ztd.text

Release 0.0.0

ThePhD & Shepherd's Oasis, LLC

Aug 13, 2021

CONTENTS:

The premiere library for handling text in different encoding forms and reducing transcoding bugs in your C++ software.

CHAPTER

ONE

WHO IS THIS LIBRARY FOR?

If:

- you want to convert from one Unicode encoding to another Unicode encoding;
- you want a no-overhead way to track and keep data in a specific encoding (Unicode-based or not);
- you want a no-memory-overhead way to archive;
- you want to prevent data in the wrong encoding from infiltrating your application and causing Mojibake;
- you want to work with higher-level primitives (code points, graphames) when iterating text that do not break your text apart;
- you want safe defaults for working with text;

then `ztd.text` is for you!

1.1 Getting Started (In Progress)

> **Warning:** This isn't finished yet! Come check back by the next major or minor version update.

1.2 Quick 'n' Dirty Tutorial (In Progress)

> **Warning:** This isn't finished yet! Come check back by the next major or minor version update.

1.3 Users in the Wild

None have come and told us about their usage, yet!

If you use the library to any success, please do not hesitate to reach out to opensource@soasis.org!

1.4 Glossary of Terms & Definitions

Occasionally, we may need to use precise language to describe what we want. This contains a list of definitions that can be linked to from the documentation to help describe key concepts that are useful for the explication of the concepts and ideas found in this documentation.

character This word carries with it 2 meanings, thanks to C-style languages and their predecessors. Sometimes, `char`s, `wchar_t`s, `char8_t`s, and similar are called "narrow character"s, "wide character"s, "UTF-8 characters" and similar. This is the result of a poor legacy in software and hardware nomenclature. These are not character types, but rather types that _may_ represent the abstract notion of a character but frequently, and often, do not. After all, you wouldn't be here reading this if it did and non-English wasn't busted in your application, now would you?

The other definition is just an abstract unit of information in human languages and writing. The closest approximation that Unicode has for the human language/writing character is a *Grapheme Cluster*.

code point A single unit of decoded information. Most typically associated with *unicode code points*, but they can be other things such as *unicode scalar values* or even a 13-bit value.

Note that a single code point does not imply a *"character"*, as that is a complex entity in human language and writing that cannot be mapped easily to a single unit of decoded information.

code unit A single unit of encoded information. This is typically, 8-, 16-, or 32-bit entites arranged in some sequential fashion that, when read or treated in a certain manner, end up composing higher-level units which make up readable text. Much of the world's most useful encodings that encode text use multiple code units in sequence to give a specific meaning to something, which makes most encodings variable length encodings.

decode Converting from a stream of input, typically code units, to a stream of output, typically code points. The output is generally in a form that is more widely consummable or easier to process than when it started. Frequently, this library expects and works with the goal that any decoding process is producing *unicode code points* or *unicode scalar values* from some set of *code units*.

encode Converting from a stream of input, typically code points, to a stream of output, typically code units. The output may be less suitable for general interchange or consumption, or is in a specific interchange format for the interoperation. Frequently, this library expects and works with the goal that any decoding process is producing *unicode code points* or *unicode scalar values* from some set of *code units*.

encoding A set of functionality that includes an encode process or a decode process (or both). The encode process takes in a stream of code points and puts out a stream of code units. The decode process takes in a stream of code units and puts out a stream of code points. In a concrete sense, there are a number of additional operations an encoding needs: see the *Lucky 7 design concept*.

execution encoding The locale-based encoding related to "multibyte characters" (C and C++ magic words) processed during program evaluation/execution. It is directly related to the `std::set_locale(LC_CTYPE, ...)` calls. Note that this is different from *literal encoding*, which is the encoding of string literals. The two may not be (and many times, are not) the same.

grapheme cluster The closest the Unicode Standard gets to recognizing a *human-readable and writable character*, grapheme cluster's are arbitrarily sized bundles of *unicode code points* that compose of a single concept that might match what a *"character"* is in any given human language.

injective An operation which can map all input information to an output. This is used for this library, particularly, to determine whether an operation is lossy (loses information) or not. For example, UTF-8 to UTF-32 is an injective operation because the values in a UTF-8 encoding are preserved in a UTF-32 encoding. UTF-16 to GB18030 is also an injective operation. But, converting something like Latin-1 to ASCII is a lossy operation, or UTF-8 to SHIFT-JIS.

literal encoding The encoding of string literals ("") during constant evaluation. This is usually controlled by command line arguments (MSVC and GCC) or fixed during compilation (Clang as UTF-8, though that may change). Typically defaults to the system's "locale" setting.

mojibake (Japanese: Pronunciation: [modibake] "unintelligible sequence of characters".) From Japanese (moji), meaning "character" and (bake), meaning change, is an occurence of incorrect unreadable characters displayed when computer software fails to render text correctly to its associated character encoding.

transcode Converting from one form of encoded information to another form of encoded information. In the context of this library, it means going from an input in one *encoding*'s code units to an output of another encoding's code units. Typically, this is done by invoking the *decode* of the original encoding to reach a common interchange format (such as *unicode code points*) before taking that intermediate output and piping it through the *encode* step of the other encoding. Different transcode operations may not need to go through a common interchange, and may transcode "directly", as a way to improve space utilization, time spent, or both.

unicode code point A single unit of decoded information for Unicode. It represents the smallest, non-encoded, and indivisible piece of information that can be used to talk about higher level algorithms, properties, and more from the Unicode Standard.

A unicode code point has been reserved to take at most 21 bits of space to identify itself.

A single unicode code point is NOT equivalent to a *character*, and multiple of them can be put together or taken apart and still have their sequence form a *"character"*. For a more holistic, human-like interpretation of code points or other data, see *grapheme clusters*.

unicode scalar value A single unit of decoded information for Unicode. It's definition is identical to that of *unicode code points*, with the additional constraint that every unicode scalar value may not be a "Surrogate Value". Surrogate values are non-characters used exclusively for the purpose of encoding and decoding specific sequences of code units, and therefore carry no useful meaning in general interchange. They may appear in text streams in certain encodings: see *Wobbly Transformation Format-8 (WTF-8)* for an example.

wide execution encoding The locale-based encoding related to "wide characters" (C and C++ magic words) processing during program evaluation/execution. It is directly related to the `std::set_locale(LC_CTYPE, ...)` calls. Note that this is different from the *wide literal encoding*, which is the encoding of wide string literals. The two may not be (and many times, are not) the same. Nominally, wide string literals are usually not like this, but there are a handful of compilers were they use neither UTF-16 or UTF-32 as the wide execution encoding, and instead use, for example, EUC-TW.

wide literal encoding The encoding of wide string literals (`L""`) during constant evaluation. This is usually controlled by command line arguments (GCC) or fixed during compilation (Clang as UTF-32, though that may change). Typically defaults to the system's "locale" setting.

1.5 Design Goals and Philosophy

The goal of this library are to

- enable people to write new code that can properly handle encoded information, specifically text;
- offer them effective means to convert that information in various ways;
- impose no run-time overhead compared to writing the code by hand; and
- statically track encodings, where possible, to make lossless or bad conversions a compile time error rather than a runtime problem;

These four goals inform the design of the library to its deepest levels and helps us go through the following important tenents:

1.5.1 First Principles - "Lucky 7" and a Liberation-First Design

One of the core premises of this library is that any text in one encoding can be converted to another, without having to know anything about external encodings. This is how the library achieves infinite extensibility! We start by noting that almost any encoding conversion can be done so long as there is an intermediary that exists between the source and the destination. For encoded text, this is the line between code units (`code_unit` for code) and code points (`code_point` for code).

Code units are single elements of a linear sequence of encoded information. That could be a sequence of bytes, a sequence of 16-bit numbers, and more. A sequence of code units is typically specific to the encoding it has and is generally impossible to reason about in a general or generic sense.

Code points are single elements of a linear sequence of information that have been decoded. They are far more accessible because they are generally an agreed upon interchange point that most others can access and reason about.

We leverage that, for text, **Unicode Code Points** are an agreed upon interchange format, giving rise to this general framework for encoding and decoding text:

The way to tap into this concept is to create an object that models an encoding concept, which is commonly referred to as the "Lucky 7" concept. The concept leverages a technique that has been used at least since the early days of Bruno Haibile's and Daiko Ueno's `iconv` library, but formalizes it for interacting between 2 encodings.

We call this concept the **Lucky 7**.

Lucky 7

Lucky 7 is a conceptual idea a single encoding object is all you need to write to fulfill your end of the encoding bargain. It is called the Lucky 7 because only 7 things are required from you, as the author of the encoding object, to get started:

- 3 type definitions (`code_point`, `code_unit`, `state`)
- 2 static member variables (`max_code_points`, `max_code_units`)
- 2 functions (`encode_one`, `decode_one`)

```cpp
#include <cstddef>
#include <span>

struct empty_struct {};

struct utf_ebcdic {
        // (1)
        using code_unit  = char;
        // (2)
        using code_point = char32_t;
        // (3)
        using state      = empty_struct;

        // (4)
        static constexpr inline std::size_t max_code_points = 1;
        // (5)
        static constexpr inline std::size_t max_code_units  = 6;

        // (6)
        ue_encode_result encode_one(
                ztd::span<const code_point> input,
                ztd::span<code_unit> output,
```

(continues on next page)

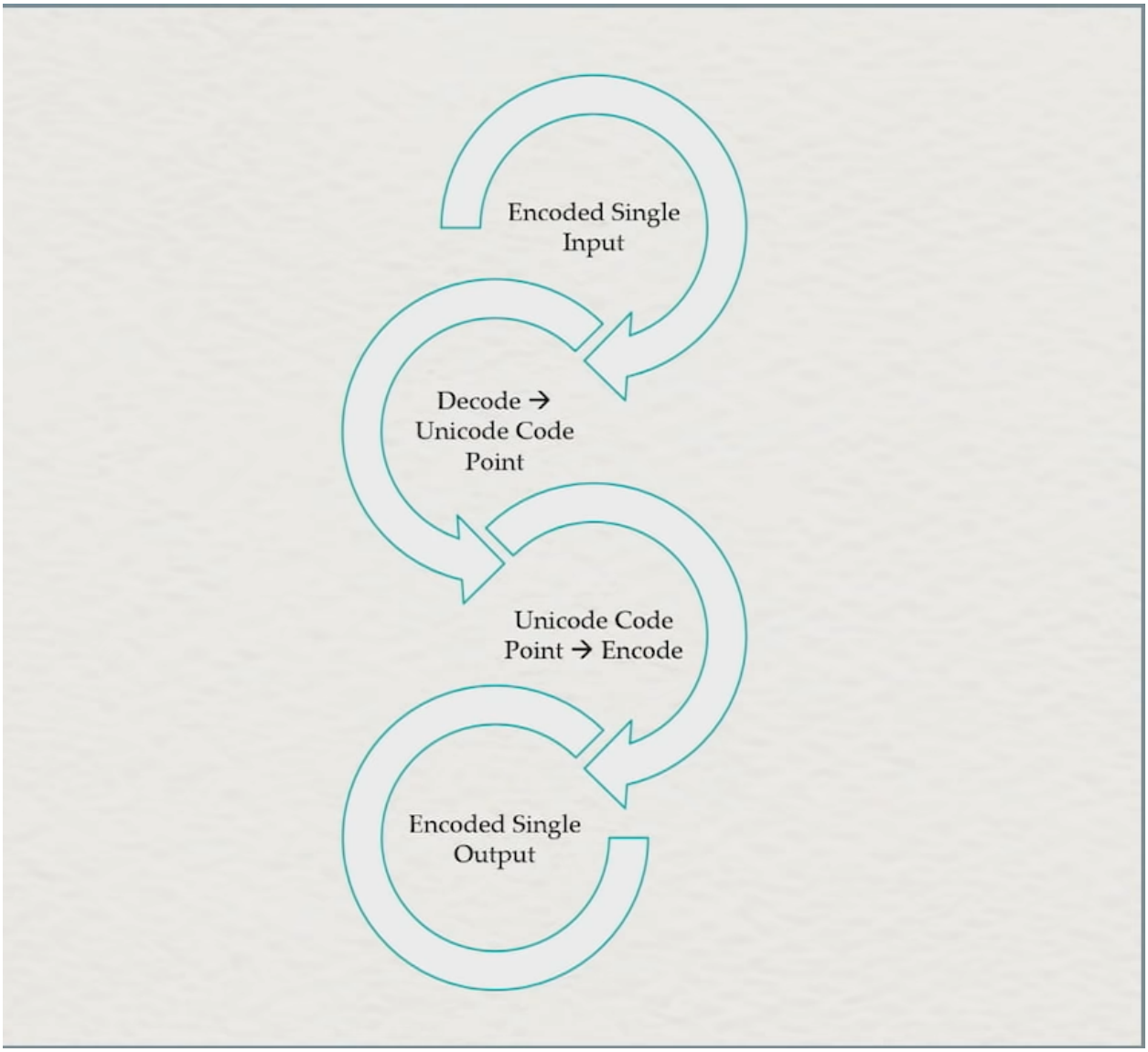


Fig. 1: The generic pathway from one encoding to another for most (all?) text Encodings.

(continued from previous page)

```
                state& current,
                ue_encode_error_handler error_handler
        );

        // (7)
        ue_decode_result decode_one(
                ztd::span<const code_unit> input,
                ztd::span<code_point> output,
                state& current,
                ue_decode_error_handler error_handler
        );
};
```

There are some supporting structures here that we will explain one by one, but this is the anatomy of a simple encoding object that you and others can define to do this job. This anatomy explicitly enables some basic work:

- encoding a single indivisible unit of work from *code points* to *code units*
- decoding a single indivisible unit of work from *code units* to *code points*
- transcoding a single indivisible unit of work from the source encoding's *code units* to the destination encoding's *code code units*, if they share a common *code point type*.

From these 3 operations above, everything else on this library can be built.

Breaking it Down

The first three typedefs are what let internal and externel machinery know what kind of values you expect out of the ranges that go into the `decode_one` and `encode_one` function calls:

- `code_unit` - the input for decoding (`decode_one`) operations and the output for encode operations.
- `code_point` - the input for encode operations and the output for decoding (`decode_one`) operations.

`char` is the code unit type that the ranges work with for incoming and outgoing encoded data. `char32_t` is the code point type that the ranges use for incoming and outgoing decoded data. Given those, that gives us the ability to define the result types we will be working with.

Result Types

Result types are specific structs in the library that mark encode and decode operations. They can be used by composing with the templated type *ztd::text::decode_result* and *ztd::text::encode_result*.

```
#include <ztd/text/encode_result.hpp>
#include <ztd/text/decode_result.hpp>

using ue_decode_result = ztd::text::decode_result<
        ztd::span<const char>,
        ztd::span<char32_t>,
        empty_struct
>;

using ue_encode_result = ztd::text::encode_result<
        ztd::span<const char32_t>,
```

(continues on next page)

(continued from previous page)

```
        ztd::span<char>,
        empty_struct
>;
```

These result structures are returned from the lowest level `encode` and `decode` operations. They contain:

- An `input` member, which is the range that is passed into the `decode_one` and `encode_one` functions;
- An `output` member;
- A `state` member, which is a reference to the `state` that was passed in to the `decode_one` or `encode_one` functions;
- An `error_code` member, which is an enumeration value from *ztd::text::encoding_error*; and
- An `handled_errors` member, which is an unsigned integral (`std::size_t`) value that says whether or not the given `error_handler` was invoked and how many times
- An `errors_were_handled()` member function, which returns a boolean value indicating whether `handled_errors` is greater than 0.

These variables can be used to query what exactly happened during the operation (`error_code` and `handled_errors`), inspect any state passed into encodings (not used for an encoding such as `utf_ebcdic`), and how much input and output has been read/what is left (by checking the `input` and `output` ranges whose `.begin()` value has been incremented compared to the input values). Understanding the result types now, we move to the error handler:

Error Handlers

The only other thing we need is the error handler, now. Generally, this is a template argument, but for the sake of illustration we use a concrete type here:

```
#include <functional>

using ue_decode_error_handler = std::function<
        ue_decode_result(
                const utf_ebcdic&,
                ue_decode_result,
                ztd::span<char>
        )
>;

using ue_encode_error_handler = std::function<
        ue_encode_result(
                const utf_ebcdic&,
                ue_encode_result,
                ztd::span<char32_t>
        )
>;
```

The error handlers use a result-in, result-out design. The parameters given are:

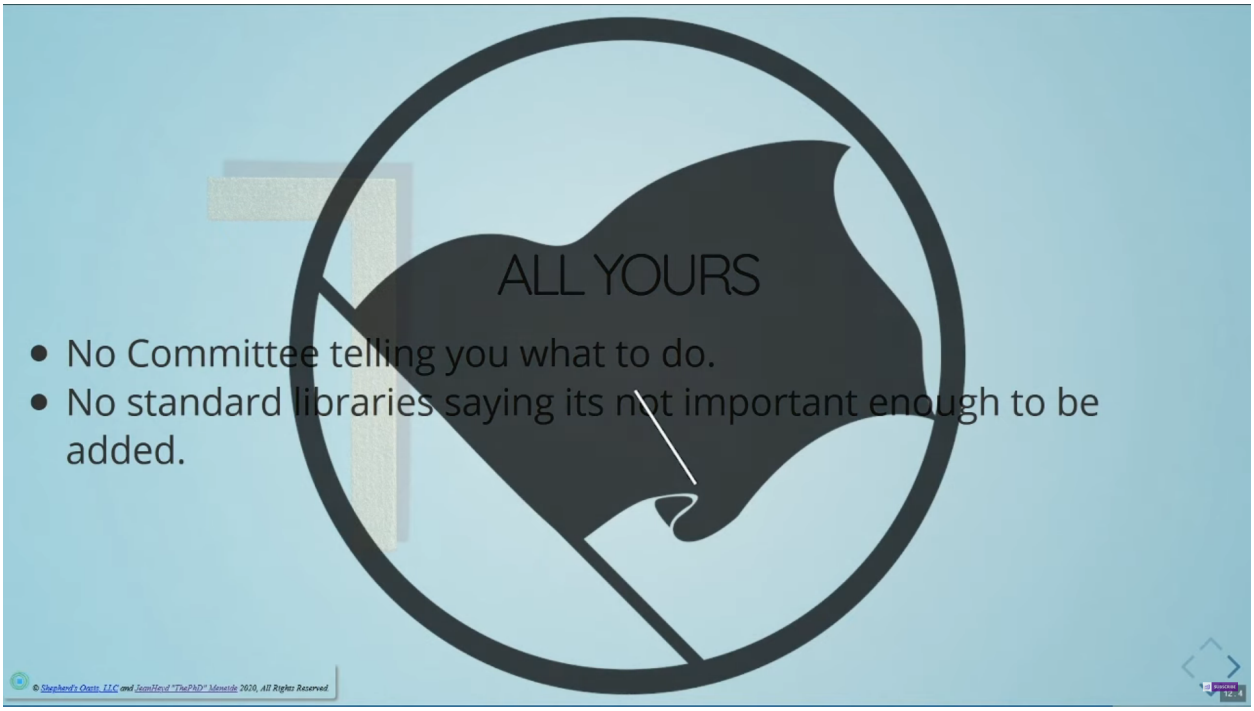
0. The encoding which triggered the error. This allows you to access any information about the encoding object type or any values stored on the encoding object itself.
1. The result object. This object has the `error_code` member set to what went wrong (see *ztd::text::encoding_error*), and any other changes made to the `input` or `output` during the operation.

2. A contiguous range (`ztd::span`) of `code_units` or `code_points` that were already read by the algorithm. This is useful for when the `input` range uses input iterators, which sometimes cannot be "rolled back" after something is read (e.g., consider std::istream_iterator).

It returns the same type as the result object. Within this function, anyone can perform any modifications they like to the type, before returning it. This is an incredibly useful behavior that comes in handy for defining custom error handling behaviors, as shown in *the Error Handling Design section*. For example, this allows us to do things like insert **REPLACEMENT_CHARACTER \uFFFD** () into a encoding through the *ztd::text::replacement_handler_t* or enable speedy encoding for pre-validated text using *ztd::text::assume_valid_handler*. When writing your `encode_one` or `decode_one` function, it is your responsibility to invoke the error handler (or not, depending on the value of *ztd::text::is_ignorable_error_handler*).

Liberation Achieved

If you achieve all these things, then we can guarantee that you can implement all of the desired functionality of an encoding library. This is the core design that underpins this whole library, and how it frees both Library Developers from needing to manically provide every possible encoding to end-users, and end-users from having to beg library developers to add support for a particular encoding.



There is more depth one can add to an encoding object, but this is the base, required set of things to know and handle when it comes to working with ztd.text. You can build quite a complex set of features from this functionality, and we encourage you to keep reading through more of the design documentation to get an understanding for how this works!

1.5.2 Lost Information

One of the biggest problems facing text processing in programming languages today is the loss of information as its carried through any given system. In C and C++, this comes in the form of all strings - especially multibyte strings - being given the same type. For example:

```
void read_name(const char* name) {
        // (1)
}
```

As the maintainer of code inside of the function `read_name`, what is the encoding of "name" at `(1)`? What is its normalization form? What system did it originate from? The function written in C++ form offers very little benefit either:

```
void read_name(std::string_view name) {
        // (1)
}
```

Even here, we've only made marginal improvements. We know the string is stored in some heap by the default allocator, we have the size of the string, but that only tells us how many `char` units are stored, not how many conceptual, human-readable *characters* there are or any other pertinent information. Is this information encoded? Is it UTF-8? Maybe it's EBCDIC Code Page 833. Maybe it's UTF-7-IMAP. You don't know, and by the time you start inspecting or poking at the individual `char` *code units*, who knows what can happen? To make matters worse, even C++ and its Standard Library have poor support for encoding/decoding, let alone Unicode in general. These problems have been explained in quite a lot of detail up to this point, but the pitfalls are many:

> ... Where are potential problems?
>
> All over the place? Let's see...
>
> —R. Martinho Fernandes, last edited April 20th, 2018

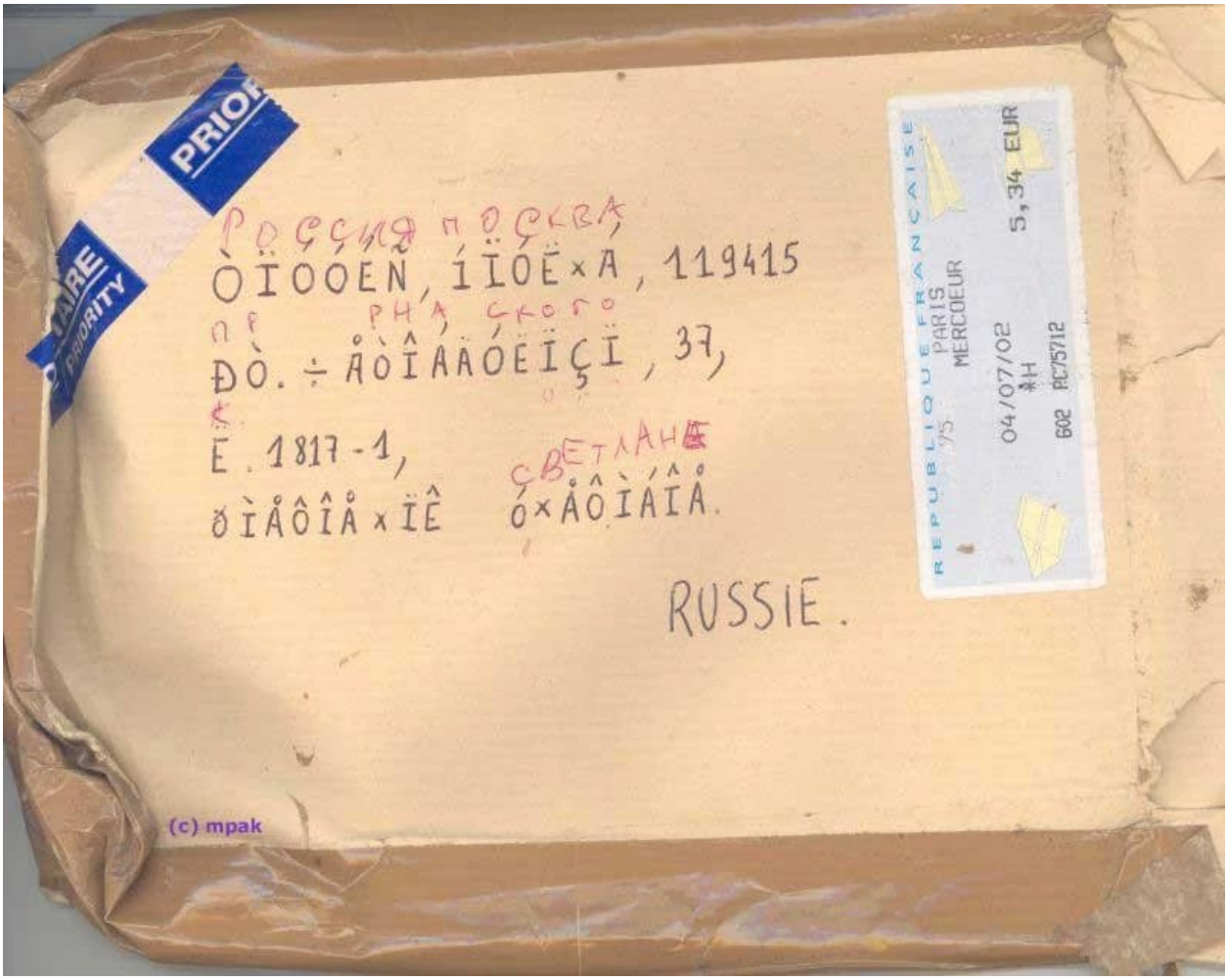
Some proponents say that if we just change everything to mean "UTF-8" (*const char**, *std::string*, and more), then we can just assume UTF-8 throughout the entire application and only accept UTF-8 and that will end all our encoding problems. Typically, these people read UTF-8 Everywhere and then just go all-in on the philosophy, all the time.

"UTF-8 Everywhere!!"

There are many in the programming space that believe that just switching everything to UTF-8 everywhere will solve the problem. This is, unfortunately, greatly inadequate as a solution. For those who actually read the entire UTF-8 Everywhere manifesto in its fullness, they will come across this FAQ entry:

> **Q: Why not just let any programmer use their favorite encoding internally, as long as they knows how to use it?**
>
> A: We have nothing against correct usage of any encoding. However, it becomes a problem when the same type, such as std::string, means different things in different contexts. While it is 'ANSI codepage' for some, for others, it means 'this code is broken and does not support non-English text'. In our programs, it means Unicode-aware UTF-8 string. This diversity is a source of many bugs and much misery. ...
>
> —FAQ Entry #6

The core problem with the "`std::string` is always UTF-8" decision (even when they are as big as Google, Apple, Facebook, or Microsoft and own everything from the data center to the browser you work with) is that they live on a planet with other people who do not share the same sweeping generalizations about their application environments. Nor have they invoked the ability to, magically, rewrite everyone's code or the data that's been put out by these programs in the last 50 or 60 years. This results in a gratuitous amount of replacement characters or *Mojibake* when things do not encode or decode properly:



There is a distinct problem that human beings are so used to computers failing them with encoding that they know how to recognize the mistranslated text:

> We get so good at it that we can even recognize the bad text . There's a wiki for it too… … It used to be a lot worse. UTF-8 definitely helps a whole lot.
>
> —Elias Daler

So, what do we do from here?

Fighting Code Rot

We need ways to fight bit rot and issues of function invariants – like expected encoding on string objects – from infesting code. While we can't rewrite every function declaration or wrap every function declaration, one of the core mechanisms this library provides is a way of tracking and tagging this kind of invariant information, particularly at compile time.

We know we can't solve interchange on a global level (e.g., demanding everyone use UTF-8) because, at some point, there is always going to be some small holdout of legacy data that has not yet been fixed or ported. The start of solving this is by having views and containers that keep encoding information with them after they are first constructed. This makes it possible to not "lose" that information as it flows through your program:

```
using utf8_view = ztd::text::decode_view<ztd::text::utf8>;

```

(continues on next page)

(continued from previous page)

```
void read_name(utf8_view name) {
        // (1)
}
```

Now, we have an *explicit decoding view* into a sequence of UTF-8 code units, that produces `unicode_code_point`s that we can inspect and work with. This is much better, as it uses C++'s strong typing mechanisms to give us a useful view. This means that not only does the person outside of the `read_name` function understand that the function expects some UTF-8 encoded text, but the person inside the function knows that they are working with UTF-8 encoded text. This solves both ends of the user and maintainer divide.

Of course, sometimes this is not always possible. ABI stability mandates some functions can't have their signatures change. Other times, you can't modify the signature of functions you don't own. This is still helpful in this case, as you can, at the nearest available point inside the function or outside of it, apply these transformations:

```
void read_name(const char* untagged_name) {
        using utf8_view = ztd::text::decode_view<
                ztd::text::basic_utf8<char>, // use "char" as the code unit type
                std::string_view // explicitly use this view type
        >;
        // constructs a std::string_view and
        // stores it in the proper place
        utf8_view name(untagged_name);
        // use it...
}
```

Because the range and container types are templated on not only encoding, but the underlying storage type, you can wrap up both parameter and return values. You can also access the underlying `std::string_view` using `.base()`, so it remains easy to interop and work with pre-existing systems using newer, more explicit types. Other ranges become possible, including, say, the *__gnu_cxx::rope <https://gcc.gnu.org/onlinedocs/gcc-10.2.0/libstdc++/api/a08538.html>* class that is part of the GCC Extensions Library. It genuinely doesn't matter what you pick: we will wrap it up and present the proper interface to you. This also follows UTF-8 Everywhere's requirements for what it would want out of a C++ Library that does text Correctly™:

> If you design a library that accepts strings, the simple, standard and lightweight std::string would do just fine. On the contrary, it would be a mistake to reinvent a new string class and force everyone through your peculiar interface. Of course, if one needs more than just passing strings around, he should then use appropriate text processing tools. However, such tools are better to be independent of the storage class used, in the spirit of the container/algorithm separation in the STL.
>
> —UTF-8 Everywhere, FAQ Entry #19

Rather than create new `std::string` or `std::string_view` types, we simply wrap existing storage interfaces and provide specific views or operations on those things. This alleviates the burden of having to reinvent things that already work fine for byte-oriented interfaces, and helps programmers control (and prevent) bugs. They also get to communicate their intent in their APIs if they so desire ("This API takes a `std::string_view`, but with the expectation that it's going to be decoded as `utf8`"). The wrapped type will always be available by calling `.base()`, which means a developer can drop down to the level they think is appropriate when they want it (with the explicit acknowledgement they're going to be ruining things).

1.5.3 Error Handling

Text is notorious for being a constant and consistent malformed source of input. From intermediate services mangling encodings and producing *Mojibake* to bungled normalization and bad programs not understanding even the slightest hint of code beyond ASCII, there is a lot of text data that is strictly bad for any program to consume.

When interfacing with range types such as *ztd::text::decode_view*, functions like *ztd::text::transcode*, and individual `.encode_one` or `.decode_one` calls on encoding objects like *ztd::text::utf8*, you can:

- give an error handler type as a template parameter and as part of the constructor; or,
- pass it in as a normal argument to the function to be used.

They can change the conversion and other operations happen works. Consider, for example, this piece of code which translates from Korean UTF-8 to ASCII:

```
#include <ztd/text/transcode.hpp>

#include <iostream>

int main(int, char*[]) {
        // (1)
        std::string my_ascii_string = ztd::text::transcode(
             // input
             u8"",
             // from this encoding
             ztd::text::utf8 {},
             // to this encoding
             ztd::text::ascii {});

        std::cout << my_ascii_string << std::endl;

        return 0;
}
```

Clearly, the Korean characters present in the UTF-8 string just cannot fit in a strict, 7-bit ASCII encoding. What, then, becomes the printed output from `std::cout` at `// (2)`? The answer is two ASCII question marks, `??`. The *ztd::text::replacement_handler_t* object passed in at `// (1)` substitutes replacement characters (zero or more) into the output for any failed operation. There are multiple kinds of error handlers with varying behaviors:

- *replacement_handler_t*, which inserts a substitution character specified by either the encoding object or some form using the default replacement character `"U+FFFD"`;
- *pass_handler*, which simply returns the error result as it and, if there is an error, halts higher-level operations from proceeding forward;
- *default_handler*, which is just a name for the `replacement_handler_t` or `throw_handler` or some other type based on compile time configuration of the library;
- *throw_handler*, for throwing an exception on any failed operation;
- *incomplete_handler*, for throwing an exception on any failed encode/decode operation; and,
- *assume_valid_handler*, which triggers no checking for many error conditions and can leads to Undefined Behavior if used on malformed input.

Warning: For the love of what little remains holy, PLEASE don't use `ztd::text::assume_valid_handler` unless you REALLY know you need it. It is a surefire way to open up vulnerabilities in your text processing

algorithm. Not a single line of code using this type should pass code review if there is even the slightest thought that this will be used on any input that is not PERFECTLY under the DIRECT, PERSONAL control of the authors, auditors, and maintainers of the code.

These are all the error handlers that you have at your disposal, but they are just pre-provided types you can instantiate yourself. Nothing stops you from making your own error handling type! In order to do that, however, you need to understand what an error handler is composed of, and what it's got inside of itself.

Error Handler Anatomy

An error handler is just a function (or an object with a function call operator) that takes 3 parameters and returns 1 result:

- takes the encoding that will call it when something goes wrong;
- takes the result object you expect to be working with (specifically, *ztd::text::encode_result* and *ztd::text::decode_result*), which contains the current state of affairs from the encoding operation;
- takes a contiguous range representing any input values that may have been read but will not be used; and,
- returns the same result type with any modifications (or not!) you'd like to make.

They are classes with a function call operator and utilizes a few templates. Here's the skeleton for one:

```
#include <ztd/text.hpp>

struct my_error_handler {
        // Helper definitions
        template <typename Encoding>
        using code_point_span
             = ztd::span<const ztd::text::code_point_t<Encoding>>;
        template <typename Encoding>
        using code_unit_span
             = ztd::span<const ztd::text::code_unit_t<Encoding>>;

        // Function call operator that returns a "deduced" (auto) type
        // Specifically, this one is called for encode failures
        template <typename Encoding, typename Input, typename Output,
             typename State>
        auto operator()(
             // First Parameter
             const Encoding& encoding,
             // Second Parameter, encode-specific
             ztd::text::encode_result<Input, Output, State> result,
             // Third Parameter
             code_point_span<Encoding> input_progress,
             // Fourth Parameter
             code_unit_span<Encoding> output_progress) const noexcept {
                // ... implementation here!
                (void)encoding;
                (void)input_progress;
                (void)output_progress;
                return result;
        }
```

(continues on next page)

(continued from previous page)

```
        // Function call operator that returns a "deduced" (auto) type
        // Specifically, this one is called for decode failures
        template <typename Encoding, typename Input, typename Output,
             typename State>
        auto operator()(
             // First Parameter
             const Encoding& encoding,
             // Second Parameter, decode-specific
             ztd::text::decode_result<Input, Output, State> result,
             // Third Parameter
             code_unit_span<Encoding> input_progress,
             // Fourth Parameter
             code_point_span<Encoding> output_progress) const noexcept {
                // ... implementation here!
                (void)encoding;
                (void)input_progress;
                (void)output_progress;
                return result;
        }
};

int main(int, char* argv[]) {

        // convert from execution encoding to utf8 encoding,
        // using our new handler
        std::string utf8_string = ztd::text::transcode(
             std::string_view(argv[0]), ztd::text::execution,
             ztd::text::basic_utf8<char> {}, my_error_handler {});

        return 0;
}
```

This skeleton, by itself, works. It doesn't do anything: it just returns the `result` object as-is. This will result in the algorithm stopping exactly where the error occurs, and returning back to the user. This is because the `result` has an `error_code` member variable, and that member variable, when it reaches the higher level algorithms, stops all encoding, decoding, transcoding, counting, validation, and etc. work and exists with the proper information.

First Parameter

The first parameter is simple enough: it is the encoding that is calling this error handler. If you invoke an `encode_one` or `decode_one` (or a higher-level conversion algorithm) on a *ztd::text::utf8* object, then you can expect a first parameter of type `ztd::text::utf8` to be passed to the error handler.

Note: If the function call `.encode_one` or `.decode_one` is a static function that has no instance, then the encoding object will create a temporary instance to pass to the function. This happens with most encodings that do not contain any pertinent information on the encoding object itself, like all the Unicode encodings and the ASCII/locale/string literal encodings.

This can be handy if you need to access information about the encoding object or encoding type. You can get information about the encoding by using:

- *ztd::text::encode_state_t*
- *ztd::text::decode_state_t*
- *ztd::text::code_unit_t<Encoding>*
- *ztd::text::code_point_t<Encoding>*
- *ztd::text::code_unit_v<Encoding>*
- *ztd::text::code_point_v<Encoding>*

Second Parameter

The second parameter is the result object. It is of the type *ztd::text::decode_result* or *ztd::text::encode_result*. The two types have identical information inside of them, but have different names so that a function call operator can tell the difference between the two, if it's necessary.

This contains all of the state and information that the decode operation/encode operation would return, if left unmodified by the error handler. If you don't want to do anything to it, simply pass it through by returning it with `return result;` . Otherwise, you have access to the `input` range, the `output` range, any `.state` relevant to the operation, the `.error_code`, and the `.error_handled` value. You can modify any one of theses, or even perform a recovery operation and change the `.error_code` to be `ztd::text::encoding_error::ok`. Literally, anything can be done!

For example, someone can see if there is space left in the `result.output` parameter, and if so attempt to serialize a replacement character in place there (this is what *ztd::text::replacement_handler_t* does).

Third Parameter

The third parameter is a contiguous range of input values that were read. Typically, this is a `ztd::span` handed to you, or something that can construct a `ztd::span` or either code units or code points (whatever the output type has). This is useful for `input_range`s and `input_iterator`s where it is impossible to guarantee a value can be written, as is the case with istream_iterator and other I/O-style iterators and ranges.

Fourth Parameter

The fourth parameter is a contiguous range of output values that were almost written to the output, but could not be because the output has no more room left. Typically, this is a `ztd::span` handed to you, or something that can construct a `ztd::span` or either code units or code points (whatever the input type has). This is particularly useful for `output_range`s and `output_iterator`s where there is no way to guarantee all characters will be successfully written, as is the case with ostream_iterator and other I/O-style iterators and ranges.

The fourth parameter is only ever filled out if the error returned is *ztd::text::encoding_error::insufficient_output*. It is **very** important for when someone does bulk-buffered writes, since multiple writes are not guaranteed to fit within the given *ztd::text::max_code_points_v* or *ztd::text::max_code_units_v* for a specific encoding. (They only represent the maximum for a single, atomic operation.)

This is useful for grabbing any would-be-written output data, and storing it for later / completing it. For example, writing to a smaller, contiguous buffer for delivery and looping around that buffer can be faster, but it runs the risk of partial reads/writes on the boundaries of said smaller, contiguous buffer.

Secret Type Definition

There is a type definition you can add to your error handler to signal that it is okay to ignore it's calls. It goes on the struct and looks like:

```
using assume_valid = std::false_type; // or std::true_type
```

This is allows any encoding which uses *ztd::text::is_ignorable_error_handler* property on your error handler to know if it's okay to ignore the error handler when bad things happen. Having this functionality means you can create a "debug handler" for text you previously know is valid, but might want to check during a debug or tracing build or something as it encodes and decodes through the system:

```
struct my_debug_handler {

        // Assume it's valid if the config value
        // is explicitly turned off
        using assume_valid = std::integral_constant<
                bool, (MY_ENCODING_TRACE_IS_TURNED_OFF != 0)
        >;

        // rest of the implementation...
};
```

Writing A Handler

When put together, it can generally look like this:

```
#include <ztd/text/encode.hpp>
#include <ztd/text/encoding.hpp>

#include <iostream>

using ascii_encode_result = ztd::text::encode_result<
     // input range type
     std::u32string_view,
     // output range type; figured out from function call
     ztd::span<char>,
     // the state type for encode operations
     ztd::text::encode_state_t<ztd::text::ascii_t>>;

ascii_encode_result my_printing_handler(const ztd::text::ascii_t& encoding,
     ascii_encode_result result,
     ztd::span<const char32_t> unused_read_characters,
     ztd::span<const char> unused_write_characters) noexcept {
        (void)encoding;
        // just printing some information
        std::cout << "An error occurred.\n"
                  << "\tError code value: "
                  << ztd::text::to_name(result.error_code) << "\n"
                  << "\t# of code unit spaces left: " << result.output.size()
                  << "\n"
                  << "\t# of unused code points: "
```

(continues on next page)

(continued from previous page)

```
                  << unused_read_characters.size() << "\n"
                  << "\n"
                  << "\t# of unused code units: "
                  << unused_write_characters.size() << "\n"
                  << "\tInput units left: " << result.input.size() << "\n";
        // setting the error to "ok"
        // tells the algorithm to keep spinning,
        // even if nothing gets written to the output
        result.error_code = ztd::text::encoding_error::ok;
        return result;
}

int main(int, char*[]) {
        std::string my_ascii_string = ztd::text::encode(
             // input
             U"",
             // to this encoding
             ztd::text::ascii,
             // handled with our function
             &my_printing_handler);

        ZTD_TEXT_ASSERT(my_ascii_string == "");

        return 0;
}
```

The result in `my_ascii_string` should be an empty string: nothing should have succeeded and therefore the function will just return an empty string. The print out will look like this:

```
An error occurred.
        Error code value: invalid_sequence
        # of unused characters: 1
        Input units left: 1
An error occurred.
        Error code value: invalid_sequence
        # of unused characters: 1
        Input units left: 0
```

If you would like the higher-level called function to return more information to you, use the lower level *encode_to/encode_into*, *decode_to/decode_into*, *transcode_to/transcode_into*.

If you need to do more, you can change from concrete types to templates, and work at increasingly higher levels of genericity in order to have the printing handler do more and more.

Lossy Operation Protection

Occasionally, you will end up in a situation where you want to convert some text from its pristine and ideal Unicode form to some other form. Maybe for interopation purposes, maybe because some function call can't properly handle embedded `NUL`s in the text so you need to *use an overlong sequence to encode the 0 value in your text*. No matter what the case is, you need to leave the world of *Unicode Code Points*, *Unicode Scalar Values*, and all the guarantees they provide you. Let's take an example, going from UTF-8 to 7-bit-clean ASCII:

```
#include <ztd/text/transcode.hpp>

#include <iostream>

int main(int, char*[]) {
        // (1)
        std::string my_ascii_string = ztd::text::transcode(
             // input
             u8"",
             // from this encoding
             ztd::text::utf8 {},
             // to this encoding
             ztd::text::ascii {});

        std::cout << my_ascii_string << std::endl;

        return 0;
}
```

This will produce a compile time error (with this error number for MSVC as an example):

> error C2338: The encode (output) portion of this transcode is a lossy, non-injective operation. This means you may lose data that you did not intend to lose; specify an 'out_handler' error handler parameter to `transcode[_to](in, in_encoding, out_encoding, in_handler, out_handler, ...)` or `transcode_into(in, in_encoding, out, out_encoding, in_handler, out_handler, ...)` explicitly in order to bypass this.

The reason this happens is because we can detect, at compile time, that the conversion from Unicode Code Points to ASCII is a lossy transformation. When this happens, we realize the conversion will be a lossy one: therefore, it makes sense that the user cannot perform the encoding or decoding operation without being explicit about how they are going to handle errors because there is such a gigantically enormous possibility that they will mangle incoming text.

Since this library is trying to prevent *Mojibake* and other encoding problems, you are required to tag any potentially-lossy encoding with an error handler, to be explicit and acknowledge that you may or may not be ruining someone's day:

```
#include <ztd/text/transcode.hpp>

#include <iostream>

int main(int, char*[]) {
        std::string my_ascii_string = ztd::text::transcode(
             // input
             u8"",
             // from this encoding
             ztd::text::utf8,
             // to this encoding
```

(continues on next page)

(continued from previous page)

```
            ztd::text::ascii,
            // (1) error handler
            ztd::text::replacement_handler);

        std::cout << my_ascii_string << std::endl; // (2)

        ZTD_TEXT_ASSERT(my_ascii_string == "??");

        return 0;
}
```

Any encoding which does not meet the requirements of either *ztd::text::is_encode_injective_v* or *ztd::text::is_decode_injective_v* (or both, for transcoding which uses both an encode and a decode operation) will throw an error if you specify no error handlers in the text. This is done through the *Injectivity Lucky 7 Extensions* that go beyond the *traditional Lucky 7* with 2 `std::true_type`/`std::false_type` definitions.

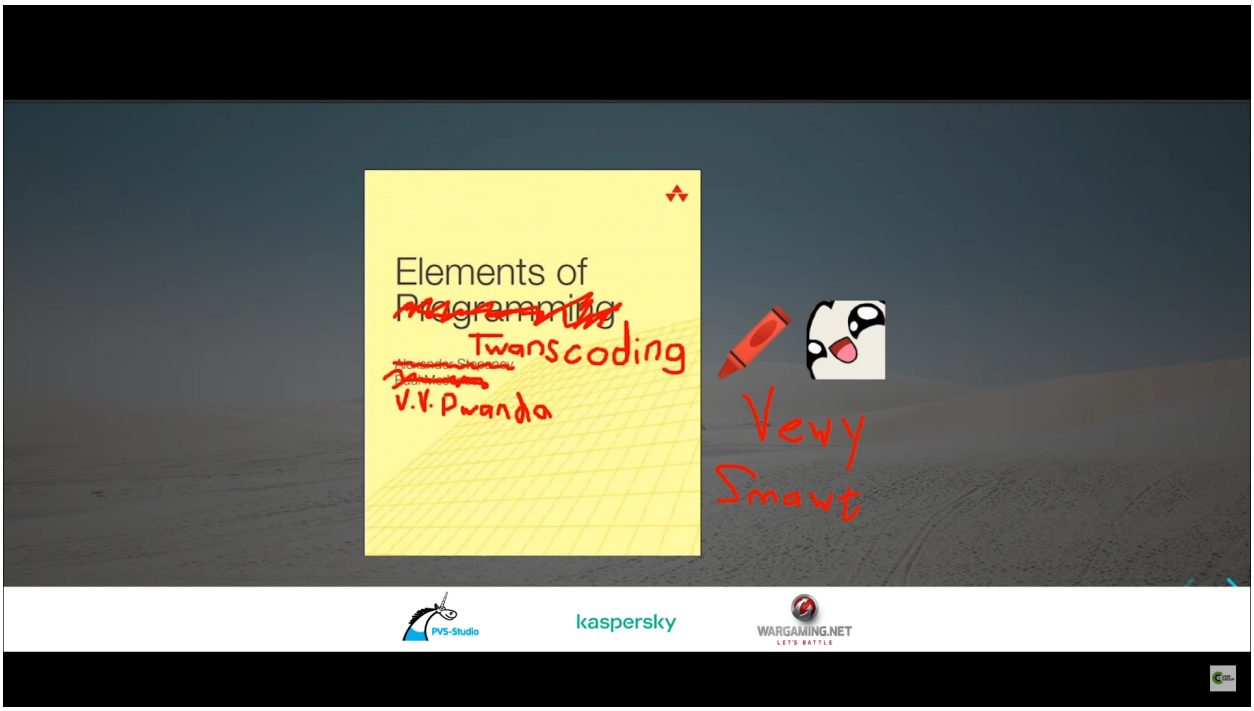
1.5.4 Converting, Counting, and Validating Text

Conversions are one of the more important aspects of dealing with textual data. To support this, ztd.text contains 7 different methods, each with various overloads and inner groupings of functions to aid in encoding, decoding, transcoding, validating, and counting code points and code units.

As shown in the *Lucky 7 Design*, everything here is supported by just having either the required one or two encoding objects with the designated functions, variables and type definitions. The core of the explanation is in this algorithm:

- Is the `input` value empty? Return the current results, everything is okay . Otherwise,
  0. Set up an `intermediate` buffer of `code_point`s using the `max_code_points` of the input encoding count for the next operation.
  1. Do the `decode_one` step from `input` (using its `begin()` and `end()`) into the `intermediate` `code_point` buffer.
     - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `output`, and `state`.
  2. Do the `encode_one` step from the `intermediate` into the `output`.
     - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `output`, and `state`.
- Update `input`'s `begin()` value to point to after what was read by the `decode_one` step.
- Go back to the start.

That's it for the core loop. Failure is determined exclusively by whether or not the `error_code` returned from the decode or encode operation's result object is *ztd::text::encoding_error::ok*. If it is OK, then the loop continues until the `input` is exhausted. Otherwise, it stops. This forms the basis of the library, and will essentially be our version of "Elements of Programming", but for working with Text:



The above algorithm can work for all the below operations:

- **transcoding**: the above loop presented as-is.
- **encoding**: take an `input` of `code_points`, and simply do not do the decoding step.
- **decoding**: take an `input` of `code_units`, and simply do not do the encoding step.
- **validating code units**: do the transcoding loop into 2 intermediate buffers, and compare the result of the final `intermediate` output to the `input`.
- **validating code points**: do the transcoding loop, but in the reverse direction for an `input` of `code_points` (encode first, then decode) into 2 intermediate buffers, and compare the result of the final `intermediate` output to the `input`.
- **counting code units**: perform the "encoding" operation into an intermediate buffer and repeatedly count the number of buffered writes, discarding or ignoring the actual contents of the buffer each time.
- **counting code points**: perform the "decoding" operation into an intermediate buffer and repeatedly count the number of buffered writes, discarding or ignoring actual the contents of the buffer each time.

This covers the full universe of potential operations you may want to perform on encoded text, for the purposes of input and output. If you implement the *base Lucky 7* or implement the *extended Lucky 7* for an encoding, you can gain access to the full ecosystem of encodings within your application.

Encode

Encoding is the action of converting from one sequence of decoded information to a sequence of encoded information. The formula given for Encoding is effectively just the first half of the diagram shown in the *main Lucky 7 documentation*, reproduced here with emphasis added:

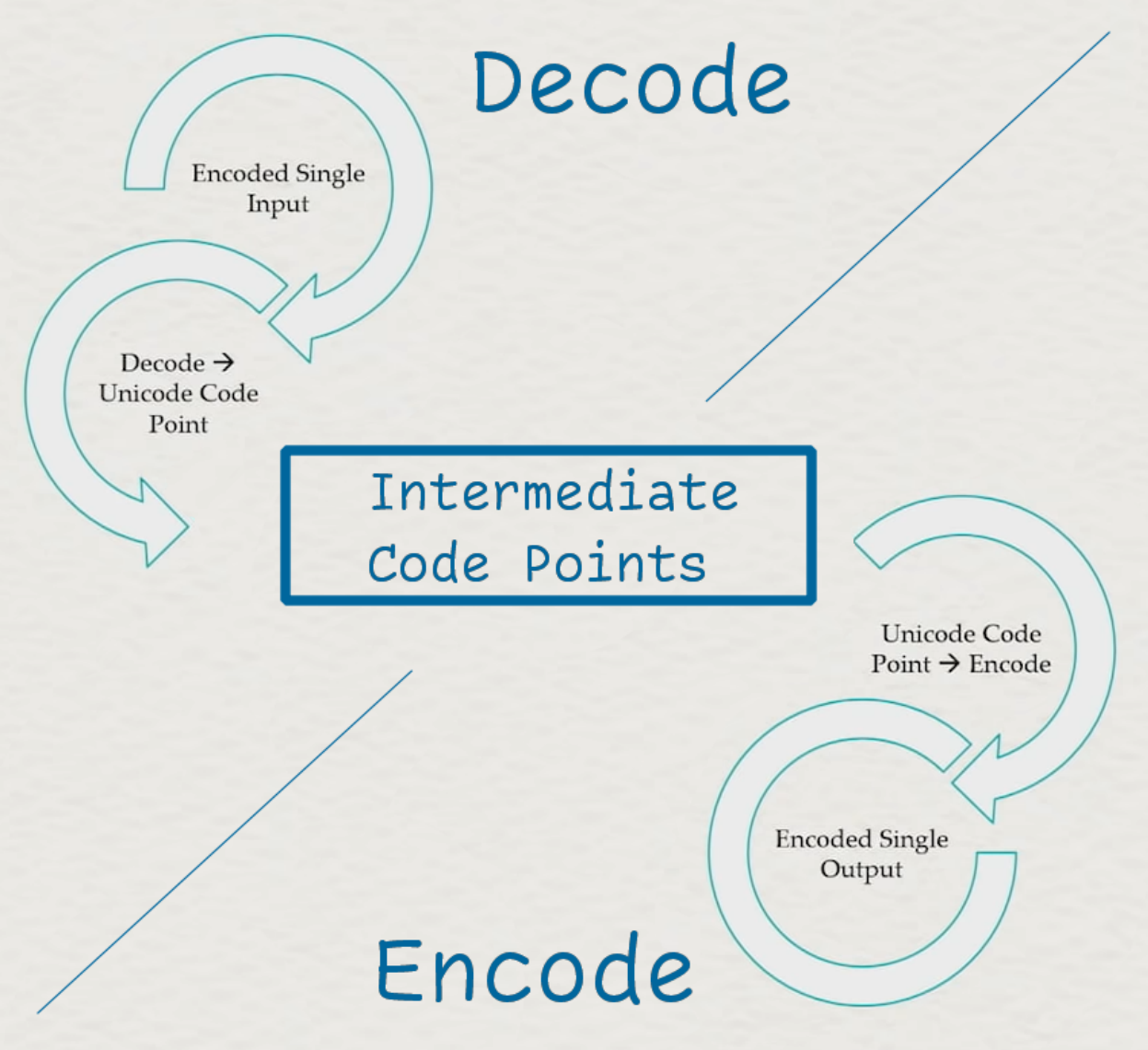


Fig. 2: The generic pathway between 2 encodings, but modified to show the exact difference between the encoding step and the decoding step.

In particular, we are interested in the operation that helps us go from the decoded input to the encoded output, which is the bottom half of the diagram. The input in this case is labeled "intermediate", because that is often what it is. But, there are many uses for working directly with the encoded data. A lot of the world does not speak directly in 21-bit Unicode Code Points, but instead speaks in UTF-8. Legacy systems are often found communicating with Code Pages (e.g., EBCDIC or SHIFT-JIS); until those systems go down or are replaced, it is imperative to send them well-formed data, whether over a network or across an inter-process communication bridge of any kind.

Thusly, we use the algorithm as below to do the work. Given an `input` of `code_points` with an `encoding`, a target `output`, and any necessary additional `state`, we can generically convert that sequence of `code_points` into its

encoded form:

- Is the `input` value empty? Return the current results with the the empty `input`, `output`, and `state`, everything is okay ! Otherwise,

  0. Do the `encode_one` step from `input` (using its `begin()` and `end()`) into the `output` `code_unit` storage location.

     - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `output`, and `state`s.

- Update `input`'s `begin()` value to point to after what was read by the `encode_one` step.
- Go back to the start.

This involves a single encoding type, and so does not need any cooperation to go from the `code_point` sequence to the `code_unit` sequence.

Check out the API documentation for *ztd::text::encode* to learn more.

Decode

Decoding is the action of converting from one sequence of encoded information to a sequence of decoded information. The formula given for Decoding is effectively just the first half of the diagram shown in the *main Lucky 7 documentation*, reproduced here with emphasis added:

In particular, we are interested in the operation that helps us go from the encoded input to the decoded output, which is the top half of the diagram. The output we are interested in is labeled as an "intermediate", because that is often what it is. But, there are many uses for working directly with the decoded data. Many Unicode algorithms are specified to work over *unicode code points* or *unicode scalar values*. In order to identify Word Breaks, classify Uppercase vs. Lowercase, perform Casefolding, Regex over certain properties properly, Normalize text for search + other operations, and many more things, one needs to be working with code points as the basic unit of operation.

Thusly, we use the algorithm as below to do the work. Given an `input` of `code_units` with an `encoding`, a target `output`, and any necessary additional `state`, we can generically bulk convert the input sequence to a form of `code_points` in the `output`:

- Is the `input` value empty? Return the current results with the the empty `input`, `output`, and `state`, everything is okay ! Otherwise,

  0. Do the `decode_one` step from `input` (using its `begin()` and `end()`) into the `output` `code_point` storage location.

     - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `output`, and `state`s.

- Update `input`'s `begin()` value to point to after what was read by the `decode_one` step.
- Go back to the start.

This involves a single encoding type, and so does not need any cooperation to go from the `code_units` to the `code_points`. Notably, the encoding's `code_point` type will hopefully be some sort of *unicode code point* type (see: *ztd::text::is_code_point* for a more code-based classification). Though, it does not have to be for *many different (and very valid) reasons*.

Check out the API documentation for *ztd::text::decode* to learn more.

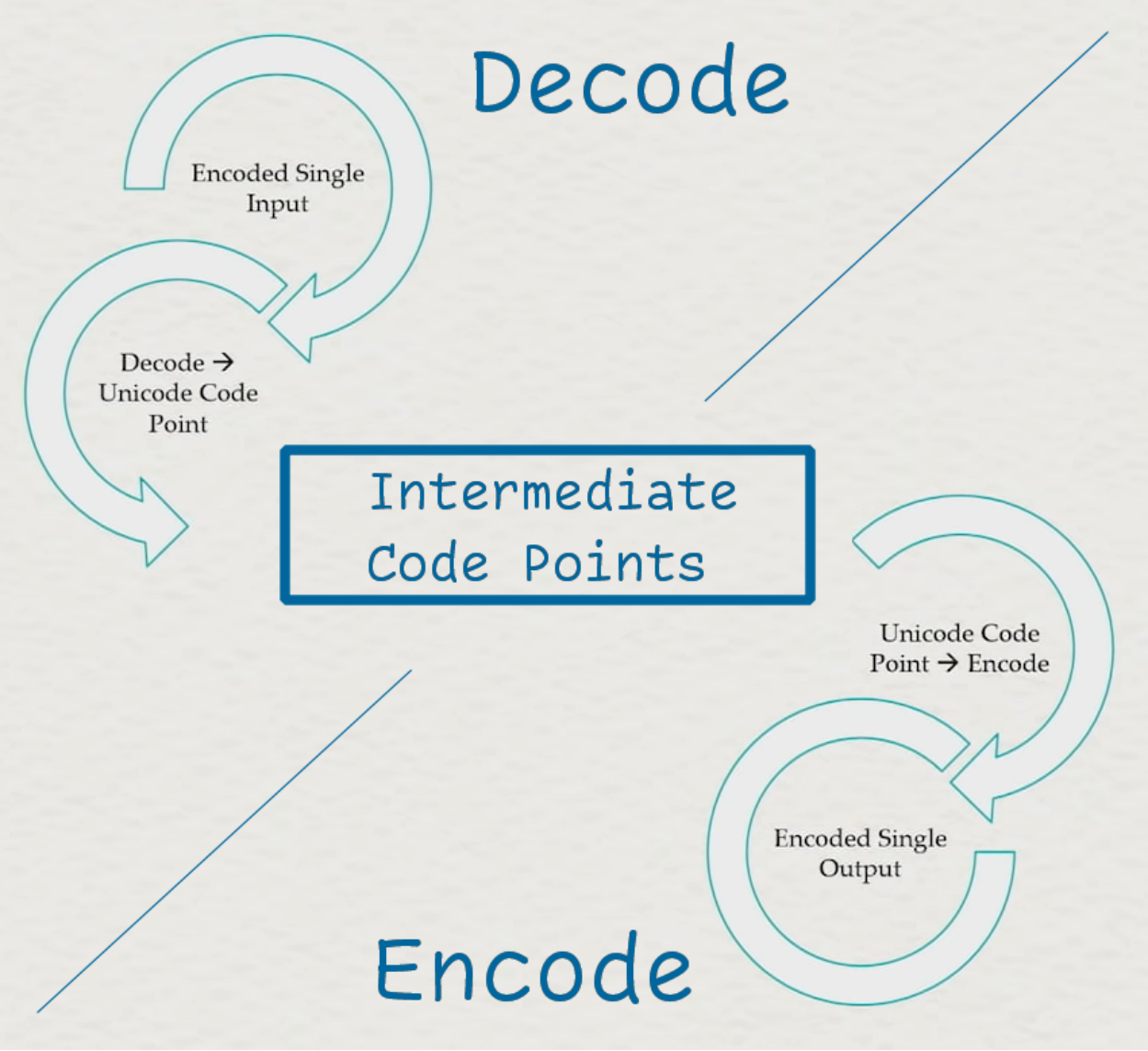


Fig. 3: The generic pathway between 2 encodings, but modified to show the exact difference between the encoding step and the decoding step.

Transcode

Transcoding is the action of converting from one sequence of encoded information to another sequence of (usually differently) encoded information. The formula given for Transcoding is actually exactly the same as the one shown in the *main Lucky 7 documentation*, reproduced here:

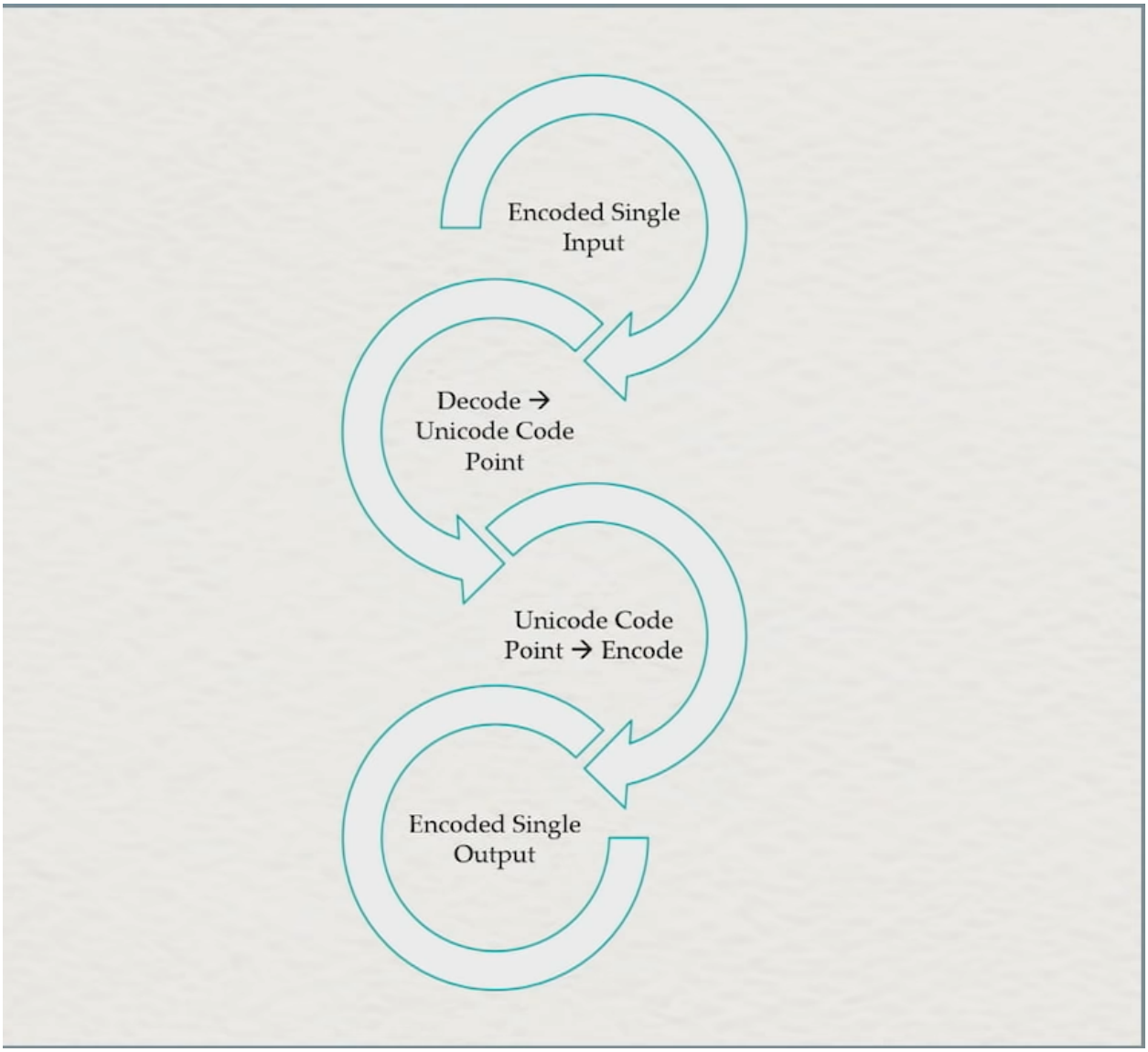


Fig. 4: The generic pathway from one encoding to another for most text Encodings.

The core tenant here is that as long as there is a common intermediary between the 2 encodings, you can decode from the given input into that shared common intermediary (e.g., *unicode code points* or *unicode scalar values*), then encode from the common intermediary to the second encoding's output. This is a pretty basic way of translating data and it's not even a particularly new idea (*iconv* has been doing this for well over a decade now, *libogonek* got this core idea rolling in a C++ interface, and in general this is quite literally how all data interchange has been done since forever). The equalizer here is that, unlike other industries that struggle to define an interchange format, Unicode Code Points has become the clear and overwhelming interoperation choice for people handling text all over the world.

Thusly, we use the algorithm as below to do the work. Given an `input` of `code_units` with a `from_encoding`, a `to_encoding` with a target `output`, and any necessary additional `states`, we can generically convert that one encoding

to the other so long as those encodings follow the Lucky 7 design:

- Is the `input` value empty? Return the current results with the the empty `input`, `output`, and `state`s, everything is okay ! Otherwise,
    0. Set up an `intermediate` storage location of `code_point`s, using the `max_code_points` of the input encoding as the maximum size of the storage location, for the next operation.
    1. Do the `decode_one` step from `input` (using its `begin()` and `end()`) into the `intermediate` `code_point` storage location.
        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `output`, and `state`s.
    2. Do the `encode_one` step from the `intermediate` into the `output`.
        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `output`, and `state`s.
- Update `input`'s `begin()` value to point to after what was read by the `decode_one` step.
- Go back to the start.

This fundamental process works for any 2 encoding pairs, and does not require the first encoding `from_encoding` to know any details about the second encoding `to_encoding`! This means a user is only responsible for upholding their end of the bargain with their encoding object, and can thusly interoperate with every other encoding that speaks in the same intermediade, decoded values (i.e. *unicode code points*).

Check out the API documentation for *ztd::text::transcode* to learn more.

Validate Encodable

Validation is the way to verify a given sequence of input can have a specific action performed on it. Particularly, we check here if the input of code points can be turned into code units of the given encoding. The way it does this, however, is two-fold:

- it first encodes the input code units, to see if it can do the transformation without loss of information; then,
- it decodes the output from the last step, to see if the final output is equivalent to the input.

The algorithm for this is as follows:

- Is the `input` value empty? Return the current results with the the empty `input`, `valid` set to `true` and `state`s, everything is okay ! Otherwise,
    0. Set up an `intermediate` storage location of `code_unit`s, using the `max_code_units` of the input encoding, for the next operations.
    1. Set up an `intermediate_checked_output` storage location of `code_point`s, using the `max_code_points` of the input encoding, for the next operations.
    2. Do the `encode_one` step from `input` (using its `begin()` and `end()`) into the `intermediate` `code_unit` storage location.
        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `valid` set to false, and `state`s.
    3. Do the `decode_one` step from the `intermediate` into the `intermediate_checked_output`.
        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `valid` set to false, and `state`s.
    4. Compare the `code_point`s of the `input` sequentially against the `code_point`s within the `intermediate_checked_output`.

    - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `valid` set to false, and `state`s.

- Update `input`'s `begin()` value to point to after what was read by the `decode_one` step.
- Go back to the start.

This fundamental process works for all encoding objects, provided they implement the basic *Lucky 7*. The reason for checking if it can be turned back is to ensure that the input code units actually match up with the output code units. If an encoding performs a lossy transformation in one direction or the other, then validation will fail if it cannot reproduce the input exactly. And, you will know the exact place in the `input` that caused such a failure.

There are extension points used in the API that allow certain encodings to get around the limitation of having to do both the `encode_one` step and the `decode_one` step, giving individual encodings control over the verification of a single unit of input and of bulk validation as well.

Check out the API documentation for *ztd::text::validate_encodable_as* to learn more.

Validate Decodable

Validation is the way to verify a given sequence of input can have a specific action performed on it. Particularly, we check here if the input of code units can be turned into code points of the given encoding. The way it does this, however, is two-fold:

- it first decodes the input code units, to see if it can do the transformation without loss of information; then,
- it encodes the output from the last step, to see if the final output is equivalent to the input.

The algorithm for this is as follows:

- Is the `input` value empty? Return the current results with the the empty `input`, `valid` set to true, and `state`s, everything is okay ! Otherwise,

    0. Set up an `intermediate` storage location of `code_point`s, using the `max_code_points` of the input encoding, for the next operations.
    1. Set up an `intermediate_checked_output` storage location of `code_unit`s, using the `max_code_units` of the input encoding, for the next operations.
    2. Do the `decode_one` step from `input` (using its `begin()` and `end()`) into the `intermediate` `code_point` storage location.

        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `valid` set to false, and `state`s.

    3. Do the `encode_one` step from the `intermediate` into the `intermediate_checked_output`.

        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `valid` set to false, and `state`s.

    4. Compare the `code_unit`s of the `input` sequentially against the `code_unit`s within the `intermediate_checked_output`.

        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `valid` set to false, and `state`s.

- Update `input`'s `begin()` value to point to after what was read by the `decode_one` step.
- Go back to the start.

This fundamental process works for all encoding objects, provided they implement the basic *Lucky 7*. The reason for checking if it can be turned back is to ensure that the input code units actually match up with the output code units. If

an encoding performs a lossy transformation in one direction or the other, then validation will fail if it cannot reproduce the input exactly. And, you will know the exact place in the `input` that caused such a failure.

There are extension points used in the API that allow certain encodings to get around the limitation of having to do both the `decode_one` step and the `encode_one` step, giving individual encodings control over the verification of a single unit of input and of bulk validation as well.

Check out the API documentation for *ztd::text::validate_decodable_as* to learn more.

Validate Decodable

Validation is the way to verify a given sequence of input can have a specific action performed on it. Particularly, we check here if the input of code units can be turned into code points of the given encoding. The way it does this, however, is two-fold:

- it first decodes the input code units, to see if it can do the transformation without loss of information; then,
- it encodes the output from the last step.

The algorithm for this is as follows:

- Is the `input` value empty? Return the current results with the the empty `input`, `valid` set to true, and `states`, everything is okay ! Otherwise,
    0. Set up an `intermediate` storage location of `code_points`, using the `max_code_points` of the input encoding, for the next operations.
    1. Set up an `intermediate_checked_output` storage location of `code_units`, using the `max_code_units` of the output encoding, for the next operations.
    2. Do the `decode_one` step from `input` (using its `begin()` and `end()`) into the `intermediate` `code_point` storage location.
        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `valid` set to false, and `states`.
    3. Do the `encode_one` step from the `intermediate` into the `intermediate_checked_output`.
        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), `valid` set to false, and `states`.
- Update `input`'s `begin()` value to point to after what was read by the `decode_one` step.
- Go back to the start.

This fundamental process works for all encoding objects, provided they implement the basic *Lucky 7*. Unlike the encode and decode validation functions, this one does not have anything to compare its output to. By virtue of converting from the source to the destination, it is *transcodable*. Whether or not it can be round-tripped in the other direction isn't particularly of concern, just that it can do so without error. This is the more general purpose forms of the encode or decode operations.

There are extension points used in the API that allow certain encodings to get around the limitation of having to do both the `decode_one` step and the `encode_one` step, giving individual encodings control over the verification of a single unit of input and of bulk validation as well.

Check out the API documentation for *ztd::text::validate_transcodable_as* to learn more.

Count as Decoded

Counting code units is the action of finding out how many code points will result from a given sequence of encoded information. Essentially, we run the *decoding algorithm* loop, but instead of giving the end user the decoded values, we instead simply provide the count for running that bulk operation.

Thusly, we use the algorithm as below to do the work. Given an `input` of `code_units` with an `encoding`, an initial `count` set at 0, and any necessary additional `state`, we can generically predict how many code units will result from a decoding operation by running the following loop:

- Is the `input` value empty? Return the current results with the the empty `input`, curent `count`, and `state`, everything is okay ! Otherwise,

    0. Set up an `intermediate` storage location of `code_points`, using the `max_code_points` of the input encoding, for the next operations.

    1. Do the `decode_one` step from `input` (using its `begin()` and `end()`) into the `intermediate` `code_point` storage location, saving the returned `intermediate_output` from the `decode_one` call.

        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), current `count`, and `state`s.

    2. Compute the difference between the `begin(intermediate)` from the original step, and the `begin(intermediate_output)` returned by `decode_one`; add that difference to the current `count`.

- Update `input`'s `begin()` value to point to after what was read by the `decode_one` step.
- Go back to the start.

This involves a single encoding type, and so does not need any cooperation to count the `code_units`. Note that this algorithm doesn't show what the error handler does; if the error handler "erases" the failure by setting the result type's `.error_code == ztd::text::encoding_error::ok`, then the algorithm will keep going. This is useful to, for example, detect the maximum size of an operation even if it errors and would result in replacement characters being inserted (e.g., from *ztd::text::replacement_handler_t*).

Check out the API documentation for *ztd::text::count_as_decoded* to learn more.

Count as Encoded

Counting encodable data is the action of finding out how many code units will result from a given sequence of already decoded information, AKA a sequence of code points. Essentially, we run the *encoding algorithm* loop, but instead of giving the end user the encoded values, we instead simply provide the count for running that bulk operation.

Thusly, we use the algorithm as below to do the work. Given an `input` of `code_units` with an `encoding`, an initial `count` set at 0, and any necessary additional `state`, we can generically predict how many code units will result from a decoding operation by running the following loop:

- Is the `input` value empty? Return the current results with the the empty `input`, curent `count`, and `state`, everything is okay ! Otherwise,

    0. Set up an `intermediate` storage location of `code_units`, using the `max_code_units` of the input encoding, for the next operations.

    1. Do the `encode_one` step from `input` (using its `begin()` and `end()`) into the `intermediate` `code_unit` storage location, saving the returned `intermediate_output` from the `encode_one` call.

        - If it failed, return with the current `input` (unmodified from before this iteration, if possible), current `count`, and `state`s.

    2. Compute the difference between the `begin(intermediate)` from the original step, and the `begin(intermediate_output)` returned by `encode_one`; add that difference to the current `count`.

- Update `input`'s `begin()` value to point to after what was read by the `encode_one` step.
- Go back to the start.

This involves a single encoding type, and so does not need any cooperation to count the `code_point`s. Note that this algorithm doesn't show what the error handler does; if the error handler "erases" the failure by setting the result type's `.error_code == ztd::text::encoding_error::ok`, then the algorithm will keep going. This is useful to, for example, detect the maximum size of an operation even if it errors and would result in replacement characters being inserted (e.g., from *ztd::text::replacement_handler_t*).

Check out the API documentation for *ztd::text::count_as_encoded* to learn more.

Count as Transcoded

This operation counts how much text will result from a transcode operation. Essentially, we run the *encoding algorithm* loop, but instead of giving the end user the re-encoded values, we instead simply provide the count for running that bulk operation.

Thusly, we use the algorithm as below to do the work. Given an `input` of `code_unit`s with an `encoding`, an initial `count` set at 0, and any necessary additional `state`, we can generically predict how many code units will result from a decoding operation by running the following loop:

- Is the `input` value empty? Return the current results with the the empty `input`, curent `count`, and `state`, everything is okay ! Otherwise,
  0. Set up an `intermediate` storage location of `code_point`s (of the input encoding), using the `max_code_points` of the input encoding; and, set up an `intermediate_output` storage location of `code_unit`s (of the output encoding), for the next operations.
  1. Do the `decode_one` step from `input` (using its `begin()` and `end()`) into the `intermediate` `code_point` storage location, saving the returned `intermediate_output` from the `decode_one` call.
     - If it failed, return with the current `input` (unmodified from before this iteration, if possible), current `count`, and `state`s.
  2. Do the `encode_one` step from `intermdiate` (using its `begin()` and `end()`) into the `intermediate_output` `code_unit` storage location, saving the returned `intermediate_output` from the `encode_one` call.
     - If it failed, return with the current `input` (unmodified from before this iteration, if possible), current `count`, and `state`s.
  3. Compute the difference between the `begin(intermediate_output)` from the original step, and the `begin(result.output)` returned by `encode_one`; add that difference to the current `count`.
- Update `input`'s `begin()` value to point to after what was read by the `encode_one` step.
- Go back to the start.

This involves a single encoding type, and so does not need any cooperation to count the `code_point`s. Note that this algorithm doesn't show what the error handler does; if the error handler "erases" the failure by setting the result type's `.error_code == ztd::text::encoding_error::ok`, then the algorithm will keep going. This is useful to, for example, detect the maximum size of an operation even if it errors and would result in replacement characters being inserted (e.g., from *ztd::text::replacement_handler_t*).

Check out the API documentation for *ztd::text::count_as_transcoded* to learn more.

1.5.5 Strong vs. Weak Code Units/Points and Legacy Encodings

Every encoding object must have `code_point` and `code_unit` type definitions on it. Typically, this is set to *ztd::text::unicode_code_point*. But, if you go through a Prior Work for this library, you will notice Tom Honermann's reference implementation for text_view has a concept of even more strictly controlled `code_unit` and `character_type` than this library. From the associated paper:

> This library defines a character class template parameterized by character set type used to represent character values. The purpose of this class template is to make explicit the association of a code point value and a character set.
>
> ...
>
> It has also been suggested that `char32_t` might suffice as the only character type; that decoding of any encoded string include implicit transcoding to Unicode code points. The author believes that this suggestion is not feasible...
>
> —Tom Honermann, P0244 text_view

The Case for Strength

This general philosophy in Honermann's text_view means that you do not just use `unsigned char` or `unicode_code_point` for code unit and code point types, but instead traffic more directly in, for example, `ebcdic_char` and `ebcdic_code_point` types. They are essentially strong type definitions and strong wrappers simpler, "lower level" types like `char32_t` and `char`. It has the following tradeoffs:

- ✓ Can directly connect a range and its `value_type` to a specific encoding (e.g., `default_code_point_encoding_t<ascii_code_point>` means `ascii`, definitively).
- ✓ Actively prevents passing one type of range/view to a function expecting another (e.g., `std::basic_string<ascii_char>` cannot accidentally be given to a function expecting `std::string`, where the expectation might be for an *execution encoded* string.)
- ✓ Easy to strip out all encoding/codec information and the range types themselves can still recover it (e.g. `ascii_code_point* u32_c_str_ptr` can be strongly associated with the `ascii` encoding, whereas `unicode_code_point* u32_c_str_ptr` loses all that information.)
- Requires `reinterpret_cast` or `std::memcpy`/`std::copy` to work with most existing code that do not have such strongly typed pointers.
- Can generate a lot of template type spam for what are essentially just `char`.
- Not very good in `constexpr`, where `reinterpret_cast` isn't allowed and there are pre-existing `constexpr` functions that are not templated.

The question boils down to: should we have strong code point and code unit types **by default** in the library?

Henri Sivonen — author of encoding_rs and expert in the text domain — strongly disagrees.

The Counterpoint

In a long piece on P0422, the C and C++ landscape, and Standardization efforts, Henri writes:

> I think the C++ standard should adopt the approach of "Unicode-only internally" for new text processing facilities and should not support non-Unicode execution encodings in newly-introduced features. This allows new features to have less abstraction obfuscation for Unicode usage, avoids digging legacy applications deeper into non-Unicode commitment, and avoids the specification and implementation effort of adapting new features to make sense for non-Unicode execution encodings.
>
> —Henri Sivonen, It's Time to Stop Adding New Features for Non-Unicode Execution Encodings in C++

This is a different set of choices and a different set of priorities from the outset. Sivonen's work specifically is that with Browsers and large code bases like Firefox; they are responsible for making very good traction and progress on encoding issues in a world that is filled primarily with Unicode, but still has millions of documents that are not in Unicode and, for the foreseeable future, won't end up as Unicode.

This is a strong argument for simply channeling `char16_t`, `char32_t`, and – since C++20 – `char8_t` as the only types one would need. Firefox at most deals in UTF-16 (due to the JavaScript engine for legacy reasons) and UTF-8, internally. At the boundaries, it deals with many more text encodings, because it has to from the world wide web. Occasionally, UTF-32 will appear in someone's codebase for interoperation purposes or algorithms that need to operate on something better than code units.

Unicode is also… well, a [UNI]versal [CODE]. Its purposes are interoperation, interchange, and common ground between all the encodings, and it has been the clear winner for this for quite some time now. Sivonen makes a compelling point for just considering Unicode — and only Unicode — for all future text endeavors.

Do we really need to focus on having support for legacy encodings? Or at least, do we really need support for legacy encodings at the level that Tom Honermann's text_view is trying to achieve?

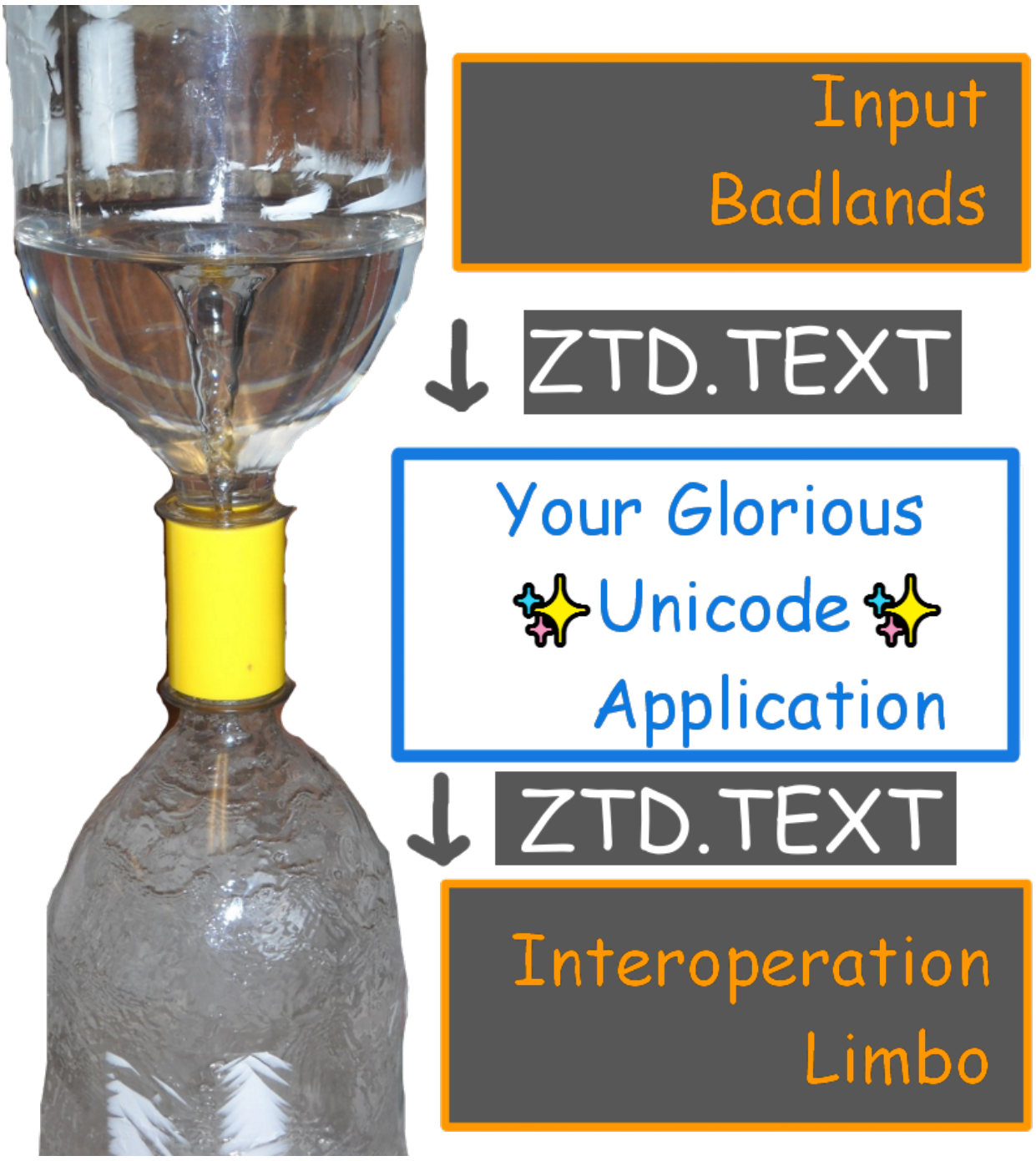
ztd.text's answer is simple:

Allow Both, Prefer One

ztd.text prefers Henri Sivonen's approach to the library in general. The `code_unit` type is generally a weakly-typed choice of one of the 6 viable code unit types in C++ (`char`, `wchar_t`, `unsigned char`, `char8_t`, `char16_t`, and `char32_t`). The `code_point` type is typically just `unicode_code_point` (an alias for `char32_t` *by default*) or `unicode_scalar_value` (a strong type *by default*, because it carries extra pertinent information about itself that can aid the library). Unfortunately, this means that *ztd::text::default_code_point_encoding_t* is not a very rich type mapping (it generally just spits out UTF-8).

This does not mean all future algorithms bear the burden of supporting an infinity of text encodings. But, the work for encoding and decoding text is isolated and constrained specifically to the *encoding objects*, *view types*, and *functions* that power this library. Down-stream algorithms — like those found in Zach Laine's Boost.Text — work only with range/iterator types whose `value_type` are either `unicode_code_point`s or `unicode_scalar_value`s.

By having a core, standard ecosystem that works primarily with `unicode_code_point` and `unicode_scalar_value`, we heavily incentivize the use of these two types as the only interchange types. Furthermore, because all of the encodings provided by this library use `unicode_code_point` as their `code_point` type, we set a strong example for the rest of the ecosystem who may work with and look at these files. This is even the case for the default byte-based encoding *ztd::text::any_encoding*, which strongly incentivizes compatibility with the ecosystem by making it clear that there is a preferred default mode of communication (which is, *ztd::text::unicode_code_point*). In effect, we produce The Unicode™ Vortex™:



This might be the perfect world for most people, but even so there's room inside that funneled vortex for more.

Leaving Room

There is room in Sivonen's world, even with perfectly-consistent and fully-Unicode internals, for Honermann's dream of never losing encoding information at even the lowest levels. After all, if someone takes the time to wrap up external interfaces (Shared Pipes, Network Connections, Terminal Interfaces, `char` Devices, and more), they should have the ability to tag these interfaces with either encoding objects or strong, `reinterpret_cast`-able pointer values.

That's why encodings can still define their own `code_unit` and `code_point` types; even if this library — or the Standard Library — traffics in strictly `unicode_code_point`s, it doesn't mean the user should be forced to do that if they are willing to put in the effort for a more type-safe world.

Being able to know, at compile time, without any objects or markup, that a particular pointer + size pairing is meant for a specific encoding is a powerful way to maintain invariants and track the flow of data without runtime cost through a program. It can also make it easy to find places where external, non-Unicode data is making it "too far" into the system, and try to push a conversion closer to the edges of the program.

While ztd.text will traffic and work with `char32_t` and consider it a `unicode_code_point` value *under most circumstances*, users are free to define and extend this classification for their own types and generally create as strict (or loose) as taxonomy as they desire.

In Sum

The library still overwhelmingly traffics in Unicode, and we believe that by making it the default and producing an overwhelming body of code that treats it as such as can push people towards that default. Using `char32_t`s, `unicode_code_point`s, and `unicode_scalar_value`s as Sivonen advocates should have a very low "activation energy". Reaching for the strict world envisioned with Honermann's `text_view` and its associated implementation is still possible, but requires more energy. We do not force the user to put in that energy.

As long as both are possible, users can find satisfaction for both of their use cases. Even if Honermann's design is more work, it is still quite useful and can lead to a very robust and statically-verifiable design in even huge, complex software systems.

1.5.6 Lucky 7 Extension - Beyond the Basics

While the given *Lucky 7* represents the simplest possible encoding object one can design, there are several more type definitions, member functions, and other things an individual can use to create more complex encoding objects. Below, we are going to review the most pertinent ones that allow for better extensibility of the core design and let you go Even Further Beyond.

Separate Encode/Decode States

It is no secret that encoding and decoding may carrying with them separate states. While converting from a legacy encoding to Unicode may require maintenance of a shift state or code unit modifier, the opposite direction may not need any at all. Therefore, as an optimization, an encoding object can define both an `encode_state` and a `decode_state`, seperate from each other. As an example, here is a (simplified) version of how *ztd::text::execution*, the encoding for the Locale-based Runtime Execution Encoding, has two seperate states that need to be initialized in different manners:

```
class runtime_locale {
public:
        struct decode_state {
                std::mbstate_t c_stdlib_state;

```

(continues on next page)

(continued from previous page)

```
                decode_state() noexcept : c_stdlib_state() {
                        // properly set for mbrtoc32 state
                        code_point ghost_ouput[2] {};
                        UCHAR_ACCESS mbrtoc32(
                             ghost_ouput, "\0", 1, &c_stdlib_state);
                }
        };

        struct encode_state {
                std::mbstate_t c_stdlib_state;

                encode_state() noexcept : c_stdlib_state() {
                        // properly set for c32rtomb state
                        code_unit ghost_ouput[MB_LEN_MAX] {};
                        UCHAR_ACCESS c32rtomb(ghost_ouput, U'\0', &c_stdlib_state);
                }
        };
        (void)argc;
```

This is the proper way to initialize a `std::mbstate_t` from the C standard library. Then, you can use it! Here's a complete implementation using the new `encode_state` and `decode_state` types:

```
class runtime_locale {
        using rtl_decode_result
             = ztd::text::decode_result<ztd::span<const code_unit>,
                  ztd::span<code_point>, decode_state>;
        using rtl_encode_result
             = ztd::text::encode_result<ztd::span<const code_point>,
                  ztd::span<code_unit>, encode_state>;
        using rtl_decode_error_handler = std::function<rtl_decode_result(
             const runtime_locale&, rtl_decode_result, ztd::span<const char>,
             ztd::span<const char32_t>)>;
        using rtl_encode_error_handler = std::function<rtl_encode_result(
             const runtime_locale&, rtl_encode_result,
             ztd::span<const char32_t>, ztd::span<const char>)>;

        using empty_code_unit_span  = ztd::span<const code_unit, 0>;
        using empty_code_point_span = ztd::span<const code_point, 0>;

public:
        rtl_decode_result decode_one(
             ztd::span<const code_unit> input, ztd::span<code_point> output,
             rtl_decode_error_handler error_handler,
             decode_state& current // decode-based state
        ) const {
                if (output.size() < 1) {
                        return error_handler(*this,
                             rtl_decode_result(input, output, current,
                                  ztd::text::encoding_error::
                                       insufficient_output_space),
                             empty_code_unit_span(), empty_code_point_span());
                }
```

(continues on next page)

(continued from previous page)

```
                std::size_t result = UCHAR_ACCESS mbrtoc32(output.data(),
                     input.data(), input.size(), &current.c_stdlib_state);
                switch (result) {
                case (std::size_t)0:
                        // '\0' was encountered in the input
                        // current.c_stdlib_state was "cleared"
                        // '\0' character was written to output
                        return rtl_decode_result(
                             input.subspan(1), output.subspan(1), current);
                        break;
                case (std::size_t)-3:
                        // no input read, pre-stored character
                        // was written out
                        return rtl_decode_result(input, output.subspan(1), current);
                case (std::size_t)-2:
                        // input was an incomplete sequence
                        return error_handler(*this,
                             rtl_decode_result(input, output, current,
                                  ztd::text::encoding_error::incomplete_sequence),
                             empty_code_unit_span(), empty_code_point_span());
                        break;
                case (std::size_t)-1:
                        // invalid sequence!
                        return error_handler(*this,
                             rtl_decode_result(input, output, current,
                                  ztd::text::encoding_error::invalid_sequence),
                             empty_code_unit_span(), empty_code_point_span());
                }
                // everything as fine, then
                return rtl_decode_result(
                     input.subspan(result), output.subspan(1), current);
        }

        rtl_encode_result encode_one(
             ztd::span<const code_point> input, ztd::span<code_unit> output,
             rtl_encode_error_handler error_handler,
             encode_state& current // encode-based state
        ) const {
                // saved, in case we need to go
                // around mulitple times to get
                // an output character
                ztd::span<const code_point> original_input = input;
                // The C standard library assumes
                // it can write out MB_CUR_MAX characters to the buffer:
                // we have no guarantee our output buffer is that big, so it
                // needs to go into an intermediate buffer instead
                code_unit intermediate_buffer[MB_LEN_MAX];

                for (int times_around = 0;; ++times_around) {
                        if (input.size() < 1) {
                                // no more input: everything is fine
                                return rtl_encode_result(input, output, current);
```

(continues on next page)

(continued from previous page)

```
                        }
                        std::size_t result
                             = UCHAR_ACCESS c32rtomb(intermediate_buffer,
                                  *input.data(), &current.c_stdlib_state);
                        if (result == (std::size_t)-1) {
                                // invalid sequence!
                                return error_handler(*this,
                                     rtl_encode_result(original_input, output, current,
                                          ztd::text::encoding_error::invalid_sequence),
                                     empty_code_point_span(), empty_code_unit_span());
                        }
                        else if (result == (std::size_t)0) {
                                // this means nothing was output
                                // we should probably go-around again,
                                // after modifying input
                                input = input.subspan(1);
                                continue;
                        }
                        // otherwise, we got something written out!
                        if (output.size() < result) {
                                // can't fit!!
                                return error_handler(*this,
                                     rtl_encode_result(original_input, output, current,
                                          ztd::text::encoding_error::
                                               insufficient_output_space),
                                     empty_code_point_span(), empty_code_unit_span());
                        }
                        ::std::memcpy(output.data(), intermediate_buffer,
                             sizeof(*intermediate_buffer) * result);
                        input  = input.subspan(1);
                        output = output.subspan(result);
                        break;
                }
                return rtl_encode_result(input, output, current);
        }
};

int main(int argc, char* argv[]) {
        (void)argc;
```

This allows you to maintain 2 different states, initialized in 2 different ways, one for each of the `encode_one` and `decode_one` paths.

Injective: Promoting Safety in Encodings

As detailed in the *Lossy Operation Protection* section, `is_encode_injective` and `is_decode_injective` help the library understand when a conversion you are doing cannot be guaranteed at compile time to be lossless. *Injectivity* is a high-brow mathematical term:

> In mathematics, an injective function (also known as injection, or one-to-one function) is a function that maps distinct elements of its domain to distinct elements of its codomain.
>
> —Wikipedia, February 2nd, 2021

This is very fancy speak for the fact that for every complete, well-formed input value, there is a well-formed, distinct output value. It does not have to cover all of the potential output values: so long as there is a one-to-one mapping that is unambigious for all the input values, it is injective. For practical purposes, it means that all of the code unit sequences that are valid can produce a unique code point sequence ("the decode operation is injective"). And, in the reverse case, it means that all the code point sequences that are valid can produce a unique code unit sequence ("the encode operation is injective").

These two properties appear on the type itself, and is a way to opt-in to saying that a conversion is not lossy (e.g., it preserves information perfectly if the input is well-formed). You can define them by placing them on your Encoding Object Type's definition:

```
struct any_unicode_byte_encoding {
        using is_decode_injective = std::true_type;
        using is_encode_injective = std::true_type;
        using code_unit = std::byte;
        using code_point = ztd::text::unicode_scalar_value;
        // ...
};
```

This signals that the `encode_one` and `decode_one` functions — if they are given well-formed input — will never be lossy between their `code_point` type and their `code_unit` types when performing the desired operation. If only one half of that equation is lossy, then you can mark only one, or the other. For example, *ztd::text::ascii* is lossy only in for the `encode_one` operation, so it has `is_decode_injective = std::true_type;` for `decode` operations, but `is_encode_injective = std::false_type;` for `encode` operations:

```
        //////
        /// @brief The individual units that result from an encode operation or are used␣
↪as input to a decode
        /// operation.
        /// @remarks ASCII can decode from its 7-bit (unpacked) code units to Unicode␣
↪Code Points. Since the converion
        /// is lossless, this property is true.
        //////
        using is_decode_injective = ::std::true_type;
        //////
        /// @brief Whether or not the encode operation can process all forms of input␣
↪into code unit values. This is
        /// not true for ASCII, as many Unicode Code Point and Unicode Scalar Values␣
↪cannot be represented in ASCII.
        /// Since the conversion is lossy, this property is false.
        //////
        using is_encode_injective = ::std::false_type;
        //////
        /// @brief The maximum code units a single complete operation of encoding can␣
↪produce.
```

(continues on next page)

(continued from previous page)

```
        inline static constexpr const ::std::size_t max_code_units = 1;
                                                    ::std::move(__inlast)),
```

If the type definition is not present and is not `std::true_type`, then the implementation assumes that this is false for a given encoding. See *ztd::text::is_decode_injective* and *ztd::text::is_encode_injective* for more information.

Replacement Characters

Replacement characters are a way to communicate to the end-user that something went wrong, without having to throw an exception that may stop the world or stop the encoding/decoding process altogether. The default error handler for text (*ztd::text::default_handler*, unless *configured* otherwise) provides room for you to provide your own encoding types, and it does so in two ways that is recognized by the library:

Always Has A Replacement

If your type always has a replacement character, regardless of the situation, it can signal this by writing one of two functions:

- `replacement_code_units()` (for any failed `encode` step)
- `replacement_code_points()` (for any failed `decode` step)

These functions return a contiguous range of either `code_units` or `code_points`, typically a `std::span<const code_unit>` or a `std::span<const code_point>`.

```
class runtime_locale {
public:
        ztd::span<const code_unit> replacement_code_units() const noexcept {
                if (this->contains_unicode_encoding()) {
                        // Probably CESU-8 or UTF-8!
                        static const char replacement[3]
                                = { '\xEF', '\xBF', '\xBD' };
                        return replacement;
                }
                else {
                        // Uh... well, it probably has this? ¯\_()_/¯
                        static const char replacement[1] = { '?' };
                        return replacement;
                }
        }

        (void)argc;
```

If the given replacement range is empty, then nothing is inserted at all (as this is a deliberate choice from the user. See the next section for how to have this function but graciously return "no replacements" for given runtime conditions).

This is employed, for example, in the *ztd::text::ascii* encoding, which uses a *'?'* as its replacement `code_unit` and `code_point` value.

Maybe Has A Replacement

If your type might not have a range of replacement characters but you will not know that until run time, regardless of the situation, the encoding type can signal this by writing different functions:

- `maybe_replacement_code_units()` (for any failed `encode` step)
- `maybe_replacement_code_points()` (for any failed `decode` step)

These functions return a `std::optional` of a contiguous range of either `code_units` or `code_points`, typically a `std::optional<std::span<const code_unit>>` or a `std::optional<std::span<const code_point>>`. If the `optional` is not engaged (it does not have a value stored), then the replacement algorithm uses its default logic to insert a replacement character, if possible. Otherwise, if it does have a value, it uses that range. If it has a value but the range is empty, it uses that empty range (and inserts nothing).

This is useful for encodings which provide runtime-erased wrappers or that wrap platform APIs like Win32, whose CPINFOEXW structure contains both a `WCHAR UnicodeDefaultChar;` and a `BYTE DefaultChar[MAX_DEFAULTCHAR];`. These can be provided as the range values after being stored on the encoding, or similar.

The Default

When none of the above can happen, the *ztd::text::replacement_handler_t* will attempt to insert a Unicode Replacement Character (, `U'\uFFFD'`) or the *'?'* character into the stream, in various ways. See *ztd::text::replacement_handler_t* for more details on that process!

Marking an encoding as Unicode-Capable

Sometimes, you need to make your own encodings. Whether for legacy reasons or for interoperation reasons, you need the ability to write an encoding that can losslessly handle all 2^21 code points. Whether it's writing a variant of UTF-7, or dealing with a very specific legacy set like Unicode v6.0 with the Softbank Private Use Area, you are going to need to be able to say "hey, my encoding can handle all of the code points and therefore deserves to be treated like a Unicode encoding". There are 2 ways to do this, one for decisions that can be made at compile time, and one for decisions that can be made at runtime (e.g., over a `variant_encoding<X, Y, Z>`).

compile time

The cheapest way to tag an encoding as Unicode Capable and have the library recognize it as such when *ztd::text::is_unicode_encoding* is used is to just define a member type definition:

```
class utf8_v6_softbank {
public:
        // ...
        using is_unicode_encoding = std::true_type;
        // ...
};
```

That is all you have to write. Both *ztd::text::is_unicode_encoding* and *ztd::text::contains_unicode_encoding* will detect this and use it.

Run-time

If your encoding cannot know at compile time whether or not it is a unicode encoding (e.g., for type-erased encodings, complex wrapping encodings, or encodings which rely on external operating system resources), you can define a method instead. When applicable, this will be picked up by the *ztd::text::contains_unicode_encoding* function. Here is an example of a runtime, locale-based encoding using platform-knowledge to pick up what the encoding might be, and determine if it can handle working in Unicode:

```
#endif


        struct encode_state {
                std::mbstate_t c_stdlib_state;

                encode_state() noexcept : c_stdlib_state() {
                        // properly set for c32rtomb state
                        code_unit ghost_ouput[MB_LEN_MAX] {};
                        UCHAR_ACCESS c32rtomb(ghost_ouput, U'\0', &c_stdlib_state);
                }
        };

        bool contains_unicode_encoding() const noexcept {
#if defined(_WIN32)
                CPINFOEXW cp_info {};
                BOOL success = GetCPInfoExW(CP_THREAD_ACP, 0, &cp_info);
                if (success == 0) {
                        return false;
                }
                switch (cp_info.CodePage) {
                case 65001: // UTF-8
                            // etc. etc. ...
                        return true;
                default:
                        break;
                }
                                        empty_code_point_span(), empty_code_unit_span());
```

That is it. *ztd::text::contains_unicode_encoding* will detect this and use your function call, so you should never be calling this or accessing the above compile time classification if necessary and always delegating to the `ztd::text::contains_unicode_encoding` function call.

Encoding-Dependent States

Some states need additional information in order to be constructed and used properly. This can be the case when the encoding has stored some type-erased information, as *ztd::text::any_encoding* does, or as if you wrote a `variant_encoding<utf8le, utf16be, ...>`. For example, given a `type_erased_encoding` like so:

```
class type_erased_encoding {
private:
        struct erased_state {
                virtual ~erased_state () {}
        };

```

(continues on next page)

(continued from previous page)

```
	struct erased_encoding {
		virtual std::unique_ptr<erased_state> create_decode_state() = 0;
		virtual std::unique_ptr<erased_state> create_encode_state() = 0;

		virtual ~erased_encoding () {}
	};

	template <typename Encoding>
	struct typed_encoding : erased_encoding {
		Encoding encoding;

		struct decode_state : erased_state {
			using state_type = ztd::text::decode_state_t<Encoding>;
			state_type state;

			decode_state(const Encoding& some_encoding)
			: state(ztd::text::make_decode_state(some_encoding)) {
				// get a decode state from the given encoding
			}
		};

		struct encode_state : erased_state {
			using state_type = ztd::text::encode_state_t<Encoding>;
			state_type state;

			decode_state(const Encoding& some_encoding)
			: state(ztd::text::make_encode_state(some_encoding)) {
				// get a decode state from the given encoding
			}
		};

		typed_encoding(Encoding&& some_encoding)
		: encoding(std::move(some_encoding)) {
			// move encoding in
		}

		typed_encoding(const Encoding& some_encoding)
		: encoding(some_encoding) {
			// copy encoding in
		}

		virtual std::unique_ptr<erased_state> create_decode_state() override {
			return std::make_unique<decode_state>(encoding);
		}

		virtual std::unique_ptr<erased_state> create_encode_state() override {
			return std::make_unique<encode_state>(encoding);
		}
	};

	std::unique_ptr<erased_encoding> stored;

```

(continues on next page)

(continued from previous page)

```
public:
        template <typename AnyEncoding>
        type_erased(AnyEncoding&& some_encoding)
        : stored_ptr(std::make_unique<typed_encoding<std::remove_cvref_t<AnyEncoding>>>(
                std::forward<AnyEncoding>(some_encoding))
        ) {
                // store any encoding in the member unique pointer
        }

        // ... rest of the implementation
};
```

We can see that creating a state with a default constructor no longer works, because the state itself requires more information than can be known by just the constructor itself. It needs access to the wrapped encoding. The solution to this problem is an opt-in when creating your state types by giving your *state* type a constructor that takes the encoding type:

```
class type_erased_encoding {
        // from above, etc. ...
public:
        // public-facing wrappers
        struct type_erased_decode_state {
        public:
                // special constructor!!
                type_erased_state (const type_erased_encoding& encoding)
                : stored(encoding.stored->create_decode_state()) {

                }
        private:
                std::unique_ptr<erased_state> stored;
        };

        struct type_erased_encode_state {
        public:
                // special constructor!!
                type_erased_state (const type_erased_encoding& encoding)
                : stored(encoding.stored->create_encode_state()) {
                        // hold onto type-erased state
                }
        private:
                std::unique_ptr<erased_state> stored;
        };

        using decode_state = type_erased_state;
        using encode_state = type_erased_state;

        // ... rest of the Lucky 7 members
};
```

These special constructors will create the necessary state using information from the `type_erased_encoding` to do it properly. This will allow us to have states that properly reflect what was erased when we perform a given higher-level *conversion operation or algorithm*.

This encoding-aware state-construction behavior is detected by the *ztd::text::is_state_independent*, *ztd::text::is_decode_state_independent*, and *ztd::text::is_encode_state_independent* classifications.

These classifications are used in the *ztd::text::make_decode_state* and *ztd::text::make_encode_state* function calls to correctly construct a state object, which is what the API uses to make states for its higher-level function calls. If you are working in a generic context, you should use these functions too when working in this minute details. However, if you're not working with templates, consider simply using the already-provided *ztd::text::any_encoding* to do exactly what this example shows, with some extra attention to detail and internal optimizations done on your behalf.

Need for Speed: Extension Points

The core encoding/decoding loops and the *Lucky 7 design*, while flexible, can come with performance degradation due to its one-by-one nature. There are many well-researched speedups to validating, counting, and converting UTF and other kinds of text. In order to accommodate these, ztd.text has a number of places to overload the core behavior by way of named Argument Dependent Lookup (ADL or Koenig Lookup, named after Andrew Koenig) functions that serve as extension points. They are listed, with their expected argument forms / counts, here.

Extension points: Arguments

For all extension points, arguments are given based on what was input to one of the original higher-level functions. They have these forms and general requimrents:

- `tag` - The first argument to every extension point that takes a single encoding. The `tag` type is *ztd::text::tag<decltype(encoding)>* with any `const`, `volatile`, or references (& and &&) removed from the `decltype` of the `encoding`.
- `duo_tag` - The first argument to every extension point that takes 2 encodings. The `tag` type is *ztd::text::tag<decltype(from_encoding), decltype(to_encoding)>* with any `const`, `volatile`, or references (& and &&) removed from the `decltype` of the two `encoding`s.
- `encoding` - The encoding used to perform the operation. Can be prefixed with `from_` or `to_` in the argument list to show it is one of two encodings used to perform e.g. a transcode operation.
- `input` - The input range. Can be of any type. Most encodings should at the very least handle basic iterator-iterator pairs correctly. These are allowed to have `const`-correct iterators that produce `const`-correct references, so never assume you can write to the input, and appropriately `const`-qualify any `std::span`s you use.
- `output` - The output range. Can be of any output range type, such as a `unbounded_view<>` with a `back_inserter` or a `std::span` for direct memory writes. The types only requirement is that you can write to it by getting an iterator from `begin(...)`, and calling `*it = value;`.
- `handler` - The error handler used to perform the operation. Can be prefixed with `from_` or `to_` in the argument list to show it is one of two error handlers used to perform e.g. a transcode operation.
- `state` - The state objects used to perform the operation. States are always passed by non-`const`, l-value reference. Can be prefixed with `from_` or `to_` in the argument list to show it is one of two states associated with an encoding with the same prefix.

Extension Points: Forms & Return Types

Overriding any one of these extension points allows you to hook that behavior. It is very much required that you either use concrete types to provide these ADL extension points, or heavily constrain them using SFINAE (preferred for C++17 and below) or Concepts (only C++20 and above).

text_decode

Form: `text_decode(tag, input, encoding, output, handler, state)`.

An extension point to speed up decoding operations for a given encoding, its input and outpuut ranges, and the associated error handler and state. This can be helpful for encodings which *may need to hide certain parts of their state*.

Must return a *ztd::text::decode_result*.

text_encode

Form: `text_encode(input, encoding, output, handler, state)`.

An extension point to speed up encoding operations for a given encoding, its input and outpuut ranges, and the associated error handler and state. This can be helpful for encodings which *may need to hide certain parts of their state*.

Must return a *ztd::text::encode_result*.

text_transcode

Form: `text_transcode(input, from_encoding, output, to_encoding, from_handler, to_handler, from_state, to_state)`

An extension point to speed up transcoding in bulk, for a given encoding pair, its input and output ranges, and its error handlers and states. Useful for known encoding pairs that have faster conversion paths between them.

Must return a *ztd::text::transcode_result*.

text_transcode_one

Form: `text_transcode_one(input, from_encoding, output, to_encoding, from_handler, to_handler, from_state, to_state)`

An extension point to provide faster one-by-one encoding transformations for a given encoding pair, its input and output ranges, and its error handlers and states. This is **not** a bulk extension point conversion. It is used in the *ztd::text::transcode_view* type to increase the speed of iteration, where possible.

Must return a *ztd::text::transcode_result*.

text_validate_encodable_as_one

Form: `text_validate_encodable_as_one(input, encoding, state)`

An extension point to provide faster one-by-one validation. Provides a shortcut to not needing to perform both a `decode_one` and an `encode_one` step during the basic validation loop.

Must return a *ztd::text::validate_result*.

text_validate_decodable_as_one

Form: `text_validate_decodable_as_one(input, encoding, state)`

An extension point to provide faster one-by-one validation. Provides a shortcut to not needing to perform both a `encode_one` and an `decode_one` step during the basic validation loop.

Must return a *ztd::text::validate_result*.

text_validate_transcodable_as_one

Form: `text_validate_decodable_as_one(input, from_encoding, to_encoding, decode_state, encode_state)`

An extension point to provide faster one-by-one validation. Provides a shortcut to not needing to perform both a `encode_one` and an `decode_one` step during the basic validation loop.

Must return a *ztd::text::validate_transcode_result*.

text_validate_encodable_as

Form: `text_validate_encodable_as(input, encoding, state)`

An extension point to provide faster bulk code point validation. There are many tricks to speed up validationg of text using bit twiddling of the input sequence and more.

Must return a *ztd::text::validate_result*.

text_validate_decodable_as

Form: `text_validate_decodable_as(input, encoding, state)`

An extension point to provide faster bulk code unit validation. There are many tricks to speed up validationg of text using bit twiddling of the input sequence and more.

Must return a *ztd::text::validate_result*.

text_count_as_encoded_one

Form: `text_count_as_encoded_one(input, encoding, handler, state)`

An extension point to provide faster one-by-one counting. Computation cycles can be saved by only needing to check a subset of things. For example, specific code point ranges can be used to get a count for UTF-16 faster than by encoding into an empty buffer.

Must return a *ztd::text::count_result*.

text_count_as_decoded_one

Form: `text_count_as_decoded_one(input, encoding, handler, state)`

An extension point to provide faster one-by-one counting. Computation cycles can be saved by only needing to check a subset of things. For example, the leading byte in UTF-8 can provide an immediate count for how many trailing bytes, leading to a faster counting algorithm.

Must return a *ztd::text::count_result*.

text_count_as_encoded

Form: `text_count_as_encoded(input, encoding, handler, state)`

An extension point for faster bulk code point validation.

Must return a *ztd::text::count_result*.

text_count_as_decoded

Form: `text_count_as_decoded(input, encoding, handler, state)`

An extension point for faster bulk code point validation.

Must return a *ztd::text::count_result*.

That's All of Them

Each of these extension points are important to one person, or another. For example, Daniel Lemire spends a lot of time optimizing *UTF-8 routines for fast validation* or *Fast Deterministic Finite Automata (DFA) decoding of UTF-8 and more*. There are many more sped up counting, validating, encoding, and decoding routines: therefore it is critical that any library writer or application developer can produce those for their encodings and, on occassion, override the base behavior and implementation-defined internal speed up written by ztd.text itself.

1.6 Available Encodings

Below is a table of encodings. Here, we track which encodings can be represented using the *Lucky 7* technique, whether or not we have produced such an implementation, and (if applicable) a link to said implementation.

As a general point, we hope to support almost all of the encodings here in one form or another! If you'd like to request prioritization of a certain encoding, let us know.

Table 1: **Encoding Progress Table**

Name	Stateful	*Implementable?*	Implemented
UTF-8	No	Yes	*Yes*
UTF-16	No	Yes	*Yes*
UTF-32	No	Yes	*Yes*
Modified UTF-8 (MUTF-8)	No	Yes	*Yes*
Wobbly Transformation Format-8 (WTF-8)	No	Yes	*Yes*
ASCII	No	Yes	*Yes*
C Locale	Yes (`std::mbstate_t`)	Yes	*Yes*
C Locale, Wide	Yes (`std::mbstate_t`)	Yes	*Yes*
String Literials	Compiler-Dependent	Yes	*Yes*
Wide String Literals	Compiler-Dependent	Yes	*Yes*
"Anything" Wrapper	Typed-Erased	Yes	*Yes*
Encoding Scheme	Wrapping-Dependent	Yes	*Yes*
`iconv` Encoding	Yes	Yes	*WIP*
`cuneicode` Encoding	Yes	Yes	*WIP*
UTF-EBCDIC	No	Yes	No
UTF-7	Yes	Yes	No
UTF-7-IMAP	Yes	Yes	No
ISO-8859-1	Unresearched	Unconfirmed	No
ISO-8859-2	Unresearched	Unconfirmed	No
ISO-8859-3	Unresearched	Unconfirmed	No
ISO-8859-4	Unresearched	Unconfirmed	No
ISO-8859-5	Unresearched	Unconfirmed	No
ISO-8859-6	Unresearched	Unconfirmed	No
ISO-8859-7	Unresearched	Unconfirmed	No
ISO-8859-8	Unresearched	Unconfirmed	No
ISO-8859-9	Unresearched	Unconfirmed	No
ISO-8859-10	Unresearched	Unconfirmed	No
ISO-8859-13	Unresearched	Unconfirmed	No
ISO-8859-14	Unresearched	Unconfirmed	No
ISO-8859-15	Unresearched	Unconfirmed	No
ISO-8859-16	Unresearched	Unconfirmed	No
KOI8-R	Unresearched	Unconfirmed	No
KOI8-U	Unresearched	Unconfirmed	No
KOI8-RU	Unresearched	Unconfirmed	No
KOI8-T	Unresearched	Unconfirmed	No
ATARIST	Unresearched	Unconfirmed	No
RISCOS-LATIN1	Unresearched	Unconfirmed	No

continues on next page

Table 1 – continued from previous page

Name	Stateful	*Implementable?*	Implemented
TDS565	Unresearched	Unconfirmed	No
CP437	Unresearched	Unconfirmed	No
CP737	Unresearched	Unconfirmed	No
CP775	Unresearched	Unconfirmed	No
CP850	Unresearched	Unconfirmed	No
CP852	Unresearched	Unconfirmed	No
CP853	Unresearched	Unconfirmed	No
CP855	Unresearched	Unconfirmed	No
CP857	Unresearched	Unconfirmed	No
CP858	Unresearched	Unconfirmed	No
CP860	Unresearched	Unconfirmed	No
CP861	Unresearched	Unconfirmed	No
CP862	Unresearched	Unconfirmed	No
CP863	Unresearched	Unconfirmed	No
CP864	Unresearched	Unconfirmed	No
CP865	Unresearched	Unconfirmed	No
CP866	Unresearched	Unconfirmed	No
CP869 (Nice)	Unresearched	Unconfirmed	No
CP874	Unresearched	Unconfirmed	No
CP932	Unresearched	Unconfirmed	No
CP936	Unresearched	Unconfirmed	No
CP949	Unresearched	Unconfirmed	No
CP1125	Unresearched	Unconfirmed	No
CP1131	Unresearched	Unconfirmed	No
CP1133	Unresearched	Unconfirmed	No
CP1250	Unresearched	Unconfirmed	No
CP1251	Unresearched	Unconfirmed	No
CP1252 (Latin-1)	Unresearched	Unconfirmed	No
CP1253	Unresearched	Unconfirmed	No
CP1254	Unresearched	Unconfirmed	No
CP1255	Unresearched	Unconfirmed	No
CP1256	Unresearched	Unconfirmed	No
CP1257	Unresearched	Unconfirmed	No
CP1258	Unresearched	Unconfirmed	No
MacRoman	Unresearched	Unconfirmed	No
MacCentralEurope	Unresearched	Unconfirmed	No
MacIceland	Unresearched	Unconfirmed	No
MacCroatian	Unresearched	Unconfirmed	No
MacRomania	Unresearched	Unconfirmed	No
MacCryllic	Unresearched	Unconfirmed	No
MacUkraine	Unresearched	Unconfirmed	No
MacGreek	Unresearched	Unconfirmed	No
MacTurkish	Unresearched	Unconfirmed	No
Macintosh	Unresearched	Unconfirmed	No
MacHebrew	Unresearched	Unconfirmed	No
MacArabic	Unresearched	Unconfirmed	No
MacThai	Unresearched	Unconfirmed	No
SHIFT-JIS	Yes, shift states	Yes	No
SHIFT-JISX0213	Yes, shift states	Yes	No

continues on next page

Table 1 – continued from previous page

Name	Stateful	*Implementable?*	Implemented
JOHAB	Unresearched	Unconfirmed	No
GB18030	No (It's a Unicode Encoding)	Yes	No
TACE	No	Yes	No
TSCII	No	Yes	No
EUC-JP	Unresearched	Unconfirmed	No
EUC-JISX0213	Unresearched	Unconfirmed	No
EUC-CN	Unresearched	Unconfirmed	No
EUC-KR	Unresearched	Unconfirmed	No
EUC-TW	Unresearched	Unconfirmed	No
BIG5	Unresearched	Unconfirmed	No
BIG5-2003	Unresearched	Unconfirmed	No
Big5-HKSCS	Unresearched	Unconfirmed	No
Big5-HKSCS:1999	Unresearched	Unconfirmed	No
Big5-HKSCS:2001	Unresearched	Unconfirmed	No
Big5-HKSCS:2004	Unresearched	Unconfirmed	No
GBK	Unresearched	Unconfirmed	No
HZ	Unresearched	Unconfirmed	No
ISO-2022-JP	Unresearched	Unconfirmed	No
ISO-2022-JP-2	Unresearched	Unconfirmed	No
ISO-2022-JP-1	Unresearched	Unconfirmed	No
ISO-2022-JP-3	Unresearched	Unconfirmed	No
ISO-2022-JP-MS	Unresearched	Unconfirmed	No
ISO-2022-CN	Unresearched	Unconfirmed	No
ISO-2022-CN-EXT	Unresearched	Unconfirmed	No
ISO-2022-KR	Unresearched	Unconfirmed	No
VISCII	Unresearched	Unconfirmed	No
ARMSCII-8	Unresearched	Unconfirmed	No
TCVN	Unresearched	Unconfirmed	No
PT154	Unresearched	Unconfirmed	No
RK1048	Unresearched	Unconfirmed	No
RK1048	Unresearched	Unconfirmed	No
TIS-620	Unresearched	Unconfirmed	No
MuleLao-1	Unresearched	Unconfirmed	No
HP-ROMAN8	Unresearched	Unconfirmed	No
NEXTSTEP	Unresearched	Unconfirmed	No
Georgian-Academy	Unresearched	Unconfirmed	No
Georgian-PS	Unresearched	Unconfirmed	No

If you know of an encoding not listed here, let us know in the issue tracker!

1.7 Known Unicode Encodings

Out of all the encodings listed on the *encodings page*, only a handful are known to be Unicode Encodings. These are as follows:

- UTF-7
- UTF-7-IMAP
- UTF-8
- UTF-16 (All Endiannesses)
- UTF-32 (All Endiannesses)
- GB18030
- CESU-8
- MUTF-8
- WTF-8
- UTF-1
- UTF-EBCDIC

When the encoding is known at compile time (e.g., it is just a plain object), it contains a `is_unicode_encoding` type member that is set to `std::true_type`. Otherwise, it is left off. This is detected by *ztd::text::contains_unicode_encoding* and *ztd::text::is_unicode_encoding*.

If you know of any others, please let us know!

1.8 Configuring the Library

There are various configuration macros and CMake/build-time switches that will change the behavior of the library or attempt to use different

- **`ZTD_TEXT_USE_CUNEICODE` (CMake: `ZTD_TEXT_USE_CUNEICODE`)**
  - Enables use of the Cuneicode project, which has low-level C routines for converting from one encoding form to another encoding form. Only includes a limited number of conversions, specifically from and to the execution encoding to other encodings.
  - Makes the `ztd::text::cnc_encoding` available (accessible directly VIA `#include <ztd/text/cnc_encoding.hpp>`)
  - Default: off.
  - Turned on if the special `__has_include` directive is present with the compiler and `__has_include(<ztd/cuneicode/cuneicode.h>)` works.
- **`ZTD_TEXT_USE_ICONV` (CMake: `ZTD_TEXT_USE_ICONV`)**
  - Enables use of the iconv project.
  - Attempts to load it from the system at runtime using `GetProcAddress`, `dlopen/dlsym/dlclose`.
  - Makes the `ztd::text::iconv_encoding` available (accessible directly VIA `#include <ztd/text/iconv_encoding.hpp>`).
  - Default: off.
  - Not turned on by-default under any conditions.

- `ZTD_TEXT_UNICODE_CODE_POINT_DISTINCT_TYPE`
  - Turns `ztd::text::unicode_code_point` from a type definition to `char32_t` to an implementation-defined class type which enforces the various invariants of being a *unicode code point*.
  - Default: off.
  - Not turned on by-default under any conditions.
- `ZTD_TEXT_UNICODE_CODE_POINT_INVARIANT_ABORT`
  - If `ztd::text::unicode_code_point` is a distinct class (as controlled by *ZTD_TEXT_UNICODE_CODE_POINT_DISTINCT_TYPE*), each construction of a *unicode_code_point* object that violates the required invariants of a *unicode code point* will trigger an abort.
  - It is normally a `ZTD_TEXT_ASSERT(...)` or equivalent.
  - Default: off.
  - Not turned on by-default under any conditions.
- `ZTD_TEXT_DEFAULT_HANDLER_THROWS`
  - Makes the `ztd::text::default_handler` into a throwing handler rather than a replacement character handler.
  - This is not at all recommended since malformed text (or text assumed to be the wrong encoding) is common, and not properly handling a thrown exception can result in what is, effectively, a denial-of-service attack for things which need to continually handle untrusted input.
  - Default: off.
  - Not turned on by-default under any conditions.
- `ZTD_TEXT_UNICODE_SCALAR_VALUE_DISTINCT_TYPE`
  - Turns `ztd::text::unicode_scalar_value` from a type definition to `char32_t` to an implemenation-defined class type which enforces the various invariants of being a *unicode scalar value*.
  - Default: **on**.
  - Not turned off by-default under any conditions.
- `ZTD_TEXT_UNICODE_SCALAR_VALUE_INVARIANT_ABORT`
  - If `ztd::text::unicode_scalar_value` is a distinct class (as controlled by *ZTD_TEXT_UNICODE_SCALAR_VALUE_DISTINCT_TYPE*), each construction of a *unicode_scalar_value* object that violates the required invariants of a *unicode scalar value* will trigger an abort.
  - It is normally a `ZTD_TEXT_ASSERT(...)` or equivalent.
  - Default: off.
  - Not turned on by-default under any conditions.
- `ZTD_CXX_COMPILE_TIME_ENCODING_NAME`
  - Gives the *ztd::text::literal encoding* a name that matches what the encoding of string literals ("") are.
  - Overrides any of library's heuristics and shenanigans to determine the string literal encoding.
  - If this does not match the actual string literal encoding, it can cause Undefined Behavior.

  - Default: A complex set of platform checks. See *ztd::text::literal encoding* for more details.
  - Not turned on normally under any circumstances.

- **ZTD_TEXT_YES_PLEASE_DESTROY_MY_LITERALS_UTTERLY_I_MEAN_IT**
  - Enables ignoring the fact that the string literal ("") encoding cannot be determined/discovered on the given platform for the *ztd::text::literal encoding*.
  - Will cause Undefined Behavior if a string literal or wide string literal is encoded or decoded to/from and the encoding does not match whatever pot-shot guess the system takes.
  - Default: off.
  - Not turned on by-default under any conditions.
  - **Please don't use this unless you have some really, really weird setup that requires messing everything up...**

- **ZTD_CXX_COMPILE_TIME_WIDE_ENCODING_NAME**
  - Gives the *ztd::text::wide_literal encoding* a name that matches what the encoding of wide string literals (L"") are.
  - Overrides any of library's heuristics and shenanigans to determine the wide string literal encoding.
  - If this does not match the actual wide string literal encoding, it can cause Undefined Behavior.
  - Default: A complex set of platform checks. See *ztd::text::wide_literal encoding* for more details.
  - Not turned on normally under any circumstances.

- **ZTD_TEXT_YES_PLEASE_DESTROY_MY_WIDE_LITERALS_UTTERLY_I_MEAN_IT**
  - Enables ignoring the fact that the wide string literal (L"") encoding cannot be determined/discovered on the given platform for the *ztd::text::wide_literal encoding*.
  - Will cause Undefined Behavior if a string literal or wide string literal is encoded or decoded to/from and the encoding does not match whatever pot-shot guess the system takes.
  - Default: off.
  - Not turned on by-default under any conditions.
  - **Please don't use this unless you have some really, really weird setup that requires messing everything up...**

- **ZTD_TEXT_INTERMEDIATE_DECODE_BUFFER_BYTE_SIZE**
  - Changes the default intermediate buffer size placed on the stack for temporary operations.
  - Default: a series of compile time platform checking heuristics to determine a generally useful buffer size that will not overflow the stack.
  - Not turned on by default under any conditions.
  - Specify a numeric value for ZTD_TEXT_INTERMEDIATE_DECODE_BUFFER_BYTE_SIZE to have it used instead.
  - Will always be used as the input to a function determining the maximum between this type and a buffer size consistent with *ztd::text::max_code_points_v* or *ztd::text::max_code_points_v*.

- **ZTD_TEXT_INTERMEDIATE_ENCODE_BUFFER_BYTE_SIZE**
  - Changes the default intermediate buffer size placed on the stack for temporary operations.
  - Default: a series of compile time platform checking heuristics to determine a generally useful buffer size that will not overflow the stack.

  - Not turned on by default under any conditions.
  - Specify a numeric value for `ZTD_TEXT_INTERMEDIATE_ENCODE_BUFFER_BYTE_SIZE` to have it used instead.
  - Will always be used as the input to a function determining the maximum between this type and a buffer size consistent with *ztd::text::max_code_points_v* or *ztd::text::max_code_points_v*.

- **`ZTD_TEXT_INTERMEDIATE_TRANSCODE_BUFFER_BYTE_SIZE`**
  - Changes the default intermediate buffer size placed on the stack for temporary operations.
  - Default: a series of compile time platform checking heuristics to determine a generally useful buffer size that will not overflow the stack.
  - Not turned on by default under any conditions.
  - Specify a numeric value for `ZTD_TEXT_INTERMEDIATE_TRANSCODE_BUFFER_BYTE_SIZE` to have it used instead.
  - Will always be used as the input to a function determining the maximum between this type and a buffer size consistent with *ztd::text::max_code_points_v* or *ztd::text::max_code_points_v*.

1.9 API Reference

This is simply a listing of all the available pages containing various APIs, or links to pages that link to API documentation.

1.9.1 Containers

basic_text (In Progress)

Warning: This isn't finished yet! Come check back by the next major or minor version update.

The `basic_text` class provides functionality similar to `std::string` but for performing it on encoded, normalized text.

template<typename **_Encoding**, typename **_NormalizationForm** = nfkc, typename **_Container** = ::std::basic_string<*code_unit_t*<*_Encoding*>>, typename **_ErrorHandler** = *default_handler_t*>
class `ztd::text::`**basic_text** : private *basic_text_view*<*_Encoding*, *_NormalizationForm*, *_Container*, *_ErrorHandler*>

A wrapper (container adapter) that takes the given `_Encoding` type and `_NormalizationForm` type and imposes it over the given chosen `_Container` storage for the purposes of allowing users to examine the text.

tparam _Encoding The encoding to store any input and presented text as.

tparam _NormalizationForm The normalization form to impose on the stored text's sequences.

tparam _Container The container type that will be stored within this *ztd::text::basic_text* using the code units from the `_Encoding` type.

tparam _ErrorHandler The default error handler to use for any and all operations on text. Generally, most operations will provide room to override this.

Public Types

using **range_type** = typename __base_t::range_type
The type that this view is wrapping.

using **encoding_type** = typename __base_t::encoding_type
The encoding type that this view is using to interpret the underlying sequence of code units.

using **state_type** = typename __base_t::state_type
The encoding type that this view is using to interpret the underlying sequence of code units.

using **normalization_type** = typename __base_t::normalization_type
The normalization form type this view is imposing on top of the encoded sequence.

using **error_handler_type** = typename __base_t::error_handler_type
The error handling type used by default for any problems in conversions.

Private Functions

inline constexpr _CodePointView<_ViewErrorHandler> **code_points**(*state_type* __state, _ViewErrorHandler &&__error_handler) const noexcept
Returns a view over the code points of this type, decoding "on the fly"/"lazily".

Remark Moves the provided `__state` in as the "starting point".

Template Parameters **_ViewErrorHandler** – The type of the passed-in error handler to use for these operations.

Parameters

- **__state** – **[in]** The state to use for this code point view.
- **__error_handler** – **[in]** The error handler to look at the code points for this code point view.

inline constexpr _CodePointView **code_points**(*state_type* __state) const noexcept
Returns a view over the code points of this type, decoding "on the fly"/"lazily".

Remark Moves the provided `__state` in as the "starting point".

Parameters **__state** – **[in]** The state to use for this code point view.

inline constexpr _CodePointView **code_points**() const noexcept
Returns a view over the code points of this type, decoding "on the fly"/"lazily".

Remark Copies the stored `state` within the *ztd::text::basic_text_view* to perform the code point iteration process.

inline constexpr *range_type* &&**base**() && noexcept
Access the storage as an r-value reference.

inline constexpr const *range_type* &**base**() const & noexcept
Access the storage as a const-qualified l-value reference.

inline constexpr *range_type* &**base**() & noexcept
Access the storage as an l-value reference.

using ztd::text::**text** = *basic_text*<*execution_t*>
A container for storing text in the locale, runtime-based encoding.

using ztd::text::**wtext** = *basic_text*<*wide_execution_t*>
A container for storing text in the locale, runtime-based wide encoding.

using ztd::text::**ltext** = *basic_text*<*literal_t*>
A container for storing text in the string *literal_t* encoding.

using ztd::text::**wltext** = *basic_text*<*wide_literal_t*>
A container for storing text in the wide string *literal_t* encoding.

using ztd::text::**u8text** = *basic_text*<*utf8_t*>
A container for storing text in the UTF-8 encoding.

using ztd::text::**u16text** = *basic_text*<*utf16_t*>
A container for storing text in the UTF-16 encoding.

using ztd::text::**u32text** = *basic_text*<*utf32_t*>
A container for storing text in the UTF-32 encoding.

1.9.2 Views

basic_text_view (In Progress)

Warning: This isn't finished yet! Come check back by the next major or minor version update.

The basic_text_view class provides a one-by-one view of the stored range's code points and other functionality in a more complete form that goes beyond just code point iteration or code unit iteration like *ztd::text::decode_view* or *ztd::text::encode_view*.

template<typename **_Encoding**, typename **_NormalizationForm** = nfkc, typename **_Range** = ::std::basic_string_view<*code_unit_t*<*_Encoding*>>, typename **_ErrorHandler** = *default_handler_t*>
class ztd::text::**basic_text_view**
A view over a sequence of code units. The code units are expected to be of the given encoding and normalization form.

Remark The default type for this is a basic_string_view templated on the code unit type from the encoding. The error handler is also the default careless error handler, meaning that any lossy conversions will automatically cause a compile time error.

tparam _Encoding The encoding to store any input and presented text as.

tparam _NormalizationForm The normalization form to impose on the stored text's sequences.

tparam _Range The range type that will be stored within this *ztd::text::basic_text_view* and examined using the iterators, following the `_Encoding` type decoding procedure.

tparam _ErrorHandler The default error handler to use for any and all operations on text. Generally, most operations will provide room to override this.

Public Types

using **range_type** = *_Range*
The type that this view is wrapping.

using **encoding_type** = *_Encoding*
The encoding type that this view is using to interpret the underlying sequence of code units.

using **state_type** = *encode_state_t*<*_Encoding*>
The encoding type that this view is using to interpret the underlying sequence of code units.

using **normalization_type** = *_NormalizationForm*
The normalization form type this view is imposing on top of the encoded sequence.

using **error_handler_type** = *_ErrorHandler*
The error handling type used by default for any problems in conversions.

Public Functions

template<typename **_ViewErrorHandler**>
inline constexpr _CodePointView<*_ViewErrorHandler*> **code_points**(*state_type* __state, *_ViewErrorHandler* &&__error_handler) const noexcept
Returns a view over the code points of this type, decoding "on the fly"/"lazily".

Remark Moves the provided `__state` in as the "starting point".

Template Parameters **_ViewErrorHandler** – The type of the passed-in error handler to use for these operations.

Parameters

- **__state** – **[in]** The state to use for this code point view.
- **__error_handler** – **[in]** The error handler to look at the code points for this code point view.

inline constexpr _CodePointView **code_points**(*state_type* __state) const noexcept
Returns a view over the code points of this type, decoding "on the fly"/"lazily".

Remark Moves the provided `__state` in as the "starting point".

Parameters **__state** – **[in]** The state to use for this code point view.

inline constexpr _CodePointView **code_points**() const noexcept
Returns a view over the code points of this type, decoding "on the fly"/"lazily".

Remark Copies the stored `state` within the *ztd::text::basic_text_view* to perform the code point iteration process.

inline constexpr *range_type* &&**base**() && noexcept
Access the storage as an r-value reference.

inline constexpr const *range_type* &**base**() const & noexcept
Access the storage as a const-qualified l-value reference.

inline constexpr *range_type* &**base**() & noexcept
Access the storage as an l-value reference.

using `ztd::text::`**text_view** = *basic_text_view*<*execution_t*>
A view for examining text in the locale, runtime-based encoding.

using `ztd::text::`**wtext_view** = *basic_text_view*<*wide_execution_t*>
A view for examining text in the locale, runtime-based wide encoding.

using `ztd::text::`**ltext_view** = *basic_text_view*<*literal_t*>
A view for examining text in the string *literal_t* encoding.

using `ztd::text::`**wltext_view** = *basic_text_view*<*wide_literal_t*>
A view for examining text in the wide string *literal_t* encoding.

using `ztd::text::`**u8text_view** = *basic_text_view*<*utf8_t*>
A view for examining text in the UTF-8 encoding.

using `ztd::text::`**u16text_view** = *basic_text_view*<*utf16_t*>
A view for examining text in the UTF-16 encoding.

using `ztd::text::`**u32text_view** = *basic_text_view*<*utf32_t*>
A view for examining text in the UTF-32 encoding.

decode_view

The `decode_view` class provides a one-by-one view of the stored range's code points as the desired encoding's code units. Dereferencing the iterators returns a single `code_point` value corresponding to the desired encoding's transformation of the internal code units.

The range-based classes are excellent ways to walk over units of information in a low-memory environment, as they only store the minimum amount of data necessary to perform their operations on the fly. This reduces the speed but is fine for one-at-a-time encoding operations. To decode eagerly and in bulk, see *the decode functions*.

template<typename **_Encoding**, typename **_Range** = ::std::basic_string_view<*code_unit_t*<*_Encoding*>>, typename **_ErrorHandler** = *default_handler_t*, typename **_State** = *decode_state_t*<*_Encoding*>>
class `ztd::text::`**decode_view**
A view over a range of code points, presenting the code points as code units. Uses the `_Encoding` specified to do so.

Remark The view presents code point one at a time, regardless of how many code points are output by one decode operation. This means if, for example, four (4) UTF-8 code units becomes two (2) UTF-16 code points, it will present one code point at a time. If you are looking to explicitly know what a single decode operation maps into as far as number of code points to code units (and vice-versa), you will have to use lower-level interfaces.

tparam _Encoding The encoding to read the underlying range of code points as.

tparam _Range The range of input that will be fed into the _FromEncoding's decode operation.

tparam _ErrorHandler The error handler for any encode-step failures.

tparam _State The state type to use for the encode operations to intermediate code points.

Public Types

using **iterator** = decode_iterator<*_Encoding*, _StoredRange, *_ErrorHandler*, *_State*>
The iterator type for this view.

using **sentinel** = decode_sentinel_t
The sentinel type for this view.

using **range_type** = *_Range*
The underlying range type.

using **encoding_type** = *_Encoding*
The encoding type used for transformations.

using **error_handler_type** = *_ErrorHandler*
The error handler when a decode operation fails.

using **state_type** = *decode_state_t*<*encoding_type*>
The state type used for decode operations.

Public Functions

template<typename **_ArgRange**, ::std::enable_if_t<!::std::is_same_v<remove_cvref_t<*_ArgRange*>, *decode_view*> && !::std::is_same_v<remove_cvref_t<*_ArgRange*>, *iterator*>>* = nullptr>
inline constexpr **decode_view**(*_ArgRange* &&__range) noexcept(::std::is_nothrow_constructible_v<*iterator*, *_ArgRange*>)
Constructs a *decode_view* from the underlying range.

Remark The stored encoding, error handler, and state type are default-constructed.

Parameters **__range** – **[in]** The input range to wrap and iterate over.

inline constexpr **decode_view**(*range_type* __range, *encoding_type* __encoding) noexcept(::std::is_nothrow_constructible_v<*iterator*, *range_type*, *encoding_type*>)
Constructs a *decode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__encoding** – **[in]** The encoding object to call `.decode` or equivalent functionality on.

inline constexpr **decode_view**(*range_type* __range, *encoding_type* __encoding, *error_handler_type* __error_handler) noexcept(::std::is_nothrow_constructible_v<*iterator*, *range_type*, *encoding_type*, *error_handler_type*>)

Constructs a *decode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__encoding** – **[in]** The encoding object to call `.decode` or equivalent functionality on.
- **__error_handler** – **[in]** The error handler to store in this view.

inline constexpr **decode_view**(*range_type* __range, *encoding_type* __encoding, *error_handler_type* __error_handler, *state_type* __state) noexcept(::std::is_nothrow_constructible_v<*iterator*, *range_type*, *encoding_type*, *error_handler_type*, *state_type*>)

Constructs a *decode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__encoding** – **[in]** The encoding object to call `.decode` or equivalent functionality on.
- **__error_handler** – **[in]** The error handler to store in this view.
- **__state** – **[in]** The state to user for the decode operation.

inline constexpr **decode_view**(*iterator* __it) noexcept(::std::is_nothrow_move_constructible_v<*iterator*>)

Constructs an encoding_view from one of its iterators, reconstituting the range.

Parameters **__it** – **[in]** A previously-made *decode_view* iterator.

constexpr **decode_view**() = default

Default constructor. Defaulted.

constexpr **decode_view**(const *decode_view*&) = default

Copy constructor. Defaulted.

constexpr **decode_view**(*decode_view*&&) = default

Move constructor. Defaulted.

constexpr *decode_view* &**operator=**(const *decode_view*&) = default

Copy assignment operator. Defaulted.

constexpr *decode_view* &**operator=**(*decode_view*&&) = default

Move assignment operator. Defaulted.

inline constexpr *iterator* **begin**() & noexcept

The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *iterator* **begin**() const & noexcept

The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *iterator* **begin**() && noexcept

The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *sentinel* **end**() const noexcept

The end of the range. Uses a sentinel type and not a special iterator.

encode_view

The `encode_view` class provides a one-by-one view of the stored range's code points as the desired encoding's code units. Dereferencing the iterators returns a single `code_unit` value corresponding to the desired encoding's transformation of the internal code points.

The range-based classes are excellent ways to walk over units of information in a low-memory environment, as they only store the minimum amount of data necessary to perform their operations on the fly. This reduces the speed but is fine for one-at-a-time encoding operations. To encode eagerly and in bulk, see *the encode functions*.

template<typename **_Encoding**, typename **_Range** = ::std::basic_string_view<*code_point_t*<*_Encoding*>>, typename **_ErrorHandler** = *default_handler_t*, typename **_State** = *encode_state_t*<*_Encoding*>>
class `ztd::text::`**encode_view**

A view over a range of code points, presenting the code points as code units. Uses the `_Encoding` specified to do so.

Remark The view presents code units one at a time, regardless of how many code units are output by one decode operation. This means if, for example, one (1) UTF-32 code point becomes four (4) UTF-8 code units, it will present each code unit one at a time. If you are looking to explicitly know what a single encode operation maps into as far as number of code points to code units (and vice-versa), you will have to use lower-level interfaces.

tparam _Encoding The encoding to read the underlying range of code points as.

tparam _Range The range of input that will be fed into the _FromEncoding's decode operation.

tparam _ErrorHandler The error handler for any encode-step failures.

tparam _State The state type to use for the encode operations to intermediate code points.

Public Types

using **iterator** = encode_iterator<*_Encoding*, _StoredRange, *_ErrorHandler*, *_State*>
The iterator type for this view.

using **sentinel** = encode_sentinel_t
The sentinel type for this view.

using **range_type** = *_Range*
The underlying range type.

using **encoding_type** = *_Encoding*
The encoding type used for transformations.

using **error_handler_type** = *_ErrorHandler*
The error handler when an encode operation fails.

using **state_type** = *encode_state_t*<*encoding_type*>
The state type used for encode operations.

Public Functions

constexpr **encode_view**() = default
Default constructor. Defaulted.

constexpr **encode_view**(const *encode_view*&) = default
Copy constructor. Defaulted.

constexpr **encode_view**(*encode_view*&&) = default
Move constructor. Defaulted.

template<typename **_ArgRange**, ::std::enable_if_t<!::std::is_same_v<remove_cvref_t<*_ArgRange*>, *encode_view*> && !::std::is_same_v<remove_cvref_t<*_ArgRange*>, *iterator*>>* = nullptr>
inline constexpr **encode_view**(*_ArgRange* &&__range) noexcept(::std::is_nothrow_constructible_v<*iterator*, *_ArgRange*>)
Constructs an *encode_view* from the underlying range.

Remark the stored encoding, error handler, and state type are default-constructed.

Parameters **__range** – **[in]** The input range to wrap and iterate over.

inline constexpr **encode_view**(*range_type* __range, *encoding_type* __encoding) noexcept(::std::is_nothrow_constructible_v<*iterator*, *range_type*, *encoding_type*>)
Constructs an *encode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__encoding** – **[in]** The encoding object to call `.encode` or equivalent functionality on.

inline constexpr **encode_view**(*range_type* __range, *encoding_type* __encoding, *error_handler_type* __error_handler) noexcept(::std::is_nothrow_constructible_v<*iterator*, *range_type*, *encoding_type*, *error_handler_type*>)
Constructs an *encode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__encoding** – **[in]** The encoding object to call `.encode` or equivalent functionality on.
- **__error_handler** – **[in]** A previously-made *encode_view* iterator.

inline constexpr **encode_view**(*range_type* __range, *encoding_type* __encoding, *error_handler_type* __error_handler, *state_type* __state) noexcept(::std::is_nothrow_constructible_v<*iterator*, *range_type*, *encoding_type*, *error_handler_type*, *state_type*>)
Constructs an *encode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__encoding** – **[in]** The encoding object to call `.encode` or equivalent functionality on.
- **__error_handler** – **[in]** A previously-made *encode_view* iterator.
- **__state** – **[in]** The state to user for the encode operation.

inline constexpr **encode_view**(*iterator* __it) noexcept(::std::is_nothrow_move_constructible_v<*iterator*>)
Constructs an encoding_view from one of its iterators, reconstituting the range.

Parameters **__it** – **[in]** A previously-made *encode_view* iterator.

constexpr *encode_view* &**operator=**(const *encode_view*&) = default
Copy assignment operator. Defaulted.

constexpr *encode_view* &**operator=**(*encode_view*&&) = default
Move assignment operator. Defaulted.

inline constexpr *iterator* **begin**() & noexcept
The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *iterator* **begin**() const & noexcept
The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *iterator* **begin**() && noexcept
The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *sentinel* **end**() const noexcept
The end of the range. Uses a sentinel type and not a special iterator.

transcode_view

The `transcode_view` class provides a one-by-one view of the stored range's code units as another encoding's code units. Dereferencing the iterators returns a single `code_unit` value corresponding to the desired encoding's type.

The range-based classes are excellent ways to walk over units of information in a low-memory environment, as they only store the minimum amount of data necessary to perform their operations on the fly. This reduces the speed but is fine for one-at-a-time encoding operations. To decode eagerly and in bulk, see *the transcode functions*.

template<typename **_FromEncoding**, typename **_ToEncoding** = *utf8_t*, typename **_Range** = ::std::basic_string_view<*code_unit_t*<*_FromEncoding*>>, typename **_FromErrorHandler** = *default_handler_t*, typename **_ToErrorHandler** = *default_handler_t*, typename **_FromState** = *decode_state_t*<*_FromEncoding*>, typename **_ToState** = *encode_state_t*<*_ToEncoding*>>
class ztd::text::**transcode_view**
A transcoding iterator that takes an input of code units and provides an output over the code units of the desired _ToEncoding after converting from the _FromEncoding in a fashion that will never produce a ztd::text::encoding_error::insufficient_output error.

Remark This type produces proxies as their reference type, and are only readable, not writable iterators. The type will also try many different shortcuts for decoding the input and encoding the intermediates, respectively, including invoking a few customization points for either `decode_one.` or `encode_one` . It may also call `transcode_one` to bypass having to do the round-trip through two encodings, which an encoding pair that a developer is interested in can use to do the conversion more quickly. The view presents code units one at a time, regardless of how many code units are output by one decode operation. This means if, for example, one (1) UTF-16 code unit becomes two (2) UTF-8 code units, it will present each code unit one at a time. If you are looking to explicitly know each collection of characters, you will have to use lower-level interfaces.

tparam _FromEncoding The encoding to read the underlying range of code points as.

tparam _ToEncoding The encoding to read the underlying range of code points as.

tparam _Range The range of input that will be fed into the _FromEncoding's decode operation.

tparam _FromErrorHandler The error handler for any decode-step failures.

tparam _ToErrorHandler The error handler for any encode-step failures.

tparam _FromState The state type to use for the decode operations to intermediate code points.

tparam _ToState The state type to use for the encode operations to intermediate code points.

Public Types

using **iterator** = transcode_iterator<*_FromEncoding*, *_ToEncoding*, *_Range*, *_FromErrorHandler*, *_ToErrorHandler*, *_FromState*, *_ToState*>
The iterator type for this view.

using **sentinel** = transcode_sentinel_t
The sentinel type for this view.

using **range_type** = *_Range*
The underlying range type.

using **from_encoding_type** = *_FromEncoding*
The encoding type used for decoding to intermediate code point storage.

using **to_encoding_type** = *_ToEncoding*
The encoding type used for encoding to the final code units storage.

using **from_error_handler_type** = *_FromErrorHandler*
The error handler when a decode operation fails.

using **to_error_handler_type** = *_ToErrorHandler*
The error handler when an encode operation fails.

using **from_state_type** = *_FromState*
The state type used for decode operations.

using **to_state_type** = *_ToState*
The state type used for encode operations.

Public Functions

inline constexpr **transcode_view**(*range_type* __range) noexcept
Constructs a *transcode_view* from the underlying range.

Parameters **__range** – **[in]** The input range to wrap and iterate over.

inline constexpr **transcode_view**(*range_type* __range, *to_encoding_type* __to_encoding) noexcept
Constructs a *transcode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__to_encoding** – **[in]** The encoding object to call encode_one or equivalent functionality on.

inline constexpr **transcode_view**(*range_type* __range, *from_encoding_type* __from_encoding, *to_encoding_type* __to_encoding) noexcept

Constructs a *transcode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__from_encoding** – **[in]** The encoding object to call `decode_one` or equivalent functionality on.
- **__to_encoding** – **[in]** The encoding object to call `encode_one` or equivalent functionality on.

inline constexpr **transcode_view**(*range_type* __range, *from_encoding_type* __from_encoding, *to_encoding_type* __to_encoding, *from_error_handler_type* __from_error_handler, *to_error_handler_type* __to_error_handler) noexcept

Constructs a *transcode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__from_encoding** – **[in]** The encoding object to call `decode_one` or equivalent functionality on.
- **__to_encoding** – **[in]** The encoding object to call `encode_one` or equivalent functionality on.
- **__from_error_handler** – **[in]** The error handler for decode operations to store in this view.
- **__to_error_handler** – **[in]** The error handler for encode operations to store in this view.

inline constexpr **transcode_view**(*range_type* __range, *from_encoding_type* __from_encoding, *to_encoding_type* __to_encoding, *from_error_handler_type* __from_error_handler, *to_error_handler_type* __to_error_handler, *from_state_type* __from_state, *to_state_type* __to_state) noexcept

Constructs a *transcode_view* from the underlying range.

Parameters

- **__range** – **[in]** The input range to wrap and iterate over.
- **__from_encoding** – **[in]** The encoding object to call `decode_one` or equivalent functionality on.
- **__to_encoding** – **[in]** The encoding object to call `encode_one` or equivalent functionality on.
- **__from_error_handler** – **[in]** The error handler for decode operations to store in this view.
- **__to_error_handler** – **[in]** The error handler for encode operations to store in this view.
- **__from_state** – **[in]** The state to user for the decode operation.
- **__to_state** – **[in]** The state to user for the decode operation.

inline constexpr *iterator* **begin**() & noexcept

The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *iterator* **begin**() const & noexcept

The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *iterator* **begin**() && noexcept
: The beginning of the range. Uses a sentinel type and not a special iterator.

inline constexpr *sentinel* **end**() const noexcept
: The end of the range. Uses a sentinel type and not a special iterator.

1.9.3 Encodings

See also *the top-level encodings page* for more details about the overall status and known vs. implemented encodings.

any_encoding

`any_encoding` is a class type whose sole purpose is to provide a type-generic, byte-based, runtime-deferred way of handling encodings.

using `ztd::text::`**any_encoding** = *any_byte_encoding*<::std::byte>
: The canonical erased encoding type which uses a `std::byte` as its code unit type and an `unicode_code_point` as its code point type, with spans for input and output operations.

Remark If the input encoding does not match `std::byte`, it will be first wrapped in a *ztd::text::encoding_scheme* first.

Base Template

template<typename **_Byte**, typename **_CodePoint** = *unicode_code_point*>

class `ztd::text::`**any_byte_encoding** : public ztd::text::*any_encoding_with*<*_Byte*, const *_CodePoint*, const *_Byte*, *_CodePoint*>
: An encoding type that wraps up other encodings to specifically traffic in the given `_Byte` type provided, which is typically set to `std::byte` .

Remark This type traffics solely in `std::span` s, which for most people is fine. Others may want to interface with different iterator types (e.g., from a custom Rope implementation or other). For those, one must first create ranges that can operate with those iterators, then use them themselves. (It's not an ideal process at the moment, and we are looking to make this experience better.) It is recommended to use the provided ztd::text::any_encoding type definition instead of accessing this directly, unless you have a reason for using a different byte type (e.g., interfacing with legacy APIs).

tparam _Byte The byte type to use. Typically, this is either `unsigned char` or `std::byte` .

Public Types

using **decode_state** = any_decode_state
The state that can be used between calls to `decode`.

Remark This is an opaque struct with no members. It follows the "encoding-dependent state" model, which means it has a constructor that takes an *ztd::text::any_encoding_with* so it can properly initialize its state.

using **encode_state** = any_encode_state
The state that can be used between calls to `encode`.

Remark This is an opaque struct with no members. It follows the "encoding-dependent state" model, which means it has a constructor that takes an *ztd::text::any_encoding_with* so it can properly initialize its state.

using **code_unit** = ranges::range_value_type_t<_EncodeCodeUnits>
The individual units that result from an encode operation or are used as input to a decode operation.

using **code_point** = ranges::range_value_type_t<_DecodeCodePoints>
The individual units that result from a decode operation or as used as input to an encode operation.

using **is_encode_injective** = ::std::false_type
Whether or not the encode operation can process all forms of input into code point values.

Remark This is always going to be false because this is a type-erased encoding; this value is determined by a runtime decision, which means that the most conservative and truthful answer is selected for this property.

using **is_decode_injective** = ::std::false_type
Whether or not the decode operation can process all forms of input into code point values.

Remark This is always going to be false because this is a type-erased encoding; this value is determined by a runtime decision, which means that the most conservative and truthful answer is selected for this property.

Public Functions

any_byte_encoding() = delete
Cannot default-construct a *ztd::text::any_byte_encoding* object.

template<typename **_Encoding**, typename ...**_Args**, ::std::enable_if_t<!::std::is_same_v<*_Encoding*, *any_byte_encoding*> && !::std::is_same_v<*_Encoding*, __base_t> && !is_specialization_of_v<remove_cvref_t<*_Encoding*>, ::std::in_place_type_t>>* = nullptr>
inline **any_byte_encoding**(*_Encoding* &&__encoding, *_Args*&&... __args)
Constructs a *ztd::text::any_byte_encoding* with the encoding object and any additional arguments.

Remark If the provided encoding does not have a byte *code_unit* type, it is wrapped in an *ztd::text::encoding_scheme* first.

Parameters

- **__encoding** – **[in]** The encoding object that informs the *ztd::text::any_byte_encoding* what encoding object to store.
- **__args** – **[in]** Any additional arguments used to construct the encoding in the erased storage.

template<typename **_Encoding**, typename ...**_Args**, ::std::enable_if_t<!::std::is_same_v<*_Byte*, *code_unit_t*<remove_cvref_t<*_Encoding*>>>>* = nullptr>
inline **any_byte_encoding**(::std::in_place_type_t<*_Encoding*>, *_Args*&&... __args)
Constructs a *ztd::text::any_byte_encoding* with the encoding object and any additional arguments.

Remark If the provided encoding does not have a byte *code_unit* type, it is wrapped in an *ztd::text::encoding_scheme* first.

Template Parameters **_Encoding** – The Encoding specified by the *text_tag* argument.

Parameters **__args** – **[in]** Any additional arguments used to construct the encoding in the erased storage.

template<typename **_Encoding**, typename ...**_Args**, ::std::enable_if_t<::std::is_same_v<*_Byte*, *code_unit_t*<remove_cvref_t<*_Encoding*>>>>* = nullptr>
inline **any_byte_encoding**(::std::in_place_type_t<*_Encoding*> __tag, *_Args*&&... __args)
Constructs a *ztd::text::any_byte_encoding* with the encoding object and any additional arguments.

Remark If the provided encoding does not have a byte *code_unit* type, it is wrapped in an *ztd::text::encoding_scheme* first.

Template Parameters **_Encoding** – The Encoding specified by the *text_tag* argument.

Parameters

- **__tag** – **[in]** A *text_tag* containing the encoding type.
- **__args** – **[in]** Any additional arguments used to construct the encoding in the erased storage.

any_byte_encoding(const *any_byte_encoding*&) = delete
Cannot copy-construct a *ztd::text::any_byte_encoding* object.

any_byte_encoding &**operator=**(const *any_byte_encoding*&) = delete
Cannot copy-assign a *ztd::text::any_byte_encoding* object.

any_byte_encoding(*any_byte_encoding*&&) = default
Move-constructs a *ztd::text::any_byte_encoding* from the provided r-value reference.

Remark This leaves the passed-in r-value reference without an encoding object. Calling any function on a moved-fron *ztd::text::any_byte_encoding*, except for destruction, is a violation and invokes Undefined Behavior (generally, a crash).

any_byte_encoding &**operator=**(*any_byte_encoding*&&) = default
Move-assigns a *ztd::text::any_byte_encoding* from the provided r-value reference.

Remark This leaves the passed-in r-value reference without an encoding object. Calling any function on a moved-fron *ztd::text::any_byte_encoding*, except for destruction, is a violation and invokes Undefined Behavior (generally, a crash).

inline ::std::optional<::ztd::span<const *code_point*>> **maybe_replacement_code_points**() const noexcept
Retrieves the replacement code points for when conversions fail and *ztd::text::replacement_handler_t* (or equivalent) needs to make a substitution.

Returns A `std::optional` of `ztd::span` of `const` *code_point*s. The returned `std::optional` value is engaged (has a value) if the stored encoding has a valid `replacement_code_points` function and it can be called. If it does not, then the library checks to see if the `maybe_replacement_code_points` function exists, and returns the `std::optional` from that type directly. If neither are present, an unengaged `std::optional` is returned.

inline ::std::optional<::ztd::span<const *code_unit*>> **maybe_replacement_code_units**() const noexcept
Retrieves the replacement code units for when conversions fail and *ztd::text::replacement_handler_t* (or equivalent) needs to make a substitution.

Returns A `std::optional` of `ztd::span` of `const` *code_unit*s. The returned `std::optional` value is engaged (has a value) if the stored encoding has a valid `replacement_code_units` function and it can be called. If it does not, then the library checks to see if the `maybe_replacement_code_units` function exists, and returns the `std::optional` from that type directly. If neither are present, an unengaged `std::optional` is returned.

inline bool **contains_unicode_encoding**() const noexcept
Returns whether or not the encoding stored in this *ztd::text::any_encoding_with* is a Unicode encoding.

Remark This can be useful to know, in advance, whether or not there is a chance for lossy behavior. Even if, at compile time, various functions will demand you use an error handler, this runtime property can help you get a decent idea of just how bad and lossy this conversion might be compared to normal UTF conversion formats.

inline __decode_result **decode_one**(_DecodeCodeUnits __input, _DecodeCodePoints __output,
__decode_error_handler __error_handler, *decode_state* &__state) const
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.

- **__state** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

inline __encode_result **encode_one**(_EncodeCodePoints __input, _EncodeCodeUnits __output, __encode_error_handler __error_handler, *encode_state* &__state) const

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__state** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points** = _MaxCodePoints

The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

static constexpr ::std::size_t **max_code_units** = _MaxCodeUnits

The maximum code units a single complete operation of encoding can produce.

any_encoding_with

This is the lowest level base template, `any_encoding_with`, that sits beneath *any_encoding and any_byte_encoding*. It is recommended for power users with specific goals for the input and output types of the encode and decode operations, where normal buffer-based I/O is unsuitable. In general, you should be relying on *any_encoding and any_byte_encoding*.

template<typename **_EncodeCodeUnits**, typename **_EncodeCodePoints**, typename **_DecodeCodeUnits**, typename **_DecodeCodePoints**, ::std::size_t **_MaxCodeUnits** = __txt_detail::__default_max_code_units_any_encoding, ::std::size_t **_MaxCodePoints** = __txt_detail::__default_max_code_points_any_encoding>

class `ztd::text::`**any_encoding_with**

An encoding class which has the given encode output and input, as well as the decode input and output ranges, provided as fixed types alongside the maximum number of code units put in and pushed out.

Remark This class is generally interacted with by using its derivate class, *ztd::text::any_byte_encoding*, and its convenience alias, ztd::text::any_encoding. This class's use is recommended only for power users who have encoding ranges that cannot be interacted with through `ztd::span` and therefore need other ways. We are looking into ways to produce a ranges::subrange<any_iterator> as a completely generic range to aid those individuals who do not want to deal in just `ztd::span` s.

tparam _EncodeCodeUnits The output of `encode_one` and related operations.

tparam _EncodeCodePoints The input of `encode_one` and related operations.

tparam _DecodeCodeUnits The input of `decode_one` and related operations.

tparam _DecodeCodePoints The output of `decode_one` and related operations.

tparam _MaxCodeUnits The maximum number of code units that can be output through a given operation. Directly related to the maximum_code_units inline constexpr variable definition.

tparam _MaxCodePoints The maximum number of code points that can be output through a given operation. Directly related to the maximum_code_points inline constexpr variable definition.

Subclassed by *any_byte_encoding< _Byte, _CodePoint >*

Public Types

using **decode_state** = *any_decode_state*
The state that can be used between calls to `decode`.

Remark This is an opaque struct with no members. It follows the "encoding-dependent state" model, which means it has a constructor that takes an *ztd::text::any_encoding_with* so it can properly initialize its state.

using **encode_state** = *any_encode_state*
The state that can be used between calls to `encode`.

Remark This is an opaque struct with no members. It follows the "encoding-dependent state" model, which means it has a constructor that takes an *ztd::text::any_encoding_with* so it can properly initialize its state.

using **code_unit** = ranges::range_value_type_t<*_EncodeCodeUnits*>
The individual units that result from an encode operation or are used as input to a decode operation.

using **code_point** = ranges::range_value_type_t<*_DecodeCodePoints*>
The individual units that result from a decode operation or as used as input to an encode operation.

using **is_encode_injective** = ::std::false_type
Whether or not the encode operation can process all forms of input into code point values.

Remark This is always going to be false because this is a type-erased encoding; this value is determined by a runtime decision, which means that the most conservative and truthful answer is selected for this property.

using **is_decode_injective** = ::std::false_type
Whether or not the decode operation can process all forms of input into code point values.

Remark This is always going to be false because this is a type-erased encoding; this value is determined by a runtime decision, which means that the most conservative and truthful answer is selected for this property.

Public Functions

any_encoding_with() = delete
Cannot default-construct a *ztd::text::any_encoding_with* object.

template<typename **_Encoding**, typename ...**_Args**, ::std::enable_if_t<!::std::is_same_v<*_Encoding*, *any_encoding_with*> && !is_specialization_of_v<remove_cvref_t<*_Encoding*>, ::std::in_place_type_t>>* = nullptr>
inline **any_encoding_with**(*_Encoding* &&__encoding, *_Args*&&... __args)
Constructs a *ztd::text::any_encoding_with* with the encoding object and any additional arguments.

Template Parameters **_Encoding** – The Encoding specified by the first argument.

Parameters

- **__encoding** – **[in]** The encoding object that informs the *ztd::text::any_encoding_with* what encoding object to store.
- **__args** – **[in]** Any additional arguments used to construct the encoding in the erased storage.

template<typename **_Encoding**, typename ...**_Args**>
inline **any_encoding_with**(::std::in_place_type_t<*_Encoding*> __tag, *_Args*&&... __args)
Constructs a *ztd::text::any_encoding_with* with the encoding type specified in the `__tag` argument.

Template Parameters **_Encoding** – The Encoding specified by the `__tag` argument.

Parameters

- **__tag** – **[in]** The type marker that informs the *ztd::text::any_encoding_with* what encoding object to store.
- **__args** – **[in]** Any additional arguments used to construct the encoding in the erased storage.

any_encoding_with(const *any_encoding_with*&) = delete
Cannot copy-construct a *ztd::text::any_encoding_with* object.

any_encoding_with &**operator=**(const *any_encoding_with*&) = delete
Cannot copy-assign a *ztd::text::any_encoding_with* object.

any_encoding_with(*any_encoding_with*&&) = default
Move-constructs a *ztd::text::any_encoding_with* from the provided r-value reference.

Remark This leaves the passed-in r-value reference without an encoding object. Calling any function on a moved-fron *ztd::text::any_encoding_with*, except for destruction, is a violation and invokes Undefined Behavior (generally, a crash).

any_encoding_with &**operator=**(*any_encoding_with*&&) = default
Move-assigns a *ztd::text::any_encoding_with* from the provided r-value reference.

Remark This leaves the passed-in r-value reference without an encoding object. Calling any function on a moved-fron *ztd::text::any_encoding_with*, except for destruction, is a violation and invokes Undefined Behavior (generally, a crash).

inline ::std::optional<::ztd::span<const *code_point*>> **maybe_replacement_code_points**() const noexcept
Retrieves the replacement code points for when conversions fail and *ztd::text::replacement_handler_t* (or equivalent) needs to make a substitution.

Returns A `std::optional` of `ztd::span` of `const` *code_point*s. The returned `std::optional` value is engaged (has a value) if the stored encoding has a valid `replacement_code_points` function and it can be called. If it does not, then the library checks to see if the `maybe_replacement_code_points` function exists, and returns the `std::optional` from that type directly. If neither are present, an unengaged `std::optional` is returned.

inline ::std::optional<::ztd::span<const *code_unit*>> **maybe_replacement_code_units**() const noexcept
Retrieves the replacement code units for when conversions fail and *ztd::text::replacement_handler_t* (or equivalent) needs to make a substitution.

Returns A `std::optional` of `ztd::span` of `const` *code_unit*s. The returned `std::optional` value is engaged (has a value) if the stored encoding has a valid `replacement_code_units` function and it can be called. If it does not, then the library checks to see if the `maybe_replacement_code_units` function exists, and returns the `std::optional` from that type directly. If neither are present, an unengaged `std::optional` is returned.

inline bool **contains_unicode_encoding**() const noexcept
Returns whether or not the encoding stored in this *ztd::text::any_encoding_with* is a Unicode encoding.

Remark This can be useful to know, in advance, whether or not there is a chance for lossy behavior. Even if, at compile time, various functions will demand you use an error handler, this runtime property can help you get a decent idea of just how bad and lossy this conversion might be compared to normal UTF conversion formats.

inline __decode_result **decode_one**(*_DecodeCodeUnits* __input, *_DecodeCodePoints* __output, __decode_error_handler __error_handler, *decode_state* &__state) const
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__state** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

inline __encode_result **encode_one**(*_EncodeCodePoints* __input, *_EncodeCodeUnits* __output, __encode_error_handler __error_handler, *encode_state* &__state) const

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__state** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points** = *_MaxCodePoints*

The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

static constexpr ::std::size_t **max_code_units** = *_MaxCodeUnits*

The maximum code units a single complete operation of encoding can produce.

class **any_decode_state**

The state for any encoding's decode state.

Public Functions

inline **any_decode_state**(const *any_encoding_with* &__encoding)

Creates a state properly initialized from the stored encoding.

any_decode_state(const *any_decode_state*&) = delete

You cannot copy construct an *any_decode_state*.

any_decode_state &**operator=**(const *any_decode_state*&) = delete

You cannot copy assign an *any_decode_state*.

any_decode_state(*any_decode_state*&&) = default

Move constructs an *any_decode_state*.

any_decode_state &**operator=**(*any_decode_state*&&) = default

Move assigns an *any_decode_state*.

class **any_encode_state**
The state for any encoding's encode state.

Public Functions

inline **any_encode_state**(const *any_encoding_with* &__encoding)
Creates a state properly initialized from the stored encoding.

any_encode_state(const *any_encode_state*&) = delete
You cannot copy construct an *any_encode_state*.

any_encode_state &**operator=**(const *any_encode_state*&) = delete
You cannot copy assign an *any_encode_state*.

any_encode_state(*any_encode_state*&&) = default
Move constructs an *any_encode_state*.

any_encode_state &**operator=**(*any_encode_state*&&) = default
Move assigns an *any_encode_state*.

ASCII

The American Standard Code for Information Interchange (ASCII). A typical 7-bit encoding that is bitwise-compatible with UTF-8.

constexpr *ascii_t* ztd::text::**ascii** = {}
An instance of the ascii_t type for ease of use.

typedef *basic_ascii*<char> ztd::text::**ascii_t**
The American Standard Code for Information Exchange (ASCII) Encoding.

Remark The most vanilla and unimaginative encoding there is in the world, excluding tons of other languages, dialects, and even common English idioms and borrowed words. Please don't pick this unless you have good reason!

Base Template

template<typename **_CodeUnit**, typename **_CodePoint** = *unicode_code_point*>

class ztd::text::**basic_ascii**
The American Standard Code for Information Exchange (ASCII) Encoding.

Remark The most vanilla and unimaginative encoding there is in the world, excluding tons of other languages, dialects, and even common English idioms and borrowed words. Please don't pick this unless you have good reason!

tparam _CodeUnit The code unit type to work over.

Public Types

using **code_unit** = *_CodeUnit*
The individual units that result from an encode operation or are used as input to a decode operation.

using **code_point** = *_CodePoint*
The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **state** = __txt_detail::__empty_state
The state that can be used between calls to the encoder and decoder.

Remark It is an empty struct because there is no shift state to preserve between complete units of encoded information. It is also only `state` and not separately *`decode_state`* and *`encode_state`* because one type suffices for both.

using **is_decode_injective** = ::std::true_type
Whether or not the decode operation can process all forms of input into code point values.

Remark ASCII can decode from its 7-bit (unpacked) code units to Unicode Code Points. Since the converion is lossless, this property is true.

using **is_encode_injective** = ::std::false_type
Whether or not the encode operation can process all forms of input into code unit values. This is not true for ASCII, as many Unicode Code Point and Unicode Scalar Values cannot be represented in ASCII. Since the conversion is lossy, this property is false.

Public Static Functions

static inline constexpr ::ztd::span<const *code_unit*, 1> **replacement_code_units**() noexcept
A range of code units representing the values to use when a replacement happen. For ASCII, this must be '?' instead of the usual Unicode Replacement Character U''.

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
static inline constexpr auto **decode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *state* &__s)
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.

- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
static inline constexpr auto **encode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *state* &__s)

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexprconst::std::size_t **max_code_units** = 1

The maximum code units a single complete operation of encoding can produce.

static constexprconst::std::size_t **max_code_points** = 1

The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

cuneicode_encoding (In Progress)

Warning: This isn't finished yet! Come check back by the next major or minor version update.

This encoding is only available if the *configuration macro for ZTD_TEXT_USE_CUNEICODE* is turned on.

This encoding is tied to the cuneicode library. The cuneicode library is a C library for validation, counting, and transcoding between a fixed set of encodings, with an additional plug for arbitrary encodings that can be added at runtime. This is in opposition to *iconv*, where additional encodings can only be added by-hand through recompiling the code or hooking specific system configuration points.

cuneicode has a variable number of encodings it can be compiled with to support. States are pre-constructed in the encoding itself and copied as necessary when `encode_state` or `decode_state`s are being created to call the desired

conversion functions. The user can inspect the output error parameter from the `cuneicode_encoding` constructor to know of failure, or not pass in the output error parameter and instead take one of a assert, thrown exception, or `abort` (preferred invocation in that order).

Encoding Scheme

The `encoding_scheme` template turns any encoding into a byte-based encoding capable of reading and writing those bytes into and out of byte-`value_type` ranges. It prevents duplicating effort to read encodings as little endian or big endian, allowing composition for any desired encoding to interface with e.g. a UTF-16 Big Endian blob of data coming over a network or shared pipe.

Aliases

using `ztd::text::basic_utf16_le` = *encoding_scheme*<*utf16_t*, endian::little, _Byte>
A UTF-16 encoding, in Little Endian format, with inputs as a sequence of bytes.

Template Parameters **_Byte** – The byte type to use. Typically, this is `std::byte` or `uchar.`

using `ztd::text::utf16_le_t` = *basic_utf16_le*<::std::byte>
A UTF-16 encoding, in Little Endian format, with inputs as a sequence of bytes.

using `ztd::text::basic_utf16_be` = *encoding_scheme*<*utf16_t*, endian::big, _Byte>
A UTF-16 encoding, in Big Endian format, with inputs as a sequence of bytes.

Template Parameters **_Byte** – The byte type to use. Typically, this is `std::byte` or `unsigned char`.

using `ztd::text::utf16_be_t` = *basic_utf16_be*<::std::byte>
A UTF-16 encoding, in Big Endian format, with inputs as a sequence of bytes.

using `ztd::text::basic_utf16_ne` = *encoding_scheme*<*utf16_t*, endian::native, _Byte>
A UTF-16 encoding, in Native Endian format, with inputs as a sequence of bytes.

Template Parameters **_Byte** – The byte type to use. Typically, this is `std::byte` or `unsigned char`.

using `ztd::text::utf16_ne_t` = *basic_utf16_ne*<::std::byte>
A UTF-16 encoding, in Native Endian format, with inputs as a sequence of bytes.

using `ztd::text::basic_utf32_le` = *encoding_scheme*<*utf32_t*, endian::little, _Byte>
A UTF-32 encoding, in Little Endian format, with inputs as a sequence of bytes.

Template Parameters **_Byte** – The byte type to use. Typically, this is `std::byte` or `unsigned char` .

using `ztd::text::utf32_le_t` = *basic_utf32_le*<::std::byte>
A UTF-32 encoding, in Little Endian format, with inputs as a sequence of bytes.

using `ztd::text::basic_utf32_be` = *encoding_scheme*<*utf32_t*, endian::big, _Byte>
A UTF-32 encoding, in Big Endian format, with inputs as a sequence of bytes.

Template Parameters **_Byte** – The byte type to use. Typically, this is `std::byte` or `unsigned char` .

using ztd::text::**utf32_be_t** = *basic_utf32_be*<::std::byte>
A UTF-32 encoding, in Big Endian format, with inputs as a sequence of bytes.

using ztd::text::**basic_utf32_ne** = *encoding_scheme*<*utf32_t*, endian::native, _Byte>
A UTF-32 encoding, in Native Endian format, with inputs as a sequence of bytes.

Template Parameters **_Byte** – The byte type to use. Typically, this is `std::byte` or `unsigned char` .

using ztd::text::**utf32_ne_t** = *basic_utf32_ne*<::std::byte>
A UTF-32 encoding, in Big Endian format, with inputs as a sequence of bytes.

Base Template

template<typename **_Encoding**, endian **_Endian** = endian::native, typename **_Byte** = ::std::byte>

class ztd::text::**encoding_scheme** : public __is_unicode_encoding_es<*encoding_scheme*<*_Encoding*, *_Endian*, *_Byte*>, remove_cvref_t<unwrap_t<*_Encoding*>>>, private ebco<*_Encoding*>
Decomposes the provided Encoding type into a specific endianness (big, little, or native) to allow for a single encoding type to be viewed in different ways.

Remark For example, this can be used to construct a Big Endian UTF-16 by using *encoding_scheme*<ztd::text::utf16_t, ztd::endian::big>. It can be made interopable with `unsigned char` buffers rather than `std::byte` buffers by doing: *ztd::text::encoding_scheme*<ztd::text::utf32_t, ztd::endian::native, unsigned char>.

tparam _Encoding The encoding type.

tparam _Endian The endianess to use. Defaults to ztd::endian::native.

tparam _Byte The byte type to use. Defaults to `std::byte`.

Public Types

using **encoding_type** = *_Encoding*
The encoding type that this scheme wraps.

using **code_point** = *code_point_t*<_UBaseEncoding>
The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **code_unit** = *_Byte*
The individual units that result from an encode operation or are used as input to a decode operation.

Remark Typically, this type is usually always some kind of byte type (unsigned char or std::byte or other `sizeof(obj) == 1` type).

using **decode_state** = *decode_state_t*<_UBaseEncoding>
The state that can be used between calls to the decode function.

Remark Even if the underlying encoding only has a single `state` type, we need to separate the two out in order to generically handle all encodings. Therefore, the *encoding_scheme* will always have both `encode_state` and `decode_state`.

using **encode_state** = *encode_state_t*<_UBaseEncoding>
The state that can be used between calls to the encode function.

Remark Even if the underlying encoding only has a single `state` type, we need to separate the two out in order to generically handle all encodings. Therefore, the *encoding_scheme* will always have both `encode_state` and `decode_state`.

using **is_encode_injective** = ::std::integral_constant<bool, *is_encode_injective_v*<_UBaseEncoding>>
Whether or not the encode operation can process all forms of input into code point values.

Remark Defers to what the underlying `encoding_type` does.

using **is_decode_injective** = ::std::integral_constant<bool, *is_decode_injective_v*<_UBaseEncoding>>
Whether or not the decode operation can process all forms of input into code point values.

Remark Defers to what the underlying `encoding_type` does.

Public Functions

inline constexpr *encoding_type* &**base**() & noexcept
Retrives the underlying encoding object.

Returns An l-value reference to the encoding object.

inline constexpr const *encoding_type* &**base**() const & noexcept
Retrives the underlying encoding object.

Returns An l-value reference to the encoding object.

inline constexpr *encoding_type* &&**base**() && noexcept
Retrives the underlying encoding object.

Returns An l-value reference to the encoding object.

template<typename **_Unused** = *encoding_type*, ::std::enable_if_t<*is_code_units_replaceable_v*<*_Unused*>>* = nullptr>
inline decltype(auto) constexpr **replacement_code_units**() const noexcept
Returns, the desired replacement code units to use.

Remark This is only callable if the function call exists on the wrapped encoding. It is broken down into a contiguous view type formulated from bytes if the wrapped code unit types do not match.

template<typename **_Unused** = *encoding_type*, ::std::enable_if_t<*is_code_points_replaceable_v*<*_Unused*>>* = nullptr>
inline decltype(auto) constexpr **replacement_code_points**() const noexcept
Returns the desired replacement code points to use.

Remark Is only callable if the function call exists on the wrapped encoding.

template<typename **_Unused** = *encoding_type*,
::std::enable_if_t<*is_code_units_maybe_replaceable_v*<*_Unused*>>* = nullptr>
inline decltype(auto) constexpr **maybe_replacement_code_units**() const noexcept
Returns the desired replacement code units to use, or an empty optional-like type if there is nothing present.

Remark This is only callable if the function call exists on the wrapped encoding. It is broken down into a contiguous view type formulated from bytes if the wrapped code unit types do not match.

template<typename **_Unused** = *encoding_type*,
::std::enable_if_t<*is_code_points_maybe_replaceable_v*<*_Unused*>>* = nullptr>
inline decltype(auto) constexpr **maybe_replacement_code_points**() const noexcept
Returns the desired replacement code units to use.

Remark This Is only callable if the function call exists on the wrapped encoding.

inline constexpr bool **contains_unicode_encoding**() const noexcept
Whether or not this encoding is some form of Unicode encoding.

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
inline constexpr auto **decode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *decode_state* &__s) const
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
inline constexpr auto **encode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *encode_state* &__s) const
Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexprconst::std::size_t **max_code_points** = *max_code_points_v*<_UBaseEncoding>
The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

static constexprconst::std::size_t **max_code_units** = (*max_code_units_v*<_UBaseEncoding> * sizeof(_BaseCodeUnit)) / (sizeof(*_Byte*))
The maximum code units a single complete operation of encoding can produce.

Execution

This is the locale-based, runtime encoding. It uses a number of compile-time and runtime heuristics to eventually be resolved to an implementation-defined encoding. It is not required to work in constant expressions either: for this, use *ztd::text::literal*, which represents the compile-time string (e.g. `"my string"`) encoding.

Currently, the hierachy of behaviors is like so:

- If the platform is MacOS, then it assumes this is *UTF-8*;
- Otherwise, if libiconv is available, then it attempts to use *iconv* configured to the `"char"`-identified encoding;
- Otherwise, if the headers <cuchar> or <uchar.h> are available, then it attempts to use a gnarly, lossy, and dangerous encoding that potentially traffics through the C Standard Library and Locale APIs;
- Otherwise, it produces a compile-time error.

Warning: The C Standard Library has many design defects in its production of code points, which may make it unsuitable even if your C Standard Library recognizes certain locales (e.g., Big5-HKSCS). The runtime will always attempt to load `iconv` if the definition is turned on, since it may do a better job than the C Standard Library's interfaces until C23.

Even if, on a given platform, it can be assumed to be a static encoding (e.g., Apple/MacOS where it always returns the "C" Locale but processes text as UTF-8), `ztd::text::execution` will always present itself as a runtime and unknowable encoding. This is to prevent portability issues from relying on, e.g., `ztd::text::is_decode_injective_v<ztd::text::execution>` being true during development and working with that assumption, only to have it break when ported to a platform where that assumption no longer holds.

constexpr *execution_t* ztd::text::**execution** = {}
An instance of the execution_t type for ease of use.

typedef no_encoding<char, *unicode_code_point*> ztd::text::**execution_t**
The Encoding that represents the "Execution" (narrow locale-based) encoding. The encoding is typically associated with the locale, which is tied to the C standard library's setlocale function.

Remark Use of this type is subject to the C Standard Library or platform defaults. Some locales (such as the Big5 Hong King Supplementary Character Set (Big5-HKSCS)) are broken when accessed without ZTD_TEXT_USE_CUNEICODE beingdefined, due to fundamental design issues in the C Standard Library and bugs in glibc/musl libc's current locale encoding support. On Apple, this is cuurrently assumed to be UTF-8 since they do not support the <cuchar> or <uchar.h> headers.

Internal Types

Warning: Names with double underscores, and within the __detail and __impl namespaces are reserved for the implementation. Referencing this entity directly is bad, and the name/functionality can be changed at any point in the future. Relying on anything not guaranteed by the documentation is Undefined Behavior.

MacOS-based

class ztd::text::__txt_impl::**__execution_mac_os** : private __utf8_with<*__execution_mac_os*, char, char32_t>
The default ("locale") encoding for Mac OS.

Remark Note that for all intents and purposes, Mac OS demands that all text is in UTF-8. However, on Big Sur, Catalina, and a few other platforms locale functionality and data has been either forgotten/left behind or intentionally kept in place on these devices. It may be possible that with very dedicated hacks one can still change the desired default encoding from UTF-8 to something else in the majority of Apple text. Their documentation states that all text "should" be UTF-8, but very explicitly goes out of its way to not make that hard guarantee. Since it is a BSD-like system and they left plenty of that data behind from C libraries, this may break in extremely obscure cases. Please be careful on Apple machines!

Public Types

using **code_point** = *code_point_t*<__base_t>
The code point type that is decoded to, and encoded from. ///.

using **code_unit** = *code_unit_t*<__base_t>
The code unit type that is decoded from, and encoded to. ///.

using **decode_state** = *decode_state_t*<__base_t>
The associated state for decode operations. ///.

using **encode_state** = *encode_state_t*<__base_t>
The associated state for encode operations. ///.

using **is_unicode_encoding** = ::std::integral_constant<bool, *is_unicode_encoding_v*<__base_t>>
Whether or not this encoding is a unicode encoding or not. ///.

using **is_decode_injective** = ::std::false_type
Whether or not this encoding's `decode_one` step is injective or not. ///.

using **is_encode_injective** = ::std::false_type
Whether or not this encoding's `encode_one` step is injective or not. ///.

Public Static Functions

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
static inline constexpr auto **decode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *decode_state* &__s)
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. Most encodings have no state, but because this is effectively a runtime encoding and therefore it is important to preserve and manage this state.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
static inline constexpr auto **encode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *encode_state* &__s)
Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. Most encodings have no state, but because this is effectively a runtime encoding and therefore it is important to preserve and manage this state.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points** = 8
The maximum code units a single complete operation of encoding can produce.

Remark There are encodings for which one input can produce 3 code points (some Tamil encodings) and there are rumours of an encoding that can produce 7 code points from a handful of input. We use a protective/conservative 8, here, to make sure ABI isn't broken later.

static constexpr ::std::size_t **max_code_units** = MB_LEN_MAX
The maximum number of code points a single complete operation of decoding can produce.

Remark This is bounded by the platform's `MB_LEN_MAX` macro, which is an integral constant expression representing the maximum value of output all C locales can produce from a single complete operation.

Private Static Functions

static inline constexpr auto **encode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *encode_state* &__s)
Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

static inline constexpr auto **decode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *decode_state* &__s)
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

iconv_encoding (In Progress)

Warning: This isn't finished yet! Come check back by the next major or minor version update.

This encoding is only available if the *configuration macro for ZTD_TEXT_USE_ICONV* is turned on.

This encoding is tied to the iconv library. It will attempt to bootstrap iconv on first use of the encoding through use of `GetProcAddress/dlsym` and friends. If it cannot find it will either assert, abort, or loudly annoy the user in some way. The code is retrieved dynamically, as iconv is under a LGPL/GPL licensed and cannot be traditionally built / statically linked with application code (though in the future we may provide a way for software to do that if the software being made with this library is also GPL-compatible software).

iconv has a fixed set of encodings it can be compiled with to support. States are pre-constructed in the encoding itself and copied as necessary when `encode_state` or `decode_state`s are being created to call the iconv functions. The user can inspect the output error parameter from the `iconv_encoding` constructor to know of failure, or not pass in the output error parameter and instead take one of a assert, thrown exception, or `abort` (preferred invocation in that order).

Literal

The `literal` encoding handles C and C++ string literals ("") used at compile time and stored in the binary. The library uses a number of heuristics to determine with any degree of certainty what the encoding of string literals are, but in some cases it is not explicitly possible to achieve this goal.

If the library cannot figure out the literal encoding, the code will typically error with a `static_assert`, loudly, that it cannot use the functions on the type when you attempt to do anything with them because it may mangle whatever input or output you are expecting.

If you know the encoding of literals for your project (you provide the command line switch, or similar), then you can define a *configuration macro named ZTD_CXX_COMPILE_TIME_ENCODING_NAME* to be a string literal of your type, such as `"UTF-8"` or `"ISO-8859-1"`.

If the library does not recognize the encoding and cannot transcode it properly, it will also loudly warn you that it does not understand the encoding of the literal (in which case, file an issue about it and we will add it to the list of acceptable literal encodings).

If you like to live dangerously and do not care for the warnings, you can define *a configuration macro named ZTD_TEXT_YES_PLEASE_DESTROY_MY_LITERALS_UTTERLY_I_MEAN_IT* and it will just blindly go with whatever weird default it ended up deciding on.

(This is usually a catastrophically terrible idea, but let is not be said that we didn't give you the power to do great things, even if it cost you your foot.)

constexpr *literal_t* `ztd::text::literal` = {}
An instance of the *literal_t* type for ease of use.

class `ztd::text::literal_t` : private ebco<__txt_detail::__literal>
The encoding of string literal_ts (e.g. "") at compile time.

Public Types

using **is_unicode_encoding** = ::std::integral_constant<bool, __idk_detail::__is_unicode_encoding_id(__txt_detail::__literal_id)>
Whether or not this *literal_t* encoding is a Unicode Transformation Format, such as UTF-8, UTF-EBCDIC, or GB18030.

using **code_unit** = *code_unit_t*<__underlying_t>
The individual units that result from an encode operation or are used as input to a decode operation.

using **code_point** = *code_point_t*<__underlying_t>
The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **encode_state** = *encode_state_t*<__underlying_t>
The state that can be used between calls to encode_one.

using **decode_state** = *decode_state_t*<__underlying_t>
The state that can be used between calls to decode_one.

using **is_decode_injective** = ::std::integral_constant<bool, *is_decode_injective_v*<__underlying_t>>
Whether or not the decode operation can process all forms of input into code point values.

Remark The decode step should always be injective because every encoding used for literal_ts in C++ needs to be capable of being represented by UCNs. Whether or not a platform is a jerk, who knows?

using **is_encode_injective** = ::std::integral_constant<bool, *is_encode_injective_v*<__underlying_t>>
Whether or not the encode operation can process all forms of input into code unit values.

Remark This is absolutely not guaranteed to be the case, and as such we must check the provided encoding name for us to be sure.

Public Functions

constexpr **literal_t**() noexcept = default
Default constructs a ztd::text::literal.

constexpr **literal_t**(const *literal_t*&) noexcept = default
Copy constructs a ztd::text::literal.

constexpr **literal_t**(*literal_t*&&) noexcept = default
Move constructs a ztd::text::literal.

constexpr *literal_t* &**operator=**(const *literal_t*&) noexcept = default
Copy assigns into a *ztd::text::literal_t* object.

constexpr *literal_t* &**operator=**(*literal_t*&&) noexcept = default
Move assigns into a *ztd::text::literal_t* object.

template<typename **_Input**, typename **_Output**, typename **_ErrorHandler**>
inline constexpr auto **decode_one**(*_Input* &&__input, *_Output* &&__output, *_ErrorHandler* &&__error_handler, *decode_state* &__state) const
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__state** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

template<typename **_Input**, typename **_Output**, typename **_ErrorHandler**>
inline constexpr auto **encode_one**(*_Input* &&__input, *_Output* &&__output, *_ErrorHandler* &&__error_handler, *encode_state* &__state) const
Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__state** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points** = 16
The maximum number of code points a single complete operation of decoding can produce.

static constexpr ::std::size_t **max_code_units** = 32
The maximum code units a single complete operation of encoding can produce.

Modified UTF-8

Modified Unicode Transformation Format 8 (MUTF-8) is a UTF-8 format employed by some Android components and other ecosystems. It's special property is that it encodes the NULL character in C-style strings (`'\0'`) as an overlong sequence. This is normally illegal in UTF-8, but allowed here to allow easier interoperation with these systems.

constexpr *mutf8_t* `ztd::text::`**mutf8** = {}
An instance of the MUTF-8 type for ease of use.

using `ztd::text::`**mutf8_t** = *basic_mutf8*<uchar8_t>
A Modified UTF-8 Encoding that traffics in char8_t. See *ztd::text::basic_mutf8* for more details.

Base Template

template<typename **_CodeUnit**, typename **_CodePoint** = *unicode_code_point*>

class `ztd::text::`**basic_mutf8** : public __utf8_with<*basic_mutf8*<*_CodeUnit*, *_CodePoint*>, *_CodeUnit*, *_CodePoint*, __txt_detail::__empty_state, __txt_detail::__empty_state, true, false, true>
A Modified UTF-8 Encoding that traffics in, specifically, the desired code unit type provided as a template argument.

Remark This type as a maximum of 6 input code points and a maximum of 1 output code point. Null values are encoded as an overlong sequence to specifically avoid problems with C-style strings, which is useful for working with bad implementations sitting on top of POSIX or other Operating System APIs. For a strict, Unicode-compliant UTF-8 Encoding, see *ztd::text::basic_utf8* .

tparam _CodeUnit The code unit type to use.

tparam _CodePoint The code point type to use.

Public Types

using **is_unicode_encoding** = ::std::true_type
Whether or not this encoding that can encode all of Unicode.

using **decode_state** = __txt_detail::__empty_state
The state that can be used between calls to the encoder and decoder. It is normally an empty struct because there is no shift state to preserve between complete units of encoded information.

using **encode_state** = __txt_detail::__empty_state
The state that can be used between calls to the encoder and decoder. It is normally an empty struct because there is no shift state to preserve between complete units of encoded information.

using **code_unit** = *_CodeUnit*

The individual units that result from an encode operation or are used as input to a decode operation. For UTF-8 formats, this is usually char8_t, but this can change (see *ztd::text::basic_utf8*).

using **code_point** = *_CodePoint*

The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **is_decode_injective** = ::std::true_type

Whether or not the decode operation can process all forms of input into code point values. Thsi is true for all Unicode Transformation Formats (UTFs), which can encode and decode without a loss of information from a valid collection of code units.

using **is_encode_injective** = ::std::true_type

Whether or not the encode operation can process all forms of input into code unit values. This is true for all Unicode Transformation Formats (UTFs), which can encode and decode without loss of information from a valid input code point.

Public Static Functions

static inline constexpr auto **encode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *encode_state* &__s)

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

static inline constexpr auto **decode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *decode_state* &__s)

Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points**
The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

static constexpr ::std::size_t **max_code_units**
The maximum code units a single complete operation of encoding can produce. If overlong sequence allowed, this is 6: otherwise, this is 4.

UTF-16

constexpr *utf16_t* ztd::text::**utf16** = {}
An instance of the UTF-16 encoding for ease of use.

typedef *basic_utf16*<char16_t, *unicode_code_point*> ztd::text::**utf16_t**
A UTF-16 Encoding that traffics in char16_t. See *ztd::text::basic_utf16* for more details.

Base Template

template<typename **_CodeUnit**, typename **_CodePoint** = *unicode_code_point*>

class ztd::text::**basic_utf16** : public __utf16_with<*basic_utf16*<*_CodeUnit*, *_CodePoint*>, *_CodeUnit*, *_CodePoint*>
A UTF-16 Encoding that traffics in, specifically, the desired code unit type provided as a template argument.

Remark This is a strict UTF-16 implementation that does not allow lone, unpaired surrogates either in or out.

tparam _CodeUnit The code unit type to use.

tparam _CodePoint The code point type to use.

Public Types

using **is_unicode_encoding** = ::std::true_type
Whether or not this encoding that can encode all of Unicode.

using **state** = __txt_detail::__empty_state
The state that can be used between calls to the encoder and decoder. It is an empty struct because there is no shift state to preserve between complete units of encoded information.

using **code_unit** = *_CodeUnit*
The individual units that result from an encode operation or are used as input to a decode operation. For UTF-16 formats, this is usually char16_t, but this can change (see *ztd::text::basic_utf16*).

using **code_point** = *_CodePoint*
The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **is_decode_injective** = ::std::true_type
Whether or not the decode operation can process all forms of input into code point values. Thsi is true for all Unicode Transformation Formats (UTFs), which can encode and decode without a loss of information from a valid collection of code units.

using **is_encode_injective** = ::std::true_type
Whether or not the encode operation can process all forms of input into code unit values. This is true for all Unicode Transformation Formats (UTFs), which can encode and decode without loss of information from a valid input code point.

Public Static Functions

static inline constexpr auto **decode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *state* &__s)
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

static inline constexpr auto **encode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *state* &__s)

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points**

The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

static constexpr ::std::size_t **max_code_units**

The maximum code units a single complete operation of encoding can produce.

UTF-32

constexpr *utf32_t* `ztd::text::utf32` = {}

An instance of the UTF-32 encoding for ease of use.

typedef *basic_utf32*<char32_t, *unicode_code_point*> `ztd::text::utf32_t`

A UTF-32 Encoding that traffics in char32_t. See *ztd::text::basic_utf32* for more details.

Base Template

template<typename **_CodeUnit**, typename **_CodePoint** = *unicode_code_point*>

class `ztd::text::basic_utf32` : public __utf32_with<*basic_utf32*<*_CodeUnit*, *_CodePoint*>, *_CodeUnit*, *_CodePoint*>

A UTF-32 Encoding that traffics in, specifically, the desired code unit type provided as a template argument.

Remark This is a strict UTF-32 implementation that does not allow lone, unpaired surrogates either in or out.

tparam _CodeUnit The code unit type to use.

tparam _CodeUnit The code point type to use.

Public Types

using **is_unicode_encoding** = ::std::true_type
Whether or not this encoding that can encode all of Unicode.

using **state** = __txt_detail::__empty_state
The state that can be used between calls to the encoder and decoder. It is an empty struct because there is no shift state to preserve between complete units of encoded information.

using **code_unit** = *_CodeUnit*
The individual units that result from an encode operation or are used as input to a decode operation. For UTF-32 formats, this is usually char32_t, but this can change (see *ztd::text::basic_utf32*).

using **code_point** = *_CodePoint*
The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **is_decode_injective** = ::std::true_type
Whether or not the decode operation can process all forms of input into code point values. Thsi is true for all Unicode Transformation Formats (UTFs), which can encode and decode without a loss of information from a valid collection of code units.

using **is_encode_injective** = ::std::true_type
Whether or not the encode operation can process all forms of input into code unit values. This is true for all Unicode Transformation Formats (UTFs), which can encode and decode without loss of information from a valid input code point.

Public Static Functions

static inline constexpr auto **decode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *state* &__s)
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

static inline constexpr auto **encode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *state* &__s)

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points**

The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

static constexpr ::std::size_t **max_code_units**

The maximum code units a single complete operation of encoding can produce.

UTF-8

Unicode Transformation Format 8 (UTF-8) is an encoding for text that traffics code units 8-bits at a time. It is ubiquitous amongst web and other shared protocols and the preferred storage format for non-legacy environments and operations. It is preferred that all text is placed in UTF-8 format when working internally in your application, unless you have some special reason (e.g., interoperation with JavaScript or Qt).

constexpr *utf8_t* ztd::text::**utf8** = { }

An instance of the UTF-8 encoding for ease of use.

typedef *basic_utf8*<uchar8_t, *unicode_code_point*> ztd::text::**utf8_t**

A UTF-8 Encoding that traffics in uchar8_t. See *ztd::text::basic_utf8* for more details.

Base Template

template<typename **_CodeUnit**, typename **_CodePoint** = *unicode_code_point*>

class `ztd::text::`**basic_utf8** : public __utf8_with<*basic_utf8*<*_CodeUnit*, *_CodePoint*>, *_CodeUnit*, *_CodePoint*>

A UTF-8 Encoding that traffics in, specifically, the desired code unit type provided as a template argument.

Remark This type as a maximum of 4 input code points and a maximum of 1 output code point. It strictly follows the Unicode Specification for allowed conversions. For overlong sequences (e.g., similar to Android or Java UTF-8 implementations) and other quirks, see *ztd::text::basic_mutf8* or *ztd::text::basic_wtf8* .

tparam _CodeUnit The code unit type to use.

tparam _CodePoint The code point type to use.

Public Types

using **is_unicode_encoding** = ::std::true_type

Whether or not this encoding that can encode all of Unicode.

using **decode_state** = __txt_detail::__empty_state

The state that can be used between calls to the encoder and decoder. It is normally an empty struct because there is no shift state to preserve between complete units of encoded information.

using **encode_state** = __txt_detail::__empty_state

The state that can be used between calls to the encoder and decoder. It is normally an empty struct because there is no shift state to preserve between complete units of encoded information.

using **code_unit** = *_CodeUnit*

The individual units that result from an encode operation or are used as input to a decode operation. For UTF-8 formats, this is usually char8_t, but this can change (see *ztd::text::basic_utf8*).

using **code_point** = *_CodePoint*

The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **is_decode_injective** = ::std::true_type

Whether or not the decode operation can process all forms of input into code point values. Thsi is true for all Unicode Transformation Formats (UTFs), which can encode and decode without a loss of information from a valid collection of code units.

using **is_encode_injective** = ::std::true_type

Whether or not the encode operation can process all forms of input into code unit values. This is true for all Unicode Transformation Formats (UTFs), which can encode and decode without loss of information from a valid input code point.

Public Static Functions

static inline constexpr auto **encode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *encode_state* &__s)

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

static inline constexpr auto **decode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *decode_state* &__s)

Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points**
: The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

static constexpr ::std::size_t **max_code_units**
: The maximum code units a single complete operation of encoding can produce. If overlong sequence allowed, this is 6: otherwise, this is 4.

Wide Execution

This is the locale-based, wide runtime encoding. It uses a number of compile-time and runtime heuristics to eventually be resolved to an implementation-defined encoding. It is not required to work in constant expressions either: for this, use *ztd::text::wide_literal*, which represents the compile-time wide string (e.g. `L"my string"`) encoding.

Currently, the hierachy of behaviors is like so:

- If the platform is Windows, then it assumes this is *UTF-16*;
- Otherwise, if libiconv is available, then it attempts to use *iconv* configured to the `"wchar_t"`-identified encoding;
- Otherwise, if the platform is MacOS and `WCHAR_MAX` is greater than the maximum of an unsigned 21-bit number, or `__STDC_ISO_10646__` is defined, then it attempts to use *UTF-32*;
- Otherwise, if the headers <cwchar> or <wchar.h> are available, then it attempts to use a gnarly, lossy, and dangerous encoding that potentially traffics through the C Standard Library and Locale APIs in conjunction with a roundtrip through the *ztd::text::execution* encoding;
- Otherwise, it produces a compile-time error.

Warning: The C Standard Library has many design defects in its production of code points, which may make it unsuitable even if your C Standard Library recognizes certain locales (e.g., Big5-HKSCS). The runtime will always attempt to load `iconv` if the definition is turned on, since it may do a better job than the C Standard Library's interfaces until C23.

Even if, on a given platform, it can be assumed to be a static encoding (e.g., Apple/MacOS where it always returns the "C" Locale but processes text as UTF-32), `ztd::text::wide_execution` will always present itself as a runtime and unknowable encoding. This is to prevent portability issues from relying on, e.g., `ztd::text::is_decode_injective_v<ztd::text::wide_execution>` being true during development and working with that assumption, only to have it break when ported to a platform where that assumption no longer holds.

constexpr *wide_execution_t* `ztd::text::`**wide_execution** = { }
: An instance of the wide_execution_t type for ease of use.

typedef __txt_impl::*__wide_execution_cwchar* `ztd::text::`**wide_execution_t**
: The Encoding that represents the "Wide Execution" (wide locale-based) encoding. The wide execution encoding is typically associated with the locale, which is tied to the C standard library's setlocale function.

Remark Windows uses UTF-16, unless you call the C Standard Library directly. If `ZTD_TEXT_USE_CUNEICODE` or `ZTD_TEXT_ICONV` are not defined, this object may use the C Standard Library to perform transcoding if certain platform facilities are disabled or not available. If this is the case, the C Standard Library has fundamental limitations which may treat your UTF-16 data like UCS-2, and result in broken input/output. This

object uses UTF-16 directly on Windows when possible to avoid some of the platform-specific shenanigans. It will attempt to do UTF-32 conversions where possible as well, relying on C Standard definitions.

Internal Type

Warning: Names with double underscores, and within the `__detail` and `__impl` namespaces are reserved for the implementation. Referencing this entity directly is bad, and the name/functionality can be changed at any point in the future. Relying on anything not guaranteed by the documentation is Undefined Behavior.

<cwchar>-based

class `ztd::text::__txt_impl::__wide_execution_cwchar`

The Encoding that represents the "Wide Execution" (wide locale-based) encoding. This iteration uses the C Standard Library to do its job.

Remark Because this encoding uses the C Standard Library's functions, it is both slower and effectively dangerous because it requires a roundtrip through the encoding to get to UTF-32, and vice-versa. This is only used when `wchar_t` and its locale-based runtime encoding cannot be determined to be UTF-32, UTF-16, or some other statically-known encoding. These conversions may also be lossy.

Public Types

using **code_unit** = wchar_t

The individual units that result from an encode operation or are used as input to a decode operation.

Remark Please note that wchar_t is a variably sized type across platforms and may not represent either UTF-16 or UTF-32, including on *nix or POSIX platforms.

using **code_point** = *unicode_code_point*

The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **decode_state** = __wide_decode_state

The state of the wide encoding used between calls, which may potentially manage shift state.

Remark This type can potentially have lots of state due to the way the C API is specified.

using **encode_state** = __wide_encode_state

The state of the wide encoding used between calls, which may potentially manage shift state.

Remark This type can potentially have lots of state due to the way the C API is specified.

using **is_decode_injective** = ::std::false_type

Whether or not the decode operation can process all forms of input into code point values.

Remark All known wide encodings can decode into Unicode just fine.

using **is_encode_injective** = ::std::false_type
Whether or not the encode operation can process all forms of input into code unit values. On Windows, this is guaranteed to be UTF-16 encoding for the platform. Normally, this is UTF-32 on *nix/POSIX machines, but it can (and has been) changed before, sometimes even at runtime.

Remark IBM encodings/computers make life interesting…

using **is_unicode_encoding** = ::std::false_type
Whether or not this encoding a Unicode encoding of some type.

Remark On Windows, this is always true. On other platforms, the guarantees are not quite there. IBM encodings/computers make life interesting…

Public Static Functions

static inline bool **contains_unicode_encoding**() noexcept
Returns whether or not this encoding is a unicode encoding.

Remark This function operates at runtime and queries the existing locale through a variety of platform-specific means (such as `nl_langinfo` for POSIX, ACP probing on Windows, or fallin back to `std::setlocale` name checking otherwise).

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
static inline auto **encode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *encode_state* &__s)
Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark Platform APIs and/or the C Standard Library may be used to properly decode one complete unit of information (alongside std::mbstate_t usage). Whether or not the state is used is based on the implementation and what it chooses. If ZTD_TEXT_USE_CUNEICODE is defined, the ztd.cuneicode library may be used to fulfill this functionality.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.

- **__s** – **[inout]** The necessary state information. Most encodings have no state, but because this is effectively a runtime encoding and therefore it is important to preserve and manage this state.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
static inline auto **decode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *decode_state* &__s)

Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark Platform APIs and/or the C Standard Library may be used to properly decode one complete unit of information (alongside std::mbstate_t usage). Whether or not the state is used is based on the implementation and what it chooses. If `ZTD_TEXT_USE_CUNEICODE` is defined, the ztd.cuneicode library may be used to fulfill this functionality.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. Most encodings have no state, but because this is effectively a runtime encoding and therefore it is important to preserve and manage this state.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexprconst::std::size_t **max_code_units** = 8

The maximum code units a single complete operation of encoding can produce.

static constexprconst::std::size_t **max_code_points** = 8

The maximum number of code points a single complete operation of decoding can produce.

MacOS-based

class ztd::text::__txt_impl::__wide_execution_iso10646 : private __utf32_with<*__wide_execution_iso10646*, wchar_t, char32_t>

The wide encoding, as envisioned by ISO 10646. Typically UTF-32 with native endianness.

Remark This is generally only turned on when the Standard Definition is turn oned ( ). It effectively uses UTF-32 since that's the only encoding that can meet the original requirement of the C Standard and C Standard Library with respect to what happens with individual `wchar_t` objects.

Public Types

using **code_point** = *code_point_t*<__base_t>

The code point type that is decoded to, and encoded from.

using **code_unit** = *code_unit_t*<__base_t>

The code unit type that is decoded from, and encoded to.

using **decode_state** = *decode_state_t*<__base_t>

The associated state for decode operations.

using **encode_state** = *encode_state_t*<__base_t>

The associated state for encode operations.

using **is_unicode_encoding** = ::std::integral_constant<bool, *is_unicode_encoding_v*<__base_t>>

Whether or not this encoding is a unicode encoding or not.

using **is_decode_injective** = ::std::false_type

Whether or not this encoding's `decode_one` step is injective or not.

using **is_encode_injective** = ::std::false_type

Whether or not this encoding's `encode_one` step is injective or not.

Public Static Functions

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
static inline constexpr auto **decode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *decode_state* &__s)

Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.

- **__s** – **[inout]** The necessary state information. Most encodings have no state, but because this is effectively a runtime encoding and therefore it is important to preserve and manage this state.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

template<typename **_InputRange**, typename **_OutputRange**, typename **_ErrorHandler**>
static inline constexpr auto **encode_one**(*_InputRange* &&__input, *_OutputRange* &&__output, *_ErrorHandler* &&__error_handler, *encode_state* &__s)

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. Most encodings have no state, but because this is effectively a runtime encoding and therefore it is important to preserve and manage this state.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexprconst::std::size_t **max_code_units** = 8

The maximum code units a single complete operation of encoding can produce.

static constexprconst::std::size_t **max_code_points** = 8

The maximum number of code points a single complete operation of decoding can produce.

Private Types

using **state** = __txt_detail::__empty_state

The state that can be used between calls to the encoder and decoder. It is an empty struct because there is no shift state to preserve between complete units of encoded information.

Private Static Functions

static inline constexpr auto **decode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *state* &__s)

Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

static inline constexpr auto **encode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *state* &__s)

Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Wide Literal

The `wide_literal` encoding handles C and C++ wide string literals (`L""`) used at compile time and stored in the binary. The library uses a number of heuristics to determine with any degree of certainty what the encoding of string literals are, but in some cases it is not explicitly possible to achieve this goal.

If the library cannot figure out the wide literal encoding, the code will typically error with a `static_assert`, loudly, that it cannot use the functions on the type when you attempt to do anything with them because it may mangle whatever input or output you are expecting.

If you know the encoding of wide literals for your project (you provide the command line switch, or similar), then you can define a *configuration macro named ZTD_CXX_COMPILE_TIME_WIDE_ENCODING_NAME* to be a string literal of your type, such as `"UTF-16"` or `"EUC-TW"`.

If the library does not recognize the encoding and cannot transcode it properly, it will also loudly warn you that it does not understand the encoding of the literal (in which case, file an issue about it and we will add it to the list of acceptable wide literal encodings).

If you like to live dangerously and do not care for the warnings, you can define *a configuration macro named ZTD_TEXT_YES_PLEASE_DESTROY_MY_WIDE_LITERALS_UTTERLY_I_MEAN_IT* and it will just blindly go with whatever weird default it ended up deciding on.

(This is usually a catastrophically terrible idea, but let is not be said that we didn't give you the power to do great things, even if it cost you your foot.)

constexpr *wide_literal_t* `ztd::text::wide_literal` = {}
An instance of the *wide_literal_t* type for ease of use.

class `ztd::text::wide_literal_t` : private ebco<__txt_detail::__wide_literal, 0>
The encoding of wide string literal_ts (e.g. "") at compile time.

Public Types

using **is_unicode_encoding** = ::std::integral_constant<bool, __idk_detail::__is_unicode_encoding_id(__txt_detail::__wide_literal_id)>
Whether or not this *wide_literal_t* encoding is a Unicode Transformation Format, such as UTF-GB18030, UTF-16, or UTF-32.

using **code_unit** = *code_unit_t*<__underlying_t>
The individual units that result from an encode operation or are used as input to a decode operation.

using **code_point** = *code_point_t*<__underlying_t>
The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **encode_state** = *encode_state_t*<__underlying_t>
The state that can be used between calls to encode_one.

using **decode_state** = *decode_state_t*<__underlying_t>
The state that can be used between calls to decode_one.

using **is_decode_injective** = ::std::integral_constant<bool, *is_decode_injective_v*<__underlying_t>>
Whether or not the decode operation can process all forms of input into code point values.

Remark The decode step is always injective because every encoding used for literal_ts in C++ needs to be capable of being represented by UCNs.

using **is_encode_injective** = ::std::integral_constant<bool, *is_encode_injective_v*<__underlying_t>>
Whether or not the encode operation can process all forms of input into code unit values.

Remark This is absolutely not guaranteed to be the case, and as such we must check the provided encoding name for wide to be sure.

Public Functions

constexpr **wide_literal_t**() noexcept = default
Default constructs a ztd::text::wide_literal.

constexpr **wide_literal_t**(const *wide_literal_t*&) noexcept = default
Copy constructs a ztd::text::wide_literal.

constexpr **wide_literal_t**(*wide_literal_t*&&) noexcept = default
Move constructs a ztd::text::wide_literal.

constexpr *wide_literal_t* &**operator=**(const *wide_literal_t*&) noexcept = default
Copy assigns into a *ztd::text::wide_literal_t* object.

constexpr *wide_literal_t* &**operator=**(*wide_literal_t*&&) noexcept = default
Move assigns into a *ztd::text::wide_literal_t* object.

template<typename **_Input**, typename **_Output**, typename **_ErrorHandler**>
inline constexpr auto **decode_one**(*_Input* &&__input, *_Output* &&__output, *_ErrorHandler* &&__error_handler, *decode_state* &__state) const
Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__state** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

template<typename **_Input**, typename **_Output**, typename **_ErrorHandler**>
inline constexpr auto **encode_one**(*_Input* &&__input, *_Output* &&__output, *_ErrorHandler* &&__error_handler, *encode_state* &__state) const
Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.

- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__state** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points** = 8
The maximum number of code points a single complete operation of decoding can produce.

static constexpr ::std::size_t **max_code_units** = 16
The maximum code units a single complete operation of encoding can produce.

WTF-8

Wobby Transformat Format 8 (WTF-8) is an encoding scheme that preserves lone-encoded surrogates, which is generally not allowed in streams composed purely of Unicode Scalar Values.

constexpr *wtf8_t* ztd::text::**wtf8** = {}
An instance of the WTF-8 type for ease of use.

using ztd::text::**wtf8_t** = *basic_wtf8*<uchar8_t>
A "Wobbly Transformation Format 8" (WTF-8) Encoding that traffics in char8_t. See *ztd::text::basic_wtf8* for more details.

Base Template

template<typename **_CodeUnit**, typename **_CodePoint** = *unicode_code_point*>

class ztd::text::**basic_wtf8** : public __utf8_with<*basic_wtf8*<*_CodeUnit*, *_CodePoint*>, *_CodeUnit*, *_CodePoint*, __txt_detail::__empty_state, __txt_detail::__empty_state, false, true, false>
A "Wobbly Transformation Format 8" (WTF-8) Encoding that traffics in, specifically, the desired code unit type provided as a template argument.

Remark This type as a maximum of 4 input code points and a maximum of 1 output code point. Unpaired surrogates are allowed in this type, which may be useful for dealing with legacy storage and implementations of the Windows Filesystem (modern Windows no longer lets non-Unicode filenames through). For a strict, Unicode-compliant UTF-8 Encoding, see *ztd::text::basic_utf8* .

tparam _CodeUnit The code unit type to use.

tparam _CodePoint The code point type to use.

Public Types

using **is_unicode_encoding** = ::std::true_type
Whether or not this encoding that can encode all of Unicode.

using **decode_state** = __txt_detail::__empty_state
The state that can be used between calls to the encoder and decoder. It is normally an empty struct because there is no shift state to preserve between complete units of encoded information.

using **encode_state** = __txt_detail::__empty_state
The state that can be used between calls to the encoder and decoder. It is normally an empty struct because there is no shift state to preserve between complete units of encoded information.

using **code_unit** = *_CodeUnit*
The individual units that result from an encode operation or are used as input to a decode operation. For UTF-8 formats, this is usually char8_t, but this can change (see *ztd::text::basic_utf8*).

using **code_point** = *_CodePoint*
The individual units that result from a decode operation or as used as input to an encode operation. For most encodings, this is going to be a Unicode Code Point or a Unicode Scalar Value.

using **is_decode_injective** = ::std::true_type
Whether or not the decode operation can process all forms of input into code point values. Thsi is true for all Unicode Transformation Formats (UTFs), which can encode and decode without a loss of information from a valid collection of code units.

using **is_encode_injective** = ::std::true_type
Whether or not the encode operation can process all forms of input into code unit values. This is true for all Unicode Transformation Formats (UTFs), which can encode and decode without loss of information from a valid input code point.

Public Static Functions

static inline constexpr auto **encode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *encode_state* &__s)
Encodes a single complete unit of information as code units and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code points from.
- **__output** – **[in]** The output view to write code units into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::encode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

static inline constexpr auto **decode_one**(_InputRange &&__input, _OutputRange &&__output, _ErrorHandler &&__error_handler, *decode_state* &__s)

Decodes a single complete unit of information as code points and produces a result with the input and output ranges moved past what was successfully read and written; or, produces an error and returns the input and output ranges untouched.

Remark To the best ability of the implementation, the iterators will be returned untouched (e.g., the input models at least a view and a forward_range). If it is not possible, returned ranges may be incremented even if an error occurs due to the semantics of any view that models an input_range.

Parameters

- **__input** – **[in]** The input view to read code uunits from.
- **__output** – **[in]** The output view to write code points into.
- **__error_handler** – **[in]** The error handler to invoke if encoding fails.
- **__s** – **[inout]** The necessary state information. For this encoding, the state is empty and means very little.

Returns A *ztd::text::decode_result* object that contains the reconstructed input range, reconstructed output range, error handler, and a reference to the passed-in state.

Public Static Attributes

static constexpr ::std::size_t **max_code_points**

The maximum number of code points a single complete operation of decoding can produce. This is 1 for all Unicode Transformation Format (UTF) encodings.

static constexpr ::std::size_t **max_code_units**

The maximum code units a single complete operation of encoding can produce. If overlong sequence allowed, this is 6: otherwise, this is 4.

1.9.4 Error Handlers

assume_valid_handler

The `assume_valid_handler` is a Undefined-Behavior invoking error handler. If an error is encountered, the encoding can legally ignore and never, ever call the error handler at all. This can invoke Undefined Behavior on malformed input.

Warning: This should **only ever be used on the most trusted of input, ever, and that input should never come from a source that is a user or connected to ANY external input sources such as the Network, Shared Pipe, Inter-Procedural Call, or similar**.

Implementers of encodings within templates can check for a potentially ignorable error handler like this one using *ztd::text::is_ignorable_error_handler_v*.

constexpr *assume_valid_handler_t* ztd::text::**assume_valid_handler** = {}
An instance of the *assume_valid_handler_t* type for ease of use.

class ztd::text::**assume_valid_handler_t**
An error handler that tells an encoding that it will pass through any errors, without doing any adjustment, correction or checking.

Remark This error handler is useful in conjunction with a ztd::text::ranges::unbounded_view for the fastest possible encoding and decoding in a general sense. However: IT IS ALSO EXTREMELY DANGEROUS AND CAN INVOKE UNDEFINED BEHAVIOR IF YOUR TEXT IS, IN FACT, MESSED UP. PLEASE DO NOT USE THIS WITHOUT A GOOD REASON!

Public Types

using **assume_valid** = ::std::integral_constant<bool, false>
A type that is true when calling code can not call this function and ignore it, and false when it cannot ignore it. See *ztd::text::assume_valid_handler_t* for details.

Public Functions

template<typename **_Encoding**, typename **_Result**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr auto **operator()**(const *_Encoding*&, *_Result* __result, const *_InputProgress*&, const *_OutputProgress*&) const
A handler for either decode or encode results that simply passes the result type back through with no changes made.

Parameters **__result** – **[in]** The current state of the encode operation to pass through.

default_handler

The default handler for all operations. A class type that simply wraps *ztd::text::replacement_handler_t* unless configured otherwise. You can change it to throw by default (NOT recommended) by using *ZTD_TEXT_DEFAULT_HANDLER_THROWS*.

Using this type, implicitly or explicitly, signals to ztd.text that you would like it to gently admonish you if any part of a conversion could be potentially lossy (valid data is put in, but it cannot be handled by the desired encode/decode/transcode operation).

constexpr *default_handler_t* ztd::text::**default_handler** = {}
An instance of the *default_handler_t* type for ease of use.

class ztd::text::**default_handler_t** : private *replacement_handler_t*
The default error handler for the entire library. Can be configured to use different strategies at build time. Without configuration, it defaults to the *ztd::text::replacement_handler_t*.

Public Types

using **error_handler** = __error_handler_base_t
The underlying error handler type.

Private Functions

template<typename **_Encoding**, typename **_InputRange**, typename **_OutputRange**, typename **_State**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr auto **operator()**(const *_Encoding* &__encoding, *encode_result*<*_InputRange*, *_OutputRange*, *_State*> __result, const *_InputProgress*&, const *_OutputProgress*&) const noexcept
The function call for inserting replacement code units at the point of failure, before returning flow back to the caller of the encode operation.

Parameters

- **__encoding** – **[in]** The Encoding that experienced the error.
- **__result** – **[in]** The current state of the encode operation.

template<typename **_Encoding**, typename **_InputRange**, typename **_OutputRange**, typename **_State**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr auto **operator()**(const *_Encoding* &__encoding, *decode_result*<*_InputRange*, *_OutputRange*, *_State*> __result, const *_InputProgress*&, const *_OutputProgress*&) const noexcept
The function call for inserting replacement code points at the point of failure, before returning flow back to the caller of the decode operation.

Parameters

- **__encoding** – **[in]** The Encoding that experienced the error.
- **__result** – **[in]** The current state of the encode operation.

incomplete_handler

This error handler takes the `ztd::text::encoding_error::incomplete_sequence` error and uses it to read from the provided "progress" contiguous range provided as the third parameter to any error handler. This can be helpful in situations here incomplete input is not to be interpreted as an error, such as in situations with networking stacks, I/O reads (particularly, non-recoverable streams like `std::istream_iterators` or `std::ostream_iterators`), and other such storage cases.

The data read but not used from an incomplete error during encode and decode operations is stored in the `incomplete_handler` object and can be accessed VIA the `code_points` and `code_units` functions.

template<typename **_Encoding**, typename **_ErrorHandler** = *default_handler_t*>

class `ztd::text::`**incomplete_handler** : private ebco<*_ErrorHandler*>
This handler detects if the error code is an incomplete seqence, and sets the error code to being okay before returning.

Remark This type is often useful in conjunction with an accumulation state or buffer, which can be very handy for I/O (e.g., Networking) operations.

tparam _Encoding The encoding type which dictates the *code_unit* and *code_point* buffers to store in the handler to catch unused input from the last parameter of error handler invocations by the encoding.

tparam _ErrorHandler An error handler to invoke if the encoding error code is **NOT** an incomplete sequence.

Public Types

using **error_handler** = *_ErrorHandler*
The underlying error handler type.

Public Functions

inline constexpr **incomplete_handler**()
noexcept(::std::is_nothrow_default_constructible_v<__error_handler_base_t>)
Constructs a *ztd::text::incomplete_handler* with a default-constructed internal error handler.

inline constexpr **incomplete_handler**(const *_ErrorHandler* &__error_handler)
noexcept(::std::is_nothrow_constructible_v<__error_handler_base_t, const *_ErrorHandler*&>)
Constructs a *ztd::text::incomplete_handler* with the provided internal error handler object.

Parameters **__error_handler** – The provided error handler object to copy in and use when the error is not an incomplete error.

inline constexpr **incomplete_handler**(*_ErrorHandler* &&__error_handler)
noexcept(::std::is_nothrow_constructible_v<__error_handler_base_t, *_ErrorHandler*&&>)
Constructs a *ztd::text::incomplete_handler* with the provided internal error handler object.

Parameters **__error_handler** – The provided error handler object to move in and use when the error is not an incomplete error.

inline constexpr *_ErrorHandler* &**base**() & noexcept
Returns the base error handler that is called when a non-incomplete error occurs.

inline constexpr const *_ErrorHandler* &**base**() const & noexcept
Returns the base error handler that is called when a non-incomplete error occurs.

inline constexpr *_ErrorHandler* &&**base**() && noexcept
Returns the base error handler that is called when a non-incomplete error occurs.

template<typename **_Result**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr auto **operator()**(const *_Encoding* &__encoding, *_Result* __result, const *_InputProgress* &__input_progress, const *_OutputProgress* &__output_progress) const & noexcept(::std::is_nothrow_invocable_v<*_ErrorHandler*, const *_Encoding*&, *_Result*&&, const *_InputProgress*&, const *_OutputProgress*&>)
Checks if the __result.error_code is *ztd::text::encoding_error::incomplete_sequence*, it saves the values from **__progress** and returns. Otherwise, invokes the provided error handler this object was constructed with.

Parameters

- **__encoding** – **[in]** The Encoding that experienced the error.
- **__result** – **[in]** The current state of the encode operation.

- **__input_progress** – **[in]** Any code units or code points that were read but not yet used before the failure occurred. These will be stored in this handler.
- **__output_progress** – **[in]** Any code points or code units that have not yet been written before the failure occurred. These will be stored in this handler.

template<typename **_Result**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr auto **operator()**(const *_Encoding* &__encoding, *_Result* __result, const *_InputProgress* &__input_progress, const *_OutputProgress* &__output_progress) & noexcept(::std::is_nothrow_invocable_v<*_ErrorHandler*, const *_Encoding*&, *_Result*&&, const *_InputProgress*&, const *_OutputProgress*&>)

Checks if the __result.error_code is *ztd::text::encoding_error::incomplete_sequence*, it saves the values from `__progress` and returns. Otherwise, invokes the provided error handler this object was constructed with.

Parameters

- **__encoding** – **[in]** The Encoding that experienced the error.
- **__result** – **[in]** The current state of the encode operation.
- **__input_progress** – **[in]** Any code units or code points that were read but not yet used before the failure occurred. These will be stored in this handler.
- **__output_progress** – **[in]** Any code points or code units that have not yet been written before the failure occurred. These will be stored in this handler.

template<typename **_Result**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr auto **operator()**(const *_Encoding* &__encoding, *_Result* __result, const *_InputProgress* &__input_progress, const *_OutputProgress* &__output_progress) && noexcept(::std::is_nothrow_invocable_v<*_ErrorHandler*, const *_Encoding*&, *_Result*&&, const *_InputProgress*&, const *_OutputProgress*&>)

Checks if the __result.error_code is *ztd::text::encoding_error::incomplete_sequence*, it saves the values from `__progress` and returns. Otherwise, invokes the provided error handler this object was constructed with.

Parameters

- **__encoding** – **[in]** The Encoding that experienced the error.
- **__result** – **[in]** The current state of the encode operation.
- **__input_progress** – **[in]** Any code units or code points that were read but not yet used before the failure occurred. These will be stored in this handler.
- **__output_progress** – **[in]** Any code points or code units that have not yet been written before the failure occurred. These will be stored in this handler.

inline ::ztd::span<_CodeUnit> **code_units**() const noexcept

Returns the code units from the last incomplete decode operations.

inline ::ztd::span<_CodePoint> **code_points**() const noexcept

Returns the code points from the last incomplete encode operations.

pass_handler

The `pass_handler` does exactly what its name implies: it passes the error as generated by the encoding object through without touching it. Unlike *ztd::text::assume_valid_handler*, this one does not invoke undefined behavior because it does not meet the *ztd::text::is_ignorable_error_handler* traits.

constexpr *pass_handler_t* `ztd::text::`**`pass_handler`** = {}
: An instance of *pass_handler_t* for ease of use.

class **`pass_handler_t`** : public __txt_detail::__pass_through_handler_with<false>
: An error handler that tells an encoding that it will pass through any errors, without doing any adjustment, correction or checking. Does not imply it is ignorable, unlike *ztd::text::assume_valid_handler_t* which can invoke UB if an error occurs.

replacement_handler

The `replacement_handler_t` is the go-to error handling class. It is also the *ztd::text::default_handler* unless configured otherwise.

Replacement works by using several different hooks on the provided encoding objects, or by falling back to some defaults if certain conditions are met. The user-controllable hooks are:

- `encoding.replacement_code_units(...)`, a function (which can be `static` or `constexpr`) that returns a range of code units to insert directly into an output stream on a failed `encode` operation. It can also be called as a secondary backup if an `decode` operation fails, whereupon it will use the values in the range to attempt `decode`ing them into the output if possible. It can be empty, to indicate that nothing is to be inserted.
- `encoding.replacement_code_points(...)`, a function (which can be `static` or `constexpr`) that returns a range of code points to insert directly into an output stream on a failed `decode` operation. It can also be called as a secondary backup if an `encode` operation fails, whereupon it will use the values in the range to attempt `encode`ing them into the output if possible. It can be empty, to indicate that nothing is to be inserted.
- `encoding.maybe_replacement_code_units(...)`, a function (which can be `static` or `constexpr`) that returns a maybe-range. If the expression `if (maybe_returned_range)` evaluates to `true`, it will get the range returned by the function by performing a dereference of `decltype(auto) returned_range = *maybe_returned_range;`. If the conditional expression does not evaluate to `true`, it will assume that nothing can be returned from the function. This is useful for runtime-only encodings or encodings that wrap other encodings and may not have a replacement function. The dereferenced returned range is used exactly as its non-`maybe` counterpart.
- `encoding.maybe_replacement_code_points(...)`, a function (which can be `static` or `constexpr`) that returns a maybe-range. If the expression `if (maybe_returned_range)` evaluates to `true`, it will get the range returned by the function by performing a dereference of `decltype(auto) returned_range = *maybe_returned_range;`. If the conditional expression does not evaluate to `true`, it will assume that nothing can be returned from the function. This is useful for runtime-only encodings or encodings that wrap other encodings and may not have a replacement function. The dereferenced returned range is used exactly as its non-`maybe` counterpart.

Each replacement handler can take the current `encode_state`/`decode_state` parameter for its desired operation, if it so chooses. This will allow replacements to hook into the statefulness of any given encoding operation. It fill first call `replacement_code_units(state)` first, if it's well-formed. Otherwise, it will call `replacement_code_units()`. It will do this with each of the 4 replacement functions mentioned above.

constexpr *replacement_handler_t* `ztd::text::`**`replacement_handler`** = {}
: A convenience variable for passing the *replacement_handler_t* handler to functions.

class ztd::text::**replacement_handler_t**
An error handler that replaces bad code points and code units with a chosen code point / code unit sequence.

Remark This class hooks into the encodings passed as the first parameter to the error handling functions to see if they define either `replacement_code_points()` or `replacement_code_units()` function. If so, they will call them and use the returned contiguous range to isnert code points or code units into the function. If neither of these exist, then it checks for a definition of a `maybe_replacement_code_points()` or a `maybe_replacement_code_units()` function. If either is present, they are expected to return a `std::optional` of a contiguous range. If it is engaged (the `std::optional` is filled) it will be used. Otherwise, if it is not engaged, then it will explicitly fall back to attempt to insert the default replacement character U+FFFD (`U''`) or `?` character. If the output is out of room for the desired object, then nothing will be inserted at all.

Subclassed by *default_handler_t*

Public Functions

template<typename **_Encoding**, typename **_InputRange**, typename **_OutputRange**, typename **_State**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr auto **operator()**(const *_Encoding* &__encoding, *encode_result*<*_InputRange*, *_OutputRange*, *_State*> __result, const *_InputProgress*&, const *_OutputProgress*&) const noexcept
The function call for inserting replacement code units at the point of failure, before returning flow back to the caller of the encode operation.

Parameters

- **__encoding** – **[in]** The Encoding that experienced the error.
- **__result** – **[in]** The current state of the encode operation.

template<typename **_Encoding**, typename **_InputRange**, typename **_OutputRange**, typename **_State**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr auto **operator()**(const *_Encoding* &__encoding, *decode_result*<*_InputRange*, *_OutputRange*, *_State*> __result, const *_InputProgress*&, const *_OutputProgress*&) const noexcept
The function call for inserting replacement code points at the point of failure, before returning flow back to the caller of the decode operation.

Parameters

- **__encoding** – **[in]** The Encoding that experienced the error.
- **__result** – **[in]** The current state of the encode operation.

throw_handler

The `throw_handler` simply throws a *ztd::text::encoding_error* as an exception. This should only EVER be used for pre-verified, trusted input sources, debugging purposes, or similar; malformed text is a common enough occurrence that throwing errors by default or using this handler by default is a **bad idea** in almost every way.

Throwing on encoding, decoding, and other errors can easily result in Denial of Service target points if this is used in conjunction with user or untrusted input sources.

constexpr *throw_handler_t* `ztd::text::`**throw_handler** = {}
An instance of *throw_handler_t* for ease of use.

class `ztd::text::`**throw_handler_t**
An error handler that throws on any encode operation failure.

Remark This class absolutely should not be used unless the user is prepared to handle spurious failure, especially for text processing that deals with input vectors. This can result in many exceptions being thrown, which for resource-intensive applications could cause issues and result in Denial of Service by way of repeated, unhandled, and unexpected failure.

Public Functions

template<typename **_Encoding**, typename **_InputRange**, typename **_OutputRange**, typename **_State**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr *encode_result*<*_InputRange*, *_OutputRange*, *_State*> **operator()**(const *_Encoding*&, *encode_result*<*_InputRange*, *_OutputRange*, *_State*> __result, const *_InputProgress*&, const *_OutputProgress*&) const noexcept(false)

Throws a ztd::text::encoding_error as an exception on an encode failure.

template<typename **_Encoding**, typename **_InputRange**, typename **_OutputRange**, typename **_State**, typename **_InputProgress**, typename **_OutputProgress**>
inline constexpr *decode_result*<*_InputRange*, *_OutputRange*, *_State*> **operator()**(const *_Encoding*&, *decode_result*<*_InputRange*, *_OutputRange*, *_State*> __result, const *_InputProgress*&, const *_OutputProgress*&) const noexcept(false)

Throws a ztd::text::encoding_error code as an exception on a decode failure.

1.9.5 Conversion and Counting Functions

count_as_decoded

`ztd::text::count_as_decoded` is a function that takes an input sequence of `code_units` and attempts to count them, according to the error handler that is given. Because the error handler is included as part of the function call (and is provided by default is one is not passed in), the count operation will also continue to count if the error handler sets the `error_code` member of the result to `ztd::text::encoding_error::ok` but still performs some action. This is, for example, the case with *ztd::text::replacement_handler_t* - output replacement code units or code points will be counted as part of the final count and returned with `result.error_code == ztd::text::encoding_error::ok`. You can differentiate error-less text from non-error text by checking `result.errors_were_handled()`, which will be true if the error handler is called regardless of whether or not the error handler "smooths" the problem over by inserting replacement characters, doing nothing, or otherwise.

The overloads of this function increase the level of control you have with each passed argument. At the last overload with four arguments, the function attempts to work call some extension points or falls back to the base function call in this order:

- The `text_count_as_decoded(input, encoding, handler, state)` extension point, if possible.
- An internal, implementation-defined customization point.
- The `basic_count_as_decoded` base function.

The base function call, `basic_count_as_decoded`, simply performs the *core counting loop* using the *Lucky 7* design.

During the `basic_count_as_decoded` loop, if it detects that there is a preferable `text_count_as_decoded_one`, it will call that method as `text_count_as_decoded_one(input, encoding, handler, state)` inside of the loop rather than doing the core design.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `decode_one` function on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_count_as_decoded` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**, typename **_State**>
constexpr auto **basic_count_as_decoded**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler, *_State* &__state)

Counts the number of code units that will result from attempting a decode operation.

Remark This method does not call ADL extension points. It attempts a combination of implementation techniques to count code units, with a loop over the `.decode` call into an intermediate, unseen buffer being the most basic guaranteed implementation attempt.

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code points there are.
- **__encoding** – **[in]** The encoding to count the input with.
- **__error_handler** – **[in]** The error handler to invoke when an encode operation fails.
- **__state** – **[inout]** The state that will be used to count code units.

Returns A *ztd::text::count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*) and a reference to the provided `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**, typename **_State**>
constexpr auto **count_as_decoded**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler, *_State* &__state)

Counts the number of code units that will result from attempting a decode operation.

Remark This method will first check if an ADL Extension Point `text_count_as_decoded` is callable with the given arguments. If it is, then that method will be used to do the work after forwarding all four arguments to that function call. Otherwise, it defers to ztd::text::basic_count_as_decoded.

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code points there are.
- **__encoding** – **[in]** The encoding to count the input with.
- **__error_handler** – **[in]** The error handler to invoke when an encode operation fails.
- **__state** – **[inout]** The state that will be used to count code units.

Returns A *ztd::text::count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*) and a reference to the provided `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**>
constexpr auto **count_as_decoded**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler)

Counts the number of code units that will result from attempting a decode operation.

Remark Calls ztd::text::count_as_decoded(Input, Encoding, ErrorHandler, State) with an `state` that is created by ztd::text::make_decode_state(Encoding).

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code points there are.
- **__encoding** – **[in]** The encoding to count the input with.
- **__error_handler** – **[in]** The error handler to invoke when an encode operation fails.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

template<typename **_Input**, typename **_Encoding**>
constexpr auto **count_as_decoded**(*_Input* &&__input, *_Encoding* &&__encoding)

Counts the number of code units that will result from attempting a decode operation.

Remark Calls ztd::text::count_as_decoded(Input, Encoding, ErrorHandler) with an `error_handler` that is similar to *ztd::text::default_handler_t*.

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code points there are.
- **__encoding** – **[in]** The encoding to count the input with.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

template<typename **_Input**>

constexpr auto **count_as_decoded**(*_Input* &&__input)
Counts the number of code units that will result from attempting a decode operation.

Remark Calls ztd::text::count_as_decoded(Input, Encoding) with an `encoding` that is derived from *ztd::text::default_code_unit_encoding*.

Parameters `__input` – **[in]** The input range (of code units) to find out how many code points there are.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

count_as_encoded

`ztd::text::count_as_encoded` is a function that takes an input sequence of `code_point`s and attempts to count them, according to the error handler that is given. Because the error handler is included as part of the function call (and is provided by default is one is not passed in), the count operation will also continue to count if the error handler sets the `error_code` member of the result to `ztd::text::encoding_error::ok` but still performs some action. This is, for example, the case with *ztd::text::replacement_handler* - output replacement code units or code points will be counted as part of the final count and returned with `result.error_code == ztd::text::encoding_error::ok`. You can differentiate error-less text from non-error text by checking `result.errors_were_handled()`, which will be true if the error handler is called regardless of whether or not the error handler "smooths" the problem over by inserting replacement characters, doing nothing, or otherwise.

The overloads of this function increase the level of control you have with each passed argument. At the last overload with four arguments, the function attempts to work call some extension points or falls back to the base function call in this order:

- The `text_count_as_encoded(input, encoding, handler, state)` extension point, if possible.
- An internal, implementation-defined customization point.
- The `basic_count_as_encoded` base function.

The base function call, `basic_count_as_encoded`, simply performs the *core counting loop* using the *Lucky 7* design.

During the `basic_count_as_encoded` loop, if it detects that there is a preferable `text_count_as_encoded_one`, it will call that method as `text_count_as_encoded_one(input, encoding, handler, state)` inside of the loop rather than doing the core design.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `encode_one` function on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_count_as_encoded` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**, typename **_State**>
constexpr auto **basic_count_as_encoded**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler, *_State* &__state)

Counts the number of code units that will result from attempting an encode operation on the input code points.

Remark This method will not check any ADL extension points. A combination of implementation techniques will be used to count code units, with a loop over the `.encode` call into an intermediate, unseen buffer being the most basic choice.

Parameters

- **__input** – **[in]** The input range (of code points) to find out how many code units there are.
- **__encoding** – **[in]** The encoding to count the input with.
- **__error_handler** – **[in]** The error handler to invoke when an encode operation fails.
- **__state** – **[inout]** The state that will be used to count code units.

Returns A *ztd::text::count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*) and a reference to the provided `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**, typename **_State**>
constexpr auto **count_as_encoded**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler, *_State* &__state)

Counts the number of code units that will result from attempting an encode operation on the input code points.

Remark This method will first check if an ADL Extension Point `text_count_as_encoded` is callable with the given arguments. If it is, then that method will be used to do the work after forwarding all four arguments to that function call. Otherwise, this defers to ztd::text::basic_count_as_encoded.

Parameters

- **__input** – **[in]** The input range (of code points) to find out how many code units there are.
- **__encoding** – **[in]** The encoding to count the input with.
- **__error_handler** – **[in]** The error handler to invoke when an encode operation fails.
- **__state** – **[inout]** The state that will be used to count code units.

Returns A *ztd::text::count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*) and a reference to the provided `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**>
constexpr auto **count_as_encoded**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler)

Counts the number of code units that will result from attempting an encode operation on the input code points.

Remark This method will call ztd::text::count_as_encoded(Input, Encoding, ErrorHandler, State) with an `state` created by ztd::text::make_encode_state(Encoding).

Parameters

- **__input** – **[in]** The input range (of code points) to find out how many code units there are.
- **__encoding** – **[in]** The encoding to count the input with.
- **__error_handler** – **[in]** The error handler to invoke when an encode operation fails.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

template<typename **_Input**, typename **_Encoding**>
constexpr auto **count_as_encoded**(*_Input* &&__input, *_Encoding* &&__encoding)

Counts the number of code units that will result from attempting an encode operation.

Remark This method will call ztd::text::count_as_encoded(Input, Encoding, ErrorHandler) by creating an `error_handler` similar to *ztd::text::default_handler_t*.

Parameters

- **__input** – **[in]** The input range (of code points) to find out how many code units there are.
- **__encoding** – **[in]** The encoding to count the input with.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

template<typename **_Input**>
constexpr auto **count_as_encoded**(*_Input* &&__input)

Counts the number of code units that will result from attempting an encode operation on the input code points.

Remark Calls ztd::text::count_as_encoded(Input, Encoding) with an `encoding` that is derived from *ztd::text::default_code_unit_encoding*.

Parameters **__input** – **[in]** The input range (of code points) to find out how many code units there are.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

count_as_transcoded

`ztd::text::count_as_transcoded` is a function that takes an input sequence of `code_units` and attempts to count them, according to the error handler that is given. Because the error handler is included as part of the function call (and is provided by default is one is not passed in), the count operation will also continue to count if the error handler sets the `error_code` member of the result to `ztd::text::encoding_error::ok` but still performs some action. This is, for example, the case with *ztd::text::replacement_handler_t* - output replacement code units or code points will be counted as part of the final count and returned with `result.error_code == ztd::text::encoding_error::ok`. You can differentiate error-less text from non-error text by checking `result.errors_were_handled()`, which will be true if the error handler is called regardless of whether or not the error handler "smooths" the problem over by inserting replacement characters, doing nothing, or otherwise.

The overloads of this function increase the level of control you have with each passed argument. At the last overload with four arguments, the function attempts to work call some extension points or falls back to the base function call in this order:

- The `text_count_as_transcoded(input, from_encoding, to_encoding, from_handler, to_handler, from_state, to_state)` extension point, if possible.
- An internal, implementation-defined customization point.
- The `basic_count_as_transcoded` base function.

The base function call, `basic_count_as_transcoded`, simply performs the *core counting loop* using the *Lucky 7* design.

During the `basic_count_as_transcoded` loop, if it detects that there is a preferable `text_count_as_transcoded_one`, it will call that method as `text_count_as_transcoded_one(input, encoding, handler, state)` inside of the loop rather than doing the core design.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `decode_one` and `encode_one` functions on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_count_as_transcoded` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**, typename **_ToState**>
constexpr auto **basic_count_as_transcoded**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state, *_ToState* &__to_state)

Counts the number of code units that will result from attempting an transcode operation on the input code points.

Remark This method will not check any ADL extension points. A combination of implementation techniques will be used to count code units, with a loop over the `.encode_one` / `.decode_one` call into an intermediate, unseen buffer being the most basic choice.

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code units of the transcoded output there are.
- **__from_encoding** – **[in]** The encoding that is going to be used to decode the input into an intermediary output.
- **__to_encoding** – **[in]** The encoding that is going to be used to encode the intermediary output.

- **__from_error_handler** – **[in]** The error handler to invoke when an intermediary decoding operation fails.
- **__to_error_handler** – **[in]** The error handler to invoke when the final encoding operation fails.
- **__from_state** – **[inout]** The state attached to the `__from_encoding` that will be used for the intermediary decode step.
- **__to_state** – **[inout]** The state related to the `__to_encoding` that will be used for the final encoding step.

Returns A *ztd::text::count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*) and a reference to the provided `__from_state` and `__to_state` .

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**, typename **_ToState**>
constexpr auto **count_as_transcoded**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state, *_ToState* &__to_state)

Counts the number of code units that will result from attempting an transcode operation on the input code points.

Remark This method checks for the ADL extension point `text_count_as_transcoded` . It will be called if it is possible. Otherwise, this function will defer to ztd::text::basic_count_as_transcoded.

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code units of the transcoded output there are.
- **__from_encoding** – **[in]** The encoding that is going to be used to decode the input into an intermediary output.
- **__to_encoding** – **[in]** The encoding that is going to be used to encode the intermediary output.
- **__from_error_handler** – **[in]** The error handler to invoke when an intermediary decoding operation fails.
- **__to_error_handler** – **[in]** The error handler to invoke when the final encoding operation fails.
- **__from_state** – **[inout]** The state attached to the `__from_encoding` that will be used for the intermediary decode step.
- **__to_state** – **[inout]** The state related to the `__to_encoding` that will be used for the final encoding step.

Returns A *ztd::text::count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*) and a reference to the provided `__from_state` and `__to_state` .

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**>

constexpr auto **count_as_transcoded**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state)

Counts the number of code units that will result from attempting an transcode operation on the input code points.

Remark This method will call ztd::text::count_as_transcoded(input, from_encoding, to_encoding, from_error_handler, to_error_handler, from_state, to_state) with an `to_state` created by ztd::text::make_encode_state(to_encoding).

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code units of the transcoded output there are.
- **__from_encoding** – **[in]** The encoding that is going to be used to decode the input into an intermediary output.
- **__to_encoding** – **[in]** The encoding that is going to be used to encode the intermediary output.
- **__from_error_handler** – **[in]** The error handler to invoke when an intermediary decoding operation fails.
- **__to_error_handler** – **[in]** The error handler to invoke when the final encoding operation fails.
- **__from_state** – **[inout]** The state attached to the `__from_encoding` that will be used for the intermediary decode step.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**>
constexpr auto **count_as_transcoded**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler)

Counts the number of code units that will result from attempting an transcode operation on the input code points.

Remark This method will call ztd::text::count_as_transcoded(input, from_encoding, to_encoding, from_error_handler, to_error_handler, from_state) with an `from_state` created by ztd::text::make_decode_state(from_encoding).

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code units of the transcoded output there are.
- **__from_encoding** – **[in]** The encoding that is going to be used to decode the input into an intermediary output.
- **__to_encoding** – **[in]** The encoding that is going to be used to encode the intermediary output.
- **__from_error_handler** – **[in]** The error handler to invoke when the decode portion of the transcode operation fails.

- **__to_error_handler** – **[in]** The error handler to invoke when the encode portion of the transcode operation fails.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**>
constexpr auto **count_as_transcoded**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler)

Counts the number of code units that will result from attempting an transcode operation.

Remark This method will call ztd::text::count_as_transcoded(input, from_encoding, to_encoding, from_error_handler, to_error_handler) by creating an `to_error_handler` similar to *ztd::text::default_handler_t*.

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code units of the transcoded output there are.
- **__from_encoding** – **[in]** The encoding that is going to be used to decode the input into an intermediary output.
- **__to_encoding** – **[in]** The encoding that is going to be used to encode the intermediary output.
- **__from_error_handler** – **[in]** The error handler to invoke when the decode portion of the transcode operation fails.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**>
constexpr auto **count_as_transcoded**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding)

Counts the number of code units that will result from attempting an transcode operation.

Remark This method will call ztd::text::count_as_transcoded(Input, Encoding, ErrorHandler) by creating an `error_handler` similar to *ztd::text::default_handler_t*.

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code units of the transcoded output there are.
- **__from_encoding** – **[in]** The encoding that is going to be used to decode the input into an intermediary output.
- **__to_encoding** – **[in]** The encoding that is going to be used to encode the intermediary output.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

template<typename **_Input**, typename **_ToEncoding**>
constexpr auto **count_as_transcoded**(*_Input* &&__input, *_ToEncoding* &&__to_encoding)

Counts the number of code units that will result from attempting an transcode operation on the input code points.

Remark Calls ztd::text::count_as_transcoded(Input, Encoding) with an `encoding` that is derived from *ztd::text::default_code_unit_encoding*.

Parameters

- **__input** – **[in]** The input range (of code units) to find out how many code units of the transcoded output there are.
- **__to_encoding** – **[in]** The encoding that is going to be used to encode the input into an intermediary output.

Returns A *ztd::text::stateless_count_result* that includes information about how many code units are present, taking into account any invoked errors (like replacement from *ztd::text::replacement_handler_t*).

decode

The `decode` grouping of functions (`decode`, `decode_to`, and `decode_into`) perform the task of doing bulk decoding from an `input` of `code_units` to the encoding's `code_point` type.

Named Groups

There are 3 named functions for this behavior, and each function comes with several function overloads. Each named function produces increasingly more information, letting you opt into just how much information and control you'd like over the algorithm and behavior. The first one simply returns a container with the transformation applied, discarding much of the operation's result information. This is useful for quick, one-off conversions where you do not care about any errors and would rather let it be handled by the error handler. The second `_to` suffixed functions return a container within a `result` type that contains additional information. The final `_into` suffixed functions take an output range to write into, letting you explicitly control just how much space there is to write into as well as returning a detailed `result` type.

The return type for these function calls is one of:

- the desired output container (highest level);
- *ztd::text::decode_result* or *ztd::text::stateless_decode_result* with the desired output container embedded as the *.output* parameter (mid level); or,
- *ztd::text::decode_result* or *ztd::text::stateless_decode_result* returning just the input and output ranges (lowest level).

decode(...)

This is the highest level bulk function.

This set of function overloads takes the provided `input`, `encoding`, `handler` and `state` and produces an output container type. The default container will either be a `std::basic_string` of the `code_point` type, or a `std::vector` if it is not a known "character" type.

The container type can be specified by passing it as an explicit template parameter to this function, such as `ztd::text::decode<std::vector<char32_t>>("bark", ztd::text::ascii{});`. The output container is default constructed.

It will either call `push_back`/`insert` directly on the target container to fill it up, or serialize data to a temporary buffer (controlled by *ZTD_TEXT_INTERMEDIATE_TRANSCODE_BUFFER_BYTE_SIZE*) before then copying it into the desired output container through any available means (bulk `.insert`, repeated `.push_back`, or repeated single `.insert` with the `.cend()` iterator in that order).

This is the "fire and forget" version of the `decode` function, returning only the container and not returning any of the result or state information used to construct it.

decode_to(...)

This is the mid level bulk function.

This set of function overloads takes the provided `input`, `encoding`, `handler` and `state` and produces an output container type that is embedded within a *ztd::text::decode_result*, or a *ztd::text::stateless_decode_result*, depending on whether or not you called the version which takes a *ztd::text::decode_state_t<Encoding>* as a parameter or if it had to create one on the stack internally and discard it after the operation was finished.

The container type can be specified by passing it as an explicit template parameter to this function, such as `ztd::text::decode_to<std::u32string>("meow", ztd::text::ascii{});`. The output container is default constructed.

It will either call `push_back`/`insert` directly on the target container to fill it up, or serialize data to a temporary buffer (controlled by *ZTD_TEXT_INTERMEDIATE_TRANSCODE_BUFFER_BYTE_SIZE*) before then copying it into the desired output container through any available means (bulk `.insert`, repeated `.push_back`, or repeated single `.insert` with the `.cend()` iterator in that order).

If nothing goes wrong or the error handler lets the algorithm continue, `.input` on the result should be empty.

decode_into(...)

This is the lowest level bulk function.

This set of function overloads takes the provided `input`, `encoding`, `output`, `handler`, and `state` and writes data into the output range specified by `output`. The result is a *ztd::text::decode_result*, or a *ztd::text::stateless_decode_result*, depending on whether or not you called the version which takes a *ztd::text::decode_state_t<Encoding>* as a parameter or if it had to create one on the stack internally and discard it after the operation was finished.

It is up to the end-user to provide a suitably-sized output range for `output`, otherwise this operation may return with *ztd::text::encoding_error::insufficient_output*. for the `result`'s `error_code` member. The amount of space consumed can be determined by checking the `std::distance` between the `.begin()` of the original `output` parameter and the `.begin()` of the returned `.output` member. The result also has error information and an `.input` member for checking how much input was consumed.

If nothing goes wrong or the error handler lets the algorithm continue, `.input` on the result should be empty.

For Everything

All named functions have 4 overloads. Each of the "higher level" functions, at the end of their overload call chain, will call the lower-level `decode_into` to perform the work. The final `decode_into` call uses the following ordering of extension points into calling the base implementation:

- `text_decode_into(input, encoding, output, handler, state)`
- An internal, implementation-defined customization point.
- `basic_decode_into`

The base function call, `basic_decode_into`, simply performs the *core decode loop* using the *Lucky 7* design. This design also means minimal stack space is used, keeping the core algorithm suitable for resource-constrained devices.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `decode_one` function on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_decode_into` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_Encoding**, typename **_Output**, typename **_ErrorHandler**, typename **_State**>
constexpr auto **basic_decode_into**(*_Input* &&__input, *_Encoding* &&__encoding, *_Output* &&__output, *_ErrorHandler* &&__error_handler, *_State* &__state)

Converts the code units of the given `__input` view through the encoding to code points into the `__output` view.

Remark This function performs the bog-standard, basic loop for decoding. It talks to no ADL extension points.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.
- **__output** – **[in]** An output_view to write code points to as the result of the decode operation from the intermediate code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__state** – **[inout]** A reference to the associated state for the `__encoding` 's decode step.

Returns A *ztd::text::decode_result* object that contains references to `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_Output**, typename **_ErrorHandler**, typename **_State**>

constexpr auto **decode_into**(*_Input* &&__input, *_Encoding* &&__encoding, *_Output* &&__output, *_ErrorHandler* &&__error_handler, *_State* &__state)

Converts the code units of the given `__input` view through the encoding to code points into the `__output` view.

Remark This function detects whether or not the ADL extension point `text_decode` can be called with the provided parameters. If so, it will use that ADL extension point over the default implementation. Otherwise, it will loop over the two encodings and attempt to decode by repeatedly calling the encoding's required `decode_one` function.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.
- **__output** – **[in]** An output_view to write code points to as the result of the decode operation from the intermediate code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__state** – **[inout]** A reference to the associated state for the `__encoding` 's decode step.

Returns A *ztd::text::decode_result* object that contains references to `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_Output**, typename **_ErrorHandler**>
constexpr auto **decode_into**(*_Input* &&__input, *_Encoding* &&__encoding, *_Output* &&__output, *_ErrorHandler* &&__error_handler)

Converts the code units of the given `__input` view through the encoding to code points into the `__output` view.

Remark Creates a default `state` using ztd::text::make_decode_state.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.
- **__output** – **[in]** An output_view to write code points to as the result of the decode operation from the intermediate code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.

Returns A *ztd::text::stateless_decode_result* object that contains references to `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_Output**>
constexpr auto **decode_into**(*_Input* &&__input, *_Encoding* &&__encoding, *_Output* &&__output)

Converts the code units of the given `__input` view through the encoding to code points into the `__output` view.

Remark Creates a default `error_handler` that is similar to *ztd::text::default_handler_t*, but marked as careless.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.
- **__output** – **[in]** An output_view to write code points to as the result of the decode operation from the intermediate code units.

Returns A *ztd::text::stateless_decode_result* object that contains references to `__state`.

template<typename **_Input**, typename **_Output**>
constexpr auto **decode_into**(*_Input* &&__input, *_Output* &&__output)

Converts the code units of the given `__input` view through the encoding to code points into the `__output` view.

Remark Creates a default `encoding` by figuring out the `value_type` of the `__input`, then passing that type into ztd::text::default_code_point_encoding_t. That encoding is that used to decode the input code units, by default.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__output** – **[in]** An output_view to write code points to as the result of the decode operation from the intermediate code units.

Returns A *ztd::text::stateless_decode_result* object that contains references to `__state`.

template<typename **_OutputContainer**, typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**, typename **_State**>
constexpr auto **decode_to**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler, *_State* &__state)

Converts the code units of the given `__input` view through the encoding to code points the specified `_OutputContainer` type.

Remark This function detects creates a container of type `_OutputContainer` and uses a typical `std::back_inserter` or `std::push_back_inserter` to fill in elements as it is written to. The result is then returned, with the `.output` value put into the container.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__state** – **[inout]** A reference to the associated state for the `__encoding` 's decode step.

Returns A *ztd::text::decode_result* object that contains references to `__state` and an output of type `_OutputContainer`.

template<typename **_OutputContainer**, typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**>

constexpr auto **decode_to**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler)

Converts the code units of the given `__input` view through the encoding to code points the specified `_OutputContainer` type.

Remark This function creates a `state` using ztd::text::make_decode_state.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.

Returns A *ztd::text::stateless_decode_result* object whose output is of type `_OutputContainer`.

template<typename **_OutputContainer**, typename **_Input**, typename **_Encoding**>
constexpr auto **decode_to**(*_Input* &&__input, *_Encoding* &&__encoding)

Converts the code units of the given `__input` view through the encoding to code points the specified `_OutputContainer` type.

Remark This function creates a `handler` using *ztd::text::default_handler_t*, but marks it as careless.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.

Returns A *ztd::text::stateless_decode_result* object whose output is of type `_OutputContainer`.

template<typename **_OutputContainer**, typename **_Input**>
constexpr auto **decode_to**(*_Input* &&__input)

Converts the code units of the given `__input` view through the encoding to code points the specified `_OutputContainer` type.

Remark This function creates an `encoding` by using the `value_type` of the `__input` which is then passed through the *ztd::text::default_code_point_encoding* type to get the default desired encoding.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.

Returns A *ztd::text::stateless_decode_result* object whose output is of type `_OutputContainer`.

template<typename **_OutputContainer** = void, typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**, typename **_State**>

constexpr auto **decode**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler, *_State* &__state)

Converts the code units of the given `__input` view through the encoding to code points the specified `_OutputContainer` type.

Remark This function detects creates a container of type `_OutputContainer` and uses a typical `std::back_inserter` or `std::push_back_inserter` to fill in elements as it is written to.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__state** – **[inout]** A reference to the associated state for the `__encoding` 's decode step.

Returns An object of type `_OutputContainer` .

template<typename **_OutputContainer** = void, typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**>
constexpr auto **decode**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler)

Converts the code units of the given `__input` view through the encoding to code points the specified `_OutputContainer` type.

Remark This function creates a `state` using ztd::text::make_decode_state.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.

Returns An object of type `_OutputContainer` .

template<typename **_OutputContainer** = void, typename **_Input**, typename **_Encoding**>
constexpr auto **decode**(*_Input* &&__input, *_Encoding* &&__encoding)

Converts the code units of the given `__input` view through the encoding to code points the specified `_OutputContainer` type.

Remark This function creates a `handler` using *ztd::text::default_handler_t*, but marks it as careless.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.
- **__encoding** – **[in]** The encoding that will be used to decode the input's code points into output code units.

Returns An object of type _OutputContainer .

template<typename **_OutputContainer** = void, typename **_Input**>
constexpr auto **decode**(*_Input* &&__input)

Converts the code units of the given __input view through the encoding to code points the specified _OutputContainer type.

Remark This function creates an encoding by using the value_type of the __input which is then passed through the *ztd::text::default_code_point_encoding* type to get the default desired encoding.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce code points.

Returns An object of type _OutputContainer .

encode

The encode grouping of functions (encode, encode_to, and encode_into) perform the task of doing bulk decoding from an input of code_points to the encoding's code_unit type.

Named Groups

There are 3 named functions for this behavior, and each function comes with several function overloads. Each named function produces increasingly more information, letting you opt into just how much information and control you'd like over the algorithm and behavior. The first one simply returns a container with the transformation applied, discarding much of the operation's result information. This is useful for quick, one-off conversions where you do not care about any errors and would rather let it be handled by the error handler. The second _to suffixed functions return a container within a result type that contains additional information. The final _into suffixed functions take an output range to write into, letting you explicitly control just how much space there is to write into as well as returning a detailed result type.

The return type for these function calls is one of:

- the desired output container (highest level);
- *ztd::text::encode_result* or *ztd::text::stateless_encode_result* with the desired output container embedded as the *.output* parameter (mid level); or,
- *ztd::text::encode_result* or *ztd::text::stateless_encode_result* returning just the input and output ranges (lowest level).

encode(...)

This is the highest level bulk function.

This set of function overloads takes the provided `input`, `encoding`, `handler` and `state` and produces an output container type. The default container will either be a `std::basic_string` of the `code_unit` type, or a `std::vector` if it is not a known "character" type.

The container type can be specified by passing it as an explicit template parameter to this function, such as `ztd::text::encode<std::vector<std::byte>>(U"bark", ztd::text::utf16_be{});`. The output container is default constructed.

It will either call `push_back`/`insert` directly on the target container to fill it up, or serialize data to a temporary buffer (controlled by *ZTD_TEXT_INTERMEDIATE_TRANSCODE_BUFFER_BYTE_SIZE*) before then copying it into the desired output container through any available means (bulk `.insert`, repeated `.push_back`, or repeated single `.insert` with the `.cend()` iterator in that order).

This is the "fire and forget" version of the `encode` function, returning only the container and not returning any of the result or state information used to construct it.

encode_to(...)

This is the mid level bulk function.

This set of function overloads takes the provided `input`, `encoding`, `handler` and `state` and produces an output container type that is embedded within a *ztd::text::encode_result*, or a *ztd::text::stateless_encode_result*, depending on whether or not you called the version which takes a *ztd::text::encode_state_t<Encoding>* as a parameter or if it had to create one on the stack internally and discard it after the operation was finished.

The container type can be specified by passing it as an explicit template parameter to this function, such as `ztd::text::encode_to<std::string>(U"meow", ascii{});`. The output container is default constructed.

It will either call `push_back`/`insert` directly on the target container to fill it up, or serialize data to a temporary buffer (controlled by *ZTD_TEXT_INTERMEDIATE_TRANSCODE_BUFFER_BYTE_SIZE*) before then copying it into the desired output container through any available means (bulk `.insert`, repeated `.push_back`, or repeated single `.insert` with the `.cend()` iterator in that order).

If nothing goes wrong or the error handler lets the algorithm continue, `.input` on the result should be empty.

encode_into(...)

This is the lowest level bulk function.

This set of function overloads takes the provided `input`, `encoding`, `output`, `handler`, and `state` and writes data into the output range specified by `output`. The result is a *ztd::text::encode_result*, or a *ztd::text::stateless_encode_result*, depending on whether or not you called the version which takes a *ztd::text::encode_state_t<Encoding>* as a parameter or if it had to create one on the stack internally and discard it after the operation was finished.

It is up to the end-user to provide a suitably-sized output range for `output`, otherwise this operation may return with *ztd::text::encoding_error::insufficient_output*. for the `result`'s `error_code` member. The amount of space consumed can be determined by checking the `std::distance` between the `.begin()` of the original `output` parameter and the `.begin()` of the returned `.output` member. The result also has error information and an `.input` member for checking how much input was consumed.

If nothing goes wrong or the error handler lets the algorithm continue, `.input` on the result should be empty.

For Everything

All named functions have 4 overloads. Each of the "higher level" functions, at the end of their overload call chain, will call the lower-level `encode_into` to perform the work. The final `encode_into` call uses the following ordering of extension points into calling the base implementation:

- `text_encode_into(input, encoding, output, handler, state)`
- An internal, implementation-defined customization point.
- `basic_encode_into(input, encoding, output, handler, state)`

The final function call, `basic_encode_into`, simply performs the *core encode loop* using the *Lucky 7* design. This design also means minimal stack space is used, keeping the core algorithm suitable for resource-constrained devices.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `encode_one` function on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_encode_into` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_Encoding**, typename **_Output**, typename **_ErrorHandler**, typename **_State**>
constexpr auto **basic_encode_into**(*_Input* &&__input, *_Encoding* &&__encoding, *_Output* &&__output, *_ErrorHandler* &&__error_handler, *_State* &__state)

Converts the code points of the given `__input` view through the encoding to code units into the `__output` view.

Remark This function does not attempt to call any extension points for encoding. It simply uses the encoding and attempts to encode by repeatedly calling the encoding's required `encode_one` function.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__state** – **[inout]** A reference to the associated state for the `__encoding` 's encode step.

Returns A *ztd::text::encode_result* object that contains references to `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_Output**, typename **_ErrorHandler**, typename **_State**>

constexpr auto **encode_into**(*_Input* &&__input, *_Encoding* &&__encoding, *_Output* &&__output, *_ErrorHandler* &&__error_handler, *_State* &__state)

Converts the code points of the given `__input` view through the encoding to code units into the `__output` view.

Remark This function detects whether or not the ADL extension point `text_encode` can be called with the provided parameters. If so, it will use that ADL extension point over the default implementation. Otherwise, it will loop over the two encodings and attempt to encode by repeatedly calling the encoding's required `encode_one` function.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__state** – **[inout]** A reference to the associated state for the `__encoding` 's encode step.

Returns A *ztd::text::encode_result* object that contains references to `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_Output**, typename **_ErrorHandler**>
constexpr auto **encode_into**(*_Input* &&__input, *_Encoding* &&__encoding, *_Output* &&__output, *_ErrorHandler* &&__error_handler)

Converts the code points of the given `__input` view through the encoding to code units into the `__output` view.

Remark Creates a default `state` using ztd::text::make_encode_state.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.

Returns A *ztd::text::stateless_encode_result* object that contains references to `__state`.

template<typename **_Input**, typename **_Encoding**, typename **_Output**>
constexpr auto **encode_into**(*_Input* &&__input, *_Encoding* &&__encoding, *_Output* &&__output)

Converts the code points of the given `__input` view through the encoding to code units into the `__output` view.

Remark Creates a default `error_handler` that is similar to *ztd::text::default_handler_t*, but marked as careless.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.

Returns A *ztd::text::stateless_encode_result* object that contains references to `__state`.

template<typename **_Input**, typename **_Output**>
constexpr auto **encode_into**(*_Input* &&__input, *_Output* &&__output)

Converts the code points of the given `__input` view through the encoding to code units into the `__output` view.

Remark Creates a default `encoding` by figuring out the `value_type` of the `__input`, then passing that type into ztd::text::default_code_point_encoding_t. That encoding is that used to encode the input code points, by default.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.

Returns A *ztd::text::stateless_encode_result* object that contains references to `__state`.

template<typename **_OutputContainer**, typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**, typename **_State**>
constexpr auto **encode_to**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler, *_State* &__state)

Converts the code points of the given `__input` view through the encoding to code units in the specified `_OutputContainer` type.

Remark This function detects creates a container of type `_OutputContainer` and uses a typical `std::back_inserter` or `std::push_back_inserter` to fill in elements as it is written to. The result is then returned, with the `.output` value put into the container.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__state** – **[inout]** A reference to the associated state for the `__encoding` 's encode step.

Returns A *ztd::text::encode_result* object that contains references to `__state` and an output of type `_OutputContainer`.

template<typename **_OutputContainer**, typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**>

constexpr auto **encode_to**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler)

Converts the code points of the given `__input` view through the encoding to code units in the specified `_OutputContainer` type.

Remark This function creates a `state` using ztd::text::make_encode_state.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.

Returns A *ztd::text::stateless_encode_result* object whose output is of type `_OutputContainer`.

template<typename **_OutputContainer**, typename **_Input**, typename **_Encoding**>
constexpr auto **encode_to**(*_Input* &&__input, *_Encoding* &&__encoding)

Converts the code points of the given `__input` view through the encoding to code units in the specified `_OutputContainer` type.

Remark This function creates a `handler` using *ztd::text::default_handler_t*, but marks it as careless.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.

Returns A *ztd::text::stateless_encode_result* object whose output is of type `_OutputContainer`.

template<typename **_OutputContainer**, typename **_Input**>
constexpr auto **encode_to**(*_Input* &&__input)

Converts the code points of the given `__input` view through the encoding to code units in the specified `_OutputContainer` type.

Remark This function creates an `encoding` by using the `value_type` of the `__input` which is then passed through the *ztd::text::default_code_point_encoding* type to get the default desired encoding.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.

Returns A *ztd::text::stateless_encode_result* object whose output is of type `_OutputContainer`.

template<typename **_OutputContainer** = void, typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**, typename **_State**>

constexpr auto **encode**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler, *_State* &__state)

Converts the code points of the given `__input` view through the encoding to code units in the specified `_OutputContainer` type.

Remark This function detects creates a container of type `_OutputContainer` and uses a typical `std::back_inserter` or `std::push_back_inserter` to fill in elements as it is written to.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__state** – **[inout]** A reference to the associated state for the `__encoding` 's encode step.

Returns An object of type `_OutputContainer` .

template<typename **_OutputContainer** = void, typename **_Input**, typename **_Encoding**, typename **_ErrorHandler**>
constexpr auto **encode**(*_Input* &&__input, *_Encoding* &&__encoding, *_ErrorHandler* &&__error_handler)

Converts the code points of the given `__input` view through the encoding to code units in the specified `_OutputContainer` type.

Remark This function creates a `state` using ztd::text::make_encode_state.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.
- **__error_handler** – **[in]** The error handlers for the from and to encodings, respectively.

Returns An object of type `_OutputContainer` .

template<typename **_OutputContainer** = void, typename **_Input**, typename **_Encoding**>
constexpr auto **encode**(*_Input* &&__input, *_Encoding* &&__encoding)

Converts the code points of the given `__input` view through the encoding to code units in the specified `_OutputContainer` type.

Remark This function creates a `handler` using *ztd::text::default_handler_t*, but marks it as careless.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters

- **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.
- **__encoding** – **[in]** The encoding that will be used to encode the input's code points into output code units.

Returns An object of type _OutputContainer .

template<typename **_OutputContainer** = void, typename **_Input**>
constexpr auto **encode**(*_Input* &&__input)

Converts the code points of the given __input view through the encoding to code units in the specified _OutputContainer type.

Remark This function creates an encoding by using the value_type of the __input which is then passed through the *ztd::text::default_code_point_encoding* type to get the default desired encoding.

Template Parameters **_OutputContainer** – The container type to serialize data into.

Parameters **__input** – **[in]** An input_view to read code points from and use in the encode operation that will produce code units.

Returns An object of type _OutputContainer .

transcode

The transcode grouping of functions (transcode, transcode_to, and transcode_into) perform the task of doing bulk transcoding from an input of code_units to a second encoding's code_unit type. It expects to traffic through the code_point type as the intermediary between the two functions.

Named Groups

There are 3 named functions for this behavior, and each function comes with several function overloads. Each named function produces increasingly more information, letting you opt into just how much information and control you'd like over the algorithm and behavior. The first one simply returns a container with the transformation applied, discarding much of the operation's result information. This is useful for quick, one-off conversions where you do not care about any errors and would rather let it be handled by the error handler. The second _to suffixed functions return a container within a result type that contains additional information. The final _into suffixed functions take an output range to write into, letting you explicitly control just how much space there is to write into as well as returning a detailed result type.

The return type for these function calls is one of:

- the desired output container (highest level);
- *ztd::text::transcode_result* or *ztd::text::stateless_transcode_result* with the desired output container embedded as the *.output* parameter (mid level); or,
- *ztd::text::transcode_result* or *ztd::text::stateless_transcode_result* returning just the input and output ranges (lowest level).

transcode(...)

This is the highest level bulk function.

This set of function overloads takes the provided `input`, `from_encoding`, `to_encoding`, `from_handler`, `to_handler`, `from_state`, and `to_state` and produces an output container type. The default container will either be a `std::basic_string` of the `code_unit` type, or a `std::vector` if it is not a known "character" type.

The container type can be specified by passing it as an explicit template parameter to this function, such as `ztd::text::transcode<std::vector<char16_t>>("bark", ztd::text::utf16{});`. The output container is default constructed.

It will either call `push_back`/`insert` directly on the target container to fill it up, or serialize data to a temporary buffer (controlled by *ZTD_TEXT_INTERMEDIATE_TRANSCODE_BUFFER_BYTE_SIZE*) before then copying it into the desired output container through any available means (bulk `.insert`, repeated `.push_back`, or repeated single `.insert` with the `.cend()` iterator in that order).

This is the "fire and forget" version of the `transcode` function, returning only the container and not returning any of the result or state information used to construct it.

transcode_to(...)

This is the mid level bulk function.

This set of function overloads takes the provided `input`, `from_encoding`, `to_encoding`, `from_handler`, `to_handler`, `from_state`, and `to_state` and produces an output container type that is embedded within a *ztd::text::transcode_result*, or a *ztd::text::stateless_transcode_result*, depending on whether or not you called the version which takes a *ztd::text::decode_state_t<Encoding>* and/or a *ztd::text::encode_state_t<Encoding>* as a parameter or if it had to create one on the stack internally and discard it after the operation was finished.

The container type can be specified by passing it as an explicit template parameter to this function, such as `ztd::text::transcode_to<std::string>(U"meow", ascii{});`. The output container is default constructed.

It will either call `push_back`/`insert` directly on the target container to fill it up, or serialize data to a temporary buffer (controlled by *ZTD_TEXT_INTERMEDIATE_TRANSCODE_BUFFER_BYTE_SIZE*) before then copying it into the desired output container through any available means (bulk `.insert`, repeated `.push_back`, or repeated single `.insert` with the `.cend()` iterator in that order).

If nothing goes wrong or the error handler lets the algorithm continue, `.input` on the result should be empty.

transcode_into(...)

This is the lowest level bulk function.

This set of function overloads takes the provided `input`, `from_encoding`, `output`, `to_encoding`, `from_handler`, `to_handler`, `from_state`, and `to_state` to write data into an output range specified by `output`. The result is a *ztd::text::transcode_result*, or a *ztd::text::stateless_transcode_result*, depending on whether or not you called the version which takes a *ztd::text::decode_state_t<Encoding>* and/or a *ztd::text::encode_state_t<Encoding>* as a parameter or if it had to create one on the stack internally and discard it after the operation was finished.

It is up to the end-user to provide a suitably-sized output range for `output`, otherwise this operation may return with *ztd::text::encoding_error::insufficient_output*. for the `result`'s `error_code` member. The amount of space consumed can be determined by checking the `std::distance` between the `.begin()` of the original `output` parameter and the `.begin()` of the returned `.output` member. The result also has error information and an `.input` member for checking how much input was consumed.

If nothing goes wrong or the error handler lets the algorithm continue, `.input` on the result should be empty.

For Everything

All named functions have 6 overloads. Each of the "higher level" functions, at the end of their overload call chain, will call the lower-level `transcode_into` to perform the work. The final `transcode_into` call uses the following ordering of extension points into calling the base implementation:

- The `text_transcode_into(input, from_encoding, output, to_encoding, ...)` extension point.
- An implementation-defined extension point if any internal optimizations are possible.
- The `basic_transcode_into(input, from_encoding, output, to_encoding, ...)` function.

The final function call, `basic_transcode_into`, simply performs the *core transcode loop* using the *Lucky 7* design. `basic_transcode_into` accommodates the lowest level transformation using just `decode_one` into a suitably sized intermediate buffer and then an `encode_one` into the output, calling the relevant error handlers along the way. This design also means minimal stack space is used, keeping the core algorithm suitable for resource-constrained devices.

However, there is a caveat: if there exists a `text_transcode_one(input, from_encoding, output, to_encoding, ...)` that is callable then it will be called to perform one unit of complete transformation. Otherwise, `decode_one/encode_one`

The `transcode_one` extension point is also used in the *ztd::text::transcode_view<...>* to speed up one-by-one translations for iteration-based types, where possible.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `decode_one` function of the `from_encoding` and the `transcode_one` of the `to_encoding` function on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_transcode_into` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_Output**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**, typename **_ToState**>
constexpr auto **basic_transcode_into**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_Output* &&__output, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state, *_ToState* &__to_state)

Converts the code units of the given input view through the from encoding to code units of the to encoding into the output view.

Remark This function detects whether or not the ADL extension point `text_transcode` can be called with the provided parameters. If so, it will use that ADL extension point over the default implementation. Otherwise, it will loop over the two encodings and attempt to transcode by first decoding the input code units to code points, then encoding the intermediate code points to the desired, output code units.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.

- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__to_error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__from_state** – **[inout]** A reference to the associated state for the `__from_encoding` 's decode step.
- **__to_state** – **[inout]** A reference to the associated state for the `__to_encoding` 's encode step.

Returns A *ztd::text::transcode_result* object that contains references to `__from_state` and `__to_state`.

template<typename **_Input**, typename **_Output**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**, typename **_ToState**>
constexpr auto **transcode_into**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_Output* &&__output, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state, *_ToState* &__to_state)

Converts the code units of the given input view through the from encoding to code units of the to encoding into the output view.

Remark This function detects whether or not the ADL extension point `text_transcode` can be called with the provided parameters. If so, it will use that ADL extension point over the default implementation. Otherwise, it will loop over the two encodings and attempt to transcode by first decoding the input code units to code points, then encoding the intermediate code points to the desired, output code units.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__to_error_handler** – **[in]** The error handlers for the from and to encodings, respectively.
- **__from_state** – **[inout]** A reference to the associated state for the `__from_encoding` 's decode step.

- **__to_state** – **[inout]** A reference to the associated state for the `__to_encoding` 's encode step.

Returns A *ztd::text::transcode_result* object that contains references to `__from_state` and `__to_state`.

template<typename **_Input**, typename **_FromEncoding**, typename **_Output**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**>
constexpr auto **transcode_into**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_Output* &&__output, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state)

Converts the code units of the given input view through the from encoding to code units of the to encoding into the output view.

Remark This function calls the base reference, the ztd::text::transcode_into after creating a `to_state` from ztd::text::make_encode_state. The result from this function returns a *ztd::text::stateless_transcode_result* as opposed to a *ztd::text::transcode_result* because the state information is on the stack, and returning the state in those types by reference will result in references to memory that has already been cleaned up. If you need access to the state parameters, call the lower-level functionality with your own created states.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.
- **__to_error_handler** – **[in]** The error handler for the `__to_encoding` 's encode step.
- **__from_state** – **[inout]** A reference to the associated state for the `__from_encoding` 's decode step.

Returns A *ztd::text::stateless_transcode_result* object.

template<typename **_Input**, typename **_FromEncoding**, typename **_Output**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**>
constexpr auto **transcode_into**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_Output* &&__output, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler)

Converts the code units of the given input view through the from encoding to code units of the to encoding into the output view.

Remark This function creates a decode state `from_state` by calling ztd::text::make_decode_state. The result from this function returns a *ztd::text::stateless_transcode_result* as opposed to a *ztd::text::transcode_result* because the state information is on the stack, and returning the state in those types by reference will result

in references to memory that has already been cleaned up. If you need access to the state parameters, call the lower-level functionality with your own created states.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.
- **__to_error_handler** – **[in]** The error handler for the `__to_encoding` 's encode step.

template<typename **_Input**, typename **_FromEncoding**, typename **_Output**, typename **_ToEncoding**, typename **_FromErrorHandler**>
constexpr auto **transcode_into**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_Output* &&__output, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler)

Converts the code units of the given input view through the from encoding to code units of the to encoding into the output view.

Remark This function creates an `to_error_handler` from a class like *ztd::text::default_handler_t*, but that is marked as careless since you did not explicitly provide it. This matters for lossy conversions that are not injective. The result from this function returns a *ztd::text::stateless_transcode_result* as opposed to a *ztd::text::transcode_result* because the state information is on the stack, and returning the state in those types by reference will result in references to memory that has already been cleaned up. If you need access to the state parameters, call the lower-level functionality with your own created states.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.

template<typename **_Input**, typename **_FromEncoding**, typename **_Output**, typename **_ToEncoding**>
constexpr auto **transcode_into**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_Output* &&__output, *_ToEncoding* &&__to_encoding)

Converts the code units of the given input view through the from encoding to code units of the to encoding into the output view.

Remark This function creates an `from_error_handler` from a class like *ztd::text::default_handler_t*, but that is marked as careless since you did not explicitly provide it. This matters for lossy conversions that are not injective. The result from this function returns a *ztd::text::stateless_transcode_result* as opposed to a *ztd::text::transcode_result* because the state information is on the stack, and returning the state in those types by reference will result in references to memory that has already been cleaned up. If you need access to the state parameters, call the lower-level functionality with your own created states.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.

template<typename **_Input**, typename **_ToEncoding**, typename **_Output**>
constexpr auto **transcode_into**(*_Input* &&__input, *_ToEncoding* &&__to_encoding, *_Output* &&__output)

Converts the code units of the given input view through the from encoding to code units of the to encoding into the output view.

Remark This function creates both: a `from_error_handler` using a *ztd::text::default_handler_t* that is marked as careless to pass to the next function overload; and, a `from_encoding` to interpret the `__input` by checking the `__input` 's `value_type.` This matters for lossy conversions that are not injective. The result from this function returns a *ztd::text::stateless_transcode_result* as opposed to a *ztd::text::transcode_result* because the state information is on the stack, and returning the state in those types by reference will result in references to memory that has already been cleaned up. If you need access to the state parameters, call the lower-level functionality with your own created states.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__output** – **[in]** An output_view to write code units to as the result of the encode operation from the intermediate code points.

template<typename **_OutputContainer**, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**, typename **_ToState**>
constexpr auto **transcode_to**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state, *_ToState* &__to_state)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.
- **__to_error_handler** – **[in]** The error handler for the `__to_encoding` 's encode step.
- **__from_state** – **[inout]** A reference to the associated state for the `__from_encoding` 's decode step.
- **__to_state** – **[inout]** A reference to the associated state for the `__to_encoding` 's encode step.

Returns A *ztd::text::transcode_result* object that contains references to `__from_state` and `__to_state` and an `output` parameter that contains the `_OutputContainer` specified. If the container has a `container.reserve` function, it is and some multiple of the input's size is used to pre-size the container, to aid with `push_back` / `insert` reallocation pains.

template<typename **_OutputContainer**, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**>
constexpr auto **transcode_to**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark A default state for the encode step of the operation is create using ztd::text::make_encode_state. The return type is stateless since both states must be passed in. If you want to have access to the states, create both of them yourself and pass them into a lower-level function that accepts those parameters.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.
- **__to_error_handler** – **[in]** The error handler for the `__to_encoding` 's encode step.

- **__from_state** – **[inout]** A reference to the associated state for the `__from_encoding` 's decode step.

Returns A *ztd::text::stateless_transcode_result* object that contains references to an `container.output` parameter that contains the `_OutputContainer` specified.

template<typename **_OutputContainer**, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**>
constexpr auto **transcode_to**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark A default state for the decode step of the operation is create using ztd::text::make_decode_state. The return type is stateless since both states must be passed in. If you want to have access to the states, create both of them yourself and pass them into a lower-level function that accepts those parameters.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.
- **__to_error_handler** – **[in]** The error handler for the `__to_encoding` 's encode step.

Returns A *ztd::text::stateless_transcode_result* object that contains references to an `container.output` parameter that contains the `_OutputContainer` specified.

template<typename **_OutputContainer**, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**>
constexpr auto **transcode_to**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark A `to_error_handler` for the encode step of the operation is created using default construction of a *ztd::text::default_handler_t* that is marked as careless. The return type is stateless since both states must be passed in. If you want to have access to the states, create both of them yourself and pass them into a lower-level function that accepts those parameters.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.

Returns A *ztd::text::stateless_transcode_result* object that contains references to an `container.output` parameter that contains the `_OutputContainer` specified.

template<typename **_OutputContainer**, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**>
constexpr auto **transcode_to**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark A `from_error_handler` for the encode step of the operation is created using default construction of a *ztd::text::default_handler_t* that is marked as careless. The return type is stateless since both states must be passed in. If you want to have access to the states, create both of them yourself and pass them into a lower-level function that accepts those parameters.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.

Returns A *ztd::text::stateless_transcode_result* object that contains references to an `container.output` parameter that contains the `_OutputContainer` specified.

template<typename **_OutputContainer**, typename **_Input**, typename **_ToEncoding**>
constexpr auto **transcode_to**(*_Input* &&__input, *_ToEncoding* &&__to_encoding)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark A `from_error_handler` for the encode step of the operation is created using default construction of a *ztd::text::default_handler_t* that is marked as careless. The return type is stateless since both states must be passed in. If you want to have access to the states, create both of them yourself and pass them into a lower-level function that accepts those parameters.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.

Returns A *ztd::text::stateless_transcode_result* object that contains references to an `container.output` parameter that contains the `_OutputContainer` specified.

template<typename **_OutputContainer** = void, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**, typename **_ToState**>
constexpr auto **transcode**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state, *_ToState* &__to_state)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.
- **__to_error_handler** – **[in]** The error handler for the `__to_encoding` 's encode step.
- **__from_state** – **[inout]** A reference to the associated state for the `__from_encoding` 's decode step.
- **__to_state** – **[inout]** A reference to the associated state for the `__to_encoding` 's encode step.

Returns An `_OutputContainer` with the result, regardless of whether an error occurs or not. If you are looking for error information and not just a quick one-off conversion function, please use ztd::text::transcode_to or ztd::text::transcode_into.

template<typename **_OutputContainer** = void, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**, typename **_FromState**>
constexpr auto **transcode**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler, *_FromState* &__from_state)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark This function creates an `to_state` for the encoding step of the operation using ztd::text::make_encode_state.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.
- **__to_error_handler** – **[in]** The error handler for the `__to_encoding` 's encode step.
- **__from_state** – **[inout]** A reference to the associated state for the `__from_encoding` 's decode step.

Returns An `_OutputContainer` with the result, regardless of whether an error occurs or not. If you are looking for error information and not just a quick one-off conversion function, please use ztd::text::transcode_to or ztd::text::transcode_into.

template<typename **_OutputContainer** = void, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**, typename **_ToErrorHandler**>
constexpr auto **transcode**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler, *_ToErrorHandler* &&__to_error_handler)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark This function creates an `from_state` for the encoding step of the operation using ztd::text::make_decode_state.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.
- **__to_error_handler** – **[in]** The error handler for the `__to_encoding` 's encode step.

Returns An `_OutputContainer` with the result, regardless of whether an error occurs or not. If you are looking for error information and not just a quick one-off conversion function, please use ztd::text::transcode_to or ztd::text::transcode_into.

template<typename **_OutputContainer** = void, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_FromErrorHandler**>
constexpr auto **transcode**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_FromErrorHandler* &&__from_error_handler)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark This function creates a `to_error_handler` from a class like *ztd::text::default_handler_t*, but that is marked as careless since you did not explicitly provide it. This matters for lossy conversions that are not injective.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.
- **__from_error_handler** – **[in]** The error handler for the `__from_encoding` 's decode step.

Returns An `_OutputContainer` with the result, regardless of whether an error occurs or not. If you are looking for error information and not just a quick one-off conversion function, please use ztd::text::transcode_to or ztd::text::transcode_into.

template<typename **_OutputContainer** = void, typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**>
constexpr auto **transcode**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark This function creates a `from_error_handler` from a class like *ztd::text::default_handler_t*, but that is marked as careless since you did not explicitly provide it. This matters for lossy conversions that are not injective.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__from_encoding** – **[in]** The encoding that will be used to decode the input's code units into intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.

Returns An `_OutputContainer` with the result, regardless of whether an error occurs or not. If you are looking for error information and not just a quick one-off conversion function, please use ztd::text::transcode_to or ztd::text::transcode_into.

template<typename **_OutputContainer** = void, typename **_Input**, typename **_ToEncoding**>
constexpr auto **transcode**(*_Input* &&__input, *_ToEncoding* &&__to_encoding)

Converts the code units of the given input view through the from encoding to code units of the to encoding for the output, which is then returned in a result structure with additional information about success.

Remark This function creates both: a `from_error_handler` from a class like *ztd::text::default_handler_t*, but that is marked as careless since you did not explicitly provide it; and, a `from_encoding` derived from the `"__input"'s value_type.` The careless marking matters for lossy conversions that are not injective.

Template Parameters **_OutputContainer** – The container to default-construct and serialize data into. Typically, a `std::basic_string` or a `std::vector` of some sort.

Parameters

- **__input** – **[in]** An input_view to read code units from and use in the decode operation that will produce intermediate code points.
- **__to_encoding** – **[in]** The encoding that will be used to encode the intermediate code points into the final code units.

Returns An `_OutputContainer` with the result, regardless of whether an error occurs or not. If you are looking for error information and not just a quick one-off conversion function, please use ztd::text::transcode_to or ztd::text::transcode_into.

validate_decodable_as

`ztd::text::validate_decodable_as` is a function that takes an input sequence of `code_unit`s and attempts to validate that they can be turned into the `code_point`s of the provided encoding. Unlike the *ztd::text::count_as_decoded* function, this does not take an error handler. **Any** error, even if it would be corrected over, produces a stop in the algorithm and a *validate_result*/*stateless_validate_result* object gets returned with the `.valid` member set to false.

The overloads of this function increase the level of control with each passed argument. At the last overload with four arguments, the function attempts to work call some extension points or falls back to the base function call in this order:

- The `text_validate_decodable_as(input, encoding, decode_state)` extension point, if possible.
- The `text_validate_decodable_as(input, encoding, decode_state, encode_state)` extension point, if possible.
- An internal, implementation-defined customization point.
- The `basic_validate_decodable_as` base function.

The base function call, `basic_validate_decodable_as`, simply performs the *core validating loop* using the *Lucky 7* design. The reason the last overload takes 2 state arguments is due to how the base implementation works from the core validating loop. If during the 3-argument overload it is detected that `text_validate_decodable_as(input, encoding, decode_state)` can be called, it will be called without attempt to create an `encode_state` value with *ztd::text::make_encode_state(...)*.

During the `basic_validate_decodable_as` loop, if it detects that there is a preferable `text_validate_decodable_as_one`, it will call that method as `text_validate_decodable_as_one(input, encoding, decode_state)` inside of the loop rather than doing the core design.

The *ztd::text::validate_result* type only includes the `decode_state` in all cases.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `decode_one` and `encode_one` functions on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_validate_decodable_as` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_Encoding**, typename **_DecodeState**, typename **_EncodeState**>
constexpr auto **basic_validate_decodable_as**(*_Input* &&__input, *_Encoding* &&__encoding, *_DecodeState* &__decode_state, *_EncodeState* &__encode_state)

Validates the code units of the `__input` according to the `__encoding` with the given states `__decode_state` and `__encode_state` to see if it can be turned into code points.

Remark This function explicitly does not call any extension points. It defers to doing a typical loop over the code points to verify it can be decoded into code points, and then encoded back into code units, with no errors and with the exact same value sequence as the original.

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__decode_state** – **[in]** The state to use for the decoding portion of the validation check.
- **__encode_state** – **[in]** The state to use for the encoding portion of the validation check.

template<typename **_Input**, typename **_Encoding**, typename **_DecodeState**, typename **_EncodeState**>
constexpr auto **validate_decodable_as**(*_Input* &&__input, *_Encoding* &&__encoding, *_DecodeState* &__decode_state, *_EncodeState* &__encode_state)

Validates the code units of the `__input` according to the `__encoding` with the given states `__decode_state` and `__encode_state` to see if it can be turned into code points.

Remark This functions checks to see if extension points for `text_validate_decodable_as` is available taking the available 4 parameters. If so, it calls this. Otherwise, it defers to ztd::text::validate_decodable_as.

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__decode_state** – **[in]** The state to use for the decoding portion of the validation check.
- **__encode_state** – **[in]** The state to use for the encoding portion of the validation check.

template<typename **_Input**, typename **_Encoding**, typename **_DecodeState**>
constexpr auto **validate_decodable_as**(*_Input* &&__input, *_Encoding* &&__encoding, *_DecodeState* &__decode_state)

Validates the code units of the `__input` according to the `__encoding` with the given state `__decode_state` to see if it can be turned into code points.

Remark This functions checks to see if extension points for `text_validate_decodable_as` is available taking the available 3 parameters. If so, it calls this. Otherwise, it creates an encoding state through ztd::text::make_encode_state before calling ztd::text::validate_decodable_as(__input, __encoding, __decode_state, __encode_state).

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__decode_state** – **[in]** The state to use for the decoding portion of the validation check.

template<typename **_Input**, typename **_Encoding**>
constexpr auto **validate_decodable_as**(*_Input* &&__input, *_Encoding* &&__encoding)

Validates the code units of the `__input` according to the `__encoding` to see if they can be turned into code points.

Remark This functions creates an encoding `state` through ztd::text::make_decode_state before calling the next overload of ztd::text::validate_decodable_as.

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__encoding** – **[in]** The encoding to verify can properly encode the input of code units.

template<typename **_Input**>
constexpr auto **validate_decodable_as**(*_Input* &&__input)

Validates the code units of the `__input` to see if it can be turned into code points.

Remark This functions creates an `encoding` by passing the `value_type` of the `__input` range through *ztd::text::default_code_unit_encoding*.

Parameters **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.

Returns A *ztd::text::stateless_validate_result* detailing whether or not the input code points can be turned into code units of the corresponding encoding.

validate_encodable_as

`ztd::text::validate_encodable_as` is a function that takes an input sequence of `code_point`s and attempts to validate that they can be turned into the `code_unit`s of the provided encoding. Unlike the *ztd::text::count_as_encoded* function, this does not take an error handler. **Any** error, even if it would be corrected over, produces a stop in the algorithm and a *validate_result*/*stateless_validate_result* object gets returned with the `.valid` member set to false.

The overloads of this function increase the level of control with each passed argument. At the last overload with four arguments, the function attempts to work call some extension points or falls back to the base function call in this order:

- The `text_validate_encodable_as(input, encoding, encode_state)` extension point, if possible.
- The `text_validate_encodable_as(input, encoding, encode_state, decode_state)` extension point, if possible.
- An internal, implementation-defined customization point.
- The `basic_validate_encodable_as` base function.

The base function call, `basic_validate_encodable_as`, simply performs the *core validating loop* using the *Lucky 7* design. The reason the last overload takes 2 state arguments is due to how the base implementation works from the core validating loop. If during the 3-argument overload it is detected that `text_validate_encodable_as(input, encoding, encode_state)` can be called, it will be called without attempt to create an `decode_state` value with *ztd::text::make_decode_state(...)*.

During the `basic_validate_encodable_as` loop, if it detects that there is a preferable `text_validate_decodable_as_one`, it will call that method as `text_validate_encodable_as_one(input, encoding, encode_state)` inside of the loop rather than doing the core design.

The *ztd::text::validate_result* type only includes the `encode_state` in all cases.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `decode_one` and `encode_one` functions on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_validate_encodable_as` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_Encoding**, typename **_EncodeState**, typename **_DecodeState**>
constexpr auto **basic_validate_encodable_as**(*_Input* &&__input, *_Encoding* &&__encoding, *_EncodeState* &__encode_state, *_DecodeState* &__decode_state)

Validates the code points of the `__input` according to the `__encoding` with the given states `__encode_state` and `__decode_state`.

Remark This function explicitly does not check any of the extension points. It defers to doing a typical loop over the code points to verify it can be encoded into code units, and then decoded into code points, with no errors.

Parameters

- **__input** – **[in]** The input range of code points to validate is possible for encoding into code units.
- **__encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__encode_state** – **[in]** The state to use for the encoding portion of the validation check.
- **__decode_state** – **[in]** The state to use for the decoding portion of the validation check, if needed.

template<typename **_Input**, typename **_Encoding**, typename **_EncodeState**, typename **_DecodeState**>
constexpr auto **validate_encodable_as**(*_Input* &&__input, *_Encoding* &&__encoding, *_EncodeState* &__encode_state, *_DecodeState* &__decode_state)

Validates the code points of the `__input` according to the `__encoding` with the given states `__encode_state` and `__decode_state`.

Remark This functions checks to see if extension points for `text_validate_encodable_as` is available taking the available 4 parameters. If so, it calls this. Otherwise, it defers to doing a typical loop over the code points to verify it can be encoded into code units, and then decoded into code points, with no errors.

Parameters

- **__input** – **[in]** The input range of code points to validate is possible for encoding into code units.
- **__encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__encode_state** – **[in]** The state to use for the encoding portion of the validation check.
- **__decode_state** – **[in]** The state to use for the decoding portion of the validation check, if needed.

template<typename **_Input**, typename **_Encoding**, typename **_EncodeState**>
constexpr auto **validate_encodable_as**(*_Input* &&__input, *_Encoding* &&__encoding, *_EncodeState* &__encode_state)

Validates the code points of the `__input` according to the `__encoding` with the given states "`__encode_state`".

Remark This functions checks to see if extension points for `text_validate_encodable_as` is available taking the available 3 parameters. If so, it calls this. Otherwise, it defers to ztd::text::validate_encodable_as.

Parameters

- **__input** – **[in]** The input range of code points to validate is possible for encoding into code units.
- **__encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__encode_state** – **[in]** The state for encoding to use.

template<typename **_Input**, typename **_Encoding**>
constexpr auto **validate_encodable_as**(*_Input* &&__input, *_Encoding* &&__encoding)

Validates the code points of the `__input` according to the "`__encoding`".

Parameters

- **__input** – **[in]** The input range of code points to validate is possible for encoding into code units.

- **__encoding** – **[in]** The encoding to verify can properly encode the input of code units.

template<typename **_Input**>
constexpr auto **validate_encodable_as**(*_Input* &&__input)
Validates the code points of the input.

Remark This passes the default encoding as inferred from the discernible `value_type` of the input range input into the *ztd::text::default_code_point_encoding*.

Parameters **__input** – **[in]** The input range of code points to validate is possible for encoding into code units.

validate_transcodable_as

`ztd::text::validate_transcodable_as` is a function that takes an input sequence of `code_units` and attempts to validate that they can be turned into the `code_points` of the provided encoding. Unlike the *ztd::text::count_as_decoded* function, this does not take an error handler. **Any** error, even if it would be corrected over, produces a stop in the algorithm and a *validate_result*/*stateless_validate_result* object gets returned with the `.valid` member set to false.

The overloads of this function increase the level of control with each passed argument. At the last overload with four arguments, the function attempts to work call some extension points or falls back to the base function call in this order:

- The `text_validate_transcodable_as(input, from_encoding, to_encoding, decode_state, encode_state)` extension point, if possible.
- An internal, implementation-defined customization point.
- The `basic_validate_transcodable_as` base function.

The base function call, `basic_validate_transcodable_as`, simply performs the *core validating loop* using the *Lucky 7* design. The reason the last overload takes 2 state arguments is due to how the base implementation works from the core validating loop. If during the 3-argument overload it is detected that `text_validate_transcodable_as(input, encoding, decode_state)` can be called, it will be called without attempt to create an `encode_state` value with *ztd::text::make_encode_state(...)*.

During the `basic_validate_transcodable_as` loop, if it detects that there is a preferable `text_validate_transcodable_as_one`, it will call that method as `text_validate_transcodable_as_one(input, encoding, decode_state)` inside of the loop rather than doing the core design.

The *ztd::text::validate_result* type only includes the `decode_state` in all cases.

Note: This means that if you implement none of the extension points whatsoever, implementing the basic `decode_one` and `encode_one` functions on your Encoding Object type will guarantee a proper, working implementation.

Note: If you need to call the "basic" form of this function that takes no secret implementation shortcuts or user-defined extension points, then call `basic_validate_transcodable_as` directly. This can be useful to stop infinity loops when your extension points cannot handle certain inputs and thereby needs to "delegate" to the basic case.

Functions

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_DecodeState**, typename **_EncodeState**>
constexpr auto **basic_validate_transcodable_as**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_DecodeState* &__decode_state, *_EncodeState* &__encode_state)

Validates the code units of the `__input` according to the `__from_encoding` with the given states `__decode_state` and `__encode_state` to see if it can be turned into code points, and then code units again.

Remark This function explicitly does not call any extension points. It defers to doing a typical loop over the code points to verify it can be decoded into code points, and then encoded back into code units, with no errors and with the exact same value sequence as the original.

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__from_encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__to_encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__decode_state** – **[in]** The state to use for the decoding portion of the validation check.
- **__encode_state** – **[in]** The state to use for the encoding portion of the validation check.

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_DecodeState**, typename **_EncodeState**>
constexpr auto **validate_transcodable_as**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_DecodeState* &__decode_state, *_EncodeState* &__encode_state)

Validates the code units of the `__input` according to the `__encoding` with the given states `__decode_state` and `__encode_state` to see if it can be turned into code points.

Remark This functions checks to see if extension points for `text_validate_transcodable_as` is available taking the available 4 parameters. If so, it calls this. Otherwise, it defers to ztd::text::validate_transcodable_as.

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__from_encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__to_encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__decode_state** – **[in]** The state to use for the decoding portion of the validation check.
- **__encode_state** – **[in]** The state to use for the encoding portion of the validation check.

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**, typename **_DecodeState**>
constexpr auto **validate_transcodable_as**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding, *_DecodeState* &__decode_state)

Validates the code units of the `__input` according to the `__from_encoding` object with the given state `__decode_state` to see if it can be turned into code units of the `__to_encoding` object.

Remark This functions will call ztd::text::make_encode_state with `__to_encoding` to create a default *encode_state*.

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__from_encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__to_encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__decode_state** – **[in]** The state to use for the decoding portion of the validation check.

template<typename **_Input**, typename **_FromEncoding**, typename **_ToEncoding**>
constexpr auto **validate_transcodable_as**(*_Input* &&__input, *_FromEncoding* &&__from_encoding, *_ToEncoding* &&__to_encoding)

Validates the code units of the `__input` according to the `__from_encoding` object to see if it can be turned into code units of the `__to_encoding` object.

Remark This functions will call ztd::text::make_decode_state with the `__from_encoding` object to create a default *decode_state* to use before passing it to the next overload.

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__from_encoding** – **[in]** The encoding to verify can properly encode the input of code units.
- **__to_encoding** – **[in]** The encoding to verify can properly encode the input of code units.

template<typename **_Input**, typename **_ToEncoding**>
constexpr auto **validate_transcodable_as**(*_Input* &&__input, *_ToEncoding* &&__to_encoding)

Validates the code units of the `__input` according to the `__from_encoding` object to see if it can be turned into code units of the `__to_encoding` object.

Remark This functions will call ztd::text::make_encode_state with `__to_encoding` to create a default *encode_state*.

Parameters

- **__input** – **[in]** The input range of code units to validate is possible for encoding into code points.
- **__to_encoding** – **[in]** The encoding to verify can properly encode the input of code units.

1.9.6 Properties and Classifications

code_point

template<typename **_Type**>

class `ztd::text::`**code_point**
Retrieves the code point type for the given type.

Public Types

using **type** = typename remove_cvref_t<*_Type*>::code_point
The code point type for the given encoding type. If it does not exist, `ztd::text::unicode_code_point` is assumed.

using `ztd::text::`**code_point_t** = typename *code_point*<_Type>::type
A typename alias for *ztd::text::code_point*.

code_unit

template<typename **_Type**>

class `ztd::text::`**code_unit**
Retrieves the code unit type for the given type.

Public Types

using **type** = typename remove_cvref_t<*_Type*>::code_unit
The code unit type for the encoding type.

using `ztd::text::`**code_unit_t** = typename *code_unit*<_Type>::type
A typename alias for *ztd::text::code_unit*.

decode_state

template<typename **_Type**>

class `ztd::text::`**decode_state**
Retrieves the *decode_state* of the encoding type if it has one, or the `state` type of the encoding.

Public Types

using **type** = typename __txt_detail::__decode_state<remove_cvref_t<*_Type*>>::type
The *decode_state* type or `state` type on a given encoding type.

using `ztd::text::`**decode_state_t** = typename *decode_state*<_Type>::type
Typename alias for *ztd::text::decode_state*.

encode_state

template<typename **_Type**>

class `ztd::text::`**encode_state**
Retrieves the *encode_state* of the encoding type if it has one, or the `state` type of the encoding.

Public Types

using **type** = typename __txt_detail::__encode_state<remove_cvref_t<*_Type*>>::type
The *encode_state* type or `state` type on a given encoding type.

using `ztd::text::`**encode_state_t** = typename *encode_state*<_Type>::type
Typename alias for *ztd::text::encode_state*.

max_code_points

The maximum number of code points needed for a given encoding object. This can be used to create a suitably-sized automatic storage duration buffer, e.g.

```
using code_point = ztd::text::code_point_t<ztd::text::utf8>;
constexpr std::size_t max_output_size = ztd::text::max_code_points_v<ztd::text::utf8>;

std::array<code_point, max_output_size> my_cxx_buffer;
code_point my_c_buffer[max_output_size];
```

template<typename **_Type**>

static constexpr ::std::size_t `ztd::text::`**max_code_points_v** = *_Type*::max_code_points
Gets the maximum number of code points that can be produced by an encoding during a decode operation, suitable for initializing a automatic storage duration ("stack-allocated") buffer.

max_code_units

The maximum number of code units needed for a given encoding object. This can be used to create a suitably-sized automatic storage duration buffer, e.g.

```
using code_unit = ztd::text::code_unit_t<ztd::text::utf8>;
constexpr std::size_t max_output_size = ztd::text::max_code_units_v<ztd::text::utf8>;

// C++-style
std::array<code_unit, max_output_size> my_cxx_buffer;
```

(continues on next page)

(continued from previous page)

```
// or C-style
code_unit my_c_buffer[max_output_size];
```

template<typename **_Type**>

static constexpr ::std::size_t `ztd::text::`**max_code_units_v** = *_Type*::max_code_units
: Gets the maximum number of code units that can be produced by an encoding during an encode operation, suitable for initializing a automatic storage duration ("stack-allocated") buffer.

is_state_independent_v

template<typename **_Encoding**, typename **_Type**>

constexpr bool `ztd::text::`**is_state_independent_v** = !::std::is_constructible_v<*_Type*, *_Encoding*> && ::std::is_default_constructible_v<*_Type*>
: Whether or not the given type can be constructed without information from the encoding itself.

Remark This value tells users at compile time whether or not they need to be careful with the state. Rather than let users have to work this independently, two functions — ztd::text::make_encode_state(_Encoding) and ztd::text::make_encode_state(_Encoding) — handle the details here.

Template Parameters

- **_Encoding** – The encoding that may contain necessary information.
- **_Type** – The state type that may need information from the encoding to be successfully constructed.

is_decode_state_independent_v

template<typename **_Encoding**>

constexpr bool `ztd::text::`**is_decode_state_independent_v** = *is_state_independent_v*<*_Encoding*, *decode_state_t*<*_Encoding*>>
: Whether or not the encoding's `decode_state` can be constructed without information from the encoding itself.

is_encode_state_independent_v

template<typename **_Encoding**>

constexpr bool `ztd::text::`**is_encode_state_independent_v** = *is_state_independent_v*<*_Encoding*, *encode_state_t*<*_Encoding*>>
: Whether or not the encoding's `decode_state` can be constructed without information from the encoding itself.

is_decode_injective_v

Looks to see if the `decode_one` operation on a given encoding type is *injective*.

This classification checks whether the given encoding type has a type definition called `is_decode_injective` on it, and if it does checks to see if its `std::true_type`. If it's not present, or if it's `std::false_type`, then the encoding is assumed to NOT be *injective*.

template<typename **_Type**>

class **is_decode_injective** : public __is_decode_injective_sfinae<*_Type*>
Checks whether or not the decoding step for `_Type` is injective (cannot possibly lose information regardless of whatever valid input is put in).

Remark If the encoding object does not define *is_decode_injective*, it is assumed to be false (the safest default).

tparam _Type The encoding type to check.

template<typename **_Type**>

constexpr bool `ztd::text::`**is_decode_injective_v** = *is_decode_injective*<*_Type*>::value
A `::value` alias for *ztd::text::is_decode_injective*.

is_encode_injective_v

Looks to see if the `encode_one` operation on a given encoding type is *injective*.

This classification checks whether the given encoding type has a type definition called `is_encode_injective` on it, and if it does checks to see if its `std::true_type`. If it's not present, or if it's `std::false_type`, then the encoding is assumed to NOT be *injective*.

template<typename **_Type**>

class **is_encode_injective** : public __is_encode_injective_sfinae<*_Type*>
Checks whether or not the encoding step for `_Type` is injective (cannot possibly lose information regardless of whatever valid input is put in).

Remark If the encoding object does not define *is_encode_injective*, it is assumed to be false (the safest default).

tparam _Type The encoding type to check.

template<typename **_Type**>

constexpr bool `ztd::text::`**is_encode_injective_v** = *is_encode_injective*<*_Type*>::value
A `::value` alias for *ztd::text::is_encode_injective*.

is_code_units_(maybe_)replaceable

These two traits detect whether or not the given Encoding type have calls on them which return either a replacement range (`is_code_units_repleacable`) or a `std::optional` of a replacement range (`is_code_units_maybe_replaceable`).

The former is useful when it is guaranteed that your encoding will have a replacement range on it and does not need the extra cost of an indirection from not knowing. The latter is useful when something like a wrapped encoding may or may not have a replacement sequence.

template<typename **_Type**, typename ...**_Args**>

class **is_code_units_replaceable** : public is_detected<__txt_detail::__detect_is_code_units_replaceable, *_Type*, *_Args*...>
Checks whether the given encoding type returns a maybe-replacement range of code units.

Remark The `::value` boolean is true if the given `_Type` has a function named `replacement_code_units()` on it that can be called from a `const` -qualified `_Type` which returns a contiguous view of code units.

tparam _Type The type to check for the proper function call.

template<typename **_Type**, typename ...**_Args**>

constexpr bool `ztd::text::`**is_code_units_replaceable_v** = *is_code_units_replaceable*<*_Type*, *_Args*...>::value
A `::value` alias for *ztd::text::is_code_units_replaceable*.

template<typename **_Type**, typename ...**_Args**>

class **is_code_units_maybe_replaceable** : public is_detected<__txt_detail::__detect_is_code_units_maybe_replaceable, *_Type*, *_Args*...>
Checks whether the given encoding type returns a maybe-replacement range of code units.

Remark The `value` boolean is true if the given `_Type` has a function named `maybe_replacement_code_units()` on it that can be called from a `const` -qualified `_Type` which returns a `std::optional` containing a contiguous view of code units.

tparam _Type The type to check for the proper function call.

template<typename **_Type**, typename ...**_Args**>

constexpr bool `ztd::text::`**is_code_units_maybe_replaceable_v** = *is_code_units_maybe_replaceable*<*_Type*, *_Args*...>::value
A `::value` alias for *ztd::text::is_code_units_maybe_replaceable*.

is_code_points_(maybe_)replaceable

These two traits detect whether or not the given Encoding type have calls on them which return either a replacement range (`is_code_points_repleacable`) or a `std::optional` of a replacement range (`is_code_points_maybe_replaceable`).

The former is useful when it is guaranteed that your encoding will have a replacement range on it and does not need the extra cost of an indirection from not knowing. The latter is useful when something like a wrapped encoding may or may not have a replacement sequence.

template<typename **_Type**, typename ...**_Args**>

class **is_code_points_replaceable** : public is_detected<__txt_detail::__detect_is_code_points_replaceable, *_Type*, *_Args*...>

Checks whether the given encoding type returns a maybe-replacement range of code points.

Remark The `value` boolean is true if the given `_Type` has a function named `replacement_code_points()` on it that can be called from a `const` -qualified `_Type` object which returns a contiguous view of code points.

tparam _Type The type to check for the proper function call.

template<typename **_Type**, typename ...**_Args**>

constexpr bool `ztd::text::`**is_code_points_replaceable_v** = *is_code_points_replaceable*<*_Type*, *_Args*...>::value

A `::value` alias for *ztd::text::is_code_points_replaceable*.

template<typename **_Type**, typename ...**_Args**>

class **is_code_points_maybe_replaceable** : public is_detected<__txt_detail::__detect_is_code_points_maybe_replaceable, *_Type*, *_Args*...>

Checks whether the given encoding type returns a maybe-replacement range of code points.

Remark The `value` boolean is true if the given `_Type` has a function named `maybe_replacement_code_points()` on it that can be called from a `const` -qualified `_Type` object which returns a `std::optional` containing a contiguous view of code points.

tparam _Type The type to check for the proper function call.

template<typename **_Type**, typename ...**_Args**>

constexpr bool `ztd::text::`**is_code_points_maybe_replaceable_v** = *is_code_points_maybe_replaceable*<*_Type*, *_Args*...>::value

A `::value` alias for *ztd::text::is_code_points_maybe_replaceable*.

is_ignorable_error_handler

template<typename **_Type**>

class **is_ignorable_error_handler** : public __is_ignorable_error_handler_sfinae<*_Type*>
Whether or not the given `_Type` is an error handler that can be ignored.

Remark An error handler type can mark itself as ignorable by using a `using assume_valid = std::integral_constant<bool, value>` where `value` determines if the type's error handling callback can be ignored. This is what ztd::text::assume_valid does. Being configurable means templated error handlers can select whether or not they should be ignorable based on compile time, safe conditions that you can make up (including checking Macros or other environment data as a means of determining whether or not validity should be ignored.) If this results in a type derived from `std::true_type` and the encoder object using it encounters an error, then it is Undefined Behavior what occurs afterwards.

tparam _Type the Error Handling type to chec.

template<typename **_Type**>

constexpr bool `ztd::text::`**is_ignorable_error_handler_v** = *is_ignorable_error_handler*<*_Type*>::value
A `::value` alias for *ztd::text::is_ignorable_error_handler*.

is_unicode_encoding

Note: User Specializations: ✓ Okay! You can add other types to this classification by specializing the class template to a definition that derives from `std::true_type`, or turn it off explicitly by having a definition that derives from `std::false_type`. Note that specializing any type not explicitly marked with this notice is Undefined Behavior.

---

template<typename **_Type**>

class **is_unicode_encoding** : public __is_unicode_encoding_sfinae<*_Type*>
Checks whether or not the encoding has declared it can handle all of Unicode.

Remark If the encoding object does not define *is_unicode_encoding*, it is assumed to be false (the safest default).

tparam _Type The encoding type to check.

template<typename **_Type**>

constexpr bool `ztd::text::`**is_unicode_encoding_v** = *is_unicode_encoding*<*_Type*>::value
A `::value` alias for *ztd::text::is_unicode_encoding*.

contains_unicode_encoding

This function determines whether or not the type is or contains a unicode encoding. This means any encoding wherein the entirety of Unicode, all 21 bits, can be represented without loss of information. For a full list of encodings which are considered Unicode Encodings by this library, see *the Known Unicode Encodings list*.

This function checks for 2 things.

- It checks to see if the call `encoding.contains_unicode_encoding()` is well-formed and returns a boolean value. If this is the case, it calls `encoding.contains_unicode_encoding()` and returns that value.
- It looks to see if the provided `encoding` has a member type called `::is_unicode_encoding`. If this is the case, then it returns `is_unicode_encoding_v<Type>`.

If none of these work, then it returns `false`.

template<typename **_Encoding**>
constexpr bool `ztd::text::`**`contains_unicode_encoding`**(const *_Encoding* &__encoding) noexcept
Whether or not the provided encoding is a Unicode encoding.

Remark This function first checks if there is a function called `contains_unicode_encoding` . If it is present, then it returns the value of that function directly. Otherwise, it checks if ztd::text::is_unicode_encoding_v is true for the provided `__encoding` . If that's the case, then `true` is returned. Otherwise, it assumes the encoding is not a Unicode-compatible encoding and returns false.

Parameters **__encoding** – **[in]** The encoding to query.

is_unicode_code_point

This checks if the provided type is a *unicode_code_point*.

Note: User Specializations: ✓ Okay! You can add other types to this classification by specializing the class template to a definition that derives from `std::true_type`, or turn it off explicitly by having a definition that derives from `std::false_type`. Note that specializing any type not explicitly marked with this notice is Undefined Behavior.

Warning: Note there are some *expectations of unicode code points*. If your type violates these expectations then code depending on them is free to execute Undefined Behavior.

template<typename **_Type**>

class **is_unicode_code_point** : public std::integral_constant<bool, ::std::is_same_v<remove_cvref_t<*_Type*>, char32_t> || ::std::is_same_v<remove_cvref_t<*_Type*>, __txt_impl::*__unicode_code_point*> || *is_unicode_scalar_value_v*<*_Type*>>

template<typename **_Type**>

constexpr bool `ztd::text::`**`is_unicode_code_point_v`** = *is_unicode_code_point*<*_Type*>::value

is_unicode_scalar_value

This checks if the provided type is a *unicode_scalar_value*.

Note: User Specializations: ✓ Okay! You can add other types to this classification by specializing the class template to a definition that derives from `std::true_type`, or turn it off explicitly by having a definition that derives from `std::false_type`. Note that specializing any type not explicitly marked with this notice is Undefined Behavior.

Warning: Note there are some *expectations of scalar value types*. If your type violates these expectations then code depending on them is free to execute Undefined Behavior.

template<typename **_Type**>

class **is_unicode_scalar_value** : public std::integral_constant<bool, ::std::is_same_v<remove_cvref_t<*_Type*>, __txt_impl::*__unicode_scalar_value*>>

template<typename **_Type**>

constexpr bool `ztd::text::`**is_unicode_scalar_value_v** = *is_unicode_scalar_value*<*_Type*>::value

is_(bitwise_)transcoding_compatible

This classification checks if two encodings are compatible, or bitwise compatible. The heuristic for normal compatibility is simple:

- it checks if the two encodings are identical;
- it checks if the two encodings are near-identical derivations of one another (e.g., *UTF-8* being converted to *MUTF-8* (but not in the other direction)); or,
- it checks if the code point types between the two encodings are the same, or if they are *both some form of unicode code point*.

This type specifically uses the first type as the `From` encoding (e.g., the one to decode the input code unit sequence) and the second type as the `To` encoding (e.g., the one to encode the intermediate decoded code point sequence).

Note: User Specializations: ✓ Okay! You can add other types to this classification by specializing the class template to a definition that derives from `std::true_type`, or turn it off explicitly by having a definition that derives from `std::false_type`. Note that specializing any type not explicitly marked with this notice is Undefined Behavior.

Warning: Specializing this type for types which are not either transcoding compatible or bitwise compatible can result in Undefined Behavior within the library.

template<typename **_From**, typename **_To**>

class **is_transcoding_compatible** : public std::integral_constant<bool, __txt_detail::__is_bitwise_transcoding_compatible_v<*_From*, *_To*>>

Checks whether or not the specified `_From` encoding can be transcoded to the `_To` encoding without invoking a lossy conversion when using the intermediate code points.

Remark First, it checks if the encodings are bitwise compatible with one another (e.g., transcoding ASCII to UTF-8). If that is not the case, then it checks if the two encodings are just identical. Finally, it checks if the code point types are the same or if it's putting unicode scalar values into unicode code points (which is valid one way, but not the other way since scalar values do not allow surrogates). If none of these are true, then, the intermediate code point likely cannot convert between the two losslessly.

tparam _From The encoding that is going to decode the input code units into the intermediate code points.

tparam _To The encoding that is going to encode the intermediate code points into the final code units.

template<typename **_From**, typename **_To**>

constexpr bool `ztd::text::is_transcoding_compatible_v` = *is_transcoding_compatible*<*_To*, *_From*>::value
A `::value` alias for *ztd::text::is_transcoding_compatible*.

template<typename **_From**, typename **_To**>

class **is_bitwise_transcoding_compatible** : public std::integral_constant<bool, __txt_detail::__is_bitwise_transcoding_compatible_v<*_From*, *_To*>>
Checks whether or not the specified `_From` encoding can be transcoded to the `_To` encoding without by form of bit copying.

tparam _From The encoding that is going to decode the input code units into the intermediate code points.

tparam _To The encoding that is going to encode the intermediate code points into the final code units.

template<typename **_From**, typename **_To**>

constexpr bool `ztd::text::is_bitwise_transcoding_compatible_v` = *is_bitwise_transcoding_compatible*<*_From*, *_To*>::value
A `::value` alias for *ztd::text::is_transcoding_compatible*.

default_code_point_encoding

Picks the default encoding for the given code point type. In all cases, this just points a given code point type to *ztd::text::utf8*. Errors if there is no default association.

Note: User Specializations: ✓ Okay! You can add other types to this classification by specializing the class template. Your specialization must have a type definition named `type` (as in, `using type = ...;` or *typedef … type;* `) inside of the class specialization that is `public:`ly accessible. Note that specializing any type not explicitly marked with this notice is Undefined Behavior.

---

template<typename **_Type**>

class **default_code_point_encoding** : public __default_code_point_encoding<*_Type*, false>
The default encoding associated with a given code point type, that serves as either input to an encode operation or output from decode operation.

tparam _Type The code point type, with no cv-qualifiers

using ztd::text::**default_code_point_encoding_t** = typename *default_code_point_encoding*<_Type>::type
A typename alias for *ztd::text::default_code_point_encoding*.

Template Parameters **_Type** – The code point type, with no cv-qualifiers

template<typename **_Type**>

class **default_consteval_code_point_encoding** : public __default_code_point_encoding<*_Type*, true>
The default encoding associated with a given code point type, that serves as either input to an encode operation or output from decode operation. This uses additional information that this is at compile time, not run time, to help make a decision as to what to do.

tparam **_Type** The code point type, with no cv-qualifiers

using ztd::text::**default_consteval_code_point_encoding_t** = typename *default_consteval_code_point_encoding*<_Type>::type
A typename alias for *ztd::text::default_consteval_code_point_encoding*.

Template Parameters **_Type** – The code point type, with no cv-qualifiers

default_code_unit_encoding

Picks the default encoding for the given code unit type (for both run time, and compile time with the appropriately selected version of this property). The default association table is below. Errors if there is no default association.

Note: User Specializations: ✓ Okay! You can add other types to this classification by specializing the class template. Your specialization must have a type definition named `type` (as in, `using type = ...;` or *typedef … type;* `) inside of the class specialization that is `public:`ly accessible. Note that specializing any type not explicitly marked with this notice is Undefined Behavior.

Type	Encoding
`signed char`	*ztd::text::basic_ascii<signed char>*
`char`	*ztd::text::execution*
`char` (compile time)	*ztd::text::literal*
`wchar_t`	*ztd::text::wide_execution*
`wchar_t` (compile time)	*ztd::text::wide_literal*
`char8_t`	*ztd::text::basic_utf8<char8_t>*
`ztd::uchar8_t`	*ztd::text::basic_utf8<uchar8_t>*
`std::byte`	*ztd::text::basic_utf8<std::byte>*
`char16_t`	*ztd::text::utf16*
`char32_t`	*ztd::text::utf32*

template<typename **_Type**>

class **default_code_unit_encoding** : public __default_code_unit_encoding<*_Type*, false>
The default encoding associated with a given code unit type, that serves as either input to a decode operation or output from an encode operation.

tparam **_Type** The code unit type, with no cv-qualifiers

using ztd::text::**default_code_unit_encoding_t** = typename *default_code_unit_encoding*<_Type>::type
A typename alias for *ztd::text::default_code_unit_encoding*.

Template Parameters **_Type** – The code unit type, with no cv-qualifiers

template<typename **_Type**>

class **default_consteval_code_unit_encoding** : public __default_code_unit_encoding<*_Type*, true>
The default encoding associated with a given code unit type, that serves as either input to a decode operation or output from an encode operation. This uses the additional information that this is compiletime, not runtime, to help make the decision on what to do.

tparam _Type The code unit type, with no cv-qualifiers

using ztd::text::**default_consteval_code_unit_encoding_t** = typename *default_consteval_code_unit_encoding*<_Type>::type
A typename alias for *ztd::text::default_consteval_code_unit_encoding*.

Template Parameters **_Type** – The code unit type, with no cv-qualifiers

1.9.7 Result Types, Status Codes and Quality Aides

encoding_error

enum ztd::text::**encoding_error**
Describes a failure to encode, decode, transcode, or count, for four core various reasons.

Remark This does not cover specific failures, like if a sequence was overlong (e.g., UTF-8) or if an encode operation produced an uunpaired surrogate value (e.g. UTF-16).

Values:

enumerator **ok**
The okay status; everything is fine.

Remark This does not necessarily mean an error handler was not called. An error handler can set the error code to *ztd::text::encoding_error::ok* after performing corrective action: see *ztd::text::replacement_handler_t* for an example.

enumerator **invalid_sequence**
Input contains ill-formed sequences. This means there were available units of input to read, but what was read resulted in an error.

enumerator **incomplete_sequence**
Input contains incomplete sequences. This means that the input was exhausted, without finding an invalid sequence, and therefore more input may be required.

Remark Depending on context, this may or may not be an error in your use case (e.g., reading part of an incomplete network buffer and waiting for more). See *ztd::text::incomplete_handler* as a way to aid with this use case.

enumerator **insufficient_output_space**
Output cannot receive the successfully encoded or decoded sequence. This means that, while there were no invalid or incomplete sequences in the input, the output ran out of space to receive it.

Remark Provide a bigger storage area or guarantee that it meets the minimum required size for potential output. This can be queried for an encoding by using ztd::text::max_code_points_v<the_encoding> for code points, and ztd::text::max_code_units_v<the_encoding> for code units.

inline constexpr ::std::string_view `ztd::text::to_name`(*encoding_error* __error_code)
Converts an encoding_error to a string value.

Remark If a value outside of the allowed encoding_error is passed, then undefined behavior happens.

Returns A null-terminated string_view to the data.

text_tag

A tag type used explicitly for *extension points*.

template<typename ...**_Args**>

class **text_tag**
A *text_tag* type, useful in helping to constrain extension points and more.

Remark The *text_tag* type will always be used

tparam _Args The types to be used as part of the tagging mechanism. As this is variadic, allows no classes to be passed.

make_decode_state

This detects when the `decode_state` of a given `encoding` requires the `encoding` itself to make said state. If so, it will call the `decode_state`'s constructor with the encoding passed in. Otherwise, it simply default-constructs a state. In either case, the constructed value is returned to the user.

The classification for this is done by *ztd::text::is_decode_state_independent*.

template<typename **_Encoding**>
constexpr *decode_state_t*<remove_cvref_t<*_Encoding*>> `ztd::text::make_decode_state`(*_Encoding* &__encoding) noexcept
Constructs the *decode_state* of the given encoding, based on whether or not the encoding and state meet the criteria of ztd::text::is_decode_state_independent_v.

Parameters **__encoding** – **[in]** The encoding object to use, if applicable, for the construction of the state.

template<typename **_Encoding**>

constexpr *decode_state_t*<remove_cvref_t<*_Encoding*>> ztd::text::**make_decode_state_with**(*_Encoding* &__encoding, const *encode_state_t*<remove_cvref_t<*_* &__encode_state) noexcept

Constructs the *decode_state* of the given encoding, based on whether or not the encoding and state meet the criteria of ztd::text::is_decode_state_independent_v or whether it can be created by copy construction from the given __encode_state.

Parameters

- **__encoding** – **[in]** The encoding object to use, if applicable, for the construction of the state.
- **__encode_state** – **[in]** A preexisting state from the encoder.

make_encode_state

This detects when the encode_state of a given encoding requires the encoding itself to make said state. If so, it will call the encode_state's constructor with the encoding passed in. Otherwise, it simply default-constructs a state. In either case, the constructed value is returned to the user.

The classification for this is done by *ztd::text::is_encode_state_independent*.

template<typename **_Encoding**>
constexpr *encode_state_t*<remove_cvref_t<*_Encoding*>> ztd::text::**make_encode_state**(*_Encoding* &__encoding) noexcept

Constructs the *encode_state* of the given encoding, based on whether or not the encoding and state meet the criteria of ztd::text::is_encode_state_independent_v.

Parameters **__encoding** – **[in]** The encoding object to use, if applicable, for the construction of the state.

template<typename **_Encoding**>
constexpr *encode_state_t*<remove_cvref_t<*_Encoding*>> ztd::text::**make_encode_state_with**(*_Encoding* &__encoding, const *decode_state_t*<remove_cvref_t<*_* &__decode_state) noexcept

Constructs the *encode_state* of the given encoding, based on whether or not the encoding and state meet the criteria of ztd::text::is_encode_state_independent_v or whether it can be created by copy construction from the given __decode_state.

Parameters

- **__encoding** – **[in]** The encoding object to use, if applicable, for the construction of the state.
- **__decode_state** – **[in]** A preexisting state from the decoder.

unicode_code_point

If *ZTD_TEXT_UNICODE_CODE_POINT_DISTINCT_TYPE* is turned on, this type definition points to an internal class which implements the invariant of being a *unicode code point*. Otherwise, it is simply `char32_t`.

A unicode code point is stored as an at least 32-bit value, but may occupy more space depending on the architecture. It requires 21 bits of space to fit the required unicode code point definition. If the distinct type is used, then this type will trap (`abort`/`assert`) if the value is greater than the allowed 21 bits.

typedef char32_t `ztd::text::`**`unicode_code_point`**

Internal Type

Warning: Names with double underscores, and within the `__detail` and `__impl` namespaces are reserved for the implementation. Referencing this entity directly is bad, and the name/functionality can be changed at any point in the future. Relying on anything not guaranteed by the documentation is Undefined Behavior.

class `ztd::text::__txt_impl::`**`__unicode_code_point`**
A 32-bit value that is within the allowed 21 bits of Unicode. Can be one of the surrogate values.

Public Functions

`__unicode_code_point``()` noexcept = default
Constructs a code point value of indeterminate value (if no parentheses/brackets are provided) or with the value 0 (if parentheses/brackets are provided for intentional value initialization).

inline constexpr **`__unicode_code_point`**`(`char32_t __code_point`)` noexcept
Constructs a code point value with the given code point value.

Remark

inline explicit constexpr **`operator char32_t`**`()` const noexcept
An explicit conversion to a typical char32_t value, bit-compatible with a normal code point value.

inline constexpr const char32_t &**`value`**`()` const & noexcept
Retrieves the underlying value.

inline constexpr char32_t &**`value`**`()` & noexcept
Retrieves the underlying value.

inline constexpr char32_t &&**`value`**`()` && noexcept
Retrieves the underlying value.

unicode_scalar_value

If *ZTD_TEXT_UNICODE_SCALAR_VALUE_DISTINCT_TYPE* is turned on, this type definition points to an internal class which implements the invariant of being a *unicode scalar value*. Otherwise, it is simply `char32_t`.

A unicode scalar value is stored as an at least 32-bit value, but may occupy more space depending on the architecture. It requires 21 bits of space to fit the required unicode code point definition. If the distinct type is used, then this type will trap (`abort`/`assert`) if the value is greater than the allowed 21 bits, or if the value results in one of the Unicode Surrogate Pair values used for UTF-16 encoding and decoding. Not recommended for *ztd::text::wtf8* usage, as that encoding produces Unicode Surrogate Pair values intentionally.

typedef char32_t `ztd::text::`**`unicode_scalar_value`**

Internal Type

Warning: Names with double underscores, and within the `__detail` and `__impl` namespaces are reserved for the implementation. Referencing this entity directly is bad, and the name/functionality can be changed at any point in the future. Relying on anything not guaranteed by the documentation is Undefined Behavior.

class `ztd::text::__txt_impl::`**`__unicode_scalar_value`**

A 32-bit value that is within the allowed 21 bits of Unicode and is not one of the Surrogate values.

Remark The invariant is enforced with an assertion in normal modes, and can optionally be enforced by turning on `ZTD_TEXT_UNICODE_SCALAR_VALUE_INVARIANT_ABORT`.

Public Functions

`__unicode_scalar_value``()` noexcept = default

Constructs a scalar value of indeterminate value (if no parentheses/brackets are provided) or with the value 0 (if parentheses/brackets are provided for intentional value initialization).

inline constexpr **`__unicode_scalar_value`**`(`char32_t __code_point`)` noexcept

Constructs a scalar value with the given code point value.

Remark

inline explicit constexpr **`operator char32_t`**`()` const noexcept

An explicit conversion to a typical char32_t value, bit-compatible with a normal code point value.

inline constexpr const char32_t &**`value`**`()` const & noexcept

Retrieves the underlying value.

inline constexpr char32_t &**`value`**`()` & noexcept

Retrieves the underlying value.

inline constexpr char32_t &&**`value`**`()` && noexcept

Retrieves the underlying value.

stateless_decode_result

template<typename **_Input**, typename **_Output**>

class `ztd::text::stateless_decode_result`
The result of all decode operations from encoding objects and higher-level calls (such as ztd_text_decode).

Subclassed by *decode_result< _Input, _Output, _State >*

Public Functions

template<typename **_ArgInput**, typename **_ArgOutput**, typename **_ArgState**>
inline constexpr **stateless_decode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *encoding_error* __error_code = *encoding_error*::*ok*) noexcept(noexcept(*stateless_decode_result*(::std::forward<*_ArgInput*>(*__input*), ::std::forward<*_ArgOutput*>(*__output*), *__error_code*, *__error_code* != *encoding_error*::*ok*)))

Constructs a *ztd::text::decode_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__error_code** – **[in]** The error code for the decoding opertion, if any.

template<typename **_ArgInput**, typename **_ArgOutput**>
inline constexpr **stateless_decode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *encoding_error* __error_code, ::std::size_t __handled_errors) noexcept(::std::is_nothrow_constructible_v<*_Input*, *_ArgInput*> && ::std::is_nothrow_constructible_v<*_Output*, *_ArgOutput*>)

Constructs a *ztd::text::decode_result* with the provided parameters and information, including whether or not an error was handled.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__error_code** – **[in]** The error code for the decode operation, taken as the first of either the decode operation that failed.
- **__handled_errors** – **[in]** Whether or not an error was handled. Some error handlers are corrective (see *ztd::text::replacement_handler_t*), and so the error code is not enough to determine if the handler was invoked. This allows the value to be provided directly when constructing this result type.

inline constexpr bool **errors_were_handled**() const noexcept
Whether or not any errors were handled.

Returns Simply checks whether `handled_errors` is greater than 0.

Public Members

_Input **input**
The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

_Output **output**
The reconstructed output_view object, with its .begin() incremented by the number of code units successfully written (can be identical to .begin() on original range on failure).

encoding_error **error_code**
The kind of error that occured, if any.

::std::size_t **handled_errors**
Whether or not the error handler was invoked, regardless of if the error_code is set or not set to *ztd::text::encoding_error::ok*.

decode_result

template<typename **_Input**, typename **_Output**, typename **_State**>

class `ztd::text::`**decode_result** : public ztd::text::*stateless_decode_result*<*_Input*, *_Output*>
The result of all decode operations from encoding objects and higher-level calls (such as ztd_text_decode).

Public Functions

template<typename **_ArgInput**, typename **_ArgOutput**, typename **_ArgState**>
inline constexpr **decode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *_ArgState* &&__state,
encoding_error __error_code = *encoding_error*::*ok*)
Constructs a *ztd::text::decode_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__state** – **[in]** The state related to the Encoding that performed the decode operation.
- **__error_code** – **[in]** The error code for the decoding opertion, if any.

template<typename **_ArgInput**, typename **_ArgOutput**, typename **_ArgState**>
inline constexpr **decode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *_ArgState* &&__state,
encoding_error __error_code, ::std::size_t __handled_errors)
Constructs a *ztd::text::decode_result* with the provided parameters and information, including whether or not an error was handled.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__state** – **[in]** The state related to the Encoding that performed the decode operation.

- **__error_code** – **[in]** The error code for the decode operation, taken as the first of either the decode operation that failed.
- **__handled_errors** – **[in]** Whether or not an error was handled. Some error handlers are corrective (see *ztd::text::replacement_handler_t*), and so the error code is not enough to determine if the handler was invoked. This allows the value to be provided directly when constructing this result type.

inline constexpr bool **errors_were_handled**() const noexcept
Whether or not any errors were handled.

Returns Simply checks whether `handled_errors` is greater than 0.

Public Members

::ztd::reference_wrapper<*_State*> **state**
The state of the associated Encoding used for decoding input code units to code points.

_Input **input**
The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

_Output **output**
The reconstructed output_view object, with its .begin() incremented by the number of code units successfully written (can be identical to .begin() on original range on failure).

encoding_error **error_code**
The kind of error that occured, if any.

::std::size_t **handled_errors**
Whether or not the error handler was invoked, regardless of if the error_code is set or not set to *ztd::text::encoding_error::ok*.

stateless_encode_result

template<typename **_Input**, typename **_Output**>

class `ztd::text::stateless_encode_result`
The result of all encode operations from encoding objects and higher-level calls (such as ztd_text_encode).

Subclassed by *encode_result< _Input, _Output, _State >*

Public Functions

template<typename **_ArgInput**, typename **_ArgOutput**>
inline constexpr **stateless_encode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *encoding_error* __error_code = *encoding_error*::*ok*) noexcept(noexcept(*stateless_encode_result*(::std::forward<*_ArgInput*>(*__input*), ::std::forward<*_ArgOutput*>(*__output*), *__error_code*, *__error_code* != *encoding_error*::*ok*)))
Constructs a *ztd::text::encode_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__error_code** – **[in]** The error code for the decoding opertion, if any.

template<typename **_ArgInput**, typename **_ArgOutput**>
inline constexpr **stateless_encode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *encoding_error* __error_code, ::std::size_t __handled_errors) noexcept(::std::is_nothrow_constructible_v<*_Input*, *_ArgInput*> && ::std::is_nothrow_constructible_v<*_Output*, *_ArgOutput*>)

Constructs a *ztd::text::encode_result* with the provided parameters and information, including whether or not an error was handled.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__error_code** – **[in]** The error code for the encode operation, if any.
- **__handled_errors** – **[in]** Whether or not an error was handled. Some error handlers are corrective (see *ztd::text::replacement_handler_t*), and so the error code is not enough to determine if the handler was invoked. This allows the value to be provided directly when constructing this result type.

inline constexpr bool **errors_were_handled**() const noexcept

Whether or not any errors were handled.

Returns Simply checks whether `handled_errors` is greater than 0.

Public Members

_Input **input**

The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

_Output **output**

The reconstructed output_view object, with its .begin() incremented by the number of code units successfully written (can be identical to .begin() on original range on failure).

encoding_error **error_code**

The kind of error that occured, if any.

::std::size_t **handled_errors**

Whether or not the error handler was invoked, regardless of if the error_code is set or not set to *ztd::text::encoding_error::ok*.

encode_result

template<typename **_Input**, typename **_Output**, typename **_State**>

class `ztd::text::encode_result` : public ztd::text::*stateless_encode_result*<*_Input*, *_Output*>

The result of all encode operations from encoding objects and higher-level calls (such as ztd_text_encode).

Public Functions

template<typename **_ArgInput**, typename **_ArgOutput**, typename **_ArgState**>
inline constexpr **encode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *_ArgState* &&__state, *encoding_error* __error_code = *encoding_error*::*ok*)

Constructs a *ztd::text::encode_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__state** – **[in]** The state related to the Encoding that performed the encode operation.
- **__error_code** – **[in]** The error code for the decoding opertion, if any.

template<typename **_ArgInput**, typename **_ArgOutput**, typename **_ArgState**>
inline constexpr **encode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *_ArgState* &&__state, *encoding_error* __error_code, ::std::size_t __handled_errors)

Constructs a *ztd::text::encode_result* with the provided parameters and information, including whether or not an error was handled.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__state** – **[in]** The state related to the Encoding that performed the encode operation.
- **__error_code** – **[in]** The error code for the encode operation, if any.
- **__handled_errors** – **[in]** Whether or not an error was handled. Some error handlers are corrective (see *ztd::text::replacement_handler_t*), and so the error code is not enough to determine if the handler was invoked. This allows the value to be provided directly when constructing this result type.

inline constexpr bool **errors_were_handled**() const noexcept

Whether or not any errors were handled.

Returns Simply checks whether `handled_errors` is greater than 0.

Public Members

_State &**state**
The state of the associated Encoding used for decoding input code points to code units.

_Input **input**
The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

_Output **output**
The reconstructed output_view object, with its .begin() incremented by the number of code units successfully written (can be identical to .begin() on original range on failure).

encoding_error **error_code**
The kind of error that occured, if any.

::std::size_t **handled_errors**
Whether or not the error handler was invoked, regardless of if the error_code is set or not set to *ztd::text::encoding_error::ok*.

stateless_transcode_result

template<typename **_Input**, typename **_Output**>

class `ztd::text::stateless_transcode_result`
The result of transcoding operations (such as ztd_text_transcode) that specifically do not include a reference to the state.

Subclassed by *transcode_result< _Input, _Output, _FromState, _ToState >*

Public Functions

template<typename **_ArgInput**, typename **_ArgOutput**>
inline constexpr **stateless_transcode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *encoding_error* __error_code = *encoding_error*::*ok*) noexcept(noexcept(*stateless_transcode_result*(::std::forward<*_ArgInput*>(*__input*), ::std::forward<*_ArgOutput*>(*__output*), *__error_code*, *__error_code* != *encoding_error*::*ok*)))
Constructs a *ztd::text::stateless_transcode_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__error_code** – **[in]** The error code for the encode operation, taken as the first of either the encode or decode operation that failed.

template<typename **_ArgInput**, typename **_ArgOutput**>

inline constexpr **stateless_transcode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *encoding_error* __error_code, ::std::size_t __handled_errors) noexcept(::std::is_nothrow_constructible_v<*_Input*, *_ArgInput*> && ::std::is_nothrow_constructible_v<*_Output*, *_ArgOutput*>)

Constructs a *ztd::text::stateless_transcode_result* with the provided parameters and information, including whether or not an error was handled.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__error_code** – **[in]** The error code for the encode operation, taken as the first of either the encode or decode operation that failed.
- **__handled_errors** – **[in]** Whether or not an error was handled. Some error handlers are corrective (see *ztd::text::replacement_handler_t*), and so the error code is not enough to determine if the handler was invoked. This allows the value to be provided directly when constructing this result type.

inline constexpr bool **errors_were_handled**() const noexcept

Whether or not any errors were handled.

Returns Simply checks whether `handled_errors` is greater than 0.

Public Members

_Input **input**

The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

_Output **output**

The reconstructed output_view object, with its .begin() incremented by the number of code units successfully written (can be identical to .begin() on original range on failure).

encoding_error **error_code**

The kind of error that occured, if any.

::std::size_t **handled_errors**

Whether or not the error handler was invoked, regardless of if the error_code is set or not set to *ztd::text::encoding_error::ok*.

transcode_result

template<typename **_Input**, typename **_Output**, typename **_FromState**, typename **_ToState**>

class `ztd::text::transcode_result` : public ztd::text::*stateless_transcode_result*<*_Input*, *_Output*>

The result of transcoding operations (such as ztd_text_transcode).

Public Functions

template<typename **_ArgInput**, typename **_ArgOutput**, typename **_ArgFromState**, typename **_ArgToState**>
inline constexpr **transcode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *_ArgFromState* &&__from_state, *_ArgToState* &&__to_state, *encoding_error* __error_code = *encoding_error*::*ok*)

Constructs a *ztd::text::transcode_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__from_state** – **[in]** The state related to the "From Encoding" that performed the decode half of the operation.
- **__to_state** – **[in]** The state related to the "To Encoding" that performed the encode half of the operation.
- **__error_code** – **[in]** The error code for the encode operation, taken as the first of either the encode or decode operation that failed.

template<typename **_ArgInput**, typename **_ArgOutput**, typename **_ArgFromState**, typename **_ArgToState**>
inline constexpr **transcode_result**(*_ArgInput* &&__input, *_ArgOutput* &&__output, *_ArgFromState* &&__from_state, *_ArgToState* &&__to_state, *encoding_error* __error_code, ::std::size_t __handled_errors)

Constructs a *ztd::text::transcode_result* with the provided parameters and information, including whether or not an error was handled.

Parameters

- **__input** – **[in]** The input range to store.
- **__output** – **[in]** The output range to store.
- **__from_state** – **[in]** The state related to the "From Encoding" that performed the decode half of the operation.
- **__to_state** – **[in]** The state related to the "To Encoding" that performed the encode half of the operation.
- **__error_code** – **[in]** The error code for the encode operation, taken as the first of either the encode or decode operation that failed.
- **__handled_errors** – **[in]** Whether or not an error was handled. Some error handlers are corrective (see *ztd::text::replacement_handler_t*), and so the error code is not enough to determine if the handler was invoked. This allows the value to be provided directly when constructing this result type.

inline constexpr bool **errors_were_handled**() const noexcept

Whether or not any errors were handled.

Returns Simply checks whether `handled_errors` is greater than 0.

Public Members

::ztd::reference_wrapper<*_FromState*> **from_state**
: A reference to the state of the associated Encoding used for decoding input code units to intermediate code points.

::ztd::reference_wrapper<*_ToState*> **to_state**
: A reference to the state of the associated Encoding used for encoding intermediate code points to code units.

_Input **input**
: The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

_Output **output**
: The reconstructed output_view object, with its .begin() incremented by the number of code units successfully written (can be identical to .begin() on original range on failure).

encoding_error **error_code**
: The kind of error that occured, if any.

::std::size_t **handled_errors**
: Whether or not the error handler was invoked, regardless of if the error_code is set or not set to *ztd::text::encoding_error::ok*.

stateless_count_result

template<typename **_Input**>

class `ztd::text::`**stateless_count_result**
: The result of counting operations (such as ztd_text_count_as_encoded and ztd_text_count_as_encoded) that specifically do not include a reference to the state.

  Subclassed by *count_result< _Input, _State >*, count_transcode_result< _Input, _FromState, _ToState >

Public Functions

template<typename **_ArgInput**>
inline constexpr **stateless_count_result**(*_ArgInput* &&__input, ::std::size_t __count, *encoding_error* __error_code = *encoding_error*::*ok*)
: Constructs a *ztd::text::stateless_count_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

  Parameters

  - **__input** – **[in]** The input range to store.
  - **__count** – **[in]** The number of code points or code units successfully counted.
  - **__error_code** – **[in]** The error code for the encode operation, taken as the first of either the encode or decode operation that failed.

template<typename **_ArgInput**>

inline constexpr **stateless_count_result**(*_ArgInput* &&__input, ::std::size_t __count, *encoding_error* __error_code, ::std::size_t __handled_errors)

Constructs a *ztd::text::stateless_count_result* with the provided parameters and information, including whether or not an error was handled.

Parameters

- **__input** – **[in]** The input range to store.
- **__count** – **[in]** The number of code points or code units successfully counted.
- **__error_code** – **[in]** The error code for the encode operation, taken as the first of either the encode or decode operation that failed.
- **__handled_errors** – **[in]** Whether or not an error was handled. Some error handlers are corrective (see *ztd::text::replacement_handler_t*), and so the error code is not enough to determine if the handler was invoked. This allows the value to be provided directly when constructing this result type.

inline constexpr bool **errors_were_handled**() const noexcept

Whether or not any errors were handled.

Returns Simply checks whether `handled_errors` is greater than 0.

Public Members

_Input **input**

The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

::std::size_t **count**

The number of code units or code points counted successfully, so far.

encoding_error **error_code**

The kind of error that occured, if any.

::std::size_t **handled_errors**

Whether or not the error handler was invoked, regardless of if the error_code is set or not set to *ztd::text::encoding_error::ok*.

count_result

template<typename **_Input**, typename **_State**>

class `ztd::text::`**count_result** : public ztd::text::*stateless_count_result*<*_Input*>

The result of counting operations (such as ztd_text_count_as_encoded and ztd_text_count_as_encoded).

Public Functions

template<typename **_ArgInput**, typename **_ArgState**>
inline constexpr **count_result**(*_ArgInput* &&__input, ::std::size_t __count, *_ArgState* &&__state, *encoding_error* __error_code = *encoding_error*::*ok*)

Constructs a *ztd::text::count_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__count** – **[in]** The number of code points or code units successfully counted.
- **__state** – **[in]** The state related to the encoding for the counting operation.
- **__error_code** – **[in]** The error code for the encode operation, taken as the first of either the encode or decode operation that failed.

template<typename **_ArgInput**, typename **_ArgState**>
inline constexpr **count_result**(*_ArgInput* &&__input, ::std::size_t __count, *_ArgState* &&__state, *encoding_error* __error_code, ::std::size_t __handled_errors)

Constructs a *ztd::text::count_result* with the provided parameters and information, including whether or not an error was handled.

Parameters

- **__input** – **[in]** The input range to store.
- **__count** – **[in]** The number of code points or code units successfully counted.
- **__state** – **[in]** The state related to the encode operation that counted the code units.
- **__error_code** – **[in]** The error code for the encode operation, taken as the first of either the encode or decode operation that failed.
- **__handled_errors** – **[in]** Whether or not an error was handled. Some error handlers are corrective (see *ztd::text::replacement_handler_t*), and so the error code is not enough to determine if the handler was invoked. This allows the value to be provided directly when constructing this result type.

inline constexpr bool **errors_were_handled**() const noexcept

Whether or not any errors were handled.

Returns Simply checks whether `handled_errors` is greater than 0.

Public Members

::ztd::reference_wrapper<*_State*> **state**

A reference to the state of the associated Encoding used for counting.

_Input **input**

The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

::std::size_t **count**

The number of code units or code points counted successfully, so far.

encoding_error **error_code**
The kind of error that occured, if any.

::std::size_t **handled_errors**
Whether or not the error handler was invoked, regardless of if the error_code is set or not set to *ztd::text::encoding_error::ok*.

stateless_validate_result

template<typename **_Input**>

class `ztd::text::`**stateless_validate_result**
The result of valdation operations (such as ztd_text_validate_decodable_as and ztd_text_validate_encodable_as) that specifically do not include a reference to the state.

Subclassed by *validate_result< _Input, _State >*, *validate_transcode_result< _Input, _DecodeState, _EncodeState >*

Public Functions

template<typename **_ArgInput**>
inline constexpr **stateless_validate_result**(*_ArgInput* &&__input, bool __is_valid)
Constructs a *ztd::text::validate_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__is_valid** – **[in]** Whether or not the validation succeeded.

inline explicit constexpr **operator bool**() const noexcept
A conversion for use in if statements and conditional operators.

Returns Whether or not the result is valid or not.

Public Members

_Input **input**
The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

bool **valid**
Whether or not the specified input is valid or not.

validate_result

template<typename **_Input**, typename **_State**>

class `ztd::text::validate_result` : public ztd::text::*stateless_validate_result*<*_Input*>
The result of validation operations (such as ztd_text_validate_decodable_as and ztd_text_validate_encodable_as).

Public Functions

template<typename **_ArgInput**, typename **_ArgState**>
inline constexpr **validate_result**(*_ArgInput* &&__input, bool __is_valid, *_ArgState* &&__state)
Constructs a *ztd::text::validate_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__is_valid** – **[in]** Whether or not the validation succeeded.
- **__state** – **[in]** The state related to the encoding that was used to do validation.

inline explicit constexpr **operator bool**() const noexcept
A conversion for use in if statements and conditional operators.

Returns Whether or not the result is valid or not.

Public Members

::ztd::reference_wrapper<*_State*> **state**
A reference to the state of the associated Encoding used for validating the input.

_Input **input**
The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

bool **valid**
Whether or not the specified input is valid or not.

validate_transcode_result

template<typename **_Input**, typename **_DecodeState**, typename **_EncodeState**>

class `ztd::text::validate_transcode_result` : public ztd::text::*stateless_validate_result*<*_Input*>
The result of a transcoding validation operations (e.g. from ztd_text_validate_transcodable_as).

Public Functions

template<typename **_ArgInput**, typename **_ArgFromState**, typename **_ArgToState**>
inline constexpr **validate_transcode_result**(*_ArgInput* &&__input, bool __is_valid, *_ArgFromState* &&__from_state, *_ArgToState* &&__to_state)

Constructs a *ztd::text::validate_result*, defaulting the error code to *ztd::text::encoding_error::ok* if not provided.

Parameters

- **__input** – **[in]** The input range to store.
- **__is_valid** – **[in]** Whether or not the validation succeeded.
- **__from_state** – **[in]** The state related to the encoding that was used to do validation.
- **__to_state** – **[in]** The state related to the encoding that was used to do validation.

inline explicit constexpr **operator bool**() const noexcept

A conversion for use in if statements and conditional operators.

Returns Whether or not the result is valid or not.

Public Members

::ztd::reference_wrapper<*_DecodeState*> **from_state**

A reference to the state of the associated Encoding used for validating the input.

::ztd::reference_wrapper<*_EncodeState*> **to_state**

A reference to the state of the associated Encoding used for validating the input.

_Input **input**

The reconstructed input_view object, with its .begin() incremented by the number of code units successfully read (can be identical to .begin() on original range on failure).

bool **valid**

Whether or not the specified input is valid or not.

propagate_error

This helper function processes an error for a transcoding operation and shuffles a result through its decode step and encode step *error handlers*. Nominally used after a solely decode portion of a transcode operation fails.

If the user is doing a direct conversion and can simply call the encode portion of the error handler directly, calling this function can be skipped entirely by the user.

template<typename **_Result**, typename **_Output**, typename **_ToEncoding**, typename **_EncodeErrorHandler**, typename **_ToState**, typename **_ToInputProgress**, typename **_ToOutputProgress**, typename **_Input**, typename **_Intermediate**, typename **_FromState**>
constexpr auto **propagate_error**(*_Output* &&__output, *_ToEncoding* &&__to_encoding, decode_result<*_Input*, *_Intermediate*, *_FromState*> &&__result, *_EncodeErrorHandler* &&__encode_error_handler, *_ToState* &__to_state, *_ToInputProgress* &&__to_input_progress, *_ToOutputProgress* &&__to_output_progress)

Transcoding helper. Takes the given `__to_encoding` and `__encode_error_handler` and launders the failed *ztd::text::decode_result* through it, producing a *ztd::text::encode_result* and transforming that into the desired *ztd::text::transcode_result* type.

Remark This function is a helper whose sole purpose is to ensure that the other half of error handling is called by transcode-style functions written by the end user (e.g., writing overriding hooks for ztd::text::transcode). This function attempts to take care of any unread/unwritten characters and other minor points in its pursuit of properly making sure the error manifests on the other side.

Template Parameters **_Result** – The exact transcode result type to use.

Parameters

- **__output** – **[in]** The output view to be writing into.
- **__to_encoding** – **[in]** The desired encoding that performs the encode portion of the transcoding step.
- **__result** – **[in]** The result value that has an error on it.
- **__encode_error_handler** – **[in]** The error handler to mill the `__result` and other relevant information through.
- **__to_state** – **[in]** The current state of the encoding step of the transcode operation.
- **__to_input_progress** – **[in]** Any unread output characters in any intermediate between the (failed) decode and encode operations.
- **__to_output_progress** – **[in]** Any unread output characters in any intermediates between the (failed) decode and encode operations.

template<typename **_Output**, typename **_ToEncoding**, typename **_EncodeErrorHandler**, typename **_ToState**, typename **_ToInputProgress**, typename **_ToOutputProgress**, typename **_Input**, typename **_Intermediate**, typename **_FromState**>
constexpr auto **propagate_error**(*_Output* &&__output, *_ToEncoding* &&__to_encoding, decode_result<*_Input*, *_Intermediate*, *_FromState*> &&__result, *_EncodeErrorHandler* &&__encode_error_handler, *_ToState* &__to_state, *_ToInputProgress* &&__to_input_progress, *_ToOutputProgress* &&__to_output_progress)

Takes the given `__to_encoding` and `__encode_error_handler` and launders the failed *ztd::text::decode_result* through it, producing a *ztd::text::encode_result* and transforming that into the desired *ztd::text::transcode_result* type.

Remark This function is a helper whose sole purpose is to ensure that the other half of error handling is called by transcode-style functions written by the end user (e.g., writing overriding hooks for ztd::text::transcode). This function attempts to take care of any unread/unwritten characters and other minor points in its pursuit of properly making sure the error manifests on the other side. Unlike it's counterpart, this function does not take an `_Result` template parameter and instead deduces the returned transcode result type from inputs.

Parameters

- **__output** – **[in]** The output view to be writing into.
- **__to_encoding** – **[in]** The desired encoding that performs the encode portion of the transcoding step.
- **__result** – **[in]** The result value that has an error on it.
- **__encode_error_handler** – **[in]** The error handler to mill the `__result` and other relevant information through.
- **__to_state** – **[in]** The current state of the encoding step of the transcode operation.

- **__to_input_progress** – **[in]** Any unread output characters in any intermediate between the (failed) decode and encode operations.
- **__to_output_progress** – **[in]** Any unread output characters in any intermediates between the (failed) decode and encode operations.

1.10 Progress & Future Work

This is where the status and progress of the library will be kept up to date. You can also check the Issue Tracker for specific issues and things being worked on! We also maintain a *very large list of encodings*, so you can check if a specific encoding you are looking for is supported (and if you will need to implement an *Encoding Object* for it).

1.10.1 Copyable State

Right now, all state parameters are assumed to be move-only. This is detrimental to creating cheap views like `.code_points()` on `basic_text_view`, and harms other types as well. Work should be done either to make copyable state, or allow passing state in more effectively (we currently do the passing technique at the moment).

- Do all states need to be copyable? Can it be done selectively? (At the moment: `basic_text_view` and `text_view` very well may need it, and as more Shift-State encodings become a part of the library, even more need...)

1.10.2 Transcoding Iterators/Transcode View

Right now these types would not work especially well for input and output ranges. They should be modified just like the internal `ztd::text::__txt_detail::__encoding_iterator` class types, so that they work with `input_iterator` and `output_iterator` types.

- Improve constructor delegation and make sure to explicitly implement default construction vs. letting it happen with `=default` (which does not work for some of the base types present).
- Modify implementation to cache data and position when an input or output iterator is detected.
- Return `const value_type&` for `reference` to enable C++20 ranges to work properly.
- Mark as `enable_borrowed_range` when C++20 is detected.

1.10.3 Normalization

`ztd::text::nfkd/nfk/nfc/nfkc/fcc` are all skeletons right now that need to be filled out for the purposes of giving this library normalization views.

- nfkc
- nfc
- nfkd
- nfd
- Hook up to `basic_text_view` and `basic_text` when finished

1.10.4 basic_text_view

`ztd::text::basic_text_view<Encoding, NormalizationForm, Range, ...>` is to be the premiere view for looking at text and preserving both the normalization and encoding form during insertion and erasure. It is not fully implemented yet, even though basic skeletons exist for it in the code base.

- Grapheme Cluster Iterators
- Code Point iterators
- Grapheme Cluster Iterators
- Comparison operators (If the normalization form is the same and *is_bitwise_transcoding_compatible*, then `memcmp`. If just normalization form and encoding is same, `memcmp`. Otherwise, code point by code point comparison.)

1.10.5 basic_text

`ztd::text::basic_text<Encoding, NormalizationForm, Storage, ...>` is to be the premiere container for storing text and preserving both the normalization and encoding form during insertion and erasure. It is not fully implemented yet, even though basic skeletons exist for it in the code base.

- Code Point iterators/ranges
- Grapheme Cluster Iterators
- Comparison operators (If the normalization form is the same and *is_bitwise_transcoding_compatible*, then `memcmp`. If just normalization form and encoding is same, `memcmp`. Otherwise, code point by code point comparison.)
- Insertion (Fast normalization-preserving splicing/inserting algorithm)
- Deletion
- Converting Constructors between compatible types (errors the same way *lossy conversion protection* describes if they are not compatible, forcing a user to pass in an error handler.)

1.10.6 iconv

There should be an encoding that loads `iconv` dynamically from the system, if it is present, before using it to do conversions.

1.10.7 cuneicode

There should be a cuneicode-based encoding, for the update C implementation of all of these things.

1.11 Benchmarks (In Progress)

Warning: This isn't finished yet! Come check back by the next major or minor version update.

It's probably fine for now.

Probably!

1.12 Licenses, Thanks and Attribution

`ztd.text` is dual-licensed under either the Apache 2 License, or a corporate license if you bought it with special support. See the LICENSE file or your copy of the corporate license agreement for more details!

1.12.1 Third-party Dependencies and Code

All third-party code is listed in the NOTICE file. It is also reproduced here. In particular:

- Martin Moene; span-lite (Boost 1.0 License) - Code included directly and made available if a C++20 is not present.

We thank Martin Moene for their hard work.

1.12.2 Previous and Related Works

Previous attempts at text and text handling libraries were made by various authors. We note them here:

- Tom Honermann; text_view.
- Zach Laine; Boost.Text.
- Henri Sivonen; encoding_rs.
- rmf; libogonek.

Their work was groundbreaking when it first came about and employed similar concepts found in this library. We thank them for their efforts in moving Text Encoding, Unicode, and Systems Programming forward.

1.12.3 Helping Hands

Whether it's just a little bit of time, a point towards the right direction, or some ideas, this library builds upon a lot of collective knowledge and effort. Here we list some of the filks who have spent some time doing this best to make sure we have the greatest text library on the planet for C++:

- CopperSpice; Talking over many of their design struggles with trying to make better text in CopperSpice/Qt (https://www.youtube.com/watch?v=w_kD-qAkoH0)
- Luna & Lambda Kitten: Kick-starting better support for Clang / Apple (https://twitter.com/lambdakitten/status/1418240846638485510)
- Much of rmf and Henri Sivonen's writings and thoughts on the subjects of Unicode.
- All of Tom Honermann's previous work on Unicode, Text Processing, and Standardization.

1.12.4 Charitable Contributions

`ztd.text` has been made possible by charitable contributions from patrons and sponsors around the world:

- Shepherd's Oasis, LLC (https://soasis.org)
- Jane Lusby
- Orfeas Zafeiris
- Tom Honermann
- Lily Foster
- Camilla Löwy
- Leonardo Lima
- Piotr Piatkowski
- Cynthia Coan
- Johan Andersson
- Erekose Craft
- Christopher Crouzet
- Michael Schellenberger Costa
- Turig Eret
- Brent Beer
- Matt Godbolt
- Erica Brescia
- Carol Chen
- Jeremy Jung
- Max Stoiber
- Evan Lock
- Anil Kumar
- Vincent Weevers
- Ólafur Waage
- Jeff Trull
- Davide Faconti
- Anthony Nandaa
- Christ Drozdowski
- Douglas Creager
- superfunc
- Michael Caisse
- Joshua Fisher
- Billy O'Neal
- Sy Brand

- Eric Tremblay
- Michał Dominiak
- Zach Toogood
- beluga
- Alex Gilding
- Kirk Shoop
- Alex Hadd
- Jimmy "junoravin"
- Joel Falcou
- Pascal Menuet
- Elias Daler
- Randomnetcat
- Robert Maynard
- Martin Hořeňovský
- Hana Dusíková
- 7 more private sponsors
- And many, many more!

(If you are new to being a patron, sponsor, or donator and you don't see your name here, I may have bungled the export list, so please e-mail opensource@soasis.org!)

1.13 Bibliography

These are all the resources that this documentation links to, in alphabetical order.

encoding_rs Henri Sivonen. "encoding_rs". February 2021. URL: https://github.com/libogonek/ogonek. A Rust library for performing encoding and decoding tasks. Takes a byte-based approach to handling encodings and decodings. The developer of this library worked on text for a very long time on Mozilla Firefox, and has great insight into the field of text on their blog, https://hsivonen.fi.

Fast UTF-8 Bob Steagall. "Fast Conversion from UTF-8 with C++, DFAs, and SSE Intrinsics". September 26th, 2019. URL: https://www.youtube.com/watch?v=5FQ87-Ecb-A. This presentation demonstrates one of the ways an underlying fast decoder for UTF-8 can be written, rather than just letting the default work. This work can be hooked into the *conversion function extension points* location.

Fast UTF-8 Validation Daniel Lemire. "Ridiculously fast unicode (UTF-8) validation". October 20th, 2020. URL: https://lemire.me/blog/2020/10/20/ridiculously-fast-unicode-utf-8-validation/. This blog post is one of many that presents a faster, more optimized way to validate that UTF-8 is in its correcty form.

glibc-25744 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. This bug report details the problem with the C standard library's ability to handle multiwide characters. This problem is also present in the "1:N" and "N:1" rules in the C++ standard library.

iconv Bruno Haible and Daiki Ueno. libiconv. August 2020. URL: https://savannah.gnu.org/git/?group=libiconv. A software library for working with and converting text. Typically ships on most, if not all, POSIX and Linux systems.

ICU Unicode Consortium. "International Components for Unicode". April 17th, 2019. URL: https://github.com/hsivonen/encoding_rs The premiere library for not only performing encoding conversions, but performing other Unicode-related algorithms on sequences of text.

libogonek

R. Martinho Fernandes. "libogonek: A C++11 Library for Unicode". September 29th, 2019. URL: http://site.icu-project.org/ One of the first influential C++11 libraries to bring the concept of iterators and ranges to not only encoding, but normalization and others. It's great design was only limited by how incapable C++11 as a language was for what its author was trying to do.

n2282 Philip K. Krause. "N2282 - Additional multibyte/wide string conversion functions". June 2018. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2282.htm. This paper attempted to add a few unicode functions to the list of things to do without changing anything.

Non-Unicode in C++ Henri Sivonen. "P0244 - Text_view: A C++ concepts and range based character encoding and code point enumeration library". URL: https://hsivonen.fi/non-unicode-in-cpp/. A rebuttal to P0244's "strong code points" and "strong code units" opinion. This is talked about in depth in the design documentation for *strong vs. weak code point and code unit types*.

p0244 Tom Honermann. "P0244 - Text_view: A C++ concepts and range based character encoding and code point enumeration library". URL: https://wg21.link/p0244. A C++ proposal written by Tom Honermann, proposing some of the first ideas for an extensible text encoding interface and lightweight ranges built on top of that. Reference implementation: https://github.com/tahonermann/text_view.

p1041

R. Martinho Fernandes. "P1041: Make char16_t/char32_t string literals be UTF-16/32". February 2019. URL: https://wg21.link/p1041. This accepted paper enabled C++ to strongly associate all `char16_t` and `char32_t` string literals with UTF-16 and UTF-32. This is not the case for C.

CHAPTER

TWO

INDICES & SEARCH

2.1 Index

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

V

W