GENERAL C LIBRARY DEFINITIONS

Reference Guide

Dinkum C Library Reference:
Limited Access Notice
Developed by P.J. Plauger
Copyright © 1992-1996
Dinkumware, Ltd.
398 Main Street
Concord MA 01742

COPYRIGHT AND LICENSE INFORMATION

The Dinkum C Library Reference presented here, in machine-readable or printed form (hereafter the Reference) is copyright © 1992-1996 by P.J. Plauger. ALL RIGHTS RESERVED.

This material is derived in part from books copyright © 1992-1996 by P.J. Plauger. Each copy of this Reference must be licensed by an authorized Licensee. This on-line copy of the Reference is for access only. You are not to copy it in whole or in part.

Dinkumware, Ltd. and P.J. Plauger retain exclusive ownership of the Reference.

You are entitled to access the on-line copy, but you may not make any copies for use by yourself or others.

You have a moral reponsibility not to aid or abet illegal copying by others.

The author recognizes that this PDF format is particularly conducive to sharing within multiuser sytems and across networks. The licensing for such use is available from Dinkumware, Ltd. The use of the on-line Reference is for access only. In particular, please note that the ability to access this Reference does not imply permission to copy it. Please note also that the author has expended considerable professional effort in the production of this Reference, and continues to do so to keep it current.

DINKUMWARE, LTD. AND P.J. PLAUGER MAKE NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE REFERENCE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. DINKUMWARE, LTD. AND P.J. PLAUGER SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF ACCESSING THIS REFERENCE.

By accessing this Reference, you agree to abide by the intellectual property laws, and all other applicable laws of the USA, and the terms of this Limited Access Notice. You may be held legally responsible for any infringement that is caused or encouraged by your failure to abide by the terms of this Notice.

Dinkumware, Ltd. retains the right to terminate access to this Reference immediately, and without notice.

ABOUT THIS GUIDE

This guide gives reference information on the standard C library functions, and summarizes them according to header file.

The following header files are provided:

Header file	*Usage*
`assert.h`	Enforcing assertions when functions execute
`ctype.h`	Classifying characters
`errno.h`	Testing error codes reported by library functions
`float.h`	Testing floating-point type properties
`iso646.h`	Alternative names for logical operators
`limits.h`	Testing integer type properties
`locale.h`	Adapting to different cultural conventions
`math.h`	Computing common mathematical functions
`setjmp.h`	Executing non-local goto statements
`signal.h`	Controlling various exceptional conditions
`stdarg.h`	Accessing a varying number of arguments
`stddef.h`	Defining several useful types and macros
`stdio.h`	Performing input and output
`stdlib.h`	Performing a variety of operations

Header file	*Usage*
`string.h`	Manipulating several kinds of strings
`time.h`	Converting between various time and date formats
`wchar.h`	Supporting wide characters
`wctype.h`	Classifying wide characters

CONVENTIONS

This guide uses the following typographical conventions:

Style	*Used for*
`computer`	Text that you type in, or that appears on the screen.
`parameter`	A label representing the actual value you should type as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
`...`	Multiple parameters can follow a command.
reference	A cross-reference to another part of this guide, or to another guide.

CONTENTS

ASSERT – assert.h

This chapter describes the standard header file `assert.h`.

DESCRIPTION

Include the standard header `assert.h` to define the macro `assert`, which is useful for diagnozing logic errors in the program. You can eliminate the test code produced by the macro `assert` without removing the macro references from the program by defining the macro `NDEBUG` in the program before you include `assert.h`. Each time the program includes this header, it redetermines the definition of the macro `assert`.

```
assert
#undef assert
#if defined NDEBUG
#define assert(test) (void)0
#else
#define assert(test) void_expression
#endif
```

If the `int` expression test equals zero, the macro writes to `stderr` a diagnostic message that includes:

- the text of `test`
- the source filename (the predefined macro `__FILE__`)
- the source line number (the predefined macro `__LINE__`)

It then calls `abort`.

You can write the macro `assert` in the program in any side-effects context.

SUMMARY

```
#undef assert
#if defined NDEBUG
#define assert(test) (void)0
#else
#define assert(test) void_expression
#endif
```

CHARACTER HANDLING – ctype.h

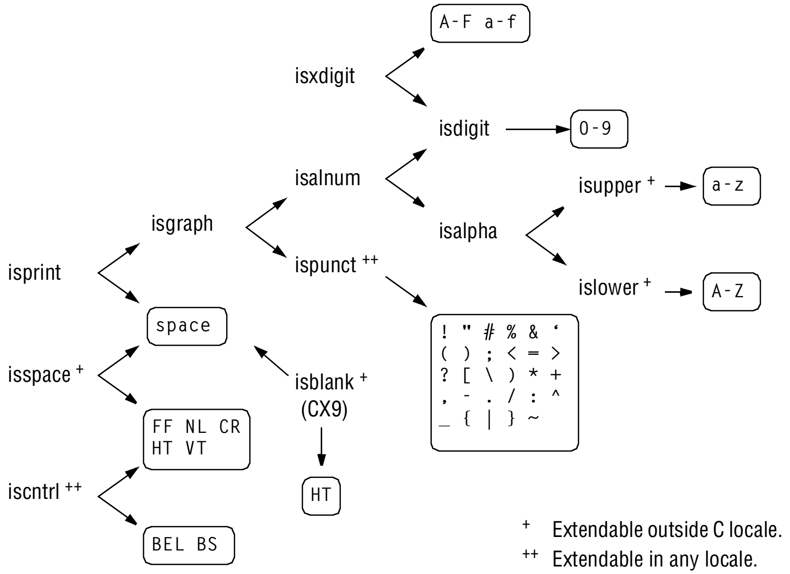
This chapter describes the standard header file `ctype.h`.

INTRODUCTION

Include the standard header `ctype.h` to declare several functions that are useful for classifying and mapping codes from the target character set. Every function that has a parameter of type `int` can accept the value of the macro `EOF` or any value representable as type `unsigned char`. Thus, the argument can be the value returned by any of the functions `fgetc`, `fputc`, `getc`, `getchar`, `putc`, `putchar`, `tolower`, `toupper`, and `ungetc`. You must not call these functions with other argument values.

Other library functions use these functions. The function `scanf`, for example, uses the function `isspace` to determine valid white space within an input field.

The character classification functions are strongly interrelated. Many are defined in terms of other functions. For characters in the basic C character set, a simple diagram shows the dependencies between these functions.



The diagram tells you that the function `isprint` returns non-zero for space or for any character for which the function `isgraph` returns non-zero. The function `isgraph`, in turn, returns non-zero for any character for which either the function `isalnum` or the function `ispunct` returns non-zero. The function `isdigit`, on the other hand, returns non-zero only for the digits 0-9.

An implementation can define additional characters that return non-zero for some of these functions. Any character set can contain additional characters that return non-zero for:

- `ispunct` (provided the characters cause `isalnum` to return zero)
- `iscntrl` (provided the characters cause `isprint` to return zero)

The diagram indicates with ++ those functions that can define additional characters in any character set. Moreover, `locales` other than the `C locale` can define additional characters that return non-zero for:

- `isalpha`, `isupper`, and `islower` (provided the characters cause `iscntrl`, `isdigit`, `ispunct`, and `isspace` to return zero)
- `isspace` (provided the characters cause `isprint` to return zero)

The diagram indicates with + those functions that can define additional characters in locales other than the `C locale`.

Notice that an implementation can define locales other than the `C locale` in which a character can cause `isalpha` (and hence `isalnum`) to return non-zero, yet still cause `isupper` and `islower` to return zero.

SUMMARY

The `ctype.h` header file contains the following functions:

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

In the following sections each function is described.

---

isalnum

SYNTAX

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

DESCRIPTION

The function returns non-zero if *c* is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
```

or any other locale-specific alphabetic character.

---

isalpha

SYNTAX

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

DESCRIPTION

The function returns non-zero if *c* is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

or any other locale-specific alphabetic character.

iscntrl

SYNTAX

`int iscntrl(int c);`

DESCRIPTION

The function returns non-zero if *c* is any of:

`BEL BS CR FF HT NL VT`

or any other implementation-defined control character.

isdigit

SYNTAX

`int isdigit(int c);`

DESCRIPTION

The function returns non-zero if *c* is any of:

`0 1 2 3 4 5 6 7 8 9`

isgraph

SYNTAX

`int isgraph(int c);`

DESCRIPTION

The function returns non-zero if *c* is any character for which either `isalnum` or `ispunct` returns non-zero.

islower

SYNTAX

`int islower(int c);`

The function returns non-zero if *c* is any of:

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

or any other locale-specific lowercase character.

isprint

SYNTAX

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

DESCRIPTION

The function returns non-zero if *c* is space or a character for which `isgraph` returns non-zero.

ispunct

SYNTAX

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

DESCRIPTION

The function returns non-zero if *c* is any of:

```
! " # % & ’ ( ) ; <
= > ? [ \ ] * + , -
. / : ^ _ { | } ~
```

or any other implementation-defined punctuation character.

isspace

SYNTAX

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

DESCRIPTION

The function returns non-zero if *c* is any of:

```
CR FF HT NL VT space
```

or any other locale-specific space character.

isupper

SYNTAX

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

DESCRIPTION

The function returns non-zero if *c* is any of:

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

or any other locale-specific uppercase character.

isxdigit

SYNTAX

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

DESCRIPTION

The function returns non-zero if *c* is any of:

`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

tolower

SYNTAX

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

DESCRIPTION

The function returns the corresponding lowercase letter if one exists and if `isupper(c)`; otherwise, it returns *c*.

toupper

SYNTAX

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

DESCRIPTION

The function returns the corresponding uppercase letter if one exists and if `islower(c)`; otherwise, it returns *c*.

ERRORS – errno.h

This chapter describes the standard header file `errno.h`.

INTRODUCTION

Include the standard header `errno.h` to test the value stored in `errno` by certain library functions. At program startup, the value stored is zero. Library functions store only values greater than zero. Any library function can alter the value stored, but only those cases where a library function is explicitly required to store a value are documented here.

To test whether a library function stores a value in `errno`, the program should store the value zero there immediately before it calls the library function. An implementation can define additional macros in this standard header that you can test for equality with the value stored. All these additional macros have names that begin with `E`.

SUMMARY

The `errno.h` header file contains the following macro definitions:

```
#define EDOM #if_expression
#define EILSEQ #if_expression
#define ERANGE #if_expression
#define errno int_modifiable_lvalue
```

In the following sections each macro definition is described.

EDOM

DEFINITION

`#define EDOM #if_expression`

DESCRIPTION

The macro yields the value stored in `errno` on a domain error.

EILSEQ

DEFINITION

`#define EILSEQ #if_expression`

DESCRIPTION

The macro yields the value stored in `errno` on an invalid multibyte sequence.

ERANGE

DEFINITION

`#define ERANGE #if_expression`

DESCRIPTION

The macro yields the value stored in `errno` on a range error.

errno

DEFINITION

`#define errno int_modifiable_lvalue`

DESCRIPTION

The macro designates an object that is assigned a value greater than zero on certain library errors.

FLOATING-POINT TYPES – float.h

This chapter describes the standard header file `float.h`.

INTRODUCTION

Include the standard header `float.h` to determine various properties of floating-point type representations. The standard header `float.h` is available even in a freestanding implementation.

You can only test the value of the macro `FLT_RADIX` in an `if` directive. (The macro expands to a `#if` expression.) All other macros defined in this header expand to expressions whose values can be determined only when the program executes. (These macros are *`rvalue`* expressions.) Some target environments can change the rounding and error-reporting properties of floating-point type representations while the program is running.

SUMMARY

The `float.h` header file contains the following macro definitions:

```
#define DBL_DIG integer_rvalue
#define DBL_EPSILON double_rvalue
#define DBL_MANT_DIG integer_rvalue
#define DBL_MAX double_rvalue
#define DBL_MAX_10_EXP integer_rvalue
#define DBL_MAX_EXP integer_rvalue
#define DBL_MIN double_rvalue
#define DBL_MIN_10_EXP integer_rvalue
#define DBL_MIN_EXP integer_rvalue

#define FLT_DIG integer_rvalue
#define FLT_EPSILON float_rvalue
#define FLT_MANT_DIG integer_rvalue
#define FLT_MAX float_rvalue
#define FLT_MAX_10_EXP integer_rvalue
#define FLT_MAX_EXP integer_rvalue
#define FLT_MIN float_rvalue
#define FLT_MIN_10_EXP integer_rvalue
#define FLT_MIN_EXP integer_rvalue
```

```
#define FLT_RADIX #if_expression
#define FLT_ROUNDS integer_rvalue
```

```
#define LDBL_DIG integer_rvalue
#define LDBL_EPSILON long_double_rvalue
#define LDBL_MANT_DIG integer_rvalue
#define LDBL_MAX long_double_rvalue
#define LDBL_MAX_10_EXP integer_rvalue
#define LDBL_MAX_EXP integer_rvalue
#define LDBL_MIN long_double_rvalue
#define LDBL_MIN_10_EXP integer_rvalue
#define LDBL_MIN_EXP integer_rvalue
```

In the following sections each definition is described.

DBL_DIG

DEFINITION

```
#define DBL_DIG integer_rvalue
```

where

integer_rvalue `>= 10`

DESCRIPTION

The macro yields the precision in decimal digits for type `double`.

DBL_EPSILON

DEFINITION

```
#define DBL_EPSILON double_rvalue
```

where

double_rvalue `<= 10^(-9)`

DESCRIPTION

The macro yields the smallest *X* of type `double` such that `1.0 + X != 1.0`.

DBL_MANT_DIG

DEFINITION

`#define DBL_MANT_DIG integer_rvalue`

DESCRIPTION

The macro yields the number of mantissa digits, base `FLT_RADIX`, for type `double`.

DBL_MAX

DEFINITION

`#define DBL_MAX double_rvalue`

where

`double_rvalue >= 10^37`

DESCRIPTION

The macro yields the largest finite representable value of type `double`.

DBL_MAX_10_EXP

DEFINITION

`#define DBL_MAX_10_EXP integer_rvalue`

where

`integer_rvalue >= 37`

DESCRIPTION

The macro yields the maximum integer `X`, such that `10^X` is a finite representable value of type `double`.

DBL_MAX_EXP

DEFINITION

`#define DBL_MAX_EXP integer_rvalue`

DESCRIPTION

The macro yields the maximum integer `X`, such that `FLT_RADIX^(X - 1)` is a finite representable value of type `double`.

DBL_MIN

DEFINITION

`#define DBL_MIN` *`double_rvalue`*

`where`

`double_rvalue` `<= 10^(-37)`

DESCRIPTION

The macro yields the smallest normalized, finite representable value of type `double`.

DBL_MIN_10_EXP

DEFINITION

`#define DBL_MIN_10_EXP` *`integer_rvalue`*

where

`integer_rvalue` `<= -37`

DESCRIPTION

The macro yields the minimum integer *`X`* such that `10^`*`X`* is a normalized, finite representable value of type `double`.

DBL_MIN_EXP

DEFINITION

`#define DBL_MIN_EXP` *`integer_rvalue`*

DESCRIPTION

The macro yields the minimum integer *`X`* such that `FLT_RADIX^(`*`X`*` - 1)` is a normalized, finite representable value of type `double`.

FLT_DIG

DEFINITION

`#define FLT_DIG` *`integer_rvalue`*

where

`integer_rvalue` `>= 6`

DESCRIPTION

The macro yields the precision in decimal digits for type `float`.

FLT_EPSILON

DEFINITION

`#define FLT_EPSILON` *`float_rvalue`*

where

`float_rvalue` `<= 10^(-5)`

DESCRIPTION

The macro yields the smallest *X* of type `float` such that `1.0 + X != 1.0`.

FLT_MANT_DIG

DEFINITION

`#define FLT_MANT_DIG` *`integer_rvalue`*

DESCRIPTION

The macro yields the number of mantissa digits, base `FLT_RADIX`, for type `float`.

FLT_MAX

DEFINITION

`#define FLT_MAX` *`float_rvalue`*

where

`float_rvalue` `>= 10^37`

DESCRIPTION

The macro yields the largest finite representable value of type `float`.

FLT_MAX_10_EXP

DEFINITION

`#define FLT_MAX_10_EXP` *`integer_rvalue`*

where

`integer_rvalue` `>= 37`

DESCRIPTION

The macro yields the maximum integer *`X`*, such that `10^`*`X`* is a finite representable value of type `float`.

FLT_MAX_EXP

DEFINITION

`#define FLT_MAX_EXP` *`integer_rvalue`*

DESCRIPTION

The macro yields the maximum integer *`X`*, such that `FLT_RADIX^(`*`X`*` - 1)` is a finite representable value of type `float`.

FLT_MIN

DEFINITION

`#define FLT_MIN` *`float_rvalue`*

where

`float_rvalue` `<= 10^(-37)`

DESCRIPTION

The macro yields the smallest normalized, finite representable value of type `float`.

FLT_MIN_10_EXP

DEFINITION

`#define FLT_MIN_10_EXP` *`integer_rvalue`*

where

`integer_rvalue` `<= -37`

DESCRIPTION

The macro yields the minimum integer *`X`*, such that `10^`*`X`* is a normalized, finite representable value of type `float`.

FLT_MIN_EXP

DEFINITION

`#define FLT_MIN_EXP` *`integer_rvalue`*

DESCRIPTION

The macro yields the minimum integer *`X`*, such that `FLT_RADIX^(`*`X`*` - 1)` is a normalized, finite representable value of type `float`.

FLT_RADIX

DEFINITION

`#define FLT_RADIX` *`#if_expression`*

where

`#if_expression` `>= 2`

DESCRIPTION

The macro yields the radix of all floating-point representations.

FLT_ROUNDS

DEFINITION

`#define FLT_ROUNDS integer_rvalue`

DESCRIPTION

The macro yields a value that describes the current rounding mode for floating-point operations. Notice that the target environment can change the rounding mode while the program executes. How it does so, however, is not specified. The values are:

Value	*Description*
`-1`	Mode is indeterminate
`0`	Rounding is toward zero
`1`	Rounding is to nearest representable value
`2`	Rounding is toward + infinity
`3`	Rounding is toward - infinity

An implementation can define additional values for this macro.

LDBL_DIG

DEFINITION

`#define LDBL_DIG integer_rvalue`

where

`integer_rvalue >= 10`

DESCRIPTION

The macro yields the precision in decimal digits for type `long double`.

LDBL_EPSILON

DEFINITION

`#define LDBL_EPSILON` *`long_double_rvalue`*

where

`long_double_rvalue` `<= 10^(-9)`

DESCRIPTION

The macro yields the smallest *`X`* of type `long double` such that `1.0 +` *`X`* `!= 1.0`.

LDBL_MANT_DIG

DEFINITION

`#define LDBL_MANT_DIG` *`integer_rvalue`*

DESCRIPTION

The macro yields the number of mantissa digits, base `FLT_RADIX`, for type `long double`.

LDBL_MAX

DEFINITION

`#define LDBL_MAX` *`long_double_rvalue`* `>= 10^37`

where

`long_double_rvalue` `>= 10^37`

DESCRIPTION

The macro yields the largest finite representable value of type `long double`.

LDBL_MAX_10_EXP

DEFINITION

`#define LDBL_MAX_10_EXP` *`integer_rvalue`*

where

`integer_rvalue` `>= 37`

DESCRIPTION

The macro yields the maximum integer *`X`*, such that `10^`*`X`* is a finite representable value of type `long double`.

LDBL_MAX_EXP

DEFINITION

`#define LDBL_MAX_EXP` *`integer_rvalue`*

DESCRIPTION

The macro yields the maximum integer *`X`*, such that `FLT_RADIX^(`*`X`*` - 1)` is a finite representable value of type `long double`.

LDBL_MIN

DEFINITION

`#define LDBL_MIN` *`long_double_rvalue`*

where

`long_double_rvalue` `<= 10^(-37)`

DESCRIPTION

The macro yields the smallest normalized, finite representable value of type `long double`.

LDBL_MIN_10_EXP

DEFINITION

`#define LDBL_MIN_10_EXP integer_rvalue`

where

`long_double_rvalue <= 10^(-37)`

DESCRIPTION

The macro yields the minimum integer `X`, such that `10^X` is a normalized, finite representable value of type `long double`.

LDBL_MIN_EXP

DEFINITION

`#define LDBL_MIN_EXP integer_rvalue`

DESCRIPTION

The macro yields the minimum integer `X`, such that `FLT_RADIX^(X - 1)` is a normalized, finite representable value of type `long double`.

ISO 646 – iso646.h

This chapter describes the standard header file `iso646.h`.

INTRODUCTION

Include the standard header `iso646.h` to provide readable alternatives to certain operators or punctuators. The standard header `iso646.h` is available even in a freestanding implementation.

SUMMARY

The `iso646.h` header file contains the following macro definitions:

```
#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 ^=
```

In the following sections each macro definition is described.

and

DEFINITION

```
#define and &&
```

DESCRIPTION

The macro yields the operator `&&`.

and_eq

DEFINITION

```
#define and_eq &=
```

DESCRIPTION

The macro yields the operator `&=`.

bitand

DEFINITION

```
#define bitand &
```

DESCRIPTION

The macro yields the operator `&`.

bitor

DEFINITION

```
#define bitor |
```

DESCRIPTION

The macro yields the operator `|`.

compl

DEFINITION

```
#define compl ~
```

DESCRIPTION

The macro yields the operator ~.

not

DEFINITION

```
#define not !
```

DESCRIPTION

The macro yields the operator `!`.

not_eq

DEFINITION

```
#define not_eq !=
```

DESCRIPTION

The macro yields the operator `!=`.

---

or

DEFINITION

```
#define or ||
```

DESCRIPTION

The macro yields the operator `||`.

---

or_eq

DEFINITION

```
#define or_eq |=
```

DESCRIPTION

The macro yields the operator `|=`.

---

xor

DEFINITION

```
#define xor ^
```

DESCRIPTION

The macro yields the operator `^`.

---

xor_eq

DEFINITION

```
#define xor_eq ^=
```

DESCRIPTION

The macro yields the operator `^=`.

INTEGRAL TYPES – limits.h

This chapter describes the standard header file `limits.h`.

INTRODUCTION

Include the standard header `limits.h` to determine various properties of the integer type representations. The standard header `limits.h` is available even in a freestanding implementation.

You can test the values of all these macros in an `if` directive. (The macros are `#if` expressions.)

SUMMARY

The `limits.h` header file contains the following macro definitions:

```
#define CHAR_BIT #if_expression
#define CHAR_MAX #if_expression
#define CHAR_MIN #if_expression
#define INT_MAX #if_expression
#define INT_MIN #if_expression
#define LONG_MAX #if_expression
#define LONG_MIN #if_expression
#define MB_LEN_MAX #if_expression
#define SCHAR_MAX #if_expression
#define SCHAR_MIN #if_expression
#define SHRT_MAX #if_expression
#define SHRT_MIN #if_expression
#define UCHAR_MAX #if_expression
#define UINT_MAX #if_expression
#define ULONG_MAX #if_expression
#define USHRT_MAX #if_expression
```

In the following sections each macro definition is described.

CHAR_BIT

DEFINITION

`#define CHAR_BIT #if_expression`

where

`#if_expression >= 8`

DESCRIPTION

The macro yields the maximum value for the number of bits used to represent an object of type `char`.

CHAR_MAX

DEFINITION

`#define CHAR_MAX #if_expression`

where

`#if_expression >= 127`

DESCRIPTION

The macro yields the maximum value for type `char`. Its value is:

- `SCHAR_MAX` if `char` represents negative values.
- `UCHAR_MAX` otherwise.

CHAR_MIN

DEFINITION

`#define CHAR_MIN #if_expression`

where

`#if_expression <= 0`

DESCRIPTION

The macro yields the minimum value for type `char`. Its value is:

- `SCHAR_MIN` if `char` represents negative values.
- Zero otherwise.

INT_MAX

DEFINITION

`#define INT_MAX` *`#if_expression`*

where

`#if_expression` `>= 32,767`

DESCRIPTION

The macro yields the maximum value for type `int`.

INT_MIN

DEFINITION

`#define INT_MIN` *`#if_expression`*

where

`#if_expression` `<= -32,767`

DESCRIPTION

The macro yields the minimum value for type `int`.

LONG_MAX

DEFINITION

`#define LONG_MAX` *`#if_expression`*

where

`#if_expression` `>= 2,147,483,647`

DESCRIPTION

The macro yields the maximum value for type `long`.

LONG_MIN

DEFINITION

`#define LONG_MIN #if_expression`

where

`#if_expression <= -2,147,483,647`

DESCRIPTION

The macro yields the minimum value for type `long`.

MB_LEN_MAX

DEFINITION

`#define MB_LEN_MAX #if_expression`

where

`#if_expression >= 1`

DESCRIPTION

The macro yields the maximum number of characters that constitute a multibyte character in any supported locale. Its value is >= `MB_CUR_MAX`.

SCHAR_MAX

DEFINITION

`#define SCHAR_MAX #if_expression`

where

`#if_expression >= 127`

DESCRIPTION

The macro yields the maximum value for type `signed char`.

SCHAR_MIN

DEFINITION

`#define SCHAR_MIN #if_expression`

where

`#if_expression <= -127`

DESCRIPTION

The macro yields the minimum value for type `signed char`.

SHRT_MAX

DEFINITION

`#define SHRT_MAX #if_expression`

where

`#if_expression >= 32,767`

DESCRIPTION

The macro yields the maximum value for type `short`.

SHRT_MIN

DEFINITION

`#define SHRT_MIN #if_expression`

where

`#if_expression <= -32,767`

DESCRIPTION

The macro yields the minimum value for type `short`.

UCHAR_MAX

DEFINITION

`#define UCHAR_MAX #`*`if_expression`*

where

`#if_expression` `>= 255`

DESCRIPTION

The macro yields the maximum value for type `unsigned char`.

UINT_MAX

DEFINITION

`#define UINT_MAX #`*`if_expression`*

where

`#if_expression` `>= 65,535`

DESCRIPTION

The macro yields the maximum value for type `unsigned int`.

ULONG_MAX

DEFINITION

`#define ULONG_MAX #`*`if_expression`*

where

`#if_expression` `>= 4,294,967,295`

DESCRIPTION

The macro yields the maximum value for type `unsigned long`.

USHRT_MAX

DEFINITION

`#define USHRT_MAX #if_expression`

where

`#if_expression >= 65,535`

DESCRIPTION

The macro yields the maximum value for type `unsigned short`.

LOCAL INFORMATION DEFINITIONS – locale.h

This chapter describes the standard header file `locale.h`.

INTRODUCTION

Include the standard header `locale.h` to alter or access properties of the current locale—a collection of culture-specific information. An implementation can define additional macros in this standard header with names that begin with `LC_`. You can use any of these macro names as the locale category argument (which selects a cohesive subset of a locale) to `setlocale`.

SUMMARY

The `locale.h` header file contains the following macro definitions and functions:

```
#define LC_ALL integer_constant_expression
#define LC_COLLATE integer_constant_expression
#define LC_CTYPE integer_constant_expression
#define LC_MONETARY integer_constant_expression
#define LC_NUMERIC integer_constant_expression
#define LC_TIME integer_constant_expression
#define NULL null_pointer_constant
struct lconv;
struct lconv *localeconv(void);
char *setlocale(int category, const char *locale);
```

In the following sections each macro definition and function is described.

LC_ALL

DEFINITION

```
#define LC_ALL integer_constant_expression
```

DESCRIPTION

The macro yields the locale category argument value that affects all locale categories.

LC_COLLATE

DEFINITION

`#define LC_COLLATE` *`integer_constant_expression`*

DESCRIPTION

The macro yields the locale category argument value that affects the collation functions `strcoll` and `strxfrm`.

LC_CTYPE

DEFINITION

`#define LC_CTYPE` *`integer_constant_expression`*

DESCRIPTION

The macro yields the locale category argument value that affects character classification functions and various multibyte conversion functions.

LC_MONETARY

DEFINITION

`#define LC_MONETARY` *`integer_constant_expression`*

DESCRIPTION

The macro yields the locale category argument value that affects monetary information returned by `localeconv`.

LC_NUMERIC

DEFINITION

`#define LC_NUMERIC` *`integer_constant_expression`*

DESCRIPTION

The macro yields the locale category argument value that affects numeric information returned by `localeconv`, including the decimal point used by numeric conversion and by the read and write functions.

LC_TIME

DEFINITION

`#define LC_TIME` *`integer_constant_expression`*

DESCRIPTION

The macro yields the locale category argument value that affects the time conversion function `strftime`.

NULL

DEFINITION

`#define NULL` *`null_pointer_constant`*

where

`null_pointer_constant` is either `0`, `0L`, or `(void *)0`.

DESCRIPTION

The macro yields a null pointer constant that is usable as an address constant expression.

lconv

DEFINITION

```
struct lconv {
    ELEMENT                   "C" LOCALE  LOCALE CATEGORY
    char *currency_symbol;    ""          LC_MONETARY
    char *decimal_point;      "."         LC_NUMERIC
    char *grouping;           ""          LC_NUMERIC
    char *int_curr_symbol;    ""          LC_MONETARY
    char *mon_decimal_point;  ""          LC_MONETARY
    char *mon_grouping;       ""          LC_MONETARY
    char *mon_thousands_sep;  ""          LC_MONETARY
    char *negative_sign;      ""          LC_MONETARY
    char *positive_sign;      ""          LC_MONETARY
    char *thousands_sep;      ""          LC_NUMERIC
    char frac_digits;         CHAR_MAX    LC_MONETARY
    char int_frac_digits;     CHAR_MAX    LC_MONETARY
    char n_cs_precedes;       CHAR_MAX    LC_MONETARY
    char n_sep_by_space;      CHAR_MAX    LC_MONETARY
    char n_sign_posn;         CHAR_MAX    LC_MONETARY
    char p_cs_precedes;       CHAR_MAX    LC_MONETARY
char p_sep_by_space;      CHAR_MAX    LC_MONETARY
char p_sign_posn;         CHAR_MAX    LC_MONETARY
};
```

DESCRIPTION

`struct lconv` contains members that describe how to format numeric and monetary values. Functions in the Standard C library use only the field `decimal_point`. The information is otherwise advisory:

- Members of type pointer to `char` all point to C strings.
- Members of type `char` have non-negative values.
- A char value of `CHAR_MAX` indicates that a meaningful value is not available in the current locale.

The members shown above can occur in arbitrary order and can be interspersed with additional members. The comment following each member shows its value for the "C" locale, the locale in effect at program startup, followed by the locale category that can affect its value.

A description of each member follows, with an example that would be suitable for a USA locale.

Member	*Description*	*Example*
`currency_symbol`	The local currency symbol.	`"$"`
`decimal_point`	The decimal point for non-monetary values.	`"."`

Member	*Description*	*Example*
`grouping`	The size of digit groups for non-monetary values. Successive elements of the string describe groups moving to the left from the decimal point: ◆ An element value of zero (the terminating null character) calls for the previous element value to be repeated indefinitely. ◆ An element value of `CHAR_MAX` ends any further grouping (and hence ends the string). Thus, the array `{3, 2, CHAR_MAX}` calls for a group of three digits, then two, then whatever remains, as in `9876,54,321`, while `"\3"` calls for repeated groups of three digits, as in `987,654,321`.	`"\3"`
`int_curr_symbol`	The international currency symbol specified by ISO 4217.	`"USD "`
`mon_decimal_point`	The decimal point for monetary values.	`"."`
`mon_grouping`	The size of digit groups for monetary values. Successive elements of the string describe groups going away from the decimal point. The encoding is the same as for `grouping`.	
`mon_thousands_sep`	The separator for digit groups to the left of the decimal point for monetary values.	`","`
`negative_sign`	The negative sign for monetary values.	`"-"`
`positive_sign`	The positive sign for monetary values.	`"+"`

Member	*Description*	*Example*
`thousands_sep`	The separator for digit groups to the left of the decimal point for non-monetary values.	`","`
`frac_digits`	The number of digits to display to the right of the decimal point for monetary values.	`2`
`int_frac_digits`	The number of digits to display to the right of the decimal point for international monetary values.	`2`
`n_cs_precedes`	Whether the currency symbol precedes or follows the value for negative monetary values: ◆ A value of `0` indicates that the symbol follows the value. ◆ A value of 1 indicates that the symbol precedes the value.	`1`
`n_sep_by_space`	Whether the currency symbol is separated by a space or by no space from the value for negative monetary values: ◆ A value of `0` indicates that no space separates symbol and value. ◆ A value of 1 indicates that a space separates symbol and value.	`0`

Member	*Description*	*Example*
`n_sign_posn`	The format for negative monetary values: ◆ A value of `0` indicates that parentheses surround the value and the `currency_symbol`. ◆ A value of `1` indicates that the negative sign precedes the value and the `currency_symbol`. ◆ A value of `2` indicates that the negative sign follows the value and the `currency_symbol`. ◆ A value of `3` indicates that the negative sign immediately precedes the `currency_symbol`. ◆ A value of `4` indicates that the negative sign immediately follows the `currency_symbol`.	`4`
`p_cs_precedes`	Whether the `currency_symbol` precedes or follows the value for positive monetary values: ◆ A value of `0` indicates that the symbol follows the value. ◆ A value of `1` indicates that the symbol precedes the value.	`1`
`p_sep_by_space`	Whether the currency symbol is separated by a space or by no space from the value for positive monetary values: ◆ A value of `0` indicates that no space separates symbol and value. ◆ A value of `1` indicates that a space separates symbol and value.	`0`

Member	*Description*	*Example*
`p_sign_posn`	The format for positive monetary values: ◆ A value of 0 indicates that parentheses surround the value and the `currency_symbol`. ◆ A value of 1 indicates that the negative sign precedes the value and the `currency_symbol`. ◆ A value of 2 indicates that the negative sign follows the value and the `currency_symbol`. ◆ A value of 3 indicates that the negative sign immediately precedes the `currency_symbol`. ◆ A value of 4 indicates that the negative sign immediately follows the `currency_symbol`.	`4`

localeconv

SYNTAX

```
struct lconv *localeconv(void);
```

DESCRIPTION

The function returns a pointer to a static-duration structure containing numeric formatting information for the current locale. You cannot alter values stored in the static-duration structure. The stored values can change on later calls to `localeconv` or on calls to `setlocale` that alter any of the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC`.

setlocale

SYNTAX

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

DESCRIPTION

The function either returns a pointer to a static-duration string describing a new locale or returns a null pointer (if the new locale cannot be selected). The value of category selects one or more locale categories, each of which must match the value of one of the macros defined in this standard header with names that begin with `LC_`.

If locale is a null pointer, the locale remains unchanged. If locale points to the string "C", the new locale is the "C" locale for the locale category specified. If locale points to the string "", the new locale is the native locale (a default locale presumably tailored for the local culture) for the locale category specified. locale can also point to a string returned on an earlier call to setlocale or to other strings that the implementation can define.

At program startup, the target environment calls `setlocale( LC_ALL, "C")` before it calls `main`.

MATHEMATICS – math.h

This chapter describes the standard header file `math.h`.

INTRODUCTION

Include the standard header `math.h` to declare several functions that perform common mathematical operations on floating-point values.

A domain error exception occurs when the function is not defined for its input argument value or values. A function reports a domain error by storing the value of `EDOM` in `errno` and returning a peculiar value defined for each implementation.

A range error exception occurs when the return value of the function is defined but cannot be represented. A function reports a range error by storing the value of `ERANGE` in `errno` and returning one of three values:

- `HUGE_VAL`, if the value of a function returning double is positive and too large in magnitude to represent.
- Zero, if the value of the function is too small to represent with a finite value.
- `-HUGE_VAL`, if the value of a function returning `double` is negative and too large in magnitude to represent.

SUMMARY

The `math.h` header file contains the following macro definitions and functions:

```
#define HUGE_VAL <double rvalue>
double acos(double x);
float acosf(float x);
long double acosl(long double x);

double asin(double x);
float asinf(float x);
long double asinl(long double x);

double atan(double x);
float atanf(float x);
long double atanl(long double x);
```

```
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);

double ceil(double x);
float ceilf(float x);
long double ceill(long double x);

double cos(double x);
float cosf(float x);
long double cosl(long double x);

double cosh(double x);
float coshf(float x);
long double coshl(long double x);

double exp(double x);
float expf(float x);
long double expl(long double x);

double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);

double floor(double x);
float floorf(float x);
long double floorl(long double x);

double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);

double frexp(double x, int *pexp);
float frexpf(float x, int *pexp);
long double frexpl(long double x, int *pexp);

double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);

double log(double x);
float logf(float x);
long double logl(long double x);

double log10(double x);
float log10f(float x);
long double log10l(long double x);
```

```
double modf(double x, double *pint);
float modff(float x, float *pint);
long double modfl(long double x, long double *pint);

double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);

double sin(double x);
float sinf(float x);
long double sinl(long double x);

double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);

double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);

double tan(double x);
float tanf(float x);
long double tanl(long double x);

double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

In the following sections each macro definition and function is described.

HUGE_VAL

DEFINITION

```
#define HUGE_VAL double_rvalue
```

DESCRIPTION

The macro yields the value returned by some functions on a range error. The value can be a representation of infinity.

acos, acosf, acosl

SYNTAX

```
double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

DESCRIPTION

The function returns the angle whose cosine is *x*, in the range [0, pi] radians.

asin, asinf, asinl

SYNTAX

```
double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

DESCRIPTION

The function returns the angle whose sine is *x*, in the range [-pi/2, + pi/2] radians.

atan, atanf, atanl

SYNTAX

```
double atan(double x);
float atanf(float x);
long double atanl(long double x);
```

DESCRIPTION

The function returns the angle whose tangent is *x*, in the range [-pi/2, + pi/2] radians.

atan2, atan2f, atan2l

SYNTAX

```
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

DESCRIPTION

The function returns the angle whose tangent is *y*/*x*, in the full angular range [-pi, +pi] radians.

ceil, ceilf, ceill

SYNTAX

```
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

DESCRIPTION

The function returns the smallest integer value not less than *x*.

cos, cosf, cosl

SYNTAX

```
double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

DESCRIPTION

The function returns the cosine of *x* for *x* in radians. If *x* is large the value returned might not be meaningful, but the function reports no error.

cosh, coshf, coshl

SYNTAX

```
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
```

DESCRIPTION

The function returns the hyperbolic cosine of *x*.

exp, expf, expl

SYNTAX

```
double exp(double x);
float expf(float x);
long double expl(long double x);
```

DESCRIPTION

The function returns the exponential of *x*, `e^`*x*.

fabs, fabsf, fabsl

SYNTAX

```
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

DESCRIPTION

The function returns the absolute value of *x*, $|x|$, the same as `abs`.

floor, floorf, floorl

SYNTAX

```
double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

DESCRIPTION

The function returns the largest integer value not greater than *x*.

fmod, fmodf, fmodl

SYNTAX

```
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

DESCRIPTION

The function returns the remainder of x/y, which is defined as follows:

- If *y* is zero, the function either reports a domain error or simply returns zero.

- Otherwise, if `0 <=` *x*, the value is *x* `-` `i*`*y* for some integer i such that:

  `0 <= i*|y| <= x < (i + 1)*|y|`

- Otherwise, *x* `< 0` and the value is *x* `-` `i*`*y* for some integer i such that:

  `i*|y| <= x < (i + 1)*|y| <= 0`

frexp, frexpf, frexpl

SYNTAX

```
double frexp(double x, int *pexp);
float frexpf(float x, int *pexp);
long double frexpl(long double x, int *pexp);
```

DESCRIPTION

The function determines a fraction `f` and base-2 integer `i` that represent the value of *x*. It returns the value `f` and stores the integer `i` in `*`*pexp*, such that `|f|` is in the interval `[1/2, 1]` or has the value `0`, and *x* equals `f*2^i`. If *x* is zero, `*`*pexp* is also zero.

ldexp, ldexpf, ldexpl

SYNTAX

```
double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

DESCRIPTION

The function returns *x*`*2^`*exp*.

log, logf, logl

SYNTAX

```
double log(double x);
float logf(float x);
long double logl(long double x);
```

DESCRIPTION

The function returns the natural logarithm of *x*.

log10, log10f, log10l

SYNTAX

```
double log10(double x);
float log10f(float x);
long double log10l(long double x);
```

DESCRIPTION

The function returns the base-10 logarithm of *x*.

modf, modff, modfl

SYNTAX

```
double modf(double x, double *pint);
float modff(float x, float *pint);
long double modfl(long double x, long double *pint);
```

DESCRIPTION

The function determines an integer `i` plus a fraction `f` that represent the value of *x*. It returns the value `f` and stores the integer `i` in `*pint`, such that `f + i ==` *x*, `|f|` is in the interval `[0, 1]`, and both `f` and `i` have the same sign as *x*.

pow, powf, powl

SYNTAX

```
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

DESCRIPTION

The function returns *x* raised to the power *y*, *x*^*y*.

sin, sinf, sinl

SYNTAX

```
double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

DESCRIPTION

The function returns the sine of *x* for *x* in radians. If *x* is large the value returned might not be meaningful, but the function reports no error.

sinh, sinhf, sinhl

SYNTAX

```
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

DESCRIPTION

The function returns the hyperbolic sine of *x*.

sqrt, sqrtf, sqrtl

SYNTAX

```
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

DESCRIPTION

The function returns the square root of *x*, `x^(1/2)`.

tan, tanf, tanl

SYNTAX

```
double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

DESCRIPTION

The function returns the tangent of *x* for *x* in radians.If *x* is large the value returned might not be meaningful, but the function reports no error.

tanh, tanhf, tanhl

SYNTAX

```
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

DESCRIPTION

The function returns the hyperbolic tangent of *x*.

NON-LOCAL JUMPS – setjmp.h

This chapter describes the standard header file `setjmp.h`.

INTRODUCTION

Include the standard header `setjmp.h` to perform control transfers that bypass the normal function call and return protocol.

SUMMARY

The `setjmp.h` header file contains the following macro definitions and functions:

```
typedef a-type jmp_buf;
void longjmp(jmp_buf env, int val);
#define setjmp(jmp_buf env) int_rvalue
```

Below each definition and function is described.

jmp_buf

DEFINITION

```
typedef a-type jmp_buf;
```

DESCRIPTION

The type is the array type `a-type` of an object that you declare to hold the context information stored by `setjmp` and accessed by `longjmp`.

longjmp

SYNTAX

```
void longjmp(jmp_buf env, int val);
```

DESCRIPTION

The function causes a second return from the execution of `setjmp` that stored the current context value in *`env`*. If *`val`* is non-zero, the return value is *`val`*; otherwise, it is 1.

The function that was active when `setjmp` stored the current context value must not have returned control to its caller. An object with dynamic duration that does not have a volatile type and whose stored value has changed since the current context value was stored will have a stored value that is indeterminate.

setjmp

SYNTAX

```
#define setjmp(jmp_buf env) int_rvalue
```

DESCRIPTION

The macro stores the current context value in the array designated by *`env`* and returns zero. A later call to `longjmp` that accesses the same context value causes `setjmp` to again return, this time with a non-zero value. You can use the macro `setjmp` only in an expression that:

- has no operators,
- has only the unary operator `!`,
- has one of the relational or equality operators (`==`, `!=`, `<`, `<=`, `>`, or `>=`) with the other operand an integer constant expression.

You can write such an expression only as the expression part of a `do`, *`expression`*, `for`, `if`, `if-else`, `switch`, or `while` statement.

SIGNAL HANDLING – signal.h

This chapter describes the standard header file `signal.h`.

INTRODUCTION

Include the standard header `signal.h` to specify how the program handles signals while it executes. A signal can report some exceptional behavior within the program, such as division by zero. Or a signal can report some asynchronous event outside the program, such as someone striking an interactive attention key on a keyboard.

You can report any signal by calling `raise`. Each implementation defines what signals it generates (if any) and under what circumstances it generates them. An implementation can define signals other than the ones listed here. The standard header `signal.h` can define additional macros with names beginning with `SIG` to specify the values of additional signals. All such values are *`integer_constant_expressions`* `>= 0`.

You can specify a signal handler for each signal. A signal handler is a function that the target environment calls when the corresponding signal occurs. The target environment suspends execution of the program until the signal handler returns or calls `longjmp`. For maximum portability, an asynchronous signal handler should only:

- make calls (that succeed) to the function `signal`
- assign values to objects of type `volatile sig_atomic_t`
- return control to its caller

If the signal reports an error within the program (and the signal is not asynchronous), the signal handler can terminate by calling `abort`, `exit`, or `longjmp`.

SUMMARY

The `signal.h` header file contains the following macro definitions and functions:

```
#define SIGABRT integer_constant_expression
#define SIGFPE integer_constant_expression
#define SIGILL integer_constant_expression
#define SIGINT integer_constant_expression
#define SIGSEGV integer_constant_expression
#define SIGTERM integer_constant_expression
#define SIG_DFL address_constant_expression
#define SIG_ERR address_constant_expression
#define SIG_IGN address_constant_expression
int raise(int sig);
typedef i-type sig_atomic_t;
void (*signal(int sig, void (*func)(int)))(int);
```

Below each definition and function is described.

SIGABRT

DEFINITION

```
#define SIGABRT integer_constant_expression
```

where:

`integer_constant_expression` `>= 0`

DESCRIPTION

The macro yields the sig argument value for the abort signal.

SIGFPE

DEFINITION

```
#define SIGFPE integer_constant_expression
```

where:

`integer_constant_expression` `>= 0`

DESCRIPTION

The macro yields the *`sig`* argument value for the arithmetic error signal, such as for division by zero or result out of range.

SIGILL

DEFINITION

`#define SIGILL integer_constant_expression`

where:

`integer_constant_expression >= 0`

DESCRIPTION

The macro yields the `sig` argument value for the invalid execution signal, such as for a corrupted function image.

SIGINT

DEFINITION

`#define SIGINT integer_constant_expression`

where:

`integer_constant_expression >= 0`

DESCRIPTION

The macro yields the `sig` argument value for the asynchronous interactive attention signal.

SIGSEGV

DEFINITION

`#define SIGSEGV integer_constant_expression`

where:

`integer_constant_expression >= 0`

DESCRIPTION

The macro yields the `sig` argument value for the invalid storage access signal, such as for an erroneous `lvalue` expression.

SIGTERM

DEFINITION

`#define SIGTERM` *`integer_constant_expression`*

where:

`integer_constant_expression` `>= 0`

DESCRIPTION

The macro yields the *`sig`* argument value for the asynchronous termination request signal.

SIG_DFL

DEFINITION

`#define SIG_DFL` *`address_constant_expression`*

DESCRIPTION

The macro yields the *`func`* argument value to signal to specify default signal handling.

SIG_ERR

DEFINITION

`#define SIG_ERR` *`address_constant_expression`*

DESCRIPTION

The macro yields the signal return value to specify an erroneous call.

SIG_IGN

DEFINITION

`#define SIG_IGN` *`address_constant_expression`*

DESCRIPTION

The macro yields the *`func`* argument value to signal to specify that the target environment is to henceforth ignore the signal.

raise

SYNTAX

```
int raise(int sig);
```

DESCRIPTION

The function sends the signal *`sig`* and returns zero if the signal is successfully reported.

sig_atomic_t

DEFINITION

```
typedef i-type sig_atomic_t;
```

DESCRIPTION

The type is the integer type `i-type` for objects whose stored value is altered by an assigning operator as an atomic operation (an operation that never has its execution suspended while partially completed). You declare such objects to communicate between signal handlers and the rest of the program.

signal

SYNTAX

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

DESCRIPTION

The function specifies the new handling for signal *`sig`* and returns the previous handling, if successful; otherwise, it returns `SIG_ERR`.

- If *`func`* is `SIG_DFL`, the target environment commences default handling (as defined by the implementation).
- If *`func`* is `SIG_IGN`, the target environment ignores subsequent reporting of the signal.
- Otherwise, *`func`* must be the address of a function returning `void` that the target environment calls with a single *`int`* argument. The target environment calls this function to handle the signal when it is next reported, with the value of the signal as its argument.

When the target environment calls a signal handler:

- The target environment can block further occurrences of the corresponding signal until the handler returns, calls `longjmp`, or calls `signal` for that signal.
- The target environment can perform default handling of further occurrences of the corresponding signal.
- For signal `SIGILL`, the target environment can leave handling unchanged for that signal.

VARIABLE ARGUMENTS – stdarg.h

This chapter describes the standard header file `stdarg.h`.

INTRODUCTION

Include the standard header `stdarg.h` to access the unnamed additional arguments (arguments with no corresponding parameter declarations) in a function that accepts a varying number of arguments. To access the additional arguments:

- The program must first execute the macro `va_start` within the body of the function to initialize an object with context information.
- Subsequent execution of the macro `va_arg`, designating the same context information, yields the values of the additional arguments in order, beginning with the first unnamed argument. You can execute the macro `va_arg` from any function that can access the context information saved by the macro `va_start`.
- If you have executed the macro `va_start` in a function, you must execute the macro `va_end` in the same function, designating the same context information, before the function returns.

You can repeat this sequence (as needed) to access the arguments as often as you want.

You declare an object of type `va_list` to store context information. `va_list` can be an array type, which affects how the program shares context information with functions that it calls. (The address of the first element of an array is passed, rather than the object itself.)

For example, here is a function that concatenates an arbitrary number of strings onto the end of an existing string (assuming that the existing string is stored in an object large enough to hold the resulting string):

```
#include <stdarg.h>
void va_cat(char *s, ...)
    {
    char *t;
    va_list ap;

    va_start(ap, s);
    while (t = va_arg(ap, char *)) null pointer ends list
        {
        s += strlen(s);            skip to end
        strcpy(s, t);              and copy a string
        }
    va_end(ap);
    }
```

SUMMARY

The `stdarg.h` header file contains the following definitions:

```
#define va_arg(va_list ap, T)
#define va_end(va_list ap)
typedef do-type va_list;
#define va_start(va_list ap, last-par)
```

Below each definition is described.

va_arg

DEFINITION

```
#define va_arg(va_list ap, T)
```

DESCRIPTION

The macro yields the value of the next argument in order, specified by the context information designated by *`ap`*. The additional argument must be of object type `T` after applying the rules for promoting arguments in the absence of a function prototype.

va_end

DEFINITION

```
#define va_end(va_list ap)
```

DESCRIPTION

The macro performs any cleanup necessary, after processing the context information designated by *`ap`*, so that the function can return.

va_list

DEFINITION

```
typedef do-type va_list;
```

DESCRIPTION

The type is the object type `do-type` that you declare to hold the context information initialized by `va_start` and used by `va_arg` to access additional unnamed arguments.

va_start

DEFINITION

```
#define va_start(va_list ap, last-par)
```

DESCRIPTION

The macro stores initial context information in the object designated by *`ap`*. *`last-par`* is the name of the last parameter you declare. For example, *`last-par`* is `b` for the function declared as `int f(int a, int b, ...)`. The last parameter must not have register storage class, and it must have a type that is not changed by the translator. It cannot have:

- an array type
- a function type
- type float
- an integer type that changes when promoted

COMMON DEFINITIONS – stddef.h

This chapter describes the standard header file `stddef.h`.

INTRODUCTION

Include the standard header `stddef.h` to define several types and macros that are of general use throughout the program. The standard header `stddef.h` is available even in a freestanding implementation.

SUMMARY

The `stddef.h` header file contains the following definitions:

```
#define NULL null_pointer_constant
#define offsetof(s-type, mbr)
typedef si-type ptrdiff_t;
typedef ui-type size_t;
typedef i-type wchar_t;
```

Below each definition is described.

NULL

DEFINITION

```
#define NULL null_pointer_constant
```

where

`null_pointer_constant` is either `0`, `0L`, or `(void *)0`

DESCRIPTION

The macro yields a null pointer constant that is usable as an address constant expression.

offsetof

DEFINITION

```
#define offsetof(s-type, mbr)
```

DESCRIPTION

The macro yields the offset in bytes, of type `size_t`, of member `mbr` from the beginning of structure type `s-type`, where for `X` of type `s-type`, `&X.mbr` is an address constant expression.

ptrdiff_t

DEFINITION

```
typedef si-type ptrdiff_t;
```

DESCRIPTION

The type is the signed integer type `si-type` of an object that you declare to store the result of subtracting two pointers.

size_t

DEFINITION

```
typedef ui-type size_t;
```

DESCRIPTION

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

wchar_t

DEFINITION

```
typedef i-type wchar_t;
```

DESCRIPTION

The type is the integer type `i-type` of a wide-character constant, such as `L'X'`. You declare an object of type `wchar_t` to hold a wide character.

INPUT/OUTPUT – stdio.h

This chapter describes the standard header file `stdio.h`.

INTRODUCTION

Include the standard header `stdio.h` so that you can perform input and output operations on streams and files.

SUMMARY

The `stdio.h` header file contains the following macro definitions and functions:

```
#define _IOFBF integer_constant_expression
#define _IOLBF integer_constant_expression
#define _IONBF integer_constant_expression
#define BUFSIZ integer_constant_expression
#define EOF integer_constant_expression
typedef o-type FILE;
#define FILENAME_MAX integer_constant_expression
#define FOPEN_MAX integer_constant_expression
#define L_tmpnam integer_constant_expression
#define NULL null_pointer_constant
#define SEEK_CUR integer_constant_expression
#define SEEK_END integer_constant_expression
#define SEEK_SET integer_constant_expression
#define TMP_MAX integer_constant_expression

void clearerr(FILE *stream);
int fclose(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fflush(FILE *stream);
int fgetc(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
char *fgets(char *s, int n, FILE *stream);
FILE *fopen(const char *filename, const char *mode);
typedef o-type fpos_t;
int fprintf(FILE *stream, const char *format, ...);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nelem, FILE
     *stream);
```

```
FILE *freopen(const char *filename, const char *mode,
     FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fseek(FILE *stream, long offset, int mode);
int fsetpos(FILE *stream, const fpos_t *pos);
long ftell(FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nelem,
     FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);
void perror(const char *s);
int printf(const char *format, ...);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int remove(const char *filename);
int rename(const char *old, const char *new);
void rewind(FILE *stream);

int scanf(const char *format, ...);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t
     size);
typedef ui-type size_t;
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
#define stderr pointer_to_FILE_rvalue
#define stdin pointer_to_FILE_rvalue
#define stdout pointer_to_FILE_rvalue
FILE *tmpfile(void)
char *tmpnam(char *s);
int ungetc(int c, FILE *stream);
int vfprintf(FILE *stream, const char *format, va_list
     ap);
int vprintf(const char *format, va_list ap);
int vsprintf(char *s, const char *format, va_list ap);
```

Below each definition and function is described.

_IOFBF

DEFINITION

`#define _IOFBF integer_constant_expression`

DESCRIPTION

The macro yields the value of the mode argument to `setvbuf` to indicate full buffering. (Flush the stream buffer only when it fills.)

_IOLBF

DEFINITION

`#define _IOLBF integer_constant_expression`

DESCRIPTION

The macro yields the value of the mode argument to `setvbuf` to indicate line buffering. (Flush the stream buffer at the end of a text line.)

_IONBF

DEFINITION

`#define _IONBF integer_constant_expression`

DESCRIPTION

The macro yields the value of the mode argument to `setvbuf` to indicate no buffering. (Flush the stream buffer at the end of each write operation.)

BUFSIZ

DEFINITION

`#define BUFSIZ integer_constant_expression`

where:

`integer_constant_expression >= 256`

DESCRIPTION

The macro yields the size of the stream buffer used by `setbuf`.

EOF

DEFINITION

```
#define EOF integer_constant_expression
```

where:

```
integer_constant_expression < 0
```

DESCRIPTION

The macro yields the return value used to signal the end of a stream or to report an error condition.

FILE

DEFINITION

```
typedef o-type FILE;
```

DESCRIPTION

The type is an object type `o-type` that stores all control information for a stream. The functions `fopen` and `freopen` allocate all `FILE` objects used by the read and write functions.

FILENAME_MAX

DEFINITION

```
#define FILENAME_MAX integer_constant_expression
```

where:

```
integer_constant_expression > 0
```

DESCRIPTION

The macro yields the maximum size array of characters that you must provide to hold a filename.

FOPEN_MAX

DEFINITION

```
#define FOPEN_MAX integer_constant_expression
```

where:

```
integer_constant_expression >= 8
```

DESCRIPTION

The macro yields the maximum number of files that the target environment permits to be simultaneously open (including `stderr`, `stdin`, and `stdout`).

L_tmpnam

DEFINITION

`#define L_tmpnam` *`integer_constant_expression`*

where:

`integer_constant_expression` `> 0`

DESCRIPTION

The macro yields the number of characters that the target environment requires for representing temporary filenames created by `tmpnam`.

NULL

DEFINITION

`#define NULL` *`null_pointer_constant`*

where:

`null_pointer_constant` is either `0`, `0L`, or `(void *)0`

DESCRIPTION

The macro yields a null pointer constant that is usable as an address constant expression.

SEEK_CUR

DEFINITION

`#define SEEK_CUR` *`integer_constant_expression`*

DESCRIPTION

The macro yields the value of the mode argument to `fseek` to indicate seeking relative to the current file-position indicator.

SEEK_END

DEFINITION

`#define SEEK_END` *`integer_constant_expression`*

DESCRIPTION

The macro yields the value of the mode argument to `fseek` to indicate seeking relative to the end of the file.

SEEK_SET

DEFINITION

`#define SEEK_SET` *`integer_constant_expression`*

DESCRIPTION

The macro yields the value of the mode argument to `fseek` to indicate seeking relative to the beginning of the file.

TMP_MAX

DEFINITION

`#define TMP_MAX` *`integer_constant_expression`*

where:

`integer_constant_expression` `>= 25`

DESCRIPTION

The macro yields the minimum number of distinct filenames created by the function `tmpnam`.

clearerr

SYNTAX

`void clearerr(FILE` *`*stream`*`);`

DESCRIPTION

The function clears the end-of-file and error indicators for the stream *`stream`*.

fclose

SYNTAX

```
int fclose(FILE *stream);
```

DESCRIPTION

The function closes the file associated with the stream *`stream`*. It returns zero if successful; otherwise, it returns `EOF`. `fclose` writes any buffered output to the file, deallocates the stream buffer if it was automatically allocated, and removes the association between the stream and the file. Do not use the value of *`stream`* in subsequent expressions.

feof

SYNTAX

```
int feof(FILE *stream);
```

DESCRIPTION

The function returns a non-zero value if the end-of-file indicator is set for the stream *`stream`*.

ferror

SYNTAX

```
int ferror(FILE *stream);
```

DESCRIPTION

The function returns a non-zero value if the error indicator is set for the stream *`stream`*.

fflush

SYNTAX

```
int fflush(FILE *stream);
```

DESCRIPTION

The function writes any buffered output to the file associated with the stream *`stream`* and returns zero if successful; otherwise, it returns `EOF`. If *`stream`* is a null pointer, `fflush` writes any buffered output to all files opened for output.

fgetc

SYNTAX

```
int fgetc(FILE *stream);
```

DESCRIPTION

The function reads the next character `c` (if present) from the input stream *`stream`*, advances the file-position indicator (if defined), and returns `(int)(unsigned char)c`. If the function sets either the end-of-file indicator or the error indicator, it returns `EOF`.

fgetpos

SYNTAX

```
int fgetpos(FILE *stream, fpos_t *pos);
```

DESCRIPTION

The function stores the file-position indicator for the stream *`stream`* in *`*pos`* and returns zero if successful; otherwise, the function stores a positive value in `errno` and returns a non-zero value.

fgets

SYNTAX

```
char *fgets(char *s, int n, FILE *stream);
```

DESCRIPTION

The function reads characters from the input stream *`stream`* and stores them in successive elements of the array beginning at *`s`* and continuing until it stores *`n`*`-1` characters, stores an `NL` character, or sets the end-of-file or error indicators. If `fgets` stores any characters, it concludes by storing a null character in the next element of the array. It returns *`s`* if it stores any characters and it has not set the error indicator for the stream; otherwise, it returns a null pointer. If it sets the error indicator, the array contents are indeterminate.

fopen

SYNTAX

`FILE *fopen(const char *filename, const char *mode);`

DESCRIPTION

The function opens the file with the filename *`filename`*, associates it with a stream, and returns a pointer to the object controlling the stream. If the open fails, it returns a null pointer. The initial characters of *`mode`* determine how the program manipulates the stream and whether it interprets the stream as text or binary. The initial characters must be one of the following sequences:

- `"r"` to open an existing text file for reading.
- `"w"` to create a text file or to open and truncate an existing text file for writing.
- `"a"` to create a text file or to open an existing text file for writing. The file-position indicator is positioned at the end of the file before each write.
- `"rb"` to open an existing binary file for reading.
- `"wb"` to create a binary file or to open and truncate an existing binary file for writing.
- `"ab"` to create a binary file or to open an existing binary file for writing. The file-position indicator is positioned at the end of the file (possibly after an arbitrary null-byte padding) before each write.
- `"r+"` to open an existing text file for reading and writing.
- `"w+"` to create a text file or to open and truncate an existing text file for reading and writing.
- `"a+"` to create a text file or to open an existing text file for reading and writing. The file-position indicator is positioned at the end of the file before each write.
- `"r+b"` or `"rb+"` to open an existing binary file for reading and writing.
- `"w+b"` or `"wb+"` to create a binary file or to open and truncate an existing binary file for reading and writing.

- `"a+b"` or `"ab+"` to create a binary file or to open an existing binary file for reading and writing. The file-position indicator is positioned at the end of the file (possibly after an arbitrary null-byte padding) before each write.

If you open a file for both reading and writing, the target environment can open a binary file instead of a text file. If the file is not interactive, the stream is fully buffered.

fpos_t

SYNTAX

```
typedef o-type fpos_t;
```

DESCRIPTION

The type is an object type `o-type` of an object that you declare to hold the value of a file-position indicator stored by `fsetpos` and accessed by `fgetpos`.

fprintf

SYNTAX

```
int fprintf(FILE *stream, const char *format, ...);
```

DESCRIPTION

The function generates formatted text, under the control of the format *`format`* and any additional arguments, and writes each generated character to the stream *`stream`*. It returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

fputc

SYNTAX

```
int fputc(int c, FILE *stream);
```

DESCRIPTION

The function writes the character (unsigned char)*`c`* to the output stream *`stream`*, advances the file-position indicator (if defined), and returns `(int)(unsigned char)`*`c`*. If the function sets the error indicator for the stream, it returns `EOF`.

fputs

SYNTAX

```
int fputs(const char *s, FILE *stream);
```

DESCRIPTION

The function accesses characters from the C string `s` and writes them to the output stream `stream`. The function does not write the terminating null character. It returns a non-negative value if it has not set the error indicator; otherwise, it returns `EOF`.

fread

SYNTAX

```
size_t fread(void *ptr, size_t size, size_t nelem, FILE
     *stream);
```

DESCRIPTION

The function reads characters from the input stream `stream` and stores them in successive elements of the array whose first element has the address (char *)`ptr` until the function stores `size*nelem` characters or sets the end-of-file or error indicator. It returns `n/size`, where `n` is the number of characters it read. If `n` is not a multiple of `size`, the value stored in the last element is indeterminate. If the function sets the error indicator, the file-position indicator is indeterminate.

freopen

SYNTAX

```
FILE *freopen(const char *filename, const char *mode,
     FILE *stream);
```

DESCRIPTION

The function closes the file associated with the stream `stream` (as if by calling `fclose`); then it opens the file with the filename `filename` and associates the file with the stream `stream` (as if by calling `fopen(filename, mode)`). It returns `stream` if the open is successful; otherwise, it returns a null pointer.

fscanf

SYNTAX

```
int fscanf(FILE *stream, const char *format, ...);
```

DESCRIPTION

The function scans formatted text, under the control of the format *`format`* and any additional arguments. It obtains each scanned character from the stream *`stream`*. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicator for the stream.

fseek

SYNTAX

```
int fseek(FILE *stream, long offset, int mode);
```

DESCRIPTION

The function sets the file-position indicator for the stream *`stream`* (as specified by *`offset`* and *`mode`*), clears the end-of-file indicator for the stream, and returns zero if successful.

For a binary stream, *`offset`* is a signed offset in bytes:

- If mode has the value `SEEK_SET`, `fseek` adds *`offset`* to the file-position indicator for the beginning of the file.
- If mode has the value `SEEK_CUR`, `fseek` adds *`offset`* to the current file-position indicator.
- If mode has the value `SEEK_END`, `fseek` adds *`offset`* to the file-position indicator for the end of the file (possibly after arbitrary null character padding).

`fseek` sets the file-position indicator to the result of this addition.

For a text stream:

- If mode has the value `SEEK_SET`, `fseek` sets the file-position indicator to the value encoded in *`offset`*, which is either a value returned by an earlier successful call to `ftell` or zero to indicate the beginning of the file.
- If mode has the value `SEEK_CUR` and *`offset`* is zero, `fseek` leaves the file-position indicator at its current value.

- If mode has the value `SEEK_END` and *`offset`* is zero, `fseek` sets the file-position indicator to indicate the end of the file.

The function defines no other combination of argument values.

fsetpos

SYNTAX

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

DESCRIPTION

The function sets the file-position indicator for the stream *`stream`* to the value stored in `*`*`pos`*, clears the end-of-file indicator for the stream, and returns zero if successful. Otherwise, the function stores a positive value in `errno` and returns a non-zero value.

ftell

SYNTAX

```
long ftell(FILE *stream);
```

DESCRIPTION

The function returns an encoded form of the file-position indicator for the stream *`stream`* or stores a positive value in `errno` and returns the value -1. For a binary file, a successful return value gives the number of bytes from the beginning of the file. For a text file, target environments can vary on the representation and range of encoded file-position indicator values.

fwrite

SYNTAX

```
size_t fwrite(const void *ptr, size_t size, size_t nelem,
     FILE *stream);
```

DESCRIPTION

The function writes characters to the output stream *`stream`*, accessing values from successive elements of the array whose first element has the address (`char *`)*`ptr`* until the function writes *`size`*`*`*`nelem`* characters or sets the error indicator. It returns *`n`*`/`*`size`*, where *`n`* is the number of characters it wrote. If the function sets the error indicator, the file-position indicator is indeterminate.

getc

SYNTAX

```
int getc(FILE *stream);
```

DESCRIPTION

The function has the same effect as `fgetc(`*`stream`*`)` except that a macro version of `getc` can evaluate *`stream`* more than once.

getchar

SYNTAX

```
int getchar(void);
```

DESCRIPTION

The function has the same effect as `fgetc(stdin)`, reading a character from the stream `stdin`.

gets

SYNTAX

`char *gets(char *s);`

DESCRIPTION

The function reads characters from the stream `stdin` and stores them in successive elements of the array whose first element has the address *`s`* until the function reads an `NL` character (which is not stored) or sets the end-of-file or error indicator. If `gets` reads any characters, it concludes by storing a null character in the next element of the array. It returns *`s`* if it reads any characters and has not set the error indicator for the stream; otherwise, it returns a null pointer. If it sets the error indicator, the array contents are indeterminate. The number of characters that `gets` reads and stores cannot be limited. Use `fgets` instead.

perror

SYNTAX

`void perror(const char *s);`

DESCRIPTION

The function writes a line of text to the stream `stderr`. If *`s`* is not a null pointer, the function first writes the C string *`s`* (as if by calling `fputs(s, stderr)`), followed by a colon (`:`) and a space. It then writes the same message C string that is returned by `strerror(errno)`, converting the value stored in `errno`, followed by an `NL`.

printf

SYNTAX

`int printf(const char *format, ...);`

DESCRIPTION

The function generates formatted text, under the control of the format *`format`* and any additional arguments, and writes each generated character to the stream `stdout`. It returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

putc

SYNTAX

```
int putc(int c, FILE *stream);
```

DESCRIPTION

The function has the same effect as `fputc(c, stream)` except that a macro version of `putc` can evaluate *`stream`* more than once.

putchar

SYNTAX

```
int putchar(int c);
```

DESCRIPTION

The function has the same effect as `fputc(c, stdout)`, writing a character to the stream `stdout`.

puts

SYNTAX

```
int puts(const char *s);
```

DESCRIPTION

The function accesses characters from the C string *`s`* and writes them to the stream `stdout`. The function writes an `NL` character to the stream in place of the terminating null character. It returns a non-negative value if it has not set the error indicator; otherwise, it returns `EOF`.

remove

SYNTAX

```
int remove(const char *filename);
```

DESCRIPTION

The function removes the file with the filename *`filename`* and returns zero if successful. If the file is open when you remove it, the result is implementation defined. After you remove it, you cannot open it as an existing file.

rename

SYNTAX

`int rename(const char *old, const char *new);`

DESCRIPTION

The function renames the file with the filename `old` to have the filename `new` and returns zero if successful. If a file with the filename `new` already exists, the result is implementation defined. After you rename it, you cannot open the file with the filename `old`.

rewind

SYNTAX

`void rewind(FILE *stream);`

DESCRIPTION

The function calls `fseek(stream, 0L, SEEK_SET)` and then clears the error indicator for the stream `stream`.

scanf

SYNTAX

`int scanf(const char *format, ...);`

DESCRIPTION

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stdin`. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicators for the stream.

setbuf

SYNTAX

```
void setbuf(FILE *stream, char *buf);
```

DESCRIPTION

If *`buf`* is not a null pointer, the function calls `setvbuf(`*`stream`*`, `*`buf`*`, _IOFBF, BUFSIZ)`, specifying full buffering with `_IOFBF` and a buffer size of `BUFSIZ` characters. Otherwise, the function calls `setvbuf(`*`stream`*`, 0, _IONBF, BUFSIZ)`, specifying no buffering with `_IONBF`.

setvbuf

SYNTAX

```
int setvbuf(FILE *stream, char *buf, int mode, size_t
     size);
```

DESCRIPTION

The function sets the buffering mode for the stream *`stream`* according to *`buf`*, *`mode`*, and *`size`*. It returns zero if successful. If *`buf`* is not a null pointer, then *`buf`* is the address of the first element of an array of char of size *`size`* that can be used as the stream buffer. Otherwise, `setvbuf` can allocate a stream buffer that is freed when the file is closed. For *`mode`* you must supply one of the following values:

- `_IOFBF` to indicate full buffering
- `_IOLBF` to indicate line buffering
- `_IONBF` to indicate no buffering

You must call `setvbuf` after you call `fopen` to associate a file with that *`stream`* and before you call a library function that performs any other operation on the *`stream`*.

size_t

DEFINITION

```
typedef ui-type size_t;
```

DESCRIPTION

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

sprintf

SYNTAX

```
int sprintf(char *s, const char *format, ...);
```

DESCRIPTION

The function generates formatted text, under the control of the format *`format`* and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address *`s`*. The function concludes by storing a null character in the next location of the array. It returns the number of characters generated—not including the null character.

sscanf

SYNTAX

```
int sscanf(const char *s, const char *format, ...);
```

DESCRIPTION

The function scans formatted text, under the control of the format *`format`* and any additional arguments. It accesses each scanned character from successive locations of the array object whose first element has the address *`s`*. It returns the number of items matched and assigned, or it returns `EOF` if the function does not store values before it accesses a null character from the array.

stderr

DEFINITION

```
#define stderr pointer_to_FILE_rvalue
```

DESCRIPTION

The macro yields a pointer to the object that controls the standard error output stream.

stdin

DEFINITION

`#define stdin` *`pointer_to_FILE_rvalue`*

DESCRIPTION

The macro yields a pointer to the object that controls the standard input stream.

stdout

DEFINITION

`#define stdout` *`pointer_to_FILE_rvalue`*

DESCRIPTION

The macro yields a pointer to the object that controls the standard output stream.

tmpfile

SYNTAX

```
FILE *tmpfile(void)
```

DESCRIPTION

The function creates a temporary binary file with the filename *`temp-name`* and then has the same effect as calling `fopen(`*`temp-name`*`, "wb+")`. The file *`temp-name`* is removed when the program closes it, either by calling `fclose` explicitly or at normal program termination. The filename *`temp-name`* does not conflict with any filenames that you create. If the open is successful, the function returns a pointer to the object controlling the stream; otherwise, it returns a null pointer.

tmpnam

SYNTAX

`char *tmpnam(char *s);`

DESCRIPTION

The function creates a unique filename *`temp-name`* and returns a pointer to the filename. If *`s`* is not a null pointer, then *`s`* must be the address of the first element of an array at least of size `L_tmpnam`. The function stores *`temp-name`* in the array and returns *`s`*. Otherwise, if *`s`* is a null pointer, the function stores *`temp-name`* in a static-duration array and returns the address of its first element. Subsequent calls to `tmpnam` can alter the values stored in this array.

The function returns unique filenames for each of the first `TMP_MAX` times it is called, after which its behavior is implementation defined. The filename *`temp-name`* does not conflict with any filenames that you create.

ungetc

SYNTAX

`int ungetc(int c, FILE *stream);`

DESCRIPTION

If *`c`* is not equal to `EOF`, the function stores (`unsigned char`)*`c`* in the object whose address is *`stream`* and clears the end-of-file indicator. If *`c`* equals `EOF` or the store cannot occur, the function returns `EOF`; otherwise, it returns (`unsigned char`)*`c`*. A subsequent library function call that reads a character from the stream *`stream`* obtains this stored value, which is then discarded.

Thus, you can effectively push back a character to a stream after reading a character. (You need not push back the same character that you read.) An implementation can let you push back additional characters before you read the first one. You read the characters in reverse order of pushing them back to the stream. You cannot portably:

- Push back more than one character.
- Push back a character if the file-position indicator is at the beginning of the file.
- Call `ftell` for a text file that has a character currently pushed back.

A call to the functions `fseek`, `fsetpos`, or `rewind` for the stream causes the stream to forget any pushed-back characters. For a binary stream, the file-position indicator is decremented for each character that is pushed back.

vfprintf

SYNTAX

```
int vfprintf(FILE *stream, const char *format,
     va_list ap);
```

DESCRIPTION

The function generates formatted text, under the control of the format *`format`* and any additional arguments, and writes each generated character to the stream *`stream`*. It returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by *`ap`*. The program must execute the macro `va_start` before it calls the function and the macro `va_end` after the function returns.

vprintf

SYNTAX

```
int vprintf(const char *format, va_list ap);
```

DESCRIPTION

The function generates formatted text, under the control of the format *`format`* and any additional arguments, and writes each generated character to the stream `stdout`. It returns the number of characters generated, or a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by *`ap`*. The program must execute the macro `va_start` before it calls the function and the macro `va_end` after the function returns.

vsprintf

SYNTAX

```
int vsprintf(char *s, const char *format, va_list ap);
```

DESCRIPTION

The function generates formatted text, under the control of the format *`format`* and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address *`s`*. The function concludes by storing a null character in the next location of the array. It returns the number of characters generated—not including the null character.

The function accesses additional arguments by using the context information designated by *`ap`*. The program must execute the macro `va_start` before it calls the function and the macro `va_end` after the function returns.

GENERAL UTILITIES – stdlib.h

This chapter describes the standard header file `stlib.h`.

DESCRIPTION

Include the standard header `stdlib.h` to declare an assortment of useful functions and to define the macros and types that help you use them.

SUMMARY

The `stdlib.h` header file contains the following functions and macro definitions:

```
#define EXIT_FAILURE rvalue_integer_expression
#define EXIT_SUCCESS rvalue_integer_expression
#define MB_CUR_MAX rvalue_integer_expression
#define NULL null_pointer_constant
#define RAND_MAX integer_constant_expression

void abort(void);
int abs(int i);
int atexit(void (*func)(void));
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
void *bsearch(const void *key, const void *base,
     size_t nelem, size_t size, int (*cmp)(const
     void *ck, const void *ce));
void *calloc(size_t nelem, size_t size);
div_t div(int numer, int denom);
typedef T div_t;
void exit(int status);

void free(void *ptr);
char *getenv(const char *name);
long labs(long i);
ldiv_t ldiv(long numer, long denom);
typedef T ldiv_t;
void *malloc(size_t size);
int mblen(const char *s, size_t n);
size_t mbstowcs(wchar_t *wcs, const char *s, size_t n);
```

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
void qsort(void *base, size_t nelem, size_t size,
     int (*cmp)(const void *e1, const void *e2));
int rand(void);
void *realloc(void *ptr, size_t size);
typedef ui-type size_t;
void srand(unsigned int seed);
double strtod(const char *s, char **endptr);
long strtol(const char *s, char **endptr, int base);
unsigned long strtoul(const char *s, char **endptr,
     int base);
int system(const char *s);
typedef i-type wchar_t;
size_t wcstombs(char *s, const wchar_t *wcs, size_t n);
int wctomb(char *s, wchar_t wchar);
```

Below each definition and function is described.

EXIT_FAILURE

DEFINITION

`#define EXIT_FAILURE` *`rvalue_integer_expression`*

DESCRIPTION

The macro yields the value of the status argument to `exit` which reports unsuccessful termination.

EXIT_SUCCESS

DEFINITION

`#define EXIT_SUCCESS` *`rvalue_integer_expression`*

DESCRIPTION

The macro yields the value of the status argument to `exit` which reports successful termination.

MB_CUR_MAX

DEFINITION

`#define MB_CUR_MAX` *`rvalue_integer_expression`*

where

`rvalue_integer_expression` `>= 1`

DESCRIPTION

The macro yields the maximum number of characters that constitute a multibyte character in the current locale. Its value is `< = MB_LEN_MAX`.

NULL

DEFINITION

`#define NULL` *`null_pointer_constant`*

where

`null_pointer_constant` is either `0`, `0L`, or `(void *)0`

DESCRIPTION

The macro yields a null pointer constant that is usable as an address constant expression.

RAND_MAX

DEFINITION

`#define RAND_MAX` *`integer_constant_expression`*

where

`integer_constant_expression` `>= 32,767`

DESCRIPTION

The macro yields the maximum value returned by `rand`.

abort

SYNTAX

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

DESCRIPTION

The function calls `raise(SIGABRT)`, which reports the abort signal, `SIGABRT`. Default handling for the abort signal is to cause abnormal program termination and report unsuccessful termination to the target environment. Whether or not the target environment flushes output streams, closes open files, or removes temporary files on abnormal termination is implementation defined. If you specify handling that causes raise to return control to `abort`, the function calls `exit(EXIT_FAILURE)`, to report unsuccessful termination with `EXIT_FAILURE`. `abort` never returns control to its caller.

abs

SYNTAX

```
int abs(int i);
```

DESCRIPTION

The function returns the absolute value of *`i`*.

atexit

SYNTAX

```
int atexit(void (*func)(void));
```

DESCRIPTION

The function registers the function whose address is *`func`* to be called by `exit` (or when `main` returns) and returns zero if successful. The functions are called in reverse order of registry. You can register at least 32 functions.

atof

SYNTAX

`double atof(const char *s);`

DESCRIPTION

The function converts the initial characters of the string *s* to an equivalent value *x* of type `double` and then returns *x*. The conversion is the same as for `strtod(s, 0)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

atoi

SYNTAX

`int atoi(const char *s);`

DESCRIPTION

The function converts the initial characters of the string *s* to an equivalent value *x* of type `int` and then returns *x*. The conversion is the same as for `(int)strtol(s, 0, 10)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

atol

SYNTAX

`long atol(const char *s);`

DESCRIPTION

The function converts the initial characters of the string *s* to an equivalent value *x* of type `long` and then returns *x*. The conversion is the same as for `strtol(s, 0, 10)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

bsearch

SYNTAX

```
void *bsearch(const void *key, const void *base,
     size_t nelem, size_t size, int (*cmp)(const
     void *ck, const void *ce));
```

DESCRIPTION

The function searches an array of ordered values and returns the address of an array element that equals the search key `key` (if one exists); otherwise, it returns a null pointer. The array consists of `nelem` elements, each of size bytes, beginning with the element whose address is `base`.

`bsearch` calls the comparison function whose address is `cmp` to compare the search key with elements of the array. The comparison function must return:

- a negative value if the search key `ck` is less than the array element `ce`
- zero if the two are equal
- a positive value if the search key is greater than the array element

`bsearch` assumes that the array elements are in ascending order according to the same comparison rules that are used by the comparison function.

calloc

SYNTAX

```
void *calloc(size_t nelem, size_t size);
```

DESCRIPTION

The function allocates an array object containing `nelem` elements each of size `size`, stores zeros in all bytes of the array, and returns the address of the first element of the array if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.

div

SYNTAX

```
div_t div(int numer, int denom);
```

DESCRIPTION

The function divides *`numer`* by *`denom`* and returns both quotient and remainder in the structure `div_t` result *x*, if the quotient can be represented. The structure member *x*`.quot` is the algebraic quotient truncated toward zero. The structure member *x*`.rem` is the remainder, such that `numer == x.quot*denom + x.rem`.

div_t

DEFINITION

```
typedef struct {
    int quot, rem;
    } div_t;
```

DESCRIPTION

The type is the structure type returned by the function `div`. The structure contains members that represent the quotient (`quot`) and remainder (`rem`) of a signed integer division with operands of type `int`. The members shown above can occur in either order.

exit

SYNTAX

```
void exit(int status);
```

DESCRIPTION

The function calls all functions registered by `atexit`, closes all files, and returns control to the target environment. If status is zero or `EXIT_SUCCESS`, the program reports successful termination. If status is `EXIT_FAILURE`, the program reports unsuccessful termination. An implementation can define additional values for status.

free

SYNTAX

`void free(void *ptr);`

DESCRIPTION

If *ptr* is not a null pointer, the function deallocates the object whose address is *ptr*; otherwise, it does nothing. You can deallocate only objects that you first allocate by calling `calloc`, `malloc`, or `realloc`.

getenv

SYNTAX

`char *getenv(const char *name);`

DESCRIPTION

The function searches an environment list, which each implementation defines, for an entry whose name matches the string *name*. If the function finds a match, it returns a pointer to a static-duration object that holds the definition associated with the target environment *name*. Otherwise, it returns a null pointer. Do not alter the value stored in the object. If you call `getenv` again, the value stored in the object can change. A target environment name is not required by all environments.

labs

SYNTAX

`long labs(long i);`

DESCRIPTION

The function returns the absolute value of *i*, the same as `abs`.

ldiv

SYNTAX

```
ldiv_t ldiv(long numer, long denom);
```

DESCRIPTION

The function divides numer by `denom` and returns both quotient and remainder in the structure `ldiv_t` result `x`, if the quotient can be represented. The structure member `x.quot` is the algebraic quotient truncated toward zero. The structure member `x.rem` is the remainder, such that `numer == x.quot*denom + x.rem`.

ldiv_t

DEFINITION

```
typedef struct {
    long quot, rem;
    } ldiv_t;
```

DESCRIPTION

The type is the structure type returned by the function `ldiv`. The structure contains members that represent the quotient (`quot`) and remainder (`rem`) of a signed integer division with operands of type `long`. The members shown in the definition above can occur in either order.

malloc

SYNTAX

```
void *malloc(size_t size);
```

DESCRIPTION

The function allocates an object of size `size`, and returns the address of the object if successful; otherwise, it returns a null pointer. The values stored in the object are indeterminate. You can safely convert the return value to an object pointer of any type whose size is not greater than `size`.

mblen

SYNTAX

```
int mblen(const char *s, size_t n);
```

DESCRIPTION

If *s* is not a null pointer, the function returns the number of bytes in the multibyte string *s* that constitute the next multibyte character, or it returns -1 if the next *n* (or the remaining) bytes do not constitute a valid multibyte character. `mblen` does not include the terminating null in the count of bytes. The function can use a conversion state stored in an internal static-duration object to determine how to interpret the multibyte string.

If *s* is a null pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its internal static-duration object and returns non-zero; otherwise, it returns zero.

mbstowcs

SYNTAX

```
size_t mbstowcs(wchar_t *wcs, const char *s, size_t n);
```

DESCRIPTION

The function stores a wide character string, in successive elements of the array whose first element has the address *wcs*, by converting, in turn, each of the multibyte characters in the multibyte string *s*. The string begins in the initial conversion state. The function converts each character as if by calling `mbtowc` (except that the internal conversion state stored for that function is unaffected). It stores at most *n* wide characters, stopping after it stores a null wide character. It returns the number of wide characters it stores, not counting the null wide character, if all conversions are successful; otherwise, it returns -1.

mbtowc

SYNTAX

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

DESCRIPTION

If *s* is not a null pointer, the function determines *x*, the number of bytes in the multibyte string *s* that constitute the next multibyte character. (*x* cannot be greater than `MB_CUR_MAX`.) If *pwc* is not a null pointer, the function converts the next multibyte character to its corresponding wide-character value and stores that value in **pwc*. It then returns *x*, or it returns -1 if the next *n* or the remaining bytes do not constitute a valid multibyte character. `mbtowc` does not include the terminating null in the count of bytes. The function can use a conversion state stored in an internal static-duration object to determine how to interpret the multibyte string.

If *s* is a null pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its internal static-duration object and returns non-zero; otherwise, it returns zero.

qsort

SYNTAX

```
void qsort(void *base, size_t nelem, size_t size,
     int (*cmp)(const void *e1, const void *e2));
```

DESCRIPTION

The function sorts, in place, an array consisting of *nelem* elements, each of *size* bytes, beginning with the element whose address is *base*. It calls the comparison function whose address is *cmp* to compare pairs of elements. The comparison function must return a negative value if *e1* is less than *e2*, zero if the two are equal, or a positive value if *e1* is greater than *e2*. Two array elements that are equal can appear in the sorted array in either order.

rand

SYNTAX

```
int rand(void);
```

DESCRIPTION

The function computes a pseudo-random number *x* based on a seed value stored in an internal static-duration object, alters the stored seed value, and returns *x* which is in the interval `[0, RAND_MAX]`.

realloc

SYNTAX

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

DESCRIPTION

The function allocates an object of size *`size`*, possibly obtaining initial stored values from the object whose address is *`ptr`*. It returns the address of the new object if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size is not greater than *`size`*.

If *`ptr`* is not a null pointer, it must be the address of an existing object that you first allocate by calling `calloc`, `malloc`, or `realloc`. If the existing object is not larger than the newly allocated object, `realloc` copies the entire existing object to the initial part of the allocated object. (The values stored in the remainder of the object are indeterminate.) Otherwise, the function copies only the initial part of the existing object that fits in the allocated object. If `realloc` succeeds in allocating a new object, it deallocates the existing object. Otherwise, the existing object is left unchanged.

If *`ptr`* is a null pointer, the function does not store initial values in the newly created object.

size_t

DEFINITION

```
typedef ui-type size_t;
```

DESCRIPTION

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

srand

SYNTAX

```
void srand(unsigned int seed);
```

DESCRIPTION

The function stores the seed value *`seed`* in a static-duration object that `rand` uses to compute a pseudo-random number. From a given *`seed`* value, that function always generates the same sequence of return values. The program behaves as if the target environment calls `srand(1)` at program startup.

strtod

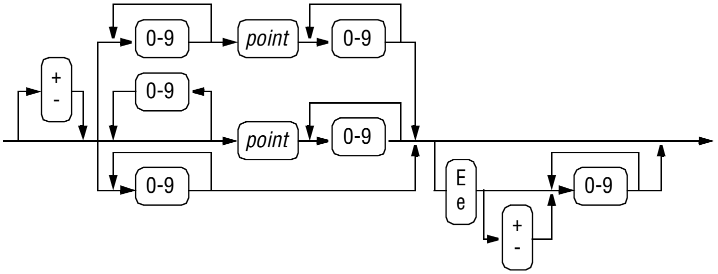
SYNTAX

```
double strtod(const char *s, char **endptr);
```

DESCRIPTION

The function converts the initial characters of the string *`s`* to an equivalent value *`x`* of type `double`. If *`endptr`* is not a null pointer, the function stores a pointer to the unconverted remainder of the string in `*`*`endptr`*. The function then returns *`x`*.

The initial characters of the string *`s`* must consist of zero or more characters for which `isspace` returns non-zero, followed by the longest sequence of one or more characters that match the pattern for `strtod` shown in the diagram.



Here, a *`point`* is the decimal-point character for the current locale. (It is the dot (.) in the "C" locale.) If the string *`s`* matches this pattern, its equivalent value is the decimal integer represented by any digits to the left of the point, plus the decimal fraction represented by any digits to the right of the point, times 10 raised to the signed decimal integer power that follows an optional `e` or `E`. A leading minus sign negates the value. In locales other than the "C" locale, `strtod` can define additional patterns as well.

If the string *s* does not match a valid pattern, the value stored in `*`*endptr* is *s*, and *x* is zero. If a range error occurs, `strtod` behaves exactly as the functions declared in `math.h`.

---

strtol

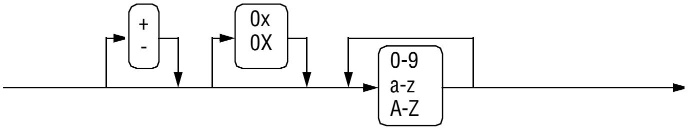
SYNTAX

```
long strtol(const char *s, char **endptr, int base);
```

DESCRIPTION

The function converts the initial characters of the string *s* to an equivalent value *x* of type `long`. If *endptr* is not a null pointer, it stores a pointer to the unconverted remainder of the string in `*`*endptr*. The function then returns *x*.

The initial characters of the string *s* must consist of zero or more characters for which `isspace` returns non-zero, followed by the longest sequence of one or more characters that match the pattern for `strtol` shown in the diagram.



The function accepts the sequences `0x` or `0X` only when *base* equals zero or 16. The letters `a-z` or `A-Z` represent digits in the range `[10, 36]`. If *base* is in the range `[2, 36]`, the function accepts only digits with values less than *base*. If `base == 0`, then a leading `0x` or `0X` (after any sign) indicates a hexadecimal (base 16) integer, a leading `0` indicates an octal (base 8) integer, and any other valid pattern indicates a decimal (base 10) integer.

If the string *s* matches this pattern, its equivalent value is the signed integer of the appropriate base represented by the digits that match the pattern. (A leading minus sign negates the value.) In locales other than the "C" locale, `strtol` can define additional patterns as well.

If the string *s* does not match a valid pattern, the value stored in `*`*`endptr`* is *s*, and *x* is zero. If the equivalent value is too large to represent as type `long`, `strtol` stores the value of `ERANGE` in `errno` and returns either `LONG_MAX`, if *x* is positive, or `LONG_MIN`, if *x* is negative.

---

strtoul

SYNTAX

```
unsigned long strtoul(const char *s, char **endptr, int
     base);
```

DESCRIPTION

The function converts the initial characters of the string *s* to an equivalent value x of type `unsigned long`. If *endptr* is not a null pointer, it stores a pointer to the unconverted remainder of the string in `*`*`endptr`*. The function then returns *x*.

`strtoul` converts strings exactly as does `strtol`, but reports a range error only if the equivalent value is too large to represent as type `unsigned long`. In this case, `strtoul` stores the value of `ERANGE` in `errno` and returns `ULONG_MAX`.

---

system

SYNTAX

```
int system(const char *s);
```

DESCRIPTION

If *s* is not a null pointer, the function passes the string *s* to be executed by a command processor, supplied by the target environment, and returns the status reported by the command processor. If *s* is a null pointer, the function returns non-zero only if the target environment supplies a command processor. Each implementation defines what strings its command processor accepts.

wchar_t

DEFINITION

`typedef i-type wchar_t;`

DESCRIPTION

The type is the integer type `i-type` of a wide-character constant, such as `L'X'`. You declare an object of type `wchar_t` to hold a wide character.

wcstombs

SYNTAX

`size_t wcstombs(char *s, const wchar_t *wcs, size_t n);`

DESCRIPTION

The function stores a multibyte string, in successive elements of the array whose first element has the address *s*, by converting in turn each of the wide characters in the string *wcs*. The multibyte string begins in the initial conversion state. The function converts each wide character as if by calling `wctomb` (except that the conversion state stored for that function is unaffected). It stores no more than *n* bytes, stopping after it stores a null byte. It returns the number of bytes it stores, not counting the null byte, if all conversions are successful; otherwise, it returns -1.

wctomb

SYNTAX

`int wctomb(char *s, wchar_t wchar);`

DESCRIPTION

If *s* is not a null pointer, the function determines *x*, the number of bytes needed to represent the multibyte character corresponding to the wide character `wchar`. *x* cannot exceed `MB_CUR_MAX`. The function converts `wchar` to its corresponding multibyte character, which it stores in successive elements of the array whose first element has the address *s*. It then returns *x*, or it returns -1 if `wchar` does not correspond to a valid multibyte character. `wctomb` includes the terminating null byte in the count of bytes. The function can use a conversion state stored in a static-duration object to determine how to interpret the multibyte character string.

If *s* is a null pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its static-duration object and returns non-zero; otherwise, it returns zero.

STRING HANDLING – string.h

This chapter describes the standard header file `string.h`.

DESCRIPTION

Include the standard header `string.h` to declare a number of functions that help you manipulate C strings and other arrays of characters.

SUMMARY

The `string.h` header file contains the following functions and macro definitions:

```
#define NULL null_pointer_constant

void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);

typedef ui-type size_t;
char *strcat(char *s1, const char *s2);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
char *strcpy(char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);
char *strerror(int errcode);
size_t strlen(const char *s);

char *strncat(char *s1, const char *s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Below each definition and function is described.

NULL

DEFINITION

`#define NULL` *`null_pointer_constant`*

where

`null_pointer_constant` is either `0`, `0L`, or `(void *)0`

DESCRIPTION

The macro yields a null pointer constant that is usable as an address constant expression.

memchr

SYNTAX

```
void *memchr(const void *s, int c, size_t n);
```

DESCRIPTION

The function searches for the first element of an array of `unsigned char`, beginning at the address *`s`* with size *`n`*, that equals `(unsigned char)`*`c`*. If successful, it returns the address of the matching element; otherwise, it returns a null pointer.

memcmp

SYNTAX

```
int memcmp(const void *s1, const void *s2, size_t n);
```

DESCRIPTION

The function compares successive elements from two arrays of `unsigned char`, beginning at the addresses *`s1`* and *`s2`* (both of size *`n`*), until it finds elements that are not equal:

- If all elements are equal, the function returns zero.
- If the differing element from *`s1`* is greater than the element from *`s2`*, the function returns a positive number.
- Otherwise, the function returns a negative number.

memcpy

SYNTAX

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

DESCRIPTION

The function copies the array of `char` beginning at the address `s2` to the array of `char` beginning at the address `s1` (both of size `n`). It returns `s1`. The elements of the arrays can be accessed and stored in any order.

memmove

SYNTAX

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

DESCRIPTION

The function copies the array of `char` beginning at `s2` to the array of `char` beginning at `s1` (both of size `n`). It returns `s1`. If the arrays overlap, the function accesses each of the element values from `s2` before it stores a new value in that element, so the copy is not corrupted.

memset

SYNTAX

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

DESCRIPTION

The function stores `(unsigned char)c` in each of the elements of the array of `unsigned char` beginning at `s`, with size `n`. It returns `s`.

size_t

DEFINITION

```
typedef ui-type size_t;
```

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

strcat

SYNTAX

```
char *strcat(char *s1, const char *s2);
```

DESCRIPTION

The function copies the string *s2*, including its terminating null character, to successive elements of the array of `char` that stores the string *s1*, beginning with the element that stores the terminating null character of *s1*. It returns *s1*.

strchr

SYNTAX

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

The function searches for the first element of the string *s* that equals (`char`)*c*. It considers the terminating null character as part of the string. If successful, the function returns the address of the matching element; otherwise, it returns a null pointer.

strcmp

SYNTAX

```
int strcmp(const char *s1, const char *s2);
```

DESCRIPTION

The function compares successive elements from two strings, *s1* and *s2*, until it finds elements that are not equal:

- If all elements are equal, the function returns zero.
- If the differing element from *s1* is greater than the element from *s2* (both taken as `unsigned char`), the function returns a positive number.
- Otherwise, the function returns a negative number.

strcoll

SYNTAX

```
int strcoll(const char *s1, const char *s2);
```

DESCRIPTION

The function compares two strings, *`s1`* and *`s2`*, using a comparison rule that depends on the current locale. If *`s1`* compares greater than *`s2`* by this rule, the function returns a positive number. If the two strings compare equal, it returns zero. Otherwise, it returns a negative number.

strcpy

SYNTAX

```
char *strcpy(char *s1, const char *s2);
```

DESCRIPTION

The function copies the string *`s2`*, including its terminating null character, to successive elements of the array of `char` whose first element has the address *`s1`*. It returns *`s1`*.

strcspn

SYNTAX

```
size_t strcspn(const char *s1, const char *s2);
```

DESCRIPTION

The function searches for the first element `s1[i]` in the string *`s1`* that equals any one of the elements of the string *`s2`* and returns *`i`*. Each terminating null character is considered part of its string.

strerror

SYNTAX

```
char *strerror(int errcode);
```

DESCRIPTION

The function returns a pointer to an internal static-duration object containing the message string corresponding to the error code *`errcode`*. The program must not alter any of the values stored in this object. A later call to `strerror` can alter the value stored in this object.

strlen

SYNTAX

`size_t strlen(const char *s);`

DESCRIPTION

The function returns the number of characters in the string *s*, not including its terminating null character.

strncat

SYNTAX

`char *strncat(char *s1, const char *s2, size_t n);`

DESCRIPTION

The function copies the string *s2*, not including its terminating null character, to successive elements of the array of `char` that stores the string *s1*, beginning with the element that stores the terminating null character of *s1*. The function copies no more than *n* characters from *s2*. It then stores a null character, in the next element to be altered in *s1*, and returns *s1*.

strncmp

SYNTAX

`int strncmp(const char *s1, const char *s2, size_t n);`

DESCRIPTION

The function compares successive elements from two strings, *s1* and *s2*, until it finds elements that are not equal or until it has compared the first *n* elements of the two strings:

- If all elements are equal, the function returns zero.
- If the differing element from *s1* is greater than the element from *s2* (both taken as `unsigned char`), the function returns a positive number.
- Otherwise, it returns a negative number.

strncpy

SYNTAX

`char *strncpy(char *s1, const char *s2, size_t n);`

DESCRIPTION

The function copies the string *s2*, not including its terminating null character, to successive elements of the array of `char` whose first element has the address *s1*. It copies no more than *n* characters from *s2*. The function then stores zero or more null characters in the next elements to be altered in *s1* until it stores a total of *n* characters. It returns *s1*.

strpbrk

SYNTAX

`char *strpbrk(const char *s1, const char *s2);`

DESCRIPTION

The function searches for the first element `s1[i]` in the string *s1* that equals any one of the elements of the string *s2*. It considers each terminating null character as part of its string. If `s1[i]` is not the terminating null character, the function returns `&s1[i]`; otherwise, it returns a null pointer.

strrchr

SYNTAX

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

DESCRIPTION

The function searches for the last element of the string *s* that equals `(char)`*c*. It considers the terminating null character as part of the string. If successful, the function returns the address of the matching element; otherwise, it returns a null pointer.

strspn

SYNTAX

```
size_t strspn(const char *s1, const char *s2);
```

DESCRIPTION

The function searches for the first element `s1[i]` in the string `s1` that equals none of the elements of the string `s2` and returns `i`. It considers the terminating null character as part of the string `s1` only.

strstr

SYNTAX

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

DESCRIPTION

The function searches for the first sequence of elements in the string `s1` that matches the sequence of elements in the string `s2`, not including its terminating null character. If successful, the function returns the address of the matching first element; otherwise, it returns a null pointer.

strtok

SYNTAX

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

DESCRIPTION

If `s1` is not a null pointer, the function begins a search of the string `s1`. Otherwise, it begins a search of the string whose address was last stored in an internal static-duration object on an earlier call to the function, as described below. The search proceeds as follows:

1. The function searches the string for begin, the address of the first element that equals none of the elements of the string `s2` (a set of token separators). It considers the terminating null character as part of the search string only.
2. If the search does not find an element, the function stores the address of the terminating null character in the internal static-duration object (so that a subsequent search beginning with that address will fail) and returns a null pointer. Otherwise, the

function searches from begin for end, the address of the first element that equals any one of the elements of the string *s2*. It again considers the terminating null character as part of the search string only.

3 If the search does not find an element, the function stores the address of the terminating null character in the internal static-duration object. Otherwise, it stores a null character in the element whose address is end. Then it stores the address of the next element after end in the internal static-duration object (so that a subsequent search beginning with that address will continue with the remaining elements of the string) and returns begin.

strxfrm

SYNTAX

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

DESCRIPTION

The function stores a string in the array of `char` whose first element has the address *s1*. It stores no more than *n* characters, including the terminating null character, and returns the number of characters needed to represent the entire string, not including the terminating null character. If the value returned is *n* or greater, the values stored in the array are indeterminate. (If *n* is zero, *s1* can be a null pointer.)

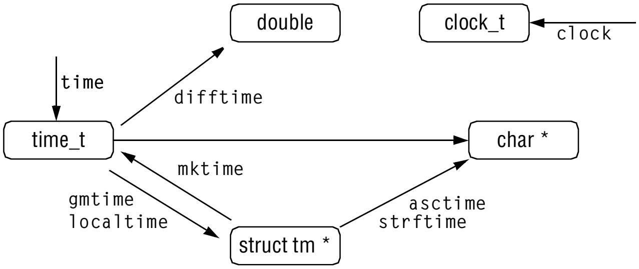
`strxfrm` generates the string it stores from the string *s2* by using a transformation rule that depends on the current locale. For example, if *x* is a transformation of *s1* and *y* is a transformation of *s2*, then `strcmp(x, y)` returns the same value as `strcoll(s1, s2)`.

TIME HANDLING – time.h

This chapter describes the standard header file `time.h`.

INTRODUCTION

Include the standard header `time.h` to declare several functions that help you manipulate times. The diagram summarizes the functions and the object types that they convert between:



The functions share two static-duration objects that hold values computed by the functions:

- a time string of type array of `char`
- a time structure of type `struct tm`

A call to one of these functions can alter the value that was stored earlier in a static-duration object by another of these functions.

SUMMARY

The `time.h` header file contains the following functions and macro definitions:

```
#define CLOCKS_PER_SEC integer_constant_expression
#define NULL null_pointer_constant
char *asctime(const struct tm *tptr);
clock_t clock(void);
typedef a-type clock_t;
char *ctime(const time_t *tod);
double difftime(time_t t1, time_t t0);
struct tm *gmtime(const time_t *tod);
struct tm *localtime(const time_t *tod);
time_t mktime(struct tm *tptr);
```

```
typedef ui-type size_t;
size_t strftime(char *s, size_t n, const char *format,
     const struct tm *tptr);
time_t time(time_t *tod);
typedef a-type time_t;
struct tm;
```

Below each definition and function is described.

CLOCKS_PER_SEC

DEFINITION

`#define CLOCKS_PER_SEC` *`integer_constant_expression`*

where

`integer_constant_expression` `> 0`

DESCRIPTION

The macro yields the number of clock ticks, returned by `clock`, in one second.

NULL

DEFINITION

`#define NULL` *`null_pointer_constant`*

where

`null_pointer_constant` is either `0`, `0L`, or `(void *)0`

DESCRIPTION

The macro yields a null pointer constant that is usable as an address constant expression.

asctime

SYNTAX

```
char *asctime(const struct tm *tptr);
```

DESCRIPTION

The function stores in the static-duration time string a 26-character English-language representation of the time encoded in `*tptr`. It returns the address of the static-duration time string. The text representation takes the form:

```
Sun Dec  2 06:55:15 1979\n\0
```

clock

SYNTAX

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

DESCRIPTION

The function returns the number of clock ticks of elapsed processor time, counting from a time related to program startup, or it returns -1 if the target environment cannot measure elapsed processor time.

clock_t

DEFINITION

```
typedef a-type clock_t;
```

DESCRIPTION

The type is the arithmetic type `a-type` of an object that you declare to hold the value returned by `clock`, representing elapsed processor time.

ctime

SYNTAX

```
char *ctime(const time_t *tod);
```

DESCRIPTION

The function converts the calendar time in `*tod` to a text representation of the local time in the static-duration time string. It returns the address of that string. It is equivalent to `asctime(localtime(tod))`.

difftime

SYNTAX

```
double difftime(time_t t1, time_t t0);
```

DESCRIPTION

The function returns the difference `t1–t0`, in seconds, between the calendar time `t0` and the calendar time `t1`.

gmtime

SYNTAX

```
struct tm *gmtime(const time_t *tod);
```

DESCRIPTION

The function stores in the static-duration time structure an encoding of the calendar time in `*tod`, expressed as Universal Time Coordinated, or UTC. (UTC was formerly Greenwich Mean Time, or GMT). It returns the address of that structure.

localtime

SYNTAX

```
struct tm *localtime(const time_t *tod);
```

DESCRIPTION

The function stores in the static-duration time structure an encoding of the calendar time in `*tod`, expressed as local time. It returns the address of that structure.

mktime

SYNTAX

```
time_t mktime(struct tm *tptr);
```

DESCRIPTION

The function alters the values stored in `*tptr` to represent an equivalent encoded local time, but with the values of all members within their normal ranges. It then determines the values `tptr->wday` and `tptr->yday` from the values of the other members. It returns the calendar time equivalent to the encoded time, or it returns a value of -1 if the calendar time cannot be represented.

size_t

DEFINITION

```
typedef ui-type size_t;
```

DESCRIPTION

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

strftime

SYNTAX

```
size_t strftime(char *s, size_t n, const char *format,
     const struct tm *tptr);
```

DESCRIPTION

The function generates formatted text, under the control of the format *`format`* and the values stored in the time structure `*`*`tptr`*. It stores each generated character in successive locations of the array object of size *`n`* whose first element has the address *`s`*. The function then stores a null character in the next location of the array. It returns *`x`*, the number of characters generated, if *`x`* `<` *`n`*; otherwise, it returns zero, and the values stored in the array are indeterminate.

For each multibyte character other than `%` in the format, the function stores that multibyte character in the array object. Each occurrence of `%` followed by another character in the format is a conversion specifier. For each conversion specifier, the function stores a replacement character sequence.

The following table lists all conversion specifiers defined for `strftime`. An example follow each conversion specifier. All examples are for the "C" locale, using the date and time Sunday, 2 December 1979 at 06:55:15 AM EST.

Conversion specifier	*Description*	*Example*
`%a`	Abbreviated weekday name	`Sun`
`%A`	Full weekday name	`Sunday`
`%b`	Abbreviated month name	`Dec`
`%B`	Full month name	`December`

Conversion specifier	*Description*	*Example*
`%c`	Date and time	`Dec  2 06:55:15 1979`
`%d`	Day of the month	`02`
`%H`	Hour of the 24-hour day	`06`
`%I`	Hour of the 12-hour day	`06`
`%j`	Day of the year, from 001	`335`
`%m`	Month of the year, from 01	`12`
`%M`	Minutes after the hour	`55`
`%p`	AM/PM indicator	`AM`
`%S`	Seconds after the minute	`15`
`%U`	Sunday week of the year, from 00	`48`
`%w`	Day of the week, from 0 for Sunday	`6`
`%W`	Monday week of the year, from 00	`47`
`%x`	Date	`Dec  2 1979`
`%X`	Time	`06:55:15`
`%y`	Year of the century, from 00	`79`
`%Y`	Year	`1979`
`%Z`	Time zone name, if any	`EST`
`%%`	Percent character	`%`

The current locale category `LC_TIME` can affect these replacement character sequences.

time

SYNTAX

```
time_t time(time_t *tod);
```

DESCRIPTION

If `tod` is not a null pointer, the function stores the current calendar time in `*tod`. The function returns the current calendar time, if the target environment can determine it; otherwise, it returns -1.

time_t

DEFINITION

```
typedef a-type time_t;
```

DESCRIPTION

The type is the arithmetic type `a-type` of an object that you declare to hold the value returned by time. The value represents calendar time.

tm

DEFINITION

```
struct tm {
    int tm_sec;         seconds after the minute (from 0)
    int tm_min;         minutes after the hour (from 0)
    int tm_hour;        hour of the day (from 0)
    int tm_mday;        day of the month (from 1)
    int tm_mon;         month of the year (from 0)
    int tm_year;        years since 1900 (from 0)
    int tm_wday;        days since Sunday (from 0)
    int tm_yday;        day of the year (from 0)
    int tm_isdst;       Daylight Saving Time flag
    };
```

DESCRIPTION

`struct tm` contains members that describe various properties of the calendar time. The members shown above can occur in any order, interspersed with additional members. The comment following each member briefly describes its meaning.

The member `tm_isdst` contains:

- a positive value if Daylight Saving Time is in effect
- zero if Daylight Saving Time is not in effect
- a negative value if the status of Daylight Saving Time is not known (so the target environment should attempt to determine its status)

WIDE CHARACTER INPUT/OUTPUT – wchar.h

This chapter describes the standard header file `wchar.h`.

INTRODUCTION

Include the standard header `wchar.h` so that you can perform input and output operations on wide streams or manipulate wide strings.

SUMMARY

The `wchar.h` header file contains the following functions and macro definitions:

```
#define NULL null_pointer_constant
#define WCHAR_MAX #if_expression
#define WCHAR_MIN #if_expression
#define WEOF wint_t_constant_expression

wint_t btowc(int c);
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t *s, FILE *stream);
int fwide(FILE *stream, int mode);
int fwprintf(FILE *stream, const wchar_t *format, ...);
int fwscanf(FILE *stream, const wchar_t *format, ...);
wint_t getwc(FILE *stream);
wint_t getwchar(void);

size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n,
     mbstate_t *ps);
int mbsinit(const mbstate_t *ps);
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t
     len, mbstate_t *ps);
typedef o-type mbstate_t;
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
typedef ui-type size_t;
int swprintf(wchar_t *s, size_t n, const wchar_t *format,
     ...);
```

```
int swscanf(const wchar_t *s, const wchar_t *format,
     ...);
struct tm;
wint_t ungetwc(wint_t c, FILE *stream);
int vfwprintf(FILE *stream, const wchar_t *format,
     va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t
     *format, va_list arg);
int vwprintf(const wchar_t *format, va_list arg);
typedef i-type wchar_t;
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);

size_t wcsftime(wchar_t *s, size_t maxsize, const wchar_t
     *format, const struct tm *timeptr);
size_t wcslen(const wchar_t *s);
wchar_t *wcsncat(wchar_t *s1, const wchar_t *s2, size_t
     n);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t
     n);
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t
     n);
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcsrtombs(char *dst, const wchar_t **src, size_t
     len, mbstate_t *ps);

size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
double wcstod(const wchar_t *nptr, wchar_t **endptr);
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2, wchar_t
     **ptr);
long wcstol(const wchar_t *nptr, wchar_t **endptr, int
     base);
unsigned long wcstoul(const wchar_t *nptr, wchar_t
     **endptr, int base);
size_t wcsxfrm(wchar_t *s1, const wchar_t *s2, size_t n);
```

```
int wctob(wint_t c);
typedef i_type wint_t;
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t
     n);
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t
     n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t
     n);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
int wprintf(const wchar_t *format, ...);
int wscanf(const wchar_t *format, ...);
```

In the following sections each function and macro definition is described.

NULL

DEFINITION

`#define NULL null_pointer_constant`

where

`null_pointer_constant` is either `0`, `0L`, or `(void *)0`.

DESCRIPTION

The macro yields a null pointer constant that is usable as an address constant expression.

WCHAR_MAX

DEFINITION

`#define WCHAR_MAX #if_expression`

where

`#if_expression >= 127`

DESCRIPTION

The macro yields the maximum value for type `wchar_t`.

WCHAR_MIN

DEFINITION

`#define WCHAR_MIN #if_expression`

where

`#if_expression <= 0`

DESCRIPTION

The macro yields the minimum value for type `wchar_t`.

WEOF

DEFINITION

`#define WEOF wint_t_constant_expression`

DESCRIPTION

The macro yields the return value, of type `wint_t`, used to signal the end of a wide stream or to report an error condition.

btowc

SYNTAX

`wint_t btowc(int c);`

DESCRIPTION

The function returns `WEOF` if *c* equals `EOF`. Otherwise, it converts (`unsigned char`)*c* as a one-byte multibyte character beginning in the initial conversion state, as if by calling `mbrtowc`. If the conversion succeeds, the function returns the wide-character conversion. Otherwise, it returns `WEOF`.

fgetwc

SYNTAX

`wint_t fgetwc(FILE *stream);`

DESCRIPTION

The function reads the next wide character *c* (if present) from the input stream *stream*, advances the file-position indicator (if defined), and returns (`wint_t`)*c*. If the function sets either the end-of-file indicator or the error indicator, it returns `WEOF`.

fgetws

SYNTAX

`wchar_t *fgetws(wchar_t *s, int n, FILE *stream);`

DESCRIPTION

The function reads wide characters from the input stream *stream* and stores them in successive elements of the array beginning at *s* and continuing until it stores *n* - 1 wide characters, stores an `NL` wide character, or sets the end-of-file or error indicators. If `fgetws` stores any wide characters, it concludes by storing a null wide character in the next element of the array. It returns *s* if it stores any wide characters and it has not set the error indicator for the stream; otherwise, it returns a null pointer. If it sets the error indicator, the array contents are indeterminate.

fputwc

SYNTAX

`wint_t fputwc(wchar_t c, FILE *stream);`

DESCRIPTION

The function writes the wide character *c* to the output stream *stream*, advances the file-position indicator (if defined), and returns (`wint_t`)*c*. If the function sets the error indicator for the stream, it returns `WEOF`.

fputws

SYNTAX

`int fputws(const wchar_t *s, FILE *stream);`

DESCRIPTION

The function accesses wide characters from the string *s* and writes them to the output stream *stream*. The function does not write the terminating null wide character. It returns a non-negative value if it has not set the error indicator; otherwise, it returns `WEOF`.

fwide

SYNTAX

`int fwide(FILE *stream, int mode);`

DESCRIPTION

The function determines the orientation of the stream *stream*. If mode is greater than zero, it first attempts to make the stream wide oriented. If mode is less than zero, it first attempts to make the stream byte oriented. In any event, the function returns:

- a value greater than zero if the stream is left wide oriented
- zero if the stream is left unbound
- a value less than zero if the stream is left byte oriented

In no event will the function alter the orientation of a stream once it has been oriented.

fwprintf

SYNTAX

`int fwprintf(FILE *stream, const wchar_t *format, ...);`

DESCRIPTION

The function generates formatted text, under the control of the format *format* and any additional arguments, and writes each generated wide character to the stream *stream*. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

fwscanf

SYNTAX

```
int fwscanf(FILE *stream, const wchar_t *format, ...);
```

DESCRIPTION

The function scans formatted text, under the control of the format *`format`* and any additional arguments. It obtains each scanned character from the stream *`stream`*. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicator for the stream.

getwc

SYNTAX

```
wint_t getwc(FILE *stream);
```

DESCRIPTION

The function has the same effect as `fgetwc(`*`stream`*`)` except that a macro version of `getwc` can evaluate *`stream`* more than once.

getwchar

SYNTAX

```
wint_t getwchar(void);
```

DESCRIPTION

The function has the same effect as `fgetwc(stdin)`.

mbrlen

SYNTAX

```
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
```

DESCRIPTION

The function is equivalent to the call:

```
mbrtowc(0, s, n, ps != 0 ? ps : &internal)
```

where *`internal`* is an object of type `mbstate_t` internal to the `mbrlen` function. At program startup, *`internal`* is initialized to the initial conversion state. No other library function alters the value stored in *`internal`*.

The function returns:

- `(size_t)-2` if after converting all *`n`* characters, the resulting conversion state indicates an incomplete multibyte character
- `(size_t)-1` if the function detects an encoding error before completing the next multibyte character, in which case the function stores the value `EILSEQ` in `errno` and leaves the resulting conversion state undefined
- zero if the next completed character is a null character, in which case the resulting conversion state is the initial conversion state
- *`x`* the number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that *`x`* bytes have been converted

Thus, `mbrlen` effectively returns the number of bytes that would be consumed in successfully converting a multibyte character to a wide character (without storing the converted wide character), or an error code if the conversion cannot succeed.

mbrtowc

SYNTAX

```
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n,
     mbstate_t *ps);
```

DESCRIPTION

The function determines the number of bytes in a multibyte string that completes the next multibyte character, if possible.

If *ps* is not a null pointer, the conversion state for the multibyte string is assumed to be **ps*. Otherwise, it is assumed to be &*internal*, where *internal* is an object of type `mbstate_t` internal to the `mbrtowc` function. At program startup, *internal* is initialized to the initial conversion state. No other library function alters the value stored in *internal*.

If *s* is not a null pointer, the function determines *x*, the number of bytes in the multibyte string *s* that complete or contribute to the next multibyte character. (*x* cannot be greater than *n*.) Otherwise, the function effectively returns `mbrtowc(0, "", 1, ps)`, ignoring *pwc* and *n*. (The function thus returns zero only if the conversion state indicates that no incomplete multibyte character is pending from a previous call to `mbrlen`, `mbrtowc`, or `mbsrtowcs` for the same string and conversion state.)

If *pwc* is not a null pointer, the function converts a completed multibyte character to its corresponding wide-character value and stores that value in **pwc*.

The function returns:

- `(size_t)-2`, if after converting all *n* characters, the resulting conversion state indicates an incomplete multibyte character
- `(size_t)-1` if the function detects an encoding error before completing the next multibyte character, in which case the function stores the value `EILSEQ` in `errno` and leaves the resulting conversion state undefined
- zero if the next completed character is a null character, in which case the resulting conversion state is the initial conversion state
- *x*, the number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that *x* bytes have been converted

mbsinit

SYNTAX

```
int mbsinit(const mbstate_t *ps);
```

DESCRIPTION

The function returns a non-zero value if *ps* is a null pointer or if **ps* designates an initial conversion state. Otherwise, it returns zero.

mbsrtowcs

SYNTAX

```
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t
     len, mbstate_t *ps);
```

DESCRIPTION

The function converts the multibyte string beginning at `*src` to a sequence of wide characters as if by repeated calls of the form:

```
x = mbrtowc(dst, *src, n, ps != 0 ? ps : &internal)
```

where *n* is some value > 0 and *`internal`* is an object of type `mbstate_t` internal to the `mbsrtowcs` function. At program startup, *`internal`* is initialized to the initial conversion state. No other library function alters the value stored in *`internal`*.

If *`dst`* is not a null pointer, the `mbsrtowcs` function stores at most *`len`* wide characters by calls to `mbrtowc`. The function effectively increments *`dst`* by one and *`*src`* by *x* after each call to `mbrtowc` that stores a converted wide character. After a call that returns zero, `mbsrtowcs` stores a null wide character at *`dst`* and stores a null pointer at *`*src`*.

If *`dst`* is a null pointer, *`len`* is effectively assigned a large value.

The function returns:

- `(size_t)-1` if a call to `mbrtowc` returns `(size_t)-1`, indicating that it has detected an encoding error before completing the next multibyte character
- the number of multibyte characters successfully converted, not including the terminating null character.

mbstate_t

DEFINITION

```
typedef o-type mbstate_t;
```

DESCRIPTION

The type is an object type `o-type` that can represent a conversion state for any of the functions `mbrlen`, `mbrtowc`, `mbsrtowcs`, `wcrtomb`, or `wcsrtombs`. A definition of the form:

```
mbstate_t mbst = {0};
```

ensures that `mbst` represents the initial conversion state. Note, however, that other values stored in an object of type `mbstate_t` can also represent this state. To test safely for this state, use the function `mbsinit`.

---

putwc

SYNTAX

```
wint_t putwc(wchar_t c, FILE *stream);
```

DESCRIPTION

The function has the same effect as `fputwc(c, stream)` except that a macro version of `putwc` can evaluate *`stream`* more than once.

---

putwchar

SYNTAX

```
wint_t putwchar(wchar_t c);
```

DESCRIPTION

The function has the same effect as `fputwc(c, stdout)`.

---

size_t

DEFINITION

```
typedef ui-type size_t;
```

DESCRIPTION

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

swprintf

SYNTAX

```
int swprintf(wchar_t *s, size_t n, const wchar_t *format,
     ...);
```

DESCRIPTION

The function generates formatted text, under the control of the format *`format`* and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address *`s`*. The function concludes by storing a null wide character in the next location of the array. It returns the number of wide characters generated—not including the null wide character.

swscanf

SYNTAX

```
int swscanf(const wchar_t *s, const wchar_t *format,
     ...);
```

DESCRIPTION

The function scans formatted text, under the control of the format *`format`* and any additional arguments. It accesses each scanned character from successive locations of the array object whose first element has the address *`s`*. It returns the number of items matched and assigned, or it returns `EOF` if the function does not store values before it accesses a null wide character from the array.

tm

DEFINITION

struct tm;

DESCRIPTION

`struct tm` contains members that describe various properties of the calendar time. The declaration in this header leaves `struct tm` an incomplete type. Include the header `time.h` to complete the type.

ungetwc

SYNTAX

```
wint_t ungetwc(wint_t c, FILE *stream);
```

DESCRIPTION

If *c* is not equal to `WEOF`, the function stores `(wchar_t)`*c* in the object whose address is *stream* and clears the end-of-file indicator. If *c* equals `WEOF` or the store cannot occur, the function returns `WEOF`; otherwise, it returns `(wchar_t)`*c*. A subsequent library function call that reads a wide character from the stream *stream* obtains this stored value, which is then discarded.

Thus, you can effectively push back a wide character to a stream after reading a wide character.

vfwprintf

SYNTAX

```
int vfwprintf(FILE *stream, const wchar_t *format,
     va_list arg);
```

DESCRIPTION

The function generates formatted text, under the control of the format *format* and any additional arguments, and writes each generated wide character to the stream *stream*. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function and the macro `va_end` after the function returns.

vswprintf

SYNTAX

```
int vswprintf(wchar_t *s, size_t n, const wchar_t
     *format, va_list arg);
```

DESCRIPTION

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated wide character in successive locations of the array object whose first element has the address `s`. The function concludes by storing a null wide character in the next location of the array. It returns the number of characters generated—not including the null wide character.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function and the macro `va_end` after the function returns.

vwprintf

SYNTAX

```
int vwprintf(const wchar_t *format, va_list arg);
```

DESCRIPTION

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stdout`. It returns the number of characters generated, or a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function and the macro `va_end` after the function returns.

wchar_t

DEFINITION

```
typedef i-type wchar_t;
```

DESCRIPTION

The type is the integer type `i-type` of a wide-character constant, such as `L'X'`. You declare an object of type `wchar_t` to hold a wide character.

wcrtomb

SYNTAX

```
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
```

DESCRIPTION

The function determines the number of bytes needed to represent the wide character *`wc`* as a multibyte character, if possible. (Not all values representable as type `wchar_t` are necessarily valid wide-character codes.)

If *`ps`* is not a null pointer, the conversion state for the multibyte string is assumed to be *`*ps`*. Otherwise, it is assumed to be *`&internal`*, where *`internal`* is an object of type `mbstate_t` internal to the `wcrtomb` function. At program startup, *`internal`* is initialized to the initial conversion state. No other library function alters the value stored in *`internal`*.

If *`s`* is not a null pointer and *`wc`* is a valid wide-character code, the function determines *`x`*, the number of bytes needed to represent *`wc`* as a multibyte character, and stores the converted bytes in the array of char beginning at *`s`*. (*`x`* cannot be greater than `MB_CUR_MAX`.) If *`wc`* is a null wide character, the function stores any shift sequence needed to restore the initial shift state. followed by a null byte. The resulting conversion state is the initial conversion state.

If *`s`* is a null pointer, the function effectively returns `wcrtomb(`*`buf`*`, L'\0', `*`ps`*`)`, where *`buf`* is a buffer internal to the function. (The function thus returns the number of bytes needed to restore the initial conversion state and to terminate the multibyte string pending from a previous call to `wcrtomb` or `wcsrtombs` for the same string and conversion state.)

The function returns:

- `(size_t)-1` if *`wc`* is an invalid wide-character code, in which case the function stores the value `EILSEQ` in `errno` and leaves the resulting conversion state undefined
- *`x`*, the number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that *`x`* bytes have been generated

wcscat

SYNTAX

```
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The function copies the wide string *s2*, including its terminating null wide character, to successive elements of the array that stores the wide string *s1*, beginning with the element that stores the terminating null wide character of *s1*. It returns *s1*.

wcschr

SYNTAX

```
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

DESCRIPTION

The function searches for the first element of the wide string *s* that equals *c*. It considers the terminating null wide character as part of the wide string. If successful, the function returns the address of the matching element; otherwise, it returns a null pointer.

wcscmp

SYNTAX

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The function compares successive elements from two wide strings, *s1* and *s2*, until it finds elements that are not equal.

- If all elements are equal, the function returns zero.
- If the differing element from *s1* is greater than the element from *s2*, the function returns a positive number.
- Otherwise, the function returns a negative number.

wcscoll

SYNTAX

```
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The function compares two wide strings, `s1` and `s2`, using a comparison rule that depends on the current locale. If `s1` compares greater than `s2` by this rule, the function returns a positive number. If the two wide strings compare equal, it returns zero. Otherwise, it returns a negative number.

wcscpy

SYNTAX

```
wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The function copies the wide string `s2`, including its terminating null wide character, to successive elements of the array whose first element has the address `s1`. It returns `s1`.

wcscspn

SYNTAX

```
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The function searches for the first element `s1[i]` in the wide string `s1` that equals any one of the elements of the wide string `s2` and returns `i`. Each terminating null wide character is considered part of its wide string.

wcsftime

SYNTAX

```
size_t wcsftime(wchar_t *s, size_t maxsize, const wchar_t
     *format, const struct tm *timeptr);
```

The function generates formatted text, under the control of the format `format` and the values stored in the time structure `*timeptr`. It stores each generated wide character in successive locations of the array object of size *n* whose first element has the address `s`. The function then stores a null wide character in the next location of the array. It returns *x*, the number of wide characters generated, if $x < n$; otherwise, it returns zero, and the values stored in the array are indeterminate.

For each wide character other than `%` in the format, the function stores that wide character in the array object. Each occurrence of `%` followed by another character in the format is a conversion specifier. For each conversion specifier, the function stores a replacement wide character sequence. Conversion specifiers are the same as for the function `strftime`. The current locale category `LC_TIME` can affect these replacement character sequences.

wcslen

SYNTAX

```
size_t wcslen(const wchar_t *s);
```

DESCRIPTION

The function returns the number of wide characters in the wide string `s`, not including its terminating null wide character.

wcsncat

SYNTAX

```
wchar_t *wcsncat(wchar_t *s1, const wchar_t *s2, size_t
     n);
```

DESCRIPTION

The function copies the wide string *s2*, not including its terminating null wide character, to successive elements of the array that stores the wide string *s1*, beginning with the element that stores the terminating null wide character of *s1*. The function copies no more than *n* wide characters from *s2*. It then stores a null wide character, in the next element to be altered in *s1*, and returns *s1*.

wcsncmp

SYNTAX

```
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t
     n);
```

DESCRIPTION

The function compares successive elements from two wide strings, *s1* and *s2*, until it finds elements that are not equal or until it has compared the first *n* elements of the two wide strings.

- If all elements are equal, the function returns zero.
- If the differing element from *s1* is greater than the element from *s2*, the function returns a positive number.
- Otherwise, it returns a negative number.

wcsncpy

SYNTAX

```
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t
     n);
```

DESCRIPTION

The function copies the wide string `s2`, not including its terminating null wide character, to successive elements of the array whose first element has the address `s1`. It copies no more than *n* wide characters from `s2`. The function then stores zero or more null wide characters in the next elements to be altered in `s1` until it stores a total of *n* wide characters. It returns `s1`.

wcspbrk

SYNTAX

```
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The function searches for the first element `s1[i]` in the wide string `s1` that equals any one of the elements of the wide string `s2`. It considers each terminating null wide character as part of its wide string. If `s1[i]` is not the terminating null wide character, the function returns `&s1[i]`; otherwise, it returns a null pointer.

wcsrchr

SYNTAX

```
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

DESCRIPTION

The function searches for the last element of the wide string *s* that equals *c*. It considers the terminating null wide character as part of the wide string. If successful, the function returns the address of the matching element; otherwise, it returns a null pointer.

wcsrtombs

SYNTAX

```
size_t wcsrtombs(char *dst, const wchar_t **src, size_t
     len, mbstate_t *ps);
```

DESCRIPTION

The function converts the wide-character string beginning at `*src` to a sequence of multibyte characters as if by repeated calls of the form:

```
x = wcrtomb(dst ? dst : buf, *src, ps != 0 ? ps :
     &internal)
```

where *buf* is an array of type `char` and *internal* is an object of type `mbstate_t`, both internal to the `wcsrtombs` function. At program startup, *internal* is initialized to the initial conversion state. No other library function alters the value stored in *internal*.

If *dst* is not a null pointer, the `wcsrtombs` function stores at most *len* bytes by calls to `wcrtomb`. The function effectively increments *dst* by *x* and **src* by one after each call to `wcrtomb` that stores a complete converted multibyte character in the remaining space available. After a call that stores a complete null multibyte character at *dst* (including any shift sequence needed to restore the initial shift state), the function stores a null pointer at **src*.

If *dst* is a null pointer, *len* is effectively assigned a large value.

The function returns:

- `(size_t)-1` if a call to `wcrtomb` returns `(size_t)-1`, indicating that it has detected an invalid wide-character code
- the number of bytes successfully converted, not including the terminating null byte.

wcsspn

SYNTAX

```
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The function searches for the first element `s1[i]` in the wide string *s1* that equals none of the elements of the wide string *s2* and returns *i*. It considers the terminating null wide character as part of the wide string *s1* only.

wcsstr

SYNTAX

```
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The function searches for the first sequence of elements in the wide string *s1* that matches the sequence of elements in the wide string *s2*, not including its terminating null wide character. If successful, the function returns the address of the matching first element; otherwise, it returns a null pointer.

wcstod

SYNTAX

```
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

DESCRIPTION

The function converts the initial wide characters of the wide string *s* to an equivalent value *x* of type `double`. If *endptr* is not a null pointer, the function stores a pointer to the unconverted remainder of the wide string in **endptr*. The function then returns *x*.

The initial wide characters of the wide string s must match the same pattern as recognized by the function `strtod`, where each wide character *wc* is converted as if by calling `wctob(`*wc*`)`.

If the wide string *s* matches this pattern, its equivalent value is the value returned by `strtod` for the converted sequence. If the wide string *s* does not match a valid pattern, the value stored in **endptr* is *s*, and *x* is zero. If a range error occurs, `wcstod` behaves exactly as the functions declared in `math.h`.

wcstok

SYNTAX

```
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2, wchar_t
     **ptr);
```

DESCRIPTION

If `s1` is not a null pointer, the function begins a search of the wide string `s1`. Otherwise, it begins a search of the wide string whose address was last stored in `*ptr` on an earlier call to the function, as described below. The search proceeds as follows:

1 The function searches the wide string for begin, the address of the first element that equals none of the elements of the wide string `s2` (a set of token separators). It considers the terminating null character as part of the search wide string only.

2 If the search does not find an element, the function stores the address of the terminating null wide character in `*ptr` (so that a subsequent search beginning with that address will fail) and returns a null pointer. Otherwise, the function searches from begin for end, the address of the first element that equals any one of the elements of the wide string `s2`. It again considers the terminating null wide character as part of the search string only.

3 If the search does not find an element, the function stores the address of the terminating null wide character in `*ptr`. Otherwise, it stores a null wide character in the element whose address is end. Then it stores the address of the next element after end in `*ptr` (so that a subsequent search beginning with that address will continue with the remaining elements of the string) and returns begin.

wcstol

SYNTAX

```
long wcstol(const wchar_t *nptr, wchar_t **endptr, int
     base);
```

DESCRIPTION

The function converts the initial wide characters of the wide string *s* to an equivalent value *x* of type `long`. If *endptr* is not a null pointer, the function stores a pointer to the unconverted remainder of the wide string in **endptr*. The function then returns *x*.

The initial wide characters of the wide string *s* must match the same pattern as recognized by the function `strtol`, with the same base argument, where each wide character *wc* is converted as if by calling `wctob(`*wc*`)`.

If the wide string *s* matches this pattern, its equivalent value is the value returned by `strtol`, with the same base argument, for the converted sequence. If the wide string *s* does not match a valid pattern, the value stored in **endptr* is *s*, and *x* is zero. If the equivalent value is too large in magnitude to represent as type `long`, `wcstol` stores the value of `ERANGE` in `errno` and returns either `LONG_MAX` if *x* is positive or `LONG_MIN` if *x* is negative.

wcstoul

SYNTAX

```
unsigned long wcstoul(const wchar_t *nptr, wchar_t
     **endptr, int base);
```

DESCRIPTION

The function converts the initial wide characters of the wide string *s* to an equivalent value *x* of type `unsigned long`. If *endptr* is not a null pointer, it stores a pointer to the unconverted remainder of the wide string in **endptr*. The function then returns *x*.

`wcstoul` converts strings exactly as does `wcstol`, but checks only if the equivalent value is too large to represent as type `unsigned long`. In this case, `wcstoul` stores the value of `ERANGE` in `errno` and returns `ULONG_MAX`.

wcsxfrm

SYNTAX

`size_t wcsxfrm(wchar_t *s1, const wchar_t *s2, size_t n);`

DESCRIPTION

The function stores a wide string in the array whose first element has the address *`s1`*. It stores no more than *`n`* wide characters, including the terminating null wide character, and returns the number of wide characters needed to represent the entire wide string, not including the terminating null wide character. If the value returned is *`n`* or greater, the values stored in the array are indeterminate. (If *`n`* is zero, *`s1`* can be a null pointer.)

`wcsxfrm` generates the wide string it stores from the wide string *`s2`* by using a transformation rule that depends on the current locale. For example, if *`x`* is a transformation of *`s1`* and *`y`* is a transformation of *`s2`*, then `wcscmp(x, y)` returns the same value as `wcscoll(s1, s2)`.

wctob

SYNTAX

`int wctob(wint_t c);`

DESCRIPTION

The function determines whether *`c`* can be represented as a one-byte multibyte character *`x`*, beginning in the initial shift state. (It effectively calls `wcrtomb` to make the conversion.) If so, the function returns *`x`*. Otherwise, it returns `WEOF`.

wint_t

DEFINITION

`typedef i_type wint_t;`

DESCRIPTION

The type is the integer type `i_type` that can represent all values of type `wchar_t` as well as the value of the macro `WEOF`, and that does not change when promoted.

wmemchr

SYNTAX

```
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

DESCRIPTION

The function searches for the first element of an array beginning at the address *s* with size *n*, that equals *c*. If successful, it returns the address of the matching element; otherwise, it returns a null pointer.

wmemcmp

SYNTAX

```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t
     n);
```

DESCRIPTION

The function compares successive elements from two arrays beginning at the addresses *s1* and *s2* (both of size *n*), until it finds elements that are not equal:

- If all elements are equal, the function returns zero.
- If the differing element from *s1* is greater than the element from *s2*, the function returns a positive number.
- Otherwise, the function returns a negative number.

wmemcpy

SYNTAX

```
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t
     n);
```

DESCRIPTION

The function copies the array beginning at the address *s2* to the array beginning at the address *s1* (both of size *n*). It returns *s1*. The elements of the arrays can be accessed and stored in any order.

wmemmove

SYNTAX

```
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t
     n);
```

DESCRIPTION

The function copies the array beginning at `s2` to the array beginning at `s1` (both of size *n*). It returns `s1`. If the arrays overlap, the function accesses each of the element values from `s2` before it stores a new value in that element, so the copy is not corrupted.

wmemset

SYNTAX

```
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

DESCRIPTION

The function stores `c` in each of the elements of the array beginning at `s`, with size *n*. It returns `s`.

wprintf

SYNTAX

```
int wprintf(const wchar_t *format, ...);
```

DESCRIPTION

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stdout`. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

wscanf

SYNTAX

`int wscanf(const wchar_t *format, ...);`

DESCRIPTION

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned wide character from the stream `stdin`. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicators for the stream.

WIDE CHARACTER HANDLING – wctype.h

This chapter describes the standard header file `wctype.h`.

INTRODUCTION

Include the standard header `wctype.h` to declare several functions that are useful for classifying and mapping codes from the target wide-character set.

Every function that has a parameter of type `wint_t` can accept the value of the macro `WEOF` or any valid wide-character code (of type `wchar_t`). Thus, the argument can be the value returned by any of the functions: `btowc`, `fgetwc`, `fputwc`, `getwc`, `getwchar`, `putwc`, `putwchar`, `towctrans`, `towlower`, `towupper`, or `ungetwc`. You must not call these functions with other wide-character argument values.

The wide-character classification functions are strongly related to the (`byte`) character classification functions. Each function `is`*`XXX`* has a corresponding wide-character classification function `isw`*`XXX`*. Moreover, the wide-character classification functions are interrelated much the same way as their corresponding byte functions, with two added provisos:

- The function `iswprint`, unlike `isprint`, can return a non-zero value for additional space characters besides the wide-character equivalent of space (`L' '`). Any such additional characters return a non-zero value for `iswspace` and return zero for `iswgraph` or `iswpunct`.
- The characters in each wide-character class are a superset of the characters in the corresponding byte class. If the call `is`*`XXX`*`(`*`c`*`)` returns a non-zero value, then the corresponding call `isw`*`XXX`*`(btowc(`*`c`*`))` also returns a non-zero value.

An implementation can define additional characters that return non-zero for some of these functions. Any character set can contain additional characters that return non-zero for:

- `iswpunct` (provided the characters cause `iswalnum` to return zero)
- `iswcntrl` (provided the characters cause `iswprint` to return zero)

Moreover, a locale other than the "C" locale can define additional characters for:

- `iswalpha`, `iswupper`, and `iswlower` (provided the characters cause `iswcntrl`, `iswdigit`, `iswpunct`, and `iswspace` to return zero)
- `iswspace` (provided the characters cause `iswpunct` to return zero)

Notice that the last rule differs slightly from the corresponding rule for the function `isspace`, as indicated above. Notice also that an implementation can define a locale other than the "C" locale in which a character can cause `iswalpha` (and hence `iswalnum`) to return non-zero, yet still cause `iswupper` and `iswlower` to return zero.

SUMMARY

The `wctype.h` header file contains the following functions and macro definitions:

```
#define WEOF wint_t_constant_expression
int iswalnum(wint_t c);
int iswalpha(wint_t c);
int iswcntrl(wint_t c);
int iswctype(wint_t c, wctype_t category);
int iswdigit(wint_t c);
int iswgraph(wint_t c);
int iswlower(wint_t c);
int iswprint(wint_t c);
int iswpunct(wint_t c);
int iswspace(wint_t c);
int iswupper(wint_t c);
int iswxdigit(wint_t c);
wint_t towctrans(wint_t c, wctrans_t category);
wint_t towlower(wint_t c);
wint_t towupper(wint_t c);
wctrans_t wctrans(const char *property);
typedef s_type wctrans_t;
wctype_t wctype(const char *property);
typedef s_type wctype_t;
typedef i_type wint_t;
```

In the following sections each function and macro definition is described.

WEOF

DEFINITION

```
#define WEOF wint_t_constant_expression
```

DESCRIPTION

The macro yields the return value, of type `wint_t`, used to signal the end of a wide stream or to report an error condition.

iswalnum

SYNTAX

```
int iswalnum(wint_t c);
```

DESCRIPTION

The function returns non-zero if *c* is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
```

or any other locale-specific alphabetic character.

iswalpha

SYNTAX

```
int iswalpha(wint_t c);
```

DESCRIPTION

The function returns non-zero if *c* is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

or any other locale-specific alphabetic character.

iswcntrl

SYNTAX

`int iswcntrl(wint_t c);`

DESCRIPTION

The function returns non-zero if *`c`* is any of:

`BEL BS CR FF HT NL VT`

or any other implementation-defined control character.

iswctype

SYNTAX

`int iswctype(wint_t c, wctype_t category);`

DESCRIPTION

The function returns non-zero if *`c`* is any character in the category *`category`*. The value of *`category`* must have been returned by an earlier successful call to `wctype`.

iswdigit

SYNTAX

`int iswdigit(wint_t c);`

DESCRIPTION

The function returns non-zero if *`c`* is any of:

`0 1 2 3 4 5 6 7 8 9`

iswgraph

SYNTAX

`int iswgraph(wint_t c);`

DESCRIPTION

The function returns non-zero if *`c`* is any character for which either `iswalnum` or `iswpunct` returns non-zero.

iswlower

SYNTAX

`int iswlower(wint_t c);`

DESCRIPTION

The function returns non-zero if *c* is any of:

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

or any other locale-specific lowercase character.

iswprint

SYNTAX

`int iswprint(wint_t c);`

DESCRIPTION

The function returns non-zero if *c* is space, a character for which `iswgraph` returns non-zero, or an implementation-defined subset of the characters for which `iswspace` returns non-zero.

iswpunct

SYNTAX

`int iswpunct(wint_t c);`

DESCRIPTION

The function returns non-zero if *c* is any of:

```
! " # % & ' ( ) ; <
= > ? [ \ ] * + , -
. / : ^ _ { | } ~
```

or any other implementation-defined punctuation character.

iswspace

SYNTAX

`int iswspace(wint_t c);`

DESCRIPTION

The function returns non-zero if *c* is any of:

`CR FF HT NL VT space`

or any other locale-specific space character.

iswupper

SYNTAX

`int iswupper(wint_t c);`

DESCRIPTION

The function returns non-zero if *c* is any of:

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

or any other locale-specific uppercase character.

iswxdigit

SYNTAX

`int iswxdigit(wint_t c);`

DESCRIPTION

The function returns non-zero if *c* is any of

`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

towctrans

SYNTAX

`wint_t towctrans(wint_t c, wctrans_t category);`

DESCRIPTION

The function returns the transformation of the character *c*, using the transform in the category *category*. The value of *category* must have been returned by an earlier successful call to `wctrans`.

towlower

SYNTAX

`wint_t towlower(wint_t c);`

DESCRIPTION

The function returns the corresponding lowercase letter if one exists and if `iswupper(c)`; otherwise, it returns *c*.

towupper

SYNTAX

`wint_t towupper(wint_t c);`

DESCRIPTION

The function returns the corresponding uppercase letter if one exists and if `iswlower(c)`; otherwise, it returns *c*.

wctrans

SYNTAX

`wctrans_t wctrans(const char *property);`

DESCRIPTION

The function determines a mapping from one set of wide-character codes to another. If the `LC_CTYPE` category of the current locale does not define a mapping whose name matches the property string *property*, the function returns zero. Otherwise, it returns a non-zero value suitable for use as the second argument to a subsequent call to `towctrans`.

The following pairs of calls have the same behavior in all locales (but an implementation can define additional mappings even in the "C" locale):

`towlower(c)` same as `towctrans(c, wctrans("tolower"))`

`towupper(c)` same as `towctrans(c, wctrans("toupper"))`

wctrans_t

DEFINITION

`typedef s_type wctrans_t;`

DESCRIPTION

The type is the scalar type `s-type` that can represent locale-specific character mappings, as specified by the return value of `wctrans`.

wctype

SYNTAX

```
wctype_t wctype(const char *property);
wctrans_t wctrans(const char *property);
```

DESCRIPTION

The function determines a classification rule for wide-character codes. If the `LC_CTYPE` category of the current locale does not define a classification rule whose name matches the property string *`property`*, the function returns zero. Otherwise, it returns a non-zero value suitable for use as the second argument to a subsequent call to `towctrans`.

The following pairs of calls have the same behavior in all locales (but an implementation can define additional classification rules even in the "C" locale):

`iswalnum(c)` same as `iswctype(c, wctype("alnum"))`

`iswalpha(c)` same as `iswctype(c, wctype("alpha"))`

`iswcntrl(c)` same as `iswctype(c, wctype("cntrl"))`

`iswdigit(c)` same as `iswctype(c, wctype("digit"))`

`iswgraph(c)` same as `iswctype(c, wctype("graph"))`

`iswlower(c)` same as `iswctype(c, wctype("lower"))`

`iswprint(c)` same as `iswctype(c, wctype("print"))`

`iswpunct(c)` same as `iswctype(c, wctype("punct"))`

`iswspace(c)` same as `iswctype(c, wctype("space"))`

`iswupper(c)` same as `iswctype(c, wctype("upper"))`

`iswxdigit(c)` same as `iswctype(c, wctype("xdigit"))`

wctype_t

DEFINITION

```
typedef s_type wctype_t;
```

DESCRIPTION

The type is the scalar type `s-type` that can represent locale-specific character classifications, as specified by the return value of `wctype`.

wint_t

DEFINITION

```
typedef i_type wint_t;
```

DESCRIPTION

The type is the integer type `i_type` that can represent all values of type `wchar_t` as well as the value of the macro `WEOF`, and that does not change when promoted.