

# Rust on bare metal:

Writing Sensor Drivers,  
delightfully.

```
Tr  
Trans  
];  
let mock = I2cMock::new(&e  
let mut sdp = Sdp8xx::new(mock  
let id = sdp.read_product_id().  
assert_eq!(  
    ProductVariant::Sdp800_125Pa { re  
    id.product_number  
);  
assert_eq!(0x445566778899aabb, id.serial_numbe  
sdp.destroy().done();
```

Rafael Bachmann



# Who am I?

Systems Software Developer (Rust)

Always chasing projects

Sometimes finishing projects

- “But this new idea is so interesting!”
- “Why finish a project if I can keep working on it?”

**Ideas are cheap!**

Implementation is expensive.

<https://github.com/barafael>



# Roadmap

- Context
  - Hardware development
- Software development
  - A sensor driver
  - The embedded-hal project
  - Testing and integration of an embedded-hal driver
- Takeaways

# Motivation



Remote Controlled Sailboat (human-operated)

<http://www.tippecanoeboats.com/t37-racing-sloop-2>


# Motivation

Flight Controller Firmware (C++, Frama-C)

- Radio Comms + Motor Control + Inertial Measurement Unit + PID control

<https://github.com/barafael/raPID>



 **barafael / raPID**

Unwatch 5

Star 11

Fork 1

> Code

Issues

Pull requests

Actions

Projects

Wiki


Security

Insights

Settings

master 7 branches 3 tags

Go to file Add file Code

 **barafael** Update README.md ✓ 41ab0c8 on 24 Nov 2018 🕒 374 commits

CalibrateMPU6050 @ 22c69f2	add calibration submodule	4 years ago
include	Revert "port watchdog back to C"	3 years ago
src	Revert "port watchdog back to C"	3 years ago
.clang-format	WS and formatting improvements	3 years ago
.gitignore	re-add git config files	3 years ago

About

flight controller running on ARM Cortex M4

remote-control

teensy

flight-controller

pid-control

flight-mode

transitional-mixers

Readme

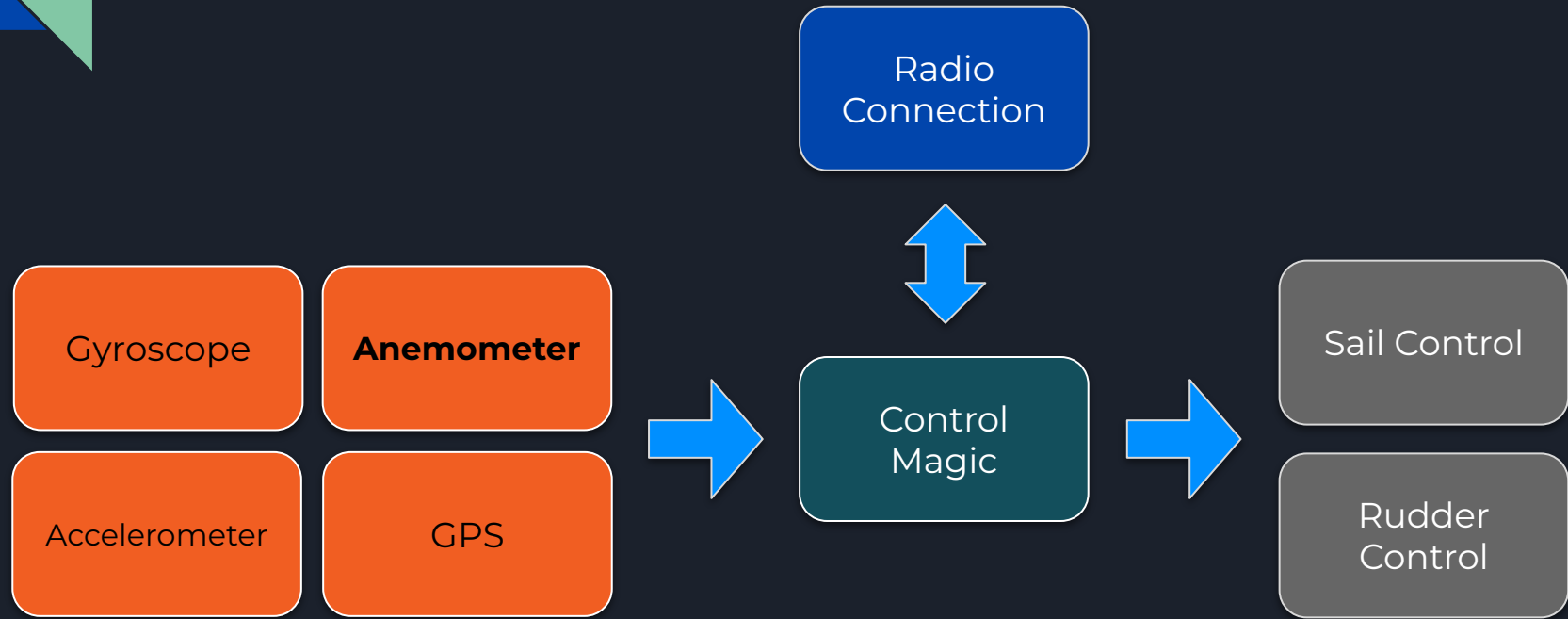
MIT License



## Idea: Autonomous Sailboat

- Combine mini sailboat with  
communication, sensing, control, autonomy
- Lots more sensors than on a multicopter
- Quite a bit more difficult
- Much much less dangerous.

# Sailboat modules



**So much stuff => focus on one module at a time.**

# Anemometer design - Sensor

“Where is the wind coming from, and how strong is it?”

Sensirion SDP8xx  
Differential Pressure Sensor

- Sensitive measurement of pressure differences
- “Wind is just pressure difference”
- I2C interface

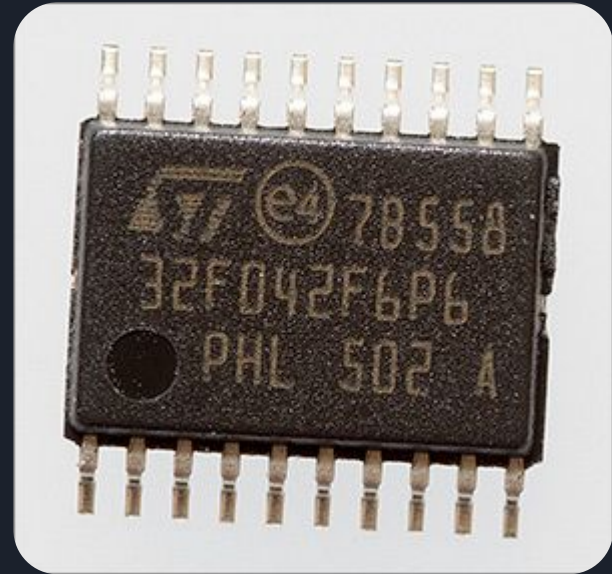




# Anemometer design - MCU

## STM32F042: Tiny Cortex-M0 MCU

- Hand Solderable
- 6kB SRAM, 32kB Flash
- I2C and serial interfaces
- Supported by the awesome rust-embedded ecosystem

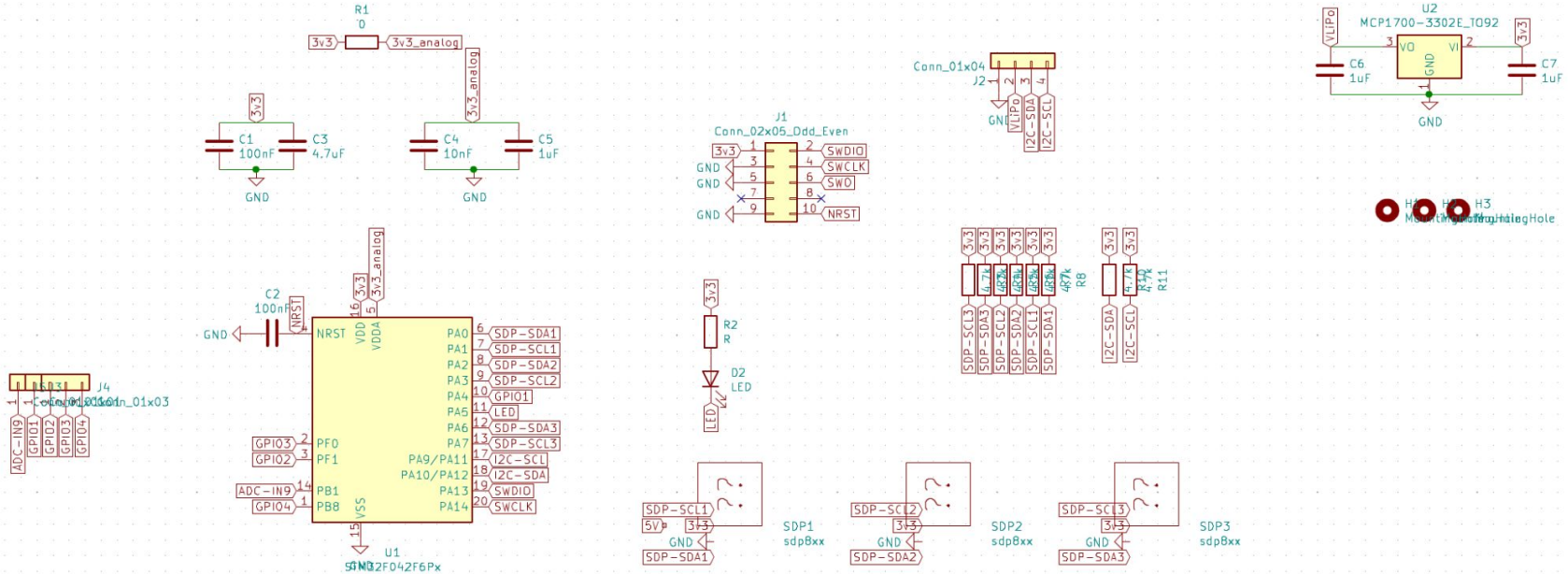


<https://github.com/stm32-rs/stm32f0xx-hal>

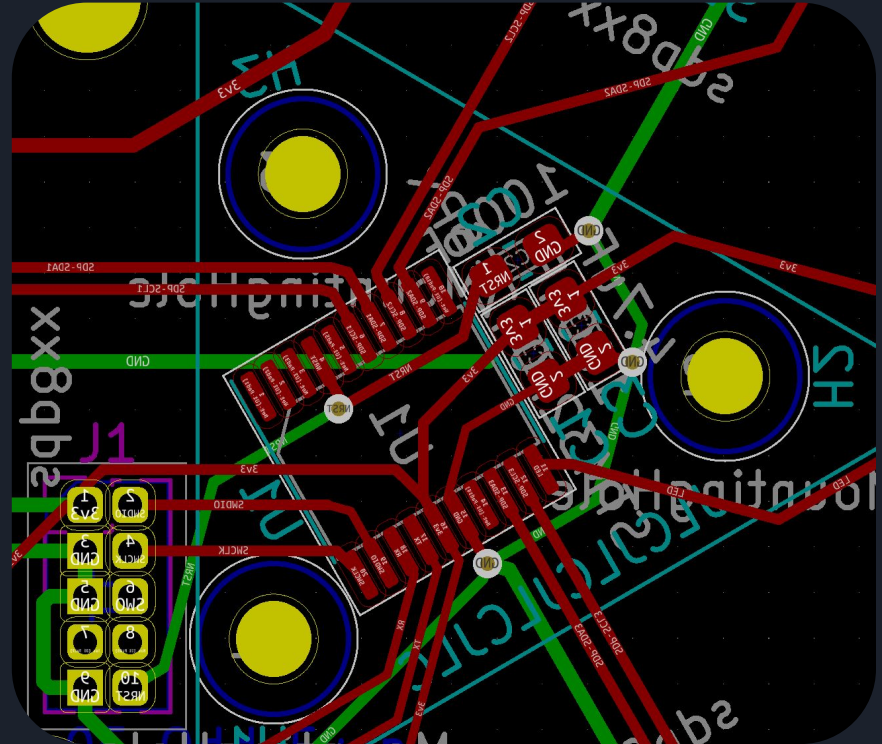
<https://www.st.com/en/microcontrollers-microprocessors/stm32f042f6.html>

# Anemometer design - Schematic

<https://github.com/barafael/sdp8xx-anemometer-pcb>

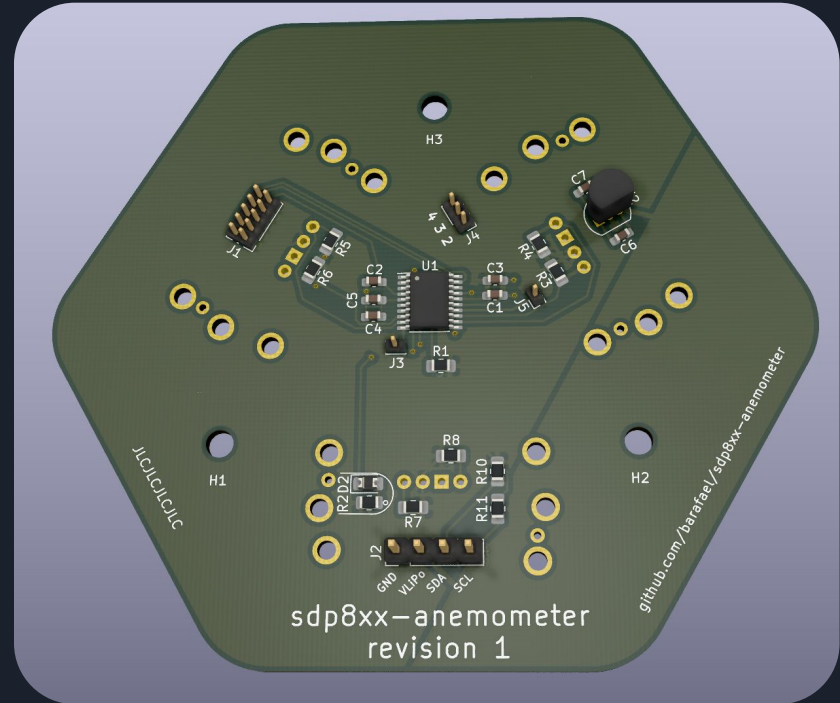
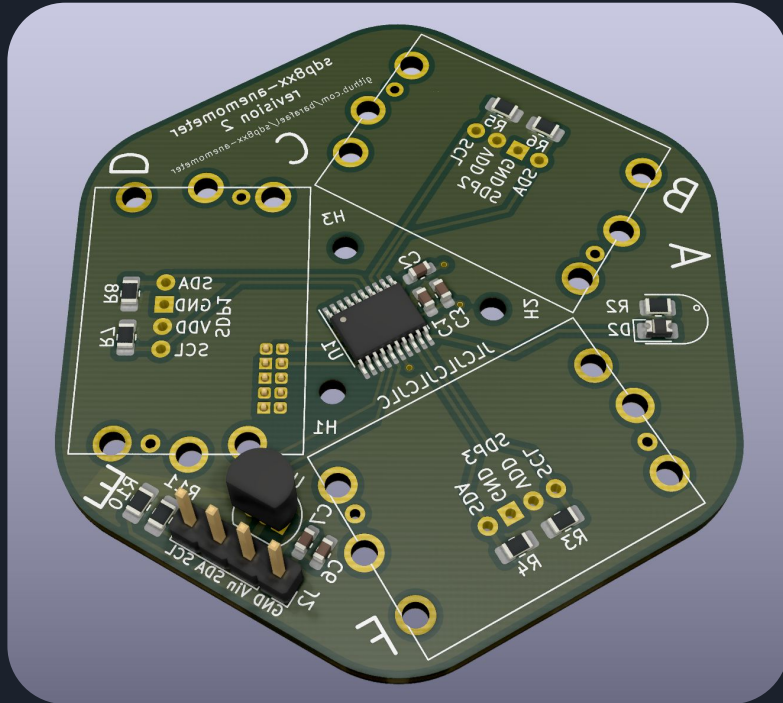


<https://github.com/barafael/sdp8xx-anemometer-pcb>



# Anemometer design - Board Renders

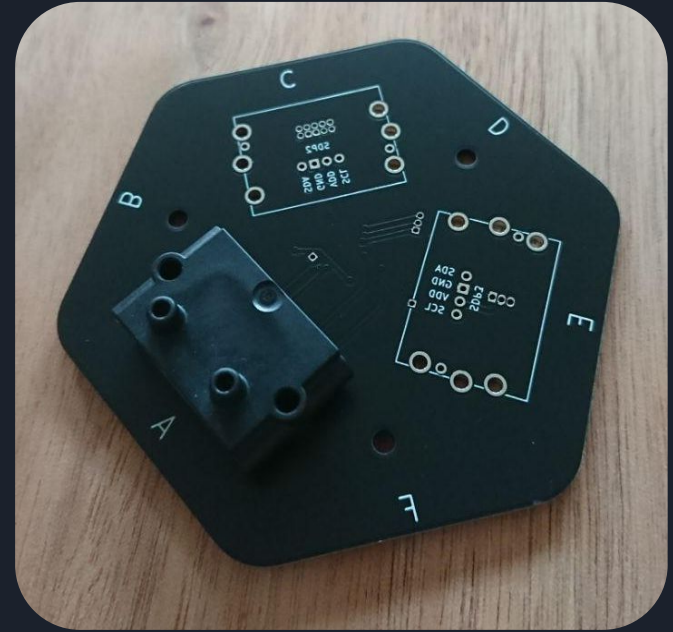
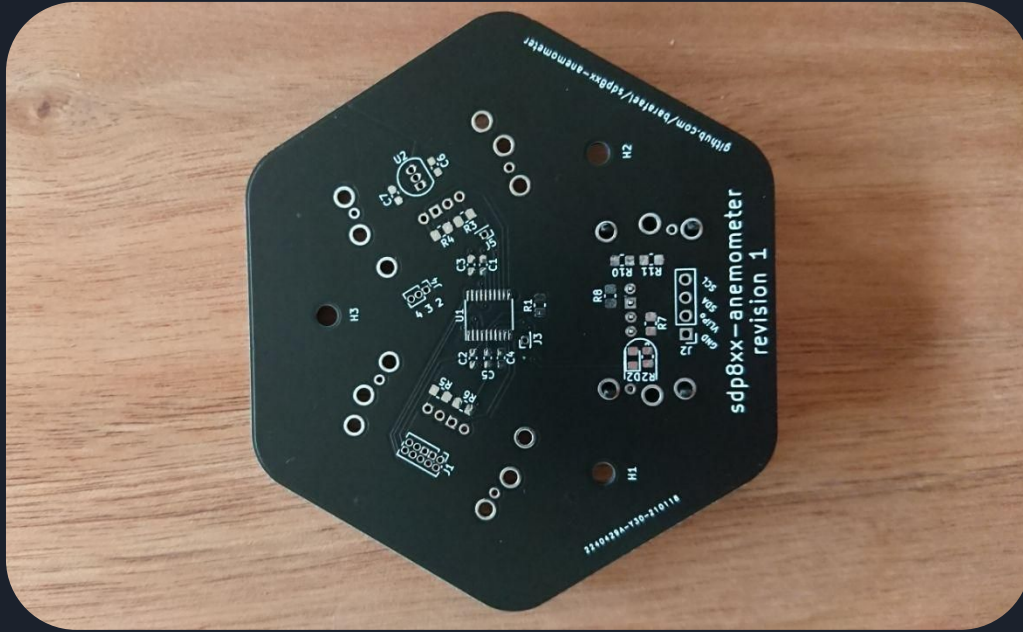
<https://github.com/barafael/sdp8xx-anemometer-pcb>





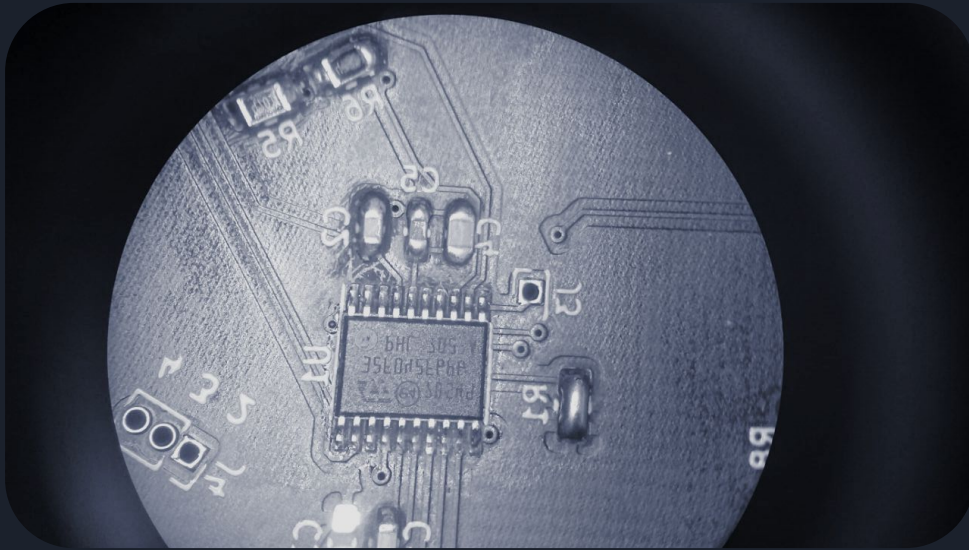
# Anemometer design - Finished PCBs

<https://github.com/barafael/sdp8xx-anemometer-pcb>



# Anemometer design - Soldering

#FixThePart #NotTheLayout



# Anemometer design - Finished Module



3D-Printed "Air Pods"



# Anemometer design - Software

Hardware dev is tedious (for me);

Lots of waiting (ordering parts and PCBs);

=> Concurrently do software and firmware, then integrate!

Now comes the easy part - software.





# The SDP8xx Driver - Objective

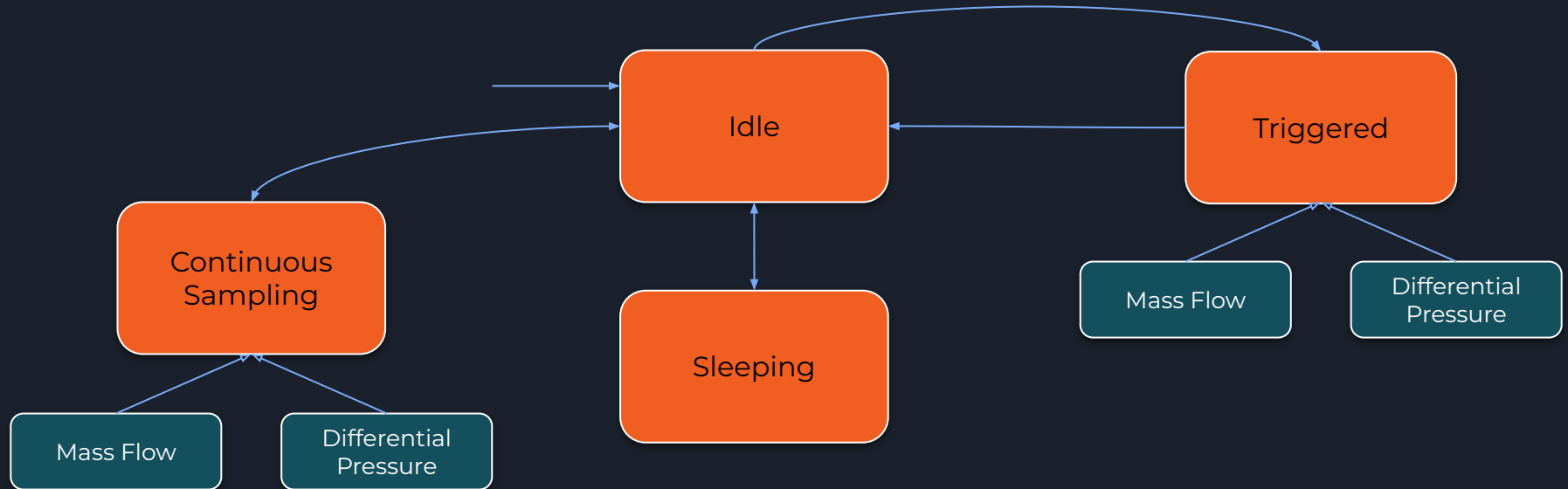
Objective:

Library for using the sdp8xx sensor!

Support for configuration, sampling, conversion.

# The SDP8xx Operation Modes

Read the datasheet :) find the **operation modes**





# Type State Programming

Implement the **operation modes** as marker types

```
pub struct IdleState;  
pub struct TriggeredState;  
pub struct SleepState;  
  
pub struct ContinuousSamplingState<MeasurementType> {  
    data_type: PhantomData<MeasurementType>,  
}  
  
pub struct DifferentialPressure;  
pub struct MassFlow;
```



# Type State Programming

State the possible **state transitions** with newtypes

```
/// Transition from Idle to Differential Pressure Sampling
pub type ToDifferentialPressureSampling<I2C, D> = Result<
    Sdp8xx<I2C, D, ContinuousSamplingState<DifferentialPressure>>,
    SdpError<I2C, I2C>,
>;

/// Transition from Continuous Sampling to Idle
pub type ToIdle<I2C, D> = Result<Sdp8xx<I2C, D, IdleState>, SdpError<I2C, I2C>>;

/// Transition from Idle to Sleep state
pub type ToSleep<I2C, D> = Result<Sdp8xx<I2C, D, SleepState>, SdpError<I2C, I2C>>;
```



# Type State Programming

Construction of initial state requires **ownership** of resources!

```
impl<I2C, D, E> Sdp8xx<I2C, D, IdleState>
  where /* snip */ {
    pub fn new(i2c: I2C, address: u8, delay: D) -> Self {
      Sdp8xx {
        i2c,
        address,
        delay,
        state: PhantomData::<IdleState>,
      }
    }
  }
}
```

No “&” or “&mut” anywhere!



# Type State Programming

Construction of initial state requires **ownership** of resources!

```
impl<I2C, D, E> Sdp8xx<I2C, D, IdleState>
  where /* snip */ {
    pub fn go_to_sleep(mut self) -> IdleToSleeping<I2C, D> {
      self.send_command(Command::EnterSleepMode)?;
      Ok(Sdp8xx {
        i2c: self.i2c,
        address: self.address,
        delay: self.delay,
        state: PhantomData::<SleepState>,
      })
    }
  }
```

No “&” or “&mut” anywhere!



# Type State Programming

Implement the operation modes as type-state transitions

```
impl<I2C, D, E> Sdp8xx<I2C, D, SleepState>
  where /* snip */ {
    pub fn wake_up(mut self) -> SleepingToIdle<I2C, D> {
      // snip...
      Ok(Sdp8xx {
        i2c: self.i2c,
        address: self.address,
        delay: self.delay,
        state: PhantomData::,
      })
    }
  }
}
```

No “&” or “&mut” anywhere!



# Type State Programming

Operations are available in their state only

The state transitions take **ownership** - no left-over invalid objects

All of this for free! No memory, binary, computation cost.

```
core::mem::size_of::<ContinuousSamplingState<MassFlow>>() == 0
```





# Deep Type State

```
let tx: TxMode<
  embedded_nrf24l01::NRF24L01<
    Infallible,
    gpio::gpioa::PA8<Output<PushPull>>,
    stm32f0xx_hal::gpio::gpioa::PA10<Output<PushPull>>,
    Spi<
      stm32::SPI1,
      gpio::gpioa::PA5<Alternate<gpio::AF0>>,
      gpio::gpioa::PA6<Alternate<gpio::AF0>>,
      gpio::gpioa::PA7<Alternate<gpio::AF0>>,
    >,
  >,
> = nrf24.tx().unwrap();
```



# Interlude: Embedded Ecosystem

Lots of sensors and actuators out there:

- Thermometers, LCD screens, Potentiometers, IMUs, Barometers, GPS sensors, Health Monitoring, Air Quality, ...

Arduino Libraries are nice (well, kinda...)

Problem: Drivers, Hardware Abstraction Layers (HALs), etc.

- not interoperable!



# Embedded HAL

Provides universal interfaces for common peripherals

- Analog to Digital Converters
- Communication Peripherals
- Timers (in/out)
- Watchdogs
- ...

Target-specific implementations for wildly varying platforms



# Embedded HAL

```
// get pins
let din = port0.p0_12.into_push_pull_output(Level::Low).degrade();
    more pins
let busy = port1.p1_09.into_floating_input();

// given pins, get SPI
let spi = Spim::new(p.SPI3, spi_pins, Frequency::K500, spim::MODE_0, 0);

// given SPI, get display
let mut epd4in2 = EPD4in2::new(spi, cs, busy, dc, rst, delay).unwrap();
```



# Embedded HAL

- Ownership + Move semantics: It is impossible to configure resources which are in use elsewhere
  - A peripheral **owns** its resources
  - A driver **owns** its peripherals
- Types + Traits + Generics:
  - Construct a peripheral with valid resources



# Embedded HAL Portability

Driver works on:

- Raspberry Pi
- Arduino
- STM32
- RISC-V Longan Nano
- ... any target for which embedded-hal implements i2c!



# Embedded HAL Mocking

Not just hardware targets - Mocking I2C for unit tests!

```
#[test]
fn trigger_mass_flow_read() {
    let bytes: [u8; 2] = Command::TriggerMassFlowRead.into();
    let data = vec![3, 4, 0x68, 6, 7, 0x4c, 0, 1, 0xb0];
    let expectations = [
        Transaction::write(0x10, bytes.into()),
        Transaction::read(0x10, data.clone()),
    ];
    let mock = I2cMock::new(&expectations);
    let mut sdp = Sdp8xx::new(mock, 0x10, DelayMock);
    let _data = sdp.trigger_mass_flow_sample().unwrap();
    sdp.release().done();
}
```

<https://crates.io/crates/embedded-hal-mock>



# Embedded HAL Bitbanging

Drop-in replace real I2C interfaces with software-based ones!

```
let scl = gpioa.pa0.into_open_drain_output(cs);  
let sda = gpioa.pa1.into_open_drain_output(cs);  
  
let timer = Timer::tim1(dp.TIM1, 200.khz(), &mut rcc);  
  
// Make bit-banged I2C with arbitrary pins  
let i2cbb = I2cBB::new(scl, sda, timer);  
  
let mut sdp8xx = Sdp8xx::new(i2cbb, 0x25, delay);
```





# Tool Shoutout: Tarpaulin!

SUPER EASY line coverage analysis.

```
May 16 23:31:18.162  INFO cargo tarpaulin::report: Coverage Results:
```

```
|| Tested/Total Lines:
```

```
|| src/command.rs: 12/14
```

```
|| src/lib.rs: 95/125
```

```
|| src/product_info.rs: 31/34
```

```
|| src/sample.rs: 28/31
```

```
|| src/test.rs: 162/163
```

```
||
```

```
89.37% coverage, 328/367 lines covered
```

<https://crates.io/crates/cargo-tarpaulin>



# Tarpaulin Example

<https://github.com/barafael/cd74hc4067/blob/main/coverage.pdf>

```
impl<P, E> CD74HC4067<P, E, EnabledState>
where
    P: OutputPin,
    P: OutputPin,
    P: OutputPin,
    P: OutputPin,
    E: OutputPin,
{
    /// Disable the mux display by pulling `pin_enable` high
    pub fn disable(mut self) -> Result<CD74HC4067<P, E, DisabledState>, Error<P, E>> {
        self.pin_enable.set_high().map_err(Error::EnablePinError)?;
        Ok(CD74HC4067 {
            pin_0: self.pin_0,
            pin_1: self.pin_1,
            pin_2: self.pin_2,
            pin_3: self.pin_3,
            pin_enable: self.pin_enable,
            state: PhantomData::<DisabledState>,
        })
    }
}
```

<https://crates.io/crates/cargo-tarpaulin>



# Crate Shoutout: Proptest!


Fuzz, but cleverly!

Example: <https://github.com/barafael/sdp8xx-rs/blob/main/src/test.rs#L224>

Idea: combine embedded-hal-mock with proptest

=> emulate misbehaving hardware / interference

<https://crates.io/crates/proptest>



# What does all this get me?

When I finally got the hardware, software worked first try!

For prototyping, used raspberry pi (actual target is `#![no_std]`)

When I needed to connect 3 I2C sensors to one MCU,  
I changed the I2C implementation to bitbanging.

Worked first try.



# Long-term sustainable projects

Some code bitrots quicker than I can write it

- Cross-toolchains, build environments, dependencies

High confidence in software

- Easy Unit tests, Mocking, Coverage
- Type-state programming
- Dependency management

Reduce SAAAD: “Spooky action at a distance”!



# Long-term sustainable projects

Solid building blocks to rely on

Pick up a project where I left it a while ago

Build abstractions on existing blocks

**=> long-term projects have a chance of success!**



Thanks :)