Charset Transcoding, Transformation, and Transliteration

Steve Downey

March 10, 2019

- Document number: P1439R1
- Date: March 10, 2019

- Author: Steve Downey <sdowney2@bloomberg.net>, Steve Downey <sdowney@gmail.com>

- Audience: SG16

Abstract

Even in a Unicode-only environment, transcoding, transforming, and transliterating text is important, and should be supported by the C++ standard library in a user extensible manner. This paper does not propose a solution, but outlines the characteristics of such a desired facility.

1 Transcoding, Transforming, and Transliteration

Even in a Unicode-only environment, transcoding, transforming, and transliterating text is important, and becomes even more important when dealing with other text encoding systems.

Transcoding Converting text without loss of fidelity between encoding systems, such as from UTF-8 to UTF-32, or from ASCII to a Latin-1 encoding. Transcodings shall be reversible.

Transforming Generalized conversion of text, case mapping, normalization, or script-to-script conversion.

Transliteration The mapping of one character set or script to another, such as from Greek to Latin, where the transformation may not be reversible. The source is approximated by one or several encoded characters.

1.1 Transcoding

It should be fairly clear that even if Unicode processing is done all in an implementation-defined encoding, communicating with the rest of the world will require supplying the expected encodings. Fortunately, conversion between the UTF encodings is straightforward, and can be easily be done code point by code point. There are common mistakes made, however, such as transcoding UTF-16 surrogate pairs into distinct UTF-8 encoded code points (often known as WTF-8).[1]

Character encodings other than Unicode are still in wide use. Native CJKV encodings are still commonly used for Asian languages, as many systems that are in place were standardized before Unicode was adopted. For example, there are also issues with mapping Japanese encodings where the same character exists in more than one location in the JIS encoding. We generally treat converting from Shift-JIS to a Unicode encoding as a lossless transcoding, even though the exact binary can not be roundtripped because there is no semantic loss of information. It is simply that the same character has multiple representations in the encoding.

Real-world systems, even ones that may only emit Unicode in a single encoding, are likely to have to deal with input in a variety of encodings.

1.2 Transformation

Unicode has standardized a variety of text transformation algorithms, such as case mappings and normalizations. They also specify various ”tailorings” for these algorithms in order to support different languages, cultures, and scripts.

ICU, the International Components for Unicode,[2] offers a generalized transformation mechanism, providing:

1. Uppercase, Lowercase, Titlecase, Full/Halfwidth conversions
2. Normalization
3. Hex and Character Name conversions
4. Script-to-Script conversion[3]

They provide a comprehensive and accurate mechanism for text-to-text conversions. An example from the ICU User Guide[4]:

```
myTrans = Transliterator::createInstance(
     ”any-NFD; [:nonspacing mark:] any-remove; any-NFC”,
     UTRANS_FORWARD,
```

[1] The WTF-8 encoding
[2] International Components for Unicode
[3] General Transforms
[4] ICU User Guide

```
    status);
myTrans.transliterate(myString);
```

This transliterates an ICU string in-place. There are other versions that may be more useful.

Searching GitHub shows in the neighborhood of 32K uses of `Transliterator::createInstance`

1.3 Transliteration

Transliteration is a subset of general transformations, but is a very common use case. Converting to commonly-available renderable characters comes up frequently.

A widely used implementation of transliteration is in the GNU iconv package, where appending "//TRANSLIT" to the requested to-encoding will be changed such that:

> when a character cannot be represented in the target character set, it can be approximated through one or several similarly looking characters.

```
sdowney@kit:~
$ echo abc ß € àbç | iconv -f UTF-8 -t ASCII//TRANSLIT
abc ss EUR abc
```

In this example, the German letter ß is transliterated to two ASCII 's'es, the Euro currency symbol is transliterated to the string 'EUR', and the letters àbç have their diacritics stripped and are converted to the letters 'abc'. A similar program, `uconv`, based on ICU, does not do the € translation, leaving the Euro currency symbol untouched.

```
sdowney@kit:~
$ echo abc ß € àbç | uconv -c -x Any-ASCII
abc ss € abc
```

The //TRANSLIT facility is exported in the character conversion APIs of many programming languages, such as R, perl, and PHP.

There are over 7 million hits on GitHub for iconv, and 320K hits for TRANSLIT and iconv.

Providing a migration path for users of "//TRANSLIT" would be a great benefit.

2 Private Character Sets and the Unicode Private Use Area

We standardized character sets, like the American Standard Code for Information Interchange (ASCII), in order to be able to communicate be-

tween systems. However, there is a long history of systems using their own encodings and symbols internally.

Figure 1: The IBM PC Character Set

As you can see in figure 1, there are glyphs rendered for code points that are non-printing in ASCII. The high characters include line drawing and accented characters. The original PC was influential enough that the character set became well-known and effectively standardized.

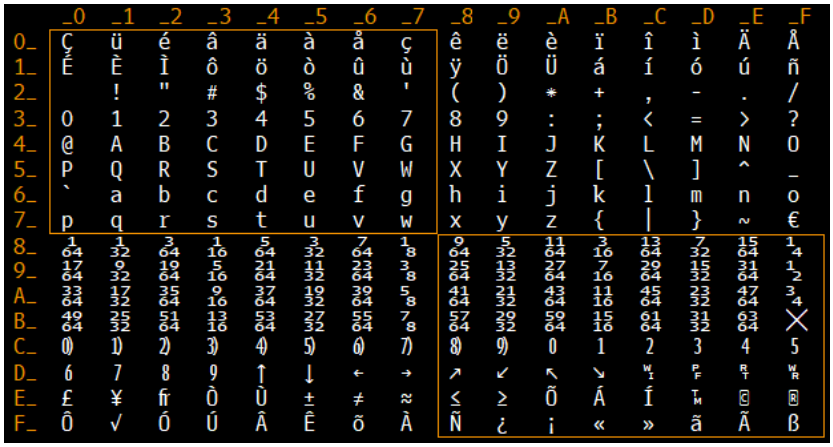

Figure 2: Bloomberg Terminal Font 0

Figure 2 contains the current form of the font originally used by Bloomberg's hardware terminal. It was designed for internationalized finance. It includes the accented characters needed for Western European languages, fractions and other special symbols used in finance, and a selection of half-width characters to minimize use of screen real estate. The only non-printing character is the space character. Even character 0x00 is in use,

for {LATIN CAPITAL LETTER C WITH CEDILLA}. Originally, null terminated strings were not used. Instead, character arrays and a size were the internal character format. Of course, this has caused issues over the years. However, it meant that almost all European languages could be used natively by the terminal.

Today, this character encoding is used only for legacy data. Data is translated to Unicode, usually UTF-8, as soon as it is accepted. It is maintained that way throughout the system, as long as it was not originally in a UTF encoding. Legacy data, where the encoding is known, are usually translated to modern encodings at the first opportunity. It is occasionally a challenge to know which encoding is being used. As the company expanded beyond the Americas and Europe, additional local encodings were added, but data was not always tagged with the proper encoding, leading to complications.

There is still a necessity to maintain the characters used for financial purposes. In particular, this is necessary to concisely and accurately communicate financial fractions. Unicode has standard fractions to 1/8th precision, ⅛ ¼ ⅜ ½ ⅝ ¾ ⅞, but in finance, fractions down to 1/64th are routinely quoted. Internally, Bloomberg uses code points in the Unicode Private Use Area to represent these fractions, as well as the rest of its legacy character sets. This allows for convenient mappings between scripts, treating the private code page as a distinct Unicode script. This is the intended use of the Private Use Area, to handle ranges of code points that will not be assigned meaning by the Unicode Consortium. [5]

Bloomberg generally transliterates private characters when externalizing data. For example, in sending out email:

[5] Private Use Area

```
Ç ü é â ä à å ç ê ë è ï î ì Ä Å
É È Ì ô ö ò û ù ÿ Ö Ü á í ó ú ñ
  ! ” # $ % & ’ ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
‘ a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~ €
1/64 1/32 3/64 1/16 5/64 3/32 7/64 ⅛ 9/64 5/32
11/64 3/16 13/64 7/32 15/64 ¼
17/64 9/32 19/64 5/16 21/64 11/32 23/64 ⅜ 25/64
13/32 27/64 7/16 29/64 15/32 31/64 ½
33/64 17/32 35/64 9/16 37/64 19/32 39/64 ⅝ 41/64
21/32 43/64 11/16 45/64 23/32 47/64 ¾
49/64 25/32 51/64 13/16 53/64 27/32 55/64 ⅞ 57/64
29/32 59/64 15/16 61/64 31/32 63/64 ×
0) 1) 2) 3) 4) 5) 6) 7) 8) 9) 0 1 2 3 4 5
6 7 8 9 ↑ ↓ ← → ↗ ↙ ↖ ↘ (WI) (PF) (RT) (WR)
£ ¥ ₣ Ò Ù ± ≠ ≈ ≤ ≥ Õ Á Í ™ © ®
Ô ✓ Ó Ú Â Ê õ À Ñ ¿ ¡ « » ã Ã ß
```

3 Request for Proposal

Transliteration is in wide use. However, none of the existing facilities fit well with modern C++ or the current proposals for standardizing Unicode text facilities. Providing extensible transliteration facilities will enable a transition to the new libraries. Transcoding is also a requirement for dealing with existing fixed APIs, such as OS HMI facilities.

3.1 Issues with existing facilities

- iconv is `char*` based, and has an impedance mismatch with modern Ranges, as well as with iterators
- iconv relies on an error code return and checking `errno` as a callback mechanism
- ’Streaming’ facilities generally involve block operations on character arrays and handling underflow

- ICU relies on inheritance for the types that can be transformed
- Interfaces that specify types as character string are not at all type safe on the operations being requested

Some initial experiments using the new Ranges facilities suggest that 'streaming' can be externalized without significant cost via iterators over a `view::join` on an underlying stream of blocks. This would certainly expand the reach of an API, while simplifying the interior implementation. Transcoding and transliteration APIs should generally not operate in place, and should accept Range views as sources and output ranges as sinks for their operations.

3.2 Desired Features

3.2.1 Ranges

It should be possible to apply any of the transcoding or transliteration algorithms on any range that exposes code units or code points. General transformation algorithms may require code points. Combining algorithms that transform charset encoded code units to code points and feed that view into an algorithm for further transformation should be both natural and efficient.

3.2.2 Open extension in build time safe way

The set of character sets and scripts is not fixed and must be developer extensible. This extension should not require initialization in main or dynamic loading of modules, as both lead to potentially disastrous runtime errors. It is entirely reasonable to require compile time definitions of character sets or scripts and require that library facilities be linked in if custom encodings are used. Using strings to indicate encoding rather than strongly typed entities are problematic, and since the universe of character sets is not fixed, standard library enums are not a good solution either. NTTPs are possible areas of research, as are invocable objects.

3.2.3 Exception neutral error handling

Unfortunately, misencodings of all kinds are not actually exceptional in text processing, particularly at the input perimeter. APIs that treat the various issues as normal would be preferred. The API should provide mechanisms for letting the library handle issues without intervention, such as by indicating substitution characters for un-decodable input, while also providing standardized callback mechanisms to allow more general intervention. The API should certainly avoid the current pattern of returning -1, checking the C errno which indicates the issue, and having the caller fix and restart the conversion.