Presentation by Dion Dokter

What’s Rust all about?

tweede golf

Introduction

Who am I?

- Dion Dokter
- Bachelor’s Applied Computer Science at Saxion Enschede
- (Embedded) Rust since 2019
- Joined TG in late-2021
- @Geoxion on Twitter

- LoRaWAN IoT
- UWB Real Time Localization System
- Async IoT with LTE

What to expect today?

Topics:

- Rust language
- Rust embedded
- Rust, FFI & C

The language

Origins

- Announced in 2010 by Mozilla & creator Graydon Hoare
- Aimed to replace C & C++ in Firefox
  - Initially with GC & green threads
- 1.0 version released in 2015
- Major edition upgrade in 2018 & 2021
  - Stable: Old code still compiles
- Now a foundation
- The project is on github
  - 74K stars
  - 4K+ contributors

Why Rust?

According to the website rust-lang.org:

- Performance -> Systems software
  - No runtime
  - No garbage collector
- Reliability -> Safe software
  - Memory safety
  - Thread safety
- Productivity -> Happy developers
  - Friendly compiler
  - Tooling & docs

Technical overview

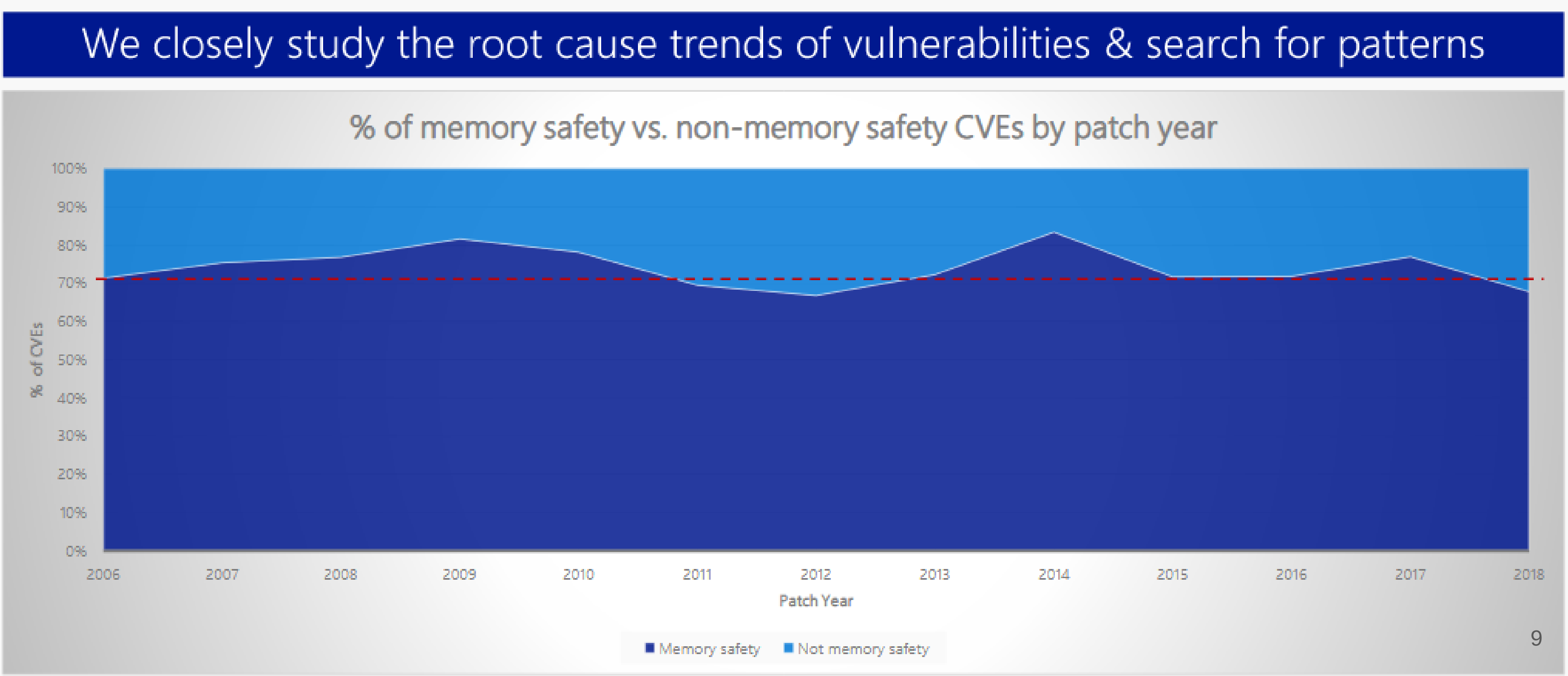
- Compiled language (machine code, not bytecode)
- Strongly statically typed
  - Elaborate type system
- Imperative with functional aspects
- No GC or runtime

Compared to C & C++

- No segfaults*
- No buffer overflows*
- No null pointers*
- No data races*
- Powerful type system
- Unified build system
- Dependency management

* In safe Rust (which is 99% of Rust)

Compared to C & C++



We closely study the root cause trends of vulnerabilities & search for patterns

% of memory safety vs. non-memory safety CVEs by patch year

Why not Rust?

- Compile times
- Learning curve
- No certifications yet
  - Ferrocene
  - AUTOSAR
- Library maturity

Syntax

C origins

Curly bracket style

ML & Haskell infused

Expression oriented

```rust
fn main() {
    println!("Hello, World!");
}

fn is_prime(n: u32) -> bool {
    let limit = (n as f32).sqrt() as u32;

    for i in 2..=limit {
        if n % i == 0 {
            return false;
        }
    }

    true
}
```

Syntax

C origins

Curly bracket style

ML & Haskell infused

Expression oriented

```rust
fn is_prime(n: u32) -> bool {
    let limit = (n as f32).sqrt() as u32;

    (2..=limit).map(|i| n % i).all(|p| p != 0)
}
```

Generates (almost) the same assembly!

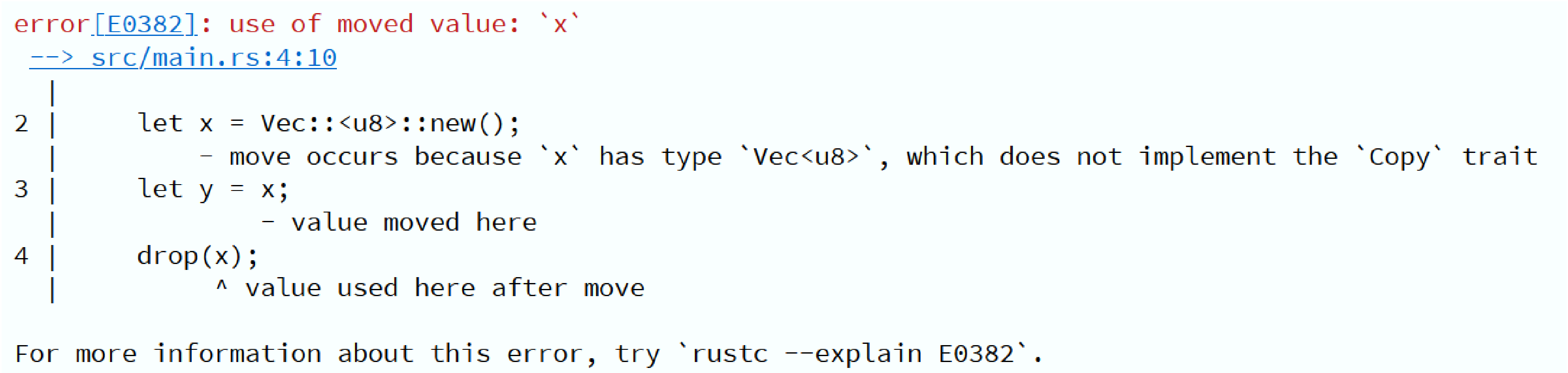
Ownership, moving & borrowing

All references (pointers) are checked at compile time:

- Every value has an owner, the variable
- Access can be borrowed by other variables
  - At most 1 mutable borrow OR infinite immutable borrows
- Ownership can be transferred by moving
- Owner out of scope = value dropped
  - Similar to C++ RAII
  - No GC required

Good compiler feedback

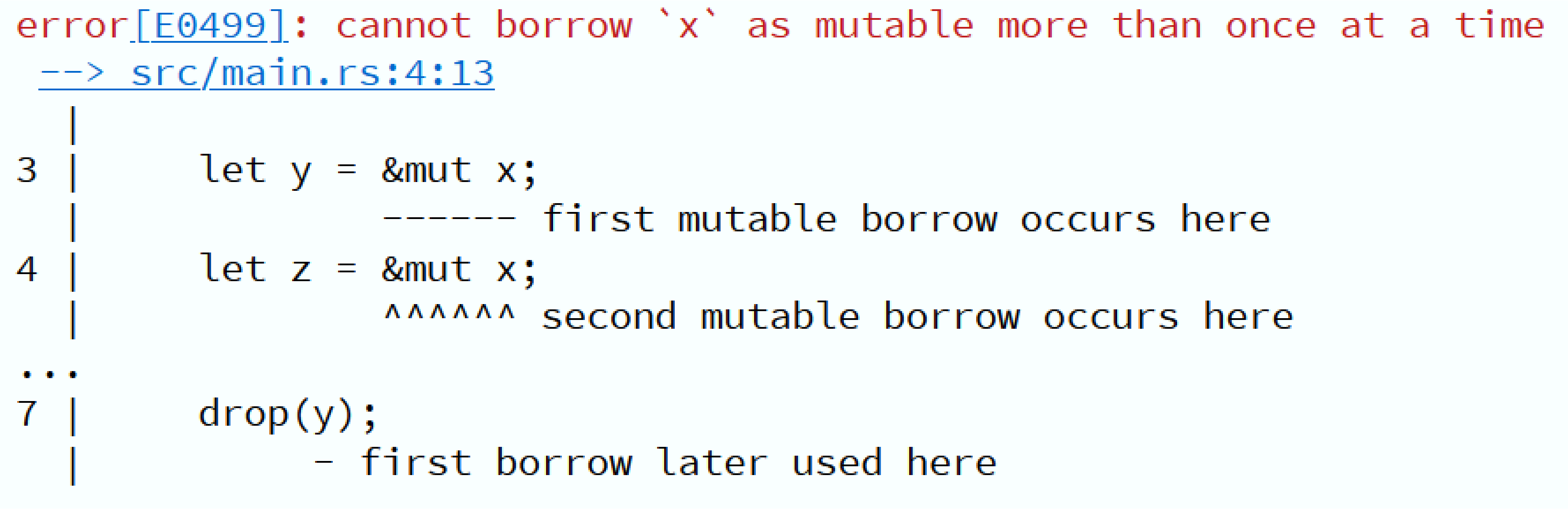
```
let x = Vec::<u8>::new();
let y = x;
drop(x);
```



```
error[E0382]: use of moved value: `x`
 --> src/main.rs:4:10
  |
2 |     let x = Vec::<u8>::new();
  |         - move occurs because `x` has type `Vec<u8>`, which does not implement the `Copy` trait
3 |     let y = x;
  |             - value moved here
4 |     drop(x);
  |          ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
```

Good compiler feedback

```
let mut x = Vec::<u8>::new();
let y = &mut x;
let z = &mut x;
drop(y);
```



```
error[E0499]: cannot borrow `x` as mutable more than once at a time
 --> src/main.rs:4:13
  |
3 |     let y = &mut x;
  |             ------ first mutable borrow occurs here
4 |     let z = &mut x;
  |             ^^^^^^ second mutable borrow occurs here
...
7 |     drop(y);
  |          - first borrow later used here
```

Traits & generics

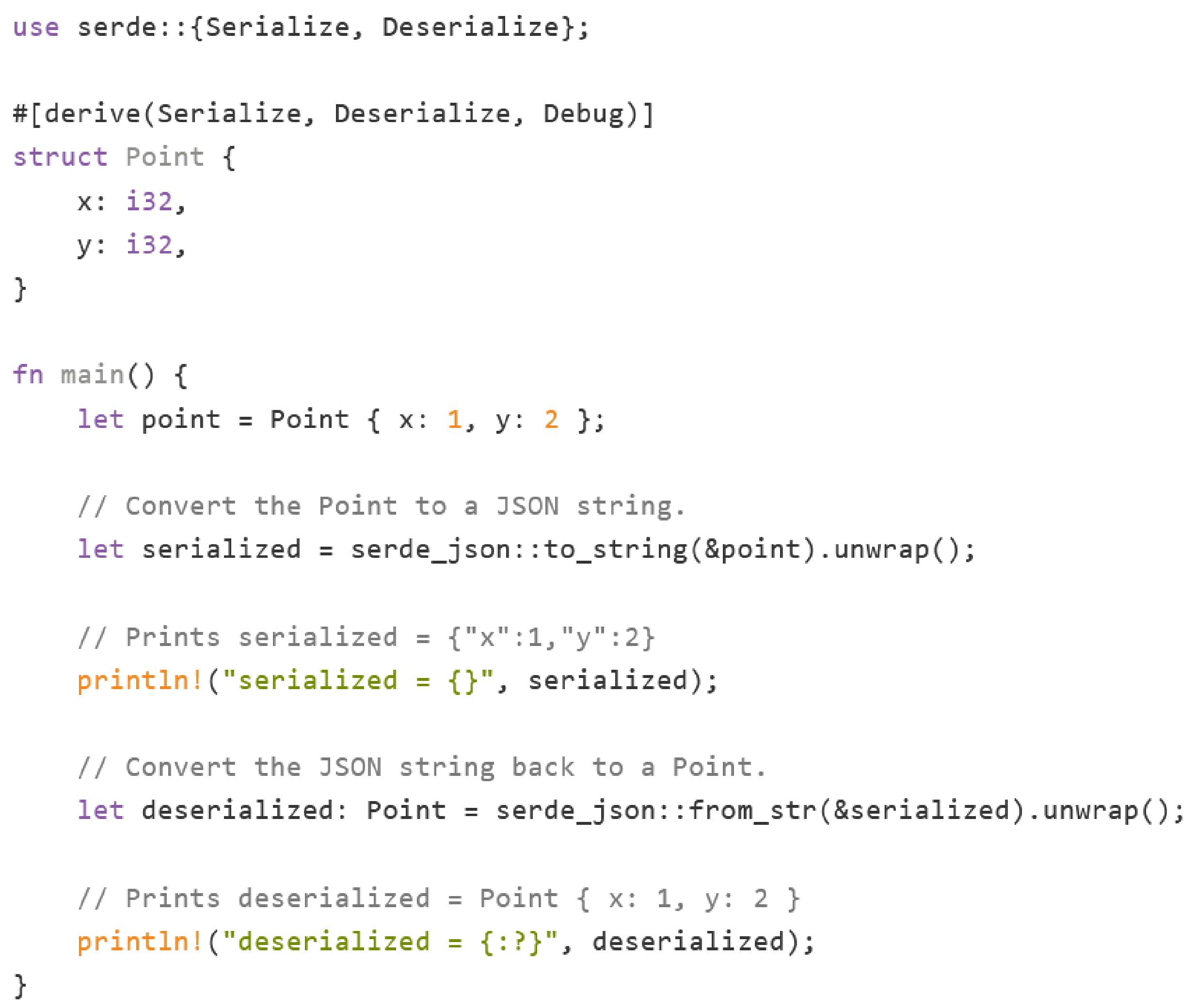
- Structs implement traits
- Traits are like interfaces in Java
- Generic bounds using traits (not unlike C++ concepts)
- Monomorphization (not unlike C++ templates)

- No classic OOP, so traits are the main abstraction mechanic

Traits & generics

Use #[derive()] to automatically implement traits

Serde is really cool btw



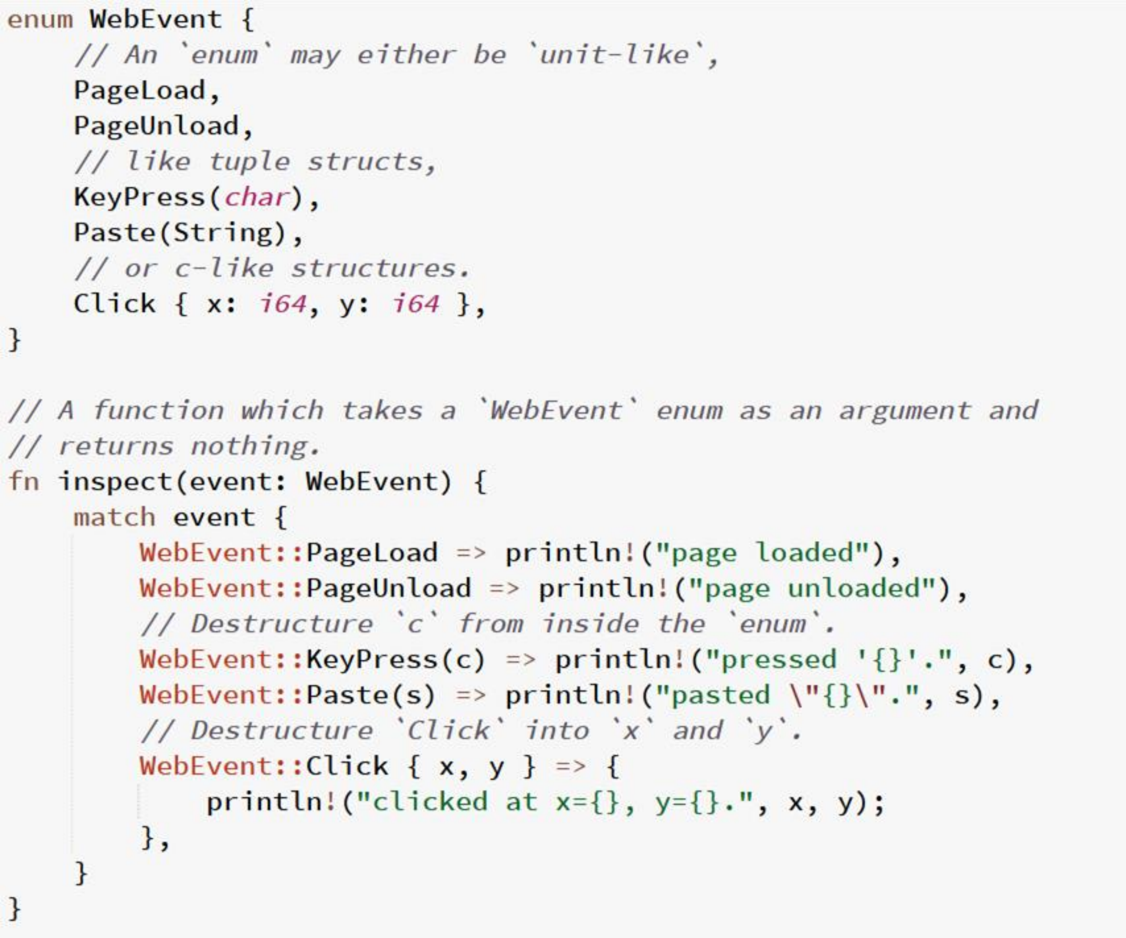
```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Enums

Enum variants can contain data.

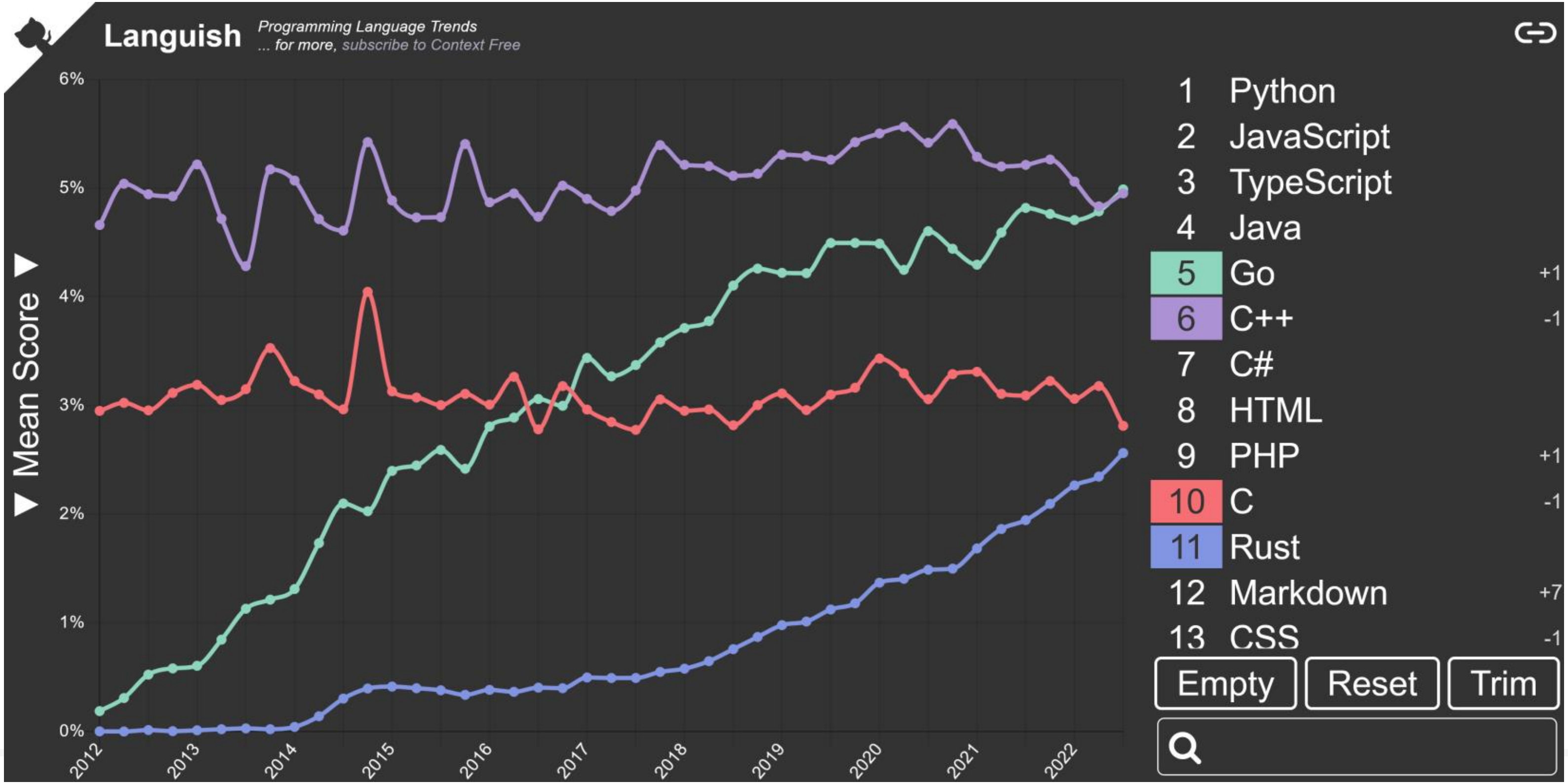
Enums can implement functions & traits.

Pattern matching on enums and much more.



```rust
enum WebEvent {
    // An `enum` may either be `unit-like`,
    PageLoad,
    PageUnload,
    // like tuple structs,
    KeyPress(char),
    Paste(String),
    // or c-like structures.
    Click { x: i64, y: i64 },
}

// A function which takes a `WebEvent` enum as an argument and
// returns nothing.
fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad => println!("page loaded"),
        WebEvent::PageUnload => println!("page unloaded"),
        // Destructure `c` from inside the `enum`.
        WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
        WebEvent::Paste(s) => println!("pasted \"{}\".", s),
        // Destructure `Click` into `x` and `y`.
        WebEvent::Click { x, y } => {
            println!("clicked at x={}, y={}.", x, y);
        },
    }
}
```

Increasingly popular

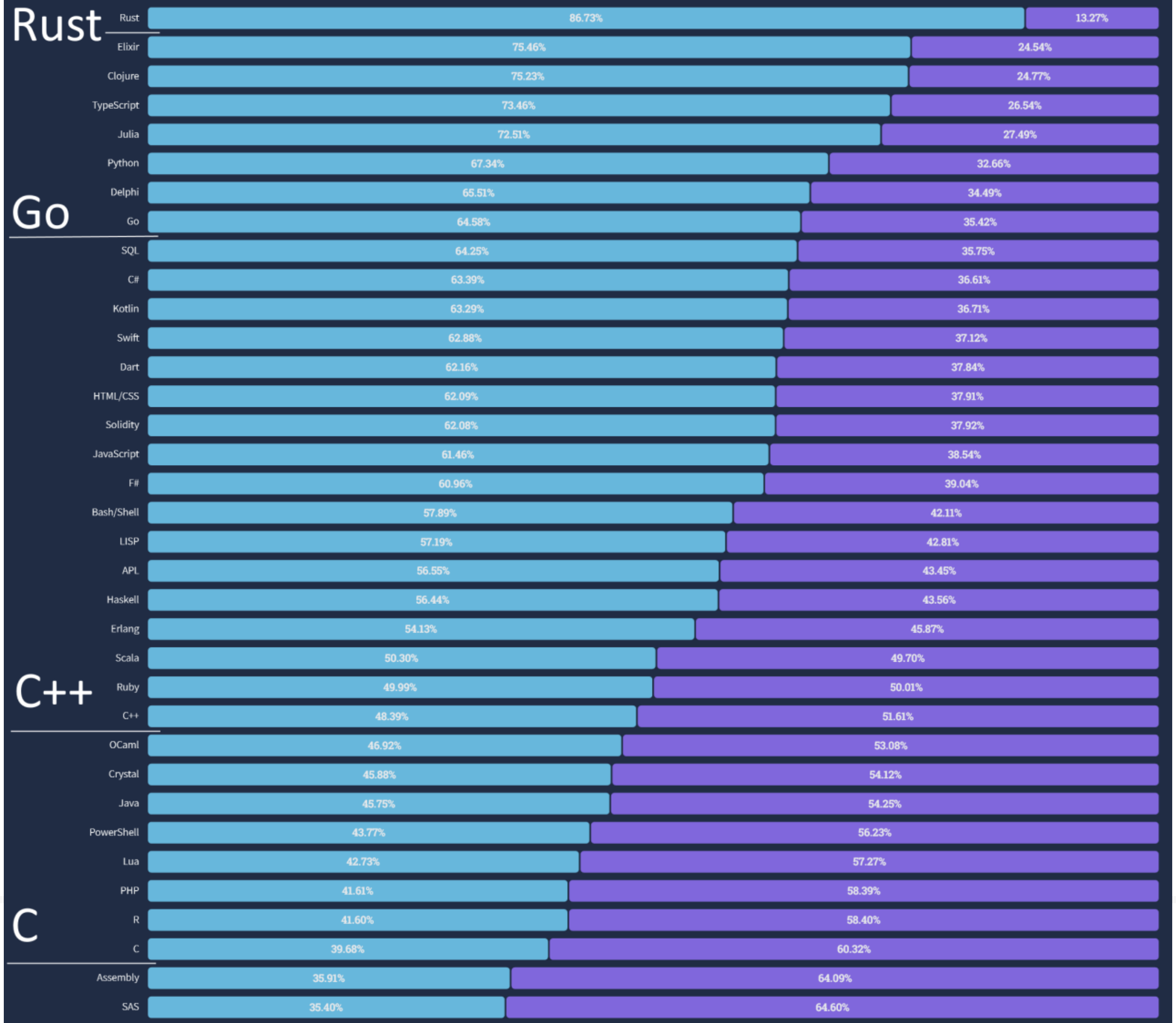


https://tjpalmer.github.io/languish/

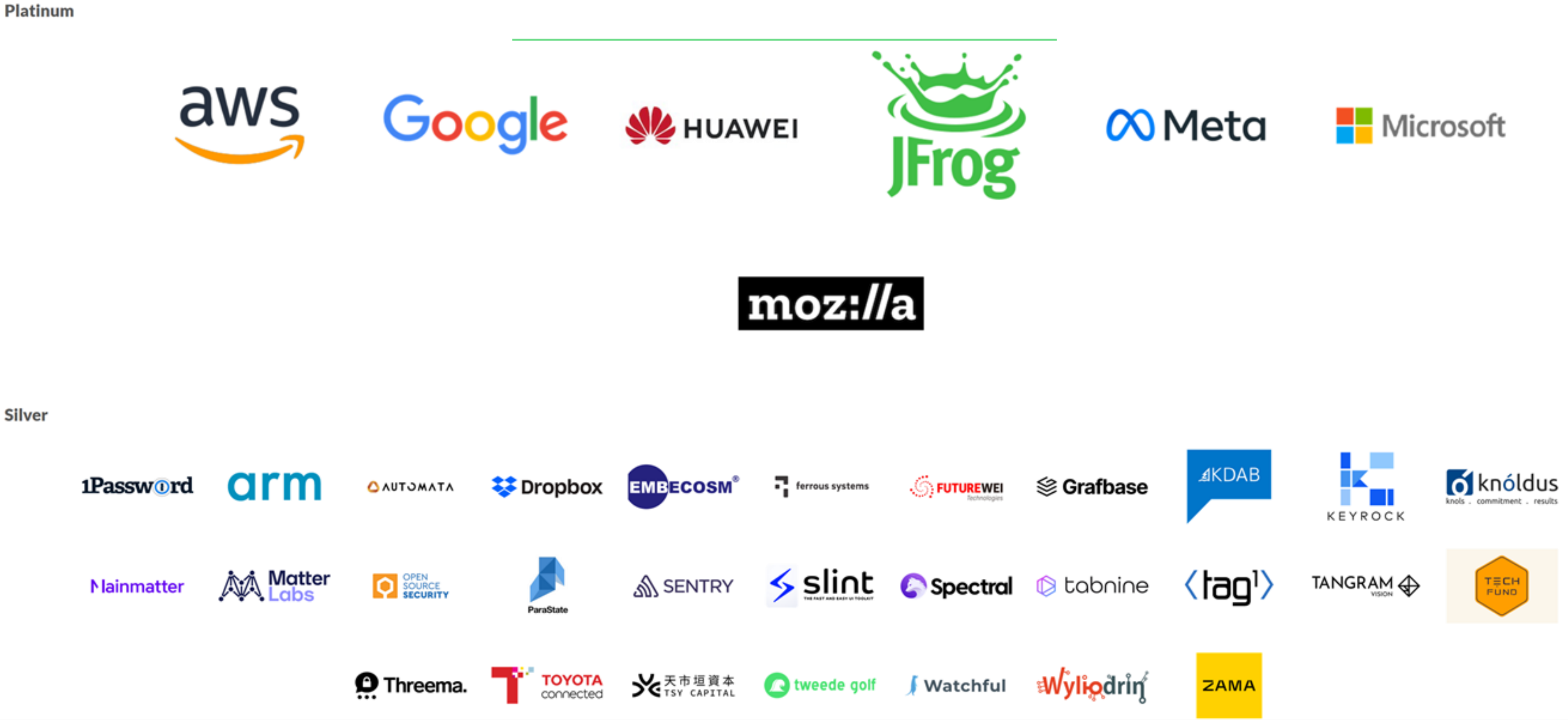
Very well liked

7 years in a row, Rust is the most loved language.

https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted



Lots of big players are investing



Platinum

aws

Google

HUAWEI

JFrog

Meta

Microsoft

moz://a

Silver

1Password

arm

AUTOMATA

Dropbox

EMBECOSM

ferrous systems

FUTUREWEI Technologies

Grafbase

KDAB

KEYROCK

knóldus knols . commitment . results

Mainmatter

Matter Labs

OPEN SOURCE SECURITY

ParaState

SENTRY

slint THE FAST AND EASY UI TOOLKIT

Spectral

tabnine

tag1

TANGRAM VISION

TECH FUND

Threema.

TOYOTA connected

天市垣資本 TSY CAPITAL

tweede golf

Watchful

Wyliodrin

ZAMA

https://foundation.rust-lang.org/members/

Embedded

Rust embedded ecosystem

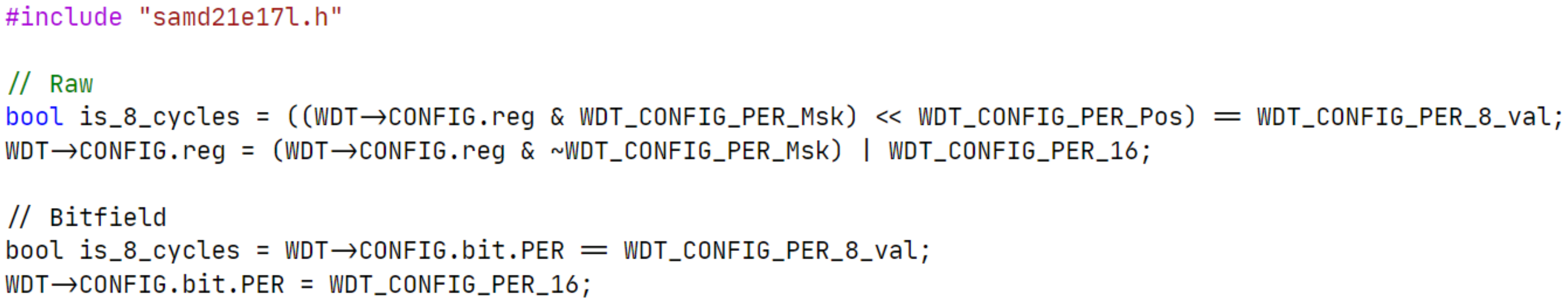
- core, alloc, std
- cargo, crates, rustc
- probe-rs & more

arm

RISC-V®

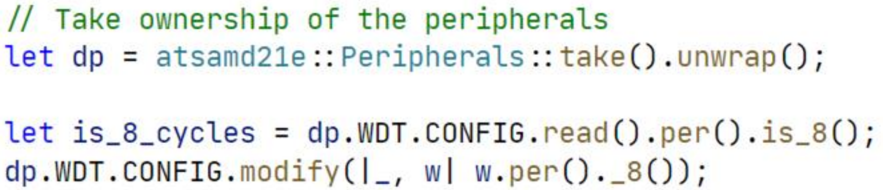
Peripheral Access Crates

C



```c
#include "samd21e17l.h"

// Raw
bool is_8_cycles = ((WDT->CONFIG.reg & WDT_CONFIG_PER_Msk) << WDT_CONFIG_PER_Pos) == WDT_CONFIG_PER_8_val;
WDT->CONFIG.reg = (WDT->CONFIG.reg & ~WDT_CONFIG_PER_Msk) | WDT_CONFIG_PER_16;

// Bitfield
bool is_8_cycles = WDT->CONFIG.bit.PER == WDT_CONFIG_PER_8_val;
WDT->CONFIG.bit.PER = WDT_CONFIG_PER_16;
```

Rust



```rust
// Take ownership of the peripherals
let dp = atsamd21e::Peripherals::take().unwrap();

let is_8_cycles = dp.WDT.CONFIG.read().per().is_8();
dp.WDT.CONFIG.modify(|_, w| w.per()._8());
```

Overview

PAC SAMD21E

PAC nRF9160

PAC nRF52840

PAC STM32H743

PAC STM32H753

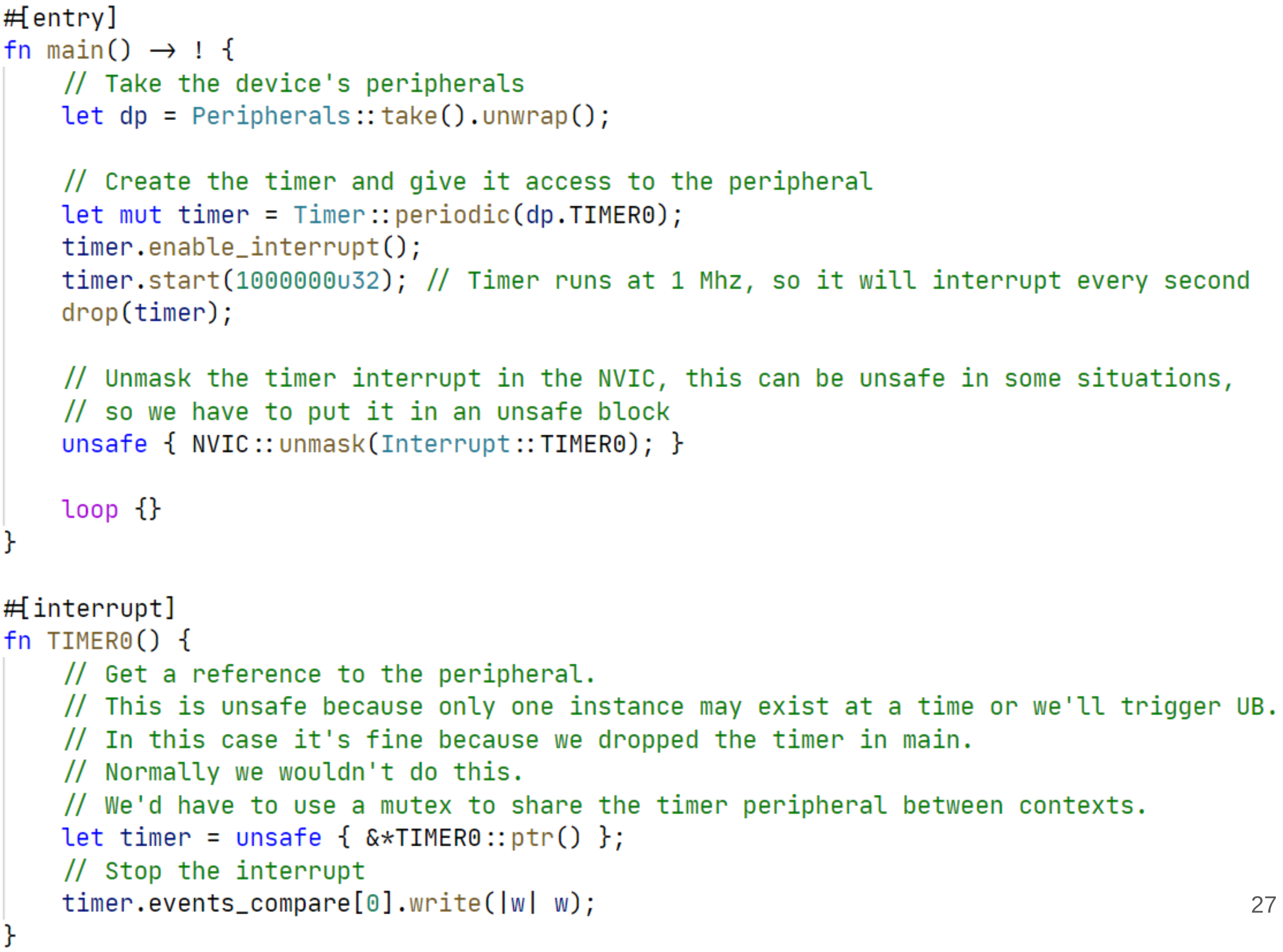
PAC STM32L476

PAC STM32L496

Device HALs

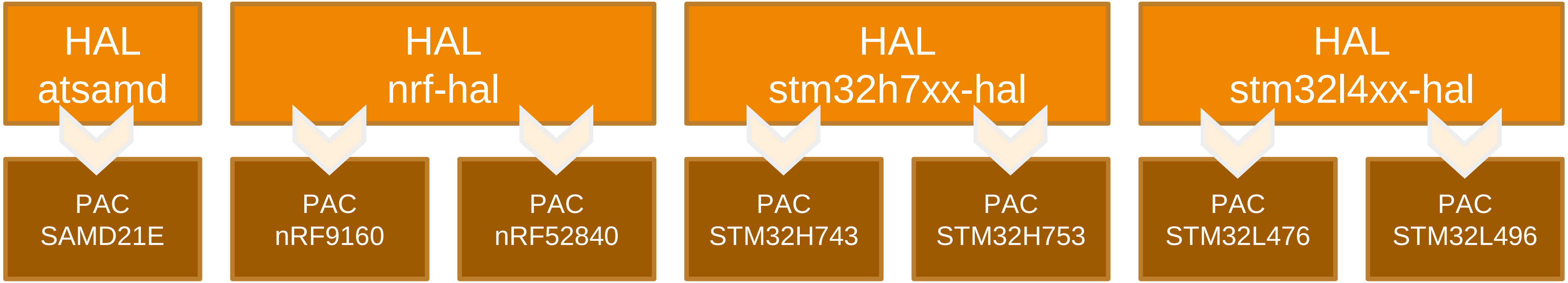
- Many open source HALs
  - Vendor HAL for RiscV ESP chips
  - Also async support (embassy)
- Implementation of high level chip features
- Built on top of PACs

Device HALs



```rust
#[entry]
fn main() -> ! {
    // Take the device's peripherals
    let dp = Peripherals::take().unwrap();

    // Create the timer and give it access to the peripheral
    let mut timer = Timer::periodic(dp.TIMER0);
    timer.enable_interrupt();
    timer.start(1000000u32); // Timer runs at 1 Mhz, so it will interrupt every second
    drop(timer);

    // Unmask the timer interrupt in the NVIC, this can be unsafe in some situations,
    // so we have to put it in an unsafe block
    unsafe { NVIC::unmask(Interrupt::TIMER0); }

    loop {}
}

#[interrupt]
fn TIMER0() {
    // Get a reference to the peripheral.
    // This is unsafe because only one instance may exist at a time or we'll trigger UB.
    // In this case it's fine because we dropped the timer in main.
    // Normally we wouldn't do this.
    // We'd have to use a mutex to share the timer peripheral between contexts.
    let timer = unsafe { &*TIMER0::ptr() };
    // Stop the interrupt
    timer.events_compare[0].write(|w| w);
}
```

Overview



HAL
atsamd

HAL
nrf-hal

HAL
stm32h7xx-hal

HAL
stm32l4xx-hal

PAC
SAMD21E

PAC
nRF9160

PAC
nRF52840

PAC
STM32H743

PAC
STM32H753

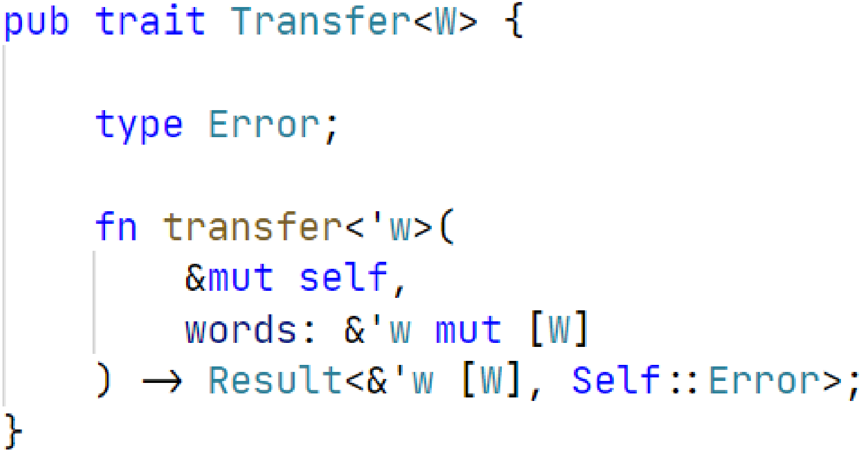
PAC
STM32L476

PAC
STM32L496

Cooperation

`Embedded-hal` is the glue of the entire ecosystem

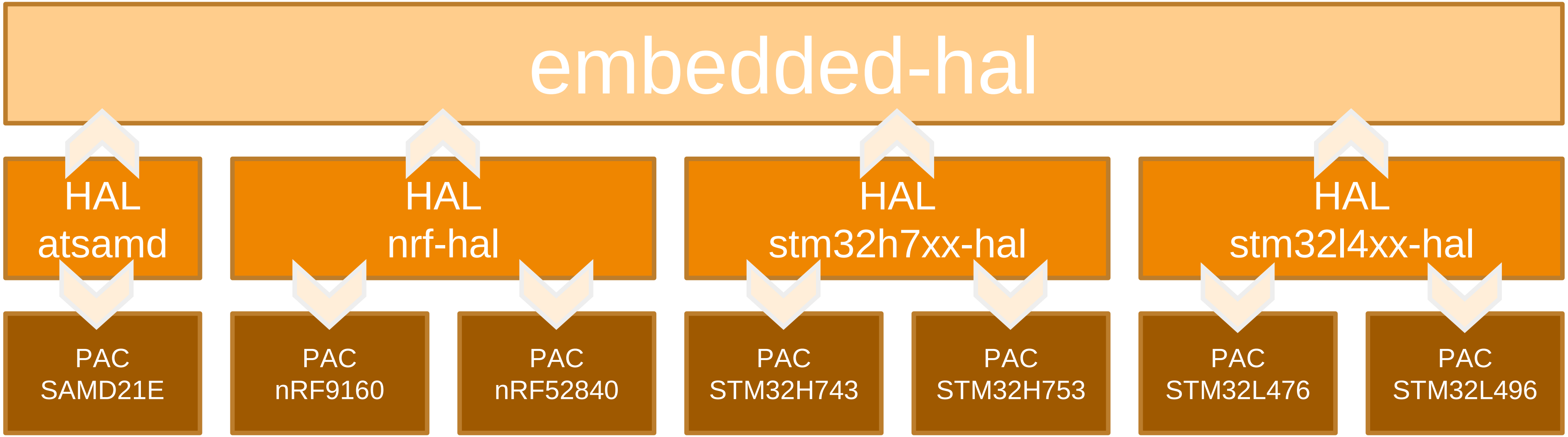
- Contains abstractions for many common operations
- SPI example trait:



```
pub trait Transfer<W> {

    type Error;

    fn transfer<'w>(
        &mut self,
        words: &'w mut [W]
    ) -> Result<&'w [W], Self::Error>;
}
```

FYI: SPI (Serial Peripheral Interface) is a common communication protocol to talk with other devices

Overview


embedded-hal
HAL
atsamd
HAL
nrf-hal
HAL
stm32h7xx-hal
HAL
stm32l4xx-hal
PAC
SAMD21E
PAC
nRF9160
PAC
nRF52840
PAC
STM32H743
PAC
STM32H753
PAC
STM32L476
PAC
STM32L496

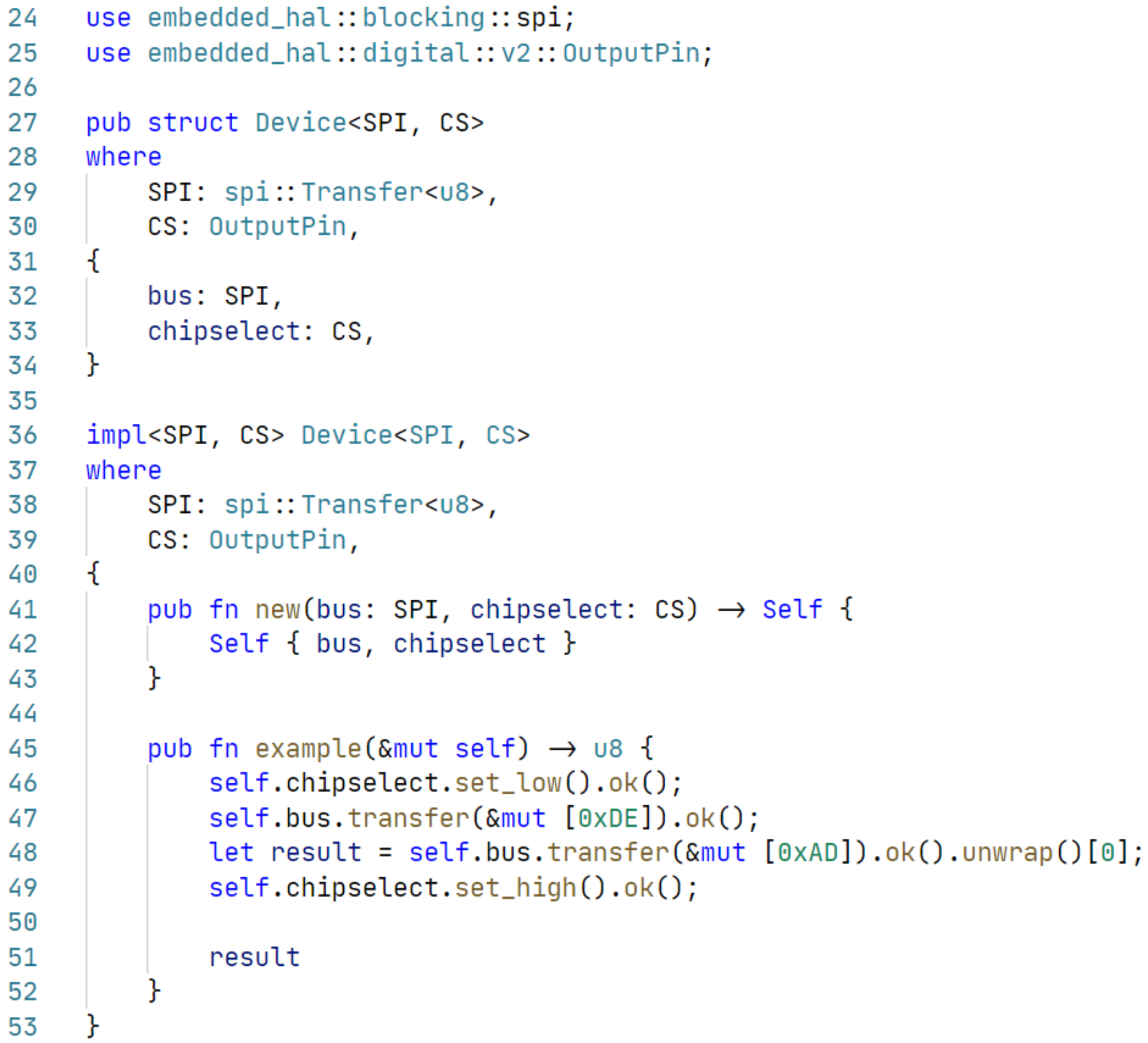
Device drivers

Traits + generics

- Reuse traits from embedded-hal
- Efficient
- Convenient

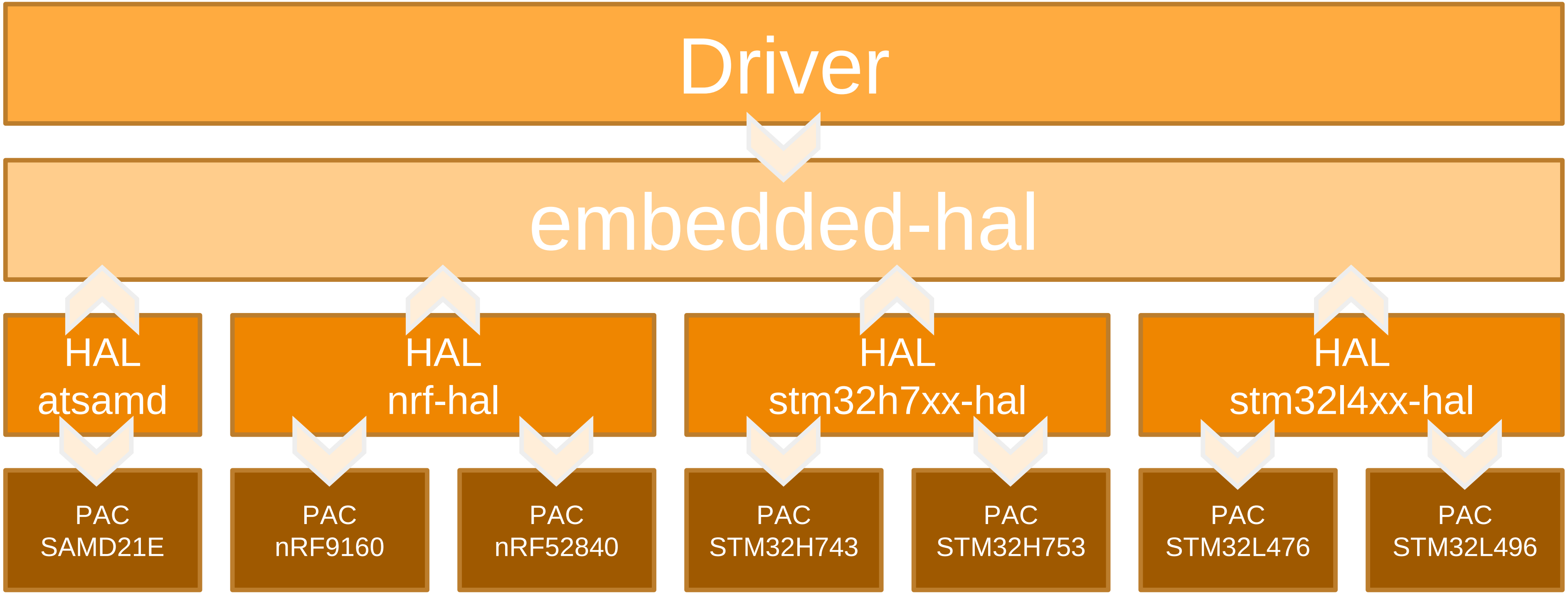
Frustrating in C

- No standards
- No abstractions
  - Function pointers?
  - Extern functions?
  - Fork & implement in library?



```rust
use embedded_hal::blocking::spi;
use embedded_hal::digital::v2::OutputPin;

pub struct Device<SPI, CS>
where
    SPI: spi::Transfer<u8>,
    CS: OutputPin,
{
    bus: SPI,
    chipselect: CS,
}

impl<SPI, CS> Device<SPI, CS>
where
    SPI: spi::Transfer<u8>,
    CS: OutputPin,
{
    pub fn new(bus: SPI, chipselect: CS) -> Self {
        Self { bus, chipselect }
    }

    pub fn example(&mut self) -> u8 {
        self.chipselect.set_low().ok();
        self.bus.transfer(&mut [0xDE]).ok();
        let result = self.bus.transfer(&mut [0xAD]).ok().unwrap()[0];
        self.chipselect.set_high().ok();

        result
    }
}
```

Overview



Driver

embedded-hal

HAL
atsamd

HAL
nrf-hal

HAL
stm32h7xx-hal

HAL
stm32l4xx-hal

PAC
SAMD21E

PAC
nRF9160

PAC
nRF52840

PAC
STM32H743

PAC
STM32H753

PAC
STM32L476

PAC
STM32L496

Runtimes

● Bare metal + interrupts

● RTIC

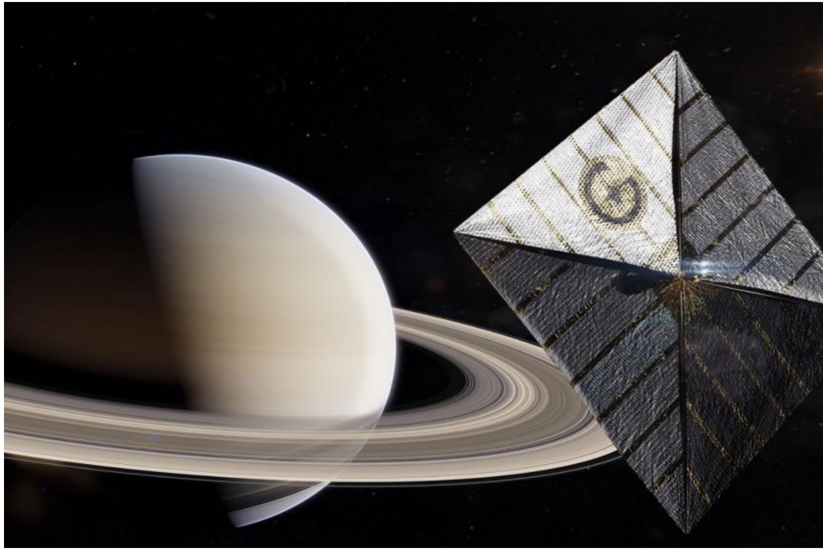
● RTOS

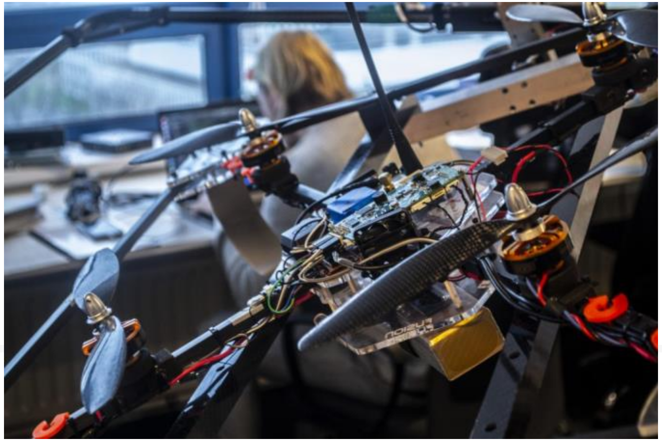
● Async

(blog posts)

So why embedded Rust?

- Program low level devices in a high level language
  - More productive
  - Fewer bugs
  - We can still do the same cool things

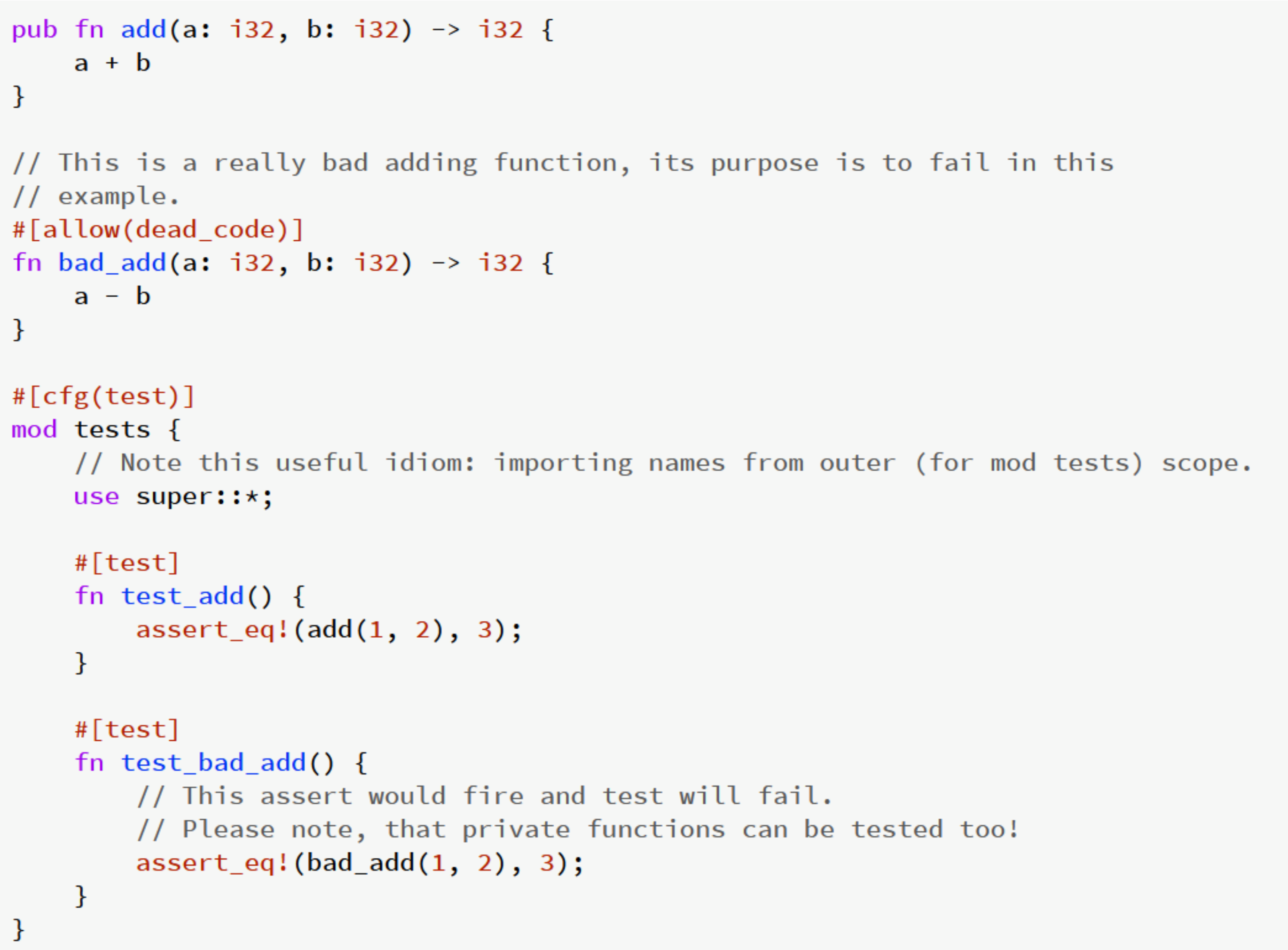




Extra tools

Tools

Built-in unit testing

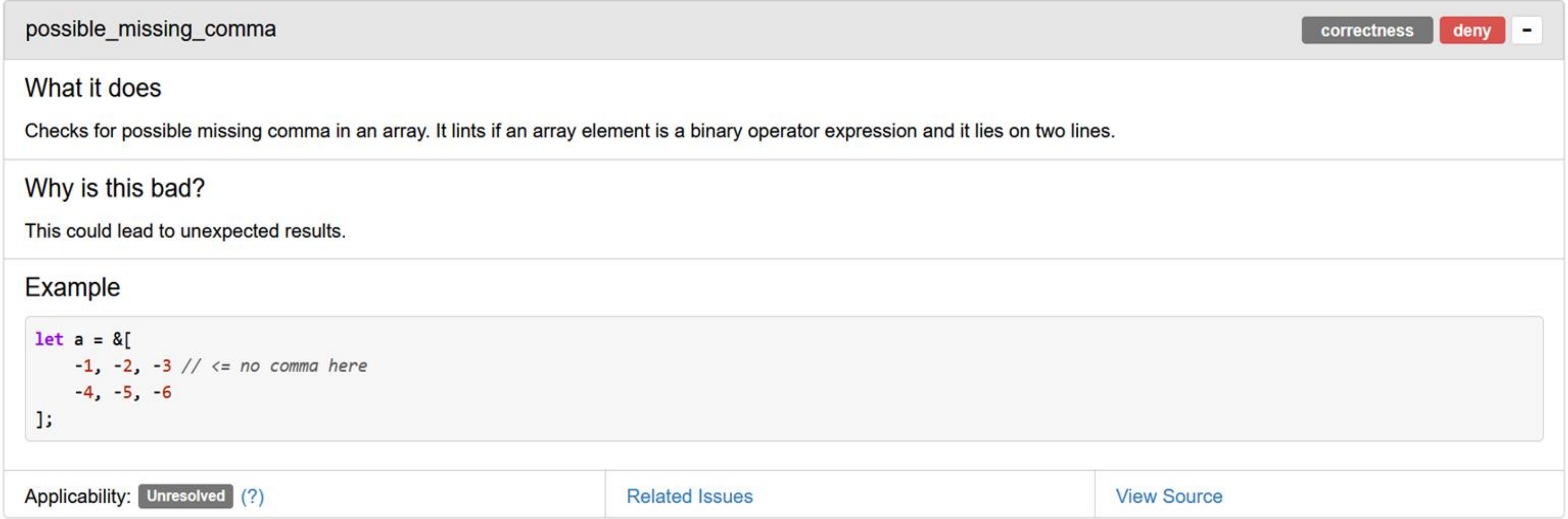


```rust
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

// This is a really bad adding function, its purpose is to fail in this
// example.
#[allow(dead_code)]
fn bad_add(a: i32, b: i32) -> i32 {
    a - b
}

#[cfg(test)]
mod tests {
    // Note this useful idiom: importing names from outer (for mod tests) scope.
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }

    #[test]
    fn test_bad_add() {
        // This assert would fire and test will fail.
        // Please note, that private functions can be tested too!
        assert_eq!(bad_add(1, 2), 3);
    }
}
```

Tools

Clippy:

- Prevent common mistakes
- Small efficiency improvements



possible_missing_comma

What it does

Checks for possible missing comma in an array. It lints if an array element is a binary operator expression and it lies on two lines.

Why is this bad?

This could lead to unexpected results.

Example

```
let a = &[
    -1, -2, -3 // <= no comma here
    -4, -5, -6
];
```

Applicability: Unresolved (?)

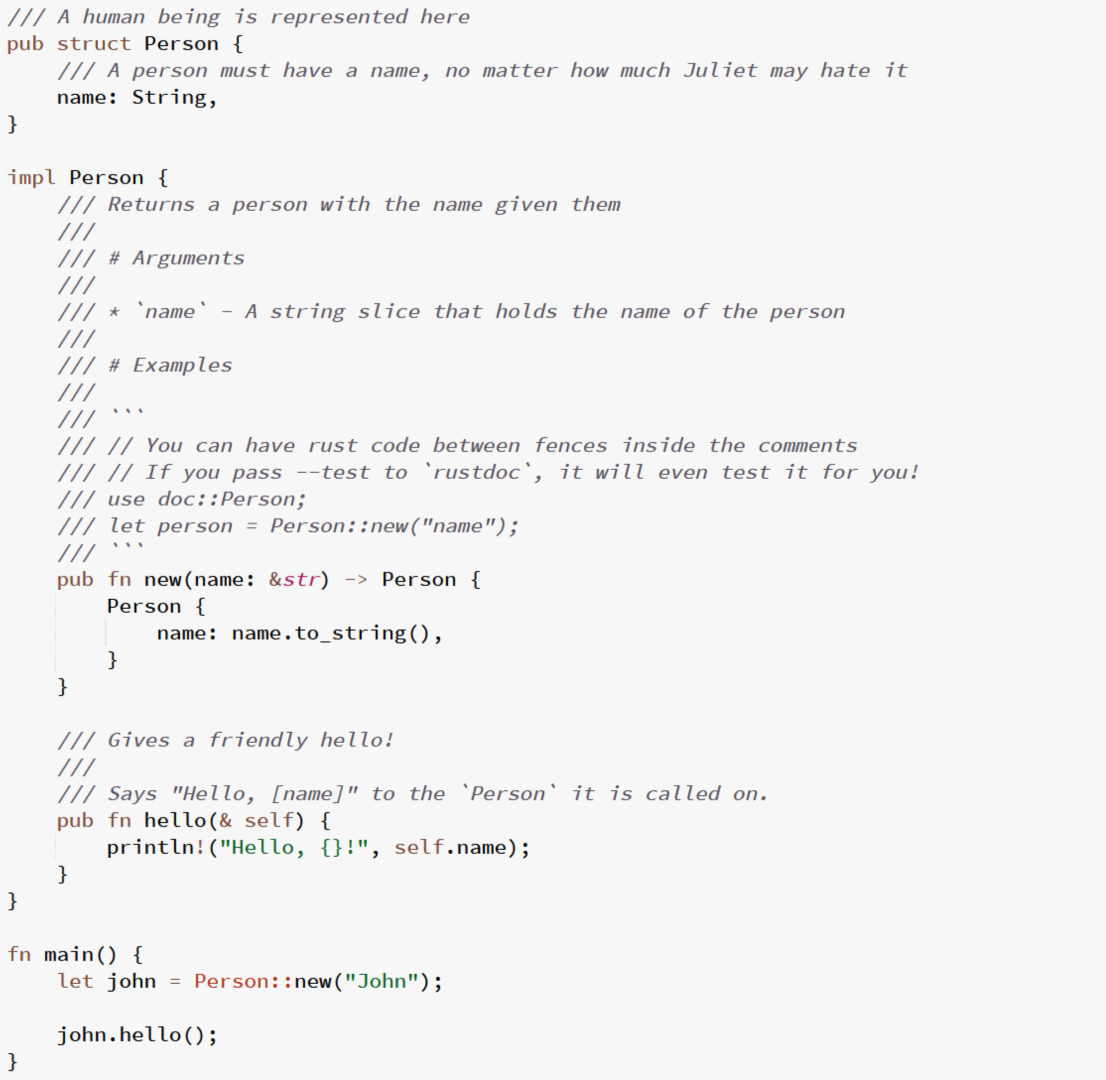
Tools

Docs:

Markdown

Generated to html (like doxygen)

docs.rs



```rust
/// A human being is represented here
pub struct Person {
    /// A person must have a name, no matter how much Juliet may hate it
    name: String,
}

impl Person {
    /// Returns a person with the name given them
    ///
    /// # Arguments
    ///
    /// * `name` - A string slice that holds the name of the person
    ///
    /// # Examples
    ///
    /// ```
    /// // You can have rust code between fences inside the comments
    /// // If you pass --test to `rustdoc`, it will even test it for you!
    /// use doc::Person;
    /// let person = Person::new("name");
    /// ```
    pub fn new(name: &str) -> Person {
        Person {
            name: name.to_string(),
        }
    }

    /// Gives a friendly hello!
    ///
    /// Says "Hello, [name]" to the `Person` it is called on.
    pub fn hello(& self) {
        println!("Hello, {}!", self.name);
    }
}

fn main() {
    let john = Person::new("John");

    john.hello();
}
```

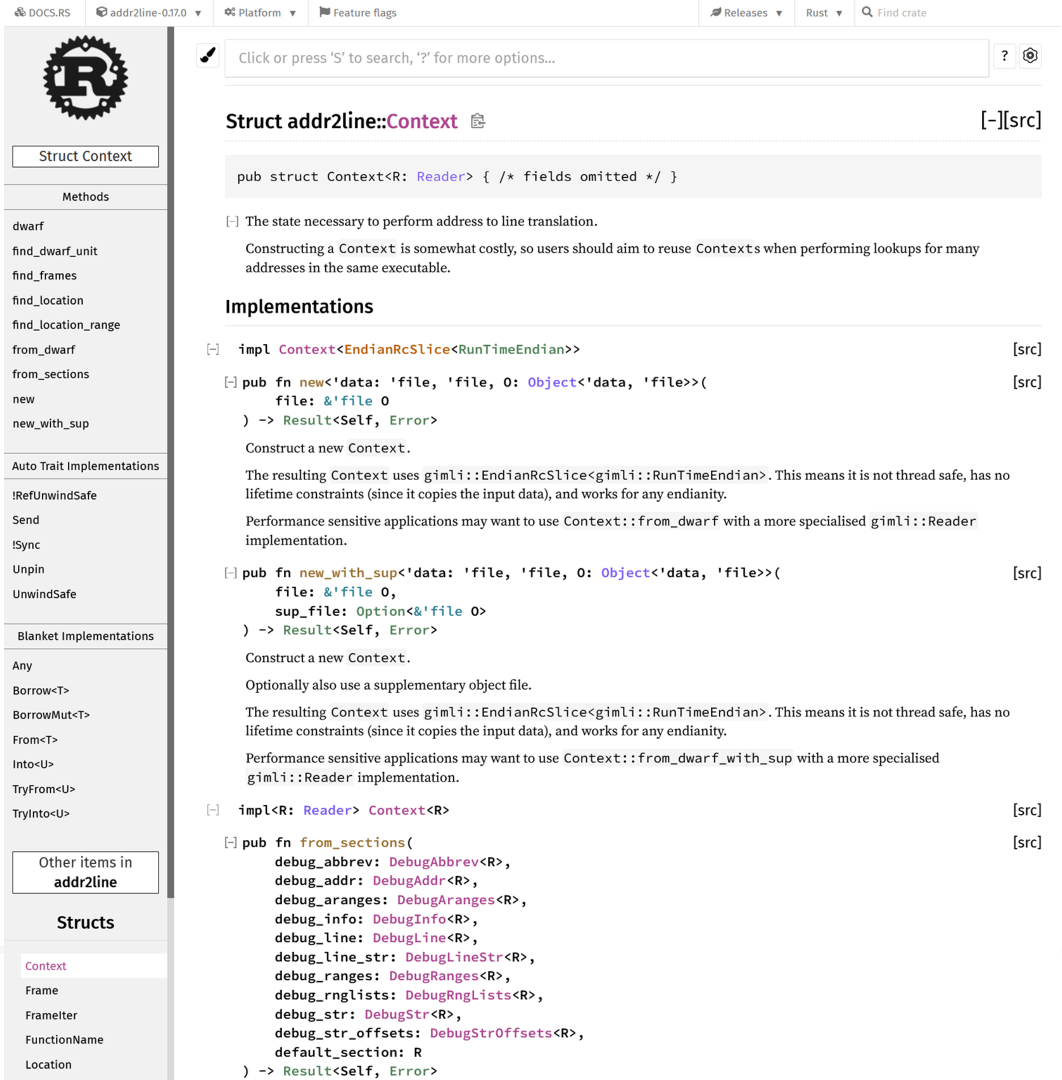
Tools

Docs:

Markdown

Generated to html (like doxygen)

docs.rs



Struct addr2line::Context

```
pub struct Context<R: Reader> { /* fields omitted */ }
```

[-] The state necessary to perform address to line translation.

Constructing a `Context` is somewhat costly, so users should aim to reuse `Context`s when performing lookups for many addresses in the same executable.

Implementations

```
[-] impl Context<EndianRcSlice<RunTimeEndian>>                                [src]
```

```
[-] pub fn new<'data: 'file, 'file, O: Object<'data, 'file>>(                   [src]
        file: &'file O
    ) -> Result<Self, Error>
```

Construct a new `Context`.

The resulting `Context` uses `gimli::EndianRcSlice<gimli::RunTimeEndian>`. This means it is not thread safe, has no lifetime constraints (since it copies the input data), and works for any endianity.

Performance sensitive applications may want to use `Context::from_dwarf` with a more specialised `gimli::Reader` implementation.

```
[-] pub fn new_with_sup<'data: 'file, 'file, O: Object<'data, 'file>>(          [src]
        file: &'file O,
        sup_file: Option<&'file O>
    ) -> Result<Self, Error>
```

Construct a new `Context`.

Optionally also use a supplementary object file.

The resulting `Context` uses `gimli::EndianRcSlice<gimli::RunTimeEndian>`. This means it is not thread safe, has no lifetime constraints (since it copies the input data), and works for any endianity.

Performance sensitive applications may want to use `Context::from_dwarf_with_sup` with a more specialised `gimli::Reader` implementation.

```
[-] impl<R: Reader> Context<R>                                                  [src]
```

```
[-] pub fn from_sections(                                                        [src]
        debug_abbrev: DebugAbbrev<R>,
        debug_addr: DebugAddr<R>,
        debug_aranges: DebugAranges<R>,
        debug_info: DebugInfo<R>,
        debug_line: DebugLine<R>,
        debug_line_str: DebugLineStr<R>,
        debug_ranges: DebugRanges<R>,
        debug_rnglists: DebugRngLists<R>,
        debug_str: DebugStr<R>,
        debug_str_offsets: DebugStrOffsets<R>,
        default_section: R
    ) -> Result<Self, Error>
```

Many more tools

- Cargo: Package manager & build system
- rustfmt: Code formatting
- Knurling-rs (embedded)
  - Probe-run
  - Defmt
- Criterion: Microbenchmarking
- Bindings
  - rust-bindgen
  - cxx
- Any text editor using LSP (for Rust Analyzer plugin)
- Any IntelliJ IDE (for IntelliJ Rust plugin)

FFI & C

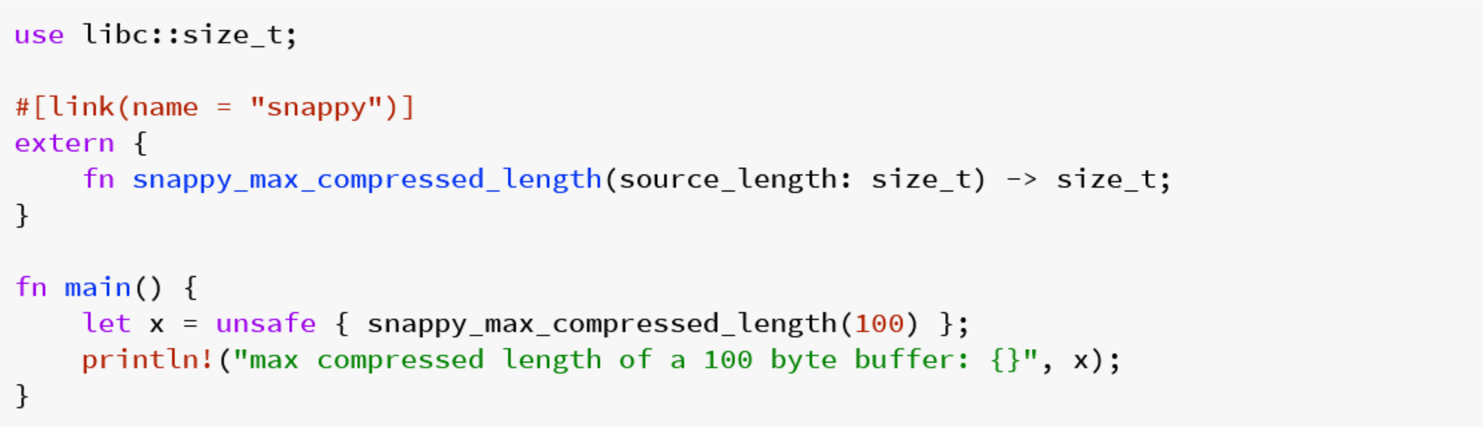
Why FFI

We cannot rewrite everything in Rust.

Sometimes we want to use a C library.

FFI

We can call C function. We need to define it and link with the C binary.

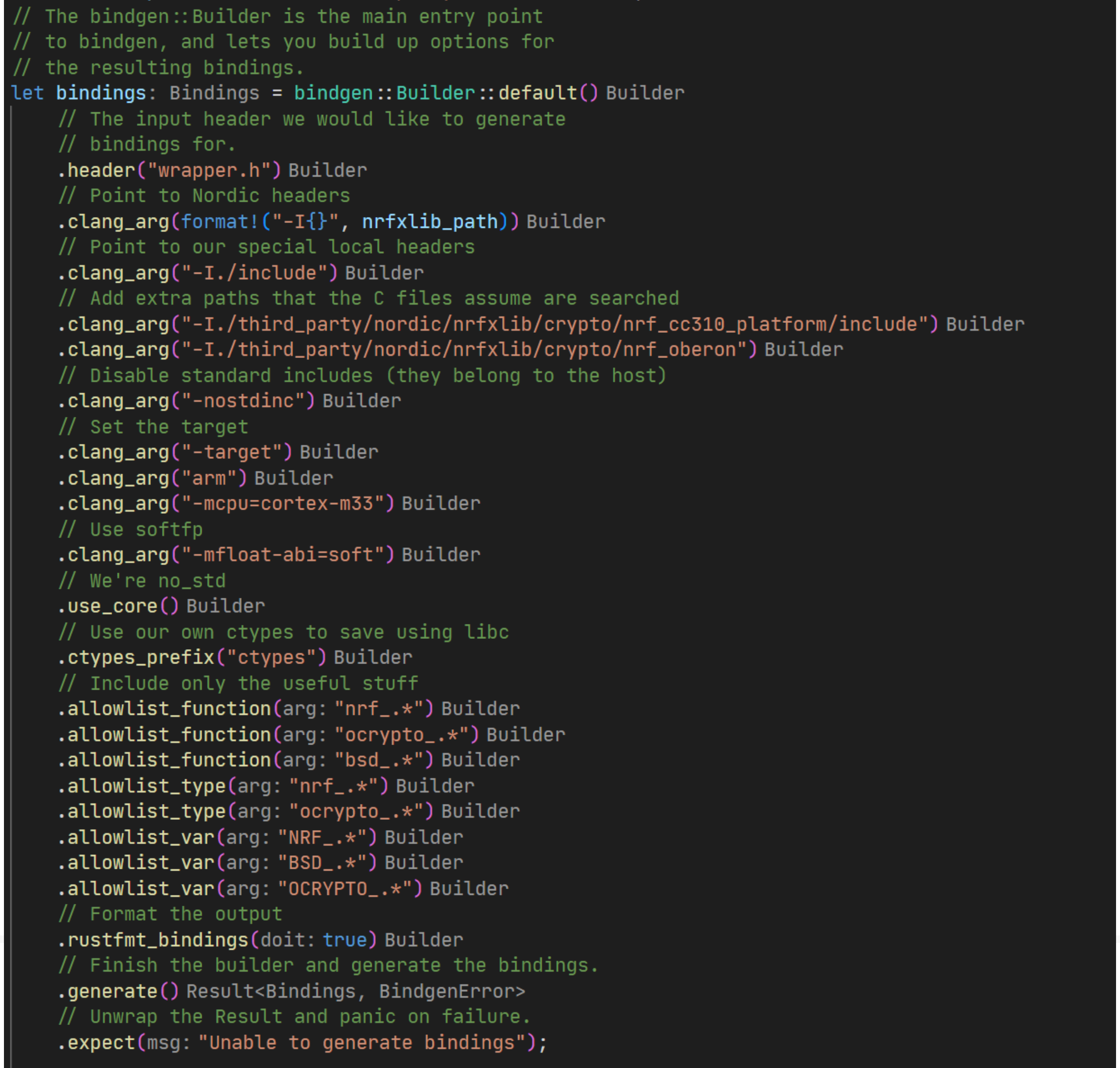


```rust
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

https://doc.rust-lang.org/nomicon/ffi.html

Let’s automate

We can generate the functions using bindgen

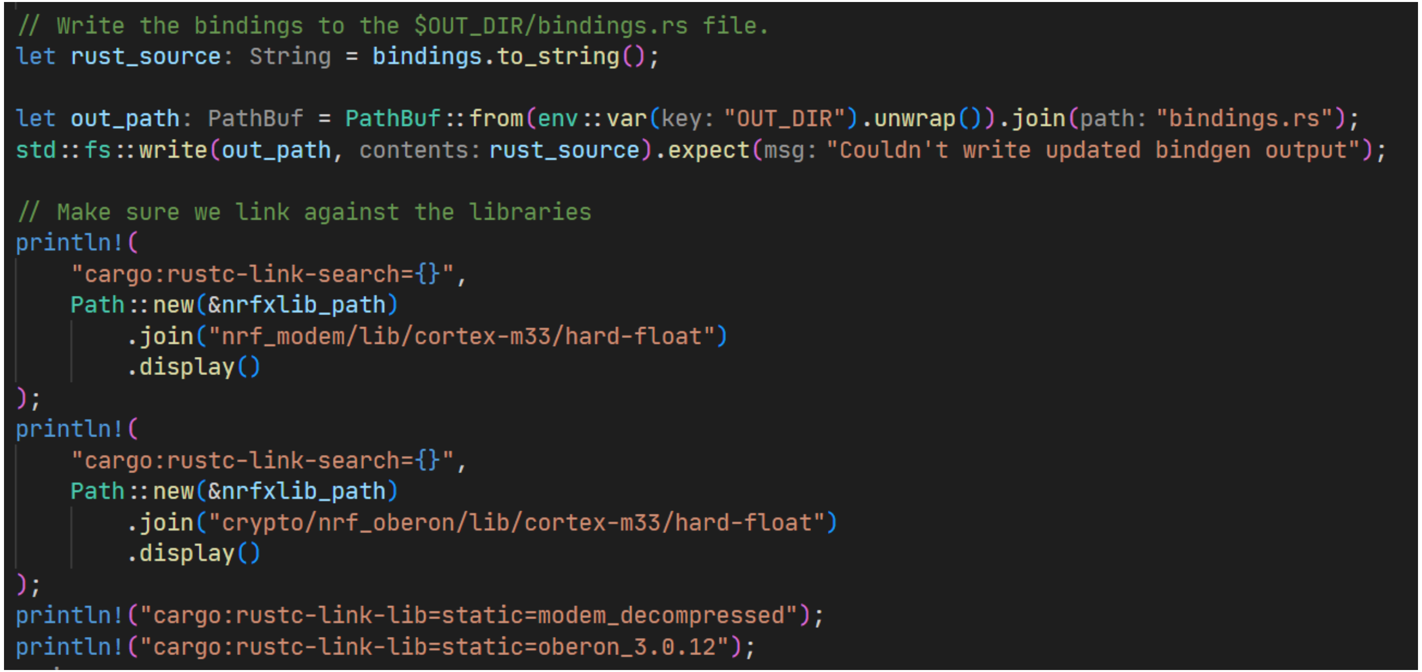


```
// The bindgen::Builder is the main entry point
// to bindgen, and lets you build up options for
// the resulting bindings.
let bindings: Bindings = bindgen::Builder::default() Builder
    // The input header we would like to generate
    // bindings for.
    .header("wrapper.h") Builder
    // Point to Nordic headers
    .clang_arg(format!("-I{}", nrfxlib_path)) Builder
    // Point to our special local headers
    .clang_arg("-I./include") Builder
    // Add extra paths that the C files assume are searched
    .clang_arg("-I./third_party/nordic/nrfxlib/crypto/nrf_cc310_platform/include") Builder
    .clang_arg("-I./third_party/nordic/nrfxlib/crypto/nrf_oberon") Builder
    // Disable standard includes (they belong to the host)
    .clang_arg("-nostdinc") Builder
    // Set the target
    .clang_arg("-target") Builder
    .clang_arg("arm") Builder
    .clang_arg("-mcpu=cortex-m33") Builder
    // Use softfp
    .clang_arg("-mfloat-abi=soft") Builder
    // We're no_std
    .use_core() Builder
    // Use our own ctypes to save using libc
    .ctypes_prefix("ctypes") Builder
    // Include only the useful stuff
    .allowlist_function(arg: "nrf_.*") Builder
    .allowlist_function(arg: "ocrypto_.*") Builder
    .allowlist_function(arg: "bsd_.*") Builder
    .allowlist_type(arg: "nrf_.*") Builder
    .allowlist_type(arg: "ocrypto_.*") Builder
    .allowlist_var(arg: "NRF_.*") Builder
    .allowlist_var(arg: "BSD_.*") Builder
    .allowlist_var(arg: "OCRYPTO_.*") Builder
    // Format the output
    .rustfmt_bindings(doit: true) Builder
    // Finish the builder and generate the bindings.
    .generate() Result<Bindings, BindgenError>
    // Unwrap the Result and panic on failure.
    .expect(msg: "Unable to generate bindings");
```

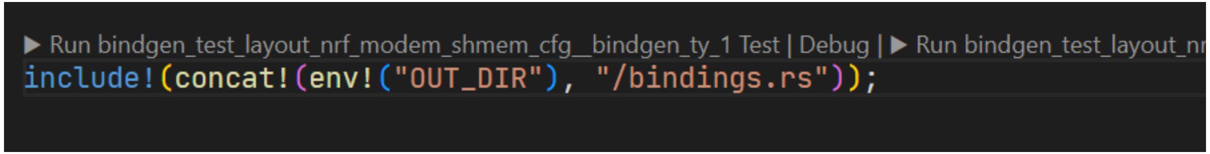
tweede golf

Let’s automate

Output the file and let the compiler link with the C binary

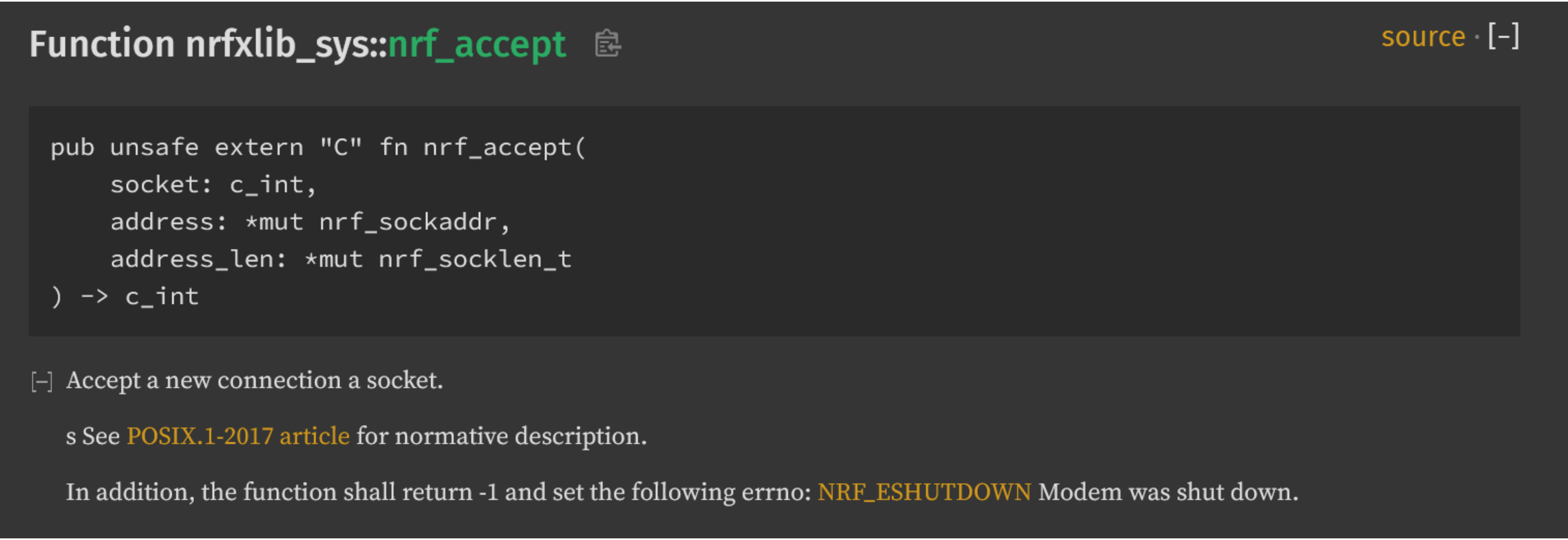


```
// Write the bindings to the $OUT_DIR/bindings.rs file.
let rust_source: String = bindings.to_string();

let out_path: PathBuf = PathBuf::from(env::var(key: "OUT_DIR").unwrap()).join(path: "bindings.rs");
std::fs::write(out_path, contents: rust_source).expect(msg: "Couldn't write updated bindgen output");

// Make sure we link against the libraries
println!(
    "cargo:rustc-link-search={}",
    Path::new(&nrfxlib_path)
        .join("nrf_modem/lib/cortex-m33/hard-float")
        .display()
);
println!(
    "cargo:rustc-link-search={}",
    Path::new(&nrfxlib_path)
        .join("crypto/nrf_oberon/lib/cortex-m33/hard-float")
        .display()
);
println!("cargo:rustc-link-lib=static=modem_decompressed");
println!("cargo:rustc-link-lib=static=oberon_3.0.12");
```



```
▶ Run bindgen_test_layout_nrf_modem_shmem_cfg__bindgen_ty_1 Test | Debug | ▶ Run bindgen_test_layout_nr
include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```

Now we can use it



Function nrfxlib_sys::nrf_accept

```
pub unsafe extern "C" fn nrf_accept(
    socket: c_int,
    address: *mut nrf_sockaddr,
    address_len: *mut nrf_socklen_t
) -> c_int
```

[−] Accept a new connection a socket.

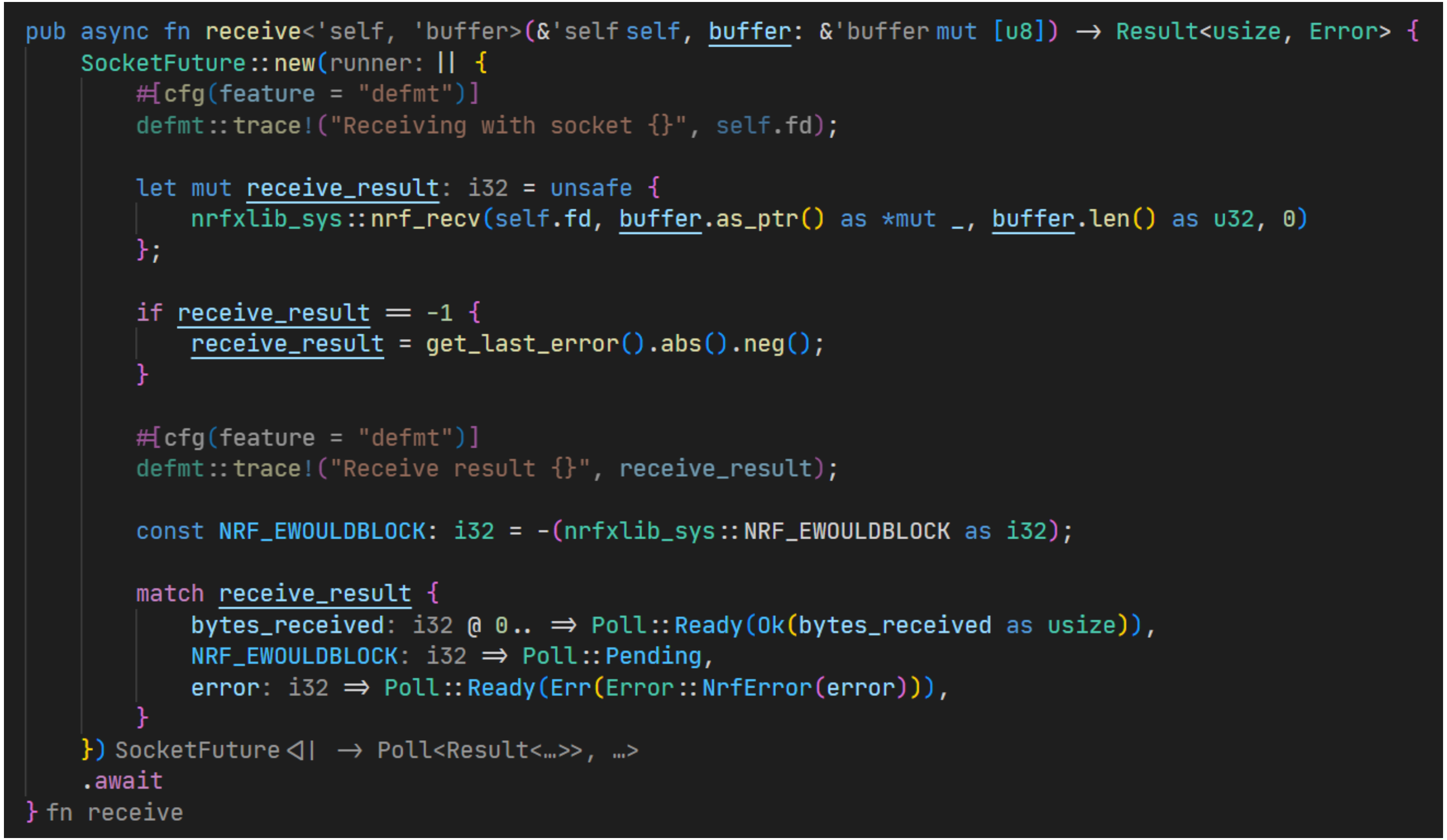
s See POSIX.1-2017 article for normative description.

In addition, the function shall return -1 and set the following errno: NRF_ESHUTDOWN Modem was shut down.

tweede golf https://github.com/nrf-rs/nrfxlib-sys

High level wrapper

Use the low level C function to create a proper Rust wrapper

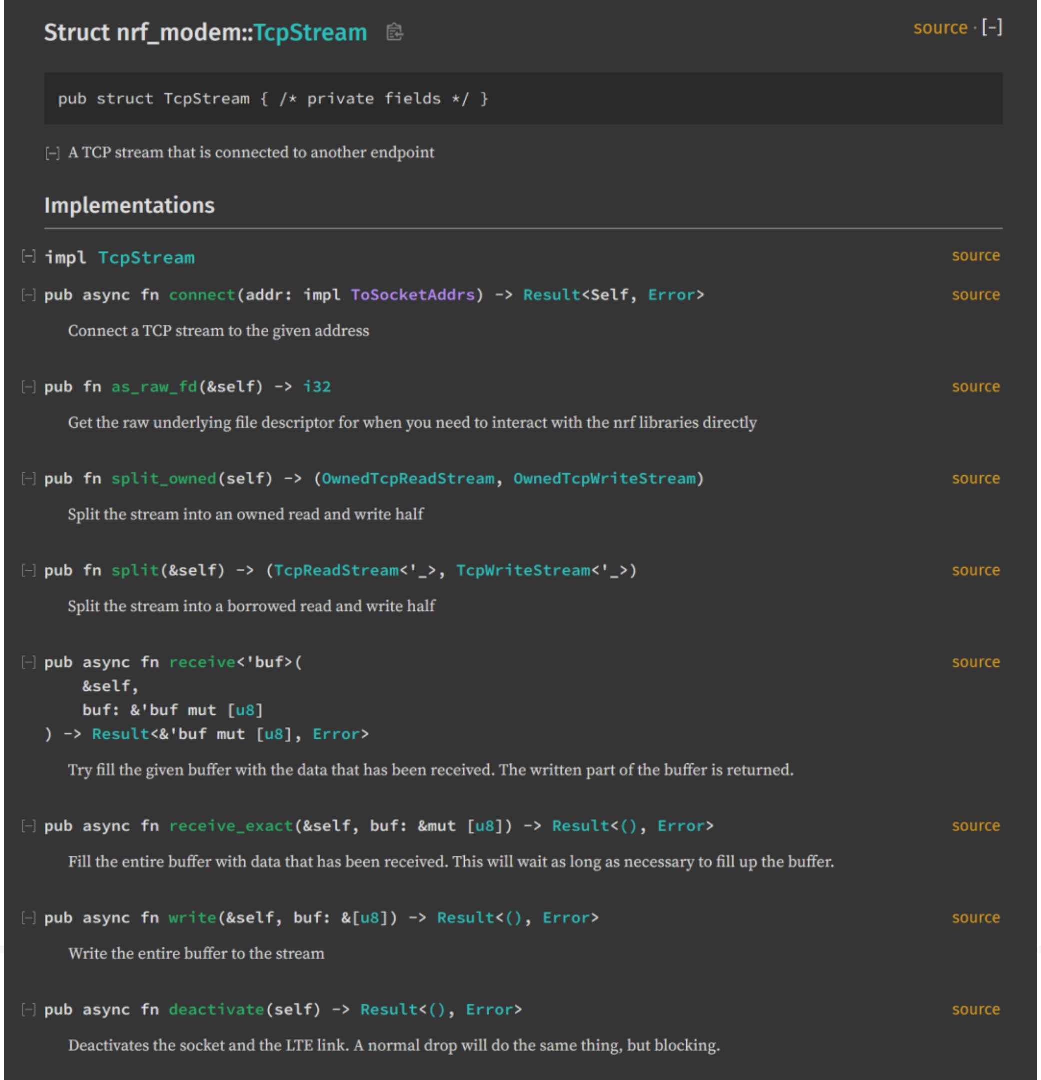


```rust
pub async fn receive<'self, 'buffer>(&'self self, buffer: &'buffer mut [u8]) -> Result<usize, Error> {
    SocketFuture::new(runner: || {
        #[cfg(feature = "defmt")]
        defmt::trace!("Receiving with socket {}", self.fd);

        let mut receive_result: i32 = unsafe {
            nrfxlib_sys::nrf_recv(self.fd, buffer.as_ptr() as *mut _, buffer.len() as u32, 0)
        };

        if receive_result == -1 {
            receive_result = get_last_error().abs().neg();
        }

        #[cfg(feature = "defmt")]
        defmt::trace!("Receive result {}", receive_result);

        const NRF_EWOULDBLOCK: i32 = -(nrfxlib_sys::NRF_EWOULDBLOCK as i32);

        match receive_result {
            bytes_received: i32 @ 0.. => Poll::Ready(Ok(bytes_received as usize)),
            NRF_EWOULDBLOCK: i32 => Poll::Pending,
            error: i32 => Poll::Ready(Err(Error::NrfError(error))),
        }
    }) SocketFuture<|| -> Poll<Result<…>>, …>
    .await
} fn receive
```

tweede golf

Very nice interface

Easy to use,
hard to misuse



Struct nrf_modem::TcpStream

source · [-]

```
pub struct TcpStream { /* private fields */ }
```

[-] A TCP stream that is connected to another endpoint

Implementations

```
[-] impl TcpStream                                                                    source
[-] pub async fn connect(addr: impl ToSocketAddrs) -> Result<Self, Error>             source
```

Connect a TCP stream to the given address

```
[-] pub fn as_raw_fd(&self) -> i32                                                    source
```

Get the raw underlying file descriptor for when you need to interact with the nrf libraries directly

```
[-] pub fn split_owned(self) -> (OwnedTcpReadStream, OwnedTcpWriteStream)             source
```

Split the stream into an owned read and write half

```
[-] pub fn split(&self) -> (TcpReadStream<'_>, TcpWriteStream<'_>)                    source
```

Split the stream into a borrowed read and write half

```
[-] pub async fn receive<'buf>(                                                       source
        &self,
        buf: &'buf mut [u8]
    ) -> Result<&'buf mut [u8], Error>
```

Try fill the given buffer with the data that has been received. The written part of the buffer is returned.

```
[-] pub async fn receive_exact(&self, buf: &mut [u8]) -> Result<(), Error>            source
```

Fill the entire buffer with data that has been received. This will wait as long as necessary to fill up the buffer.

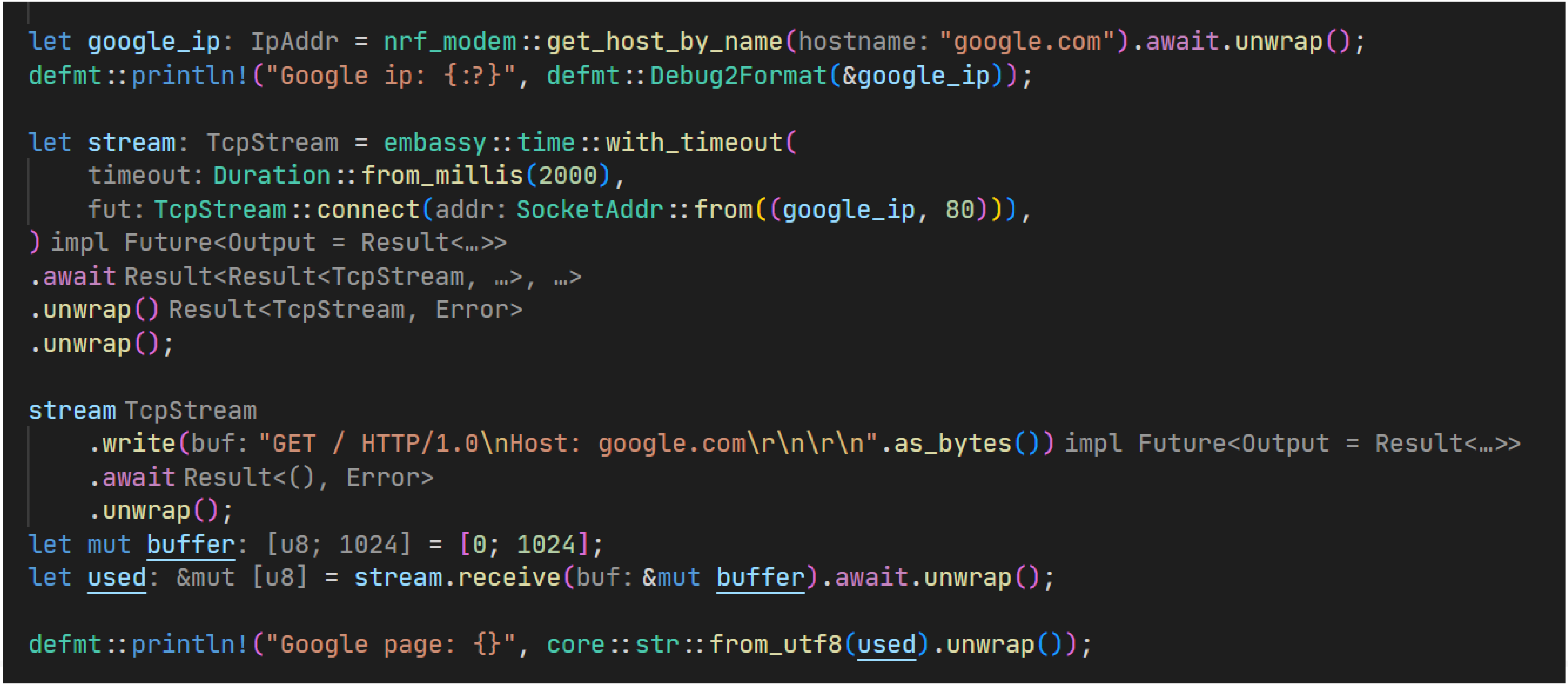
```
[-] pub async fn write(&self, buf: &[u8]) -> Result<(), Error>                        source
```

Write the entire buffer to the stream

```
[-] pub async fn deactivate(self) -> Result<(), Error>                                source
```

Deactivates the socket and the LTE link. A normal drop will do the same thing, but blocking.

Now we can use it



```
let google_ip: IpAddr = nrf_modem::get_host_by_name(hostname: "google.com").await.unwrap();
defmt::println!("Google ip: {:?}", defmt::Debug2Format(&google_ip));

let stream: TcpStream = embassy::time::with_timeout(
    timeout: Duration::from_millis(2000),
    fut: TcpStream::connect(addr: SocketAddr::from((google_ip, 80))),
) impl Future<Output = Result<…>>
.await Result<Result<TcpStream, …>, …>
.unwrap() Result<TcpStream, Error>
.unwrap();

stream TcpStream
    .write(buf: "GET / HTTP/1.0\nHost: google.com\r\n\r\n".as_bytes()) impl Future<Output = Result<…>>
    .await Result<(), Error>
    .unwrap();
let mut buffer: [u8; 1024] = [0; 1024];
let used: &mut [u8] = stream.receive(buf: &mut buffer).await.unwrap();

defmt::println!("Google page: {}", core::str::from_utf8(used).unwrap());
```

tweede golf

Castellastraat 26, 6512 EX Nijmegen

024 3010 484