Adding TR 19769 to the C Standard Library

Date: 2008-08-01
Document: N1326

6.4.4.4 Character constants Syntax add the **u**' *c-char-sequence* ' after ' *c-char-sequence* '

6.4.4.4 Character constants Syntax add the **U**' *c-char-sequence* ' after **L**' *c-char-sequence* '

character-constant:
'*c-char-sequence* **'**
u' c-char-sequence '
L'*c-char-sequence* **'**
U' *c-char-sequence* '

Also add the above to annex A, **A.1.5 Constants**

6.4.4.4 change the second sentence in the first paragraph in the Description to:
A wide character constant is the same, except prefixed by the letter **u**, **L**, or **U** respectively.

6.4.5 String literals Syntax add **u**" *s-char-sequence* " after " *s-char-sequence* "

6.4.5 String literals Syntax add **U**" s-char-sequence" after **L**" *s-char-sequence* "

string-literal:
" $s\text{-}char\text{-}sequence_{opt}$ "
u" $s\text{-}char\text{-}sequence_{opt}$ "
L" *s-char-sequence* $_{opt}$ "
U" *s-char-sequence* $_{opt}$ "

Also add the above to annex A, **A.1.6 String Literals**

Change the last sentence to the first paragraph on 6.4.5 Description to:
A *wide string literal* is the same, except prefixed by the letter **u**, **L**, or **U** respectively.

Add a paragraph to 6.5.4 Description:

String literals with the **u** or **U** format can be concatenated. If both strings have the same format, the resulting concatenated string has that format. If one string has no prefix, it is treated as a string of the same format as the other operand. (**u**"str" and **U**"str") Any other concatenations are implementation-defined. Here are some examples of valid concatenations:

```
u"a" u"b"→ u"ab"   U"a" U"b"→ U"ab"   L"a" L"b"→ L"ab"
u"a"  "b"→ u"ab"   U"a"  "b"→ U"ab"   L"a"  "b"→ L"ab"
 "a" u"b"→ u"ab"    "a" U"b"→ U"ab"    "a" L"b"→ L"ab"
```

Add to 7.2 paragraph #2 the header **<uchar.h>** between **<time.h>** and **<wchar.h>**

Replace 7.24 with 7.24 Unicode utilities **<uchar.h>**

Renumber 7.24 through 7.26

Add the following text to the new 7.24

The following two new typedefs, **char16_t** and **char32_t**

```
typedef     T1      char16_t;
typedef     T2      char32_t;
```

where *T1* has the same type as **uint_least16_t** and *T2* has the same type as **uint_least32_t**.

The macro
__STDC_UTF_16__
is defined if the values of the type **char16_t** are to be UTF-16 encoded. This allows the use of UTF-16 in **char16_t** even when **wchar_t** uses a non-Unicode encoding. In certain cases the compile-time conversion to UTF-16 may be restricted to members of the basic character set and universal character names (\U*nnnnnnnn* and \u*nnnn*) because for these the conversion to UTF-16 is defined unambiguously.

The macro
__STDC_UTF_32__
is defined if the values of the char32_t are to be UTD-32 encoded.

If the macro **__STDC_UTF_16__** is not defined, the encoding of **char16_t** is implementation defined. Similarly, if the macro **__STDC_UTF_32__** is not defined, the encoding of **char32_t** is implementation defined.

7.24.1 The mbrtoc16 function

Synopsis

```
#include <uchar.h>
size_t mbrtoc16(char16_t * restrict pc16,
   const char * restrict s,
   size_t n,
   mbstate_t * restrict ps);
```

Description

If **s** is a null pointer, the **mbrtoc16** function is equivalent to the call:

```
mbrtoc16(NULL, "", 1, ps)
```

In this case, the values of the parameters **pc16** and **n** are ignored.

If **s** is not a null pointer, the **mbrtoc16** 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 complete and valid, it determines the value of the corresponding wide character and then, if **pc16** is not a null pointer, stores that value in the object pointed to by **pc16**. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

The **mbrtoc16** 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).
between **1** *and* **n** *inclusive*	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)(-3)	if the multibyte sequence converted more than one corresponding **char16_t** character and not all these characters have yet been stored; the next character in the sequence has now been stored and no bytes from the input have been consumed by this call.
(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).[1]
(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 unspecified.

7.24.2 The c16rtomb function

Synopsis

```
#include <uchar.h>
size_t c16rtomb(char * restrict s,
   char16_t c16,
   mbstate_t * restrict ps);
```

Description

[1] 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).

If **s** is a null pointer, the **c16rtomb** function is equivalent to the call **c16rtomb(buf, L'\0', ps)** where **buf** is an internal buffer. If **s** is not a null pointer, the **c16rtomb** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **c16** (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **c16** 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 **c16rtomb** function returns the number of bytes stored in the array object; this may be 0 (including any shift sequences). When **c16** 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 unspecified.

7.24.3 The mbrtoc32 function

Synopsis

```
#include <uchar.h>
size_t mbrtoc32(char32_t * restrict pc32,
   const char * restrict s,
   size_t n,
   mbstate_t * restrict ps);
```

Description

If **s** is a null pointer, the **mbrtoc32** function is equivalent to the call:

```
mbrtoc32(NULL, "", 1, ps)
```

In this case, the values of the parameters **pc32** and **n** are ignored.

If **s** is not a null pointer, the **mbrtoc32** 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 complete and valid, it determines the value of the corresponding wide character and then, if **pc32** is not a null pointer, stores that value in the object pointed to by **pc32**. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

The **mbrtoc32** 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).
between **1** *and* **n** *inclusive*	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)(-3)	if the multibyte sequence converted more than one corresponding **char32_t** character and not all these characters have yet been stored; the next character in the sequence has now been stored and no bytes from the input have been consumed by this call.
(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).[2]
(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 unspecified.

7.24.4 The c32rtomb function

Synopsis

```
#include <uchar.h>
size_t c32rtomb(char * restrict s,
   char32_t c32,
   mbstate_t * restrict ps);
```

Description

If **s** is a null pointer, the **c32rtomb** function is equivalent to the call **c32rtomb(buf, L'\0', ps)** where **buf** is an internal buffer. If **s** is not a null pointer, the **c32rtomb** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **c32** (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **c32** 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 **c32rtomb** function returns the number of bytes stored in the array object; this may be 0 (including any shift sequences). When **c32** 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 unspecified.

---

[2] 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).