ztd.idk

Release 0.0.0

ThePhD & Shepherd's Oasis, LLC

Aug 01, 2022

CONTENTS:

1 Who Is This Library For? 3

2 Indices & Search 37

Index 39

This is the IDK (Industrial Development Kit) library, part of the ZTD collection. The IDK is a small, useful toolbox of supplementary things, including

- The ztd.idk core library:
  - A small collection of type traits, optimizations, and other semi-niche utilities for accelerating development.
  - Small, header-only.
  - CMake: `ztd::idk` (also pulls in `ztd::tag_invoke` and `ztd::version`)
- The ztd.tag_invoke customization point library:
  - Modeled after C++ proposal p1895.
  - Makes for a single extension point to be written, `tag_invoke(...)`, whose first argument is the name of the extension point to be hooking into. E.g., `tag_invoke(tag_t<lua_push>, ...)`.
  - Tiny, header-only.
  - CMake: `ztd::tag_invoke` (also pulls in `ztd::version`)
- The ztd.version configuration macro library:
  - A formalization of the principles found in this post and this post.
  - Mistake-resistant configuration and default-on/off vs. deliberate on/off detection.
  - Infinitesimally tiny, header-only.
  - CMake: `ztd::version`

CHAPTER

ONE

WHO IS THIS LIBRARY FOR?

Ideally, no one.

1.1 Users in the Wild

I mean. … Should you really be using this directly…?

1.2 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.

¯_()_/¯ So far? There's none.

1.3 Configuring the Library

- **`ZTD_DEBUG`:**
  - Signals to ztd.idk and downstream users that this should be considered a "debugging" build.
  - Affects many things, such as error printouts, warnings given, and more.
  - Turned on by default if compiler/platform-specific debug macros are detected, or `NDEBUG` is not defined by the compiler/library.

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

1.4 API Reference

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

1.4.1 C++ APIs

Alignment

This API is identical to the one defined in the C APIs, which can be *found here*.

assertions

This API is identical to the one defined in the C APIs, which can be *found here*.

char(8/16/32)_t

This makes `char(8/16/32)_t` available under the type definitions of `ztd::uchar(8/16/32)_t`. This allows their use uniformly in C and C++, regardless of whether or not the type definition is present in the proper place.

using **ztd::uchar8_t** = ZTD_CHAR8_T_I_
An alias to a unsigned representation of an 8-bit (or greater) code unit type.

Remark

This will be a type alias for the type given in the `ZTD_CHAR8_T` define if it is defined by the user. Otherwise, it will be a type alias for char8_t if present. If neither are available, it will alias `unsigned char` for the type.

---

using **ztd::uchar16_t** = char16_t
An alias to a unsigned representation of an 16-bit (or greater) code unit type.

Remark

This alias will always point to `char16_t`, because C++ has this as a built-in type.

---

using **ztd::uchar32_t** = char32_t
An alias to a unsigned representation of an 32-bit (or greater) code unit type.

Remark

This alias will always point to `char32_t`, because C++ has this as a built-in type.

---

ebco

`ebco` is a way to gain the benefits of what is called the Empty Base Class Optimization (EBCO). It is meant to be used as a base class with the type and tag used to identify the member variable is replacing. Mostly superseded by `[[no_unique_address]]`, except on one compiler that decided to make a Fractally Bad Decision™.

template<typename **_Type**, ::std::size_t **_Tag** = 0, typename = void>

class **ebco**
A class for optimizing the amount of space a certain member of type `_Type` might use.

Remark

The only reason this class continues to be necessary is because of Microsoft Visual C++. Every other compiler respects the new C++20 attribute [[no_unique_address]] - it is only Microsoft that explicitly decided that our opt-in indication that we care more about the object's size is not important.

Template Parameters

- **_Type** – The type of the member.
- **_Tag** – A differentiating tag to separate this member from others when there are multiple bases of the same `_Type`.

Public Functions

ebco() = default
Default construction.

ebco(const *ebco*&) = default
Copy construction.

ebco(*ebco*&&) = default
Move construction.

ebco &**operator=**(const *ebco*&) = default
Copy assignment operator.

ebco &**operator=**(*ebco*&&) = default
Move assignment operator.

inline constexpr **ebco**(const *_Type* &__value) noexcept(::std::is_nothrow_copy_constructible_v<*_Type*>)
Copies the object into storage.

inline constexpr **ebco**(*_Type* &&__value) noexcept(::std::is_nothrow_move_constructible_v<*_Type*>)
Moves the object into storage.

inline constexpr *ebco* &**operator=**(const *_Type* &__value)
noexcept(::std::is_nothrow_copy_assignable_v<*_Type*>)
Copy assigns into the previous object into storage.

inline constexpr *ebco* &**operator=**(*_Type* &&__value)
noexcept(::std::is_nothrow_move_assignable_v<*_Type*>)
Move assigns into the previous object into storage.

template<typename **_Arg**, typename ...**_Args**, typename =
::std::enable_if_t<!::std::is_same_v<::std::remove_reference_t<::std::remove_cv_t<*_Arg*>>, *ebco*> &&
!::std::is_same_v<::std::remove_reference_t<::std::remove_cv_t<*_Arg*>>, *_Type*>>>
inline constexpr **ebco**(*_Arg* &&__arg, *_Args*&&... __args)
noexcept(::std::is_nothrow_constructible_v<*_Type*, *_Arg*, *_Args*...>)
Constructs the object in storage from the given arguments.

inline constexpr *_Type* &**get_value**() & noexcept
Gets the wrapped value.

inline constexpr *_Type* const &**get_value**() const & noexcept
Gets the wrapped value.

inline constexpr *_Type* &&**get_value**() && noexcept
Gets the wrapped value.

endian

The endian enumeration is a very simple `enum class` used to communicate what kind of byte ordering certain parts of the library should use to interpret incoming byte sequences. The C version uses macros and can be found *here*.

The values are `ztd::endian::little`, `ztd::endian::big`, or `ztd::endian::native`.

using `ztd::`**endian** = ::std::endian
An endian enumeration.

Remark

It may include little, big, or native values. The native value can be the same as the little or big values, but if on a middle-endian machine it may be an implementation-defined "middle endian" value that is not equal to either little or big (as on the PDP-11). We don't expect many relevant architectures to be using middle-endian, though.

---

span

A polyfill ("shim", fill-in-layer) meant to emulate std::span.

Available in the namespace uner the name `ztd::span`.

tag_invoke

`tag_invoke` is a way of doing customization points in Modern C++ that is meant to be easier to work with and less hassle for end-users. It follows the paper P1895. A presentation for `tag_invoke` that covers its uses and its improvements over the status quo by Gašper Ažman can be found here.

Warning: doxygenvariable: Cannot find variable "ztd::tag_invoke" in doxygen xml output for project "ztd.idk" from directory: /home/docs/checkouts/readthedocs.org/user_builds/ztdidk/checkouts/latest/documentation/source/_build/cmake-build/documentation/doxygen/xml

template<typename **_Tag**, typename ...**_Args**>

class **is_tag_invocable** : public std::is_invocable<decltype(tag_invoke), *_Tag*, *_Args*...>
Whether or not a given tag type and its arguments are tag invocable.

template<typename **_Tag**, typename ...**_Args**>

constexpr bool **ztd::is_tag_invocable_v** = *is_tag_invocable*<*_Tag*, *_Args*...>::value
A _v alias for *ztd::is_tag_invocable*.

template<typename **_Tag**, typename ...**_Args**>

class **is_nothrow_tag_invocable** : public __is_nothrow_tag_invocable_i<*is_tag_invocable_v*<*_Tag*, *_Args*...>, *_Tag*, *_Args*...>
Whether or not a given tag type and its arguments are both invocable and marked as a `noexcept` invocation.

template<typename **_Tag**, typename ...**_Args**>

constexpr bool **ztd::is_nothrow_tag_invocable_v** = *is_nothrow_tag_invocable*<*_Tag*, *_Args*...>::value
A _v alias for *ztd::is_nothrow_tag_invocable*.

using **ztd::tag_invoke_result** = ::std::invoke_result<decltype(tag_invoke), _Tag, _Args...>
A class representing the type that results from a tag invocation.

using **ztd::tag_invoke_result_t** = typename *tag_invoke_result*<_Tag, _Args...>::type
A _t alias that gives the actual type that results from a tag invocation.

uninit

The `ztd::uninit` type is for holding a type that may be initialized by-default into an uninitialized state (e.g., for C-style arrays that are a member of a class).

template<typename **_Type**>

class **uninit**
A class for holding a value inside of an unnamed union which is composed of two objects, one of `char` and one of `_Type`.

Public Functions

inline constexpr **uninit**()
Constructs an empty placeholder.

template<typename ...**_Args**>
inline constexpr **uninit**(::std::in_place_t, *_Args*&&... __args)
Constructs the `value` from the given arguments.

Parameters **__args** – **[in]** The arguments to construct `value` with.

inline **~uninit**()
An empty destructor. Required, as there is a union object present.

Public Members

char **placeholder**
Placeholder empty value for default / empty initialization, esp. with arrays.

_Type **value**
Actual value.

Friends

inline friend *_Type* &**unwrap**(*uninit* &__wrapped_value) noexcept
Extension point for returning the value inside of this uninitialized type.

inline friend const *_Type* &**unwrap**(const *uninit* &__wrapped_value) noexcept
Extension point for returning the value inside of this uninitialized type.

inline friend *_Type* &&**unwrap**(*uninit* &&__wrapped_value) noexcept
Extension point for returning the value inside of this uninitialized type.

unwrap / unwrap_iterator

Utility extension points to transform a potentially wrapped value (like *ztd::uninit*) so that the "real" value an be used. Often used in the guts of generic code rather than anywhere truly important, but a useful little utility nonetheless.

Warning: doxygenvariable: Cannot find variable "ztd::unwrap_iterator" in doxygen xml output for project "ztd.idk" from directory: /home/docs/checkouts/readthedocs.org/user_builds/ztdidk/checkouts/latest/documentation/source/_build/cmake-build/documentation/doxygen/xml

Warning: doxygenvariable: Cannot find variable "ztd::unwrap" in doxygen xml output for project "ztd.idk" from directory: /home/docs/checkouts/readthedocs.org/user_builds/ztdidk/checkouts/latest/documentation/source/_build/cmake-build/documentation/doxygen/xml

detection

The "detection idiom" is a means to provide "detectors" (code from a `using` type definition whose definition has an expression wrapped in `decltype(...)`) that can tell if a given expression compiles.

template<typename **_Default**, typename **_Void**, template<typename...> typename **_Op**, typename ...**_Args**>

class **detector**
A class to be used for the "detection idiom". Provides `value_t` for the true_type/false_type dichotomy and provides `type` for the detected type.

Remark

This is more efficient and useful at the member declarations level, especially when needing to dispatch to functionality that may or may not exist in wrapped or base classes.

Public Types

using **value_t** = ::std::false_type
The type that provides the `value` static member variable.

using **type** = *_Default*
The type chosen from the detection operation.

class **nonesuch**
A class specifically for the case where the detection idiom cannot detect the requirements.

using `ztd::is_detected` = typename *detector*<*nonesuch*, void, _Op, _Args...>::value_t
A commonly-used alias for getting a `true_type` or `false_type` indicating whether the operation was successful.

template<template<typename...> typename **_Op**, typename ...**_Args**>

constexpr bool `ztd::is_detected_v` = *is_detected*<*_Op*, *_Args*...>::value
A _v shortcut for ztd::is_detected.

using `ztd::detected_t` = typename *detector*<*nonesuch*, void, _Op, _Args...>::type
A `_t` shortcut for using the *ztd::detector* to provide either ztd::nonsuch or the given type as yielded by the operation applied to the arguments.

using `ztd::detected_or` = *detector*<_Default, void, _Op, _Args...>
A shortcut for using the *ztd::detector* to provide either `_Default` or the given type as yielded by the operation applied to the arguments.

is_character

The `is_character` detects the typical "`char`" types in C++ (`char`, `signed char`, `char8_t`, `char16_t`, and `char32_t`).

template<typename **_Type**>

class **is_character** : public std::integral_constant<bool, ::std::is_same_v<*_Type*, char> || ::std::is_same_v<*_Type*, wchar_t> || ::std::is_same_v<*_Type*, unsigned char> || ::std::is_same_v<*_Type*, signed char> || ::std::is_same_v<*_Type*, char16_t> || ::std::is_same_v<*_Type*, char32_t>>
Checks if the given type is one of the plain character types.

template<typename **_Type**>

constexpr bool `ztd::is_character_v` = *is_character*<*_Type*>::value
An _v alias for *ztd::is_character*.

type_identity

The `type_identity` and related `type_identity_t` are useful in controlling function template declarations where the arguments need to have their types prevent from being mutated or changed in undesirable ways. Otherwise, it does exactly what it says on the tin: launders the given type parameter into the `::type` aspect.

template<typename **_Type**>

class **type_identity**
A type for giving the exact same type out as was put in.

using `ztd::type_identity_t` = typename *type_identity*<_Type>::type
A `_t` typename alias for *ztd::type_identity*.

1.4.2 C APIs

Alignment

These APIs aid in aligning pointers and types. They are typically available for both C and C++.

ZTD_ASSUME_ALIGNED(_ALIGNMENT, ...)
Returns a pointer suitable-aligned for `_ALIGNMENT`.

Remark

This function does NOT align the pointer, just marks it as such. This uses builtins or other tricks depending on the compiler. It can trigger Undefined Behavior if it is not properly checked and protected against, so make sure the pointer is properly aligned.

Parameters

- **_ALIGNMENT** – **[in]** An integer constant expression indicating the alignment of the pointer value.
- **...** – **[in]** The pointer to assume alignment of.

Returns A pointer (assumed to be) suitably-aligned to `_ALIGNMENT`.

Assertions

This API defines 2 assertion macros. One is named `ZTD_ASSERT`, and the other is named `ZTD_ASSERT_MESSAGE`. The first takes only one or more conditional tokens, the second takes a mandatory message token as the first parameter, and then one or more conditional parameters.

The user can override the behavior of each of these by defining both of `ZTD_ASSERT_USER` and `ZTD_ASSERT_MESSAGE_USER`.

When *debug mode is detected* and user-defined assertions are not macro-defined, then a default implementation is used. Typically, these:

- check the condition, and if it is true:
  - print (`std::cerr` or `fprintf(stderr, ...)`, depending on the language) a message including line, file, etc.; and,

- exit the program cleanly (`std::terminate` or `exit`, depending on the language)

Note that no side-effects should ever go into assertions, because assertions can be compiled to do nothing.

ZTD_ASSERT(...)
A macro for asserting over a given (set of) conditions.

Remark

The conditions must result in a value that is convertible to a boolean in a boolean context. This macro does nothing when ZTD_DEBUG is not detected. (It will still (void)-cast the used items, to prevent unused warnings.) If the condition is not reached, this function will perform either a user-defined action or terminate/exit (not abort).

Parameters

- **...** – **[in]** The conditional expressions to check against.

ZTD_ASSERT_MESSAGE(_MESSAGE, ...)
A macro for asserting over a given (set of) conditions.

Remark

The conditions must result in a value that is convertible to a boolean in a boolean context. This macro does nothing when ZTD_DEBUG is not detected. (It will still (void)-cast the used items, to prevent unused warnings.) If the condition is not reached, this function will perform either a user-defined action or terminate/exit (not abort).

Parameters

- **_MESSAGE** – **[in]** The message to pass through.
- **...** – **[in]** The conditional expressions to check against.

Bit Intrinsics

Bit intrinsics are functions that map as closely as possible to behavior and functionality in ISAs without needing to deal with the undefined behavior and non-portability of said architectures. It provides vital functionality that can greatly speed up work on specific kinds of bit operations. The provided intrinsics here are a large subset of the most efficient operations, offered in various flavors for ease-of-use.

"Leading" refers to the most significant bit in a given value. This is the "left side" of an integer when writing source code, such that `0b10` has a most significant bit of `1`. "Trailing" refers to the least significant bit in a given value. This is the "left side" of an integer when writing source code, such that `0b10` has a least significant bit of `0`.

ztdc_count_ones(...)
Counts the number of ones in a given unsigned integer.

Parameters

- **...** – **[in]** The input value.

Returns An `int` (or suitably large signed integer type) with the count.

ztdc_count_zeros(...)
Counts the number of zeros in a given unsigned integer.

Parameters

- `...` – **[in]** The input value.

Returns An `int` (or suitably large signed integer type) with the count.

`ztdc_count_leading_zeros`(...)
Counts the number of leading zeros in a given unsigned integer.

Parameters

- `...` – **[in]** The input value.

Returns An `int` (or suitably large signed integer type) with the count.

`ztdc_count_trailing_zeros`(...)
Counts the number of trailing zeros in a given unsigned integer.

Parameters

- `...` – **[in]** The input value.

Returns An `int` (or suitably large signed integer type) with the count.

`ztdc_count_leading_ones`(...)
Counts the number of leading ones in a given unsigned integer.

Parameters

- `...` – **[in]** The input value.

Returns An `int` (or suitably large signed integer type) with the count.

`ztdc_count_trailing_ones`(...)
Counts the number of trailing ones in a given unsigned integer.

Parameters

- `...` – **[in]** The input value.

Returns An `int` (or suitably large signed integer type) with the count.

`ztdc_first_leading_zero`(...)
Finds the first trailing zero in a given unsigned integer value.

Parameters

- `...` – **[in]** The input value.

Returns If the bit is not found, returns `0`. Otherwise, returns an `int` (or suitably large enough signed integer) indicating the index of the found bit, **plus one**.

`ztdc_first_trailing_zero`(...)
Finds the first trailing zero in a given unsigned integer value.

Parameters

- `...` – **[in]** The input value.

Returns If the bit is not found, returns `0`. Otherwise, returns an `int` (or suitably large enough signed integer) indicating the index of the found bit, **plus one**.

`ztdc_first_leading_one`(...)
Finds the first leading one in a given unsigned integer value.

Parameters

- `...` – **[in]** The input value.

Returns If the bit is not found, returns `0`. Otherwise, returns an `int` (or suitably large enough signed integer) indicating the index of the found bit, **plus one**.

`ztdc_first_trailing_one`(...)
Finds the first trailing one in a given unsigned integer value.

Parameters

- `...` – **[in]** The input value.

Returns If the bit is not found, returns `0`. Otherwise, returns an `int` (or suitably large enough signed integer) indicating the index of the found bit, **plus one**.

`ztdc_rotate_left`(_VALUE, ...)
Performs a cyclical shift left.

Remark

If the rotation value is negative, calls ztdc_rotate_right with the negated modulus of the rotation.

Parameters

- `_VALUE` – **[in]** The value to perform the cyclical shift left.
- `...` – **[in]** The rotation value.

`ztdc_rotate_right`(_VALUE, ...)
Performs a cyclical shift right.

Remark

If the rotation value is negative, calls ztdc_rotate_right with the negated modulus of the rotation.

Parameters

- `_VALUE` – **[in]** The value to perform the cyclical shift right.
- `...` – **[in]** The rotation value.

`ztdc_has_single_bit`(...)
Returns whether or not there is a single bit set in this unsigned integer value (this making it a power of 2).

Parameters

- `...` – **[in]** The input value.

`ztdc_bit_width`(...)
Returns the number of bits needed to represent the value exactly.

Parameters

- `...` – **[in]** The input value.

`ztdc_bit_ceil`(...)
Returns the value that is the greatest power of 2 that is less than the input value.

Parameters

- **...** – **[in]** The input value.

Returns `0` when the input value is `0`. Otherwise, produces the greatest power of 2 that is less than the input value.

`ztdc_bit_floor`(...)
Returns the value that is the next power of 2.

Parameters

- **...** – **[in]** The input value.

Returns 1 when the input value is less than or equal to 1. Otherwise, produces the power of 2 that is higher than the input value.

8-bit Memory Reverse

The 8-bit memory reverse swaps 8-bit bytes, regardless of the size of `CHAR_BIT` on the given platform. In order to achieve this in a platform-agnostic manner, it requires that `CHAR_BIT % 8` is `0`. When `CHAR_BIT` is larger than 8 (16, 24, 32, 64, and other values that are multiples of 8), each 8-bit byte within an `unsigned char` is masked off with `0xFF << (8 * byte_index)`, and then serialized for storing/loading. `byte_index` is a value from [0, `CHAR_BIT / 8`) and it is swapped with the reverse 8-bit byte, which is computed with `0xFF << (8 * ((CHAR_BIT / 8) - 1 - byte_index))`.

void **ztdc_memreverse8**(size_t __n, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(*__n*)])
Reverses each 8-bit byte in a region of memory.

Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Remark

Constraints:

- `CHAR_BIT` is a multiple of 8.

Parameters

- **__n** – **[in]** The number of bytes to reverse.
- **__ptr** – **[in]** The pointer whose 8-bit bytes will be reversed.

uintN_t **ztdc_memreverse8uN**(uintN_t __value)
Reverses the 8-bits of a given *N*-width integer type.

Remark

Equivalent to: `ztdc_memreverse8(sizeof(__value), (unsigned char*)(&__value)); return __value;`.

Parameters **__value** – **[in]** The exact-width integer value to be reversed.

8-bit Endian Load/Store

The 8-bit loads and stores put values in a format suitable for bit-by-bit transition over the network or to the filesystem. Because it will serialize exactly enough bytes to memory so that it is suitable for transition over the network, it has the general requirement that when it tries to load *N* bit integers it expects exactly *N* bits to be present in the array. Therefore, `CHAR_BIT % 8` must be `0` and `N % 8` must be `0`.

When `CHAR_BIT` is larger than 8 (16, 24, 32, 64, and other values that are multiples of 8), each 8-bit byte within an `unsigned char` is masked off with `0xFF << (8 * byte_index)`, and then serialized for storing/loading.

Unsigned Variants

void **ztdc_store8_leuN**(uint_leastN_t __value, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Stores an 8-bit byte-specific unsigned integer in little endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores *N* bits, as if by performing `__value = __value & (UINTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Parameters

- **`__value`** – **[in]** The value to be stored.
- **`__ptr`** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

void **ztdc_store8_beuN**(uint_leastN_t __value, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Stores an 8-bit byte-specific unsigned integer in big endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores *N* bits, as if by performing `__value = __value & (UINTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Parameters

- **__value** – **[in]** The value to be stored.
- **__ptr** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

uint_leastN_t **ztdc_load8_leuN**(const unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Loads an 8-bit byte-specific unsigned integer in little endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only loads *N* bits and leaves the rest at 0. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Parameters **__ptr** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

uint_leastN_t **ztdc_load8_beuN**(const unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Loads an 8-bit byte-specific unsigned integer in big endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only loads *N* bits and leaves the rest at 0. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Parameters **__ptr** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

void **ztdc_store8_aligned_leuN**(uint_leastN_t __value, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Stores an 8-bit byte-specific unsigned integer in little endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores *N* bits, as if by performing `__value = __value & (UINTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Precondition The input pointer `__ptr` has an alignment suitable to be treated as an integral type of width *N*.

Parameters

- **`__value`** – **[in]** The value to be stored.
- **`__ptr`** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

void **ztdc_store8_aligned_beuN**(uint_leastN_t __value, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Stores an 8-bit byte-specific unsigned integer in big endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores *N* bits, as if by performing `__value = __value & (UINTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Precondition The input pointer `__ptr` has an alignment suitable to be treated as an integral type of width *N*.

Parameters

- **`__value`** – **[in]** The value to be stored.
- **`__ptr`** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

uint_leastN_t **ztdc_load8_aligned_leuN**(const unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Loads an 8-bit byte-specific unsigned integer in little endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only loads *N* bits and leaves the rest at 0. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Precondition The input pointer `__ptr` has an alignment suitable to be treated as an integral type of width *N*.

Parameters `__ptr` – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

uint_leastN_t **`ztdc_load8_aligned_beuN`**(const unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Loads an 8-bit byte-specific unsigned integer in big endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only loads *N* bits and leaves the rest at 0. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`).

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Precondition The input pointer `__ptr` has an alignment suitable to be treated as an integral type of width *N*.

Parameters `__ptr` – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

Signed Variants

void **`ztdc_store8_lesN`**(int_leastN_t __value, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Stores an 8-bit byte-specific signed integer in little endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores (*N* - 1) bits, as if by performing `__value = __value & (INTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`). The sign bit is serialized into the proper location in the array as the leading (high) bit, and the mask for that is `0x7F << multiple-of-8`.

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Parameters

- **__value** – **[in]** The value to be stored.
- **__ptr** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

void **ztdc_store8_besN**(int_leastN_t __value, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Stores an 8-bit byte-specific signed integer in big endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores (*N* - 1) bits, as if by performing `__value = __value & (INTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`). The sign bit is serialized into the proper location in the array as the leading (high) bit, and the mask for that is `0x7F << multiple-of-8`.

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Parameters

- **__value** – **[in]** The value to be stored.
- **__ptr** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

int_leastN_t **ztdc_load8_lesN**(const unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Loads an 8-bit byte-specific signed integer in little endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores (*N* - 1) bits, as if by performing `__value = __value & (INTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`). The sign bit is serialized into the proper location in the array as the leading (high) bit, and the mask for that is `0x7F << multiple-of-8`.

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Parameters **__ptr** – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

int_leastN_t **ztdc_load8_besN**(const unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Loads an 8-bit byte-specific signed integer in big endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores (*N* - 1) bits, as if by performing `__value = __value & (INTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`). The sign bit is serialized into the proper location in the array as the leading (high) bit, and the mask for that is `0x7F << multiple-of-8`.

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Parameters `__ptr` – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

void **ztdc_store8_aligned_lesN**(int_leastN_t __value, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Stores an 8-bit byte-specific signed integer in little endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores (*N* - 1) bits, as if by performing `__value = __value & (INTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0, `CHAR_BIT`). The sign bit is serialized into the proper location in the array as the leading (high) bit, and the mask for that is `0x7F << multiple-of-8`.

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Precondition The input pointer `__ptr` has an alignment suitable to be treated as an integral type of width *N*.

Parameters

- `__value` – **[in]** The value to be stored.
- `__ptr` – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

void **ztdc_store8_aligned_besN**(int_leastN_t __value, unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Stores an 8-bit byte-specific signed integer in big endian format in the array pointed to by `__ptr` by reading from `__value`.

Remark

Only stores (*N* - 1) bits, as if by performing `__value = __value & (INTN_MAX)` first. Each 8-bit byte is considered according to `0xFF << multiple-of-8`, where `multiple-of-8` is a multiple of in the range [0,

CHAR_BIT). The sign bit is serialized into the proper location in the array as the leading (high) bit, and the mask for that is 0x7F << multiple-of-8.

Constraints

- CHAR_BIT is a multiple if 8.
- *N* is a multiple of 8.

Precondition The input pointer __ptr has an alignment suitable to be treated as an integral type of width *N*.

Parameters

- **__value** – **[in]** The value to be stored.
- **__ptr** – **[in]** A non-null pointer to the at least N / CHAR_BIT elements.

int_leastN_t **ztdc_load8_aligned_lesN**(const unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Loads an 8-bit byte-specific signed integer in little endian format in the array pointed to by __ptr by reading from __value.

Remark

Only stores (*N* - 1) bits, as if by performing __value = __value & (INTN_MAX) first. Each 8-bit byte is considered according to 0xFF << multiple-of-8, where multiple-of-8 is a multiple of in the range [0, CHAR_BIT). The sign bit is serialized into the proper location in the array as the leading (high) bit, and the mask for that is 0x7F << multiple-of-8.

Constraints

- CHAR_BIT is a multiple if 8.
- *N* is a multiple of 8.

Precondition The input pointer __ptr has an alignment suitable to be treated as an integral type of width *N*.

Parameters **__ptr** – **[in]** A non-null pointer to the at least N / CHAR_BIT elements.

int_leastN_t **ztdc_load8_aligned_besN**(const unsigned char __ptr[ZTD_STATIC_PTR_EXTENT_I_(N / CHAR_BIT)])

Loads an 8-bit byte-specific signed integer in big endian format in the array pointed to by __ptr by reading from __value.

Remark

Only stores (*N* - 1) bits, as if by performing __value = __value & (INTN_MAX) first. Each 8-bit byte is considered according to 0xFF << multiple-of-8, where multiple-of-8 is a multiple of in the range [0, CHAR_BIT). The sign bit is serialized into the proper location in the array as the leading (high) bit, and the mask for that is 0x7F << multiple-of-8.

Constraints

- `CHAR_BIT` is a multiple if 8.
- *N* is a multiple of 8.

Precondition The input pointer `__ptr` has an alignment suitable to be treated as an integral type of width *N*.

Parameters `__ptr` – **[in]** A non-null pointer to the at least `N / CHAR_BIT` elements.

c_span

`c_span` is a type that is generated by defining the macro `ZTD_IDK_C_SPAN_TYPE` to a specific type name and including the header `#include <ztd/idk/c_span.g.h>`. Occasionally, some types include spaces or similar, and therefore need some additional tweaking in order to handle it all properly. This comes up to forming 3 different macros which can help control configuration:

- `ZTD_IDK_C_SPAN_TYPE`, the type;
- `ZTD_IDK_C_SPAN_TYPE_IS_CONST`, an optional definition that, if defined, must be either 1 or 0. If 1, it indicates that the stored pointer should be to a *const T* type;
- `ZTD_IDK_C_SPAN_TYPE_NAME`, an optional name of the type if it should not be derived directly from the type itself (defaults to `ZTD_ID_C_SPAN_TYPE`);
- `ZTD_IDK_C_SPAN_SIZE_TYPE`, an optional type name used to control the type for the storage of the `size` (defaults to `size_t`).
- `ZTD_IDK_C_SPAN_SIZE_TYPE_NAME`, an optional suffix for the `c_span`'s name to override the default which is generated from the type (defaults to `ZTD_ID_C_SPAN_SIZE_TYPE`); and,
- `ZTD_IDK_C_SPAN_NAME`, an optional override for the entire name of the structure and its functions (ignores all previous name-based derivations).
- `ZTD_IDK_C_SPAN_SIZE_FIRST`, an optional definition that, if defined, must be either 1 or 0. If 1, it indicates that the size member should go first.

The final name is composed of either just the type name suffixed on `c_span`; the type name and the size type name (if defined) suffixed onto `c_span`; or, the the full name provided in the override.

Important: Any macros that are consumed by this header are undefined by the end of the header, including the ones listed above.

The `<ztd/idk/c_span.h>` header includes some common definitions of a `c_span` to be used, most notably `c_span_uchar`. The documentation below is for `c_span_uchar`, but works for all entities.

Note: `ztd_generic_type` is a name used as a placeholder. When it appears as a name (or within a name) or a type, it can be substituted out for another type name!

Structure + Functions

void **copy_c_span**(*c_span* *__destination, *c_span* __source)
Copies on *c_span* into the memory of another.

Remark

Preconditions:

- `__destination != NULL`

Parameters

- **__destination** – **[in]** Pointer to the destination.
- **__source** – **[in]** The source span to copy.

c_span **make_c_span**(ztd_generic_type *__first, ztd_generic_type *__last)
Create a *c_span* from two pointers which denote a region of memory.

Remark

Preconditions:

- `__first` < `__last` (`__first` is reachable from `__last`).
- `__first` and `__last` are part of the same storage and form a valid range.

Parameters

- **__first** – **[in]** The start of the region of memory, inclusive.
- **__last** – **[in]** The end of the region of memory, non-inclusive.

c_span **make_sized_c_span**(ztd_generic_type *__first, size_t __size)
Create a *c_span* from two pointers which denote a region of memory.

Remark

Preconditions:

- `__first` and `__first + __size` are part of the same storage and form a valid range.

Parameters

- **__first** – **[in]** The start of the region of memory, inclusive.
- **__size** – **[in]** The number of elements of the region of memory.

ztd_generic_type *c_span_data(*c_span* __span)
Retrieves a pointer to the start of this span of memory.

Parameters **__span** – **[in]** The "self" object.

size_t c_span_size(*c_span* __span)
Retrieves the size of this span of memory, in number of elements.

Parameters **__span** – **[in]** The "self" object.

bool c_span_empty(*c_span* __span)
Returns whether or not this span is empty.

Parameters **__span** – **[in]** The "self" object.

ztd_generic_type c_span_front(*c_span* __span)
Retrieves the first element of this span of elements.

Remark

Preconditions:

- `__span.size > 0`.

Parameters **__span** – **[in]** The "self" object.

ztd_generic_type c_span_back(*c_span* __span)
Retrieves the last element of this span of elements.

Remark

Preconditions:

- `__span.size > 0`.

---

ztd_generic_type c_span_at(*c_span* __span, size_t __index)
Retrieves the the element at the provided index.

Remark

Preconditions:

- `__span.size > __index`.

Parameters

- **__span** – **[in]** The "self" object.
- **__index** – **[in]** The offset into the span of elements to access.

void **c_span_set**(*c_span* __span, size_t __index, ztd_generic_type __value)
Retrieves the the element at the provided index.

Remark

Preconditions:

- `__span.size > __index`.

Parameters

- **__span** – **[in]** The "self" object.
- **__index** – **[in]** The offset into the span of elements to access.
- **__value** – **[in]** The value to insert.

ztd_generic_type ***c_span_ptr_at**(*c_span* __span, size_t __index)
Retrieves the the element at the provided index.

Remark

Preconditions:

- `__span.size > __index`.

Parameters

- **__span** – **[in]** The "self" object.
- **__index** – **[in]** The offset into the span of elements to access.

ztd_generic_type ***c_span_maybe_ptr_at**(*c_span* __span, size_t __index)
Retrieves the the element at the provided index.

Remark

This function checks size so there are no preconditions.

Parameters

- **__span** – **[in]** The "self" object.
- **__index** – **[in]** The offset into the span of elements to access.

size_t **c_span_byte_size**(*c_span* __span)
Retrieves the size of this span of memory, in number of `unsigned chars`.

Parameters **__span** – **[in]** The "self" object.

c_span **c_span_begin**(*c_span* __span)
An iterator to the beginning of the span of elements.

Parameters **__span** – **[in]** The "self" object.

ztd_generic_type ***c_span_end**(*c_span* __span)
An iterator to the end of the span of elements.

Parameters **__span** – **[in]** The "self" object.

c_span **c_span_subspan**(*c_span* __span, size_t __offset_index, size_t __size)
Creates a smaller span from this span, using the given offset into the span and the desired size.

Remark

Preconditions:

- `__span.size >= (__offset_index + __size)`.

Parameters

- **__span** – **[in]** The "self" object.
- **__offset_index** – **[in]** The offset into the span.
- **__size** – **[in]** The size of the resulting span.

c_span **c_span_subspan_at**(*c_span* __span, size_t __offset_index)
Creates a smaller span from this span, from the given offset. The resulting size is the offset minus the `__span`'s current size.

Remark

Preconditions:

- `__span.size >= __offset_index`.

Parameters

- **__span** – **[in]** The "self" object.
- **__offset_index** – **[in]** The offset into the span.

c_span **c_span_subspan_prefix**(*c_span* __span, size_t __size)
Creates a smaller span from this span, from the given size. The resulting offset is from 0 and has the given size.

Remark

Preconditions:

- `__span.size >= __size`.

Parameters

- **__span** – **[in]** The "self" object.
- **__size** – **[in]** The size of the span, from the beginning.

c_span **c_span_subspan_suffix**(*c_span* __span, size_t __size)
Creates a smaller span from this span, from the given size.

Remark

The resulting offset is from the current span's size minus the desired size, and has the given `__size`.

Preconditions:

- `__span.size >= __size`.

Parameters

- **__span** – **[in]** The "self" object.
- **__size** – **[in]** The size of the span, from the beginning.

struct **c_span**
#include <c_span.h> A structured pointer which keeps its size type, which represents a non-owning view into a buffer.

This type can be initialized with designated initializers.

Remark

This type is meant to be "immutable", which is why the members are marked const. This can present some issues when dealing with, for example, trying to fill out members manually in structures that are heap-allocated. Instead, copy it using `memcpy`, like `memcpy(my_span_ptr, &some_span, sizeof(some_span));` rather than `my_span_ptr->data = some_ptr; my_span_ptr->size = some_size;`.

Defines

ZTD_IDK_C_SPAN_TYPE
The type used to create a new *c_span* type.

Remark

This definition is required. If a type is not provided and the generation header is included, then an error will be produced.

ZTD_IDK_C_SPAN_TYPE_NAME
The name to use when generating the function and structure names.

Remark

This definition is optional. The default is whatever ZTD_IDK_C_SPAN_TYPE is. However, that may be bad since sometimes type names can have spaces in them (such as `unsigned char`). Therefore, one can se names to make it all better, like `uchar` to represent `unsigned char`.

ZTD_IDK_C_SPAN_SIZE_TYPE
The size type used to create a new *c_span* type.

Remark

This definition is optional. The default is `size_t`. In certain cases, a more compact size type may be beneficial than the original size_type. Some may also want to provide a signed type rather than an unsigned type. Note that contract checks will still check for things such as `> 0` or `< size`, even if what is provided is a signed size type (span will not allow negative indexing, where viable).

ZTD_IDK_C_SPAN_SIZE_TYPE_NAME
The name to use when generating the function and structure names.

Remark

This definition is optional. Normally, it would be defaulted to whatever ZTD_IDK_C_SPAN_SIZE_TYPE is. However, that may be bad since sometimes type names can have spaces in them (such as `long long`). Therefore, one can se names to make it all better, like `uchar` to represent `unsigned char`.

ZTD_IDK_C_SPAN_NAME
The whole name of the generated type.

Remark

This definition is optional. When not provided, a sequence of checks are gone through to define a hopefully unique name for the newly generated *c_span*. The first generated attempt is just using *c_span*`{type name}{size type}`, where the size type is only used if ZTD_IDK_C_SPAN_SIZE_TYPE is also defined by you. Otherwise, it defaults to just *c_span*`{type name}` (without the brackets and with the names substituted in).

ZTD_IDK_C_SPAN_SIZE_FIRST
: Whether or not the size type comes before the pointer.

Remark

This definition is optional. When not provided, the default layout is `{ pointer_type , size_type }`. If this is defined and its value is 1, the layout is `{ size_type, pointer_type }`. This can aid when generating certain types that are meant to be compatible with other kinds of buffers, e.g. with POSIX's `iovec`.

char(8/16/32)_t

This makes `char(8/16/32)_t` available under the type definitions of `ztd_char(8/16/32)_t`. This allows their use uniformly in C and C++, regardless of whether or not the type definition is present in the proper place.

typedef ZTD_CHAR8_T_I_ **ztd_char8_t**
: An alias to a unsigned representation of an 8-bit (or greater) code unit type.

Remark

This will be a type alias for the type given in `ZTD_CHAR8_T` if it is defined by the user. Otherwise, it will be a type alias for `char8_t` if present. If neither are available, it will alias `unsigned char` for the type.

typedef uint_least16_t **ztd_char16_t**
: An alias to a unsigned representation of an 16-bit (or greater) code unit type.

Remark

Certain platforms lack the header `uchar.h`, and therefore sometimes this will be aliased to its standard-defined `uint_least16_t` rather than just `char16_t`.

typedef uint_least32_t **ztd_char32_t**
: An alias to a unsigned representation of an 32-bit (or greater) code unit type.

Remark

Certain platforms lack the header `uchar.h`, and therefore sometimes this will be aliased to its standard-defined `uint_least32_t` rather than just `char32_t`.

endian

The endian enumeration is a very simple enum used to communicate what kind of byte ordering certain parts of the library should use to interpret incoming byte sequences. The C version uses macros and can be found *here*.

ZTDC_LITTLE_ENDIAN
Little endian, in which the least significant byte as the first byte value.

ZTDC_BIG_ENDIAN
Big endian, in which the most significant byte as the first byte value.

ZTDC_NATIVE_ENDIAN
Native endian, which is one of big, little, or some implementation-defined ordering (e.g., middle endian). If it is big or little, then `ZTD_NATIVE_ENDIAN == ZTD_LITTLE_ENDIAN`, or `ZTD_NATIVE_ENDIAN == ZTD_BIG_ENDIAN`.

Extent

These utilities are for handling extents (arrays and pointers) in C and C++.

ZTD_PTR_EXTENT(...)
Provides the `T arg[static N]` functionality ("sized at least `N` large" hint).

Remark

Expands to the proper notation for C compilers, and expands to nothing for C++ compilers. It is meant to be used as in the declaration: `void f(T arg[ZTD_PTR_EXTENT(N)]);`.

Parameters

- **...** – **[in]** An expression which computes the intended size of the pointer argument.

1.5 Progress & Future Work

This is where the status and progress of the library will be kept. You can also check the Issue Tracker for specific issues and things being worked on!

1.5.1 Containers

We should work on some spicy containers. Probably.

- `fixed_vector` (`noexcept`-throughout)
- `small_vector` (`noexcept`-throughout)
- `vector` (`noexcept`-throughout)

1.5.2 Allocators

We should release some spicy allocators. Maybe wrap a few of the existing ones.

- That shiny new Linux allocator everyone was talking about early in the Pandemic
- `mimalloc` (eww)
- `jemalloc` (requires fixing their godawful build system)

1.6 Benchmarks

As benchmarks are crafted, they will be added to this repository for the relevant materials! Benchmarks are meant to explore various scenarios, and typical are used to improve the quality of code in various places where it can matter most or prove a point to those who are curious.

Unless it is relevant to the benchmark, we program in the most efficient way for the given benchmark for the given tools. For example

Browse the categories of benchmarks:

1.6.1 Bit Function Benchmarks

Note: This is not an exhaustive benchmark suite, nor is it representative of all machines or architectures. All numbers should be taken in the context of the reported environment and standard library below, as well as any additional caveats listed.

The below benchmarks are done on a machine with the following relevant compiler and architecture details:

- *Compiler*: Clang 13.0.0 x86_64-pc-windows-msvc
- *Standard Library*: Microsoft Visual C++ Standard Library, Visual Studio 2022 (Version 17.0)
- *Operating System*: Windows 10 64-bit
- *CPU*: Intel Core i7: 8 X 2592 MHz CPUs
- *CPU Caches*:
  - L1 Data 32 KiB (x4)
  - L1 Instruction 32 KiB (x4)
  - L2 Unified 256 KiB (x4)
  - L3 Unified 6144 KiB (x1)

There are 4 benchmarks, and about 7 kinds of categories for each. Each one represents a way of doing work being measured.

- **naive**: Writing a loop over a `std::array` of `bool` objects.
- **naive_packed**: Writing a loop over a `std::array` of `std::size_t` objects and using masking / OR / AND operations to achieve the desired effect.
- **ztdc_packed** (this library): Writing a loop over a `std::array` of `std::size_t` objects and using bit operations to search for the bit.

- **cpp_std_array_bool**: Using the analogous `std::` algorithm (such as `std::find`) on a `std::array` of `bool` objects.
- **cpp_std_vector_bool**: Using the analogous `std::` algorithm (such as `std::find`) on a `std::vector<bool>`, or one of its custom methods to perform the desired operation.
- **cpp_std_bitset**: Using the analogous `std::` algorithm (such as `std::find`) on a `std::bitset<...>` or one of its custom methods to perform the desired operation.

Each individual bar on the graph includes an error bar demonstrating the standard deviation of that measurement. The transparent circles around each bar display individual samples, so the spread can be accurately seen. Each sample can have anything from ten thousand to a million iterations in it, and for these graphs there's 50 samples, resulting in anywhere from hundreds of thousands to tens of millions of iterations.

Details

As of December 5th, 2021, many standard libraries (including the one tested) use 32-bit integers for their `bitset` and `vector<bool>` implementations. This means that, or many of these, we can beat out their implementations (even if they employ the exact same bit manipulation operations we do) by virtue of using larger integer types.

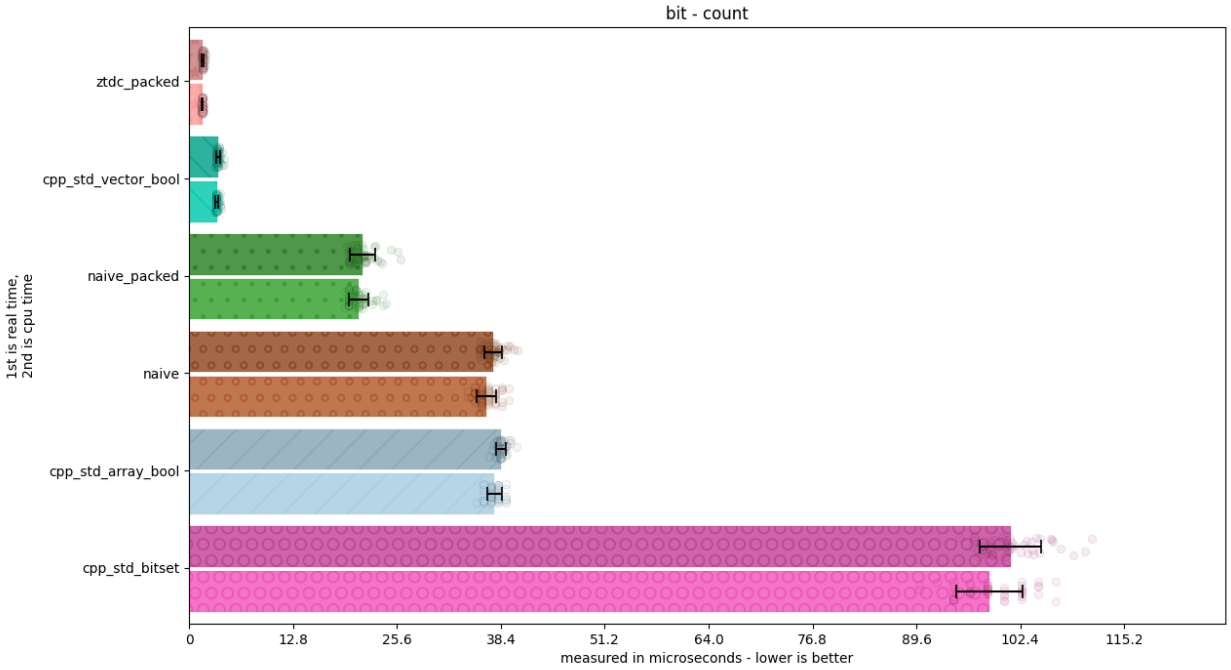
For example, we are faster for the `count` operation despite Michael Schellenberger Costa optimizing MSVC's `std::vector<bool>` iterators in conjunction with its `count` operation, simply because we work on 64-bit integers (and roughly, the graph shows us as twice as fast).

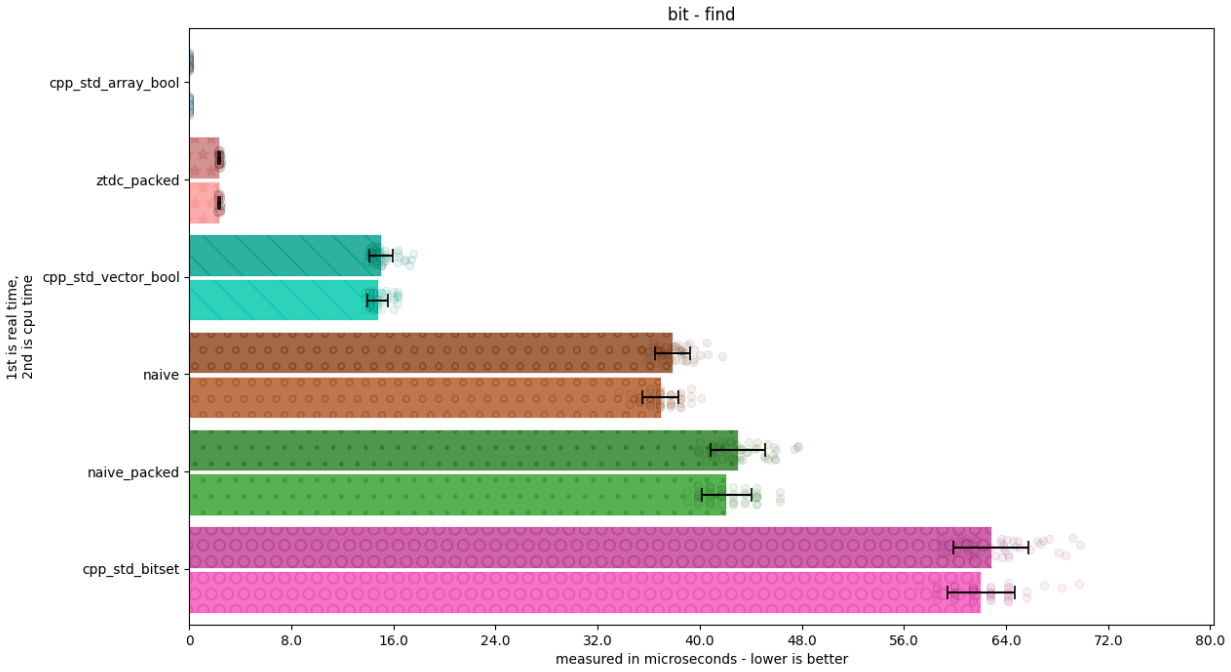
Note: This is a consequence of having a permanently fixed ABI for standard library types, meaning that even if theoretically MSVC could be faster, a person can always beat out the standard library every single time **if** that standard library has long-lasting ABI compatibility requirements.

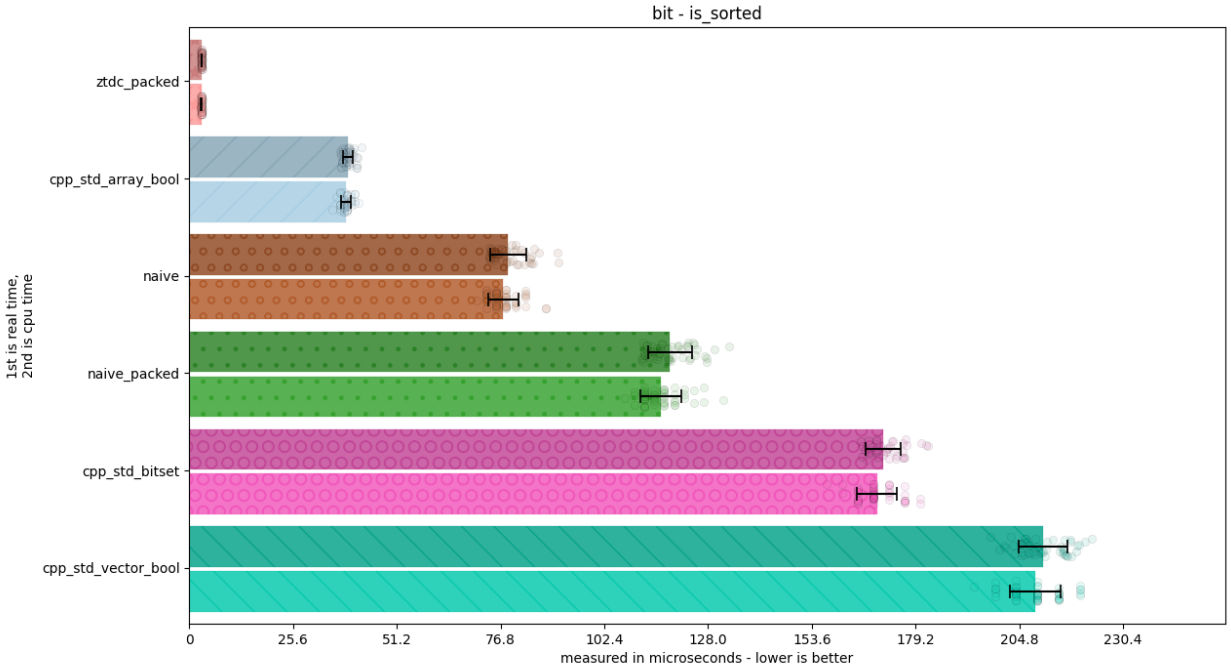
There are further optimizations that can be done in quite a few algorithms when comparisons are involved. For example, `std::find` can be implemented in terms of `memchr` for pointers to fundamental types: this is what makes the "find" for `cpp_std_array_bool` so fast compared to even the bit-intrinsic-improved `ztdc_packed`.

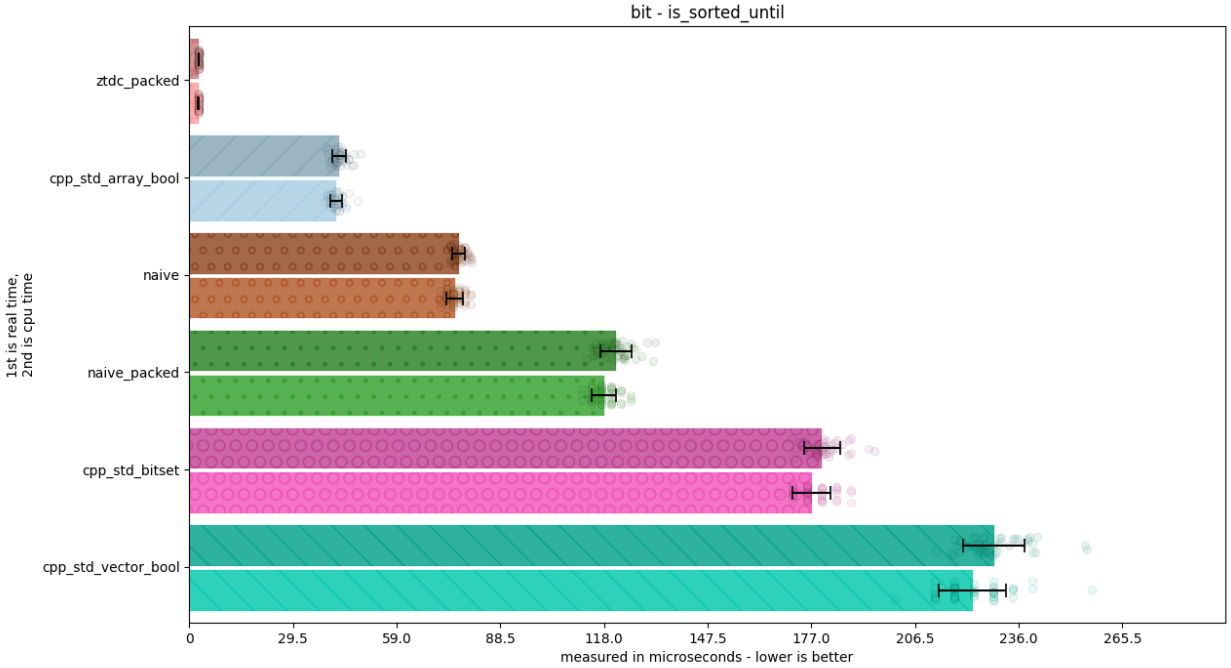
Note: Therefore, despite the last note, standard libraries still perform more optimizations than what a regular user or librarian can do! The Standard Library is not all depressing.

Benchmarks


bit - count
ztdc_packed
cpp_std_vector_bool
naive_packed
naive
cpp_std_array_bool
cpp_std_bitset
1st is real time,
2nd is cpu time
0
12.8
25.6
38.4
51.2
64.0
76.8
89.6
102.4
115.2
measured in microseconds - lower is better


bit - find
cpp_std_array_bool
ztdc_packed
cpp_std_vector_bool
naive
naive_packed
cpp_std_bitset
1st is real time,
2nd is cpu time
0
8.0
16.0
24.0
32.0
40.0
48.0
56.0
64.0
72.0
80.0
measured in microseconds - lower is better


bit - is_sorted
ztdc_packed
cpp_std_array_bool
naive
naive_packed
cpp_std_bitset
cpp_std_vector_bool
1st is real time,
2nd is cpu time
0
25.6
51.2
76.8
102.4
128.0
153.6
179.2
204.8
230.4
measured in microseconds - lower is better


bit - is_sorted_until
ztdc_packed
cpp_std_array_bool
naive
naive_packed
cpp_std_bitset
cpp_std_vector_bool
1st is real time,
2nd is cpu time
0
29.5
59.0
88.5
118.0
147.5
177.0
206.5
236.0
265.5
measured in microseconds - lower is better

1.7 Licenses, Thanks and Attribution

`ztd.idk` 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!

1.7.1 Heartfelt Thanks

Thank you to the Macromancer, Jordan Rose, for suggesting the expansion of "idk" as the "Industrial Development Kit" and Ólafur Waage for deeply encouraging "idk" as the acronym. It's a brilliant name!

1.8 Bibliography

These are all the resources that this documentation links to, in alphabetical order.

() 'Eeey nothin' yet, boss!

CHAPTER

TWO

INDICES & SEARCH

2.1 Index

INDEX

Symbols

¯_()_/¯, **3**

C

c_span (*C++ struct*), 27
c_span_at (*C++ function*), 24
c_span_back (*C++ function*), 24
c_span_begin (*C++ function*), 25
c_span_byte_size (*C++ function*), 25
c_span_data (*C++ function*), 24
c_span_empty (*C++ function*), 24
c_span_end (*C++ function*), 26
c_span_front (*C++ function*), 24
c_span_maybe_ptr_at (*C++ function*), 25
c_span_ptr_at (*C++ function*), 25
c_span_set (*C++ function*), 25
c_span_size (*C++ function*), 24
c_span_subspan (*C++ function*), 26
c_span_subspan_at (*C++ function*), 26
c_span_subspan_prefix (*C++ function*), 26
c_span_subspan_suffix (*C++ function*), 27
copy_c_span (*C++ function*), 23

D

detected_or (*C++ type*), 9
detected_t (*C++ type*), 9
detector (*C++ class*), 8
detector::type (*C++ type*), 9
detector::value_t (*C++ type*), 9

E

ebco (*C++ class*), 5
ebco::ebco (*C++ function*), 5
ebco::get_value (*C++ function*), 6
ebco::operator= (*C++ function*), 5
endian (*C++ type*), 6

I

is_character (*C++ class*), 9
is_character_v (*C++ member*), 9
is_detected (*C++ type*), 9
is_detected_v (*C++ member*), 9
is_nothrow_tag_invocable (*C++ class*), 7
is_nothrow_tag_invocable_v (*C++ member*), 7
is_tag_invocable (*C++ class*), 6
is_tag_invocable_v (*C++ member*), 7

M

make_c_span (*C++ function*), 23
make_sized_c_span (*C++ function*), 23

N

nonesuch (*C++ class*), 9

T

tag_invoke_result (*C++ type*), 7
tag_invoke_result_t (*C++ type*), 7
type_identity (*C++ class*), 10
type_identity_t (*C++ type*), 10

U

uchar16_t (*C++ type*), 4
uchar32_t (*C++ type*), 4
uchar8_t (*C++ type*), 4
uninit (*C++ class*), 7
uninit::~uninit (*C++ function*), 7
uninit::placeholder (*C++ member*), 8
uninit::uninit (*C++ function*), 7
uninit::unwrap (*C++ function*), 8
uninit::value (*C++ member*), 8

Z

ZTD_ASSERT (*C macro*), 11
ZTD_ASSERT_MESSAGE (*C macro*), 11
ZTD_ASSUME_ALIGNED (*C macro*), 10
ztd_char16_t (*C++ type*), 29
ztd_char32_t (*C++ type*), 29
ztd_char8_t (*C++ type*), 29
ZTD_IDK_C_SPAN_NAME (*C macro*), 28
ZTD_IDK_C_SPAN_SIZE_FIRST (*C macro*), 28
ZTD_IDK_C_SPAN_SIZE_TYPE (*C macro*), 28
ZTD_IDK_C_SPAN_SIZE_TYPE_NAME (*C macro*), 28

ZTD_IDK_C_SPAN_TYPE (*C macro*), 27
ZTD_IDK_C_SPAN_TYPE_NAME (*C macro*), 28
ZTD_PTR_EXTENT (*C macro*), 30
ZTDC_BIG_ENDIAN (*C macro*), 30
ztdc_bit_ceil (*C macro*), 13
ztdc_bit_floor (*C macro*), 14
ztdc_bit_width (*C macro*), 13
ztdc_count_leading_ones (*C macro*), 12
ztdc_count_leading_zeros (*C macro*), 12
ztdc_count_ones (*C macro*), 11
ztdc_count_trailing_ones (*C macro*), 12
ztdc_count_trailing_zeros (*C macro*), 12
ztdc_count_zeros (*C macro*), 11
ztdc_first_leading_one (*C macro*), 12
ztdc_first_leading_zero (*C macro*), 12
ztdc_first_trailing_one (*C macro*), 13
ztdc_first_trailing_zero (*C macro*), 12
ztdc_has_single_bit (*C macro*), 13
ZTDC_LITTLE_ENDIAN (*C macro*), 30
ztdc_load8_aligned_besN (*C++ function*), 21
ztdc_load8_aligned_beuN (*C++ function*), 18
ztdc_load8_aligned_lesN (*C++ function*), 21
ztdc_load8_aligned_leuN (*C++ function*), 17
ztdc_load8_besN (*C++ function*), 19
ztdc_load8_beuN (*C++ function*), 16
ztdc_load8_lesN (*C++ function*), 19
ztdc_load8_leuN (*C++ function*), 16
ztdc_memreverse8 (*C++ function*), 14
ztdc_memreverse8uN (*C++ function*), 14
ZTDC_NATIVE_ENDIAN (*C macro*), 30
ztdc_rotate_left (*C macro*), 13
ztdc_rotate_right (*C macro*), 13
ztdc_store8_aligned_besN (*C++ function*), 20
ztdc_store8_aligned_beuN (*C++ function*), 17
ztdc_store8_aligned_lesN (*C++ function*), 20
ztdc_store8_aligned_leuN (*C++ function*), 16
ztdc_store8_besN (*C++ function*), 19
ztdc_store8_beuN (*C++ function*), 15
ztdc_store8_lesN (*C++ function*), 18
ztdc_store8_leuN (*C++ function*), 15