

Restartable and Non-Restartable Functions for Efficient Character Conversions

JeanHeyd Meneide <phdofthehouse@gmail.com>

October 1st, 2019

Document: n2436

Previous Revisions: n2431

Audience: WG14

Proposal Category: New Library Features

Target Audience: General Developers, Text Processing Developers

Latest Revision: https://thephd.github.io/vendor/future_cxx/papers/source/C - Efficient Character Conversions.html

Abstract:

Implementations firmly control what both the Wide Character and Multibyte Character literals are interpreted as for the encoding, as well as how they are treated at runtime by the Standard Library. While this control is fine, users of the Standard Library have no portability guarantees about how these library functions may behave, especially in the face of encodings that do not support each other's full codepage. And, despite additions to C11 for maybe-UTF16 and maybe-UTF32 encoded types, these functions only offer conversions of a single unit of information at a time, leaving orders of magnitude of performance on the table.

This paper proposes and explores additional library functionality to allow users to retrieve multibyte and wide character into a statically known encoding to enhance the ability to work with text.

1 Introduction and Motivation

C adopted conversion routines for the current active locale-derived/LC_TYPE-controlled/implementation-defined encoding for Multibyte (mb) Strings and Wide (wc) Strings. While the rationale for having such conversion routines to and from Multibyte and Wide strings in the C library are not explicitly stated in the documents, it is easy to derive the many benefits of a full ecosystem of both restarting (r) and non-restarting conversion routines for both single units and string-based bulk conversions for mb and wc strings. From ease of use with string literals to performance optimizations from bulk processing with vectorization and SIMD operations, the mbs(r)towcs — and vice-versa — granted a rich and fertile ground upon which C library developers took advantage of platform amenities, encoding specifics, and hardware support to provide useful and fast abstractions upon which encoding-aware applications could build.

Unfortunately, none of these API designs were granted to char16_t (c16) or char32_t (c32) conversion functions. Nor were they given a way to work with a well-defined 8-bit multibyte encoding such as UTF8 without having to first pin it down with platform-specific setlocale(...) calls. This has resulted in a series of extremely vexing problems when trying to write a portable, reliable C library code that is not locked to a specific vendor.

This paper looks at the problems, and then proposes a solution (without C Standard wording) with the goal of hoping to arrive at a solution that is worth implementing for the C Standard Library.

1.1 Problem 1: Lack of Portability

Already, Windows, z/OS, and POSIX platforms greatly differ in what they offer for `char`-typed, Multibyte string encodings. EBCDIC is still in play after many decades. Windows's Active Code Page functionality on its machine prevents portability even within its own ecosystem. Platforms where LANG environment variables control functionality make communication between even processes on the same hardware a silent and often unforeseen gamble for library developers. Using functions which convert to/from `mbs` make it impossible to have stability guarantees not only between platforms, but for individual machines. Sometimes even cross-process communication becomes exceedingly problematic without opting into a serious amount of platform-specific or vendor-specific code and functionality to lock encodings in, harming the portability of C code greatly.

`wchar_t` does not fare better. By definition, a wide character type must be capable of holding the entire character set in a single unit of `wchar_t`. Reality, however, is different: this has been a fundamental impossibility for decades for implementers that switched to 16-bit UCS-2 early. IBM machines persist with this issue for all 32-bit builds, though some IBM platforms took advantage of the 64-bit change to do an ABI break and use UTF32 like other Linux distributions settled on. Even if one were to know this knowledge about IBM and program exclusively on their machines, certain IBM platforms can still end up in a situation where `wchar_t` is neither 32-bit UTF32 or 16-bit UCS-2/UTF16: the encoding can change to something else in certain Chinese locales, becoming completely different.

Windows is permanently stuck on having to explicitly detail that its implementation is “16-bit, UCS-2 as per the standard”, before explicitly informing developers to use vendor-specific

`WideCharToMultibyte/MultibyteToWideChar` to handle UTF16-encoded characters in `wchar_t`.

These solutions provide ways to achieve a local maxima for a specific vendor or platform. Unfortunately, this comes at the extreme cost of portability: the code has no guarantee it will work anywhere but your machine, and in a world that is increasingly interconnected by devices that interface with networks it makes sharing both data and code troublesome and hard to work with.

1.2 Problem 2: What is the Encoding?

With `setlocale` and `getlocale` only responding to and returning implementation-defined `(const)char*`, there is no way to portably determine what the locale (and any associated encoding) should or should not be. The typical solution for this has been to code and program only for what is guaranteed by the Standard as what is in the Basic Character Set. While this works fine for source code itself, this produces an extremely hostile environment:

- conversion functions in the standard mangle and truncate data in (sometimes troubling, sometimes hilarious) fashion;
- programs which are not careful to meticulously track encoding of incoming text often lose the ability to understand that text;
- programmers can never trust the platform will support even the Latin characters in any representation of data beyond the 7th bit of a byte;
- and, interchange between cultures with different default encodings makes it impossible to communicate with others without entirely forsaking the standard library.

Abandoning the C **Standard** Library – to get **standard** behavior across platforms – is an exceedingly bitter pill to have to swallow as an enthusiastic C developer.

1.3 Problem 3: Performance

The current version of the C Standard includes functions which attempt to alleviate Problems 1 and 2 by providing conversions from the per-process (and sometimes per-thread), locale-sensitive black box encoding of multibyte `char*` strings. They do this by providing conversions to `char16_t` units or `char32_t` units with `mbrtoc(16|32)` and `c(16|32)rtomb` functions. We will for a brief moment ignore the presence of the `__STD_C_UTF16__` and `__STD_C_UTF32__` macros and assume the two types mean that string literals and library functions convert to and from UTF16 and UTF32 respectively. We will also ignore that `wchar_t`'s encoding – which is just as locale-sensitive and unknown at compile and runtime as `char`'s encoding is – has no such conversion functions. These givens make it possible to say that we, as C programmers, have 2 known encodings which we can use to shepherd data into a stable state for manipulation and processing as library developers.

Even with that knowledge, these one-unit-at-a-time conversions functions are slower than they should be.

On many platforms, these one-at-a-time function calls come from the operating system, dynamically loaded libraries, or other places which otherwise inhibit compiler observation and optimizer inspection. Attempts to vectorize code or unroll loops built around these functions is thoroughly thwarted by this. Building static libraries or from source is very often a non-starter for many platforms. Since the encoding used for multibyte strings and wide strings are controlled by the implementation, it becomes increasingly difficult to provide the functionality to convert long segments of data with decent performance characteristics without needing to opt into vendor or platform specific tricks.

1.4 Problem 4: `wchar_t` cannot roundtrip

With no `wctoc32` or `wctoc16` functions, the only way to convert a wide character or wide character string to a program-controlled, statically known encoding is to first invoke the wide character to multibyte function, and then invoke the multibyte function to either `char16_t` or `char32_t`.

This means that even if we have a well-behaved `wchar_t` that is not sensitive to the locale (e.g., on Windows machines), we lose data if the locale-controlled `char` encoding is not set to something that can handle all incoming code unit sequences. The locale-based encoding in a program can thus tank what is simply meant to be a pass-through encoding from `wchar_t` to `char16_t`/`char32_t`, all because the only Standards-compliant conversion channels data through the locale-based multibyte encoding `mb(s)(r)toX` functions.

For example, it was fundamentally impossible to engage in a successful conversion from `wchar_t` strings to `char` multibyte strings on Windows using the C Standard Library. Until a very recent Windows 10 update, UTF8 could **not** be set as the active system codepage either programmatically or through an experimental, deeply-buried setting. This has changed with Windows Version 1903 (May 2019 Update), but the problems do not stop there.

Because other library functions can be used to change or alter the locale in some manner, it once again becomes impossible to have a portable, compliant program with deterministic behavior if even one library changes the locale of the program, let alone if the encoding or locale is unexpected by the developer because they do not know of that culture or its locale setting. This hidden state is nearly impossible to account for, and ends up with software systems that cannot properly handle text in a meaningful way without abandoning C's encoding facilities, relying on vendor-specific extensions/encodings/tools, or confining one's program to only the 7-bit plane of existence.

1.5 Motivation

In short, the problems C developers face today with respect to encoding and dealing with vendor and platform-specific black boxes is a staggering trifecta: non-portability between processes running on the same physical hardware, performance degradation from using standard facilities, and potentially having a locale changed out from under your program to prevent roundtripping.

This serves as the core motivation for this proposal.

2 Prior Art

The Small Device C Compiler (SDCC) has already begun some of this work. One of its principle contributors, Philip K. Krause, wrote papers addressing exactly this problem[\[1\]](#). Krause’s work focuses entirely on non-restartable conversions from Multibyte Strings to `char16_t`. and `char32_t`. There is no need for a conversion to a UTF8 `char` style string for SDCC, since the Multibyte String in SDCC is always UTF8. This means that `mbstoc16s` and `mbstoc32s` and the “reverse direction” functions encompass an entire ecosystem of UTF8, UTF16, and UTF32.

While this is good for SDCC, this is not quite enough for other developers who attempt to write code in a cross-platform manner. While the non-restartable functions can save quite a bit of code size (see [\[1\]](#)), unfortunately there are many encodings which are not as nice and require state to be processed correctly (e.g., Shift JIS and other ISO-2022 encodings). Not being able to retain that state between potential calls in a `mbstate_t` is detrimental to the ability to move forward with any encoding endeavor that wishes to bridge the gap between these disparate platform encodings and the current locale.

SDCC’s work is still important, however: it demonstrates that these functions are implementable, even for small devices. With additional work being done to implement them for other platforms, there is strong evidence that this can be implemented in a cross-platform manner and thusly is suitable for the Standard Library.

3 Proposed Changes

To understand what this paper proposes, an explanation of the current landscape is in order. The below table is meant to be read as being `{row}(r)to{column}`. The symbols provide the following information:

- ✓: Function exists in both its restartable (function name has the indicative `r` in it) and non-restartable form.
- R: Function exists only in its restartable form.
- ✗: Function does not exist at all.
- P: Modifying marker indicates intention to standardize either the restartable function (R) or both restartable and non-restartable functions (✓).

Here is what exists in the C Standard Library so far:

| | mb | wc | mbs | wcs | c8 | c16 | c32 | c8s | c16s | c32s |
|------|----|----|-----|-----|----|-----|-----|-----|------|------|
| mb | — | ✓ | | | ✗ | R | R | | | |
| wc | ✓ | — | | | ✗ | ✗ | ✗ | | | |
| mbs | | | — | ✓ | | | | ✗ | ✗ | ✗ |
| wcs | | | ✓ | — | | | | ✗ | ✗ | ✗ |
| c8 | ✗ | ✗ | | | — | ✗ | ✗ | | | |
| c16 | R | ✗ | | | ✗ | — | ✗ | | | |
| c32 | R | ✗ | | | ✗ | ✗ | — | | | |
| c8s | | | ✗ | ✗ | | | | — | ✗ | ✗ |
| c16s | | | ✗ | ✗ | | | | ✗ | — | ✗ |
| c32s | | | ✗ | ✗ | | | | ✗ | ✗ | — |

To support getting data losslessly out of `wchar_t` and `char` strings controlled firmly by the implementation – and back into those types if the code units in the characters are supported –, the following functionality is proposed:

| | mb | wc | mbs | wcs | c8 | c16 | c32 | c8s | c16s | c32s |
|------|---|---|---|---|---|---|---|---|---|---|
| mb | — | ✓ | | | P R | R | R | | | |
| wc | ✓ | — | | | P R | P R | P R | | | |
| mbs | | | — | ✓ | | | | P ✓ | P ✓ | P ✓ |
| wcs | | | ✓ | — | | | | P ✓ | P ✓ | P ✓ |
| c8 | P R | P R | | | — | ✗ | ✗ | | | |
| c16 | R | P R | | | ✗ | — | ✗ | | | |
| c32 | R | P R | | | ✗ | ✗ | — | | | |
| c8s | | | P ✓ | P ✓ | | | | — | ✗ | ✗ |
| c16s | | | P ✓ | P ✓ | | | | ✗ | — | ✗ |
| c32s | | | P ✓ | P ✓ | | | | ✗ | ✗ | — |

In particular, it is imperative to recognize that the implementation is the “sole proprietor” of the wide character (`wc`) and Multibyte (`mb`) encodings for its string literals (compiler) and library functions (standard library).

3.1 Single-Unit Functions

Focus should be applied on adding the one-at-a-time functions for `char` and `wchar_t`, which begets the start of this proposal:

- Multibyte Character:
 - `mbrtoc8` and `c8rtomb`
- Wide Character:
 - `wcrtoc8` and `c8rtowc`
 - `wcrtoc16` and `c16rtowc`
 - `wcrtoc32` and `c32rtowc`

Only the “r” (restarting) versions of these functions are proposed here because otherwise single code unit conversions would not be able to respect multiple code units of either `char16_t` or `char32_t`. For more information about multi-unit encodings and the trouble that comes with not using a restartable version and not returning sufficient information, see the discussion related to [N1991](#) and [DR488\[2\]](#).

The forms of such functions would be as follows:

```
/* Multibyte Character, Single Unit (UTF8): */
size_t mbrtoc8(char* pc8, const char* src, size_t src_len, mbstate_t* state);
size_t c8rtomb(char* pc, const char* src, size_t src_len, mbstate_t* state);

/* Wide Character, Single Unit: */
size_t wcrtocX(charX_t* pcX, const wchar_t* src, size_t src_len, mbstate_t* state);
size_t cXrtowc(wchar_t* pwc, const charX_t* src, size_t src_len, mbstate_t* state);
```

where x and charX_t is one of { 8, char }, { 16, char16_t }, or { 32, char32_t } for the function's specification.

3.2 Multi-Unit Functions

Additionally, the following is also proposed:

- Multibyte Character Strings:
 - mbstoc8s and c8stombs
 - mbsrtoc8s and c8srtombs
 - mbstoc16s and c16stombs
 - mbsrtoc16s and c16srtombs
 - mbstoc32s and c32stombs
 - mbsrtoc32s and c32srtombs
- Wide Character Strings:
 - wstoc8s and c8stowcs
 - wsrtoc8s and c8srtowcs
 - wstoc16s and c16stowcs
 - wsrtoc16s and c16srtowcs
 - wstoc32s and c32stowcs
 - wsrtoc32s and c32srtowcs

The functions follow the same conventions as their counterparts, mbstowcs and wstombs (or mbsrtowcs and wsrtombs, for the restartable versions). These allow for the implementation to bulk-convert to and from a statically-known encoding. Bulk conversions has significant performance benefits in both C and C++ code: see [4] and [5] (both authors have shown that their code can be ported to use a C interface or just be written directly in C itself).

The forms of such functions would be as follows:


```
/* Multibyte Character Strings: */
size_t mbstocXs(charX_t* dest, const char* src, size_t dest_len);
size_t cXstombs(char* dest, const charX_t* src, size_t dest_len);
size_t mbsrtocXs(charX_t* dest, const char** src, size_t dest_len, mbstate_t* state);
size_t cXsrtombs(char* dest, const charX_t** src, size_t dest_len, mbstate_t* state);

/* Wide Character Strings: */
size_t wstocXs(charX_t* dest, const wchar_t* src, size_t dest_len);
size_t cXstowcs(wchar_t* dest, const charX_t* src, size_t dest_len);
size_t wsrtocXs(charX_t* dest, const wchar_t** src, size_t dest_len, mbstate_t* state);
size_t cXsrtowcs(wchar_t* dest, const charX_t** src, size_t dest_len, mbstate_t* state);
```

where x and charX_t is one of { 8, char }, { 16, char16_t }, or { 32, char32_t } for the function's specification.

3.3 What about UTF{X} UTF{Y} functions?

Function interconverting between different Unicode Transformation Formats are not proposed here because – while useful – both sides of the encoding are statically known by the developer. The C Standard only wants to consider functionality strictly in the case where the implementation has more information / private information that the developer cannot access in a well-defined and standard manner. A developer can write their own Unicode Transformation Format conversion routines and get them completely right, whereas a developer cannot write the Wide Character and Multibyte Character functions without incredible heroics and/or error-prone assumptions.



This brings up an interesting point, however: if `__STD_C_UTF16__` and `__STD_C_UTF32__` both exist, does that not mean the implementation controls what `c16` and `c32` mean? This is true, **however**: within a (admittedly limited) survey of implementations, there has been no suggestion or report of an implementation which does not use UTF16 and UTF32 for their `char16_t` and `char32_t` literals, respectively. This motivation was, in fact, why a paper percolating through the WG21 Committee – [p1041 “Make `char16_t`/`char32_t` literals be UTF16/UTF32”](#)[6] – was accepted. If this changes, then the conversion functions `c{x}toc{y}` marked with an  will become important.

Thankfully, that does not seem to be the case at this time. If such changes or such an implementation is demonstrated, these functions can be added to what should be added.

3.4 Potential Extension: Sized Conversion Functions

Following the conventions of the string-based conversion functions already present, the above functions will use null termination as a marker for stopping. Many streams of text data today have embedded nulls in them, and have thusly required many creative solutions for avoiding embedded nulls (including encodings like Modified UTF-8 (MUTF8)). Thusly, as an extension for this proposal targeting the Standard Library, sized versions of the above functions which take a `size_t` are also proposed. This would specifies the number of code units in the source string.

Previously, sized functions for certain string operations were attempted by trying to duplicate current library functionality but with an `R_SIZE_MAX`-respecting parameter introduced the C 11 Standard, Annex K (for functions like `strncpy_s`). While the intention and rationale (N1570, §K.3.2 in [3]) made it explicitly clear the goal was to prevent potential size errors when going from a signed number to `size_t` and promote safety, the effect of such changes was different. `R_SIZE_MAX` values on certain platforms were restrictively tiny, taking payloads of reasonable sizes but still rejecting them. C programmers used to developing on certain platforms would use these functions in one area, port that code to another platform, and then would experience what amounted to a Denial of Service as their payloads exceeded the restrictively small `R_SIZE_MAX` values.

Given Annex K’s history and issues, this paper does not propose to implement anything like the `rsize` functions. Instead, this paper would like to promote `size_t`-sized function for all of the above currently existing () and desired () functions in the above table. Particularly:

- Multibyte Character Strings:
 - `mbsntoc8s` and `c8sntombs`
 - `mbsnrtoc8s` and `c8snrtombs`
 - `mbsntoc16s` and `c16sntombs`
 - `mbsnrtoc16s` and `c16snrtombs`
 - `mbsntoc32s` and `c32sntombs`
 - `mbsnrtoc32s` and `c32snrtombs`

- Wide Character Strings:
 - `wcsntoc8s` and `c8sntowcs`
 - `wcsnrtoc8s` and `c8snrtowcs`
 - `wcsntoc16s` and `c16sntowcs`
 - `wcsnrtoc16s` and `c16snrtowcs`
 - `wcsntoc32s` and `c32sntowcs`
 - `wcsnrtoc32s` and `c32snrtowcs`

The forms of such functions would be as follows:

```
/* Multibyte Character Strings: */
size_t mbsntocXs(charX_t* dest, const char* src, size_t dest_len, size_t src_len);
size_t cXsntombs(char* dest, const charX_t* src, size_t dest_len, size_t src_len);
size_t mbsnrtocXs(charX_t* dest, const char** src, size_t dest_len, size_t src_len,
    mbstate_t* state);
size_t cXsnrtombs(char* dest, const charX_t** src, size_t dest_len, size_t src_len,
    mbstate_t* state);

/* Wide Character Strings: */
size_t wcsntocXs(charX_t* dest, const wchar_t* src, size_t dest_len);
size_t cXsntowcs(wchar_t* dest, const charX_t* src, size_t dest_len);
size_t wcsnrtocXs(charX_t* dest, const wchar_t** src, size_t dest_len, size_t src_len,
    mbstate_t* state);
size_t cXsnrtowcs(wchar_t* dest, const charX_t** src, size_t dest_len, size_t src_len,
    mbstate_t* state);
```

where `x` and `charX_t` is one of `{ 8, char }`, `{ 16, char16_t }`, or `{ 32, char32_t }` for the function's specification. Similar additions can be made for the currently existing `mbs(r)towcs` and `wcs(r)tombs` functions as well.

4 Conclusion

This is a lot of functionality to ask for. Therefore, this paper was written mostly to obtain the Committee's general feedback about the ideas present here. An independent library implementation^[7] is underway, and already this implementation that has come up against all of the issues above. The goal is to submit patches to glibc, musl libc, and other Standard Library implementations when the time comes. However, getting early feedback from the C Committee about this functionality is crucial to ensuring that hardworking contributors to these open source implementations do not feel as if their time is being wasted by pursuing this functionality.

5 Acknowledgements

Thank you to Philipp K. Krause for responding to the e-mails of a newcomer to matters of C and providing me with helpful guidance. Thank you to Rajan Bhakta, Daniel Plakosh, and David Keaton for guidance on how to submit these papers and get started in WG14. Thank you to Tom Honermann for lighting the passionate fire for proper text handling in me for not just C++, but for our sibling language C.

6 References

- [1]: Philip K. Krause. N2282: Additional multibyte/wide string conversion functions. June 2018. Published: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2282.htm>.
- [2]: WG14. Clarification Request Summary for C11, Version 1.13. October 2017. Published: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2244.htm>.
- [3]: ISO/IEC, WG14. Programming Languages - C (Committee Draft). April 12, 2011. Published: <http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf>.
- [4]: Henri Sivonen. `encoding_rs`: a Web-Compatible Character Encoding Library in Rust. December 2018. Published: https://hsivonen.fi/encoding_rs/#results.
- [5]: Bob Steagall. Fast Conversion From UTF-8 with C++, DFAs, an SSE Intrinsics. September 2018. Published: <https://www.youtube.com/watch?v=5FQ87-Ecb-A>
- [6]: Robot Martinho Fernandes. p1041. February 2019. Published: <https://wg21.link/p1041>.
- [7]: JeanHeyd Meneide. Ooficode (alpha version). September 2019. Published: <https://github.com/ThePhD/ooficode/tree/develop/source>.

May the Tower of Babel's curse be defeated.