Making Great Docs with Rustdoc

What are the best practices for making quality Rust documentation with Rustdoc?

Jeremy Steward
,
Staff Perception Engineer

Brandon Minor
,
CEO + Co-founder

Programming Insights

Mar 15, 2021

At Tangram Vision, One of the things we've come to love about Rust is the tight integration of the tooling and ecosystem. Rustdoc, the official documentation system, is no exception to this; it's simple to use, creates beautiful pages, and

makes documentation a joy.

We take documentation seriously. Documentation is for many the first entry point into the code, and good documentation is a primary driver of code adoption (along with functionality). This left us wondering: what *are* the best practices for making quality Rust documentation via Rustdoc? Through the course of our research, we found a couple of key resources that the community has provided:

- The rustdoc book: A great place to start if you're looking to learn how to write documentation in your crates.
- RFC 505: API Comment Conventions and RFC 1574: More API Documentation Conventions: Explanations on how the core Rust team documents the standard library and language interfaces.

However, while these are useful resources, we felt that a more approachable guide to crate documentation would be helpful to those starting out with their own crates. Through the course of this article, we'll find that Rust's crate+module structure naturally facilitates great documentation creation; we simply play to Rust's strengths.

Note: this guide is about achieving **good documentation quality**. It does not lay out any technical or formatting guidelines. For instance, we internally wrap our documentation at 120 characters a line... but a 120-character line limit won't do much to improve bad docs.

We assume a bit of knowledge with rustdocs below. If you are unfamiliar with the program, we encourage you to read the rustdoc book linked above as a primer.

Goals of Documentation

Here at Tangram Vision, we structure our documentation around two ideas:

- *What*: Explaining what was built
- *How*: Explaining how to use it

Anything beyond this is generally not helpful, for one singular reason: our documentation assumes that our users know Rust. We believe this assumption is critical to creating good documentation around the What and How. The docs should rarely discuss *why* a decision was made, unless that decision goes against intuition; instead, the docs only explain how to capitalize on that decision in one's

own code.

This guideline of User Knowledge naturally leads us to another rule: *Keep It Short.* The aim is to be succinct by telling the user the What and the How in as little time as possible. This often means using simple, active language and avoiding technical jargon. If a longer discussion is needed, make sure to put the What and the How first, and the discussion after.

Both the What and the How can be seen at all levels of documentation in Rust. We'll see how best to organize that in a large repository below while maintaining consistent style and language.

Documentation Across a Crate

There's a natural top-down pattern to follow for Rust documentation. The top level `lib.rs` or `main.rs` is the first things users see, so it's the perfect spot to introduce the big What ideas. As the documentation gets more into the weeds, from modules to types to functions, documentation shifts more to the How. We'll see this play out as we discuss the different levels below.

We have added relevant links to our own documentation in the realsense-rust crate maintained by Tangram Vision OSS. Check these out for added context.

Crate

Crate-level example here.

The crate documentation in `lib.rs` or `main.rs` should describe the purpose of the crate (the big What) alongside instructions and examples for getting started (the big How). This is a place for big ideas, since the details will come later in the lower levels. Any counter-intuitive design decisions or common gotchas should also be documented here, but remember: the What and the How always come first, and discussion follows.

Notice how the first sections are

1. Summary
2. Features
3. Usage

...all explaining the big What and How. The documentation goes into more detail afterwards, with headings like "Architecture & Guiding Principles", "Prefer Rust-native types to types used through the FFI", etc. However, these sections are there only to explain the *non-intuitive* design decisions that go behind creating an FFI interface like this one. For those that don't care, the What and How are presented first, front and center.

Modules

Module-level example here.

Modules should contain more direct documentation than the crate itself. The focus again should be on describing the types found in that module and how they interact with the rest of the crate. From this alone, users should understand the Why, i.e. why they would reach for a module from your crate.

This can get a bit trickier with sub-modules. Avoid deeply nested sub-modules, since they complicate the structure of a crate. Modules with sub-modules more than two layers deep can probably be flattened out. Exceptions exist, but if this layering is needed, it makes sense to add a **Why** discussion to explain what made this necessary.

Types

Type-level example here.

Types are our primary way of defining abstraction and ontological separations in our code. Documentation here should focus on:

- Construction of the type (when and how, if users are allowed to construct their own).
- Destruction / dropping the type → what happens when you drop a value of a given type? If you don't implement the `Drop` trait, then this is probably OK to ignore.
- Performance characteristics of the type, if it is a data structure.

As one can see, the documentation naturally de-emphasizes the What and builds on the How as we go down. Again, counter-intuitive or non-obvious cases might have a Why, but the What and the How together should suffice.

Functions

Function-level example here.

The last thing to document is functions and associated functions (functions in `impl` blocks). This could include constructors, mutable functions, or data returned by accessors on a type or types. Semantic and code examples are especially welcome here because they describe the How in practical terms.

Common Sections Across a Crate

Crate and module level documentation can be broken down into multiple sections with heading levels 1 (`#` in markdown), 2 (`##`), and 3 (`###`). As you move towards documenting types and functions, aim to be as flat as possible; only use heading level 1 if a section is needed.

These headings are ordered below (in our humble opinion) according to their usefulness for the user in conveying the What and the How for a Rust crate. It's important to note that not all headings need to be present at all times. If it doesn't make sense, don't add it, since it just increases the cognitive burden on the user.

`# Examples`

Examples are, by far, the easiest and most concise way to convey How. They are welcome at all levels: module, type, function, or crate. Write examples for both common use cases and corner cases, like with `Error` or `None` results.

Examples in `rustdoc` documentation will compile and run as doc-tests. This is an **important point**: all code in the documentation will actually compile and run! This means they automatically provide users with starting point for understanding. This is one of Rustdoc's greatest strengths, and it should be utilized whenever possible.

a frame of reference for the HowMake these examples functional code whenever possible. If this is *not* possible, e.g. with an example explaining improper usage, tha frame of reference for the Howen use the heading `text` or `ignore` next to the code block:

```
/// # Examples
```

```
///
/// ```ignore
/// let bummer = this_code_wont_work(); // but it's an illustrati
/// ```
```

Notice the heading is "Examples", plural. Be consistent with the plurality here. Even if there is only one example, the consistency helps with searching.

`# Errors`

At their best, Errors help a user understand why a certain action is prevented by the crate and how to respond to it. It's a rare instance where explaining the *Why* is not just encouraged, but necessary for proper use.

At a function level, this section is only needed if that function returns a `Result` type. In this case, it describes the type returned and when the error should be expected, e.g. an invalid internal state, Foreign Function Interface (FFI) interactions, bad arguments, etc.

Make errors actionable by either passing them to a higher level function or allowing a reaction from the caller. It is easier for users to understand how they got there in the first place by writing error types with reaction in mind. Moreover, if a caller can't act on an error, then there's not a strong reason to present it in the first place.

`# Safety`

First and foremost: Try to minimize unsafe interfaces where possible. Rust is a language built on memory safety, and violating this tenet should only be done with conscious intention. The Safety section should convey that intention.

When documenting Safety, be explicit about what is "unsafe" about the interface and explain best practices for its use with safe code. If possible, try to specify undefined behavior if an unsafe interface will leave your program in an undefined state.

`# Panics`

Use this section if a function can `panic!`. Complete coverage would aim to document any and all calls to `unwrap()`, `debug_assert!`, etc.

document any and all calls to `.unwrap()`, `debug_assert!`, etc.

Complete coverage is a good goal to aim for, but realize that a `panic!` call isn't always necessary. Many cases can be guarded against with small code changes. Returning `Result` types can avoid this entirely in exchange for better error handling. Low-level FFI calls can unwrap and `panic!` if passed a null pointer; yet this can be prevented if you start with `NonNull<t>` as the input, making an `unwrap()` call superfluous.

In any case, you should aim to have all error cases implemented whenever possible. If there is a case that can cause a `panic!`, list it in the docs.

`# Lifetimes`

Include a Lifetimes section if a type or function has special lifetime considerations that need to be taken into consideration by the user. Most of the time, a lifetime itself doesn't need describing; again, always assume users know Rust. Rather, this section should explain why the lifetime has been modeled in a certain way.

Rule of thumb: If you only have one lifetime (explicit or implicit), it probably doesn't need documentation.

```
/// Wrapper type for an underlying C-FFI pointer
///
/// # Lifetimes
///
/// The underlying pointer is generated from some C-FFI.
///
/// Adding a lifetime that only references phantom data may seem
/// and artificial. However, enforcing this lifetime is useful be
/// the C API may have undefined behavior or odd semantics outsid
/// whatever "assumed" lifetime the library writers intended. We
/// this explicit in Rust with the hope of preventing mistakes in
/// this API.
///
pub struct SomeType<'a> {
    /// Pointer to data from a C-FFI
    ///
    /// We need to store this as NonNull because its use in the C
    /// is covariant.
    pointer: std::ptr::NonNull<std::os::raw::c_void>,
    /// Phantom to annotate this type with an explicit lifetime.
    ///
```

```
        /// See type documentation for why this is done.
        _phantom: std::marker::PhantomData<&'a ()="">,
    }
```

The above example shows one instance in where a (single) strange lifetime is applied to enforce Rust's lifetime rules on a type. The lifetime *seems* superfluous, but may exist because there is some implicit assumption in C that is being made more explicit here.

`# Arguments`

Avoid listing arguments explicitly. Instead, names and types themselves should adequately describe arguments and their relationships.

That being said, an `# Arguments` section can make sense if there is a non-obvious assumption about the arguments or their types that needs to be made explicit, like if passing in certain values for a type invokes *undefined behaviour.* In such a case, an `# Arguments` or `# Generic Arguments` section is useful to clarify that care is needed when passing in data.

For generic arguments that may require trait bounds, "document" these by adding `where` clauses to your function or type. This is much more descriptive and useful, and has the added benefit of your compiler doing some of the work to validate these bounds for you.

Take a look at the standard library for examples of good argument names and types.

A Few Last Details

The below points deserve to be noted, but don't necessarily fit into any specific framework.

Document Traits, Not Trait Implementations

Traits are trickier than regular types and associated functions. Since a trait can be applied to multiple types (even those outside of its crate), don't document implementations of a trait. Instead, document the trait itself.

Good

```
/// A trait providing methods for adding different types of numbe
pub trait NumberAdder {
    /// Adds an integer to the type
    fn add_integer(&mut self, number: i32);

    /// Adds a float to the type
    fn add_float(&mut self, number: f32);
}
```

Bad

```
impl NumberAdder for Foo {
    /// Adds an integer to Foo
    fn add_integer(&mut self, number: i32) {
        // ...
    }

    /// Adds a float to Foo
    fn add_float(&mut self, number: f32) {
        // ...
    }
}
```

Traits are used primarily in generic code, so users will look at the trait itself to understand the interfaces. Any two types implementing the same trait should share common functionality; if two implementations of a trait require vastly differing documentation, the trait itself may not be modeled correctly.

References Should Always Be Links

Whenever a reference is made, whether it be to another part of the documentation, a website, a paper, or another crate: add a link to it. This applies even when referencing a common type found in the standard library (e.g. `String`, `Vec`, etc.).

Linking to other parts of the documentation avoids repeating information. It is also an easy way to point to higher level architecture decisions that might affect the

an easy way to point to higher level architecture decisions that might affect the lower level documentation.

`rustdoc` handles documentation links natively. Example:

```
/// See [`String`](std::string::String) documentation for more de
```

Examples and Resources

Open-Source Documentation Examples

We've worked hard at Tangram Vision to follow our own guidelines and create world-class Rust documentation. You can read the fruits of our labor by visiting any of the repositories at our Open-Source software group. The RealSense Rust package that we maintain is some of our most complete documentation, and acts a good starting point.

Templates

Most of these examples come in the form:

```
/// Summary line → what is this
///
/// Longer description of what is returned, or semantics regardin
/// ...
///
/// # Examples
///
/// ```
/// <some-rust-code>
/// ```
```

Types

```
/// Type for describing errors that result from trying to set an
/// on a sensor.
#[derive(Debug)]
pub enum SetOptionError {
    /// The option is not supported on the sensor.
    OptionNotSupported,
    /// The option is supported on the sensor but is immutable.
    OptionIsImmutable,
    /// Setting the option failed due to an internal exception.
    ///
    /// See the enclosed string for a reason why the internal exc
    CouldNotSetOption(String),
}
```

Functions

```
/// Sets a `value` for the provided `option` in `self`.
///
/// Returns `Ok(())` on success, otherwise returns an error.
///
/// # Errors
///
/// Returns [`OptionNotSupported`](SetOptionError::OptionNotSuppo
/// option is not supported on this sensor.
///
/// Returns [`OptionIsImmutable`](SetOptionError::OptionIsImmutab
/// option is supported but is immutable.
///
/// Returns [`CouldNotSetOption`](SetOptionError::CouldNotSetOpti
/// option could not be set due to an internal exception.
///
/// # Examples
///
/// ```
/// let option = SomeOption::Foo;
/// let value = 100.5;
///
/// match sensor.set_option(option, value) {
///     Ok(()) => {
///         println!("Success!");
///     }
///     Err(SetOptionError::OptionNotSupported) => {
///         println!("This option isn't supported, try another on
///     }
```

```
///     Err(SetOptionError::OptionIsImmutable) => {
///         println!("This option is immutable, we can't set it!"
///     }
///     _ => {
///         panic!();
///     }
/// }
/// ```
pub fn set_option(
    &self,
    option: SomeOption,
    value: f32,
) -> Result<(), SetOptionError> {
    // implementation here
    unimplemented!();
}
```

Bad Documentation

```
/// Get the value associated with the provided Rs2Option for the
///
/// # Arguments
///
/// - `option` - The option key that we want the associated value
///
/// # Returns
///
/// An f32 value corresponding to that option within the libreals
/// or None if the option is not supported.
///
pub fn get_option(&self, option: Rs2Option) -> Option<f32>;
```

Why is this bad?

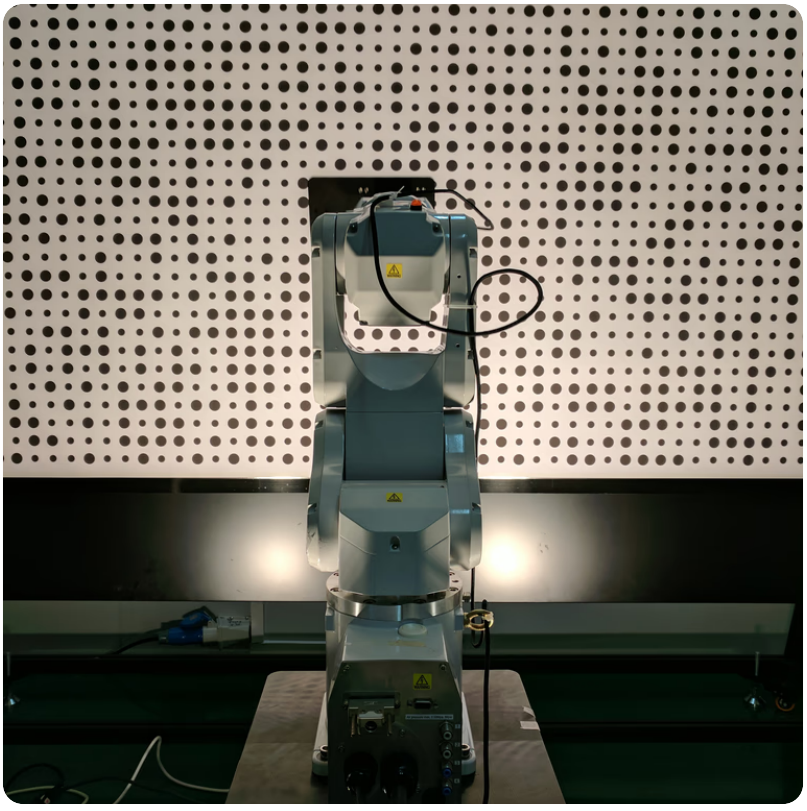
1. The Arguments section is superfluous, since the names and types of the arguments make their use self-evident. See the Arguments section above.
2. The `# Returns` section isn't needed at all. First off, "Returns" should not be a header category; this information can be more concisely expressed in the function summary. Secondly, the return type that is there (`Option<f32>`) makes the possible return values clear already to the user.

A more correct way to write this example would be:

```
/// Get the value associated with the provided `option` for the s
/// or `None` if no such value exists.
pub fn get_option(&self, option: Rs2Option) → Option<f32>;
```

‹ Cleaning Sensors for Perception

Exploring Ansible via Setting Up a WireGuard VPN ›



Can't Wait to Calibrate?

MetriCal delivers industry-leading calibration technology for your cameras and

sensors, ensuring optimal performance in even the most demanding environments.

Plans and Pricing

Learn More

Contact Us

Call in the experts to solve your hardest perception problems. Grow your product and scale quickly.

Note: Tangram Vision needs the contact information you provide to us to contact you about our products and services. You may unsubscribe from these communications at any time.

First Name (required)

Jane

Last Name (required)

Smith

Email (required)

Janesmith@gmail.com

Message

How can we help?

Subscribe to Newsletter

Submit

Tangram Newsletter

Subscribe to our newsletter and keep up with latest calibration insights and Tangram Vision news.

Sign Up

PRODUCT

MetriCal

Enterprise Services

MetriCal Documentatic

COMPANY

Contact Us

Careers

Terms Of Service

Privacy Policy

RESOURCES

Blog

Calibration Desk Refer

LiDAR Visualizer

Depth Sensor Visualize

Cookie Settings