Restartable and Non-Restartable Functions for Efficient Character Conversions | r4

JeanHeyd Meneide <phdofthehouse@gmail.com>

Shepherd (Shepherd's Oasis) <shepherd@soasis.org>

October 27th, 2020

Document: n2595
Previous Revisions: n2440, n2431, n2500
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/n2595.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 Changelog

1.1 Revision 3 - October 27th, 2020

— Completely Reformulate Paper based on community, musl-libc, and glibc feedback.
— Completely rewrite every section past Proposed Changes, and change many more.

1.2 Revision 0-2 - March 2nd, 2020

— Introduce new functions and gather consensus to move forward.
— Attempt to implement in other standard libraries and gather feedback.

2 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.

2.1 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.

2.2 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.

2.3 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.

2.4 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 UTF 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(s)` 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.

No dedicated UTF-8 support (the standard mandates no specific encodings or charsets) leaves developers to write the routines themselves. Sometimes worse, roundtrip it through the locale after forcing a change to a UTF-8 locale, which may not be supported. While the non-restartable functions can save quite a bit of code size, 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.

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 just 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: the result is 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.

2.5 Problem 5: The C Standard Cannot Handle Existing Practice

The C standard does not allow a wide variety of encodings that implementations have already crammed into their backing locale blocks to work, resulting in the abandonment of locale-related text facilities by those with double-byte character sets, primarily from East Asia. For example, there is a serious bug that cannot be fixed without non-conforming, broken behavior[1]:

> …
>
> This call writes the second Unicode code point, but does not consume any input. 0 is returned since no input is consumed. According to the C standard, a return of 0 is reserved for when a null character is written, but since the C standard doesn't acknowledge the existence of characters that can't be represented in a single `wchar_t`, we're already operating outside the scope of the standard.

The standard cannot handle encodings that must return two or more `wchar_t` for however many – up to `MB_MAX_LEN` – it consumes. This is even for when the target `wchar_t` "wide execution" encoding is UTF-32; this is a **fundamental limitation of the C Standard Library that is absolutely insurmountable by the current specification**. This is exacerbated by the standard's insistence that a single `wchar_t` must be capable of representing all characters as a single element, a philosophy which has been bled into the relevant interfaces such as `mbrtowc` and other `*wc*` related types. As the values cannot be properly represented in the standard, this leaves people to either make stuff up or abandon it altogether:

2.6 In Summary

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.

3 Prior Art

There are many sources of prior art for the desired feature set. Some functions (with fixes) were implemented directly in implementations, embedded and otherwise. Others rely exclusively platform-specific code in both Windows and POSIX implementations. Others have cross-platform libraries that work across a myriad of platforms, such as ICU or iconv. We discuss the most diverse and exemplary implementations.

3.1 Standard C

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

— ✓: Function exists in both its restartable (function name has the indicative `r` in it) and its canonical non-restartable form (`{row}to{column}` and `{row}rto{column}`).
— R: Function exists only in its "restartable" form (`{row}rto{column}`).
— ✗: Function does not exist at all.

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			✗	✗				✗	✗	—

There is a lot of missing functionality here in this table, and it is important to note that a large amount of this comes from both not being willing to standardize more than the bare minimum and not having a cohesive vision for improving encoding conversions in the C Standard. Notably, string-based `{prefix}s` functions are missing, leaving performance-oriented multi-unit conversions out of the standard. There are also severe API flaws in the C standard, as discussed above.

3.2 Win32

`WideCharToMultiByte` and `MultiByteToWideChar` are the APIs of choice for those in Win32 environments to get to and from the run-time execution encoding and – if it matches – the translation-time execution encoding. Unfortunately, these APIs are locked within the Windows ecosystem entirely as they are not available as a standalone library. Furthermore, as an operating system Windows exclusively controls what it can and cannot convert from and to; some of these functions power the underlying portions of the character conversion functions in their Standard Library, but they notably truncate multi-code-unit characters for their UTF-16 `wchar_t`. This produces a broken, deprecated UCS-2 encoding when e.g. `mbrtowc` is used instead of directly relying on the operating system functionality, making the C standard’s functions of dubious use.

3.3 `nl_langinfo`

`nl_langinfo` is a POSIX function that returns various pieces of information based on an enumerated input and some extra parameters. It has been suggested that this be standardized over anything else, to make it easier to determine what to do with a given locale.

The first problem with this is it returns a string-based identifier that can be whatever an implementation decides it should be. This makes `nl_langinfo` is no better than `setlocale(LC_CHARSET, NULL)` in its design:

> Specifies the name of the coded character set for which the **charmap** file is defined. This value determines the value returned by the `nl_langinfo` subroutine. The `<code_set_name>` must be specified using any character from the portable character set, except for control and space characters.

Any name can be chosen that fits this description, and POSIX nails nothing down for portability or identification reasons. There is no canonical list, just whatever implementations happen to supply as their “charmap” definitions.

3.4 SDCC

The Small Device C Compiler (SDCC) has already begun some of this work. One of its principle contributors, Dr. Philip K. Krause, wrote papers addressing exactly this problem[2]. 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.

Nevertheless, SDCC's work is still important: 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.

3.5 iconv/ICU

The C functions presented below is motivated primarily by concepts found in a popular POSIX library, iconv[3]. We do not provide the full power of iconv here but we do mimic its interface to allow for a better definition of functions, as explained in Problem 5. The core of the functionality can be embodied in this parameterized function signature:

```
size_t XstoYs(const charX** input, size_t* input_bytes, const charY**
        output, size_t* output_bytes);
```

In `iconv`'s case, an additional first parameter describing the conversion (of type `iconv_t`). That is not needed for this proposal, because we are not making a generic conversion API. This proposal is focused on doing 2 things and doing them extremely well:

— Getting data from the current execution encoding (`char`) to a Unicode encoding (`unsigned char`/UTF-8, `char16_t`/UTF-16, `char32_t`/UTF-32), and the reverse.
— Getting data from the current wide execution encoding (`wchar_t`) to a Unicode encoding (`unsigned char`/UTF-8, `char16_t`/UTF-16, `char32_t`/UTF-32), and the reverse.

iconv can do the above conversions, but also supports a complete list of pairwise conversions between about 49 different encodings. It can also be extended at translation time by programming more functionality into its library. This proposal is focusing just in doing conversions to and from encodings that the implementation owns to/from Unicode. This results in the design found below.

4 Solution

Given the problems before, the prior art, the implementation experience, and the vendor experience, it is clear that we need something outside of `nl_langinfo`, lighter weight than all of `iconv`, and more resilient and encompassing than what the C Standard offers. Therefore, the solution to our problem of having a wide variety of implementation encodings is to expand the contract of `wchar_t` for an **entirely new set of functions** which avoid the problems and pitfalls of the old mechanism.

Notably, both of the multibyte string's function design and the wide character string's definition of a single character is broken in terms of existing practice today. The primary problem relies in the inability for both APIs in either direction to handle `N:M` encodings, rather than `N:1` or `1:M`. Therefore, these new functions focus on providing an interface to allow multi-code-unit conversions, in both directions.

To facilitate this, new headers – `<stdmchar.h>` – will be introduced. Each header will contain the "multi character" (`mc`) and "multi wide character" (`mwc`) conversion routines, respectively. 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 using the new multi (wide) character (`m[w]c`) prefixes and suffixes:

	mc	mwc	mcs	mwcs	c8	c16	c32	c8s	c16s	c32s
mc	—	✓			P ✓	P ✓	P ✓			
mwc	✓	—			P ✓	P ✓	P ✓			
mcs			—	✓				P ✓	P ✓	P ✓
mwcs			✓	—				P ✓	P ✓	P ✓
c8	P ✓	P ✓			—	✗	✗			
c16	P ✓	P ✓			✗	—	✗			
c32	P ✓	P ✓			✗	✗	—			
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 locale encodings and multibyte locale encodings for its string literals (compiler) and library functions (standard library). Therefore, the `mc` and `mwc` functions simply focus on providing a good interface for these encodings. The form of both the individual and string conversion functions are:

```c
size_t XntoYn(const charX** input, size_t* input_size, const charY**
        output, size_t* output_size);
size_t XnrtoYn(const charX** input, size_t* input_size, const charY**
        output, size_t* output_size, mcstate_t* state);
size_t XsntoYsn(const charX** input, size_t* input_size, const
        charY** output, size_t* output_size);
size_t XsnrtoYsn(const charX** input, size_t* input_size, const
        charY** output, size_t* output_size, mcstate_t* state);
```

The input and output sizes are expressed in terms of the # of `charX`s. They take the input/output sizes as pointers, and decrement the value by the amount of input/output consumed. Similarly, the input/output data pointers themselves are incremented by the amount of spaces consumed / written to. This only happens when an irreversible and successful conversion of input data can successfully and without error be written to the output. The `s` functions work on whole strings rather than just a single complete irreversible conversion, the `n` stands for taking a size value.

The error codes are as follows:

— `(size_t)-3` the input is correct but there is not enough output space
— `(size_t)-2` an incomplete input was found after exhausting the input
— `(size_t)-1` an encoding error occurred
— `(size_t) 0` the operation was successful

The behaviors are as follows:

— if `output` is `NULL`, then no output will be written. `*output_size` will be decremented the amount of characters that would have been written.
— if `output` is non-`NULL` and `output_size` is `NULL`, then enough space is assumed in the output buffer.
— for the restartable (`r`) functions, if `input` is `NULL`, then `state` is set to the initial conversion sequence and no other actions are performed; otherwise, `input` must not be `NULL`.

Finally, it would be prudent to prevent the class of `(size_t)-3` errors from showing up in your code if you know you have enough space. For the non-string (the functions lacking `s`) that perform a single conversion, a user can pre-allocate a suitably sized static buffer in automatic duration storage space. This will be facilitated by a group of integral constant expressions contained in macros, which would be;

— `STDC_MC_MAX`, which is the maximum output for a call to one of the X to multi character functions
— `STDC_MWC_MAX`, which is the maximum output for a call to one of the X to multi wide character functions
— `STDC_MC8_MAX`, which is the maximum output for a call to one of the X to UTF-8 character functions
— `STDC_MC16_MAX`, which is the maximum output for a call to one of the X to UTF-16 character functions
— `STDC_MC32_MAX`, which is the maximum output for a call to one of the X to UTF-32 character functions

these values are suitable for use as the size of an array, allowing a properly sized buffer to hold all of the output from the non-string functions. These limits apply **only** to the non-string functions, which perform a single unit of irreversible input consumption and output (or fail with one of the error codes and outputs nothing).

Here is the full list of proposed functions:

```
#include <stdmchar.h>

#define STDC_C8_MAX  16
#define STDC_C16_MAX 8
#define STDC_C32_MAX 4
#define STDC_MC_MAX  16
#define STDC_MWC_MAX 4

enum : size_t { // N2575 - otherwise, will just use const size_t
        declarations here
  MCHAR_OK                  = (size_t)0,
  MCHAR_ENCODING_ERROR      = (size_t)-1,
  MCHAR_INCOMPLETE_INPUT    = (size_t)-2,
  MCHAR_INSUFFICIENT_OUTPUT = (size_t)-3,
};

size_t mcntomwcn(const char** input, size_t* input_size, const
        wchar_t** output, size_t* output_size);
size_t mcnrtomwcn(const char** input, size_t* input_size, const
        wchar_t** output, size_t* output_size, mcstate_t* state);
size_t mcsntomwcsn(const char** input, size_t* input_size, const
        wchar_t** output, size_t* output_size);
size_t mcsnrtomwcsn(const char** input, size_t* input_size, const
        wchar_t** output, size_t* output_size, mcstate_t* state);

size_t mcntoc8n(const char** input, size_t* input_size, const
        unsigned char** output, size_t* output_size);
size_t mcnrtoc8n(const char** input, size_t* input_size, const
        unsigned char** output, size_t* output_size, mcstate_t*
        state);
size_t mcsntoc8sn(const char** input, size_t* input_size, const
        unsigned char** output, size_t* output_size);
size_t mcsnrtoc8sn(const char** input, size_t* input_size, const
        unsigned char** output, size_t* output_size, mcstate_t*
        state);

size_t mcntoc16n(const char** input, size_t* input_size, const
        char16_t** output, size_t* output_size);
size_t mcnrtoc16n(const char** input, size_t* input_size, const
        char16_t** output, size_t* output_size, mcstate_t* state);
```

```
size_t mcsntoc16sn(const char** input, size_t* input_size, const
        char16_t** output, size_t* output_size);
size_t mcsnrtoc16sn(const char** input, size_t* input_size, const
        char16_t** output, size_t* output_size, mcstate_t* state);

size_t mcntoc32n(const char** input, size_t* input_size, const
        char32_t** output, size_t* output_size);
size_t mcnrtoc32n(const char** input, size_t* input_size, const
        char32_t** output, size_t* output_size, mcstate_t* state);
size_t mcsntoc32sn(const char** input, size_t* input_size, const
        char32_t** output, size_t* output_size);
size_t mcsnrtoc32sn(const char** input, size_t* input_size, const
        char32_t** output, size_t* output_size, mcstate_t* state);

size_t mwcntomcn(const wchar_t** input, size_t* input_size, const
        char** output, size_t* output_size);
size_t mwcnrtomcn(const wchar_t** input, size_t* input_size, const
        char** output, size_t* output_size, mcstate_t* state);
size_t mwcsntomcsn(const wchar_t** input, size_t* input_size, const
        char** output, size_t* output_size);
size_t mwcsnrtomcsn(const wchar_t** input, size_t* input_size, const
        char** output, size_t* output_size, mcstate_t* state);

size_t mwcntoc8n(const wchar_t** input, size_t* input_size, const
        unsigned char** output, size_t* output_size);
size_t mwcnrtoc8n(const wchar_t** input, size_t* input_size, const
        unsigned char** output, size_t* output_size, mwcstate_t*
        state);
size_t mwcsntoc8sn(const wchar_t** input, size_t* input_size, const
        unsigned char** output, size_t* output_size);
size_t mwcsnrtoc8sn(const wchar_t** input, size_t* input_size, const
        unsigned char** output, size_t* output_size, mwcstate_t*
        state);

size_t mwcntoc16n(const wchar_t** input, size_t* input_size, const
        char16_t** output, size_t* output_size);
size_t mwcnrtoc16n(const wchar_t** input, size_t* input_size, const
        char16_t** output, size_t* output_size, mwcstate_t* state);
size_t mwcsntoc16sn(const wchar_t** input, size_t* input_size, const
        char16_t** output, size_t* output_size);
size_t mwcsnrtoc16sn(const wchar_t** input, size_t* input_size, const
        char16_t** output, size_t* output_size, mwcstate_t* state);

size_t mwcntoc32n(const wchar_t** input, size_t* input_size, const
        char32_t** output, size_t* output_size);
size_t mwcnrtoc32n(const wchar_t** input, size_t* input_size, const
        char32_t** output, size_t* output_size, mwcstate_t* state);
```

```
size_t mwcsntoc32sn(const wchar_t** input, size_t* input_size, const
        char32_t** output, size_t* output_size);
size_t mwcsnrtoc32sn(const wchar_t** input, size_t* input_size, const
        char32_t** output, size_t* output_size, mwcstate_t* state);
```

4.1 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"[4] – 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 aid in portability.

5 Conclusion

The ecosystem deserves ways to get to a statically-known encoding and not rely on implementation and locale-parameterized encodings. This allows developers a way to perform cross-platform text processing without needing to go through fantastic gymnastics to support different languages and platforms. An independent library implementation, *cuneicode*[5] [6], is available upon request to the author. A patch to major libraries will be worked on again.

6 Wording

There is no wording at the moment, because there is more implementation to do and more approval to gain!

7 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.

8 References

May the Tower of Babel’s curse be defeated.

---

1. Tom Honermann and Carlos O’Donnell. `mbrtowc` with Big5-HKSCS returns 2 instead of 1 when consuming the second byte of certain double byte characters. https://sourceware.org/bugzilla/show_bug.cgi?id=25744↩

2. 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.↩

3. Bruno Haible and Daiki Ueno. libiconv. August 2020. Published: https://savannah.gnu.org/git/?group=libiconv.↩

4. Robot Martinho Fernandes. p1041. February 2019. Published: https://wg21.link/p1041.↩

5. JeanHeyd Meneide. Catching ⬆️: Unicode for C++ in Greater Detail". November 2019. Published Meeting C++: https://www.youtube.com/watch?v=FQHofyOgQtM.↩

6. JeanHeyd Meneide. Deep C Diving - Fast and Scalable Text Interfaces at the Bottom. July 2020. Published C++ On Sea: https://youtu.be/X-FLGsa8LVc.↩