How to SEGFAULT in Rust

- Exploring the Foreign Function Interface

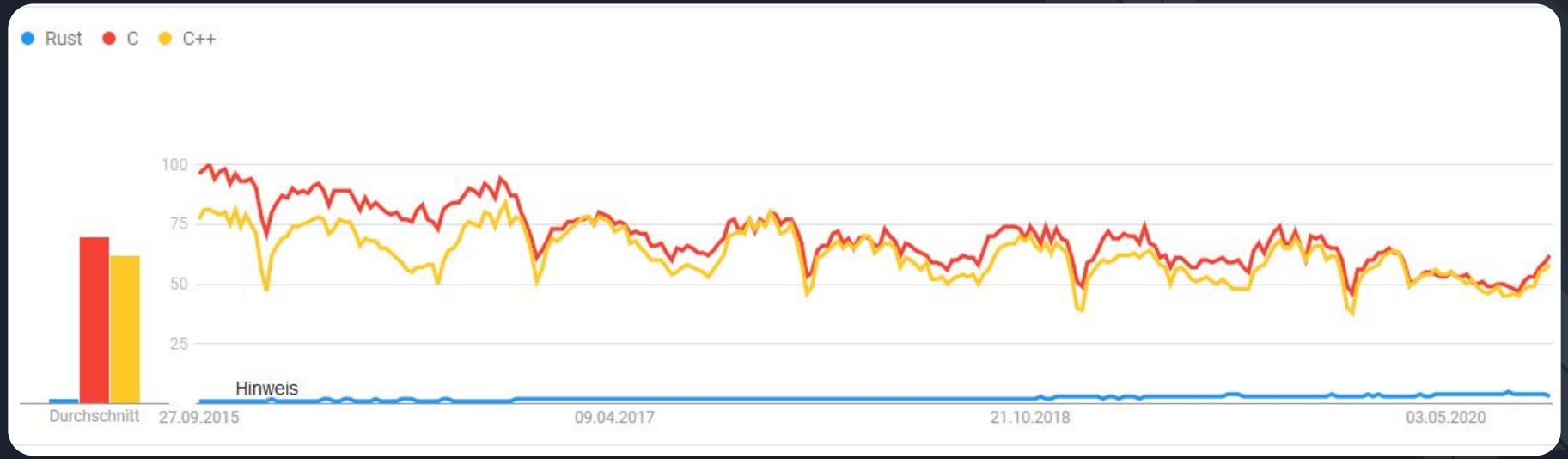
About $presenter

- Currently working on MCU firmware in C, C++
- Passionate about Rust
- C isn’t going anywhere though (and I love it)

- => A while ago, I looked for a way to combine C, C++, Rust
  - Initially, as an excuse to work on Rust at my workplace at the time :)

Rust, C, and C++ in Google Trends

Rust is small; must cooperate with the big guys



https://trends.google.com/trends/explore?date=today%205-y&q=%2Fm%2F0dsbpg6,%2Fm%2F01t6b,%2Fm%2F0jgqg

Josh Triplett about supporting Rust in the Linux Kernel:

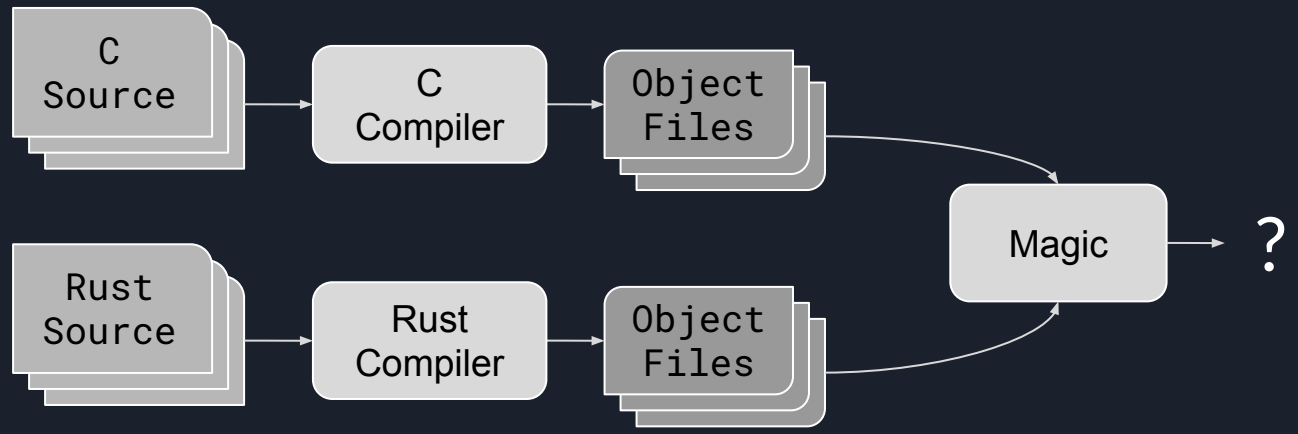
```
- There [need to be] appropriate Rustic interfaces that are natural and safe to
  use (not just C FFI, and not *just* trivial transformations like
  slices instead of buffer+len pairs).
- Those Rustic interfaces [must be] easy to maintain and evolve with the kernel.
- We [must] provide compelling use cases that go beyond just basic safety, such
  as concurrency checking, or lifetimes for object ownership.
```

=> While the FFI makes many things possible, actually creating an idiomatic interface seems to be a challenge.

https://lkml.org/lkml/2020/7/9/952

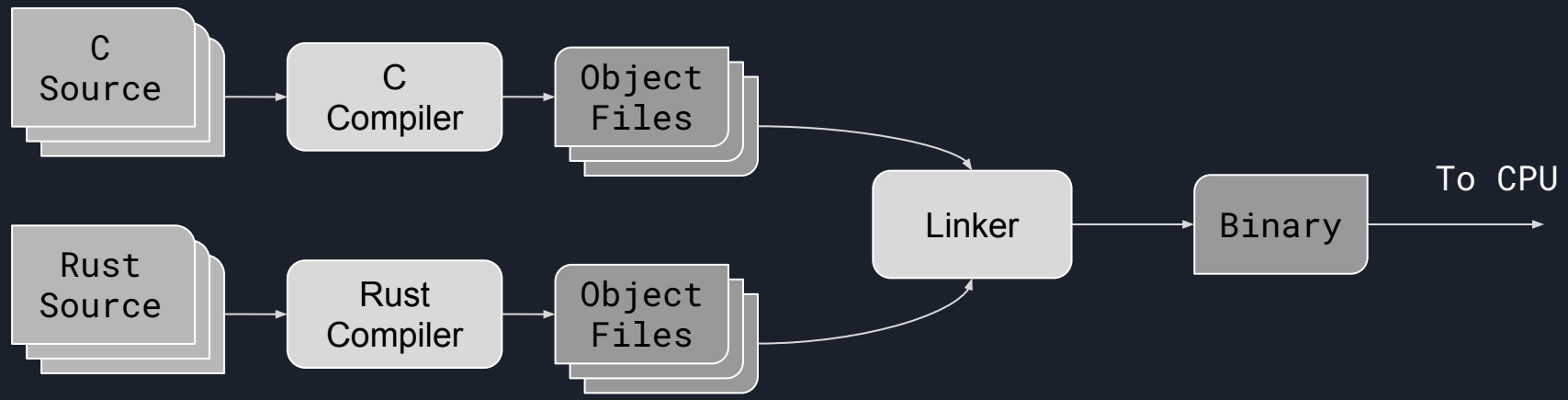
FFI Basics

- Processor: “It’s all binary to me.”
  - => C and Rust can be linked together in a single binary



FFI Basics

- Processor: “It’s all binary to me.”
  - => C and Rust can be linked together in a single binary



Tools for defining the FFI Boundary

`bindgen`: Generate Rust Source Code from C Headers

- Generated Rust source gets quite big
- `unions`, `bitfields` a bit difficult to map

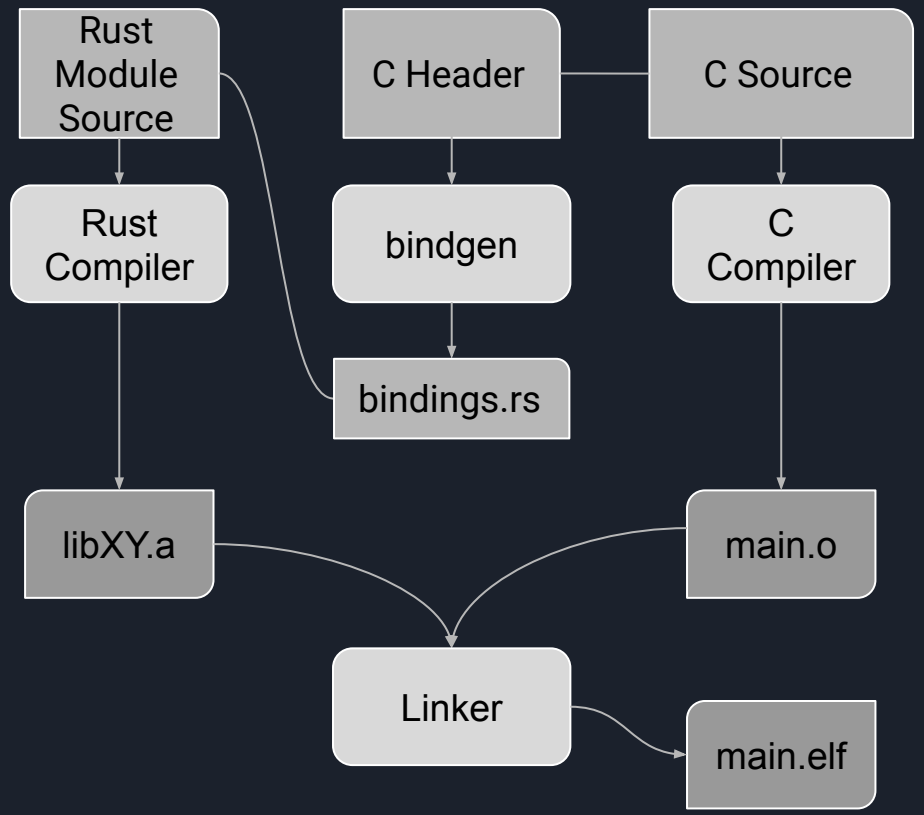
`cbindgen`: Generate C Headers from Rust Source

- Actually quite compact C headers!
- Enums become tagged unions
- ...

FFI using static linking

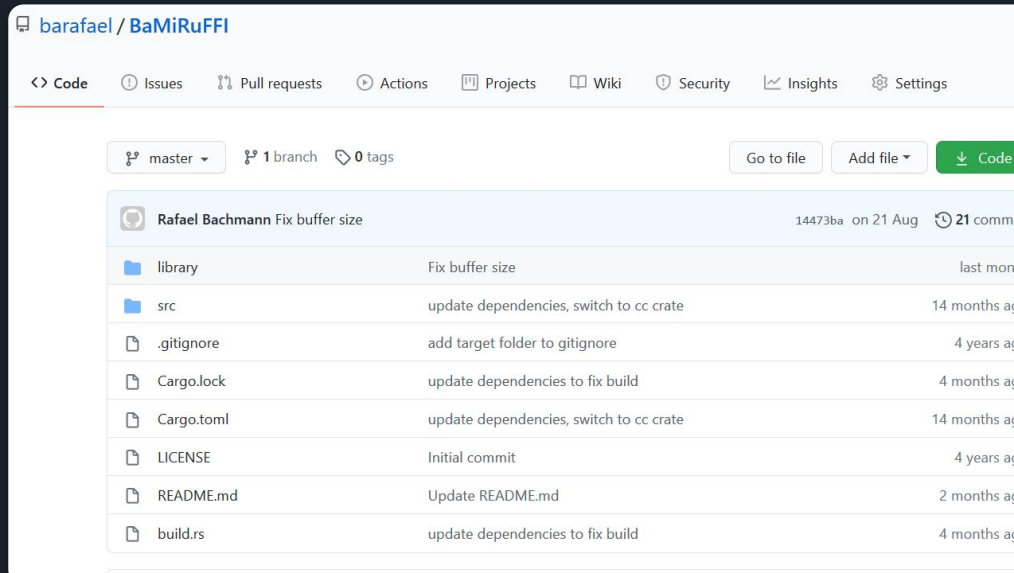
But why?

- C Libraries
- Use legacy C code
- Proprietary code blobs
- **Rust modules in larger C programs**



Bare **Mi**nimum **Ru**st **FFI** Example

```rust
let bindings = bindgen::Builder::default()
    .generate_comments(true)
    .emit_builtins()
    .header("library/library.h")
    .generate()
    .expect("Unable to generate bindings!");
```



https://github.com/barafael/BaMiRuFFI

Bare **Mi**nimum **Ru**st **FFI** Example

- Cargo.toml has **build-only** dependencies
  - bindgen
  - cc
- `build.rs` runs pre-build, generates binaries and bindings
- in `main.rs`:

```
include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```

Entire tooling in Rust and via Cargo!

https://github.com/barafael/BaMiRuFFI

Bare **Mi**nimum **Ru**st **FFI** Example

C FFI tooling remained relatively stable (for small example)

Dependencies needed occasional updates (compilation actually broke)

Commits on May 19, 2020

update dependencies to fix build
barafael committed on 19 May

Commits on Jul 18, 2019

update dependencies, switch to cc crate
barafael committed on 18 Jul 2019

Commits on Jun 26, 2018

update C string functions
barafael committed on 26 Jun 2018

stop using deprecated gcc compile interface
barafael committed on 26 Jun 2018

update dependencies
barafael committed on 26 Jun 2018

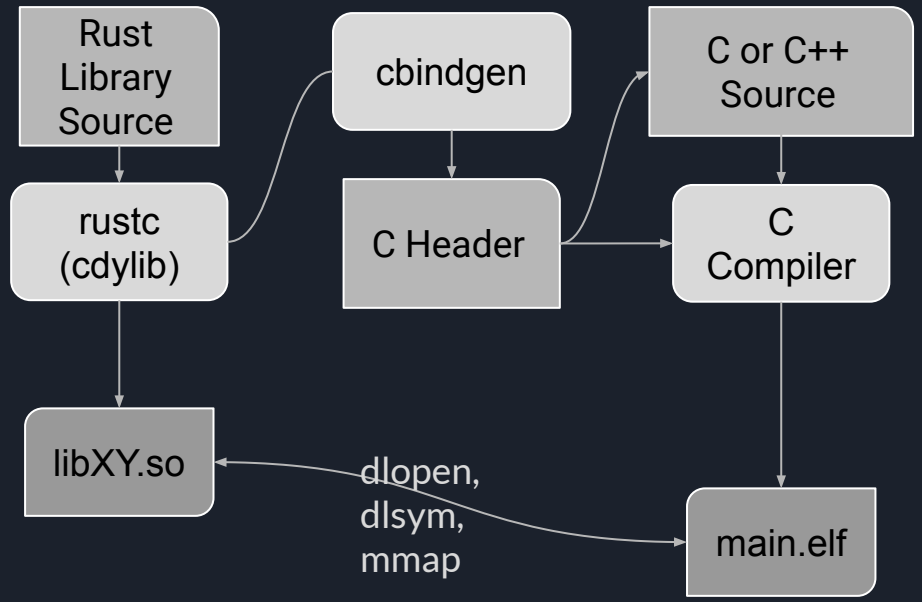
Commits on Jun 27, 2017

https://github.com/barafael/BaMiRuFFI

FFI (dynamic linking) on Host Platform

Use Case:

- Plugins (VST)
- Shared Library Code (SSL, ncurses, QT, ...)
- Package Management



Note: don’t forget to compile for shared lib :)

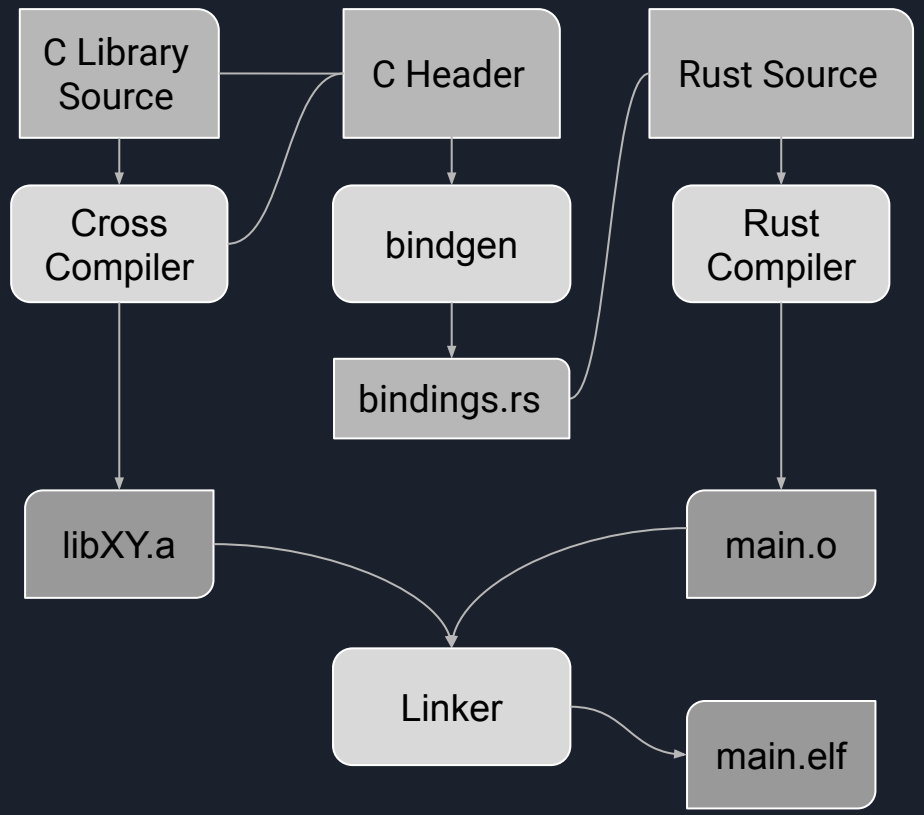
```
// in Cargo.toml:

[lib]
crate-type = [“cdylib”]

// that’s all!
```

error while loading shared libraries libncurses.so.5

FFI (static linking) on a Microcontroller



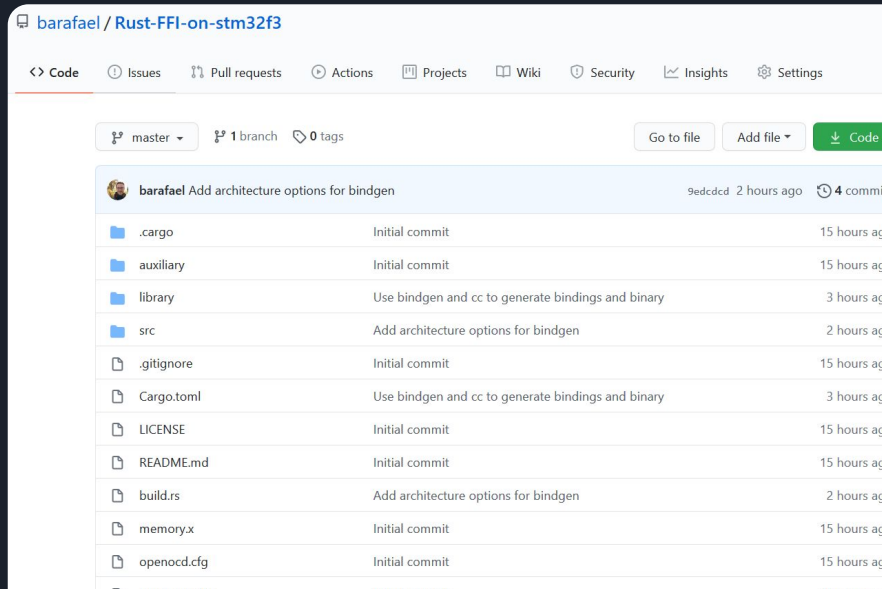
But why?

- DSP algorithms
- C HAL libraries
- Proprietary blobs (Sensor Fusion etc.)
- Run customer code blob
- Bootloader
- Firmware Upgrade

Note: don’t forget target-specific flags :)

FFI on MCU Example Repo

```
cc::Build::new()
    .file("library/foo.c")
    .include("library")
    .flag("-mcpu=cortex-m4")
    .flag("-mfpu=fpv4-sp-d16")
    .flag("-mfloat-abi=hard")
    .compile("foo");
```



https://github.com/barafael/Rust-FFI-on-stm32f3

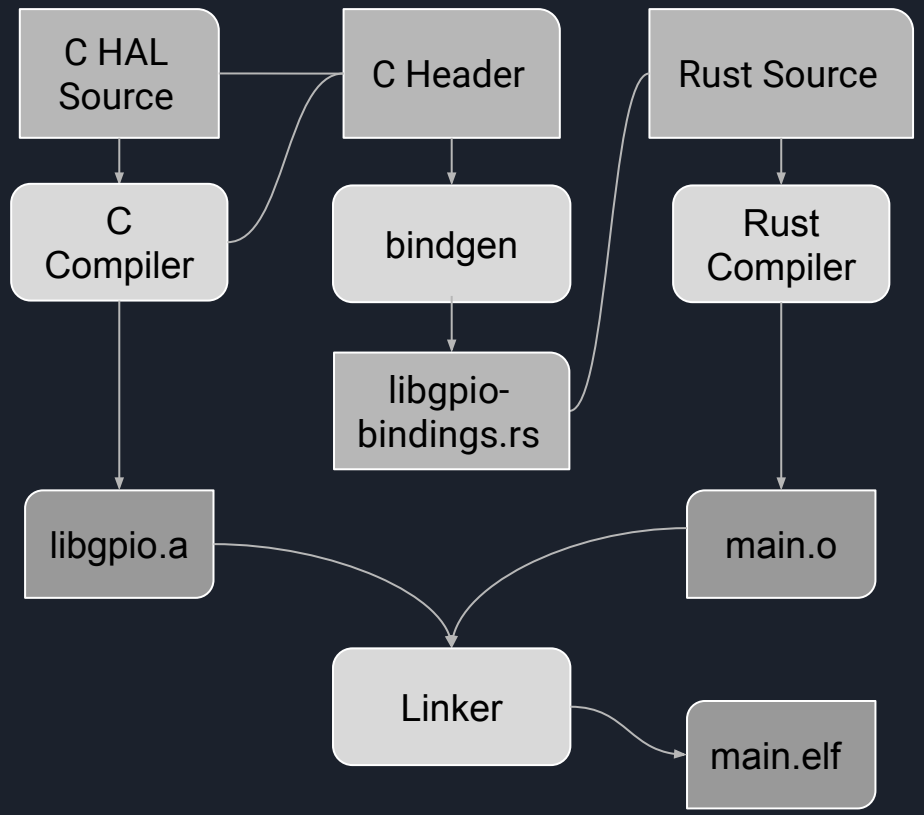
FFI on MCU **Hardware Abstraction Layer**

Just an example of FFI on MCU

HAL sits below application logic

But why?

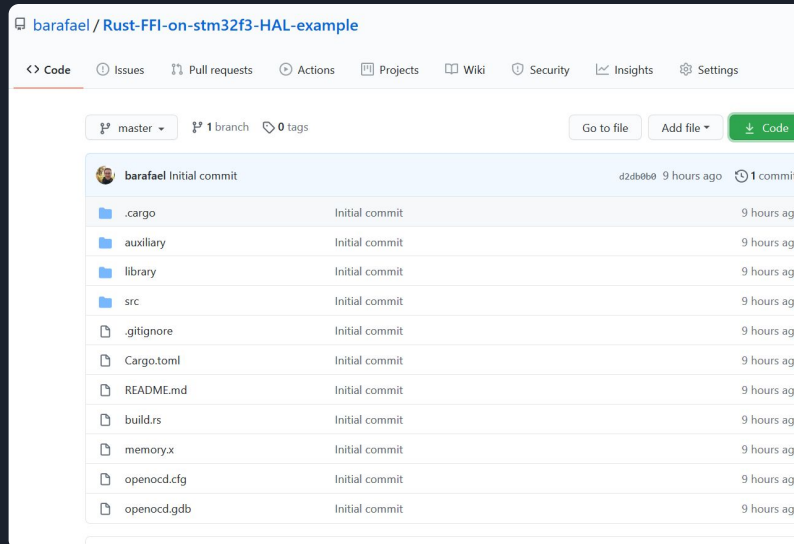
- embedded-hal not yet mature
- using existing codebases
- as show-off



FFI on MCU **Hardware Abstraction Layer**

```rust
let mut gpioe_handle: *mut GPIO_TypeDef =
        GPIOE_BASE as *mut GPIO_TypeDef;

loop {
    unsafe {
        HAL_GPIO_TogglePin(gpioe_handle, 0x0100u16);
    }
    delay.delay_ms(100u16);
}
```



https://github.com/barafael/Rust-FFI-on-stm32f3-HAL-example

“Poor Man’s libc”
-- what is `std::os::raw::c_int`?

```
let bindings = bindgen::Builder::default()
    .clang_arg("-mcpu=cortex-m4")
    //...
    .ctypes_prefix("libc")
    .use_core()
    .raw_line("#[no_std]")
    .raw_line("mod libc {
    pub type c_uint = u32;\
    // ... all other basic C types
    pub enum c_void {}\
    }")
    .header("library/header.h")
    .generate()
```

https://github.com/rust-lang/rust-bindgen/pull/218

General FFI Challenges

- Common Types, Memory Layout
- Name Mangling?
  - Generics: Monomorphisation mangles names by design
  - Macros in general can generate functions
- Calling Convention?
- Cross-language boundary Resource Management
  - Language Runtimes/Interpreters
  - Files, Sockets, Reference Counters, Garbage Collection, …
- Cross-language Exceptions?!
  - Foreign **FUNCTION** Interface; Exceptions “pollute” the call stack

Some of these problems even apply to FFI with C++ to C!

Rust FFI Challenges

- Keeping Rust invariants alive when calling into C code:
  - Null pointer
  - Pointer to dead stackframe
  - Misaligned pointer
  - Dangling Pointer
  - Aliased Pointer
  - … probably many more

- Designing an API around `unsafe` C access
  - Probably way harder than anything discussed so far