Embedded Rust Sales Pitch

Ben Brown

bbrown1867/embedded-rust-pitch

> Overview

Target Audience

➔ C/C++ developers working with microcontrollers

➔ Other curious developers!

Outline

➔ What is “Embedded Rust”?

➔ Peripheral access in Rust

➔ Application of language features for embedded

➔ Embedded ecosystem and tooling

References

[1] *The Rust Programming Language*, *Klabnik and Nichols*

[2] *Rust: Putting Ownership to Use*, *Niko Matsakis*

[3] *The Embedded Rust Book*

[4] *Rust by Example*

> History

Rust

- ➔ Designed by Graydon Hoare at Mozilla Research
- ➔ Compiled, statically typed, multi-paradigm language
- ➔ 1.0 release in 2015

Embedded Rust

- ➔ Working Group formed to bring Rust to embedded devices
- ➔ “1.0” in 2018
- ➔ Key features:
  - ◆ #![no_std] - Bring your own runtime
  - ◆ Stable support for ARM Cortex-M
  - ◆ Large ecosystem of embedded crates

Success Stories

- ➔ Dropbox
- ➔ Oxide Computer

> FizzBuzz

```
fn fizz_buzz(limit: u32) {
    for i in 1..limit {
       match (i % 3, i % 5) {
            (0, 0) => println!("fizzbuzz"),
            (0, _) => println!("fizz"),
            (_, 0) => println!("buzz"),
            (_, _) => println!("{}", i),
       }
    }
}

fn main() {
    let limit = 101;
    fizz_buzz(limit);
}
```

> Why choose Rust?

	Modern Conveniences	Zero-Cost Abstractions	Memory Safety	Great Ecosystem and Tooling	Hardware Vendor Support
C	⛔	⛔	⛔	⛔	✅
C++	✅	✅	⛔	⛔	✅
Rust	✅	✅	✅	✅	🕒

> Minimal Embedded Rust Application

```rust
#![no_std]
#![no_main]

use cortex_m_rt::entry;
use stm32f7::stm32f7x7;

// All Rust programs need a panic handler
extern crate panic_halt;

#[entry]
fn main() -> ! {
    loop {}
}
```

> Minimal Embedded Rust Application

```
[dependencies]
cortex-m = "0.6.0"
cortex-m-rt = "0.6.10"
stm32f7 = { version = "0.14.0", features = ["rt", "stm32f7x7"] }
panic-halt = { version = "0.2.0" }
```

> Peripheral Access

Embedded C

```
// Vendor header files
struct GPIO {
   uint32_t ODR;
   ...
}

#define pGPIOB          ((GPIO*)  0x40000000)
#define GPIO_ODR_BITP_7  (7)

// Your code: Clear output on pin 7 of GPIOB
pGPIOB->ODR &= ~(1 << GPIO_ODR_BITP_7)
```

- ➔ Error prone
- ➔ Not ergonomic
- ➔ No access control or ownership

> Peripheral Access

- ➔ Can replicate the embedded C code using unsafe Rust
  - ◆ Raw pointer dereference

- ➔ But **Peripheral Access Crates (PAC)** provide a safe interface
  - ◆ Peripherals implemented as singletons
  - ◆ Register/field access with zero-cost functions (release)
  - ◆ Register access functions only exposed when relevant: read, write, modify

> Peripheral Access

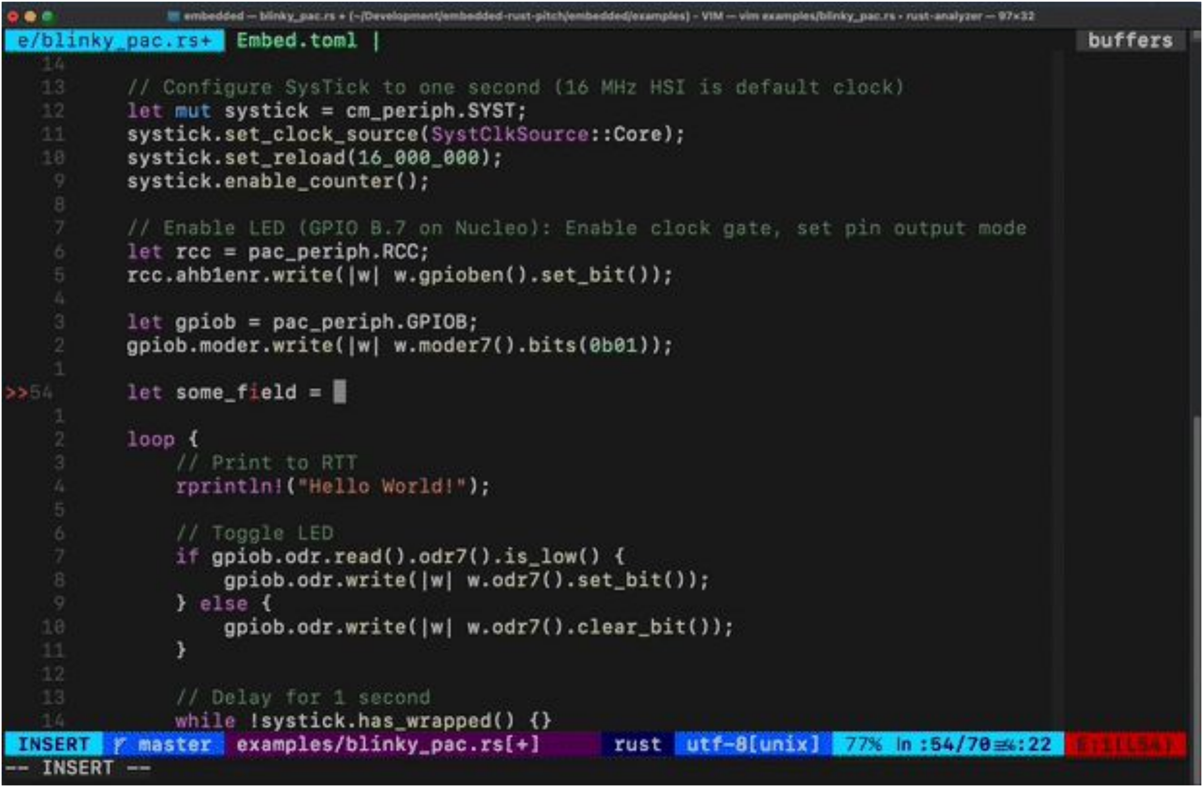
Rust

```
let pac_periph = stm32f7x7::Peripherals::take().unwrap();
let gpiob = pac_periph.GPIOB;
gpiob.odr.write(|w| w.odr7().clear_bit());
```

Why is this better?

- No bitwise operations necessary
- Type system prevents writing a read-only field
- Register-level and field-level autocomplete in IDE/editor
- Exclusive access to peripheral via take API
  - Sharing will follow Rust’s memory model

> Peripheral Access



embedded — blinky_pac.rs + (~/Development/embedded-rust-pitch/embedded/examples) - VIM — vim examples/blinky_pac.rs • rust-analyzer — 97×32

```
e/blinky_pac.rs+  Embed.toml |                                                        buffers

    // Configure SysTick to one second (16 MHz HSI is default clock)
    let mut systick = cm_periph.SYST;
    systick.set_clock_source(SystClkSource::Core);
    systick.set_reload(16_000_000);
    systick.enable_counter();

    // Enable LED (GPIO B.7 on Nucleo): Enable clock gate, set pin output mode
    let rcc = pac_periph.RCC;
    rcc.ahb1enr.write(|w| w.gpioben().set_bit());

    let gpiob = pac_periph.GPIOB;
    gpiob.moder.write(|w| w.moder7().bits(0b01));

    let some_field = 

    loop {
        // Print to RTT
        rprintln!("Hello World!");

        // Toggle LED
        if gpiob.odr.read().odr7().is_low() {
            gpiob.odr.write(|w| w.odr7().set_bit());
        } else {
            gpiob.odr.write(|w| w.odr7().clear_bit());
        }

        // Delay for 1 second
        while !systick.has_wrapped() {}
 INSERT  ᚠ master  examples/blinky_pac.rs[+]        rust  utf-8[unix]  77% ln :54/70≡:22  E:1(L54)
-- INSERT --
```

> PAC Blinky

```
let cm_periph = cortex_m::Peripherals::take().unwrap();
let pac_periph = stm32f7x7::Peripherals::take().unwrap();

// Configure SysTick to one second (16 MHz HSI is default clock)
let mut systick = cm_periph.SYST;
systick.set_clock_source(SystClkSource ::Core);
systick.set_reload(16_000_000);
systick.enable_counter();

// Enable LED (GPIO B.7 on Nucleo)
let rcc = pac_periph.RCC;
rcc.ahb1enr.write(|w| w.gpioben().set_bit());

let gpiob = pac_periph.GPIOB;
gpiob.moder.write(|w| w.moder7().bits(0b01));
```

> PAC Blinky

```
loop {
     // Toggle LED
     if gpiob.odr.read().odr7().is_low() {
         gpiob.odr.write(|w| w.odr7().set_bit());
     } else {
         gpiob.odr.write(|w| w.odr7().clear_bit());
     }

     // Delay for 1 second
     while !systick.has_wrapped() {}
 }
```

> Modern Conveniences :: Enums

```
enum Option<T> {
    None,
    Some(T),
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

> Modern Conveniences :: Enums

Embedded C

```c
error_t read_sensor_data(uint16_t* pResult) {
    if (pResult == NULL) {
        printf("Explode!");
    }
    *pResult = 42;
    return SUCCESS;
}

error_t process_sensor_data(void) {
     uint16_t sensor_value;
     error_t err = read_sensor_data(&sensor_value);
     if (err == SUCCESS) {
        printf("Value = %d\n", sensor_value);
        return SUCCESS;
     } else {
        return err;
    }
}
```

> Modern Conveniences :: Enums

Rust

```rust
fn read_sensor_data() -> Result<u16, Error> {
    Ok(42)
}

fn process_sensor_data() -> Result<(), Error> {
    let sensor_value = read_sensor_data()?;
    println!("Value = {}", sensor_value);
    Ok(())
}
```

> Zero-Cost Abstractions :: Traits

```
pub trait Mul<Rhs = Self> {
    type Output;
    fn mul(self, rhs: Rhs) -> Self::Output;
}
```

> Zero-Cost Abstractions :: Traits

```
use std::ops::Mul;

struct Point<T> {
    x: T,
    y: T,
}

impl<T: Mul<Output = T> + Copy> Mul for Point<T> {
    type Output = Self;

    fn mul(self, rhs: Self) -> Self {
        Self {
            x: self.x * rhs.x,
            y: self.y * rhs.y,
        }
    }
}
```

> Zero-Cost Abstractions :: Traits

Platform Agnostic Drivers: embedded-hal

- ➔ Set of traits for common embedded software APIs
- ➔ HAL provides implementations of the traits
- ➔ Used to abstract higher-level drivers (e.g. sensors)

I2C Read Trait

```
pub trait Read<A: AddressMode = SevenBitAddress> {
    type Error;
    fn read(
        &mut self,
        address: A,
        buffer: &mut [u8]
    ) -> Result<(), Self::Error>;
}
```

> Zero-Cost Abstractions :: Traits

Usage

```
impl<I2C, E> Sensor<I2C>
where
    I2C: i2c::Read<Error = E> + i2c::Write<Error = E>,
{
    fn read_sensor(&self, i2c: &mut I2C, buf: &mut [u8]) {
        match i2c.read(self.i2c_address, buf) {
            ...
        }
    }
```

> Zero-Cost Abstractions :: Typestate Programming

➔ Use type system to ensure drivers are not misconfigured

➔ Move run-time checks (e.g. is driver enabled?) to compile-time

```
/// GPIO interface
struct GpioConfig<ENABLED, DIR, MODE> {
    periph: GPIO,
    enabled: ENABLED,
    direction: DIR,
    mode: MODE,
}

// Will be optimized away by compiler
struct Disabled;
struct Enabled;
struct Output;
struct Input;
struct DontCare;
...
```

```
/// This function may be used on an Output Pin
impl GpioConfig<Enabled, Output, DontCare> {
    pub fn set_bit(&mut self, set_high: bool) {
        self.periph.modify(...);
    }
}
```

> Tooling

cargo

- → Define dependencies and features in *Cargo.toml*
- → Compile/Link: `cargo build`
- → Run: `cargo run`
- → Test: `cargo test`
- → HTML Docs: `cargo doc`
- → Formatting: `cargo fmt`
- → Install: `cargo install <crate>`
- → Extendable: `cargo <your_subcommand>` (e.g. `cargo watch`)

crates.io

- → Public crate (package) registry

rust-analyzer

- → Language Server Protocol for Rust
- → Easy integration with editors (VSCode, Vim, Emacs)

> Tooling

Embedded C

gcc/clang + gdb/lldb + cmake + make/ninja + doxygen + astyle + clang-analyzer + openocd

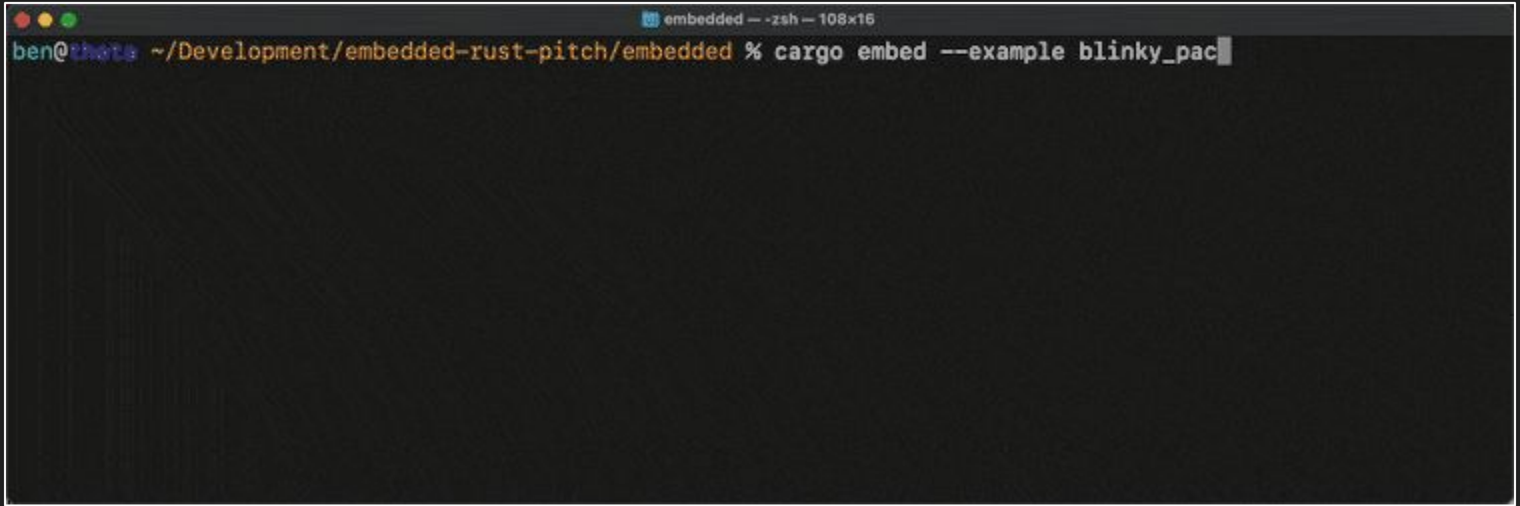
Rust

cargo (+ gdb)

> Tooling :: probe-rs

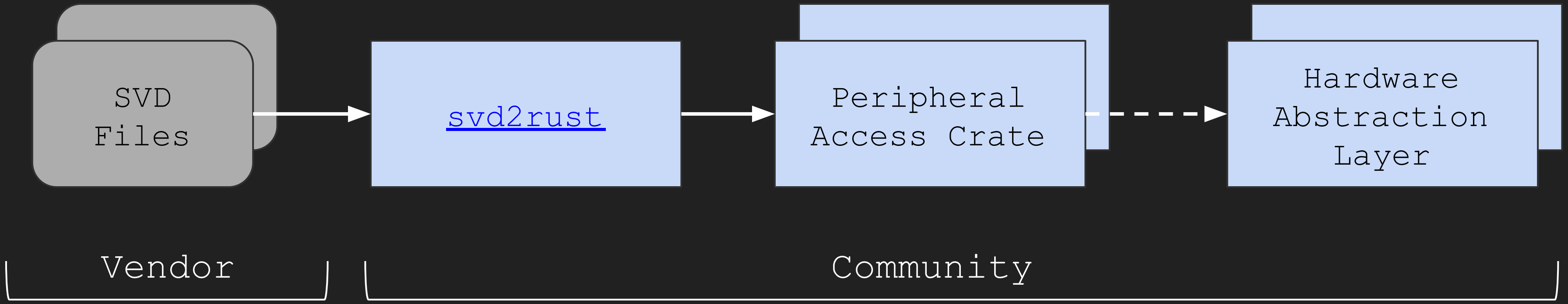
Embedded Debugging Toolkit

- → cargo flash - Replaces cargo run for embedded targets
- → cargo embed - Like cargo flash, but with RTT/GDB integration
- → Rust API for controlling the debugger
- → Supports ST-Link, J-Link, CMSIS-DAP today
- → VSCode integration (Microsoft Debug Adapter Protocol)



```
ben@[redacted] ~/Development/embedded-rust-pitch/embedded % cargo embed --example blinky_pac
```

> Crate Ecosystem :: Device Support



Hardware Abstraction Layer (HAL) Crates

- ➔ Peripheral drivers (e.g. GPIO, DMA, SPI, I2C)
- ➔ Common patterns, but unique implementation for each chip family

> Crate Ecosystem :: More!

- ➔ Hubris - RTOS
  - ◆ Memory protection for tasks
- ➔ RTIC - Real-Time Interrupt-driven Concurrency
  - ◆ Not quite an RTOS, but close
- ➔ defmt - Efficient logging framework for embedded systems
  - ◆ Support for `cargo test` on embedded devices

> Next Steps

Blocker	***Solution Ideas***
Lack of organizational knowledge	Learn Rust! Technical decisions are usually “bottom-up”
Large, legacy C/C++ codebases	Start small, with projects that are low risk (e.g. internal tools)
No PAC or HAL for hardware platform	Convey interest to vendors, try svd2rust, write your own HAL
No compiler support for hardware platform	Convey interest to vendors, GCC backend was accepted recently