ISO/IEC/WG14 N1010

Extensions for the programming language C to support new character data types

VERSION FOR PDTR APPROVAL BALLOT

Contents

1 Introduction

The C language has matured over the last decades, yet the character concept has remained stable. Various code pages and multibyte libraries have been introduced in the past; however, the character data type in the C language has remained 8 bit based. Today, the introduction and the success of the Unicode/ISO10646 standard and of its implementation in modern computer languages creates ever increasing demands on the C language to give Unicode better support. This paper addresses the introduction of new character data types in the C language in order to support future character encoding forms, including Unicode.

The Unicode standard supports 3 encoding forms:

- UTF-8
- UTF-16
- UTF-32

Each encoding form has advantages and disadvantages, so that the choice of the encoding form should be left to the application. Currently, some C applications implement UTF-8 using *char* type, UTF-16 using *unsigned short* or *wchar_t*, and UTF-32 using *unsigned int* or *wchar_t*. The current situation, however, faces the following major problems:

- The size of *wchar_t* is implementation defined. While Unicode offers the possibility to write platform independent applications, *wchar_ t* does not offer platform portability for C applications or platform independent data format.
- There is no string literal for 16- or 32-bit based integer types, but the Unicode encoding forms require string literals.

It is sensible to give all the Unicode encoding forms appropriate data type support. UTF-8 is normally considered as one of character sets for *char*. This paper suggests the implementation of 16 and 32 bit based character data types: *char16_t* and *char32_t*. The new data types guarantee program portability through clearly defined character width. The encoding of the new data types should be as generic as possible in order to support not only Unicode but also future character encodings.

It is desirable in general that the C applications process strings rather than a character and character arrays. This paper does not specify the detail of library functions for the new data types, except one set of character conversion functions.

2 General

2.1 Scope

This Technical Report specifies two character data types as the extensions of the programming language C, specified by the international standard ISO/IEC 9899:1999.

2.2 References

The following standards contain provisions which, through reference in this text, constitute provisions of Technical Report. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred applies. Members of IEC and ISO maintain registers of current valid International Standards.

ISO/IEC 9899:1999, *Information technology – Programming languages, their environments an system software interfaces – Programming Language C.*

ISO/IEC 10646-1:2000, *Universal multi-octet character set – UCS – Part 1 : Architecture and Basic Multilingual Plane*

ISO/IEC 10646-2: 2000, *Universal multi-octet character set – UCS – Part21 : CJK Unified, Ideographs Supplementary plane, General Scripts and Symbols Plane, General Purpose Plane*

3 The new typedefs

This Technical Report introduces the following two new typedefs *char16_t* and *char32_t* :

```
typedef     uint_least16_t char16_t;
typedef     uint_least32_t char32_t;
```

The new typedefs guarantee the certain width of the data types whereas the width of *wchar_t* was implementation defined. The data values are unsigned while *char* could take signed values. This Technical Report also introduces the new header:

```
uchar.h
```

The new typedefs*, char16_t* and *char32_t,* are defined in **uchar.h**

4 Encoding

C99 subclause 6.10.8 specifies that the value of the macro __STDC_ISO_10646__ shall be "an integer constant of the form yyyymmL (for example, 199712L), intended to indicate that values of type *wchar_t* are the coded representations of the characters defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month." C99 subclause 6.4.5p5 specifies that wide string literals are initialized with a sequence of wide characters as defined by the *mbstowcs* function with an implementation-defined current locale. Analogue to this macro, two new macros will be introduced.

If the macro __STDC_UTF_16 is defined, the type char16_t shall have the UTF-16 encoding. This allows the use of UTF-16 in *char16_t* also 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.

If the macro __STDC_UTF_32__ is defined, the type char32_t shall have the UTF-32 encoding.

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

In absence of the mentioned macros, an implementation may define other macro's to indicate a different encoding; in this case the provisions of sections 5 and 6 of this document still hold, as they are independent of the actual encoding of the *char16_t* and *char32_t* characters.

5 String literals and character constants

5.1 String literals and character constants notations

The notations for string literals and character constants for *char16_t* are defined analogue to the wide character string literals and wide character constants:

u"*s-char-sequence*"

denotes a char16_t type string literal and initializes an array of char16_t. The corresponding character constant is denoted by

u'*c-char-sequence'*

and has the type *char16_t*. Likewise, the string literal and character constant for *char32_t* are,

U"*s-char-sequence*" and

U '*c-char-sequence'*.

5.2 The string concatenation

The new string literal formats (u"str" and U"str") should follow the same concatenation rules as the existing L"str" strings; i.e., when adjacent literals of the same format are concatenated the result is widened to the representation of the other string literal also if one of the adjacent literals is a “narrow” string. Here some examples:

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"

Any other catenations are implementation-defined (they might or might not be supported).

6 Library functions

Speaking in general, it is desirable to free the C applications from character-based operations and encourage string-based operations. The detail of the library for the new character data types should be left to the future enhancements of the C standard. This technical report specifies merely the 4 minimum character conversions among 3 character data types: *char, char16_t* and *char32_t*.

6.1 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)(-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.

[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).

6.2 The `c16rtomb` function

Synopsis

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

Description

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 (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.

6.3 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)(-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.

6.4 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

[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).

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 (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.

7 ANNEX A Unicode encoding forms: UTF-16, UTF-32

See Section 2.3 "Encoding Forms" and Section 3.8 "Transformations" in *The Unicode Standard*, Version 3.0. Addison Wesley, 2000.

Online Edition
Section 2.3 Encoding Forms
http://www.unicode.org/uni2book/ch02.pdf

Section 3.8 Transformations
http://www.unicode.org/uni2book/ch02.pdf

Technical Report
http://www.unicode.org/reports/tr19/

See also the Annex C of ISO10646-1.

Online Edition
http://anubis.dkuug.dk/JTC1/SC2/WG2/docs/n2005/n2005-2.doc