The following is WG14's contribution to WG20 for an internationalization API. These are excerpts from the ISO C Standard (ISO/IEC 9899:1990).

7.1.1 Definitions of terms

The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences. It is represented in the text and examples by a period, but may be changed by the **`setlocale`** function.

A *wide character* is a code value (a binary encoded integer) of an object of type **`wchar_t`** that corresponds to a member of the extended character set.[1]

A *null wide character* is a wide character with code value zero.

A *wide string* is a contiguous sequence of wide characters terminated by and including the first null wide character. A *pointer to a wide string* is a pointer to its initial (lowest addressed) wide character. The *length of a wide string* is the number of wide characters preceding the null wide character and the *value of a wide string* is the sequence of code values of the contained wide characters, in order.

A *shift sequence* is a contiguous sequence of bytes within a multibyte string that (potentially) causes a change in shift state. (See subclause 5.2.1.2 of ISO/IEC 9899:1990.) A shift sequence shall not have a corresponding wide character; it is instead taken to be an adjunct to an adjacent multibyte character.

7.1.4 Errors `<errno.h>`

The list of macros defined in the standard header file **`<errno.h>`** (see 7.1.4 of ISO/IEC 9899:1990) should be adjusted to include the macro, **`EILSEQ`**.

7.1.6 Common definitions `<stddef.h>`

```
wchar_t
```

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

7.4 Localization `<locale.h>`

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

The type is

```
struct lconv
```

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges is explained in 7.5.2.1. In the **"C"** locale, the members shall have the values specified in the comments.

```
char *decimal_point;       /* "." */
char *thousands_sep;       /* "" */
char *grouping;            /* "" */
char *int_curr_symbol;     /* "" */
char *currency_symbol;     /* "" */
char *mon_decimal_point;   /* "" */
char *mon_thousands_sep;   /* "" */
char *mon_grouping;        /* "" */
char *positive_sign;       /* "" */
char *negative_sign;       /* "" */
char int_frac_digits;      /* CHAR_MAX */
char frac_digits;          /* CHAR_MAX */
char p_cs_precedes;        /* CHAR_MAX */
char p_sep_by_space;       /* CHAR_MAX */
char n_cs_precedes;        /* CHAR_MAX */
char n_sep_by_space;       /* CHAR_MAX */
char p_sign_posn;          /* CHAR_MAX */
char n_sign_posn;          /* CHAR_MAX */
```

The macros defined are **NULL** (described in 7.1.6); and

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

which expand to integral constant expressions with distinct values, suitable for use as the first argument to the **setlocale** function. Additional macro definitions, beginning with the characters **LC_** and an uppercase letter,[2] may also be specified by the implementation.

7.4.1 Locale control

7.4.1.1 The `setlocale` function

Synopsis

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Description

The **setlocale** function selects the appropriate portion of the program's locale as specified by the **category** and **locale** arguments. The **setlocale** function may be used to change or query the program's entire current locale or portions thereof. The value **LC_ALL** for **category** names the program's entire locale; the other values for **category** name only a portion of the program's locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** functions. **LC_CTYPE** affects the behavior of the character handling functions[3] and the multibyte functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** function. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the **localeconv** function. **LC_TIME** affects the behavior of the **strftime** function.

A value of **"C"** for **locale** specifies the minimal environment for C translation; a value of **""** for **locale** specifies the locale-specific native environment. Other implementation-defined strings may be passed as the second argument to **setlocale**.

At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

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

Returns

If a pointer to a string is given for **locale** and the selection can be honored, the **setlocale** function returns a pointer to the string associated with the specified **category** for the new locale. If the selection cannot be honored, the **setlocale** function returns a null pointer and the program's locale is not changed.

---

2. See ‘‘future library directions’’ (7.14.3).
3. The only functions in 7.3 whose behavior is not affected by the current locale are **isdigit** and **isxdigit**.

A null pointer for **locale** causes the **setlocale** function to return a pointer to the string associated with the **category** for the program's current locale; the program's locale is not changed.[4]

The pointer to string returned by the **setlocale** function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **setlocale** function.

Forward references: formatted input/output functions (7.10.6), the multibyte character functions (7.11.7), the multibyte string functions (7.11.8), string conversion functions (7.11.1), the **strcoll** function (7.12.4.3), the **strftime** function (7.13.3.5), the **strxfrm** function (7.12.4.5).

7.4.2 Numeric formatting convention inquiry

7.4.2.1 The **localeconv** function

Synopsis

```
#include <locale.h>
struct lconv *localeconv(void);
```

Description

The **localeconv** function sets the components of an object with type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type **char *** are pointers to strings, any of which (except **decimal_point**) can point to **""**, to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are nonnegative numbers, any of which can be **CHAR_MAX** to indicate that the value is not available in the current locale. The members include the following:

char *decimal_point
The decimal-point character used to format nonmonetary quantities.

char *thousands_sep
The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

char *grouping
A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

4. The implementation must arrange to encode in a string the various categories due to a heterogeneous locale when **category** has the value **LC_ALL**.

`char *int_curr_symbol`
The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217:1987. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

`char *currency_symbol`
The local currency symbol applicable to the current locale.

`char *mon_decimal_point`
The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`
The separator for groups of digits before the decimal-point in formatted monetary quantities.

`char *mon_grouping`
A string whose elements indicate the size of each group of digits in formatted monetary quantities.

`char *positive_sign`
The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`
The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`
The number of fractional digits (those after the decimal-point) to be displayed in a internationally formatted monetary quantity.

`char frac_digits`
The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

`char p_cs_precedes`
Set to 1 or 0 if the **`currency_symbol`** respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

`char p_sep_by_space`
Set to 1 or 0 if the **`currency_symbol`** respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

`char n_cs_precedes`
Set to 1 or 0 if the **`currency_symbol`** respectively precedes or succeeds the value for a negative formatted monetary quantity.

`char n_sep_by_space`
Set to 1 or 0 if the **`currency_symbol`** respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

`char p_sign_posn`
Set to a value indicating the positioning of the **`positive_sign`** for a nonnegative formatted monetary quantity.

`char n_sign_posn`
Set to a value indicating the positioning of the **`negative_sign`** for a negative formatted monetary quantity.

The elements of **`grouping`** and **`mon_grouping`** are interpreted according to the following:

`CHAR_MAX` No further grouping is to be performed.

`0` The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The value of **`p_sign_posn`** and **`n_sign_posn`** is interpreted according to the following:

`0` Parentheses surround the quantity and **`currency_symbol`**.

`1` The sign string precedes the quantity and **`currency_symbol`**.

`2` The sign string succeeds the quantity and **`currency_symbol`**.

`3` The sign string immediately precedes the **`currency_symbol`**.

`4` The sign string immediately succeeds the **`currency_symbol`**.

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

Returns

The **`localeconv`** function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the **`localeconv`** function. In addition, calls to the **`setlocale`** function with categories **`LC_ALL`**, **`LC_MONETARY`**, or **`LC_NUMERIC`** may overwrite the contents of the structure.

Example

The following table illustrates the rules which may well be used by four countries to format monetary quantities.

Country	Positive format	Negative format	International format
Italy	**L.1.234**	**-L.1.234**	**ITL.1.234**
Netherlands	**F 1.234,56**	**F -1.234,56**	**NLG 1.234,56**
Norway	**kr1.234,56**	**kr1.234,56-**	**NOK 1.234,56**
Switzerland	**SFrs.1,234.56**		**SFrs.1,234.56CCHF 1,234.56**

For these four countries, the respective values for the monetary members of the structure returned by **localeconv** are:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	**"ITL."**	**"NLG "**	**"NOK "**	**"CHF "**
currency_symbol	**"L."**	**"F"**	**"kr"**	**"SFrs."**
mon_decimal_point	**""**	**","**	**","**	**"."**
mon_thousands_sep	**"."**	**"."**	**"."**	**","**
mon_grouping	**"\3"**	**"\3"**	**"\3"**	**"\3"**
positive_sign	**""**	**""**	**""**	**""**
negative_sign	**"-"**	**"-"**	**"-"**	**"C"**
int_frac_digits	**0**	**2**	**2**	**2**
frac_digits	**0**	**2**	**2**	**2**
p_cs_precedes	**1**	**1**	**1**	**1**
p_sep_by_space	**0**	**1**	**0**	**0**
n_cs_precedes	**1**	**1**	**1**	**1**
n_sep_by_space	**0**	**1**	**0**	**0**
p_sign_posn	**1**	**1**	**1**	**1**
n_sign_posn	**1**	**4**	**2**	**2**

7.10 General utilities <stdlib.h>

MB_CUR_MAX

which expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category **LC_CTYPE**), and whose value is never greater than **MB_LEN_MAX**.

7.10.7 Multibyte character functions

The behavior of the multibyte character functions is affected by the **LC_CTYPE** category of the current locale. For a state-dependent encoding, each function is placed into its initial state by a call for which its character pointer argument, **s**, is a null pointer. Subsequent calls with **s** as other than a null pointer cause the internal state of the function to be altered as necessary. A call with **s** as a null pointer causes these functions to return a nonzero value if encodings have state dependency, and zero otherwise.[5] Changing the **LC_CTYPE** category causes the shift state of these functions to be indeterminate.

7.10.7.1 The mblen function

Synopsis

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

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

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

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

Returns

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

Forward references: the **mbtowc** function (7.11.7.2).

7.10.7.2 The mbtowc function

Synopsis

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

If **s** is not a null pointer, the **mbtowc** function determines the number of bytes that are contained in the multibyte character pointed to by **s**. It then determines the code for the value of type **wchar_t** that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and **pwc** is not a null pointer, the **mbtowc** function stores the code in the object pointed to by **pwc**. At most **n** bytes of the array pointed to by **s** will be examined.

5. If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

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

Returns

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

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

7.10.7.3 The wctomb function

Synopsis

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Description

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

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

Returns

If **s** is a null pointer, the **wctomb** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If **s** is not a null pointer, the **wctomb** function returns −1 if the value of **wchar** does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of **wchar**.

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

7.10.8 Multibyte string functions

The behavior of the multibyte string functions is affected by the **LC_CTYPE** category of the current locale.

7.10.8.1 The mbstowcs function

Synopsis

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Description

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

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

Returns

If an invalid multibyte character is encountered, the **mbstowcs** function returns **(size_t)-1**. Otherwise, the **mbstowcs** function returns the number of array elements modified, not including a terminating zero code, if any.[6]

7.10.8.2 The wcstombs function

Synopsis

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Description

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

6. The array will not be null- or zero-terminated if the value returned is **n**.

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

Returns

If a code is encountered that does not correspond to a valid multibyte character, the **wcstombs** function returns **(size_t)-1**. Otherwise, the **wcstombs** function returns the number of bytes modified, not including a terminating null character, if any.[6]

7.13.9 Wide-character classification and mapping utilities <wctype.h>

Function names that begin with **is** or **to** and a lowercase letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<wctype.h>** header.

7.13.10 Extended multibyte and wide-character utilities <wchar.h>

Function names that begin with **wcs** and a lowercase letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<wchar.h>** header.

Lowercase letters may be added to the conversion specifiers in **fwprintf** and **fwscanf**.

7.15 Wide-character classification and mapping utilities

<wctype.h>

7.15.1 Introduction

The header **<wctype.h>** declares three data types, one macro, and many functions.[7]

The types declared are

wint_t

which is an integral type unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (See **WEOF** below).[8]

wctrans_t

which is a scalar type that can hold values which represent locale-specific character mappings, and

7. See ‘‘future library directions’’ (7.14).

8. **wchar_t** and **wint_t** can be the same integral type.

wctype_t

which is a scalar type that can hold values which represent locale-specific character classifications.

The macro defined is

WEOF

which expands to a constant expression of type **wint_t** whose value does not correspond to any member of the extended character set.[9] It is accepted (and returned) by several functions in this subclause to indicate *end-of-file*, that is, no more input from a stream. It is also used as a wide-character value that does not correspond to any member of the extended character set.

The functions declared are grouped as follows:

- Functions that provide wide-character classification;
- Extensible functions that provide wide-character classification;
- Functions that provide wide-character case mapping;
- Extensible functions that provide wide-character mapping.

For all functions described in this subclause that accept an argument of type **wint_t**, the value shall be representable as a **wchar_t** or shall equal the value of the macro **WEOF**. If this argument has any other value, the behavior is undefined.

The behavior of these functions is affected by the **LC_CTYPE** category of the current locale.

7.15.2 Wide-character classification utilities

The header **<wctype.h>** declares several functions useful for classifying wide characters.

The term *printing wide character* refers to a member of a locale-specific set of wide characters, each of which occupies at least one printing position on a display device. The term *control wide character* refers to a member of a locale-specific set of wide characters that are not printing wide characters.

9. The value of the macro **WEOF** may differ from that of **EOF** and need not be negative.

7.15.2.1 Wide-character classification functions

The functions in this subclause return nonzero (true) if and only if the value of the argument **`wc`** conforms to that in the description of the function.

Except for the **`iswgraph`** and **`iswpunct`** functions with respect to printing white-space wide characters other than **`L' '`**, each of the following eleven functions returns true for each wide character that corresponds (as if by a call to the **`wctob`** function) to a character (byte) for which the respectively matching character testing function from subclause 7.3.1 returns true.[10]

Forward References: the **`wctob`** function (7.17.6.1.2).

7.15.2.1.1 The `iswalnum` function

Synopsis

```
#include <wctype.h>
int iswalnum(wint_t wc);
```

Description

The **`iswalnum`** function tests for any wide character for which **`iswalpha`** or **`iswdigit`** is true.

7.15.2.1.2 The `iswalpha` function

Synopsis

```
#include <wctype.h>
int iswalpha(wint_t wc);
```

Description

The **`iswalpha`** function tests for any wide character for which **`iswupper`** or **`iswlower`** is true, or any wide character that is one of a locale-specific set of wide characters for which none of **`iswcntrl`**, **`iswdigit`**, **`iswpunct`**, or **`iswspace`** is true.

7.15.2.1.3 The `iswcntrl` function

Synopsis

```
#include <wctype.h>
int iswcntrl(wint_t wc);
```

---

10. For example, if the expression **`isalpha(wctob(wc))`** evaluates to true, then the call **`iswalpha(wc)`** must also return true. But, if the expression **`isgraph(wctob(wc))`** evaluates to true (which cannot occur for **`wc == L' '`** of course), then either **`iswgraph(wc)`** or **`iswprint(wc) && iswspace(wc)`** must be true, but not both.

Description

The **iswcntrl** function tests for any control wide character.

7.15.2.1.4 The iswdigit function

Synopsis

```
#include <wctype.h>
int iswdigit(wint_t wc);
```

Description

The **iswdigit** function tests for any wide character that corresponds to a decimal-digit character (as defined in subclause 5.2.1).

7.15.2.1.5 The iswgraph function

Synopsis

```
#include <wctype.h>
int iswgraph(wint_t wc);
```

Description

The **iswgraph** function tests for any wide character for which **iswprint** is true and **iswspace** is false.[11]

7.15.2.1.6 The iswlower function

Synopsis

```
#include <wctype.h>
int iswlower(wint_t wc);
```

Description

The **iswlower** function tests for any wide character that corresponds to a lowercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

7.15.2.1.7 The iswprint function

Synopsis

```
#include <wctype.h>
int iswprint(wint_t wc);
```

Description

---

11. Note that the behavior of the **iswgraph** and **iswpunct** functions may differ from their matching functions in subclause 7.3.1 with respect to printing white space basic execution characters other than **' '**.

The **iswprint** function tests for any printing wide character.

7.15.2.1.8 The iswpunct function

Synopsis

```
#include <wctype.h>
int iswpunct(wint_t wc);
```

Description

The **iswpunct** function tests for any printing wide character that is one of a locale-specific set of wide characters for which neither **iswspace** nor **iswalnum** is true.

7.15.2.1.9 The iswspace function

Synopsis

```
#include <wctype.h>
int iswspace(wint_t wc);
```

Description

The **iswspace** function tests for any wide character that corresponds to a locale-specific set of wide characters for which none of **iswalnum**, **iswgraph**, or **iswpunct** is true.

7.15.2.1.10 The iswupper function

Synopsis

```
#include <wctype.h>
int iswupper(wint_t wc);
```

Description

The **iswupper** function tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

7.15.2.1.11 The iswxdigit function

Synopsis

```
#include <wctype.h>
int iswxdigit(wint_t wc);
```

Description

The **iswxdigit** function tests for any wide character that corresponds to a hexadecimal-digit character (as defined in subclause 6.1.3.2).

7.15.2.2 Extensible wide-character classification functions

The functions **wctype** and **iswctype** provide extensible wide-character classification as well as testing equivalent to that performed by the functions described in the previous subclause (4.5.2.1).

7.15.2.2.1 The wctype function

Synopsis

```
#include <wctype.h>
wctype_t wctype(const char *property);
```

Description

The **wctype** function constructs a value with type **wctype_t** that describes a class of wide characters identified by the string argument **property**.

The eleven strings listed in the description of the **iswctype** function shall be valid in all locales as **property** arguments to the **wctype** function.

Returns

If **property** identifies a valid class of wide characters according to the **LC_CTYPE** category of the current locale, the **wctype** function returns a nonzero value that is valid as the second argument to the **iswctype** function; otherwise, it returns zero.

7.15.2.2.2 The iswctype function

Synopsis

```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t desc);
```

Description

The **iswctype** function determines whether the wide character **wc** has the property described by **desc**. The current setting of the **LC_CTYPE** category shall be the same as during the call to **wctype** that returned the value **desc**.

Each of the following eleven expressions has a truth-value equivalent to the call to the wide-character testing function (4.5.2.1) in the comment that follows the expression:

```
iswctype(wc, wctype("alnum"))    /* iswalnum(wc) */
iswctype(wc, wctype("alpha"))    /* iswalpha(wc) */
iswctype(wc, wctype("cntrl"))    /* iswcntrl(wc) */
iswctype(wc, wctype("digit"))    /* iswdigit(wc) */
iswctype(wc, wctype("graph"))    /* iswgraph(wc) */
iswctype(wc, wctype("lower"))    /* iswlower(wc) */
iswctype(wc, wctype("print"))    /* iswprint(wc) */
iswctype(wc, wctype("punct"))    /* iswpunct(wc) */
iswctype(wc, wctype("space"))    /* iswspace(wc) */
iswctype(wc, wctype("upper"))    /* iswupper(wc) */
iswctype(wc, wctype("xdigit"))   /* iswxdigit(wc) */
```

Returns

The **iswctype** function returns nonzero (true) if and only if the value of the wide character **wc** has the property described by **desc**.

7.15.3 Wide-character mapping utilities

The header **<wctype.h>** declares several functions useful for mapping wide characters.

7.15.3.1 Wide-character case-mapping functions

7.15.3.1.1 The towlower function

Synopsis

```
#include <wctype.h>
wint_t towlower(wint_t wc);
```

Description

The **towlower** function converts an uppercase letter to the corresponding lowercase letter.

Returns

If the argument is a wide character for which **iswupper** is true and there is a corresponding wide character for which **iswlower** is true, the **towlower** function returns the corresponding wide character; otherwise, the argument is returned unchanged.

7.15.3.1.2 The towupper function

Synopsis

```
#include <wctype.h>
wint_t towupper(wint_t wc);
```

Description

The **towupper** function converts a lowercase letter to the corresponding uppercase letter.

Returns

If the argument is a wide character for which **`iswlower`** is true and there is a corresponding wide character for which **`iswupper`** is true, the **`towupper`** function returns the corresponding wide character; otherwise, the argument is returned unchanged.

7.15.3.2 Extensible wide-character mapping functions

The functions **`wctrans`** and **`towctrans`** provide extensible wide-character mapping as well as case mapping equivalent to that performed by the functions described in the previous subclause (4.5.3.1).

7.15.3.2.1 The `wctrans` function

Synopsis

```
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

Description

The **`wctrans`** function constructs a value with type **`wctrans_t`** that describes a mapping between wide characters identified by the string argument **`property`**.

The two strings listed in the description of the **`towctrans`** function shall be valid in all locales as **`property`** arguments to the **`wctrans`** function.

Returns

If **`property`** identifies a valid mapping of wide characters according to the **`LC_CTYPE`** category of the current locale, the **`wctrans`** function returns a nonzero value that is valid as the second argument to the **`towctrans`** function; otherwise, it returns zero.

7.15.3.2.2 The `towctrans` function

Synopsis

```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description

The **`towctrans`** function maps the wide character **`wc`** using the mapping described by **`desc`**. The current setting of the **`LC_CTYPE`** category shall be the same as during the call to **`towctrans`** that returned the value **`desc`**.

Each of the following two expressions behaves the same as the call to the wide-character case-mapping function (4.5.3.1) in the comment that follows the expression:

```
towctrans(wc, wctrans("tolower"))    /* towlower(wc) */
towctrans(wc, wctrans("toupper"))    /* towupper(wc) */
```

Returns

The **towctrans** function returns the mapped value of **wc** using the mapping described by **desc**.

7.16 Extended multibyte and wide-character utilities <wchar.h>

7.16.1 Introduction

The header **<wchar.h>** declares four data types, one tag, four macros, and many functions.[12]

The types declared are **wchar_t** and **size_t** (both described in subclause 7.1.6),

mbstate_t

which is an object type other than an array type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters, and

wint_t

described in subclause 4.5.1.

The tag **tm** is declared as naming an incomplete structure type, the contents of which are described in subclause 7.13.1.

The macros defined are **NULL** (described in subclause 7.1.6),

WCHAR_MAX

which is the maximum value representable by an object of type **wchar_t**,[13]

WCHAR_MIN

which is the minimum value representable by an object of type **wchar_t**, and

WEOF

described in subclause 4.5.2.

The functions declared are grouped as follows:

- Functions that perform input and output of wide characters, or multibyte characters, or both;
- Functions that provide wide-string numeric conversion;
- Functions that perform general wide-string manipulation;

12. See ‘‘future library directions’’ (7.14).
13. The values **WCHAR_MAX** and **WCHAR_MIN** do not necessarily correspond to members of the extended character set.

- A function for wide-string date and time conversion; and
- Functions that provide extended capabilities for conversion between multibyte and wide-character sequences.

Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the behavior is undefined.

7.16.2 Formatted wide-character input/output functions

7.16.2.1 The `fwprintf` function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE *stream, const wchar_t *format, ...);
```

Description

The **`fwprintf`** function writes output to the stream pointed to by **`stream`**, under control of the wide string pointed to by **`format`** that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **`fwprintf`** function returns when the end of the format string is encountered.

The format is composed of zero or more directives: ordinary wide characters (not **`%`**), and conversion specifications. The processing of conversion specifications is as if they were replaced in the format string by wide-character strings that are each the result of fetching zero or more subsequent arguments and converting them, if applicable, according to the corresponding conversion specifier. The expanded wide-character format string is then written to the output stream.

Each conversion specification is introduced by the wide character **`%`**. After the **`%`**, the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer wide characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk **`*`** (described later) or a decimal integer.[14]

---

14. Note that **`0`** is taken as a flag, not as the beginning of a field width.

- An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of wide characters to be written from a string in **s** conversion. The precision takes the form of a period (**.**) followed either by an asterisk ***** (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional **l** (ell) specifying that a following **c** conversion specifier applies to a **wint_t** argument; an optional **l** specifying that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; an optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument is promoted according to the integral promotions, and its value is converted to **short int** or **unsigned short int** before printing); an optional **h** specifying that a following **n** conversion specifier applies to a pointer to a **short int** argument; an optional **l** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; an optional **l** specifying that a following **n** conversion specifier applies to a pointer to a **long int** argument; or an optional **L** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion specifier applies to a **long double** argument. If an **h**, **l**, or **L** appears with any other conversion specifier, the behavior is undefined.
- A wide character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a **-** flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag wide characters and their meanings are

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)

space If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide characters, a space is prefixed to the result. If the *space* and **+** flags both appear, the *space* flag is ignored.

The result is to be converted to an ‘‘alternate form.’’ For **o** conversion, it increases the precision to force the first digit of the result to be a zero, if necessary. For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result always contains a decimal-point wide character, even if no digits follow it. (Normally, a decimal-point wide character appears in the result of these conversions only if a digit follows it.) For **g** and **G** conversions, trailing zeros are *not* be removed from the result. For other conversions, the behavior is undefined.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **-** flags both appear, the **0** flag is ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are

d,i The **int** argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

o,u,x,X The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

f The **double** argument is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point wide character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point wide character appears. If a decimal-point wide character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

e,E The **double** argument is converted in the style [-]d.ddde±dd, where there is one digit before the decimal-point wide character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point wide character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

g,G The **double** argument is converted in style **f** or **e** (or in style **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point wide character appears only if it is followed by a digit.

c If no **l** qualifier is present, the **int** argument is converted to a wide character as if by calling **btowc** and the resulting wide character is written. Otherwise, the **wint_t** argument is converted to **wchar_t** and written.

s If no **l** qualifier is present, the argument shall be a pointer to a character array containing a multibyte sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the converted array, the converted array shall contain a null wide character.

If an **l** qualifier is present, the argument shall be a pointer to an array of **wchar_t** type. Wide characters from the array are written up to (but not including) a terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null wide character.

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable wide characters, in an implementation-defined manner.

n The argument shall be a pointer to an integer into which is *written* the number of wide characters written to the output stream so far by this call to **fwprintf**. No argument is converted.

% A **%** wide character is written. No argument is converted. The complete conversion specification shall be **%%**.

If a conversion specification is invalid, the behavior is undefined.[15]

If any argument is, or points to, a union or an aggregate (except for an array of **char** type using **%s** conversion, an array of **wchar_t** type using **%ls** conversion, or a pointer using **%p** conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

---

15. See ''future library directions'' (7.14).

Returns

The **fwprintf** function returns the number of wide characters transmitted, or a negative value if an output error occurred.

Environmental limit

The minimum value for the maximum number of wide characters produced by any single conversion shall be 509.

Example

To print a date and time in the form ‘‘Sunday, July 3, 10:02’’ followed by **pi** to five decimal places:

```
#include <math.h>
#include <stdio.h>
#include <wchar.h>
/*...*/
wchar_t *weekday, *month;  /* pointers to wide strings */
int day, hour, min;
fwprintf(stdout, L"%ls, %ls %d, %.2d:%.2d\n",
     weekday, month, day, hour, min);
fwprintf(stdout, L"pi = %.5f\n", 4 * atan(1.0));
```

Forward References: the **btowc** function (7.17.6.1.1), the **mbrtowc** function (7.17.6.3.2).

7.16.2.2 The fwscanf function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE *stream, const wchar_t *format, ...);
```

Description

The **fwscanf** function reads input from the stream pointed to by **stream**, under control of the wide string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format is composed of zero or more directives: one or more white-space wide characters; an ordinary wide character (neither **%** nor a white-space wide character); or a conversion specification. Each conversion specification is introduced by a **%**. After the **%**, the following appear in sequence:

- An optional assignment-suppressing wide character **`*`**.
- An optional nonzero decimal integer that specifies the maximum field width (in wide characters).
- An optional **`h`**, **`l`** (ell), or **`L`** indicating the size of the receiving object. The conversion specifiers **`c`**, **`s`**, and **`[`** shall be preceded by **`l`** if the corresponding argument is a pointer to **`wchar_t`** rather than a pointer to a character type. The conversion specifiers **`d`**, **`i`**, and **`n`** shall be preceded by **`h`** if the corresponding argument is a pointer to **`short int`** rather than a pointer to **`int`**, or by **`l`** if it is a pointer to **`long int`**. Similarly, the conversion specifiers **`o`**, **`u`**, and **`x`** shall be preceded by **`h`** if the corresponding argument is a pointer to **`unsigned short int`** rather than a pointer to **`unsigned int`**, or by **`l`** if it is a pointer to **`unsigned long int`**. Finally, the conversion specifiers **`e`**, **`f`**, and **`g`** shall be preceded by **`l`** if the corresponding argument is a pointer to **`double`** rather than a pointer to **`float`**, or by **`L`** if it is a pointer to **`long double`**. If an **`h`**, **`l`**, or **`L`** appears with any other conversion specifier, the behavior is undefined.
- A wide character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The **`fwscanf`** function executes each directive of the format in turn. If a directive fails, as detailed below, the **`fwscanf`** function returns. Failures are described as input failures (if an encoding error occurs or due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space wide character(s) is executed by reading input up to the first non-white-space wide character (which remains unread), or until no more wide characters can be read.

A directive that is an ordinary wide character is executed by reading the next wide character of the stream. If the wide character differs from the directive, the directive fails, and the differing and subsequent wide characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space wide characters (as specified by the **`iswspace`** function) are skipped, unless the specification includes a **`c`** or **`n`** specifier.[16]

An input item is read from the stream, unless the specification includes an **`n`** specifier. An input item is defined as the longest sequence of input wide characters, not exceeding any specified field width, which is, or is a prefix of, a matching sequence. The first wide character, if any, after the input item remains unread. If the length of

16. These white-space wide characters are not counted against a specified field width.

the input item is zero, the execution of the directive fails: this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a **%** specifier, the input item (or, in the case of a **%n** directive, the count of input wide characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **wcstol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **wcstol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

e,f,g Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of the **wcstod** function. The corresponding argument shall be a pointer to floating.

s Matches a sequence of non-white-space wide characters. If no **l** qualifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

Otherwise, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** type large enough to accept the sequence and a terminating null wide character, which will be added automatically.

[Matches a nonempty sequence of wide characters from a set of expected characters (the *scanset*). If no **l** qualifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an **l** qualifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** type large enough to accept the sequence and a terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent wide characters in the **format** string, up to and including the matching right bracket wide character (**]**). The wide characters between the brackets (the *scanlist*) comprise the scanset, unless the wide character after the left bracket is a circumflex (**^**), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[]** or **[^]**, the right bracket wide character is in the scanlist and the next right bracket wide character is the matching right bracket that ends the specification; otherwise the first right bracket wide character is the one that ends the specification. If a **-** wide character is in the scanlist and is not the first, nor the second where the first wide character is a **^**, nor the last character, the behavior is implementation-defined.

c Matches a sequence of wide characters of the number specified by the field width (1 if no field width is present in the directive). If no **l** qualifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to a character array large enough to accept the sequence. No null character is added.

If an **l** qualifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** type large enough to accept the sequence. No null wide character is added.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fwprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the **%p** conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of wide characters read from the input stream so far by this call to the **fwscanf** function. Execution of a **%n** directive does not affect the assignment count returned at the completion of execution of the **fwscanf** function.

% Matches a single **%**; no conversion or assignment occurs. The complete conversion specification shall be **%%**

If a conversion specification is invalid, the behavior is undefined.[17]

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as, respectively, **e**, **g**, and **x**.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than **%n**, if any) is terminated with an input failure.

Trailing white space (including new-line wide characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

Returns

The **fwscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **fwscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Examples

1. The call:

```
#include <stdio.h>
#include <wchar.h>
/*...*/
int n, i; float x; wchar_t name[50];
n = fwscanf(stdin, L"%d%f%ls", &i, &x, name);
```

with the input line:

17. See ‘‘future library directions’’ (7.14).

```
25 54.32E-1 thompson
```

will assign to **n** the value **3**, to **i** the value **25**, to **x** the value **5.432**, and to **name** the sequence **thompson\0**.

2. The call:

```
#include <stdio.h>
#include <wchar.h>
/*...*/
int i; float x; double y;
fwscanf(stdin, L"%2d%f%*d %lf", &i, &x, &y);
```

with input:

```
56789 0123 56a72
```

will assign to **i** the value **56** and to **x** the value **789.0**, will skip past **0123**, and will assign to **y** the value **56.0**. The next wide character read from the input stream will be **a**.

Forward References: the **wcstod** function (7.17.4.1.1), the **wcstol** function (7.17.4.1.2), the **wcstoul** function (7.17.4.1.3), the **wcrtomb** function (7.17.6.3.3).

7.16.2.3 The wprintf function

Synopsis

```
#include <wchar.h>
int wprintf(const wchar_t *format, ...);
```

Description

The **wprintf** function is equivalent to **fwprintf** with the argument **stdout** interposed before the arguments to **wprintf**.

Returns

The **wprintf** function returns the number of wide characters transmitted, or a negative value if an output error occurred.

7.16.2.4 The wscanf function

Synopsis

```
#include <wchar.h>
int wscanf(const wchar_t *format, ...);
```

Description

The **wscanf** function is equivalent to **fwscanf** with the argument **stdin** interposed before the arguments to **wscanf**.

Returns

The **wscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **wscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.16.2.5 The `swprintf` function

Synopsis

```
#include <wchar.h>
int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);
```

Description

The **swprintf** function is equivalent to **fwprintf**, except that the argument **s** specifies an array of wide characters into which the generated output is to be written, rather than written to a stream. No more than **n** wide characters are written, including a terminating null wide character, which is always added (unless **n** is zero).

Returns

The **swprintf** function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if **n** or more wide characters were requested to be written.

7.16.2.6 The `swscanf` function

Synopsis

```
#include <wchar.h>
int swscanf(const wchar_t *s, const wchar_t *format, ...);
```

Description

The **swscanf** function is equivalent to **fwscanf**, except that the argument **s** specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the **fwscanf** function.

Returns

The **swscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **swscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.16.2.7 The `vfwprintf` function

Synopsis

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
```

Description

The **vfwprintf** function is equivalent to **fwprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfwprintf** function does not invoke the **va_end** macro.[18]

Returns

The **vfwprintf** function returns the number of wide characters transmitted, or a negative value if an output error occurred.

Example

The following shows the use of the **vfwprintf** function in a general error-reporting routine.

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

void error(char *function_name, wchar_t *format, ...)
{
     va_list args;

     va_start(args, format);
     /* print out name of function causing error */
     fwprintf(stderr, L"ERROR in %s: ", function_name);
     /* print out remainder of message */
     vfwprintf(stderr, format, args);
     va_end(args);
}
```

7.16.2.8 The vwprintf function

Synopsis

```
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t *format, va_list arg);
```

Description

The **vwprintf** function is equivalent to **wprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and

18. As the functions **vfwprintf**, **vswprintf**, and **vwprintf** invoke the **va_arg** macro, the value of **arg** after the return is unspecified.

possibly subsequent **va_arg** calls). The **vwprintf** function does not invoke the **va_end** macro.

Returns

The **vwprintf** function returns the number of wide characters transmitted, or a negative value if an output error occurred.

7.16.2.9 The vswprintf function

Synopsis

```
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t *s, size_t n, const wchar_t *format,
     va_list arg);
```

Description

The **vswprintf** function is equivalent to **swprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vswprintf** function does not invoke the **va_end** macro.

Returns

The **vswprintf** function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if **n** or more wide characters were requested to be generated.

7.16.3 Wide-character input/output functions

7.16.3.1 The fgetwc function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

Description

The **fgetwc** function obtains the next wide character (if present) from the input stream pointed to by **stream**, and advances the associated file position indicator for the stream (if defined).

Returns

The **fgetwc** function returns the next wide character from the input stream pointed to by **stream**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **fgetwc** returns **WEOF**. If a read error occurs, the error indicator for the stream is set and **fgetwc** returns **WEOF**. If an encoding error occurs (including too few bytes), the value of the macro **EILSEQ** is stored in **errno** and **fgetwc** returns **WEOF**.[19]

7.16.3.2 The `fgetws` function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
```

description

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

Returns

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

7.16.3.3 The `fputwc` function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t fputwc(wchar_t c, FILE *stream);
```

Description

The **fputwc** function writes the wide character specified by **c** to the output stream pointed to by **stream**, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns

The **fputwc** function returns the wide character written. If a write error occurs, the error indicator for the stream is set and **fputwc** returns **WEOF**. If an encoding error occurs, the value of the macro **EILSEQ** is stored in **errno** and **fputwc** returns **WEOF**.

19. An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions. Also, **errno** will be set to **EILSEQ** by input/output functions only if an encoding error occurs.

7.16.3.4 The **fputws** function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fputws(const wchar_t *s, FILE *stream);
```

Description

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

Returns

The **fputws** function returns **EOF** if a write or encoding error occurs; otherwise, it returns a nonnegative value.

7.16.3.5 The **getwc** function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

Description

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

Returns

The **getwc** function returns the next wide character from the input stream pointed to by **stream** or **WEOF**.

7.16.3.6 The **getwchar** function

Synopsis

```
#include <wchar.h>
wint_t getwchar(void);
```

Description

The **getwchar** function is equivalent to **getwc** with the argument **stdin**.

Returns

The **getwchar** function returns the next wide character from the input stream pointed to by **stdin** or **WEOF**.

7.16.3.7 The **putwc** function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t c, FILE *stream);
```

Description

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

Returns

The **putwc** function returns the wide character written or **WEOF**.

7.16.3.8 The **putwchar** function

Synopsis

```
#include <wchar.h>
wint_t putwchar(wchar_t c);
```

Description

The **putwchar** function is equivalent to **putwc** with the second argument **stdout**.

Returns

The **putwchar** function returns the character written or **WEOF**.

7.16.3.9 The **ungetwc** function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t c, FILE *stream);
```

Description

The **ungetwc** function pushes the wide character specified by **c** back onto the input stream pointed to by **stream**. The pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by **stream**) to a file positioning function (**fseek**, **fsetpos**, or **rewind**) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged.

One wide character of pushback is guaranteed, even if the call to the **ungetwc** function follows just after a call to a formatted wide character input function (**fwscanf** or **wscanf**). If the **ungetwc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of **c** equals that of the macro **WEOF**, the operation fails and the input stream is unchanged.

A successful call to the **ungetwc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back. For a text or binary stream, the value of its file position indicator after a successful call to the **ungetwc** function is unspecified until all pushed-back wide characters are read or discarded.

Returns

The **ungetwc** function returns the wide character pushed back, or **WEOF** if the operation fails.

7.16.3.10 The fwide function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

Description

The **fwide** function determines the orientation of the stream pointed to by **stream**. If **mode** is greater than zero, the function first attempts to make the stream wide oriented. If **mode** is less than zero, the function first attempts to make the stream byte oriented.[20] Otherwise, **mode** is zero and the function does not alter the orientation of the stream.

Returns

The **fwide** function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

20. If the orientation of the stream has already been determined, **fwide** does not change it.

7.16.4 General wide-string utilities

The header **`<wchar.h>`** declares a number of functions useful for wide-string manipulation. Various methods are used for determining the lengths of the arrays, but in all cases a **`wchar_t *`** argument points to the initial (lowest addressed) element of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

7.16.4.1 Wide-string numeric conversion functions

7.16.4.1.1 The `wcstod` function

Synopsis

```
#include <wchar.h>
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

Description

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

The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters in subclause 6.1.3.1, but no floating suffix. The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is empty or consists entirely of white space, or if the first non-white-space wide character is other than a sign, a digit, or a decimal-point wide character.

If the subject sequence has the expected form, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of subclause 6.1.3.1, except that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears, a decimal point is assumed to follow the last digit in the wide string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide string is stored in the object pointed to by **`endptr`**, provided that **`endptr`** is not a null pointer.

Additional implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **`nptr`** is stored in the object pointed to by **`endptr`**, provided that **endptr** is not a null pointer.

Returns

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

7.16.4.1.2 The `wcstol` function

Synopsis

```
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

Description

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

If the value of **base** is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters in subclause 6.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the wide characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

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

If the subject sequence has the expected form and the value of **base** is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of subclause 6.1.3.2. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Additional implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns

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

7.16.4.1.3 The wcstoul function

Synopsis

```
#include <wchar.h>
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr,
     int base);
```

Description

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

If the value of **base** is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters in subclause 6.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the wide characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

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

If the subject sequence has the expected form and the value of **base** is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of subclause 6.1.3.2. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Additional implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns

The **wcstoul** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **ULONG_MAX** is returned, and the value of the macro **ERANGE** is stored in **errno**.

7.16.4.2 Wide-string copying functions

7.16.4.2.1 The wcscpy function

Synopsis

```
#include <wchar.h>
wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);
```

Description

The **wcscpy** function copies the wide string pointed to by **s2** (including the terminating null wide character) into the array pointed to by **s1**.

Returns

The **wcscpy** function returns the value of **s1**.

7.16.4.2.2 The wcsncpy function

Synopsis

```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The **wcsncpy** function copies not more than **n** wide characters (those that follow a null wide character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**.[21]

If the array pointed to by **s2** is a wide string that is shorter than **n** wide characters, null wide characters are appended to the copy in the array pointed to by **s1**, until **n** wide characters in all have been written.

Returns

The **wcsncpy** function returns the value of **s1**.

7.16.4.3 Wide-string concatenation functions

7.16.4.3.1 The wcscat function

Synopsis

```
#include <wchar.h>
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
```

Description

The **wcscat** function appends a copy of the wide string pointed to by **s2** (including the terminating null wide character) to the end of the wide string pointed to by **s1**. The initial wide character of **s2** overwrites the null wide character at the end of **s1**.

Returns

The **wcscat** function returns the value of **s1**.

7.16.4.3.2 The wcsncat function

Synopsis

```
#include <wchar.h>
wchar_t *wcsncat(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The **wcsncat** function appends not more than **n** wide characters (a null wide character and those that follow it are not appended) from the array pointed to by **s2** to the end of the wide string pointed to by **s1**. The initial wide character of **s2** overwrites the null wide character at the end of **s1**. A terminating null wide character is always appended to the result.[22]

---

21. Thus, if there is no null wide character in the first **n** wide characters of the array pointed to by **s2**, the result will not be null-terminated.

22. Thus, the maximum number of wide characters that can end up in the array pointed to by **s1** is **wcslen(s1)+n+1**.

Returns

The **wcsncat** function returns the value of **s1**.

7.16.4.4 Wide-string comparison functions

Unless explicitly stated otherwise, the functions described in this subclause order two wide characters the same way as two integers of the underlying integral type designated by **wchar_t**.

7.16.4.4.1 The wcscmp function

Synopsis

```
#include <wchar.h>
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

Description

The **wcscmp** function compares the wide string pointed to by **s1** to the wide string pointed to by **s2**.

Returns

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

7.16.4.4.2 The wcscoll function

Synopsis

```
#include <wchar.h>
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

Description

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

Returns

The **wcscoll** function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by **s1** is greater than, equal to, or less than the wide string pointed to by **s2** when both are interpreted as appropriate to the current locale.

7.16.4.4.3 The wcsncmp function

Synopsis

```
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

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

Returns

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

7.16.4.4.4 The wcsxfrm function

Synopsis

```
#include <wchar.h>
size_t wcsxfrm(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The **wcsxfrm** function transforms the wide string pointed to by **s2** and places the resulting wide string into the array pointed to by **s1**. The transformation is such that if the **wcscmp** function is applied to two transformed wide strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **wcscoll** function applied to the same two original wide strings. No more than **n** wide characters are placed into the resulting array pointed to by **s1**, including the terminating null wide character. If **n** is zero, **s1** is permitted to be a null pointer.

Returns

The **wcsxfrm** function returns the length of the transformed wide string (not including the terminating null wide character). If the value returned is **n** or greater, the contents of the array pointed to by **s1** are indeterminate.

Example

The value of the following expression is the length of the array needed to hold the transformation of the wide string pointed to by **s**:

```
1 + wcsxfrm(NULL, s, 0)
```

7.16.4.5 Wide-string search functions

7.16.4.5.1 The wcschr function

Synopsis

```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Description

The **wcschr** function locates the first occurrence of **c** in the wide string pointed to by **s**. The terminating null wide character is considered to be part of the wide string.

Returns

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

7.16.4.5.2 The wcscspn function

Synopsis

```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description

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

Returns

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

7.16.4.5.3 The wcspbrk function

Synopsis

```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

Description

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

Returns

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

7.16.4.5.4 The wcsrchr function

Synopsis

```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

Description

The **wcsrchr** function locates the last occurrence of **c** in the wide string pointed to by **s**. The terminating null wide character is considered to be part of the wide string.

Returns

The **wcsrchr** function returns a pointer to the wide character, or a null pointer if **c** does not occur in the wide string.

7.16.4.5.5 The `wcsspn` function

Synopsis

```
#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

Description

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

Returns

The **`wcsspn`** function returns the length of the segment.

7.16.4.5.6 The `wcsstr` function

Synopsis

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

Description

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

Returns

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

7.16.4.5.7 The `wcstok` function

Synopsis

```
#include <wchar.h>
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2, wchar_t **ptr);
```

Description

A sequence of calls to the **`wcstok`** function breaks the wide string pointed to by **`s1`** into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by **`s2`**. The third argument points to a caller-provided **`wchar_t`** pointer into which the **`wcstok`** function stores information necessary for it to continue scanning the same wide string.

For the first call in the sequence, **`s1`** shall point to a wide string, while in subsequent calls for the same string, **`s1`** shall be a null pointer. If **`s1`** is a null pointer, the value pointed to by **`ptr`** shall match that stored by the previous call for the same wide string; otherwise the value pointed to by **`ptr`** is ignored. The separator wide string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the wide string pointed to by **s1** for the first wide character that is *not* contained in the current separator wide string pointed to by **s2**. If no such wide character is found, then there are no tokens in the wide string pointed to by **s1** and the **wcstok** function returns a null pointer. If such a wide character is found, it is the start of the first token.

The **wcstok** function then searches from there for a wide character that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by **s1**, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

In all cases, the **wcstok** function stores sufficient information in the pointer pointed to by **ptr** so that subsequent calls, with a null pointer for **s1** and the unmodified pointer value for **ptr**, shall start searching just past the element overwritten by a null wide character (if any).

Returns

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

Example

```
     #include <wchar.h>
     static wchar_t str1[] = L"?a???b,,,#c";
     static wchar_t str2[] = L"\t \t";
     wchar_t *t, *ptr1, *ptr2;

     t = wcstok(str1, L"?", &ptr1);   /* t points to the token
L"a" */
     t = wcstok(NULL, L",", &ptr1);   /* t points to the token
L"??b" */
     t = wcstok(str2, L" \t", &ptr2); /* t is a null pointer */
     t = wcstok(NULL, L"#,", &ptr1);  /* t points to the token
L"c" */
     t = wcstok(NULL, L"?", &ptr1);   /* t is a null pointer */
```

7.16.4.5.8 The wcslen function

Synopsis

```
     #include <wchar.h>
     size_t wcslen(const wchar_t *s);
```

Description

The **wcslen** function computes the length of the wide string pointed to by **s**.

Returns

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

7.16.4.6 Wide-character array functions

These functions operate on arrays of type **wchar_t** whose size is specified by a separate count argument. These functions are not affected by locale and all **wchar_t** values are treated identically. The null wide character and **wchar_t** values not corresponding to valid multibyte characters are not treated specially.

Unless explicitly stated otherwise, the functions described in this subclause order two wide characters the same way as two integers of the underlying integral type designated by **wchar_t**.

Where an argument declared as **size_t n** determines the length of the array for a function, **n** can have the value zero on a call to that function. Unless stated explicitly otherwise in the description of a particular function in this subclause, pointer arguments on such a call must still have valid values, as described in subclause 7.1.7. On such a call, a function that locates a wide character finds no occurrence, a function that compares two wide character sequences returns zero, and a function that copies wide characters copies zero wide characters.

7.16.4.6.1 The wmemchr function

Synopsis

```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

Description

The **wmemchr** function locates the first occurrence of **c** in the initial **n** wide characters of the object pointed to by **s**.

Returns

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

7.16.4.6.2 The wmemcmp function

Synopsis

```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

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

Returns

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

7.16.4.6.3 The wmemcpy function

Synopsis

```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The **wmemcpy** function copies **n** wide characters from the object pointed to by **s2** to the object pointed to by **s1**.

Returns

The **wmemcpy** function returns the value of **s1**.

7.16.4.6.4 The wmemmove function

Synopsis

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The **wmemmove** function copies **n** wide characters from the object pointed to by **s2** to the object pointed to by **s1**. Copying takes place as if the **n** wide characters from the object pointed to by **s2** are first copied into a temporary array of **n** wide characters that does not overlap the objects pointed to by **s1** or **s2**, and then the **n** wide characters from the temporary array are copied into the object pointed to by **s1**.

Returns

The **wmemmove** function returns the value of **s1**.

7.16.4.6.5 The wmemset function

Synopsis

```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Description

The **wmemset** function copies the value of **c** into each of the first **n** wide characters of the object pointed to by **s**.

Returns

The **wmemset** function returns the value of **s**.

7.16.5 The `wcsftime` function

Synopsis

```
#include <wchar.h>
size_t wcsftime(wchar_t *s, size_t maxsize,
     const wchar_t *format, const struct tm *timeptr);
```

Description

The **`wcsftime`** function is equivalent to the **`strftime`** function, except that:

- The argument **`s`** points to the initial element of an array of wide characters into which the generated output is to be placed.
- The argument **`maxsize`** indicates the limiting number of wide characters.
- The argument **`format`** is a wide string and the conversion specifiers are replaced by corresponding sequences of wide characters.
- The return value indicates the number of wide characters.

Returns

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

7.16.6 Extended multibyte and wide-character conversion

utilities

The header **`<wchar.h>`** declares an extended set of functions useful for conversion between multibyte characters and wide characters.

Most of the following functions — those that are listed as ‘‘restartable’’, subclauses 7.17.6.3 and 7.17.6.4 — take as a last argument a pointer to an object of type **`mbstate_t`** that is used to describe the current *conversion state* from a particular multibyte character sequence to a wide-character sequence (or the reverse) under the rules of a particular setting for the **`LC_CTYPE`** category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new multibyte character in the initial shift state. A zero-valued **`mbstate_t`** object is (at least) one way to describe an initial conversion state. A zero-valued **`mbstate_t`** object can be used to initiate conversion involving any multibyte character sequence, in any **`LC_CTYPE`** category setting. If an **`mbstate_t`** object has been altered by any of the functions described in this subclause, and is then used with a different multibyte character sequence, or in the other conversion direction, or with a different **`LC_CTYPE`** category setting than on earlier function calls, the behavior is undefined.[23]

On entry, each function takes the described conversion state (either internal or pointed to by **ps**) as current. The conversion state described by the pointed-to object is altered as needed to track the shift state, and the position within a multibyte character, for the associated multibyte character sequence.

7.16.6.1 Single-byte wide-character conversion functions

7.16.6.1.1 The btowc function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t btowc(int c);
```

Description

The **btowc** function determines whether **c** constitutes a valid (one-byte) multibyte character in the initial shift state.

Returns

The **btowc** returns **WEOF** if **c** has the value **EOF** or if **(unsigned char)c** does not constitute a valid (one-byte) multibyte character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

7.16.6.1.2 The wctob function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t c);
```

Description

The **wctob** function determines whether **c** corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

Returns

The **wctob** returns **EOF** if **c** does not correspond to a multibyte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character.

23. Thus a particular **mbstate_t** object can be used, for example, with both the **mbrtowc** and **mbsrtowcs** functions as long as they are used to step sequentially through the same multibyte character string.

7.16.6.2 The `mbsinit` function

Synopsis

```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

Description

If **`ps`** is not a null pointer, the **`mbsinit`** function determines whether the pointed-to **`mbstate_t`** object describes an initial conversion state.

Returns

The **`mbsinit`** function returns nonzero if **`ps`** is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, it returns zero.

7.16.6.3 Restartable multibyte/wide-character conversion functions

These functions differ from the corresponding multibyte character functions of subclause 7.11.7 (**`mblen`**, **`mbtowc`**, and **`wctomb`**) in that they have an extra parameter, **`ps`**, of type pointer to **`mbstate_t`** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If **`ps`** is a null pointer, each function uses its own internal **`mbstate_t`** object instead, which is initialized at program startup to the initial conversion state. The implementation behaves as if no library function calls these functions with a null pointer for **`ps`**.

Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

7.16.6.3.1 The `mbrlen` function

Synopsis

```
#include <wchar.h>
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
```

Description

The **`mbrlen`** function is equivalent to the call:

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

where ***`internal`*** is the **`mbstate_t`** object for the **`mbrlen`** function.

Returns

The **`mbrlen`** function returns **`(size_t)-2`**, **`(size_t)-1`**, a value between zero and **`n`**, inclusive.

Forward References: the **`mbrtowc`** function (7.17.6.3.2).

7.16.6.3.2 The `mbrtowc` function

Synopsis

```
#include <wchar.h>
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n,
     mbstate_t *ps);
```

Description

If **`s`** is a null pointer, the **`mbrtowc`** function is equivalent to the call:

```
mbrtowc(NULL, "", 1, ps)
```

In this case, the values of the parameters **`pwc`** and **`n`** are ignored.

If **`s`** is not a null pointer, the **`mbrtowc`** function inspects at most **`n`** bytes beginning with the byte pointed to by **`s`** to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is completed, it determines the value of the corresponding wide character and then, if **`pwc`** is not a null pointer, stores that value in the object pointed to by **`pwc`**. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

The **`mbrtowc`** function returns the first of the following that applies (given the current conversion state):

0 if the next **`n`** or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

positive if the next **`n`** or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)-2` if the next **`n`** bytes contribute to an incomplete (but potentially valid) multibyte character, and all **`n`** bytes have been processed (no value is stored).[24]

`(size_t)-1` if an encoding error occurs, in which case the next **`n`** or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **`EILSEQ`** is stored in **`errno`**, and the conversion state is undefined.

---

24. When **`n`** has at least the value of the **`MB_CUR_MAX`** macro, this case can only occur if **`s`** points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

7.16.6.3.3 The `wcrtomb` function

Synopsis

```
#include <wchar.h>
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
```

Description

If **s** is a null pointer, the **wcrtomb** function is equivalent to the call

wcrtomb(*buf*, L'\0', ps)

where ***buf*** is an internal buffer.

If **s** is not a null pointer, the **wcrtomb** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **wc** (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **wc** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

The **wcrtomb** function returns the number of bytes stored in the array object (including any shift sequences). When **wc** is not a valid wide character, an encoding error occurs: the function stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)-1**; the conversion state is undefined.

7.16.6.4 Restartable multibyte/wide-string conversion functions

These functions differ from the corresponding multibyte string functions of subclause 7.11.8 (**mbstowcs** and **wcstombs**) in that they have an extra parameter, **ps**, of type pointer to **mbstate_t** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If **ps** is a null pointer, each function uses its own internal **mbstate_t** object instead, which is initialized at program startup to the initial conversion state. The implementation behaves as if no library function calls these functions with a null pointer for **ps**.

Also unlike their corresponding functions, the conversion source parameter, **src**, has a pointer-to-pointer type. When the function is storing the results of conversions (that is, when **dst** is not a null pointer), the pointer object pointed to by this parameter is updated to reflect the amount of the source processed by that invocation.

7.16.6.4.1 The `mbsrtowcs` function

Synopsis

```
#include <wchar.h>
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len,
     mbstate_t *ps);
```

Description

The **mbsrtowcs** function converts a sequence of multibyte characters, beginning in the conversion state described by the object pointed to by **ps**, from the array indirectly pointed to by **src** into a sequence of corresponding wide characters. If **dst** is not a null pointer, the converted characters are stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if **dst** is not a null pointer) when **len** codes have been stored into the array pointed to by **dst**.[25] Each conversion takes place as if by a call to the **mbrtowc** function.

If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if **dst** is not a null pointer, the resulting state described is the initial conversion state.

Returns

If the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **mbsrtowcs** function stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)-1**; the conversion state is undefined. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null (if any).

7.16.6.4.2 The wcsrtombs function

Synopsis

```
#include <wchar.h>
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len,
    mbstate_t *ps);
```

Description

The **wcsrtombs** function converts a sequence of wide characters from the array indirectly pointed to by **src** into a sequence of corresponding multibyte characters, beginning in the conversion state described by the object pointed to by **ps**. If **dst** is not a null pointer, the converted characters are then stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases: when a code is reached that does not correspond to a valid multibyte character, or (if **dst** is not a null pointer) when the next multibyte character would exceed the limit of **len** total bytes to be stored into the array pointed to by **dst**. Each conversion takes place as if by a

25. Thus, the value of **len** is ignored if **dst** is a null pointer.

call to the **wcrtomb** function.[26]

If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Returns

If conversion stops because a code is reached that does not correspond to a valid multibyte character, an encoding error occurs: the **wcsrtombs** function stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)-1**; the conversion state is undefined. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, not including the terminating null (if any).

26. If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte.

--   --