Restartable and Non-Restartable Functions for Efficient Character Conversions

JeanHeyd Meneide <phdofthehouse@gmail.com>

September 23rd, 2019

Document: n2431
Audience: WG14
Proposal Category: New Library Features
Target Audience: General Developers, Text Processing Developers

Abstract:

Implementations firmly control what both 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.

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`) functions, let alone provide 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 it has come to trying to write 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.

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 on its own platforms. Environments where LANG controls functionality make communication between even processes 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`: this has been a fundamental impossibility for decades for implementers which switched to 16-bit UCS-2 early. IBM machines persist with this issue for all 32-bit builds, though on some platforms took advantage of the 64-bit change to do an ABI break and use UTF32 like making Linux distributions settled on. This, of course, still does not help: a developer programming exclusively on IBM machines 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 the bare metal it makes sharing both data and code troublesome and hard to work with.

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;
- 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.

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, locale-sensitive black box encoding of multibyte `char*` strings to either `char16_t` strings 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.

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, thwarting attempts to inline function content, vectorize code or unroll loops built around these functions. 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.

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. In fact, this was fundamentally impossible to perform on Windows 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 until Windows Version 1903 (May 2019 Update).

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. 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.

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.

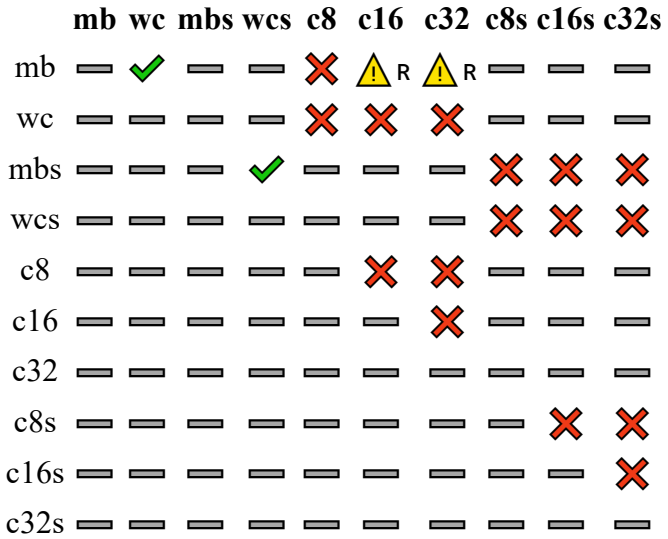
This serves as the core motivation for this proposal.

Prior Art

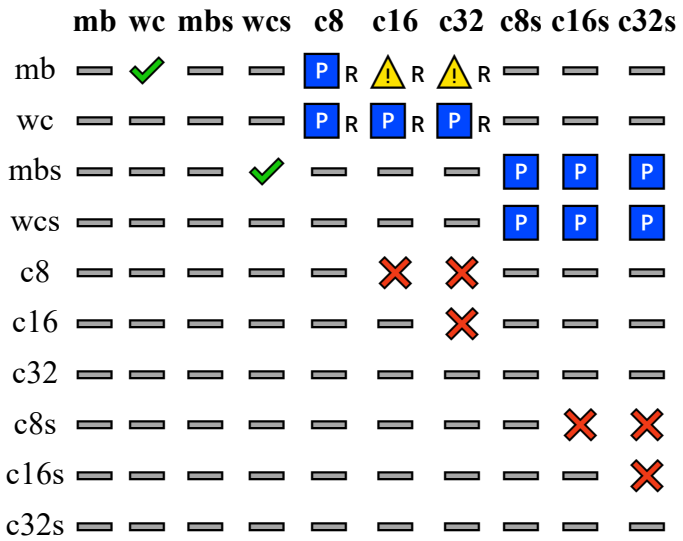
The Small Device C Compiler (SDCC) has already begun some of this work. One of its principle contributors, Philip K. Krause, has wrote papers addressing exactly this problem[1].

Proposed Changes

An observation of the current landscape reveals that we have the immediately following table of functionality. Everything below the diagonal is duplicate, so only the upper triangular portion of this matrix is filled out:



The proposed functionality that we need to at the very minimum support getting data losslessly out of `wchar_t` and `char` strings controlled firmly by the implementation is as follows:



In particular, we 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). We propose, then, to focus on adding the one-at-a-time functions for `char` and `wchar_t`:

- 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`. See the discussion related to N1991 and DR488[2]. Additionally, we also propose the following:

- Multibyte Character Strings:
    - `mbstoc8s` and `c8stombs`
    - `mbsrtoc8s` and `c8srtombs`
    - `mbstoc16s` and `c16stombs`
    - `mbsrtoc16s` and `c16srtombs`
    - `mbstoc32s` and `c32stombs`
    - `mbsrtoc32s` and `c32srtombs`
- Wide Character Strings:
    - `wcstoc8s` and `c8stowcs`
    - `wcsrtoc8s` and `c8srtowcs`
    - `wcstoc16s` and `c16stowcs`
    - `wcsrtoc16s` and `c16srtowcs`
    - `wcstoc32s` and `c32stowcs`
    - `wcsrtoc32s` and `c32srtowcs`

The functions follow the same conventions as their counterparts, `mbstowcs` and `wcstombs` (or `mbsrtowcs` and `wcsrtombs`, 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).

What about UTF{X} 🔄 UTF{Y} functions?

We do not propose these functions because – while useful – both sides of the encoding are statically known by the user to be of this. We only want to consider functionality strictly in the case where the implementation has more information / private information that the user cannot access in a well-defined and standard manner. A user can write their own Unicode Transformation Format.

Within my incredibly limited purview, 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. 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.

Addendum: Sized Conversion Functions

If we follow the conventions of the string-based conversion functions already present, they 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, we also propose all of the above functions, but as with an explicitly sized version with takes a `size_t` that specifies the number of code units in the source string.

Previously, sized functions for certain string operations was attempted by trying to duplicate current library

functionality but with an `RSIZE_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. `RSIZE_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 `RSIZE_MAX` values that were restrictively tiny.

Therefore, we have no desire to introduce functions like Annex K's security features into the C standard. Instead, we simply request a `size_t`-sized function for all of the above currently existing (✔) and desired (🅿) functions in the above table.

Conclusion

This is a lot of functionality to ask for. Therefore, while this paper was written and submitted, I wanted to get the Committee's general feedback about the ideas present here. I am working on an independent library implementation[6] that has come up against all of the issues above, with goals to submit patches to glibc, musl, 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 these large and hardworking open source implementations do not feel as if I am wasting their time by pursuing this functionality.

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 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.

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]: 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.