thread safety

Safe.

4.20.3.2 wcschr()

Description

Find character within string, forward.

Prototype

```
wchar_t *wcschr(const wchar_t * s,
                      wchar_t   c);
```

Parameters

Parameter	Description
s	Wide string to search.
c	Wide character to search for.

Return value

Returns a pointer to the located wide character, or a null pointer if c does not occur in the wide string.

Additional information

Locates the first occurrence of c in the wide string pointed to by s. The terminating wide null character is considered to be part of the string.

Thread safety

Safe.

4.20.3.3 wcsnchr()

Description

Find character within string, forward, limit length.

Prototype

```
wchar_t *wcsnchr(const wchar_t * s,
                       size_t    n,
                       wchar_t   c);
```

Parameters

Parameter	Description
s	Pointer to wide string to search.
n	Number of wide characters to search.
c	Wide character to search for.

Return value

Returns a pointer to the located wide character, or a null pointer if c does not occur in the string.

Additional information

Searches not more than n wide characters to locate the first occurrence of c in the wide string pointed to by s. The terminating wide null character is considered to be part of the wide string.

Thread safety

Safe.

4.20.3.4 wcsrchr()

Description

Find character within string, reverse.

Prototype

```
wchar_t *wcsrchr(const wchar_t * s,
                       wchar_t   c);
```

Parameters

Parameter	Description
s	Pointer to wide string to search.
c	Wide character to search for.

Return value

Returns a pointer to the located wide character, or a null pointer if c does not occur in the string.

Additional information

Locates the last occurrence of c in the wide string pointed to by s. The terminating wide null character is considered to be part of the string.

Thread safety

Safe.

4.20.3.5 wcslen()

Description

Calculate length of string.

Prototype

```
size_t wcslen(const wchar_t * s);
```

Parameters

Parameter	Description
s	Pointer to zero-terminated wide string.

Return value

Returns the length of the wide string pointed to by s, that is the number of wide characters that precede the terminating wide null character.

Thread safety

Safe.

4.20.3.6 wcsnlen()

Description

Calculate length of string, limit length (POSIX.1).

Prototype

```
size_t wcsnlen(const wchar_t * s,
                     size_t    n);
```

Parameters

Parameter	Description
s	Pointer to wide string.
n	Maximum number of wide characters to examine.

Return value

Returns the length of the wide string pointed to by s, up to a maximum of n wide characters. wcsnlen() only examines the first n wide characters of the string s.

Notes

Conforms to POSIX.1-2008.

Thread safety

Safe.

4.20.3.7 wcsstr()

Description

Find string within string, forward.

Prototype

```
wchar_t *wcsstr(const wchar_t * s1,
                const wchar_t * s2);
```

Parameters

Parameter	Description
s1	Pointer to wide string to search.
s2	Pointer to wide string to search for.

Return value

Returns a pointer to the located wide string, or a null pointer if the wide string is not found. If s2 points to a wide string with zero length, `wcsstr()` returns s1.

Additional information

Locates the first occurrence in the wide string pointed to by s1 of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by s2.

Thread safety

Safe.

4.20.3.8 wcsnstr()

Description

Find string within string, forward, limit length (BSD).

Prototype

```
wchar_t *wcsnstr(const wchar_t * s1,
                 const wchar_t * s2,
                       size_t    n);
```

Parameters

Parameter	Description
s1	Pointer to wide string to search.
s2	Pointer to wide string to search for.
n	Maximum number of characters to search for.

Return value

Returns a pointer to the located wide string, or a null pointer if the wide string is not found. If s2 points to a wide string with zero length, wcsnstr() returns s1.

Additional information

Searches at most n wide characters to locate the first occurrence in the wide string pointed to by s1 of the sequence of wide characters (excluding the terminating wide null character) in the string pointed to by s2.

Notes

Commonly found in Linux and BSD C libraries.

Thread safety

Safe.

4.20.3.9 wcspbrk()

Description

Find first occurrence of characters within string.

Prototype

```
wchar_t *wcspbrk(const wchar_t * s1,
                 const wchar_t * s2);
```

Parameters

Parameter	Description
s1	Pointer to wide string to search.
s2	Pointer to wide string to search for.

Return value

Returns a pointer to the first wide character, or a null pointer if no wide character from s2 occurs in s1.

Additional information

Locates the first occurrence in the wide string pointed to by s1 of any wide character from the string pointed to by s2.

Thread safety

Safe.

4.20.3.10 wcsspn()

Description

Compute size of string prefixed by a set of characters.

Prototype

```
size_t wcsspn(const wchar_t * s1,
              const wchar_t * s2);
```

Parameters

Parameter	Description
s1	Pointer to zero-terminated wide string to search.
s2	Pointer to zero-terminated acceptable-set wide string.

Return value

Returns the length of the wide string pointed to by s1 which consists entirely of wide characters from the wide string pointed to by s2

Additional information

Computes the length of the maximum initial segment of the wide string pointed to by s1 which consists entirely of wide characters from the string pointed to by s2.

Thread safety

Safe.

4.20.3.11 wcscspn()

Description

Compute size of string not prefixed by a set of characters.

Prototype

```
size_t wcscspn(const wchar_t * s1,
               const wchar_t * s2);
```

Parameters

Parameter	Description
s1	Pointer to wide string to search.
s2	Pointer to wide string containing characters to skip.

Return value

Returns the length of the segment of wide string s1 prefixed by wide characters from s2.

Additional information

Computes the length of the maximum initial segment of the wide string pointed to by s1 which consists entirely of wide characters not from the wide string pointed to by s2.

Thread safety

Safe.

4.20.3.12 wcstok()

Description

Break string into tokens.

Prototype

```
wchar_t *wcstok(      wchar_t  * s1,
                const wchar_t  * s2,
                      wchar_t ** ptr);
```

Parameters

Parameter	Description
s1	Pointer to zero-terminated wide string to parse.
s2	Pointer to zero-terminated set of separators.
ptr	Pointer to object that maintains parse state.

Return value

NULL if no further tokens else a pointer to the next token.

Additional information

A sequence of calls to wcstok() breaks the wide string pointed to by s1 into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by s2. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator wide string pointed to by s2 may be different from call to call.

The first call in the sequence searches the wide string pointed to by s1 for the wide first character that is not contained in the current separator wide string pointed to by s2. If no such wide character is found, then there are no tokens in the string pointed to by s1 and wcstok() returns a null pointer. If such a wide character is found, it is the start of the first token.

wcstok() then searches from there for a wide character that is contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by s1, and subsequent searches for a token will return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token. wcstok() saves a pointer to the following wide character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Thread safety

Safe.

See also

wcssep().

4.20.3.13 wcssep()

Description

Break string into tokens (BSD).

Prototype

```
wchar_t *wcssep(      wchar_t ** stringp,
                const wchar_t  * delim);
```

Parameters

Parameter	Description
stringp	Pointer to pointer to zero-terminated wide string.
delim	Pointer to delimiter set wide string.

Return value

See below.

Additional information

Locates, in the wide string referenced by *stringp, the first occurrence of any wide character in the wide string delim (or the terminating null character) and replaces it with a null wide character. The location of the next wide character after the delimiter wide character (or NULL, if the end of the wide string was reached) is stored in *stringp. The original value of *stringp is returned.

An empty field (that is, a wide character in the string delim occurs as the first character of *stringp) can be detected by comparing the location referenced by the returned pointer to the null wide character.

If *stringp is initially null, wcssep() returns null.

Notes

Commonly found in Linux and BSD C libraries.

Thread safety

Safe.

4.20.4 Multi-byte/wide string conversion functions

Function	Description
`mbsinit()`	Query initial conversion state.
`mbrlen()`	Count number of bytes in multi-byte character, restartable.
`mbrlen_l()`	Count number of bytes in multi-byte character, restartable, per locale (POSIX.1).
`mbrtowc()`	Convert multi-byte character to wide character, restartable.
`mbrtowc_l()`	Convert multi-byte character to wide character, restartable, per locale (POSIX.1).
`wctob()`	Convert wide character to single-byte character.
`wctob_l()`	Convert wide character to single-byte character, per locale (POSIX.1).
`wcrtomb()`	Convert wide character to multi-byte character, restartable.
`wcrtomb_l()`	Convert wide character to multi-byte character, restartable, per locale (POSIX.1).
`wcsrtombs()`	Convert wide string to multi-byte string, restartable.
`wcsrtombs_l()`	Convert wide string to multi-byte string, restartable (POSIX.1).
`wcsnrtombs()`	Convert wide string to multi-byte string, restartable (POSIX.1).
`wcsnrtombs_l()`	Convert wide string to multi-byte string, restartable (POSIX.1).

4.20.4.1 mbsinit()

Description

Query initial conversion state.

Prototype

```
int mbsinit(const mbstate_t * ps);
```

Parameters

Parameter	Description
ps	Pointer to conversion state.

Return value

Returns nonzero (true) if ps is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, returns zero.

Thread safety

Safe.

4.20.4.2 mbrlen()

Description

Count number of bytes in multi-byte character, restartable.

Prototype

```
size_t mbrlen(const char      * s,
                    size_t      n,
                    mbstate_t * ps);
```

Parameters

Parameter	Description
s	Pointer to multi-byte character.
n	Maximum number of bytes to examine.
ps	Pointer to multi-byte conversion state.

Return value

Number of bytes in multi-byte character.

Additional information

Determines the number of bytes contained in the multi-byte character pointed to by `s` in the current locale.

Except that except that the expression designated by `ps` is evaluated only once, this function is equivalent to the call:

```
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

where internal is the `mbstate_t` object for the `mbrlen()` function.

Thread safety

Safe [if configured].

See also

`mbrlen_l()`, `mbrtowc()`

4.20.4.3 mbrlen_l()

Description

Count number of bytes in multi-byte character, restartable, per locale (POSIX.1).

Prototype

```
size_t mbrlen_l(const char      * s,
                      size_t      n,
                      mbstate_t * ps,
                                  locale_t loc);
```

Parameters

Parameter	Description
s	Pointer to multi-byte character.
n	Maximum number of bytes to examine.
ps	Pointer to multi-byte conversion state.
loc	Locale used for conversion.

Return value

Number of bytes in multi-byte character.

Additional information

Determines the number of bytes contained in the multi-byte character pointed to by s in the locale loc.

Except that except that the expression designated by ps is evaluated only once, this function is equivalent to the call:

```
mbrtowc_l(NULL, s, n, ps != NULL ? ps : &internal, loc);
```

where internal is the mbstate_t object for the mbrlen() function,

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

See also

mbrlen_l(), mbrtowc()

4.20.4.4 mbrtowc()

Description

Convert multi-byte character to wide character, restartable.

Prototype

```
size_t mbrtowc(      wchar_t   * pwc,
               const char      * s,
                     size_t      n,
                     mbstate_t * ps);
```

Parameters

Parameter	Description
pwc	Pointer to object that receives the wide character.
s	Pointer to multi-byte character string.
n	Maximum number of bytes that will be examined.
ps	Pointer to multi-byte conversion state.

Return value

If s is a null pointer, mbrtowc() is equivalent to mbrtowc(NULL, "", 1, ps), ignoring pwc and n.

If s is not null and the object that s points to is a wide null character, mbrtowc() returns 0.

If s is not null and the object that s points to forms a valid multi-byte character in the current locale with a most n bytes, mbrtowc() returns the length in bytes of the multi-byte character and stores that wide character to the object pointed to by pwc (if pwc is not null).

If the object that s points to forms an incomplete, but possibly valid, multi-byte character, mbrtowc() returns -2.

If the object that s points to does not form a partial multi-byte character, mbrtowc() returns -1.

Additional information

Converts a single multi-byte character to a wide character in the current locale.

Thread safety

Safe [if configured].

See also

mbtowc(), mbrtowc_l()

4.20.4.5 mbrtowc_l()

Description

Convert multi-byte character to wide character, restartable, per locale (POSIX.1).

Prototype

```
size_t mbrtowc_l(      wchar_t   * pwc,
                 const char      * s,
                       size_t      n,
                       mbstate_t * ps,
                                   locale_t loc);
```

Parameters

Parameter	Description
pwc	Pointer to object that receives the wide character.
s	Pointer to multi-byte character string.
n	Maximum number of bytes that will be examined.
ps	Pointer to multi-byte conversion state.
loc	Locale used for conversion.

Return value

If s is a null pointer, `mbrtowc()` is equivalent to mbrtowc(NULL, "", 1, ps), ignoring pwc and n.

If s is not null and the object that s points to is a wide null character, `mbrtowc()` returns 0.

If s is not null and the object that s points to forms a valid multi-byte character in the locale loc with a most n bytes, `mbrtowc()` returns the length in bytes of the multi-byte character and stores that wide character to the object pointed to by pwc (if pwc is not null).

If the object that s points to forms an incomplete, but possibly valid, multi-byte character, `mbrtowc()` returns -2.

If the object that s points to does not form a partial multi-byte character, `mbrtowc()` returns -1.

Additional information

Converts a single multi-byte character to a wide character in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

See also

`mbtowc()`, `mbrtowc_l()`

4.20.4.6 wctob()

Description

Convert wide character to single-byte character.

Prototype

```
int wctob(wint_t c);
```

Parameters

Parameter	Description
c	Character to convert.

Return value

Returns EOF if c does not correspond to a multi-byte character with length one in the initial shift state in the current locale. Otherwise, it returns the single-byte representation of that character as an unsigned char converted to an int.

Additional information

Determines whether c corresponds to a member of the extended character set whose multi-byte character representation is a single byte in the current locale when in the initial shift state.

Thread safety

Safe [if configured].

4.20.4.7 wctob_l()

Description

Convert wide character to single-byte character, per locale (POSIX.1).

Prototype

```
int wctob_l(wint_t c,
            locale_t loc);
```

Parameters

Parameter	Description
c	Character to convert.
loc	Locale used for conversion.

Return value

Returns EOF if c does not correspond to a multi-byte character with length one in the initial shift state in the locale loc. Otherwise, it returns the single-byte representation of that character as an unsigned char converted to an int.

Additional information

Determines whether c corresponds to a member of the extended character set whose multi-byte character representation is a single byte in the locale loc when in the initial shift state.

Thread safety

Safe.

4.20.4.8 wcrtomb()

Description

Convert wide character to multi-byte character, restartable.

Prototype

```
size_t wcrtomb(char      * s,
                         wchar_t wc,
               mbstate_t * ps);
```

Parameters

Parameter	Description
s	Pointer to array that receives the multi-byte character.
wc	Wide character to convert.
ps	Pointer to multi-byte conversion state.

Return value

Returns the number of bytes stored in the array object. When wc is not a valid wide character, an encoding error occurs: `wcrtomb()` stores the value of the macro EILSEQ in errno and returns `(size_t)(-1)`; the conversion state is unspecified.

Additional information

If s is a null pointer, `wcrtomb()` is equivalent to the call wcrtomb(buf, 0, ps) where buf is an internal buffer.

If s is not a null pointer, `wcrtomb()` determines the number of bytes needed to represent the multi-byte character that corresponds to the wide character given by wc in the locale loc, and stores the multi-byte character representation in the array whose first element is pointed to by s. At most `MB_CUR_MAX` bytes are stored. If wc is a null wide character, a null byte is stored; the resulting state described is the initial conversion state.

Thread safety

Safe [if configured].

4.20.4.9 wcrtomb_l()

Description

Convert wide character to multi-byte character, restartable, per locale (POSIX.1).

Prototype

```
size_t wcrtomb_l(char      * s,
                   wchar_t wc,
             mbstate_t * ps,
                   locale_t loc);
```

Parameters

Parameter	Description
s	Pointer to array that receives the multi-byte character.
wc	Wide character to convert.
ps	Pointer to multi-byte conversion state.
loc	Locale used for conversion.

Return value

Returns the number of bytes stored in the array object. When wc is not a valid wide character, an encoding error occurs: `wcrtomb_l()` stores the value of the macro EILSEQ in errno and returns (size_t)(-1); the conversion state is unspecified.

Additional information

If s is a null pointer, `wcrtomb()` is equivalent to the call wcrtomb(buf, 0, ps) where buf is an internal buffer.

If s is not a null pointer, `wcrtomb()` determines the number of bytes needed to represent the multi-byte character that corresponds to the wide character given by wc in the current locale, and stores the multi-byte character representation in the array whose first element is pointed to by s. At most `MB_CUR_MAX` bytes are stored. If wc is a null wide character, a null byte is stored; the resulting state described is the initial conversion state.

Thread safety

Safe.

4.20.4.10 wcsrtombs()

Description

Convert wide string to multi-byte string, restartable.

Prototype

```
size_t wcsrtombs(      char        * dst,
                 const wchar_t    ** src,
                       size_t        len,
                       mbstate_t   * ps);
```

Parameters

Parameter	Description
dst	Pointer to array that receives the multi-byte string.
src	Indirect pointer to wide character string being converted.
len	Maximum number of bytes to write into the array pointed to by dst.
ps	Pointer to multi-byte conversion state.

Return value

If conversion stops because a wide character is reached that does not correspond to a valid multi-byte character, an encoding error occurs: wcsrtombs() stores the value of the macro EILSEQ in errno and returns (size_t)(-1); the conversion state is unspecified. Otherwise, it returns the number of bytes in the resulting multi-byte character sequence, not including the terminating null character (if any).

Additional information

Converts a sequence of wide characters in the current locale from the array indirectly pointed to by src into a sequence of corresponding multi-byte characters that begins in the conversion state described by the object pointed to by ps. If dst is not a null pointer, the converted characters are then stored into the array pointed to by dst. Conversion continues up to and including a terminating null wide character, which is also stored.

Conversion stops earlier in two cases: when a wide character is reached that does not correspond to a valid multi-byte character, or (if dst is not a null pointer) when the next multi-byte character would exceed the limit of len total bytes to be stored into the array pointed to by dst. Each conversion takes place as if by a call to wcrtomb().

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Thread safety

Safe [if configured].

4.20.4.11 wcsrtombs_l()

Description

Convert wide string to multi-byte string, restartable (POSIX.1).

Prototype

```
size_t wcsrtombs_l(      char       * dst,
                   const wchar_t   ** src,
                         size_t        len,
                         mbstate_t   * ps,
                                       locale_t loc);
```

Parameters

Parameter	Description
dst	Pointer to array that receives the multi-byte string.
src	Indirect pointer to wide character string being converted.
len	Maximum number of bytes to write into the array pointed to by dst.
ps	Pointer to multi-byte conversion state.
loc	Locale used for conversion.

Return value

If conversion stops because a wide character is reached that does not correspond to a valid multi-byte character, an encoding error occurs: wcsrtombs() stores the value of the macro EILSEQ in errno and returns (size_t)(-1); the conversion state is unspecified. Otherwise, it returns the number of bytes in the resulting multi-byte character sequence, not including the terminating null character (if any).

Additional information

Converts a sequence of wide characters in the locale loc from the array indirectly pointed to by src into a sequence of corresponding multi-byte characters that begins in the conversion state described by the object pointed to by ps. If dst is not a null pointer, the converted characters are then stored into the array pointed to by dst. Conversion continues up to and including a terminating null wide character, which is also stored.

Conversion stops earlier in two cases: when a wide character is reached that does not correspond to a valid multi-byte character, or (if dst is not a null pointer) when the next multi-byte character would exceed the limit of len total bytes to be stored into the array pointed to by dst. Each conversion takes place as if by a call to wcrtomb_l().

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.20.4.12 wcsnrtombs()

Description

Convert wide string to multi-byte string, restartable (POSIX.1).

Prototype

```
size_t wcsnrtombs(      char       * dst,
                  const wchar_t   ** src,
                        size_t        nwc,
                        size_t        len,
                        mbstate_t  * ps);
```

Parameters

Parameter	Description
dst	Pointer to array that receives the multi-byte string.
src	Indirect pointer to wide character string being converted.
nwc	Maximum number of wide characters to read from src.
len	Maximum number of bytes to write into the array pointed to by dst.
ps	Pointer to multi-byte conversion state.

Return value

If conversion stops because a wide character is reached that does not correspond to a valid multi-byte character, an encoding error occurs: wcsrtombs() stores the value of the macro EILSEQ in errno and returns (size_t)(-1); the conversion state is unspecified. Otherwise, it returns the number of bytes in the resulting multi-byte character sequence, not including the terminating null character (if any).

Additional information

Converts a sequence of wide characters in the locale loc from the array indirectly pointed to by src into a sequence of corresponding multi-byte characters that begins in the conversion state described by the object pointed to by ps. If dst is not a null pointer, the converted characters are then stored into the array pointed to by dst. Conversion continues up to and including a terminating null wide character, which is also stored.

Conversion stops earlier in two cases: when a wide character is reached that does not correspond to a valid multi-byte character, or (if dst is not a null pointer) when the next multi-byte character would exceed the limit of len total bytes to be stored into the array pointed to by dst. Each conversion takes place as if by a call to wcrtomb_l().

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe [if configured].

4.20.4.13 wcsnrtombs_l()

Description

Convert wide string to multi-byte string, restartable (POSIX.1).

Prototype

```
size_t wcsnrtombs_l(      char       * dst,
                    const wchar_t   ** src,
                          size_t        nwc,
                          size_t        len,
                          mbstate_t  * ps,
                                        locale_t loc);
```

Parameters

Parameter	Description
dst	Pointer to array that receives the multi-byte string.
src	Indirect pointer to wide character string being converted.
nwc	Maximum number of wide characters to read from src.
len	Maximum number of bytes to write into the array pointed to by dst.
ps	Pointer to multi-byte conversion state.
loc	Locale used for conversion.

Return value

If conversion stops because a wide character is reached that does not correspond to a valid multi-byte character, an encoding error occurs: wcsrtombs() stores the value of the macro EILSEQ in errno and returns (size_t)(-1); the conversion state is unspecified. Otherwise, it returns the number of bytes in the resulting multi-byte character sequence, not including the terminating null character (if any).

Additional information

Converts a sequence of wide characters in the locale loc from the array indirectly pointed to by src into a sequence of corresponding multi-byte characters that begins in the conversion state described by the object pointed to by ps. If dst is not a null pointer, the converted characters are then stored into the array pointed to by dst. Conversion continues up to and including a terminating null wide character, which is also stored.

Conversion stops earlier in two cases: when a wide character is reached that does not correspond to a valid multi-byte character, or (if dst is not a null pointer) when the next multi-byte character would exceed the limit of len total bytes to be stored into the array pointed to by dst. Each conversion takes place as if by a call to wcrtomb_l().

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21 <wctype.h>

4.21.1 Classification functions

Function	Description
iswcntrl()	Is character a control?
iswcntrl_l()	Is character a control, per locale? (POSIX.1).
iswblank()	Is character a blank?
iswblank_l()	Is character a blank, per locale? (POSIX.1).
iswspace()	Is character a whitespace character?
iswspace_l()	Is character a whitespace character, per locale? (POSIX.1).
iswpunct()	Is character a punctuation mark?
iswpunct_l()	Is character a punctuation mark, per locale? (POSIX.1).
iswdigit()	Is character a decimal digit?
iswdigit_l()	Is character a decimal digit, per locale? (POSIX.
iswxdigit()	Is character a hexadecimal digit?
iswxdigit_l()	Is character a hexadecimal digit, per locale? (POSIX.1).
iswalpha()	Is character alphabetic?
iswalpha_l()	Is character alphabetic, per locale? (POSIX.1).
iswalnum()	Is character alphanumeric?
iswalnum_l()	Is character alphanumeric, per locale? (POSIX.1).
iswupper()	Is character an uppercase letter?
iswupper_l()	Is character an uppercase letter, per locale? (POSIX.1).
iswlower()	Is character a lowercase letter?
iswlower_l()	Is character a lowercase letter, per locale? (POSIX.1).
iswprint()	Is character printable?
iswprint_l()	Is character printable, per locale? (POSIX.1).
iswgraph()	Is character any printing character?
iswgraph_l()	Is character any printing character, per locale? (POSIX.1).
iswctype()	Construct character mapping.
iswctype_l()	Construct character mapping, per locale (POSIX.1).
wctype()	Construct character class.

4.21.1.1 iswcntrl()

Description

Is character a control?

Prototype

```
int iswcntrl(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is a control character in the current locale.

Thread safety

Safe [if configured].

4.21.1.2 iswcntrl_l()

Description

Is character a control, per locale? (POSIX.1).

Prototype

```
int iswcntrl_l(wint_t c,
                         locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a control character in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.3 iswblank()

Description

Is character a blank?

Prototype

```
int iswblank(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is either a space character or tab character in the current locale.

Thread safety

Safe [if configured].

4.21.1.4 iswblank_l()

Description

Is character a blank, per locale? (POSIX.1).

Prototype

```
int iswblank_l(wint_t c,
                     locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is either a space character or the tab character in locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.5 iswspace()

Description

Is character a whitespace character?

Prototype

```
int iswspace(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is a standard white-space character in the current locale. The standard white-space characters are space, form feed, new-line, carriage return, horizontal tab, and vertical tab.

Thread safety

Safe [if configured].

4.21.1.6 iswspace_l()

Description

Is character a whitespace character, per locale? (POSIX.1).

Prototype

```
int iswspace_l(wint_t c,
                     locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a standard white-space character in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.7 iswpunct()

Description

Is character a punctuation mark?

Prototype

```
int iswpunct(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) for every printing character for which neither `isspace()` nor `isalnum()` is true in the current locale.

Thread safety

Safe [if configured].

4.21.1.8 iswpunct_l()

Description

Is character a punctuation mark, per locale? (POSIX.1).

Prototype

```
int iswpunct_l(wint_t c,
                     locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) for every printing character for which neither `isspace()` nor `isalnum()` is true in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.9 iswdigit()

Description

Is character a decimal digit?

Prototype

```
int iswdigit(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is a digit in the current locale.

Thread safety

Safe [if configured].

4.21.1.10 iswdigit_l()

Description

Is character a decimal digit, per locale? (POSIX.1)

Prototype

```
int iswdigit_l(wint_t c,
                          locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a digit in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.11 iswxdigit()

Description

Is character a hexadecimal digit?

Prototype

```
int iswxdigit(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is a hexadecimal digit in the current locale.

Thread safety

Safe [if configured].

4.21.1.12 iswxdigit_l()

Description

Is character a hexadecimal digit, per locale? (POSIX.1).

Prototype

```
int iswxdigit_l(wint_t c,
                        locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a hexadecimal digit in the current locale.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.13 iswalpha()

Description

Is character alphabetic?

Prototype

```
int iswalpha(wint_t c);
```

Parameters

Parameter	Description
`c`	Wide character to test.

Return value

Returns true if the character `c` is alphabetic in the current locale. That is, any character for which `iswupper()` or `iswlower()` returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of `iswcntrl()`, `iswdigit()`, `iswpunct()`, or `isspace()` is true.

In the C locale, `isalpha()` returns nonzero (true) if and only if `isupper()` or `islower()` return true for value of the argument c.

Thread safety

Safe [if configured].

4.21.1.14 iswalpha_l()

Description

Is character alphabetic, per locale? (POSIX.1).

Prototype

```
int iswalpha_l(wint_t c,
                         locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns true if the wide character c is alphabetic in the locale loc. That is, any character for which iswupper() or iswlower() returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of iswcntrl_l(), iswdigit_l(), iswpunct_l(), or iswspace_l() is true in the locale loc.

In the C locale, iswalpha_l() returns nonzero (true) if and only if iswupper_l() or iswlower_l() return true for value of the argument c.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.15 iswalnum()

Description

Is character alphanumeric?

Prototype

```
int iswalnum(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is an alphabetic or numeric character in the current locale.

Thread safety

Safe [if configured].

4.21.1.16 iswalnum_l()

Description

Is character alphanumeric, per locale? (POSIX.1).

Prototype

```
int iswalnum_l(wint_t c,
                     locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is an alphabetic or numeric character in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.17 iswupper()

Description

Is character an uppercase letter?

Prototype

```
int iswupper(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is an uppercase letter in the current locale.

Thread safety

Safe [if configured].

4.21.1.18 iswupper_l()

Description

Is character an uppercase letter, per locale? (POSIX.1).

Prototype

```
int iswupper_l(wint_t c,
                         locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is an uppercase letter in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.19 iswlower()

Description

Is character a lowercase letter?

Prototype

```
int iswlower(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is a lowercase letter in the current locale.

Thread safety

Safe [if configured].

4.21.1.20 iswlower_l()

Description

Is character a lowercase letter, per locale? (POSIX.1).

Prototype

```
int iswlower_l(wint_t c,
                     locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is a lowercase letter in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.21 iswprint()

Description

Is character printable?

Prototype

```
int iswprint(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is any printing character including space in the current locale.

Thread safety

Safe [if configured].

4.21.1.22 iswprint_l()

Description

Is character printable, per locale? (POSIX.1).

Prototype

```
int iswprint_l(wint_t c,
                     locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is any printing character including space in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.23 iswgraph()

Description

Is character any printing character?

Prototype

```
int iswgraph(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to test.

Return value

Returns nonzero (true) if and only if the value of the argument c is any printing character except space in the current locale.

Thread safety

Safe [if configured].

4.21.1.24 iswgraph_l()

Description

Is character any printing character, per locale? (POSIX.1).

Prototype

```
int iswgraph_l(wint_t c,
               locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
loc	Locale used to test c.

Return value

Returns nonzero (true) if and only if the value of the argument c is any printing character except space in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.25 iswctype()

Description

Construct character mapping.

Prototype

```
int iswctype(wint_t   c,
             wctype_t t);
```

Parameters

Parameter	Description
c	Wide character to test.
t	Property to test, typically delivered by calling `wctype()`.

Return value

Returns nonzero (true) if and only if the wide character `c` has the property `t` in the current locale.

Additional information

Determines whether the wide character `c` has the property described by `t` in the current locale.

Thread safety

Safe [if configured].

4.21.1.26 iswctype_l()

Description

Construct character mapping, per locale (POSIX.1).

Prototype

```
int iswctype_l(wint_t   c,
               wctype_t t,
                        locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to test.
t	Property to test, typically delivered by calling `wctype_l()`.
loc	Locale used for mapping.

Return value

Returns nonzero (true) if and only if the wide character c has the property t in the locale loc.

Additional information

Determines whether the wide character c has the property described by t in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.1.27 wctype()

Description

Construct character class.

Prototype

```
wctype_t wctype(char const * name);
```

Parameters

Parameter	Description
name	Name of mapping.

Return value

Character class; zero if class unrecognized.

Additional information

Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string argument property.

If property identifies a valid class of wide characters in the current locale, returns a nonzero value that is valid as the second argument to `iswctype()`; otherwise, it returns zero.

Notes

The only mappings supported are:

- “alnum”
- “alpha”,
- “blank”
- “cntrl”
- “digit”
- “graph”
- “lower”
- “print”
- “punct”
- “space”
- “upper”
- “xdigit”

Thread safety

Safe [if configured].

4.21.2 Conversion functions

Function	Description
towupper()	Convert uppercase character to lowercase.
towupper_l()	Convert uppercase character to lowercase, per locale (POSIX.1).
towlower()	Convert uppercase character to lowercase.
towlower_l()	Convert uppercase character to lowercase, per locale (POSIX.1).
towctrans()	Translate character.
towctrans_l()	Translate character, per locale (POSIX.1).
wctrans()	Construct character mapping.
wctrans_l()	Construct character mapping, per locale (POSIX.1).

4.21.2.1 towupper()

Description

Convert uppercase character to lowercase.

Prototype

```
wint_t towupper(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to convert.

Return value

Converted wide character.

Additional information

Converts a lowercase letter to a corresponding uppercase letter.

If the argument c is a wide character for which `iswlower()` is true and there are one or more corresponding wide characters, in the current current locale, for which `iswupper()` is true, `towupper()` returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, c is returned unchanged.

Thread safety

Safe [if configured].

4.21.2.2 towupper_l()

Description

Convert uppercase character to lowercase, per locale (POSIX.1).

Prototype

```
wint_t towupper_l(wint_t c,
                         locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to convert.
loc	Locale used to convert c.

Return value

Converted wide character.

Additional information

Converts a lowercase letter to a corresponding uppercase letter.

If the argument c is a wide character for which iswlower_l() is true and there are one or more corresponding wide characters, in the current locale loc, for which iswupper_l() is true, towupper_l() returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, c is returned unchanged.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.2.3 towlower()

Description

Convert uppercase character to lowercase.

Prototype

```
wint_t towlower(wint_t c);
```

Parameters

Parameter	Description
c	Wide character to convert.

Return value

Converted wide character.

Additional information

Converts an uppercase letter to a corresponding lowercase letter.

If the argument c is a wide character for which iswupper() is true and there are one or more corresponding wide characters, in the current locale, for which iswlower() is true, towlower() returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, c is returned unchanged.

Thread safety

Safe [if configured].

4.21.2.4 towlower_l()

Description

Convert uppercase character to lowercase, per locale (POSIX.1).

Prototype

```
wint_t towlower_l(wint_t c,
                  locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to convert.
loc	Locale used to convert c.

Return value

Converted wide character.

Additional information

Converts an uppercase letter to a corresponding lowercase letter.

If the argument c is a wide character for which iswupper_l() is true and there are one or more corresponding wide characters, in the locale loc, for which iswlower_l() is true, towlower_l() returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, c is returned unchanged.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.2.5 towctrans()

Description

Translate character.

Prototype

```
wint_t towctrans(wint_t    c,
                 wctrans_t t);
```

Parameters

Parameter	Description
c	Wide character to convert.
t	Mapping to use for conversion.

Return value

Converted wide character.

Additional information

Maps the wide character c using the mapping described by t in the current locale.

Thread safety

Safe [if configured].

4.21.2.6 towctrans_l()

Description

Translate character, per locale (POSIX.1).

Prototype

```
wint_t towctrans_l(wint_t    c,
                   wctrans_t t,
                             locale_t loc);
```

Parameters

Parameter	Description
c	Wide character to convert.
t	Mapping to use for conversion.
loc	Locale used for conversion.

Return value

Converted wide character.

Additional information

Maps the wide character c using the mapping described by t in the locale loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.21.2.7 wctrans()

Description

Construct character mapping.

Prototype

```
wctrans_t wctrans(const char * name);
```

Parameters

Parameter	Description
`name`	Name of mapping.

Return value

Transformation mapping; zero if mapping unrecognized.

Additional information

Constructs a value of type `wctrans_t` that describes a mapping between wide characters identified by the string argument property.

If property identifies a valid mapping of wide characters in the current locale, `wctrans_l()` returns a nonzero value that is valid as the second argument to `towctrans()`; otherwise, it returns zero.

The only mappings supported are “tolower” and “toupper”.

Thread safety

Safe.

4.21.2.8 wctrans_l()

Description

Construct character mapping, per locale (POSIX.1).

Prototype

```
wctrans_t wctrans_l(const char * name,
                                locale_t loc);
```

Parameters

Parameter	Description
name	Name of mapping.
loc	Locale used for mapping.

Return value

Transformation mapping; zero if mapping unrecognized.

Additional information

Constructs a value of type `wctrans_t` that describes a mapping between wide characters identified by the string argument property.

If property identifies a valid mapping of wide characters in the locale `loc`, `wctrans_l()` returns a nonzero value that is valid as the second argument to `towctrans()`; otherwise, it returns zero.

The only mappings supported are “tolower” and “toupper”.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

4.22 <xlocale.h>

4.22.1 Locale management

Function	Description
`newlocale()`	Create or modify locale (POSIX.1).
`duplocale()`	Duplicate locale data (POSIX.1).
`uselocale()`	Set current locale (POSIX.1).
`freelocale()`	Free locale (POSIX.1).
`localeconv_l()`	Get locale data (POSIX.1).

4.22.1.1 newlocale()

Description

Create or modify locale (POSIX.1).

Prototype

```
locale_t newlocale(      int        category_mask,
                   const char     * loc,
                         locale_t   base);
```

Parameters

Parameter	Description
category_mask	Locale categories to be set or modified.
loc	Locale name.
base	Base to modify or NULL to create a new locale.

Return value

Pointer to modified locale.

Additional information

Creates a new locale object or modifies an existing one. If the base argument is NULL, a new locale object is created.

category_mask specifies the locale categories to be set or modified. Values for category_mask are constructed by a bitwise-inclusive OR of the symbolic constants LC_CTYPE_MASK, LC_NUMERIC_MASK, LC_TIME_MASK, LC_COLLATE_MASK, LC_MONETARY_MASK, and LC_MESSAGES_MASK.

For each category with the corresponding bit set in category_mask, the data from the locale named by loc is used. In the case of modifying an existing locale object, the data from the locale named by loc replaces the existing data within the locale object. If a completely new locale object is created, the data for all sections not requested by category_mask are taken from the default locale.

The locales "C" and "POSIX" are equivalent and defined for all settings of category_mask:

If loc is NULL, then the "C" locale is used. If loc is an empty string, newlocale() will use the default locale.

If base is NULL, the current locale is used. If base is LC_GLOBAL_LOCALE, the global locale is used.

If mask is LC_ALL_MASK, base is ignored.

Notes

Conforms to POSIX.1-2017.

POSIX.1-2017 does not specify whether the locale object pointed to by base is modified or whether it is freed and a new locale object created.

The category mask LC_MESSAGES_MASK is not implemented as POSIX messages are not implemented.

Thread safety

Safe [if configured].

4.22.1.2 duplocale()

Description

Duplicate locale data (POSIX.1).

Prototype

```
locale_t duplocale(locale_t base);
```

Parameters

Parameter	Description
`base`	Locale to duplicate.

Return value

If there is insufficient memory to duplicate loc, returns a `NULL` and sets errno to ENOMEM as required by POSIX.1-2017. Otherwise, returns a new, duplicated locale.

Additional information

Duplicates the locale object referenced by loc. Duplicated locales must be freed with `freelocale()`.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe [if configured].

4.22.1.3 uselocale()

Description

Set current locale (POSIX.1).

Prototype

```
locale_t uselocale(locale_t loc);
```

Parameters

Parameter	Description
loc	Locale to use.

Return value

The locale set using the previous call to `uselocale()`, or `LC_GLOBAL_LOCALE` if none was set.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe [if configured].

4.22.1.4 freelocale()

Description

Free locale (POSIX.1).

Prototype

```
int freelocale(locale_t loc);
```

Parameters

Parameter	Description
loc	Locale to free.

Return value

Zero on success, -1 on error.

Additional information

Frees the storage associated with loc.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe [if configured].

4.22.1.5 localeconv_l()

Description

Get locale data (POSIX.1).

Prototype

```
localeconv_l(locale_t loc);
```

Parameters

Parameter	Description
loc	Locale to inquire.

Return value

Returns a pointer to a structure of type lconv with the corresponding values for the locale loc filled in.

Notes

Conforms to POSIX.1-2017.

Thread safety

Safe.

Chapter 5

Compiler support API

5.1 Environment support

This section summarises the functions proved by emRun to support the C library and environment.

Function	Description
`__SEGGER_RTL_execute_at_exit_fns`	Execute at-exit functions.
`__SEGGER_RTL_set_locale_name_buffer`	Set buffer used to store locale names.

5.1.1 __SEGGER_RTL_execute_at_exit_fns()

Description

Execute at-exit functions.

Prototype

```
void __SEGGER_RTL_execute_at_exit_fns(void);
```

Additional information

Executes all functions registered by calls to `atexit()` in reverse order of their registration. It does this in the caller's execution context.

Thread safety

Not applicable.

5.1.2 __SEGGER_RTL_set_locale_name_buffer()

Description

Set buffer used to store locale names.

Prototype

```
void __SEGGER_RTL_set_locale_name_buffer(char * buf);
```

Parameters

Parameter	Description
buf	Pointer to object that stores name buffer.

Additional information

The function `setlocale()` returns a pointer to an object that contains the previously-set locale. As such, that object must be thread-local, but its size is not known at compile time. Additionally, to be thread-safe, this buffer must be local to the thread and not shared between separate execution contexts.

The user can set the location for the name buffer used by `setlocale()` using this function. The name buffer must be large enough to contain six locale names separated by semicolons and terminated by a final null. Assuming a locale name "`hu_HU.iso_8859_2`" of 16 characters, a buffer of at least 102 characters is required for correct operation.

Note that if no name buffer is set using this function, `setlocale()` may still be used but will return `NULL` as the result. This enables use of `setlocale()` but does incur an overhead for a thread-local or global buffer which may never be required.

Thread safety

Safe [if configured].

5.2 Arm AEABI library API

The emRun provides an implementation of the Arm AEABI functions.

The assembly language floating-point funnctions are contained in separate files:

- For Arm this is found in `floatasmops_arm.s`.

The interface to the AEABI functions differs from the standard calling convention when the hard-floating ABI is used: all floatting-point AEABI functions receive their parameters in integer registers and return their results in integer regsisters.

5.2.1 Floating arithmetic

Function	Description
`__aeabi_fadd`	Add, float.
`__aeabi_dadd`	Add, double.
`__aeabi_fsub`	Subtract, float.
`__aeabi_dsub`	Subtract, double.
`__aeabi_frsub`	Reverse subtract, float.
`__aeabi_drsub`	Reverse subtract, double.
`__aeabi_fmul`	Multiply, float.
`__aeabi_dmul`	Multiply, double.
`__aeabi_fdiv`	Divide, float.
`__aeabi_ddiv`	Divide, double.

5.2.1.1 __aeabi_fadd()

Description

Add, float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_fadd(__SEGGER_RTL_U32 x,
                              __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Augend.
y	Addend.

Return value

Sum.

Thread safety

Safe.

5.2.1.2 __aeabi_dadd()

Description

Add, double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_dadd(__SEGGER_RTL_U64 x,
                              __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Augend.
y	Addend.

Return value

Sum.

Thread safety

Safe.

5.2.1.3 __aeabi_fsub()

Description

Subtract, float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_fsub(__SEGGER_RTL_U32 x,
                              __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Minuend.
y	Subtrahend.

Return value

Difference.

Thread safety

Safe.

5.2.1.4 __aeabi_dsub()

Description

Subtract, double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_dsub(__SEGGER_RTL_U64 x,
                              __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Minuend.
y	Subtrahend.

Return value

Difference.

Thread safety

Safe.

5.2.1.5 __aeabi_frsub()

Description

Reverse subtract, float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_frsub(__SEGGER_RTL_U32 x,
                               __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Minuend.
y	Subtrahend.

Return value

Difference.

Thread safety

Safe.

5.2.1.6 __aeabi_drsub()

Description

Reverse subtract, double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_drsub(__SEGGER_RTL_U64 x,
                               __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Minuend.
y	Subtrahend.

Return value

Difference.

Thread safety

Safe.

5.2.1.7 __aeabi_fmul()

Description

Multiply, float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_fmul(__SEGGER_RTL_U32 x,
                              __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Multiplicand.
y	Multiplier.

Return value

Product.

Thread safety

Safe.

5.2.1.8 __aeabi_dmul()

Description

Multiply, double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_dmul(__SEGGER_RTL_U64 x,
                              __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Multiplicand.
y	Multiplier.

Return value

Product.

Thread safety

Safe.

5.2.1.9 __aeabi_fdiv()

Description

Divide, float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_fdiv(__SEGGER_RTL_U32 x,
                              __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Dividend.
y	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.2.1.10 __aeabi_ddiv()

Description

Divide, double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_ddiv(__SEGGER_RTL_U64 x,
                              __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Dividend.
y	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.2.2 Floating conversions

Function	Description
`__aeabi_f2iz`	Convert float to int.
`__aeabi_d2iz`	Convert double to int.
`__aeabi_f2uiz`	Convert float to unsigned int.
`__aeabi_d2uiz`	Convert double to unsigned.
`__aeabi_f2lz`	Convert float to long long.
`__aeabi_d2lz`	Convert double to long long.
`__aeabi_f2ulz`	Convert float to unsigned long long.
`__aeabi_d2ulz`	Convert double to unsigned long long.
`__aeabi_i2f`	Convert int to float.
`__aeabi_i2d`	Convert int to double.
`__aeabi_ui2f`	Convert unsigned to float.
`__aeabi_ui2d`	Convert unsigned to double.
`__aeabi_l2f`	Convert long long to float.
`__aeabi_l2d`	Convert long long to double.
`__aeabi_ul2f`	Convert unsigned long long to float.
`__aeabi_ul2d`	Convert unsigned long long to double.
`__aeabi_f2d`	Extend float to double.
`__aeabi_d2f`	Truncate double to float.
`__aeabi_f2h`	Truncate float to IEEE half-precision float.
`__aeabi_d2h`	Truncate double to IEEE half-precision float.
`__aeabi_h2f`	Convert IEEE half-precision float to float.
`__aeabi_h2d`	Convert IEEE half-precision float to double.

5.2.2.1 __aeabi_f2iz()

Description

Convert float to int.

Prototype

```
__SEGGER_RTL_I32 __aeabi_f2iz(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.2.2.2 __aeabi_d2iz()

Description

Convert double to int.

Prototype

```
__SEGGER_RTL_I32 __aeabi_d2iz(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.2.2.3 __aeabi_f2uiz()

Description

Convert float to unsigned int.

Prototype

```
__SEGGER_RTL_U32 __aeabi_f2uiz(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.2.2.4 __aeabi_d2uiz()

Description

Convert double to unsigned.

Prototype

```
__SEGGER_RTL_U32 __aeabi_d2uiz(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Double value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.2.2.5 __aeabi_f2lz()

Description

Convert float to long long.

Prototype

```
__SEGGER_RTL_I64 __aeabi_f2lz(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Notes

The RV32 compiler converts a `__SEGGER_RTL_U32` to a 64-bit integer by calling runtime support to handle it.

Thread safety

Safe.

5.2.2.6 __aeabi_d2lz()

Description

Convert double to long long.

Prototype

```
__SEGGER_RTL_I64 __aeabi_d2lz(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Notes

RV32 always calls runtime for `__SEGGER_RTL_U64` to int64 conversion.

Thread safety

Safe.

5.2.2.7 __aeabi_f2ulz()

Description

Convert float to unsigned long long.

Prototype

```
__SEGGER_RTL_U64 __aeabi_f2ulz(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.2.2.8 __aeabi_d2ulz()

Description

Convert double to unsigned long long.

Prototype

```
__SEGGER_RTL_U64 __aeabi_d2ulz(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.2.2.9 __aeabi_i2f()

Description

Convert int to float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_i2f(__SEGGER_RTL_I32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.2.2.10 __aeabi_i2d()

Description

Convert int to double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_i2d(__SEGGER_RTL_I32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.2.2.11 __aeabi_ui2f()

Description

Convert unsigned to float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_ui2f(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.2.2.12 __aeabi_ui2d()

Description

Convert unsigned to double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_ui2d(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Unsigned value to convert.

Return value

`__SEGGER_RTL_U64` value.

Thread safety

Safe.

5.2.2.13 __aeabi_l2f()

Description

Convert long long to float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_l2f(__SEGGER_RTL_I64 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.2.2.14 __aeabi_l2d()

Description

Convert long long to double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_l2d(__SEGGER_RTL_I64 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.2.2.15 __aeabi_ul2f()

Description

Convert unsigned long long to float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_ul2f(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Unsigned long long value to convert.

Return value

`__SEGGER_RTL_U32` value.

Thread safety

Safe.

5.2.2.16 __aeabi_ul2d()

Description

Convert unsigned long long to double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_ul2d(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Unsigned long long value to convert.

Return value

__SEGGER_RTL_U64 value.

Thread safety

Safe.

5.2.2.17 __aeabi_f2d()

Description

Extend float to double.

Prototype

```
__SEGGER_RTL_U64 __aeabi_f2d(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Floating value to extend.

Return value

`__SEGGER_RTL_U64` value.

Thread safety

Safe.

5.2.2.18 __aeabi_d2f()

Description

Truncate double to float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_d2f(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Double value to truncate.

Return value

Float value.

Thread safety

Safe.

5.2.2.19 __aeabi_f2h()

Description

Truncate float to IEEE half-precision float.

Prototype

```
__SEGGER_RTL_U16 __aeabi_f2h(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Float value to truncate.

Return value

Float value.

Thread safety

Safe.

5.2.2.20 __aeabi_d2h()

Description

Truncate double to IEEE half-precision float.

Prototype

```
__SEGGER_RTL_U16 __aeabi_d2h(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Double value to truncate.

Return value

Half-precision value.

Thread safety

Safe.

5.2.2.21 __aeabi_h2f()

Description

Convert IEEE half-precision float to float.

Prototype

```
__SEGGER_RTL_U32 __aeabi_h2f(__SEGGER_RTL_U16 x);
```

Parameters

Parameter	Description
x	Half-precision float.

Return value

Single-precision float.

Thread safety

Safe.

5.2.2.22 __aeabi_f2h()

Description

Truncate float to IEEE half-precision float.

Prototype

```
__SEGGER_RTL_U16 __aeabi_f2h(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Float value to truncate.

Return value

Float value.

Thread safety

Safe.

5.2.3 Floating comparisons

Function	Description
`__aeabi_fcmpeq`	Equal, float.
`__aeabi_dcmpeq`	Equal, double.
`__aeabi_fcmplt`	Less than, float.
`__aeabi_dcmplt`	Less than, double.
`__aeabi_fcmple`	Less than or equal, float.
`__aeabi_dcmple`	Less than, double.
`__aeabi_fcmpgt`	Less than, float.
`__aeabi_dcmpgt`	Less than, double.
`__aeabi_fcmpge`	Less than, float.
`__aeabi_dcmpge`	Less than, double.

5.2.3.1 __aeabi_fcmpeq()

Description

Equal, float.

Prototype

```
int __aeabi_fcmpeq(__SEGGER_RTL_U32 x,
                   __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not equal to y.
1	x is equal to y.

Thread safety

Safe.

5.2.3.2 __aeabi_dcmpeq()

Description

Equal, double.

Prototype

```
int __aeabi_dcmpeq(__SEGGER_RTL_U64 x,
                   __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not equal to y.
1	x is equal to y.

Thread safety

Safe.

5.2.3.3 __aeabi_fcmplt()

Description

Less than, float.

Prototype

```
int __aeabi_fcmplt(__SEGGER_RTL_U32 x,
                   __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not less than y.
1	x is less than y.

Thread safety

Safe.

5.2.3.4 __aeabi_dcmplt()

Description

Less than, double.

Prototype

```
int __aeabi_dcmplt(__SEGGER_RTL_U64 x,
                   __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not less than y.
1	x is less than y.

Thread safety

Safe.

5.2.3.5 __aeabi_fcmple()

Description

Less than or equal, float.

Prototype

```
int __aeabi_fcmple(__SEGGER_RTL_U32 x,
                   __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not less than or equal to y.
1	x is less than or equal to y.

Thread safety

Safe.

5.2.3.6 __aeabi_dcmple()

Description

Less than, double.

Prototype

```
int __aeabi_dcmple(__SEGGER_RTL_U64 x,
                   __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not less than or equal to y.
1	x is less than or equal to y.

Thread safety

Safe.

5.2.3.7 __aeabi_fcmpgt()

Description

Less than, float.

Prototype

```
int __aeabi_fcmpgt(__SEGGER_RTL_U32 x,
                   __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not greater than y.
1	x is greater than y.

Thread safety

Safe.

5.2.3.8 __aeabi_dcmpgt()

Description

Less than, double.

Prototype

```
int __aeabi_dcmpgt(__SEGGER_RTL_U64 x,
                   __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not greater than y.
1	x is greater than y.

Thread safety

Safe.

5.2.3.9 __aeabi_fcmpge()

Description

Less than, float.

Prototype

```
int __aeabi_fcmpge(__SEGGER_RTL_U32 x,
                   __SEGGER_RTL_U32 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not greater than or equal to y.
1	x is greater than or equal to y.

Thread safety

Safe.

5.2.3.10 __aeabi_dcmpge()

Description

Less than, double.

Prototype

```
int __aeabi_dcmpge(__SEGGER_RTL_U64 x,
                   __SEGGER_RTL_U64 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

0	x is not greater than or equal to y.
1	x is greater than or equal to y.

Thread safety

Safe.

5.3 GNU library API

5.3.1 Integer arithmetic

Function	Description
`__mulsi3`	Multiply, signed 32-bit integer.
`__muldi3`	Multiply, signed 64-bit integer.
`__multi3`	Multiply, signed 128-bit integer.
`__divsi3`	Divide, signed 32-bit integer.
`__divdi3`	Divide, signed 64-bit integer.
`__divti3`	Divide, signed 128-bit integer.
`__udivsi3`	Divide, unsigned 32-bit integer.
`__udivdi3`	Divide, unsigned 64-bit integer.
`__udivti3`	Divide, unsigned 128-bit integer.
`__modsi3`	Remainder after divide, signed 32-bit integer.
`__moddi3`	Remainder after divide, signed 64-bit integer.
`__modti3`	Remainder after divide, signed 128-bit integer.
`__umodsi3`	Remainder after divide, unsigned 32-bit integer.
`__umoddi3`	Remainder after divide, unsigned 64-bit integer.
`__umodti3`	Remainder after divide, unsigned 128-bit integer.
`__udivmodsi4`	Divide with remainder, unsigned 32-bit integer.
`__udivmoddi4`	Divide with remainder, unsigned 64-bit integer.
`__clzsi2`	Count leading zeros, 32-bit integer.
`__clzdi2`	Count leading zeros, 64-bit integer.
`__popcountsi2`	Population count, 32-bit integer.
`__popcountdi2`	Population count, 64-bit integer.
`__paritysi2`	Parity, 32-bit integer.
`__paritydi2`	Parity, 64-bit integer.

5.3.1.1 __mulsi3()

Description

Multiply, signed 32-bit integer.

Prototype

```
__SEGGER_RTL_U32 __mulsi3(__SEGGER_RTL_U32 a,
                          __SEGGER_RTL_U32 b);
```

Parameters

Parameter	Description
a	Multiplier.
b	Multiplicand.

Return value

Product.

Thread safety

Safe.

5.3.1.2 __muldi3()

Description

Multiply, signed 64-bit integer.

Prototype

```
__SEGGER_RTL_U64 __muldi3(__SEGGER_RTL_U64 a,
                          __SEGGER_RTL_U64 b);
```

Parameters

Parameter	Description
a	Multiplier.
b	Multiplicand.

Return value

Product.

Thread safety

Safe.

5.3.1.3 __multi3()

Description

Multiply, signed 128-bit integer.

Prototype

```
__SEGGER_RTL_U128 __multi3(__SEGGER_RTL_U128 a,
                           __SEGGER_RTL_U128 b);
```

Parameters

Parameter	Description
a	Multiplier.
b	Multiplicand.

Return value

Product.

Thread safety

Safe.

5.3.1.4 __divsi3()

Description

Divide, signed 32-bit integer.

Prototype

```
int32_t __divsi3(int32_t u,
                 int32_t v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.1.5 __divdi3()

Description

Divide, signed 64-bit integer.

Prototype

```
int64_t __divdi3(int64_t u,
                 int64_t v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.1.6 __divti3()

Description

Divide, signed 128-bit integer.

Prototype

```
__SEGGER_RTL_U128 __divti3(__SEGGER_RTL_U128 u,
                           __SEGGER_RTL_U128 v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.1.7 __udivsi3()

Description

Divide, unsigned 32-bit integer.

Prototype

```
__SEGGER_RTL_U32 __udivsi3(__SEGGER_RTL_U32 u,
                           __SEGGER_RTL_U32 v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.1.8 __udivdi3()

Description

Divide, unsigned 64-bit integer.

Prototype

```
__SEGGER_RTL_U64 __udivdi3(__SEGGER_RTL_U64 u,
                           __SEGGER_RTL_U64 v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

floor(u / v).

Thread safety

Safe.

5.3.1.9 __udivti3()

Description

Divide, unsigned 128-bit integer.

Prototype

```
__SEGGER_RTL_U128 __udivti3(__SEGGER_RTL_U128 u,
                            __SEGGER_RTL_U128 v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

floor(u / v).

Thread safety

Safe.

5.3.1.10 __modsi3()

Description

Remainder after divide, signed 32-bit integer.

Prototype

```
int32_t __modsi3(int32_t u,
                 int32_t v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Remainder.

Thread safety

Safe.

5.3.1.11 __moddi3()

Description

Remainder after divide, signed 64-bit integer.

Prototype

```
int64_t __moddi3(int64_t u,
                 int64_t v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Remainder.

Thread safety

Safe.

5.3.1.12 __modti3()

Description

Remainder after divide, signed 128-bit integer.

Prototype

```
__SEGGER_RTL_U128 __modti3(__SEGGER_RTL_U128 u,
                           __SEGGER_RTL_U128 v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Remainder.

Thread safety

Safe.

5.3.1.13 __umodsi3()

Description

Remainder after divide, unsigned 32-bit integer.

Prototype

```
__SEGGER_RTL_U32 __umodsi3(__SEGGER_RTL_U32 u,
                           __SEGGER_RTL_U32 v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Remainder.

Thread safety

Safe.

5.3.1.14 __umoddi3()

Description

Remainder after divide, unsigned 64-bit integer.

Prototype

```
__SEGGER_RTL_U64 __umoddi3(__SEGGER_RTL_U64 u,
                           __SEGGER_RTL_U64 v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Remainder.

Thread safety

Safe.

5.3.1.15 __umodti3()

Description

Remainder after divide, unsigned 128-bit integer.

Prototype

```
__SEGGER_RTL_U128 __umodti3(__SEGGER_RTL_U128 u,
                            __SEGGER_RTL_U128 v);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.

Return value

Remainder.

Thread safety

Safe.

5.3.1.16 __udivmodsi4()

Description

Divide with remainder, unsigned 32-bit integer.

Prototype

```
__SEGGER_RTL_U32 __udivmodsi4(__SEGGER_RTL_U32 u,
                              __SEGGER_RTL_U32 v,
                              __SEGGER_RTL_U32 *rem);
```

Parameters

Parameter	Description
u	Divivdend.
v	Divisor.
rem	Pointer to object that receives the remainder.

Return value

Quotient.

Thread safety

Safe.

5.3.1.17 __udivmoddi4()

Description

Divide with remainder, unsigned 64-bit integer.

Prototype

```
__SEGGER_RTL_U64 __udivmoddi4(__SEGGER_RTL_U64 u,
                              __SEGGER_RTL_U64 v,
                              __SEGGER_RTL_U64 *rem);
```

Parameters

Parameter	Description
u	Dividend.
v	Divisor.
rem	Pointer to object that receives the remainder.

Return value

Quotient.

Thread safety

Safe.

5.3.1.18 __clzsi2()

Description

Count leading zeros, 32-bit integer.

Prototype

```
int __clzsi2(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Argument; x must not be zero.

Return value

Number of leading zeros in x.

Thread safety

Safe.

5.3.1.19 __clzdi2()

Description

Count leading zeros, 64-bit integer.

Prototype

```
int __clzdi2(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Argument; x must not be zero.

Return value

Number of leading zeros in x.

Thread safety

Safe.

5.3.1.20 __popcountsi2()

Description

Population count, 32-bit integer.

Prototype

```
int __popcountsi2(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Argument.

Return value

Count of number of one bits in x.

Thread safety

Safe.

5.3.1.21 __popcountdi2()

Description

Population count, 64-bit integer.

Prototype

```
int __popcountdi2(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Argument.

Return value

Count of number of one bits in x.

Thread safety

Safe.

5.3.1.22 __paritysi2()

Description

Parity, 32-bit integer.

Prototype

```
int __paritysi2(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Argument.

Return value

1	number of one bits in x is odd.
0	number of one bits in x is even.

Thread safety

Safe.

5.3.1.23 __paritydi2()

Description

Parity, 64-bit integer.

Prototype

```
int __paritydi2(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Argument.

Return value

1	number of one bits in x is odd.
0	number of one bits in x is even.

Thread safety

Safe.

5.3.2 Floating arithmetic

Function	Description
__addsf3	Add, float.
__adddf3	Add, double.
__addtf3	Add, long double.
__subsf3	Subtract, float.
__subdf3	Subtract, double.
__subtf3	Subtract, long double.
__mulsf3	Multiply, float.
__muldf3	Multiply, double.
__multf3	Multiply, long double.
__divsf3	Divide, float.
__divdf3	Divide, double.
__divtf3	Divide, long double.

5.3.2.1 __addsf3()

Description

Add, float.

Prototype

```
float __addsf3(float x,
               float y);
```

Parameters

Parameter	Description
x	Augend.
y	Addend.

Return value

Sum.

Thread safety

Safe.

5.3.2.2 __adddf3()

Description

Add, double.

Prototype

```
double __adddf3(double x,
                double y);
```

Parameters

Parameter	Description
x	Augend.
y	Addend.

Return value

Sum.

Thread safety

Safe.

5.3.2.3 __addtf3()

Description

Add, long double.

Prototype

```
long double __addtf3(long double x,
                     long double y);
```

Parameters

Parameter	Description
x	Augend.
y	Addend.

Return value

Sum.

Thread safety

Safe.

5.3.2.4 __subsf3()

Description

Subtract, float.

Prototype

```
float __subsf3(float x,
               float y);
```

Parameters

Parameter	Description
x	Minuend.
y	Subtrahend.

Return value

Difference.

Thread safety

Safe.

5.3.2.5 __subdf3()

Description

Subtract, double.

Prototype

```
double __subdf3(double x,
                double y);
```

Parameters

Parameter	Description
x	Minuend.
y	Subtrahend.

Return value

Difference.

Thread safety

Safe.

5.3.2.6 __subtf3()

Description

Subtract, long double.

Prototype

```
long double __subtf3(long double x,
                     long double y);
```

Parameters

Parameter	Description
x	Minuend.
y	Subtrahend.

Return value

Difference.

Thread safety

Safe.

5.3.2.7 __mulsf3()

Description

Multiply, float.

Prototype

```
float __mulsf3(float x,
               float y);
```

Parameters

Parameter	Description
x	Multiplicand.
y	Multiplier.

Return value

Product.

Thread safety

Safe.

5.3.2.8 __muldf3()

Description

Multiply, double.

Prototype

```
double __muldf3(double x,
                double y);
```

Parameters

Parameter	Description
x	Multiplicand.
y	Multiplier.

Return value

Product.

Thread safety

Safe.

5.3.2.9 __multf3()

Description

Multiply, long double.

Prototype

```
long double __multf3(long double x,
                     long double y);
```

Parameters

Parameter	Description
x	Multiplicand.
y	Multiplier.

Return value

Product.

Thread safety

Safe.

5.3.2.10 __divsf3()

Description

Divide, float.

Prototype

```
float __divsf3(float x,
               float y);
```

Parameters

Parameter	Description
x	Dividend.
y	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.2.11 __divdf3()

Description

Divide, double.

Prototype

```
double __divdf3(double x,
                double y);
```

Parameters

Parameter	Description
x	Dividend.
y	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.2.12 __divtf3()

Description

Divide, long double.

Prototype

```
long double __divtf3(long double x,
                     long double y);
```

Parameters

Parameter	Description
x	Dividend.
y	Divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.3 Floating conversions

Function	Description
__fixhfsi	Convert half-precision float to int.
__fixsfsi	Convert float to int.
__fixdfsi	Convert double to int.
__fixtfsi	Convert long double to int.
__fixhfdi	Convert half-precision float to int.
__fixsfdi	Convert float to long long.
__fixdfdi	Convert double to long long.
__fixtfdi	Convert long double to long long.
__fixunshfsi	Convert half-precision float to unsigned.
__fixunssfsi	Convert float to unsigned.
__fixunsdfsi	Convert double to unsigned.
__fixunstfsi	Convert long double to int.
__fixunshfdi	Convert half-precision float to unsigned.
__fixunssfdi	Convert float to unsigned long long.
__fixunsdfdi	Convert double to unsigned long long.
__fixunstfdi	Convert long double to unsigned long long.
__floatsihf	Convert int to half-precision float.
__floatsisf	Convert int to float.
__floatsidf	Convert int to double.
__floatsitf	Convert int to long double.
__floatdihf	Convert long long to half-precision float.
__floatdisf	Convert long long to float.
__floatdidf	Convert long long to double.
__floatditf	Convert long long to long double.
__floatunsihf	Convert unsigned to half-precision float.
__floatunsisf	Convert unsigned to float.
__floatunsidf	Convert unsigned to double.
__floatunsitf	Convert unsigned to long double.
__floatundihf	Convert unsigned long long to half-precision float.
__floatundisf	Convert unsigned long long to float.
__floatundidf	Convert unsigned long long to double.
__floatunditf	Convert unsigned long long to long double.
__extendhfsf2	Convert IEEE half-precision float to float.
__extendhfdf2	Convert IEEE half-precision float to double.
__extendhftf2	Convert IEEE half-precision float to long double.
__extendsfdf2	Extend float to double.
__extendsftf2	Extend float to long double.
__extenddftf2	Extend double to long double.
__trunctfdf2	Truncate long double to double.
__trunctfsf2	Truncate long double to float.
__trunctfhf2	Truncate long double to IEEE half-precision float.
__truncdfsf2	Truncate double to float.

Function	Description
__truncdfhf2	Truncate double to IEEE half-precision float.
__truncsfhf2	Truncate float to IEEE half-precision float.

5.3.3.1 __fixhfsi()

Description

Convert half-precision float to int.

Prototype

```
__SEGGER_RTL_I32 __fixhfsi(__SEGGER_RTL_FLOAT16 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.2 __fixsfsi()

Description

Convert float to int.

Prototype

```
__SEGGER_RTL_I32 __fixsfsi(float x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.3 __fixdfsi()

Description

Convert double to int.

Prototype

```
__SEGGER_RTL_I32 __fixdfsi(double x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.4 __fixtfsi()

Description

Convert long double to int.

Prototype

```
__SEGGER_RTL_I32 __fixtfsi(long double x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.5 __fixhfdi()

Description

Convert half-precision float to int.

Prototype

```
__SEGGER_RTL_I64 __fixhfdi(__SEGGER_RTL_FLOAT16 x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.6 __fixsfdi()

Description

Convert float to long long.

Prototype

```
__SEGGER_RTL_I64 __fixsfdi(float f);
```

Parameters

Parameter	Description
f	Floating value to convert.

Return value

Integerized value.

Notes

The RV32 compiler converts a float to a 64-bit integer by calling runtime support to handle it.

Thread safety

Safe.

5.3.3.7 __fixdfdi()

Description

Convert double to long long.

Prototype

```
__SEGGER_RTL_I64 __fixdfdi(double x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Notes

RV32 always calls runtime for double to int64 conversion.

Thread safety

Safe.

5.3.3.8 __fixtfdi()

Description

Convert long double to long long.

Prototype

```
__SEGGER_RTL_I64 __fixtfdi(long double x);
```

Parameters

Parameter	Description
x	Floating value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.9 __fixunshfsi()

Description

Convert half-precision float to unsigned.

Prototype

```
unsigned __fixunshfsi(__SEGGER_RTL_FLOAT16 x);
```

Parameters

Parameter	Description
x	Float value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.10 __fixunssfsi()

Description

Convert float to unsigned.

Prototype

```
__SEGGER_RTL_U32 __fixunssfsi(float x);
```

Parameters

Parameter	Description
x	Float value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.11 __fixunsdfsi()

Description

Convert double to unsigned.

Prototype

```
__SEGGER_RTL_U32 __fixunsdfsi(double x);
```

Parameters

Parameter	Description
x	Float value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.12 __fixunstfsi()

Description

Convert long double to int.

Prototype

```
int __fixunstfsi(long double x);
```

Parameters

Parameter	Description
x	Float value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.13 __fixunshfdi()

Description

Convert half-precision float to unsigned.

Prototype

```
__SEGGER_RTL_I64 __fixunshfdi(__SEGGER_RTL_FLOAT16 x);
```

Parameters

Parameter	Description
x	Float value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.14 __fixunssfdi()

Description

Convert float to unsigned long long.

Prototype

```
__SEGGER_RTL_U64 __fixunssfdi(float f);
```

Parameters

Parameter	Description
f	Float value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.15 __fixunsdfdi()

Description

Convert double to unsigned long long.

Prototype

```
__SEGGER_RTL_U64 __fixunsdfdi(double x);
```

Parameters

Parameter	Description
x	Float value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.16 __fixunstfdi()

Description

Convert long double to unsigned long long.

Prototype

```
__SEGGER_RTL_U64 __fixunstfdi(long double x);
```

Parameters

Parameter	Description
x	Float value to convert.

Return value

Integerized value.

Thread safety

Safe.

5.3.3.17 __floatsihf()

Description

Convert int to half-precision float.

Prototype

```
__SEGGER_RTL_FLOAT16 __floatsihf(__SEGGER_RTL_I32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.18 __floatsisf()

Description

Convert int to float.

Prototype

```
float __floatsisf(__SEGGER_RTL_I32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.19 __floatsidf()

Description

Convert int to double.

Prototype

```
double __floatsidf(__SEGGER_RTL_I32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.20 __floatsitf()

Description

Convert int to long double.

Prototype

```
long double __floatsitf(__SEGGER_RTL_I32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.21 __floatdihf()

Description

Convert long long to half-precision float.

Prototype

```
__SEGGER_RTL_FLOAT16 __floatdihf(__SEGGER_RTL_I64 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.22 __floatdisf()

Description

Convert long long to float.

Prototype

```
float __floatdisf(__SEGGER_RTL_I64 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.23 __floatdidf()

Description

Convert long long to double.

Prototype

```
double __floatdidf(__SEGGER_RTL_I64 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.24 __floatditf()

Description

Convert long long to long double.

Prototype

```
long double __floatditf(__SEGGER_RTL_I64 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.25 __floatunsihf()

Description

Convert unsigned to half-precision float.

Prototype

```
__SEGGER_RTL_FLOAT16 __floatunsihf(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.26 __floatunsisf()

Description

Convert unsigned to float.

Prototype

```
float __floatunsisf(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Integer value to convert.

Return value

Floating value.

Thread safety

Safe.

5.3.3.27 __floatunsidf()

Description

Convert unsigned to double.

Prototype

```
double __floatunsidf(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Unsigned value to convert.

Return value

Double value.

Thread safety

Safe.

5.3.3.28 __floatunsitf()

Description

Convert unsigned to long double.

Prototype

```
long double __floatunsitf(__SEGGER_RTL_U32 x);
```

Parameters

Parameter	Description
x	Unsigned value to convert.

Return value

Long double value.

Thread safety

Safe.

5.3.3.29 __floatundihf()

Description

Convert unsigned long long to half-precision float.

Prototype

```
__SEGGER_RTL_FLOAT16 __floatundihf(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Unsigned long long value to convert.

Return value

Float value.

Thread safety

Safe.

5.3.3.30 __floatundisf()

Description

Convert unsigned long long to float.

Prototype

```
float __floatundisf(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Unsigned long long value to convert.

Return value

Float value.

Thread safety

Safe.

5.3.3.31 __floatundidf()

Description

Convert unsigned long long to double.

Prototype

```
double __floatundidf(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Unsigned long long value to convert.

Return value

Double value.

Thread safety

Safe.

5.3.3.32 __floatunditf()

Description

Convert unsigned long long to long double.

Prototype

```
long double __floatunditf(__SEGGER_RTL_U64 x);
```

Parameters

Parameter	Description
x	Unsigned long long value to convert.

Return value

Long double value.

Thread safety

Safe.

5.3.3.33 __extendhfsf2()

Description

Convert IEEE half-precision float to float.

Prototype

```
float __extendhfsf2(__SEGGER_RTL_FLOAT16 x);
```

Parameters

Parameter	Description
x	Half-precision float.

Return value

Single-precision float.

Thread safety

Safe.

5.3.3.34 __extendhfdf2()

Description

Convert IEEE half-precision float to double.

Prototype

```
double __extendhfdf2(__SEGGER_RTL_FLOAT16 x);
```

Parameters

Parameter	Description
x	Half-precision float.

Return value

Double-precision float.

Thread safety

Safe.

5.3.3.35 __extendhftf2()

Description

Convert IEEE half-precision float to long double.

Prototype

```
long double __extendhftf2(__SEGGER_RTL_FLOAT16 x);
```

Parameters

Parameter	Description
x	Half-precision float.

Return value

Long-double float.

Thread safety

Safe.

5.3.3.36 __extendsfdf2()

Description

Extend float to double.

Prototype

```
double __extendsfdf2(float x);
```

Parameters

Parameter	Description
x	Float value to extend.

Return value

Double value.

Thread safety

Safe.

5.3.3.37 __extendsftf2()

Description

Extend float to long double.

Prototype

```
long double __extendsftf2(float x);
```

Parameters

Parameter	Description
x	Float value to extend.

Return value

Double value.

Thread safety

Safe.

5.3.3.38 __extenddftf2()

Description

Extend double to long double.

Prototype

```
long double __extenddftf2(double x);
```

Parameters

Parameter	Description
x	Double value to extend.

Return value

Long double value.

Thread safety

Safe.

© 2003-2022 SEGGER Microcontroller GmbH

5.3.3.39 __trunctfdf2()

Description

Truncate long double to double.

Prototype

```
double __trunctfdf2(long double x);
```

Parameters

Parameter	Description
x	Long double value to truncate.

Return value

Double value.

Thread safety

Safe.

5.3.3.40 __trunctfsf2()

Description

Truncate long double to float.

Prototype

```
float __trunctfsf2(long double x);
```

Parameters

Parameter	Description
x	Long double value to truncate.

Return value

Float value.

Thread safety

Safe.

5.3.3.41 __trunctfhf2()

Description

Truncate long double to IEEE half-precision float.

Prototype

```
__SEGGER_RTL_FLOAT16 __trunctfhf2(long double x);
```

Parameters

Parameter	Description
x	Long-double value to truncate.

Return value

Half-precision value.

Thread safety

Safe.

5.3.3.42 __truncdfsf2()

Description

Truncate double to float.

Prototype

```
float __truncdfsf2(double x);
```

Parameters

Parameter	Description
x	Double value to truncate.

Return value

Float value.

Thread safety

Safe.

5.3.3.43 __truncdfhf2()

Description

Truncate double to IEEE half-precision float.

Prototype

```
__SEGGER_RTL_FLOAT16 __truncdfhf2(double x);
```

Parameters

Parameter	Description
x	Double value to truncate.

Return value

Half-precision value.

Thread safety

Safe.

5.3.3.44 __truncsfhf2()

Description

Truncate float to IEEE half-precision float.

Prototype

```
__SEGGER_RTL_FLOAT16 __truncsfhf2(float x);
```

Parameters

Parameter	Description
x	Float value to truncate.

Return value

Float value.

Thread safety

Safe.

5.3.4 Complex arithmetic

Function	Description
__mulsc3	Multiply, float complex.
__muldc3	Multiply, double complex.
__divsc3	Divide, float complex.
__divdc3	Divide, double complex.

5.3.4.1 __mulsc3()

Description

Multiply, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX __mulsc3(float a,
                                         float b,
                                         float c,
                                         float d);
```

Parameters

Parameter	Description
a	Real part of multiplicand.
b	Imaginary part of multiplicand.
c	Real part of multiplier.
d	Imaginary part of multiplier.

Return value

Product.

Thread safety

Safe.

5.3.4.2 __muldc3()

Description

Multiply, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX __muldc3(double a,
                                        double b,
                                        double c,
                                               double d);
```

Parameters

Parameter	Description
a	Real part of multiplicand.
b	Imaginary part of multiplicand.
c	Real part of multiplier.
d	Imaginary part of multiplier.

Return value

Product.

Thread safety

Safe.

5.3.4.3 __multc3()

Description

Multiply, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX __multc3(long double a,
                                        long double b,
                                        long double c,
                                        long        double d);
```

Parameters

Parameter	Description
a	Real part of multiplicand.
b	Imaginary part of multiplicand.
c	Real part of multiplier.
d	Imaginary part of multiplier.

Return value

Product.

Thread safety

Safe.

5.3.4.4 __divsc3()

Description

Divide, float complex.

Prototype

```
__SEGGER_RTL_FLOAT32_C_COMPLEX __divsc3(float a,
                                        float b,
                                        float c,
                                        float d);
```

Parameters

Parameter	Description
a	Real part of dividend.
b	Imaginary part of dividend.
c	Real part of divisor.
d	Imaginary part of divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.4.5 __divdc3()

Description

Divide, double complex.

Prototype

```
__SEGGER_RTL_FLOAT64_C_COMPLEX __divdc3(double a,
                                        double b,
                                        double c,
                                               double d);
```

Parameters

Parameter	Description
a	Real part of dividend.
b	Imaginary part of dividend.
c	Real part of divisor.
d	Imaginary part of divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.4.6 __divtc3()

Description

Divide, long double complex.

Prototype

```
__SEGGER_RTL_LDOUBLE_C_COMPLEX __divtc3(long double a,
                                        long double b,
                                        long double c,
                                        long        double d);
```

Parameters

Parameter	Description
a	Real part of dividend.
b	Imaginary part of dividend.
c	Real part of divisor.
d	Imaginary part of divisor.

Return value

Quotient.

Thread safety

Safe.

5.3.5 Floating comparisons

Function	Description
__eqhf2	Equal, half-precision float.
__eqsf2	Equal, float.
__eqdf2	Equal, double.
__eqtf2	Equal, long double.
__nehf2	Not equal, half-precision float.
__nesf2	Not equal, float.
__nedf2	Not equal, double.
__netf2	Not equal, long double.
__lthf2	Less than, half-precision float.
__ltsf2	Less than, float.
__ltdf2	Less than, double.
__lttf2	Less than, long double.
__lehf2	Less than or equal, half-precision float.
__lesf2	Less than or equal, float.
__ledf2	Less than or equal, double.
__letf2	Less than or equal, long double.
__gthf2	Greater than, half-precision float.
__gtsf2	Greater than, float.
__gtdf2	Greater than, double.
__gttf2	Greater than, long double.
__gehf2	Greater than or equal, half-precision float.
__gesf2	Greater than or equal, float.
__gedf2	Greater than or equal, double.
__getf2	Greater than or equal, long double.
__unordsf2	Unordered operand query, float.
__unorddf2	Unordered operand query, double.
__unordtf2	Unordered operand query, long double.

5.3.5.1 __eqhf2()

Description

Equal, half-precision float.

Prototype

```
int __eqhf2(__SEGGER_RTL_FLOAT16 x,
            __SEGGER_RTL_FLOAT16 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return = 0 if both operands are non-NaN and a = b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.2 __eqsf2()

Description

Equal, float.

Prototype

```
int __eqsf2(float x,
            float y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return = 0 if both operands are non-NaN and a = b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.3 __eqdf2()

Description

Equal, double.

Prototype

```
int __eqdf2(double x,
            double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return = 0 if both operands are non-NaN and a = b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.4 __eqtf2()

Description

Equal, long double.

Prototype

```
int __eqtf2(long double x,
            long double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return = 0 if both operands are non-NaN and a = b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.5 __nehf2()

Description

Not equal, half-precision float.

Prototype

```
int __nehf2(__SEGGER_RTL_FLOAT16 x,
            __SEGGER_RTL_FLOAT16 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return = 0 if both operands are non-NaN and a = b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.6 __nesf2()

Description

Not equal, float.

Prototype

```
int __nesf2(float x,
            float y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return = 0 if both operands are non-NaN and a = b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.7 __nedf2()

Description

Not equal, double.

Prototype

```
int __nedf2(double x,
            double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return = 0 if both operands are non-NaN and a = b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.8 __netf2()

Description

Not equal, long double.

Prototype

```
int __netf2(long double x,
            long double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return = 0 if both operands are non-NaN and a = b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.9 __lthf2()

Description

Less than, half-precision float.

Prototype

```
int __lthf2(__SEGGER_RTL_FLOAT16 x,
            __SEGGER_RTL_FLOAT16 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return < 0 if both operands are non-NaN and a < b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.10 __ltsf2()

Description

Less than, float.

Prototype

```
int __ltsf2(float x,
            float y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return < 0 if both operands are non-NaN and a < b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.11 __ltdf2()

Description

Less than, double.

Prototype

```
int __ltdf2(double x,
            double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return < 0 if both operands are non-NaN and a < b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.12 __lttf2()

Description

Less than, long double.

Prototype

```
int __lttf2(long double x,
            long double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return < 0 if both operands are non-NaN and a < b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.13 __lehf2()

Description

Less than or equal, half-precision float.

Prototype

```
int __lehf2(__SEGGER_RTL_FLOAT16 x,
            __SEGGER_RTL_FLOAT16 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return ≤ 0 if both operands are non-NaN and a < b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.14 __lesf2()

Description

Less than or equal, float.

Prototype

```
int __lesf2(float x,
            float y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return ≤ 0 if both operands are non-NaN and a < b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.15 __ledf2()

Description

Less than or equal, double.

Prototype

```
int __ledf2(double x,
            double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return ≤ 0 if both operands are non-NaN and a < b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.16 __letf2()

Description

Less than or equal, long double.

Prototype

```
int __letf2(long double x,
            long double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return ≤ 0 if both operands are non-NaN and a ≤ b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.17 __gthf2()

Description

Greater than, half-precision float.

Prototype

```
int __gthf2(__SEGGER_RTL_FLOAT16 x,
            __SEGGER_RTL_FLOAT16 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return > 0 if both operands are non-NaN and a > b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.18 __gtsf2()

Description

Greater than, float.

Prototype

```
int __gtsf2(float x,
            float y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return > 0 if both operands are non-NaN and a > b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.19 __gtdf2()

Description

Greater than, double.

Prototype

```
int __gtdf2(double x,
            double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return > 0 if both operands are non-NaN and a > b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.20 __gttf2()

Description

Greater than, long double.

Prototype

```
int __gttf2(long double x,
            long double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return > 0 if both operands are non-NaN and a > b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.21 __gehf2()

Description

Greater than or equal, half-precision float.

Prototype

```
int __gehf2(__SEGGER_RTL_FLOAT16 x,
            __SEGGER_RTL_FLOAT16 y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return ≥ 0 if both operands are non-NaN and a ≥ b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.22 __gesf2()

Description

Greater than or equal, float.

Prototype

```
int __gesf2(float x,
            float y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return ≥ 0 if both operands are non-NaN and a ≥ b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.23 __gedf2()

Description

Greater than or equal, double.

Prototype

```
int __gedf2(double x,
            double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return ≥ 0 if both operands are non-NaN and a ≥ b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.24 __getf2()

Description

Greater than or equal, long double.

Prototype

```
int __getf2(long double x,
            long double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return ≥ 0 if both operands are non-NaN and a ≥ b (GNU three-way boolean).

Thread safety

Safe.

5.3.5.25 __unordsf2()

Description

Unordered operand query, float.

Prototype

```
int __unordsf2(float x,
               float y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return nonzero if comparison between operands is unordered.

Thread safety

Safe.

5.3.5.26 __unorddf2()

Description

Unordered operand query, double.

Prototype

```
int __unorddf2(double x,
               double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return nonzero if comparison between operands is unordered.

Thread safety

Safe.

5.3.5.27 __unordtf2()

Description

Unordered operand query, long double.

Prototype

```
int __unordtf2(long double x,
               long double y);
```

Parameters

Parameter	Description
x	Left-hand operand.
y	Right-hand operand.

Return value

Return nonzero if comparison between operands is unordered.

Thread safety

Safe.

5.3.6 Atomic operations

Function	Description
__atomic_load	Generic atomic load.
__atomic_store	Generic atomic store.
__atomic_is_lock_free	Inquire whether objects always generate lock-free atomic instructions.
__atomic_load_1	Atomic fetch, 8-bit object.
__atomic_load_2	Atomic fetch, 16-bit object.
__atomic_load_4	Atomic fetch, 32-bit object.
__atomic_load_8	Atomic fetch, 64-bit object.
__atomic_load_16	Atomic fetch, 128-bit object.
__atomic_store_1	Atomic store, 8-bit object.
__atomic_store_2	Atomic store, 16-bit object.
__atomic_store_4	Atomic store, 32-bit object.
__atomic_store_8	Atomic store, 64-bit object.
__atomic_store_16	Atomic store, 64-bit object.
__atomic_add_fetch_1	Atomic fetch with increment, 8-bit object.
__atomic_add_fetch_2	Atomic fetch with increment, 16-bit object.
__atomic_add_fetch_4	Atomic fetch with increment, 32-bit object.
__atomic_add_fetch_8	Atomic fetch with increment, 64-bit object.
__atomic_add_fetch_16	Atomic fetch with increment, 128-bit object.
__atomic_sub_fetch_1	Atomic fetch with decrement, 8-bit object.
__atomic_sub_fetch_2	Atomic fetch with decrement, 16-bit object.
__atomic_sub_fetch_4	Atomic fetch with decrement, 32-bit object.
__atomic_sub_fetch_8	Atomic fetch with decrement, 64-bit object.
__atomic_sub_fetch_16	Atomic fetch with decrement, 128-bit object.
__atomic_and_fetch_1	Atomic fetch with bitwise-and, 8-bit object.
__atomic_and_fetch_2	Atomic fetch with bitwise-and, 16-bit object.
__atomic_and_fetch_4	Atomic fetch with bitwise-and, 32-bit object.
__atomic_and_fetch_8	Atomic fetch with bitwise-and, 64-bit object.
__atomic_and_fetch_16	Atomic fetch with bitwise-and, 128-bit object.
__atomic_or_fetch_1	Atomic fetch with bitwise-or, 8-bit object.
__atomic_or_fetch_2	Atomic fetch with bitwise-or, 16-bit object.
__atomic_or_fetch_4	Atomic fetch with bitwise-or, 32-bit object.
__atomic_or_fetch_8	Atomic fetch with bitwise-or, 64-bit object.
__atomic_or_fetch_16	Atomic fetch with bitwise-or, 128-bit object.
__atomic_xor_fetch_1	Atomic fetch with bitwise-exclusive-or, 8-bit object.
__atomic_xor_fetch_2	Atomic fetch with bitwise-exclusive-or, 16-bit object.
__atomic_xor_fetch_4	Atomic fetch with bitwise-exclusive-or, 32-bit object.
__atomic_xor_fetch_8	Atomic fetch with bitwise-exclusive-or, 64-bit object.
__atomic_xor_fetch_16	Atomic fetch with bitwise-exclusive-or, 128-bit object.
__atomic_nand_fetch_1	Atomic fetch with bitwise-nand, 8-bit object.
__atomic_nand_fetch_2	Atomic fetch with bitwise-nand, 16-bit object.
__atomic_nand_fetch_4	Atomic fetch with bitwise-nand, 32-bit object.

Function	Description
__atomic_nand_fetch_8	Atomic fetch with bitwise-nand, 64-bit object.
__atomic_nand_fetch_16	Atomic fetch with bitwise-nand, 128-bit object.
__atomic_fetch_add_1	Atomic fetch with increment, 8-bit object.
__atomic_fetch_add_2	Atomic fetch with increment, 16-bit object.
__atomic_fetch_add_4	Atomic fetch with increment, 32-bit object.
__atomic_fetch_add_8	Atomic fetch with increment, 64-bit object.
__atomic_fetch_add_16	Atomic fetch with increment, 128-bit object.
__atomic_fetch_sub_1	Atomic fetch with increment, 8-bit object.
__atomic_fetch_sub_2	Atomic fetch with increment, 16-bit object.
__atomic_fetch_sub_4	Atomic fetch with increment, 32-bit object.
__atomic_fetch_sub_8	Atomic fetch with increment, 64-bit object.
__atomic_fetch_sub_16	Atomic fetch with increment, 128-bit object.
__atomic_fetch_and_1	Atomic fetch with increment, 8-bit object.
__atomic_fetch_and_2	Atomic fetch with increment, 16-bit object.
__atomic_fetch_and_4	Atomic fetch with increment, 32-bit object.
__atomic_fetch_and_8	Atomic fetch with increment, 64-bit object.
__atomic_fetch_and_16	Atomic fetch with increment, 128-bit object.
__atomic_fetch_or_1	Atomic fetch with increment, 8-bit object.
__atomic_fetch_or_2	Atomic fetch with increment, 16-bit object.
__atomic_fetch_or_4	Atomic fetch with increment, 32-bit object.
__atomic_fetch_or_8	Atomic fetch with increment, 64-bit object.
__atomic_fetch_or_16	Atomic fetch with increment, 128-bit object.
__atomic_fetch_xor_1	Atomic fetch with increment, 8-bit object.
__atomic_fetch_xor_2	Atomic fetch with increment, 16-bit object.
__atomic_fetch_xor_4	Atomic fetch with increment, 32-bit object.
__atomic_fetch_xor_8	Atomic fetch with increment, 64-bit object.
__atomic_fetch_xor_16	Atomic fetch with increment, 128-bit object.
__atomic_fetch_nand_1	Atomic fetch with increment, 8-bit object.
__atomic_fetch_nand_2	Atomic fetch with increment, 16-bit object.
__atomic_fetch_nand_4	Atomic fetch with increment, 32-bit object.
__atomic_fetch_nand_8	Atomic fetch with increment, 64-bit object.
__atomic_fetch_nand_16	Atomic fetch with increment, 128-bit object.
Legacy atomic operations	
__sync_fetch_and_add_1	Atomic fetch with increment, 8-bit object.
__sync_fetch_and_add_2	Atomic fetch with increment, 16-bit object.
__sync_fetch_and_add_4	Atomic fetch with increment, 32-bit object.
__sync_fetch_and_add_8	Atomic fetch with increment, 64-bit object.
__sync_fetch_and_add_16	Atomic fetch with increment, 128-bit object.
__sync_fetch_and_sub_1	Atomic fetch with decrement, 8-bit object.
__sync_fetch_and_sub_2	Atomic fetch with decrement, 16-bit object.
__sync_fetch_and_sub_4	Atomic fetch with decrement, 32-bit object.
__sync_fetch_and_sub_8	Atomic fetch with decrement, 64-bit object.
__sync_fetch_and_sub_16	Atomic fetch with decrement, 128-bit object.

Function	Description
__sync_fetch_and_and_1	Atomic fetch with bitwise-and, 8-bit object.
__sync_fetch_and_and_2	Atomic fetch with bitwise-and, 16-bit object.
__sync_fetch_and_and_4	Atomic fetch with bitwise-and, 32-bit object.
__sync_fetch_and_and_8	Atomic fetch with bitwise-and, 64-bit object.
__sync_fetch_and_and_16	Atomic fetch with bitwise-and, 128-bit object.
__sync_fetch_and_or_1	Atomic fetch with bitwise-or, 8-bit object.
__sync_fetch_and_or_2	Atomic fetch with bitwise-or, 16-bit object.
__sync_fetch_and_or_4	Atomic fetch with bitwise-or, 32-bit object.
__sync_fetch_and_or_8	Atomic fetch with bitwise-or, 64-bit object.
__sync_fetch_and_or_16	Atomic fetch with bitwise-or, 128-bit object.
__sync_fetch_and_xor_1	Atomic fetch with bitwise-exclusive-or, 8-bit object.
__sync_fetch_and_xor_2	Atomic fetch with bitwise-exclusive-or, 16-bit object.
__sync_fetch_and_xor_4	Atomic fetch with bitwise-exclusive-or, 32-bit object.
__sync_fetch_and_xor_8	Atomic fetch with bitwise-exclusive-or, 64-bit object.
__sync_fetch_and_xor_16	Atomic fetch with bitwise-exclusive-or, 128-bit object.
__sync_fetch_and_nand_1	Atomic fetch with bitwise-nand, 8-bit object.
__sync_fetch_and_nand_2	Atomic fetch with bitwise-nand, 16-bit object.
__sync_fetch_and_nand_4	Atomic fetch with bitwise-nand, 32-bit object.
__sync_fetch_and_nand_8	Atomic fetch with bitwise-nand, 64-bit object.
__sync_fetch_and_nand_16	Atomic fetch with bitwise-nand, 128-bit object.
__sync_add_and_fetch_1	Atomic fetch with increment, 8-bit object.
__sync_add_and_fetch_2	Atomic fetch with increment, 16-bit object.
__sync_add_and_fetch_4	Atomic fetch with increment, 32-bit object.
__sync_add_and_fetch_8	Atomic fetch with increment, 64-bit object.
__sync_add_and_fetch_16	Atomic fetch with increment, 128-bit object.
__sync_sub_and_fetch_1	Atomic fetch with decrement, 8-bit object.
__sync_sub_and_fetch_2	Atomic fetch with decrement, 16-bit object.
__sync_sub_and_fetch_4	Atomic fetch with decrement, 32-bit object.
__sync_sub_and_fetch_8	Atomic fetch with decrement, 64-bit object.
__sync_sub_and_fetch_16	Atomic fetch with decrement, 128-bit object.
__sync_and_and_fetch_1	Atomic bitwise-and and fetch, 8-bit object.
__sync_and_and_fetch_2	Atomic bitwise-and and fetch, 16-bit object.
__sync_and_and_fetch_4	Atomic bitwise-and and fetch, 32-bit object.
__sync_and_and_fetch_8	Atomic bitwise-and and fetch, 64-bit object.
__sync_and_and_fetch_16	Atomic bitwise-and and fetch, 128-bit object.
__sync_or_and_fetch_1	Atomic bitwise-or and fetch, 8-bit object.
__sync_or_and_fetch_2	Atomic bitwise-or and fetch, 16-bit object.
__sync_or_and_fetch_4	Atomic bitwise-or and fetch, 32-bit object.
__sync_or_and_fetch_8	Atomic bitwise-or and fetch, 64-bit object.
__sync_or_and_fetch_16	Atomic bitwise-or and fetch, 128-bit object.
__sync_xor_and_fetch_1	Atomic bitwise-exclusive-or and fetch, 8-bit object.
__sync_xor_and_fetch_2	Atomic bitwise-exclusive-or and fetch, 16-bit object.
__sync_xor_and_fetch_4	Atomic bitwise-exclusive-or and fetch, 32-bit object.

Function	Description
__sync_xor_and_fetch_8	Atomic bitwise-exclusive-or and fetch, 64-bit object.
__sync_xor_and_fetch_16	Atomic bitwise-exclusive-or and fetch, 128-bit object.
__sync_nand_and_fetch_1	Atomic bitwise-nand and fetch, 8-bit object.
__sync_nand_and_fetch_2	Atomic bitwise-nand and fetch, 16-bit object.
__sync_nand_and_fetch_4	Atomic bitwise-nand and fetch, 32-bit object.
__sync_nand_and_fetch_8	Atomic bitwise-nand and fetch, 64-bit object.
__sync_nand_and_fetch_16	Atomic bitwise-nand and fetch, 128-bit object.
__sync_bool_compare_and_swap_1	Atomic compare and swap, 8-bit object.
__sync_bool_compare_and_swap_2	Atomic compare and swap, 16-bit object.
__sync_bool_compare_and_swap_4	Atomic compare and swap, 32-bit object.
__sync_bool_compare_and_swap_8	Atomic compare and swap, 64-bit object.
__sync_bool_compare_and_swap_16	Atomic compare and swap, 128-bit object.
__sync_val_compare_and_swap_1	Atomic compare and swap, 8-bit object.
__sync_val_compare_and_swap_2	Atomic compare and swap, 16-bit object.
__sync_val_compare_and_swap_4	Atomic compare and swap, 32-bit object.
__sync_val_compare_and_swap_8	Atomic compare and swap, 64-bit object.
__sync_val_compare_and_swap_16	Atomic compare and swap, 128-bit object.
__sync_lock_test_and_set_1	Acquire lock, 8-bit object.
__sync_lock_test_and_set_2	Acquire lock, 16-bit object.
__sync_lock_test_and_set_4	Acquire lock, 32-bit object.
__sync_lock_test_and_set_8	Acquire lock, 64-bit object.
__sync_lock_test_and_set_16	Acquire lock, 128-bit object.
__sync_lock_release_1	Release lock, 8-bit object.
__sync_lock_release_2	Release lock, 16-bit object.
__sync_lock_release_4	Release lock, 32-bit object.
__sync_lock_release_8	Release lock, 64-bit object.
__sync_lock_release_16	Release lock, 128-bit object.

5.3.6.1 __atomic_load()

Description

Generic atomic load.

Prototype

```
void __atomic_load(                          size_t size,
                   const volatile void * ptr,
                                  void * ret,
                                  int    memorder);
```

Parameters

Parameter	Description
size	Size of object to load.
ptr	Pointer to object to load.
ret	Pointer to where object is written.
memorder	Memory ordering (__ATOMIC_RELAXED, __ATOMIC_SEQ_CST, or __ATOMIC_RELEASE).

5.3.6.2 __atomic_store()

Description

Generic atomic store.

Prototype

```
void __atomic_store(                size_t size,
                    volatile void * ptr,
                             void * val,
                             int    memorder);
```

Parameters

Parameter	Description
size	Size of object to store.
ptr	Pointer to where object is written.
val	Pointer to object to store.
memorder	Memory ordering.

5.3.6.3 __atomic_is_lock_free()

Description

Inquire whether objects always generate lock-free atomic instructions.

Prototype

```
__SEGGER_RTL_BOOL __atomic_is_lock_free(                          size_t size,
                                        const volatile void * ptr);
```

Parameters

Parameter	Description
size	Size of object under consideration.
ptr	Pointer to object under consideration.

Return value

True if objects of size bytes always generate lock-free atomic instructions for the target architecture.

Additional information

ptr is an optional pointer to the object that may be used to determine alignment. A NULL pointer indicates typical alignment should be used.

5.3.7 __atomic_load_1()

Description

Atomic fetch, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_load_1(const volatile void * ptr,
                                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to load.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.8 __atomic_load_2()

Description

Atomic fetch, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_load_2(const volatile void * ptr,
                                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to load.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.9 __atomic_load_4()

Description

Atomic fetch, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_load_4(const volatile void * ptr,
                                 int                    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to load.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.10 __atomic_load_8()

Description

Atomic fetch, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_load_8(const volatile void * ptr,
                                                     int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to load.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.11 __atomic_load_16()

Description

Atomic fetch, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_load_16(const volatile void * ptr,
                                                   int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to load.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.12 __atomic_store_1()

Description

Atomic store, 8-bit object.

Prototype

```
void __atomic_store_1(volatile void * ptr,
                                      __SEGGER_RTL_U8 val,
                                  int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to store to.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.13 __atomic_store_2()

Description

Atomic store, 16-bit object.

Prototype

```
void __atomic_store_2(volatile void * ptr,
                                       __SEGGER_RTL_U16 val,
                                       int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to store to.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.14 __atomic_store_4()

Description

Atomic store, 32-bit object.

Prototype

```
void __atomic_store_4(volatile void * ptr,
                                       __SEGGER_RTL_U32 val,
                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to store to.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.15 __atomic_store_8()

Description

Atomic store, 64-bit object.

Prototype

```
void __atomic_store_8(volatile void * ptr,
                                      __SEGGER_RTL_U64 val,
                                      int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to store to.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.16 __atomic_store_16()

Description

Atomic store, 64-bit object.

Prototype

```
void __atomic_store_16(volatile void * ptr,
                                       __SEGGER_RTL_U128 val,
                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to store to.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr.

5.3.17 __atomic_add_fetch_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_add_fetch_1(volatile void * ptr,
                                                     __SEGGER_RTL_U8 value,
                                                     int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.18 __atomic_add_fetch_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_add_fetch_2(volatile void * ptr,
                                                      __SEGGER_RTL_U16 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.19 __atomic_add_fetch_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_add_fetch_4(volatile void * ptr,
                                                     __SEGGER_RTL_U32 value,
                                              int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.20 __atomic_add_fetch_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_add_fetch_8(volatile void * ptr,
                                                   __SEGGER_RTL_U64 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.21 __atomic_add_fetch_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_add_fetch_16(volatile void * ptr,
                                                         __SEGGER_RTL_U128 value,
                                                  int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.22 __atomic_sub_fetch_1()

Description

Atomic fetch with decrement, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_sub_fetch_1(volatile void * ptr,
                                                  __SEGGER_RTL_U8 value,
                                             int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.23 __atomic_sub_fetch_2()

Description

Atomic fetch with decrement, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_sub_fetch_2(volatile void * ptr,
                                                      __SEGGER_RTL_U16 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.24 __atomic_sub_fetch_4()

Description

Atomic fetch with decrement, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_sub_fetch_4(volatile void * ptr,
                                                     __SEGGER_RTL_U32 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.25 __atomic_sub_fetch_8()

Description

Atomic fetch with decrement, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_sub_fetch_8(volatile void * ptr,
                                                        __SEGGER_RTL_U64 value,
                                                 int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.26 __atomic_sub_fetch_16()

Description

Atomic fetch with decrement, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_sub_fetch_16(volatile void * ptr,
                                                     __SEGGER_RTL_U128 value,
                                                 int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.27 __atomic_and_fetch_1()

Description

Atomic fetch with bitwise-and, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_and_fetch_1(volatile void * ptr,
                                                     __SEGGER_RTL_U8 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.28 __atomic_and_fetch_2()

Description

Atomic fetch with bitwise-and, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_and_fetch_2(volatile void * ptr,
                                                  __SEGGER_RTL_U16 value,
                                           int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.29 __atomic_and_fetch_4()

Description

Atomic fetch with bitwise-and, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_and_fetch_4(volatile void * ptr,
                                                    __SEGGER_RTL_U32 value,
                                                    int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.30 __atomic_and_fetch_8()

Description

Atomic fetch with bitwise-and, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_and_fetch_8(volatile void * ptr,
                                                   __SEGGER_RTL_U64 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.31 __atomic_and_fetch_16()

Description

Atomic fetch with bitwise-and, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_and_fetch_16(volatile void * ptr,
                                                         __SEGGER_RTL_U128 value,
                                                 int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.32 __atomic_or_fetch_1()

Description

Atomic fetch with bitwise-or, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_or_fetch_1(volatile void * ptr,
                                                    __SEGGER_RTL_U8 value,
                                             int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.33 __atomic_or_fetch_2()

Description

Atomic fetch with bitwise-or, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_or_fetch_2(volatile void * ptr,
                                                      __SEGGER_RTL_U16 value,
                                                 int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.34 __atomic_or_fetch_4()

Description

Atomic fetch with bitwise-or, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_or_fetch_4(volatile void * ptr,
                                             __SEGGER_RTL_U32 value,
                                     int      memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.35 __atomic_or_fetch_8()

Description

Atomic fetch with bitwise-or, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_or_fetch_8(volatile void * ptr,
                                                 __SEGGER_RTL_U64 value,
                                                 int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.36 __atomic_or_fetch_16()

Description

Atomic fetch with bitwise-or, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_or_fetch_16(volatile void * ptr,
                                                  __SEGGER_RTL_U128 value,
                                                  int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.37 __atomic_xor_fetch_1()

Description

Atomic fetch with bitwise-exclusive-or, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_xor_fetch_1(volatile void * ptr,
                                                     __SEGGER_RTL_U8 value,
                                            int      memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.38 __atomic_xor_fetch_2()

Description

Atomic fetch with bitwise-exclusive-or, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_xor_fetch_2(volatile void * ptr,
                                                      __SEGGER_RTL_U16 value,
                                               int    memorder);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.
`memorder`	Memory ordering.

Return value

Value of object pointed to by `ptr` after the update.

5.3.39 __atomic_xor_fetch_4()

Description

Atomic fetch with bitwise-exclusive-or, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_xor_fetch_4(volatile void * ptr,
                                                     __SEGGER_RTL_U32 value,
                                              int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.40 __atomic_xor_fetch_8()

Description

Atomic fetch with bitwise-exclusive-or, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_xor_fetch_8(volatile void * ptr,
                                                     __SEGGER_RTL_U64 value,
                                              int    memorder);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.
`memorder`	Memory ordering.

Return value

Value of object pointed to by `ptr` after the update.

5.3.41 __atomic_xor_fetch_16()

Description

Atomic fetch with bitwise-exclusive-or, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_xor_fetch_16(volatile void * ptr,
                                                      __SEGGER_RTL_U128 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.42 __atomic_nand_fetch_1()

Description

Atomic fetch with bitwise-nand, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_nand_fetch_1(volatile void * ptr,
                                                      __SEGGER_RTL_U8 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.43 __atomic_nand_fetch_2()

Description

Atomic fetch with bitwise-nand, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_nand_fetch_2(volatile void * ptr,
                                                       __SEGGER_RTL_U16 value,
                                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.44 __atomic_nand_fetch_4()

Description

Atomic fetch with bitwise-nand, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_nand_fetch_4(volatile void * ptr,
                                                        __SEGGER_RTL_U32 value,
                                                 int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.45 __atomic_nand_fetch_8()

Description

Atomic fetch with bitwise-nand, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_nand_fetch_8(volatile void * ptr,
                                                         __SEGGER_RTL_U64 value,
                                                  int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.46 __atomic_nand_fetch_16()

Description

Atomic fetch with bitwise-nand, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_nand_fetch_16(volatile void * ptr,
                                                         __SEGGER_RTL_U128 value,
                                                  int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr after the update.

5.3.47 __atomic_fetch_add_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_fetch_add_1(volatile void * ptr,
                                                     __SEGGER_RTL_U8 value,
                                              int    memorder);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.
`memorder`	Memory ordering.

Return value

Value of object pointed to by `ptr` prior to the update.

5.3.48 __atomic_fetch_add_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_fetch_add_2(volatile void * ptr,
                                                      __SEGGER_RTL_U16 value,
                                               int    memorder);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.
`memorder`	Memory ordering.

Return value

Value of object pointed to by `ptr` prior to the update.

5.3.49 __atomic_fetch_add_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_fetch_add_4(volatile void * ptr,
                                                 __SEGGER_RTL_U32 value,
                                                 int     memorder);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.
`memorder`	Memory ordering.

Return value

Value of object pointed to by `ptr` prior to the update.

5.3.50 __atomic_fetch_add_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_fetch_add_8(volatile void * ptr,
                                                      __SEGGER_RTL_U64 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.51 __atomic_fetch_add_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_fetch_add_16(volatile void * ptr,
                                                         __SEGGER_RTL_U128 value,
                                                  int     memorder);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.
`memorder`	Memory ordering.

Return value

Value of object pointed to by `ptr` prior to the update.

5.3.52 __atomic_fetch_sub_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_fetch_sub_1(volatile void * ptr,
                                                    __SEGGER_RTL_U8 value,
                                             int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.53 __atomic_fetch_sub_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_fetch_sub_2(volatile void * ptr,
                                                       __SEGGER_RTL_U16 value,
                                                int    memorder);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.
`memorder`	Memory ordering.

Return value

Value of object pointed to by `ptr` prior to the update.

5.3.54 __atomic_fetch_sub_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_fetch_sub_4(volatile void * ptr,
                                                       __SEGGER_RTL_U32 value,
                                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.55 __atomic_fetch_sub_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_fetch_sub_8(volatile void * ptr,
                                                       __SEGGER_RTL_U64 value,
                                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.56 __atomic_fetch_sub_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_fetch_sub_16(volatile void * ptr,
                                                        __SEGGER_RTL_U128 value,
                                                 int    memorder);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.
`memorder`	Memory ordering.

Return value

Value of object pointed to by `ptr` prior to the update.

5.3.57 __atomic_fetch_and_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_fetch_and_1(volatile void * ptr,
                                                     __SEGGER_RTL_U8 value,
                                                     int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.58 __atomic_fetch_and_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_fetch_and_2(volatile void * ptr,
                                                      __SEGGER_RTL_U16 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.59 __atomic_fetch_and_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_fetch_and_4(volatile void * ptr,
                                                      __SEGGER_RTL_U32 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.60 __atomic_fetch_and_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_fetch_and_8(volatile void * ptr,
                                                 __SEGGER_RTL_U64 value,
                                             int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.61 __atomic_fetch_and_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_fetch_and_16(volatile void * ptr,
                                                  __SEGGER_RTL_U128 value,
                                        int       memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.62 __atomic_fetch_or_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_fetch_or_1(volatile void * ptr,
                                                   __SEGGER_RTL_U8 value,
                                                   int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.63 __atomic_fetch_or_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_fetch_or_2(volatile void * ptr,
                                                  __SEGGER_RTL_U16 value,
                                             int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.64 __atomic_fetch_or_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_fetch_or_4(volatile void * ptr,
                                                     __SEGGER_RTL_U32 value,
                                              int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.65 __atomic_fetch_or_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_fetch_or_8(volatile void * ptr,
                                                 __SEGGER_RTL_U64 value,
                                             int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.66 __atomic_fetch_or_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_fetch_or_16(volatile void * ptr,
                                                       __SEGGER_RTL_U128 value,
                                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.67 __atomic_fetch_xor_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_fetch_xor_1(volatile void * ptr,
                                                     __SEGGER_RTL_U8 value,
                                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.68 __atomic_fetch_xor_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_fetch_xor_2(volatile void * ptr,
                                                __SEGGER_RTL_U16 value,
                                      int              memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.69 __atomic_fetch_xor_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_fetch_xor_4(volatile void * ptr,
                                                       __SEGGER_RTL_U32 value,
                                               int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.70 __atomic_fetch_xor_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_fetch_xor_8(volatile void * ptr,
                                                        __SEGGER_RTL_U64 value,
                                               int      memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.71 __atomic_fetch_xor_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_fetch_xor_16(volatile void * ptr,
                                                __SEGGER_RTL_U128 value,
                                        int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.72 __atomic_fetch_nand_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __atomic_fetch_nand_1(volatile void * ptr,
                                                      __SEGGER_RTL_U8 value,
                                               int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.73 __atomic_fetch_nand_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __atomic_fetch_nand_2(volatile void * ptr,
                                                       __SEGGER_RTL_U16 value,
                                                int    memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.74 __atomic_fetch_nand_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __atomic_fetch_nand_4(volatile void * ptr,
                                                          __SEGGER_RTL_U32 value,
                                                int       memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.75 __atomic_fetch_nand_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __atomic_fetch_nand_8(volatile void * ptr,
                                                   __SEGGER_RTL_U64 value,
                                                int      memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.76 __atomic_fetch_nand_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __atomic_fetch_nand_16(volatile void * ptr,
                                                         __SEGGER_RTL_U128 value,
                                                         int     memorder);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.
memorder	Memory ordering.

Return value

Value of object pointed to by ptr prior to the update.

5.3.76.1 __sync_synchronize()

Description

Issue a full memory barrier.

Prototype

```
void __sync_synchronize(void);
```

5.3.76.2 __sync_fetch_and_add_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_fetch_and_add_1(volatile void * ptr,
                                                      __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.3 __sync_fetch_and_add_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_fetch_and_add_2(volatile void * ptr,
                                        __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.4 __sync_fetch_and_add_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_fetch_and_add_4(volatile void * ptr,
                                                      __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.5 __sync_fetch_and_add_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_fetch_and_add_8(volatile void * ptr,
                                        __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.6 __sync_fetch_and_add_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_fetch_and_add_16(volatile void * ptr,
                                          __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.7 __sync_fetch_and_sub_1()

Description

Atomic fetch with decrement, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_fetch_and_sub_1(volatile void * ptr,
                                                        __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

© 2003-2022 SEGGER Microcontroller GmbH

5.3.76.8 __sync_fetch_and_sub_2()

Description

Atomic fetch with decrement, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_fetch_and_sub_2(volatile void * ptr,
                                        __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.9 __sync_fetch_and_sub_4()

Description

Atomic fetch with decrement, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_fetch_and_sub_4(volatile void * ptr,
                                                      __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.10 __sync_fetch_and_sub_8()

Description

Atomic fetch with decrement, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_fetch_and_sub_8(volatile void * ptr,
                                         __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.11 __sync_fetch_and_sub_16()

Description

Atomic fetch with decrement, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_fetch_and_sub_16(volatile void * ptr,
                                                          __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.12 __sync_fetch_and_and_1()

Description

Atomic fetch with bitwise-and, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_fetch_and_and_1(volatile void * ptr,
                                                   __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.13 __sync_fetch_and_and_2()

Description

Atomic fetch with bitwise-and, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_fetch_and_and_2(volatile void * ptr,
                                         __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.14 __sync_fetch_and_and_4()

Description

Atomic fetch with bitwise-and, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_fetch_and_and_4(volatile void * ptr,
                                                        __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.15 __sync_fetch_and_and_8()

Description

Atomic fetch with bitwise-and, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_fetch_and_and_8(volatile void * ptr,
                                                         __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.16 __sync_fetch_and_and_16()

Description

Atomic fetch with bitwise-and, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_fetch_and_and_16(volatile void * ptr,
                                          __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.17 __sync_fetch_and_or_1()

Description

Atomic fetch with bitwise-or, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_fetch_and_or_1(volatile void * ptr,
                                      __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.18 __sync_fetch_and_or_2()

Description

Atomic fetch with bitwise-or, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_fetch_and_or_2(volatile void * ptr,
                                            __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.19 __sync_fetch_and_or_4()

Description

Atomic fetch with bitwise-or, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_fetch_and_or_4(volatile void * ptr,
                                                       __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.20 __sync_fetch_and_or_8()

Description

Atomic fetch with bitwise-or, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_fetch_and_or_8(volatile void * ptr,
                                                  __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.21 __sync_fetch_and_or_16()

Description

Atomic fetch with bitwise-or, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_fetch_and_or_16(volatile void * ptr,
                                                         __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.22 __sync_fetch_and_xor_1()

Description

Atomic fetch with bitwise-exclusive-or, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_fetch_and_xor_1(volatile void * ptr,
                                                      __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.23 __sync_fetch_and_xor_2()

Description

Atomic fetch with bitwise-exclusive-or, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_fetch_and_xor_2(volatile void * ptr,
                                                        __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.24 __sync_fetch_and_xor_4()

Description

Atomic fetch with bitwise-exclusive-or, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_fetch_and_xor_4(volatile void * ptr,
                                         __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.25 __sync_fetch_and_xor_8()

Description

Atomic fetch with bitwise-exclusive-or, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_fetch_and_xor_8(volatile void * ptr,
                                        __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.26 __sync_fetch_and_xor_16()

Description

Atomic fetch with bitwise-exclusive-or, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_fetch_and_xor_16(volatile void * ptr,
                                                          __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.27 __sync_fetch_and_nand_1()

Description

Atomic fetch with bitwise-nand, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_fetch_and_nand_1(volatile void * ptr,
                                                        __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.28 __sync_fetch_and_nand_2()

Description

Atomic fetch with bitwise-nand, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_fetch_and_nand_2(volatile void * ptr,
                                                          __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.29 __sync_fetch_and_nand_4()

Description

Atomic fetch with bitwise-nand, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_fetch_and_nand_4(volatile void * ptr,
                                          __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.30 __sync_fetch_and_nand_8()

Description

Atomic fetch with bitwise-nand, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_fetch_and_nand_8(volatile void * ptr,
                                                         __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` prior to update.

5.3.76.31 __sync_fetch_and_nand_16()

Description

Atomic fetch with bitwise-nand, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_fetch_and_nand_16
                                         (volatile void * ptr,
                                                          __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr prior to update.

5.3.76.32 __sync_add_and_fetch_1()

Description

Atomic fetch with increment, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_add_and_fetch_1(volatile void * ptr,
                                                        __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr after the update.

5.3.76.33 __sync_add_and_fetch_2()

Description

Atomic fetch with increment, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_add_and_fetch_2(volatile void * ptr,
                                         __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr after the update.

5.3.76.34 __sync_add_and_fetch_4()

Description

Atomic fetch with increment, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_add_and_fetch_4(volatile void * ptr,
                                              __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.35 __sync_add_and_fetch_8()

Description

Atomic fetch with increment, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_add_and_fetch_8(volatile void * ptr,
                                                       __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.36 __sync_add_and_fetch_16()

Description

Atomic fetch with increment, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_add_and_fetch_16(volatile void * ptr,
                                          __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr after the update.

5.3.76.37 __sync_sub_and_fetch_1()

Description

Atomic fetch with decrement, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_sub_and_fetch_1(volatile void * ptr,
                                       __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr after the update.

5.3.76.38 __sync_sub_and_fetch_2()

Description

Atomic fetch with decrement, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_sub_and_fetch_2(volatile void * ptr,
                                        __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.39 __sync_sub_and_fetch_4()

Description

Atomic fetch with decrement, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_sub_and_fetch_4(volatile void * ptr,
                                         __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.40 __sync_sub_and_fetch_8()

Description

Atomic fetch with decrement, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_sub_and_fetch_8(volatile void * ptr,
                                                          __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.41 __sync_sub_and_fetch_16()

Description

Atomic fetch with decrement, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_sub_and_fetch_16(volatile void * ptr,
                                                         __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.42 __sync_and_and_fetch_1()

Description

Atomic bitwise-and and fetch, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_and_and_fetch_1(volatile void * ptr,
                                       __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.43 __sync_and_and_fetch_2()

Description

Atomic bitwise-and and fetch, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_and_and_fetch_2(volatile void * ptr,
                                        __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.44 __sync_and_and_fetch_4()

Description

Atomic bitwise-and and fetch, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_and_and_fetch_4(volatile void * ptr,
                                                        __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.45 __sync_and_and_fetch_8()

Description

Atomic bitwise-and and fetch, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_and_and_fetch_8(volatile void * ptr,
                                                        __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.46 __sync_and_and_fetch_16()

Description

Atomic bitwise-and and fetch, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_and_and_fetch_16(volatile void * ptr,
                                           __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.47 __sync_or_and_fetch_1()

Description

Atomic bitwise-or and fetch, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_or_and_fetch_1(volatile void * ptr,
                                                     __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.48 __sync_or_and_fetch_2()

Description

Atomic bitwise-or and fetch, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_or_and_fetch_2(volatile void * ptr,
                                       __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.49 __sync_or_and_fetch_4()

Description

Atomic bitwise-or and fetch, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_or_and_fetch_4(volatile void * ptr,
                                       __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.50 __sync_or_and_fetch_8()

Description

Atomic bitwise-or and fetch, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_or_and_fetch_8(volatile void * ptr,
                                       __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
value	Value to add to object.

Return value

Value of object pointed to by ptr after the update.

5.3.76.51 __sync_or_and_fetch_16()

Description

Atomic bitwise-or and fetch, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_or_and_fetch_16(volatile void * ptr,
                                                     __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.52 __sync_xor_and_fetch_1()

Description

Atomic bitwise-exclusive-or and fetch, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_xor_and_fetch_1(volatile void * ptr,
                                        __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.53 __sync_xor_and_fetch_2()

Description

Atomic bitwise-exclusive-or and fetch, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_xor_and_fetch_2(volatile void * ptr,
                                                        __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.54 __sync_xor_and_fetch_4()

Description

Atomic bitwise-exclusive-or and fetch, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_xor_and_fetch_4(volatile void * ptr,
                                                       __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.55 __sync_xor_and_fetch_8()

Description

Atomic bitwise-exclusive-or and fetch, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_xor_and_fetch_8(volatile void * ptr,
                                        __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.56 __sync_xor_and_fetch_16()

Description

Atomic bitwise-exclusive-or and fetch, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_xor_and_fetch_16(volatile void * ptr,
                                                          __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.57 __sync_nand_and_fetch_1()

Description

Atomic bitwise-nand and fetch, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_nand_and_fetch_1(volatile void * ptr,
                                          __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.58 __sync_nand_and_fetch_2()

Description

Atomic bitwise-nand and fetch, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_nand_and_fetch_2(volatile void * ptr,
                                          __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.59 __sync_nand_and_fetch_4()

Description

Atomic bitwise-nand and fetch, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_nand_and_fetch_4(volatile void * ptr,
                                                          __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.60 __sync_nand_and_fetch_8()

Description

Atomic bitwise-nand and fetch, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_nand_and_fetch_8(volatile void * ptr,
                                                         __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.61 __sync_nand_and_fetch_16()

Description

Atomic bitwise-nand and fetch, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_nand_and_fetch_16
                                       (volatile void * ptr,
                                                        __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`value`	Value to add to object.

Return value

Value of object pointed to by `ptr` after the update.

5.3.76.62 __sync_bool_compare_and_swap_1()

Description

Atomic compare and swap, 8-bit object.

Prototype

```
__SEGGER_RTL_BOOL __sync_bool_compare_and_swap_1
                                          (volatile void * ptr,
                                                           __SEGGER_RTL_U8 oldval,
                                                           __SEGGER_RTL_U8 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

5.3.76.63 __sync_bool_compare_and_swap_2()

Description

Atomic compare and swap, 16-bit object.

Prototype

```
__SEGGER_RTL_BOOL __sync_bool_compare_and_swap_2
                                     (volatile void * ptr,
                                                      __SEGGER_RTL_U16 oldval,
                                                      __SEGGER_RTL_U16 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

5.3.76.64 __sync_bool_compare_and_swap_4()

Description

Atomic compare and swap, 32-bit object.

Prototype

```
__SEGGER_RTL_BOOL __sync_bool_compare_and_swap_4
                                        (volatile void * ptr,
                                                     __SEGGER_RTL_U32 oldval,
                                                     __SEGGER_RTL_U32 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

© 2003-2022 SEGGER Microcontroller GmbH

5.3.76.65 __sync_bool_compare_and_swap_8()

Description

Atomic compare and swap, 64-bit object.

Prototype

```
__SEGGER_RTL_BOOL __sync_bool_compare_and_swap_8
                                         (volatile void * ptr,
                                                      __SEGGER_RTL_U64 oldval,
                                                      __SEGGER_RTL_U64 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

5.3.76.66 __sync_bool_compare_and_swap_16()

Description

Atomic compare and swap, 128-bit object.

Prototype

```
__SEGGER_RTL_BOOL __sync_bool_compare_and_swap_16
                                     (volatile void * ptr,
                                                      __SEGGER_RTL_U128 oldval,
                                                      __SEGGER_RTL_U128 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

5.3.76.67 __sync_val_compare_and_swap_1()

Description

Atomic compare and swap, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_val_compare_and_swap_1
                                          (volatile void * ptr,
                                                           __SEGGER_RTL_U8 oldval,
                                                           __SEGGER_RTL_U8 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

5.3.76.68 __sync_val_compare_and_swap_2()

Description

Atomic compare and swap, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_val_compare_and_swap_2
                                          (volatile void * ptr,
                                                           __SEGGER_RTL_U16 oldval,
                                                           __SEGGER_RTL_U16 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

5.3.76.69 __sync_val_compare_and_swap_4()

Description

Atomic compare and swap, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_val_compare_and_swap_4
                                         (volatile void * ptr,
                                                          __SEGGER_RTL_U32 oldval,
                                                          __SEGGER_RTL_U32 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

5.3.76.70 __sync_val_compare_and_swap_8()

Description

Atomic compare and swap, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_val_compare_and_swap_8
                                         (volatile void * ptr,
                                                          __SEGGER_RTL_U64 oldval,
                                                          __SEGGER_RTL_U64 newval);
```

Parameters

Parameter	Description
ptr	Pointer to object to update.
oldval	Expected value of object on entry.
newval	Value to attempt to write to object.

Return value

Nonzero if and only if newval was written to the object pointed to by ptr.

5.3.76.71 __sync_val_compare_and_swap_16()

Description

Atomic compare and swap, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_val_compare_and_swap_16
                                        (volatile void * ptr,
                                                         __SEGGER_RTL_U128 oldval,
                                                         __SEGGER_RTL_U128 newval);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to update.
`oldval`	Expected value of object on entry.
`newval`	Value to attempt to write to object.

Return value

Nonzero if and only if `newval` was written to the object pointed to by ptr.

5.3.76.72 __sync_lock_test_and_set_1()

Description

Acquire lock, 8-bit object.

Prototype

```
__SEGGER_RTL_U8 __sync_lock_test_and_set_1(volatile void * ptr,
                                           __SEGGER_RTL_U8 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to be locked.
`value`	Value to be written to object.

Return value

The `value` of the object pointed to by `ptr` prior to update.

5.3.76.73 __sync_lock_test_and_set_2()

Description

Acquire lock, 16-bit object.

Prototype

```
__SEGGER_RTL_U16 __sync_lock_test_and_set_2
                                          (volatile void * ptr,
                                                           __SEGGER_RTL_U16 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to be locked.
`value`	Value to be written to object.

Return value

The `value` of the object pointed to by `ptr` prior to update.

5.3.76.74 __sync_lock_test_and_set_4()

Description

Acquire lock, 32-bit object.

Prototype

```
__SEGGER_RTL_U32 __sync_lock_test_and_set_4
                                          (volatile void * ptr,
                                                           __SEGGER_RTL_U32 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to be locked.
value	Value to be written to object.

Return value

The value of the object pointed to by ptr prior to update.

5.3.76.75 __sync_lock_test_and_set_8()

Description

Acquire lock, 64-bit object.

Prototype

```
__SEGGER_RTL_U64 __sync_lock_test_and_set_8
                                          (volatile void * ptr,
                                                           __SEGGER_RTL_U64 value);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to be locked.
`value`	Value to be written to object.

Return value

The `value` of the object pointed to by `ptr` prior to update.

5.3.76.76 __sync_lock_test_and_set_16()

Description

Acquire lock, 128-bit object.

Prototype

```
__SEGGER_RTL_U128 __sync_lock_test_and_set_16
                                        (volatile void * ptr,
                                                         __SEGGER_RTL_U128 value);
```

Parameters

Parameter	Description
ptr	Pointer to object to be locked.
value	Value to be written to object.

Return value

The value of the object pointed to by ptr prior to update.

5.3.76.77 __sync_lock_release_1()

Description

Release lock, 8-bit object.

Prototype

```
void __sync_lock_release_1(volatile void * ptr);
```

Parameters

Parameter	Description
ptr	Pointer to object to unlock.

5.3.76.78 __sync_lock_release_2()

Description

Release lock, 16-bit object.

Prototype

```
void __sync_lock_release_2(volatile void * ptr);
```

Parameters

Parameter	Description
ptr	Pointer to object to unlock.

5.3.76.79 __sync_lock_release_4()

Description

Release lock, 32-bit object.

Prototype

```
void __sync_lock_release_4(volatile void * ptr);
```

Parameters

Parameter	Description
`ptr`	Pointer to object to unlock.

5.3.76.80 __sync_lock_release_8()

Description

Release lock, 64-bit object.

Prototype

```
void __sync_lock_release_8(volatile void * ptr);
```

Parameters

Parameter	Description
ptr	Pointer to object to unlock.

5.3.76.81 __sync_lock_release_16()

Description

Release lock, 128-bit object.

Prototype

```
void __sync_lock_release_16(volatile void * ptr);
```

Parameters

Parameter	Description
ptr	Pointer to object to unlock.

Chapter 6

External function interface

This section summarises the functions to be provided by the implementor when integrating emRun into an application or library.

6.1 I/O functions

Function	Description
`__SEGGER_RTL_X_file_open`	Open file.
`__SEGGER_RTL_X_file_error`	Test for file-error condition.
`__SEGGER_RTL_X_file_end`	Test for end-of-file condition.
`__SEGGER_RTL_X_file_stat`	Get file status.
`__SEGGER_RTL_X_file_bufsize`	Get stream buffer size.
`__SEGGER_RTL_X_file_flush`	Flush unwritten data to file.
`__SEGGER_RTL_X_file_getpos`	Get file position.
`__SEGGER_RTL_X_file_seek`	Set file position.
`__SEGGER_RTL_X_file_clrerr`	Clear file-error status.
`__SEGGER_RTL_X_file_close`	Close file.
`__SEGGER_RTL_X_file_read`	Read from file.
`__SEGGER_RTL_X_file_write`	Read from file.
`__SEGGER_RTL_X_file_rename`	Rename file.
`__SEGGER_RTL_X_file_remove`	Remove file.
`__SEGGER_RTL_X_file_tmpnam`	Generate name for temporary file.
`__SEGGER_RTL_X_file_tmpfile`	Generate temporary file.
`__SEGGER_RTL_X_file_unget`	Push character back to file.

6.1.1 __SEGGER_RTL_X_file_open()

Description

Open file.

Prototype

```
__SEGGER_RTL_FILE *__SEGGER_RTL_X_file_open(const char * filename,
                                            const char * mode);
```

Parameters

Parameter	Description
filename	Pointer to zero-terminated file name.
mode	Pointer to zero-terminated file mode.

Return value

= NULL	File not opened.
≠ NULL	File opened.

6.1.2 __SEGGER_RTL_X_file_error()

Description

Test for file-error condition.

Prototype

```
int __SEGGER_RTL_X_file_error(__SEGGER_RTL_FILE *stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file.

Return value

< 0	Failure, `stream` is closed.
= 0	Success, `stream` is not in error.
> 0	Success, `stream` is in error.

6.1.3 __SEGGER_RTL_X_file_end()

Description

Test for end-of-file condition.

Prototype

```
int __SEGGER_RTL_X_file_end(__SEGGER_RTL_FILE *stream);
```

Parameters

Parameter	Description
stream	Pointer to file.

Return value

< 0	Failure, stream is closed.
= 0	Success, stream is not at end of file.
> 0	Success, stream is at end of file.

6.1.4 __SEGGER_RTL_X_file_stat()

Description

Get file status.

Prototype

```
int __SEGGER_RTL_X_file_stat(__SEGGER_RTL_FILE *stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file.

Return value

< 0	Failure, `stream` is not a valid file.
≥ 0	Success, `stream` is a valid file.

Additional information

Low-overhead test to determine if `stream` is valid. If `stream` is a valid pointer and the `stream` is open, this function must succeed. If `stream` is a valid pointer and the `stream` is closed, this function must fail.

The implementation may optionally determine whether `stream` is a valid pointer: this may not always be possible and is not required, but may assist debugging when clients provide wild pointers.

6.1.5 __SEGGER_RTL_X_file_bufsize()

Description

Get stream buffer size.

Prototype

```
int __SEGGER_RTL_X_file_bufsize(__SEGGER_RTL_FILE *stream);
```

Parameters

Parameter	Description
stream	Pointer to file.

Return value

Nonzero number of characters to use for buffered I/O; for unbuffered I/O, return 1.

Additional information

Returns the number of characters to use for buffered I/O on the file stream. The I/O buffer is allocated on the stack for the duration of the I/O call, therefore this value should not be set arbitrarily large.

For unbuffered I/O, return 1.

6.1.6 __SEGGER_RTL_X_file_flush()

Description

Flush unwritten data to file.

Prototype

```
int __SEGGER_RTL_X_file_flush(__SEGGER_RTL_FILE *stream);
```

Parameters

Parameter	Description
stream	Pointer to file.

Return value

< 0	Failure, file cannot be flushed or was not successfully flushed.
= 0	Success, unwritten data is flushed.

6.1.7 __SEGGER_RTL_X_file_getpos()

Description

Get file position.

Prototype

```
int __SEGGER_RTL_X_file_getpos(         __SEGGER_RTL_FILE *stream,
                               fpos_t * pos);
```

Parameters

Parameter	Description
`stream`	Pointer to file.
`pos`	Pointer to object that receives the position.

Return value

= 0	Position retrieved successfully.
< 0	Position not retrieved successfully.

6.1.8 __SEGGER_RTL_X_file_seek()

Description

Set file position.

Prototype

```
int __SEGGER_RTL_X_file_seek(     __SEGGER_RTL_FILE *stream,
                              long offset,
                              int  whence);
```

Parameters

Parameter	Description
stream	Pointer to file to position.
offset	Offset relative to anchor specified by whence.
whence	Where offset is relative to.

Return value

= 0	Position is set.
≠ 0	Position is not set.

6.1.9 __SEGGER_RTL_X_file_clrerr()

Description

Clear file-error status.

Prototype

```
void __SEGGER_RTL_X_file_clrerr(__SEGGER_RTL_FILE *stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file.

6.1.10 __SEGGER_RTL_X_file_close()

Description

Close file.

Prototype

```
int __SEGGER_RTL_X_file_close(__SEGGER_RTL_FILE *stream);
```

Parameters

Parameter	Description
`stream`	Pointer to file.

Return value

< 0	Failure, `stream` is already closed.
≥ 0	Success, `stream` is closed.

Additional information

Close the file `stream`. If the `stream` is connected to a temporary file (by use of `tmpfile()`), the temporary file is deleted.

6.1.11 __SEGGER_RTL_X_file_read()

Description

Read from file.

Prototype

```
int __SEGGER_RTL_X_file_read(           __SEGGER_RTL_FILE *stream,
                             char      * s,
                             unsigned    len);
```

Parameters

Parameter	Description
`stream`	Pointer to file to read from.
`s`	Pointer to object to write to.
`len`	Number of characters to read.

Return value

The number of characters successfully read, which may be less than `len` if a read error or end-of-file is encountered.

6.1.12 __SEGGER_RTL_X_file_write()

Description

Read from file.

Prototype

```
int __SEGGER_RTL_X_file_write(                  __SEGGER_RTL_FILE *stream,
                                      const char     * s,
                                            unsigned   len);
```

Parameters

Parameter	Description
stream	Pointer to file to read from.
s	Pointer to object to write.
len	Number of characters to write.

Return value

The number of characters successfully written, which may be less than len if an error occurs.

6.1.13 __SEGGER_RTL_X_file_rename()

Description

Rename file.

Prototype

```
int __SEGGER_RTL_X_file_rename(const char * oldname,
                               const char * newname);
```

Parameters

Parameter	Description
oldname	Pointer to string denoting old file name.
newname	Pointer to string denoting new file name.

Return value

= 0	Rename succeeded.
≠ 0	Rename failed.

6.1.14 __SEGGER_RTL_X_file_remove()

Description

Remove file.

Prototype

```
int __SEGGER_RTL_X_file_remove(const char * filename);
```

Parameters

Parameter	Description
filename	Pointer to string denoting file name to remove.

Return value

= 0	Remove succeeded.
≠ 0	Remove failed.

6.1.15 __SEGGER_RTL_X_file_tmpnam()

Description

Generate name for temporary file.

Prototype

```
char *__SEGGER_RTL_X_file_tmpnam(char     * s,
                                 unsigned   max);
```

Parameters

Parameter	Description
s	Pointer to object that receives the temporary file name, or NULL indicating that a (shared) internal buffer is used for the temporary name.
max	Maxumum number of characters acceptable in the object s.

Return value

= NULL	Cannot generate a unique temporary name.
≠ NULL	Pointer to temporary name generated.

6.1.16 __SEGGER_RTL_X_file_tmpfile()

Description

Generate temporary file.

Prototype

```
__SEGGER_RTL_FILE *__SEGGER_RTL_X_file_tmpfile(void);
```

Return value

= `NULL`	Cannot generate a unique temporary file.
≠ `NULL`	Pointer to temporary file.

6.1.17 __SEGGER_RTL_X_file_unget()

Description

Push character back to file.

Prototype

```
int __SEGGER_RTL_X_file_unget(    __SEGGER_RTL_FILE *stream,
                                  int c);
```

Parameters

Parameter	Description
stream	File to push character to.
c	Character to push back to file.

Return value

= EOF	Failed to push character back.
≠ EOF	The character pushed back to the file.

Additional information

This function pushes the character c back to the file stream so that it can be read again. If c is EOF, the function fails and EOF is returned. One character of pushback is guaranteed; if more than one character is pushed back without an intervening read, the pushback may fail.

6.2 Heap functions

Function	Description
`__SEGGER_RTL_init_heap`	Initializes the heap.
`__SEGGER_RTL_X_heap_lock`	Lock heap.
`__SEGGER_RTL_X_heap_unlock`	Unlock heap.

6.2.1 __SEGGER_RTL_init_heap()

Description

Initializes the heap.

Prototype

```
void __SEGGER_RTL_init_heap(void * ptr,
                                   size_t size);
```

Parameters

Parameter	Description
ptr	Pointer to correctly-aligned heap memory to manage.
size	Size of managed area in bytes.

6.2.2 __SEGGER_RTL_X_heap_lock()

Description

Lock heap.

Prototype

```
void __SEGGER_RTL_X_heap_lock(void);
```

Additional information

This function is called to lock access to the heap before allocation or deallocation is processed. This is only required for multitasking systems where heap operations may possibly be called called from different threads.

6.2.3 __SEGGER_RTL_X_heap_unlock()

Description

Unlock heap.

Prototype

```
void __SEGGER_RTL_X_heap_unlock(void);
```

Additional information

This function is called to unlock access to the heap after allocation or deallocation has completed. This is only required for multitasking systems where heap operations may possibly be called called from different threads.

6.3 Atomic functions

Function	Description
`__SEGGER_RTL_X_atomic_lock`	Lock for atomic access.
`__SEGGER_RTL_X_atomic_unlock`	Unlock atomic access.
`__SEGGER_RTL_X_atomic_synchronize`	Full memory barrier.

6.3.1 __SEGGER_RTL_X_atomic_lock()

Description

Lock for atomic access.

Prototype

```
int __SEGGER_RTL_X_atomic_lock(void);
```

Return value

User-defined value to be passed to matching atomic unlock.

Additional information

This function is called to provide exclusive access to an object. Typically, interrupts are disabled by the function, but in a multi-tasking system a mutex can be used.

6.3.2 __SEGGER_RTL_X_atomic_unlock()

Description

Unlock atomic access.

Prototype

```
void __SEGGER_RTL_X_atomic_unlock(int state);
```

Parameters

Parameter	Description
state	Value returned by __SEGGER_RTL_X_atomic_lock().

Additional information

This function is called to relinquish previously-locked atomic access.

6.3.3 __SEGGER_RTL_X_atomic_synchronize()

Description

Full memory barrier.

Prototype

```
void __SEGGER_RTL_X_atomic_synchronize(void);
```

Additional information

This function is called to ensure all memory reads and writes are complete before continuing.

6.4 Error and assertion functions

Function	Description
`__SEGGER_RTL_X_assert`	User-defined behavior for the assert macro.
`__SEGGER_RTL_X_errno_addr`	Return pointer to object holding errno.

6.4.1 __SEGGER_RTL_X_assert()

Description

User-defined behavior for the assert macro.

Prototype

```
void __SEGGER_RTL_X_assert(const char * expr,
                           const char * filename,
                                 int    line);
```

Parameters

Parameter	Description
expr	Stringized expression that caused failure.
filename	Filename of the source file where the failure was signaled.
line	Line number of the failed assertion.

Additional information

The default implementation of `__SEGGER_RTL_X_assert()` prints the `filename`, `line`, and error message to standard output and then calls `abort()`.

`__SEGGER_RTL_X_assert()` is defined as a weak function and can be replaced by user code.

6.4.2 __SEGGER_RTL_X_errno_addr()

Description

Return pointer to object holding errno.

Prototype

```
int *__SEGGER_RTL_X_errno_addr(void);
```

Return value

Pointer to errno object.

Additional information

The default implementation of this function is to return the address of a variable declared with the `__SEGGER_RTL_STATE_THREAD` storage class. Thus, for multithreaded environments that implement thread-local variables through `__SEGGER_RTL_STATE_THREAD`, each thread receives its own thread-local errno.

It is beyond the scope of this manual to describe how thread-local variables are implemented by the compiler and any associated real-time operating system.

When `__SEGGER_RTL_STATE_THREAD` is defined as an empty macro, this function returns the address of a singleton errno object.

Chapter 7

Appendices

7.1 Benchmarking performance

The following benchmarks of the low-level floating-point arithmetic functions show expected performance of each archttecture, measured on the same device. For the RISC-V benchmarks, the device is a TLS9518A executing from instruction-local memory.

7.1.1 RV32I benchmarks

```
IEEE-754 Floating-point Library Benchmarks
Copyright (c) 2018-2022 SEGGER Microcontroller GmbH.

System: emRun v2.26.0
Target: RV32I
Target: Little-endian byte order
Config: SEGGER_RTL_OPTIMIZE = 2
Config: SEGGER_RTL_FP_HW    = 0   // No FPU, software floating point
Config: SEGGER_RTL_FP_ABI   = 0   // Floats and doubles in core registers
Config: With assembly-coded acceleration
Config: With fully conformant NaNs

======================
    GNU libgcc API
======================

Function            Min     Max     Avg    Description
--------------   ------  ------  ------    -------------------------------
__addsf3             37      50    43.1    Random distribution over (0, 1), operands differ
__subsf3             31      58    48.9    Random distribution over (0, 1), operands differ
__mulsf3            238     324   269.3    Random distribution over (0, 1), operands differ
__divsf3            149     222   186.7    Random distribution over (0, 1), operands differ
__ltsf2               9      13     9.8    Random distribution over (0, 1), operands differ
__lesf2               8      12     9.6    Random distribution over (0, 1), operands differ
__gtsf2               9      12    10.2    Random distribution over (0, 1), operands differ
__gesf2              10      14    11.7    Random distribution over (0, 1), operands differ
__eqsf2               9      12    10.1    Random distribution over (0, 1), operands differ
__nesf2               9      12    10.1    Random distribution over (0, 1), operands differ
__adddf3             43      70    56.0    Random distribution over (0, 1), operands differ
__subdf3             51     110    74.3    Random distribution over (0, 1), operands differ
__muldf3            877    1010   936.8    Random distribution over (0, 1), operands differ
__divdf3            707     991   898.8    Random distribution over (0, 1), operands differ
__ltdf2              11      17    13.4    Random distribution over (0, 1), operands differ
__ledf2              12      19    14.1    Random distribution over (0, 1), operands differ
__gtdf2              11      17    13.7    Random distribution over (0, 1), operands differ
__gedf2              12      18    14.4    Random distribution over (0, 1), operands differ
__eqdf2              13      16    14.3    Random distribution over (0, 1), operands differ
__eqdf2              13      16    14.3    Random distribution over (0, 1), operands differ
__fixsfsi             4      16     9.2    Random distribution over (-2^31.., 1..2^31)
__fixunssfsi          2      12     5.2    Random distribution over (1..2^31)
__fixsfdi             5      22    18.5    Random distribution over (-2^63..1, 1..2^63)
__fixunssfdi          5      19    15.5    Random distribution over (-2^63..2^63)
__floatsisf          27      36    31.2    Random distribution over (-2^31.., 1..2^31)
__floatunsisf        20      28    24.3    Random distribution over (1..2^31)
__floatdisf          28      51    35.3    Random distribution over (-2^63..1, 1..2^63)
__floatundisf        23      43    31.0    Random distribution over (-2^63..2^63)
__fixdfsi             3      16     8.2    Random distribution over (-2^31.., 1..2^31)
__fixunsdfsi          3       8     4.9    Random distribution over (1..2^31)
__fixdfdi             6      29    23.7    Random distribution over (-2^63..1, 1..2^63)
__fixunsdfdi          6      23    19.3    Random distribution over (-2^63..2^63)
__floatsidf          20      28    23.2    Random distribution over (-2^31.., 1..2^31)
__floatunsidf        15      23    19.1    Random distribution over (1..2^31)
__floatdidf          22      55    37.3    Random distribution over (-2^63..1, 1..2^63)
__floatundidf        21      47    31.4    Random distribution over (-2^63..2^63)
__extendsfdf2         8      12     9.9    Random distribution over (-2^63..1, 1..2^63)
__truncdfsf2          4      25    21.9    Random distribution over (-2^63..1, 1..2^63)

Total cycles: 10924741
```

7.1.2 RV32IMC benchmarks

```
IEEE-754 Floating-point Library Benchmarks
Copyright (c) 2018-2022 SEGGER Microcontroller GmbH.

System: emRun v2.26.0
Target: RV32IMC
Target: Little-endian byte order
Config: SEGGER_RTL_OPTIMIZE = 2
Config: SEGGER_RTL_FP_HW    = 0   // No FPU, software floating point
Config: SEGGER_RTL_FP_ABI   = 0   // Floats and doubles in core registers
Config: With assembly-coded acceleration
Config: With fully conformant NaNs

======================
    GNU libgcc API
======================

Function            Min     Max     Avg    Description
--------------   ------  ------  ------    -------------------------------
__addsf3             36      46    40.6    Random distribution over (0, 1), operands differ
__subsf3             29      66    49.8    Random distribution over (0, 1), operands differ
__mulsf3             30      38    33.1    Random distribution over (0, 1), operands differ
__divsf3             64      66    64.5    Random distribution over (0, 1), operands differ
__ltsf2               7       9     8.4    Random distribution over (0, 1), operands differ
__lesf2               7       8     7.0    Random distribution over (0, 1), operands differ
__gtsf2               6       8     6.8    Random distribution over (0, 1), operands differ
__gesf2               7       9     8.4    Random distribution over (0, 1), operands differ
__eqsf2               6       8     6.7    Random distribution over (0, 1), operands differ
__nesf2               6       8     6.7    Random distribution over (0, 1), operands differ
__adddf3             43      67    55.0    Random distribution over (0, 1), operands differ
__subdf3             47     105    72.4    Random distribution over (0, 1), operands differ
__muldf3             64      79    69.8    Random distribution over (0, 1), operands differ
__divdf3            194     201   197.7    Random distribution over (0, 1), operands differ
__ltdf2               9      13     9.8    Random distribution over (0, 1), operands differ
__ledf2               9      14    10.5    Random distribution over (0, 1), operands differ
__gtdf2              10      15    11.2    Random distribution over (0, 1), operands differ
__gedf2              10      14    10.9    Random distribution over (0, 1), operands differ
__eqdf2              10      12    10.7    Random distribution over (0, 1), operands differ
__eqdf2              10      12    11.5    Random distribution over (0, 1), operands differ
__fixsfsi             1      13     6.3    Random distribution over (-2^31.., 1..2^31)
__fixunssfsi          1       8     3.3    Random distribution over (1..2^31)
__fixsfdi             3      18    15.5    Random distribution over (-2^63..1, 1..2^63)
__fixunssfdi          3      14    11.3    Random distribution over (-2^63..2^63)
__floatsisf          24      33    28.6    Random distribution over (-2^31.., 1..2^31)
__floatunsisf        20      26    22.7    Random distribution over (1..2^31)
__floatdisf          32      52    37.7    Random distribution over (-2^63..1, 1..2^63)
__floatundisf        27      45    33.0    Random distribution over (-2^63..2^63)
__fixdfsi             2      17     6.6    Random distribution over (-2^31.., 1..2^31)
__fixunsdfsi          2       7     3.6    Random distribution over (1..2^31)
__fixdfdi             6      28    22.2    Random distribution over (-2^63..1, 1..2^63)
__fixunsdfdi          5      22    18.2    Random distribution over (-2^63..2^63)
__floatsidf          19      25    21.8    Random distribution over (-2^31.., 1..2^31)
__floatunsidf        12      20    15.8    Random distribution over (1..2^31)
__floatdidf          23      52    37.8    Random distribution over (-2^63..1, 1..2^63)
__floatundidf        21      44    30.8    Random distribution over (-2^63..2^63)
__extendsfdf2        10      11    10.3    Random distribution over (-2^63..1, 1..2^63)
__truncdfsf2          6      26    23.7    Random distribution over (-2^63..1, 1..2^63)

Total cycles: 2862096
```

7.1.3 RV32IMCP

```
IEEE-754 Floating-point Library Benchmarks
Copyright (c) 2018-2022 SEGGER Microcontroller GmbH.

System: emRun v2.26.0
Target: RV32IMACP
Target: Little-endian byte order
Config: SEGGER_RTL_OPTIMIZE = 2
Config: SEGGER_RTL_FP_HW    = 0   // No FPU, software floating point
Config: SEGGER_RTL_FP_ABI   = 0   // Floats and doubles in core registers
Config: With assembly-coded acceleration
Config: With RISC-V SIMD acceleration
Config: With fully conformant NaNs

======================
    GNU libgcc API
======================

Function            Min     Max     Avg    Description
--------------   ------  ------  ------    ------------------------------
__addsf3             35      45    40.5    Random distribution over (0, 1), operands differ
__subsf3             29      51    37.4    Random distribution over (0, 1), operands differ
__mulsf3             29      37    32.8    Random distribution over (0, 1), operands differ
__divsf3             63      65    63.7    Random distribution over (0, 1), operands differ
__ltsf2               7      10     8.4    Random distribution over (0, 1), operands differ
__lesf2               6       7     6.5    Random distribution over (0, 1), operands differ
__gtsf2               6       7     6.5    Random distribution over (0, 1), operands differ
__gesf2               7       9     8.0    Random distribution over (0, 1), operands differ
__eqsf2               6       8     6.4    Random distribution over (0, 1), operands differ
__nesf2               6       8     6.4    Random distribution over (0, 1), operands differ
__adddf3             42      66    54.3    Random distribution over (0, 1), operands differ
__subdf3             48      87    64.7    Random distribution over (0, 1), operands differ
__muldf3             63      78    69.5    Random distribution over (0, 1), operands differ
__divdf3            193     198   197.0    Random distribution over (0, 1), operands differ
__ltdf2              10      13    11.4    Random distribution over (0, 1), operands differ
__ledf2              10      13    10.8    Random distribution over (0, 1), operands differ
__gtdf2              10      14    11.5    Random distribution over (0, 1), operands differ
__gedf2              10      14    11.9    Random distribution over (0, 1), operands differ
__eqdf2              11      13    11.5    Random distribution over (0, 1), operands differ
__eqdf2              11      13    11.8    Random distribution over (0, 1), operands differ
__fixsfsi             1      13     6.4    Random distribution over (-2^31.., 1..2^31)
__fixunssfsi          1       8     3.5    Random distribution over (1..2^31)
__fixsfdi             1      17    13.6    Random distribution over (-2^63..1, 1..2^63)
__fixunssfdi          4      16    11.8    Random distribution over (-2^63..2^63)
__floatsisf          15      19    16.4    Random distribution over (-2^31.., 1..2^31)
__floatunsisf         8      13     9.0    Random distribution over (1..2^31)
__floatdisf          19      28    23.4    Random distribution over (-2^63..1, 1..2^63)
__floatundisf        11      21    17.6    Random distribution over (-2^63..2^63)
__fixdfsi             3      17     7.4    Random distribution over (-2^31.., 1..2^31)
__fixunsdfsi          1       7     4.0    Random distribution over (1..2^31)
__fixdfdi             4      26    20.8    Random distribution over (-2^63..1, 1..2^63)
__fixunsdfdi          3      21    17.2    Random distribution over (-2^63..2^63)
__floatsidf           9      11     9.6    Random distribution over (-2^31.., 1..2^31)
__floatunsidf         3       4     3.7    Random distribution over (1..2^31)
__floatdidf          14      35    25.0    Random distribution over (-2^63..1, 1..2^63)
__floatundidf        11      28    19.9    Random distribution over (-2^63..2^63)
__extendsfdf2         5       7     6.0    Random distribution over (-2^63..1, 1..2^63)
__truncdfsf2          6      26    24.0    Random distribution over (-2^63..1, 1..2^63)

Total cycles: 2771332
```

7.1.4 RV32IMCP with Andes Performance Extensions

```
IEEE-754 Floating-point Library Benchmarks
Copyright (c) 2018-2022 SEGGER Microcontroller GmbH.

System: emRun v2.26.0
Target: RV32IMACP
Target: Little-endian byte order
Config: SEGGER_RTL_OPTIMIZE = 2
Config: SEGGER_RTL_FP_HW    = 0   // No FPU, software floating point
Config: SEGGER_RTL_FP_ABI   = 0   // Floats and doubles in core registers
Config: With assembly-coded acceleration
Config: With RISC-V SIMD acceleration
Config: With Andes V5 Performance Extension acceleration
Config: With fully conformant NaNs

======================
    GNU libgcc API
======================

Function            Min     Max     Avg    Description
--------------   ------  ------  ------    ------------------------------
__addsf3             32      41    36.8    Random distribution over (0, 1), operands differ
__subsf3             26      47    34.3    Random distribution over (0, 1), operands differ
__mulsf3             27      35    30.7    Random distribution over (0, 1), operands differ
__divsf3             57      58    58.0    Random distribution over (0, 1), operands differ
__ltsf2               8      10     8.5    Random distribution over (0, 1), operands differ
__lesf2               8       9     8.4    Random distribution over (0, 1), operands differ
__gtsf2               7      10     8.2    Random distribution over (0, 1), operands differ
__gesf2               8      10     8.5    Random distribution over (0, 1), operands differ
__eqsf2               7       8     7.5    Random distribution over (0, 1), operands differ
__nesf2               7       8     7.5    Random distribution over (0, 1), operands differ
__adddf3             39      63    51.6    Random distribution over (0, 1), operands differ
__subdf3             45      77    60.8    Random distribution over (0, 1), operands differ
__muldf3             64      76    69.2    Random distribution over (0, 1), operands differ
__divdf3            193     200   197.0    Random distribution over (0, 1), operands differ
__ltdf2              10      13    11.6    Random distribution over (0, 1), operands differ
__ledf2               9      12    10.7    Random distribution over (0, 1), operands differ
__gtdf2               9      14    11.4    Random distribution over (0, 1), operands differ
__gedf2              11      12    11.2    Random distribution over (0, 1), operands differ
__eqdf2              11      12    12.0    Random distribution over (0, 1), operands differ
__eqdf2              11      12    12.0    Random distribution over (0, 1), operands differ
__fixsfsi             2      13     6.7    Random distribution over (-2^31.., 1..2^31)
__fixunssfsi          1       8     3.9    Random distribution over (1..2^31)
__fixsfdi             2      17    14.8    Random distribution over (-2^63..1, 1..2^63)
__fixunssfdi          4      16    12.8    Random distribution over (-2^63..2^63)
__floatsisf          13      17    14.2    Random distribution over (-2^31.., 1..2^31)
__floatunsisf         7      11     8.0    Random distribution over (1..2^31)
__floatdisf          18      26    21.9    Random distribution over (-2^63..1, 1..2^63)
__floatundisf        10      19    15.7    Random distribution over (-2^63..2^63)
__fixdfsi             3      17     8.1    Random distribution over (-2^31.., 1..2^31)
__fixunsdfsi          2       9     4.4    Random distribution over (1..2^31)
__fixdfdi             3      26    20.5    Random distribution over (-2^63..1, 1..2^63)
__fixunsdfdi          4      22    17.7    Random distribution over (-2^63..2^63)
__floatsidf           9      11     9.8    Random distribution over (-2^31.., 1..2^31)
__floatunsidf         4       5     4.1    Random distribution over (1..2^31)
__floatdidf          14      32    23.7    Random distribution over (-2^63..1, 1..2^63)
__floatundidf        10      28    19.0    Random distribution over (-2^63..2^63)
__extendsfdf2         6       8     6.7    Random distribution over (-2^63..1, 1..2^63)
__truncdfsf2          6      22    20.0    Random distribution over (-2^63..1, 1..2^63)

Total cycles: 2707282
```

Chapter 8

Indexes

8.1 Index of types

8.2 Index of functions