Persistence of Vision Display

What's POV?

Persistence of vision traditionally refers to the optical illusion that occurs when visual perception of an object does not cease for some time after the rays of light proceeding from it have ceased to enter the eye.

How to do POV

The Easy Part

- Spin some lights real fast.

The Hard Part

- Turn the lights on and off at the right time to draw “pixels.”

BOM

- STM32F405 MCU
- APA102C LEDs
- I2C Rotary Encoders
- OLED Display
- Stepper Motor & Driver
- Less interesting stuff (power supply, nuts, bolts, slipring, etc.)

Immediate Challenges

- The LEDs have to be turned on and off within about a millisecond of precision.
- We need to know the exact position of the display at each update.
- To enable more complex graphics, we need to track other state like the current time, current spin count, etc.
- System needs to be highly-responsive to multiple sources of input:
  - e.g. timer interrupts, control knobs, system clock, etc.

```rust
pub struct EffectState {
    /// Current revolution
    pub rev: u32,
    /// Current position in the revolution
    pub n: u32,
    /// Number of microseconds since start
    pub us: u64,
}
```

Some of the state that'll be available to the update function.

Embedded Programming Challenges

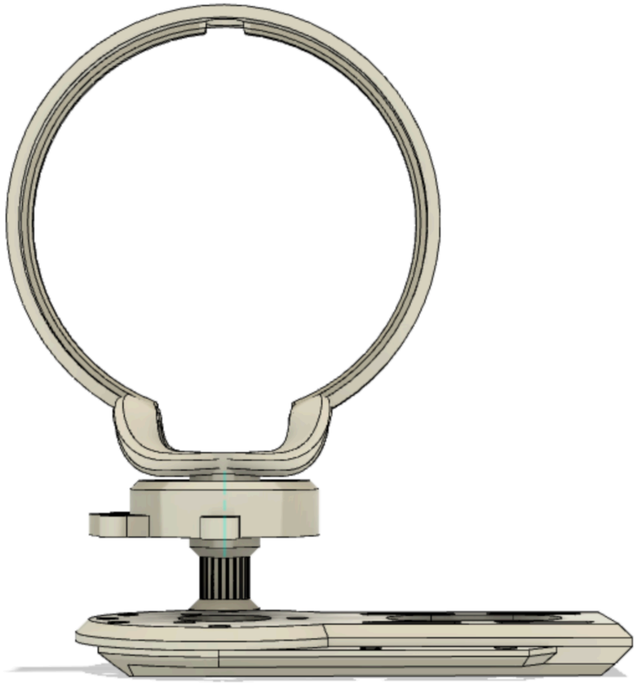
- There is no heap, so no dynamic memory allocation
- Very little available in terms of software abstractions for complex peripherals
  - e.g. using a timer requires reading the MCU reference manual and configuring registers
- Not a lot of memory to work with; all data is shared by reference
- No threads, concurrency only vaguely possible through IRQs

Embedded Programming Solutions

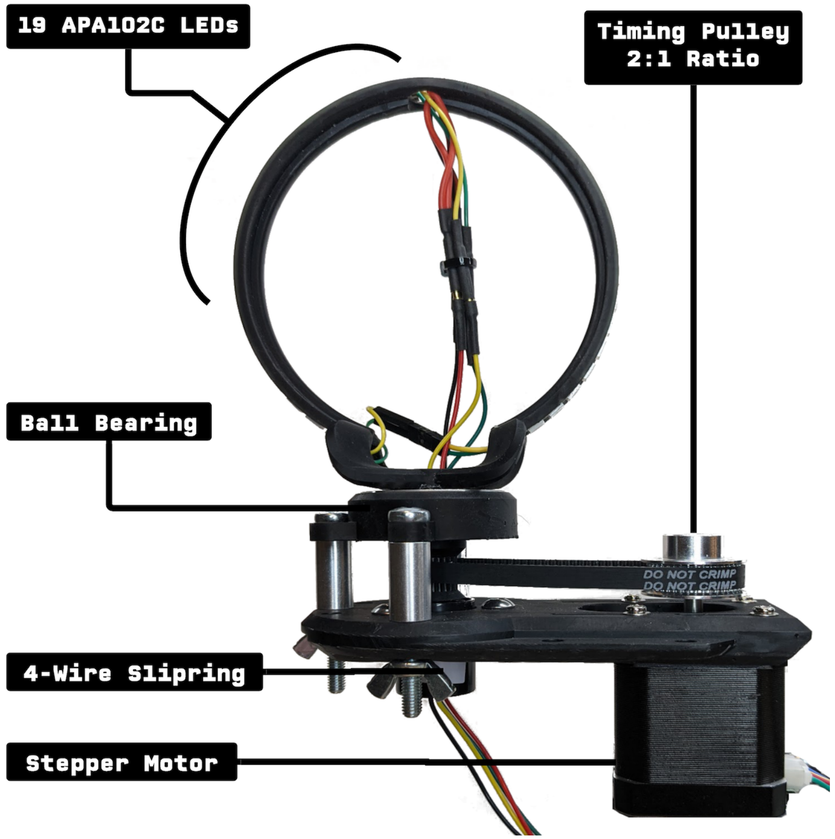
- We’re using Rust ❤️
  - 100% memory-safe; shared references and null-pointers are a non-issue
  - No memory overhead; rust has no need for a garbage-collector or a heap
  - It’s not C++ 🎉
- We’re using the RTIC framework (Real-Time Interrupt-driven Concurrency)
  - Provides mutex-like primitives for safe, ergonomic access to shared resources
  - Enables compile-time checking of potentially-dangerous resource modifications
  - Built-in interrupt management and prioritization

Starting with the easy part

Make something that can spin



(designed these components using Fusion 360)

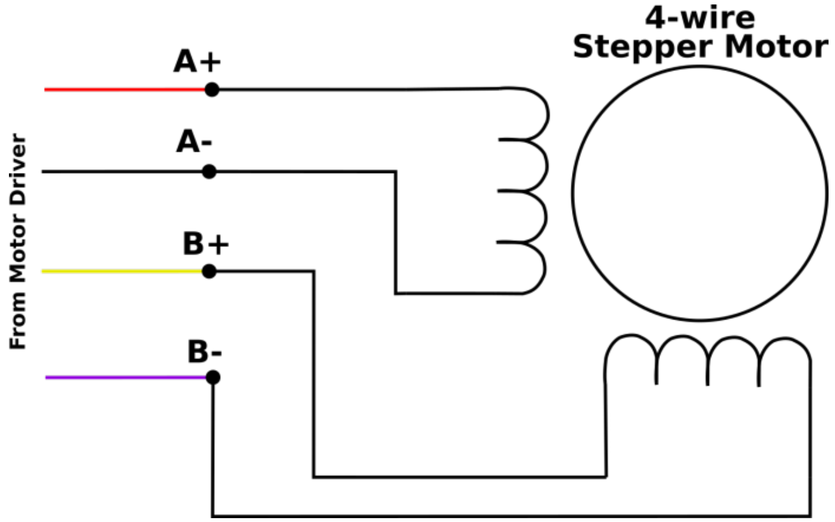


Component Design

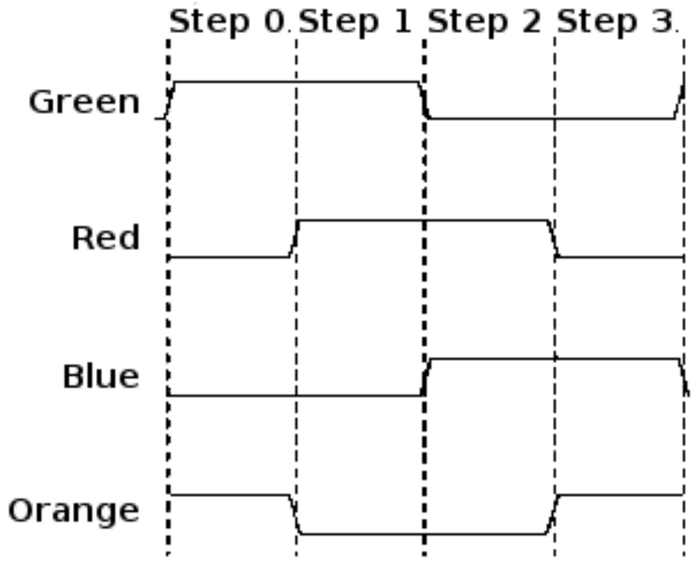
Printed Assembly

Stepper Motors

- Type of DC motor
- Allows controlled, precise motion
- A single rotation is divided into discrete “steps”
  - e.g. our N=200 stepper means that it moves in increments of 1.8˚ (360˚ / 200)
- Current and target position can be known at all times through software
- Signal waveforms (bottom right) are a pain to manage manually; you’ll almost always use a stepper driver



Stepper motor wiring



Signals to drive a single “step”

Driving the Stepper

We'll be using the stepper driver seen here. It expects 3 electrical inputs from us:

1. **Direction**
   - +5V motor will move forward, 0V motor will move in reverse
   - We don't actually care about direction for our display, so this is set to ground (0V)
2. **Enable**
   - +5V motor is active, 0V motor is idle (no power)
   - We do control this in software, otherwise motor is in a "brake" state even when not moving, which is a waste of power
3. **Pulse**
   - Each rising edge (signal goes from 0V to 5V) will drive the motor one step forward
   - ***Motor speed is therefore a function of the frequency of the digital signal sent to this input***

Driving the Stepper
Continued

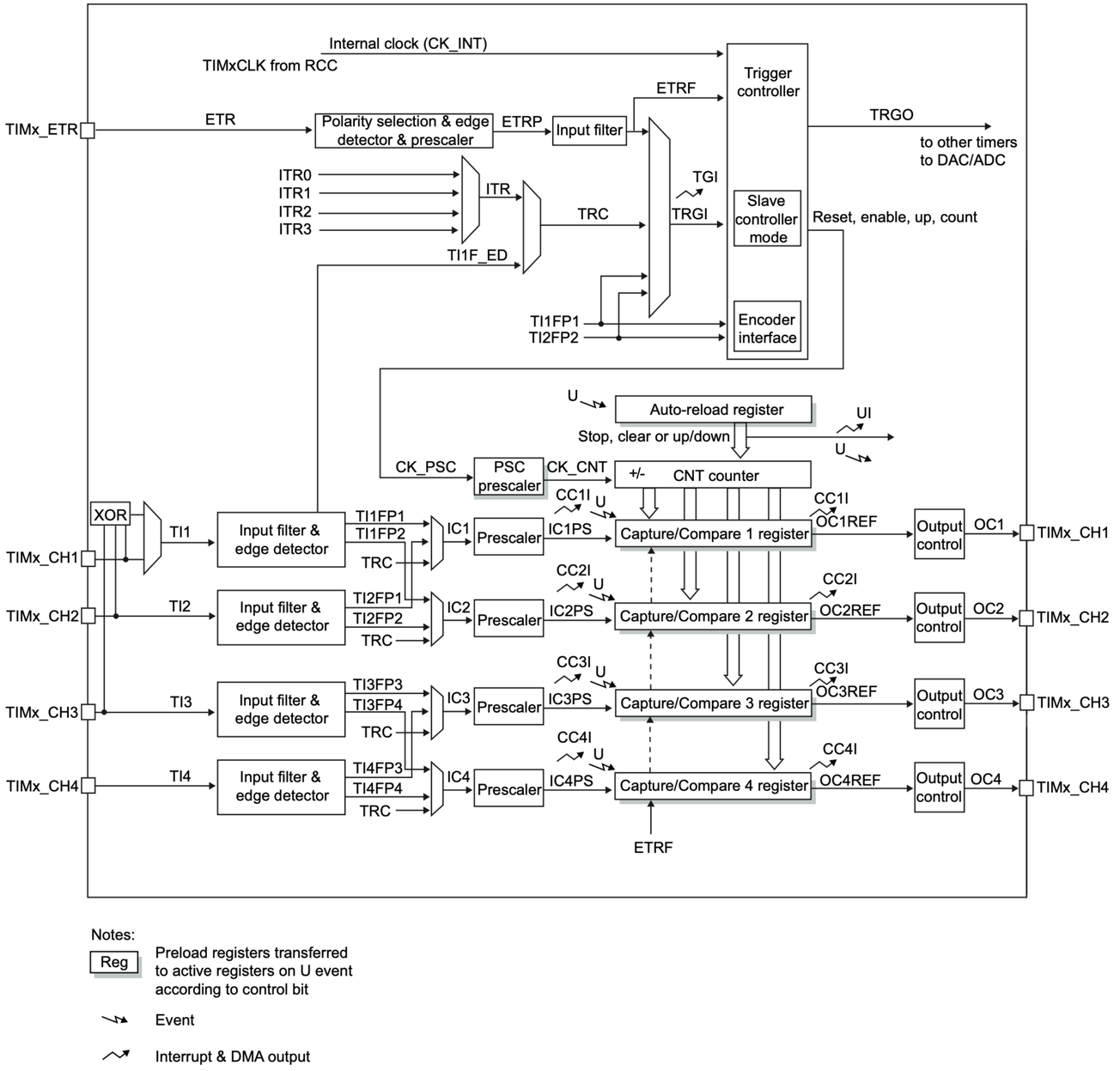
- The stepper driver we're using can be configured to use "micro steps"
- Allows for more precise motion in smaller increments
- For our display, we're using a micro step divisor of 32
- So, this means it will require 6,400 pulses (32 x 200) to move the motor one full rotation
- Our assembly uses timing pulleys with a 2:1 ratio, so one rotation of the POV display is equal to exactly 3,200 pulses (aka "ticks")

1 spin
= 3,200
ticks

Generating & Counting Ticks

Enter hardware timer unit peripherals

- Timers are common to all processors; the MCU we're using has 14 of them
- Extremely versatile
- Maintains a set of counter registers that can be read at any time by software
- Giant PITAs to work with; requires reading hundreds of pages of reference manual
- Our display's motor speed is a function of the pulse frequency, so we have to turn one of these timer units into a **frequency generator**



RUN

Frequency Generation

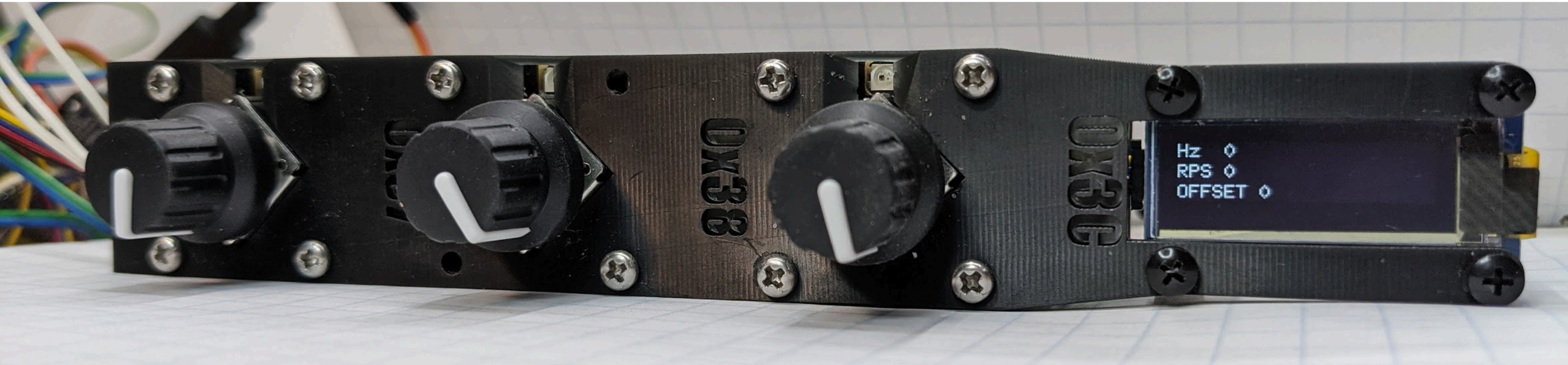
- The code at right is the constructor for a simple abstraction of our frequency generator
- It configures TIM4 to output its signal on pin D9, which we wire up to the stepper driver
- The code below performs the calculations and register writes to actually set the frequency
- **We'll use the timer's interrupt functionality to preempt a function that updates the display LEDs on each tick**

```rust
impl FreqGen<TIM4> {
    pub fn new(tim: TIM4, clocks: Clocks) -> Self {
        unsafe {
            // NOTE(unsafe) this ref used for atomic writes with no side effects
            let rcc = &(*RCC::ptr());
            // Enable and reset TIMx peripherals to a clean slate state
            rcc.apb1enr.modify(|_, w| w.tim4en().enabled());
            asm::dsb(); // Apparently a workaround for some erratum
            rcc.apb1rstr.modify(|_, w| w.tim4rst().set_bit());
            rcc.apb1rstr.modify(|_, w| w.tim4rst().clear_bit());
        }

        // Set pulse timer channel 3 (pin D9) as output in PWM mode 1
        tim.ccmr2_output(): &Reg<u32, _CCMR2_OUTPUT>
            .modify(|_, w| w.oc3pe().enabled().oc3m().pwm_mode1());

        // Enable pulse timer channel 3
        tim.ccer.modify(|_, w| w.cc3e().set_bit());

        // Output a trigger signal on each pulse timer update event
        // This will be used as in input by the spin counter
        tim.cr2.modify(|_, w| w.mms().update());

        // Configure the timer output form
        tim.cr1.modify(|_, w| {
            w.arpe(): ARPE_W
                .enabled(): &mut W<u32, Reg<u32, _CR1>>
                .cms(): CMS_W
                .bits(0b00): &mut W<u32, Reg<u32, _CR1>> // center aligned
                .dir(): DIR_W
                .clear_bit(): &mut W<u32, Reg<u32, _CR1>> // count up
                .opm(): OPM_W
                .clear_bit() // one pulse disabled
```

```rust
pub fn set_freq<F: Into<Hertz>>(&mut self, freq: F) {
    let clk = self.clocks.pclk1().0 * if self.clocks.ppre1() == 1 { 1 } else { 2 };
    let freq: u32 = freq.into().0;
    let ticks = clk / if freq != 0 { freq } else { 1 };
    let psc = ((ticks - 1) / (1 << 16)) as u16;
    self.tim.psc.modify(|_, w| w.psc().bits(psc));
    let arr = (ticks / (psc + 1) as u32) as u16;
    self.tim.arr.modify(|_, w| w.arr().bits(arr));
    self.tim.ccr3.modify(|_, w| w.ccr().bits(arr / 2));
}
```

Frequency Control

- A stepper motor, especially one intended to spin fast, has to be “brought up” to speed
- For that, human control is easiest, for which we’re using some I2C-driven rotary encoders and an OLED display held together by 3D-printed components
- Uses IRQs to prevent unnecessary reads; knob can be pressed for emergency stops
- This is a separate project that spun out of the POV thing



The APA102 LED

And why we're using them

- Driven by SPI
- Dead-simple data protocol
- Each LED contains a tiny little processor and can be individually updated
- Spec'd at a 1.2MHz clock speed, in practice they seem to run fine at well over 10MHz

Talking to the LEDs

APA102s' Protocol

- Data packets start with a 4 byte header frame

header	0x00	0x00	0x00	0x00

- Each subsequent frame is 4 bytes per LED in the strand. First byte is always 0xFF, and the following 3 bytes correspond to the blue, green, and red value of the LED, respectively.

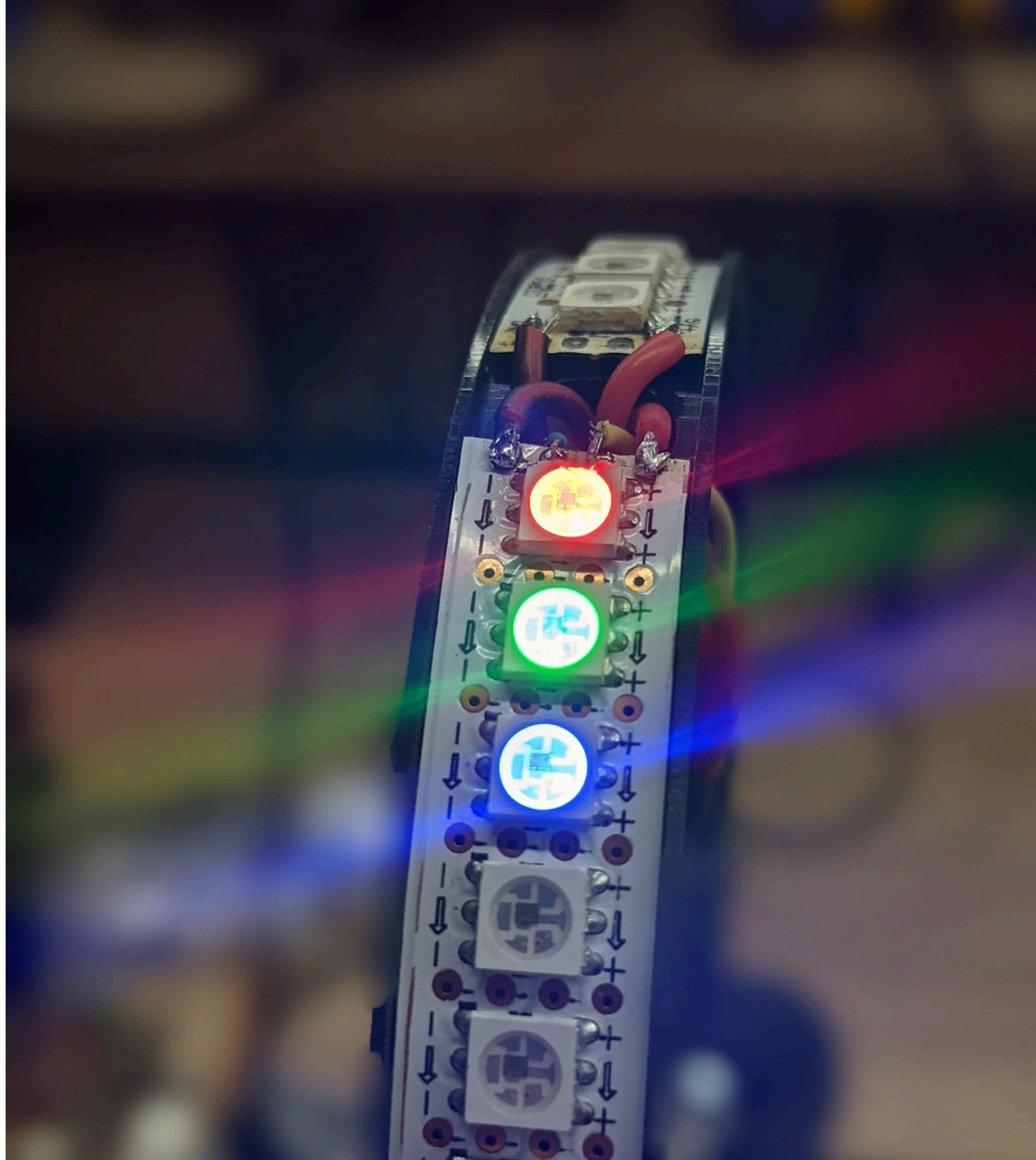
led_n	0xFF	blue	green	red

- Once we've sent our packet o'four-byte frames down the SPI data line, the first LED in the strand will eat the first frame and update its color to the values specified. Then it'll pass the rest of the data down to the next LED in the strand which will do the same, etc.

Talking to the LEDs
Continued

- So, if we've got 3 LEDs and send this packet of frames, we get the picture at the right.

header	0x00	0x00	0x00	0x00
led_0	0xFF	0x00	0x00	0xFF
led_1	0xFF	0x00	0xFF	0x00
led_2	0xFF	0xFF	0x00	0x00



Spatial Modeling

Discretizing motion

- Display can be thought of as a 2D array of pixels projected onto a sphere
- Each update of the LEDs creates a vertical strip of pixels, this will be the **m-axis**, **where M = 19** because we have 19 LEDs in the strand
- The (curvy) horizontal axis will be the **n-axis**, where N = 3,200, the number of discrete steps of the stepper to complete one spin
- But, our eyes can’t process 3,200 updates of the LEDS per revolution (all of the colors blend together to create white), so we’ll only update the LEDs every 40th tick, leaving us with an x resolution of **N = 80**

Data Modeling

- A pixel is a 3-byte array [R, G, B]
- There are 19 LEDs in the strand, so we store a “column” as an array of 19 pixels
- We’ve split the rotational n-axis into 80 columns, so we’re left with an 80x19 array of 3-byte values
- We have to know these values in advance because remember, we can’t dynamically allocate memory onto the heap
- Const generics are a neat feature of rust that, among other things, allow stack memory allocation to be performed at compile-time as part of the type system

```
type ColorBuffer<
    const M: usize,
    const N: usize
> = [[RGB8; M]; N];
```

For us, M = 19 and N = 80

Abstracting Simple Effects

- We need a generic abstraction to describe animations and effects
- Rust has a thing called **traits** that allow defining behavior without the implementation
  - They're *kinda* like interfaces in Java, except not really, but a little bit
- We'll define an Effect trait that requires implementation of a "next" function.
  - "next" will be called for each n-tick of the stepper and should return an array of pixels with length M
  - The pixel array will be sent directly to the LED strip
  - "next" will be provided with some information about the current system state, for convenience

```rust
pub trait Effect<const M: usize> {
    fn next(
        &mut self,
        s1: &EffectState,
        s0: &EffectState
    ) -> [RGB8; M];
}
```

Effects with Buffers

- We'll define another trait for effects that maintain a buffer in memory and therefore need to perform some sort of initialization step
- The BufferedEffect trait requires implementation of a "create" function, where implementors can perform buffer allocation and setup
- Memory mutation is intrinsically fallible, so the "create" function returns a "Result"
- The trait specifies that in order to implement BufferedEffect, the Effect trait must also be implemented

```rust
pub trait BufferedEffect<const M: usize>
where
    Self: Effect<M> + Sized,
{
    type Error;
    fn create() -> Result<Self, Self::Error>;
}
```

Drawing Graphics

Using the embedded-graphics library

- The embedded-graphics library provides mechanisms for simple drawing of things like geometry and text
  - Think of it as like the canvas API in a web browser
  - Specifically designed to be used with embedded displays
- We just have to provide the library with a function (specified by the library's DrawTarget trait) that specifies how to store pixels in memory
- The library will give us x and y coordinates and a color, and we dictate how to store that in an effect buffer (not pictured here)
- Implementation of DrawTarget at right

```rust
impl<const M: usize, const N: usize> DrawTarget for EffectBuffer<M, N> {
    type Color = Rgb888;
    type Error = EffectError;

    fn draw_iter<I>(&mut self, pixels: I) -> Result<(), Self::Error>
    where
        I: IntoIterator<Item = Pixel<Self::Color>>,
    {
        let bb = self.bounding_box();

        pixels: I
            .into_iter(): <I as IntoIterator>::IntoIter
            .filter(|Pixel(pos, _)| bb.contains(*pos)): impl Iterator<Item = Pi
            .for_each(|Pixel(pos, color)| {
                self.set(
                    pos.x as usize,
                    pos.y as usize,
                    (color.r(), color.g(), color.b()),
                )
            });

        Ok(())
    }
}
```

GlobeEffect

Drawing Stuff

- The struct at right defines a GlobeEffect
  - Structs are *kinda* like classes in Java, except not really, but a little bit
- Our GlobeEffect has a single property, which is an EffectBuffer that'll hold its pixel values

```
pub struct GlobeEffect<
    const M: usize,
    const N: usize
>(pub EffectBuffer<M, N>);
```

Implement BufferedEffect

The initialization step

- Since our effect will maintain memory in a buffer, we need to implement our BufferedEffect trait
- Here's where we get to use the API provided by the embedded-graphics library to draw the horizontal and vertical lines of our globe

```rust
impl<const M: usize, const N: usize> BufferedEffect<M> for GlobeEffect<M, N> {
    type Error = EffectError;

    fn create() -> Result<Self, Self::Error> {
        let mut buffer = EffectBuffer::<M, N>::new();

        let green = PrimitiveStyleBuilder::new(): PrimitiveStyleBuilder<Rgb888>
            .stroke_width(1): PrimitiveStyleBuilder<Rgb888>
            .stroke_color(Rgb888::GREEN): PrimitiveStyleBuilder<Rgb888>
            .build();

        // Draw a vertical line at every 10th n
        (0..N): Range<usize>
            .into_iter(): Range<usize>
            .filter(|n| n % 10 == 0): impl Iterator<Item = usize>
            .try_for_each(|n| {
                Line::new(Point::new(n as i32, 0), Point::new(n as i32, M as i32)): Line
                    .into_styled(green): Styled<Line, PrimitiveStyle<Rgb888>>
                    .draw(&mut buffer)
            }): Result<(), EffectError>
            .map_err(|_| EffectError::Draw)?;

        // Draw a horizontal line at every 7th m
        (0..M): Range<usize>
            .into_iter(): Range<usize>
            .filter(|m| m % 7 == 0): impl Iterator<Item = usize>
            .try_for_each(|m| {
                Line::new(Point::new(0, m as i32), Point::new(N as i32, m as i32)): Line
                    .into_styled(green): Styled<Line, PrimitiveStyle<Rgb888>>
                    .draw(&mut buffer)
            }): Result<(), EffectError>
            .map_err(|_| EffectError::Draw)?;

        Ok(Self(buffer))
    }
}
```

Implement Effect

Update Step

- Now we just have to tell the system what to do with our effect on each update, by implementing our Effect trait
- First we check the current state and previous state; if the display is on a new revolution, we rotate our globe by (-1, 0)
- Then we return the current vertical strip of pixels using the current tick (n value) as an index to access the buffer

```rust
impl<const M: usize, const N: usize> Effect<M> for GlobeEffect<M, N> {
    fn next(
        &mut self,
        state: &EffectState,
        prev_state: &EffectState
    ) -> [RGB8; M] {
        if state.rev != prev_state.rev {
            self.0.translate_mut(Point::new(-1, 0));
        }

        self.0[state.tick as usize]
    }
}
```

DEMO