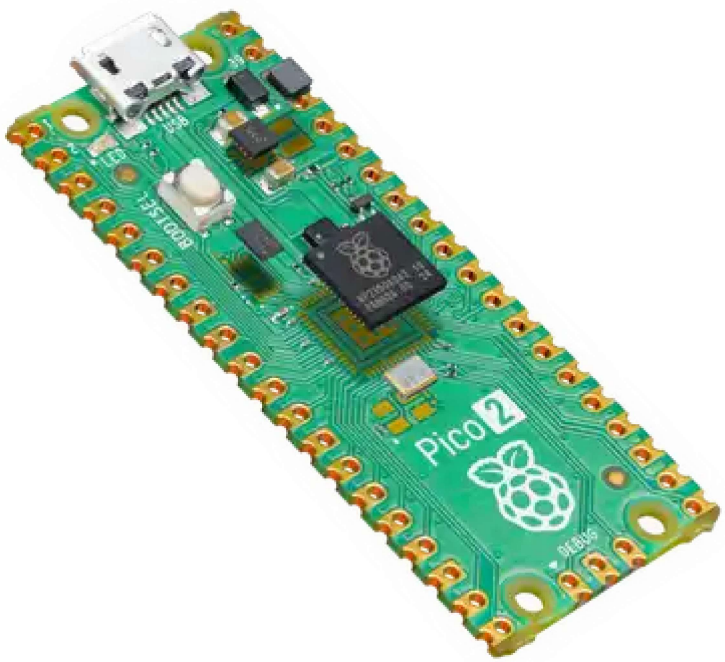
Pico Pico - Intro

In this book, we use the Raspberry Pi Pico 2 and program it in Rust to explore various exciting projects. You'll work on exercises like dimming an LED, controlling a servo motor, measuring distance with a sonic sensor, displaying the Ferris (🦀) image on an OLED display, using RFID Reader, playing songs on a buzzer, turning on an LED when the room light is off, measuring temperature, and much more.

Meet the hardware - Pico 2

We will be using the Raspberry Pi Pico 2 (RP2350 chip). It features the new RP2350 chip with dual-core flexibility;offering Arm Cortex-M33 cores and optional RISC-V Hazard3 cores. You can operate it as the standard ARM core or switch to the RISC-V architecture. You find more details from the official website.



Datasheets

For detailed technical information, specifications, and guidelines, refer to the official datasheets:

- Pico 2 Datasheet
- RP2350 chip Datasheet

License

The Pico Pico book(this project) is distributed under the following licenses:

- The code samples and free-standing Cargo projects contained within this book are licensed under the terms of both the MIT License and the Apache License v2.0.
- The written prose contained within this book is licensed under the terms of the Creative Commons CC-BY-SA v4.0 license.

Support this project

You can support this book by starring this project on GitHub or sharing this book with others 😊

Disclaimer:

The experiments and projects shared in this book have worked for me, but results may vary. I'm not responsible for any issues or damage that may occur while you're experimenting. Please proceed with caution and take necessary safety precautions.

Setup

Picotool

picotool is a tool for working with RP2040/RP2350 binaries, and interacting with RP2040/RP2350 devices when they are in BOOTSEL mode.

Picotool Repo

> **Pre-built binaries**
>
> Alternatively, you can download the pre-built binaries of the SDK tools from here, which is a simpler option than following these steps.

Here's a quick summary of the steps I followed:

```
# Install dependencies
sudo apt install build-essential pkg-config libusb-1.0-0-dev cmake

mkdir embedded && cd embedded

# Clone the Pico SDK
git clone https://github.com/raspberrypi/pico-sdk
cd pico-sdk
git submodule update --init lib/mbedtls
cd ../

# Set the environment variable for the Pico SDK
PICO_SDK_PATH=/MY_PATH/embedded/pico-sdk

# Clone the Picotool repository
git clone https://github.com/raspberrypi/picotool
```

Build and install Picotool

```
cd picotool
mkdir build && cd build
# cmake ../
cmake -DPICO_SDK_PATH=/var/ws/embedded/pico-sdk/ ../
make -j8
sudo make install
```

On Linux you can add udev rules in order to run picotool without sudo:

```
cd ../
# In picotool cloned directory
sudo cp udev/99-picotool.rules /etc/udev/rules.d/
```

Rust Targets

To build and deploy Rust code for the RP2350 chip, you'll need to add the appropriate targets:

```
rustup target add thumbv8m.main-none-eabihf
rustup target add riscv32imac-unknown-none-elf
```

Quick Start

Before diving into the theory and concepts of how everything works, let's jump straight into action. Use this simple code to turn on the onboard LED of the Pico2.

Blink LED with Embassy Framework

Embassy framework is a robust framework for developing asynchronous embedded applications in Rust.

This example code is taken from embassy repo (It also has additional examples): "https://github.com/embassy-rs/embassy/tree/main/examples/rp/src/bin"

It creates a blinking effect by toggling the pin's output state between high and low.

The code snippet

This is only part of the code. You'll need to set up some initial configurations and import the necessary crates.

```rust
#[embassy_executor::main]
async fn main(_spawner: Spawner) {
    let p = embassy_rp::init(Default::default());

    // The onboard LED is actually connected to pin 25
    let mut led = Output::new(p.PIN_25, Level::Low);

    loop {
        led.set_high(); // <- Turn on the LED
        Timer::after_millis(500).await;

        led.set_low(); // <- Turn off the LED
        Timer::after_millis(500).await;
    }
}
```

Clone the Quick start project

You can clone the quick start project I created and navigate to the project folder and run it.

```
git clone https://github.com/ImplFerris/pico2-quick
cd pico2-quick
```

How to Run?

You refer the "Running The Program" section

Project Template with `cargo-generate`

"cargo-generate is a developer tool to help you get up and running quickly with a new Rust project by leveraging a pre-existing git repository as a template."

Read more about here.

Prerequisites

Before starting, ensure you have the following tools installed:

- Rust
- cargo-generate for generating the project template.

You can install `cargo-generate` using the following command:

```
cargo install cargo-generate
```

Step 1: Generate the Project

Run the following command to generate the project from the template:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

This will prompt you to answer a few questions: Project name: Name your project. HAL choice: You can choose between embassy or rp-hal.

Step 2: Default LED Blink Example

By default, the project will be generated with a simple LED blink example. The code structure may look like this:

`src/main.rs`
: Contains the default blink logic.

`Cargo.toml`
: Includes dependencies for the selected HAL.

Step 3: Choose Your HAL and Modify Code

Once the project is generated, you can decide to keep the default LED blink code or remove it and replace it with your own code based on the HAL you selected.

Removing Unwanted Code

You can remove the blink logic from src/main.rs and replace it with your own code. Modify the Cargo.toml dependencies and project structure as needed for your project.

Running the program

Before we explore further examples, let's cover the general steps to build and run any program on the Raspberry Pi Pico 2. The Pico 2 contains both ARM Cortex-M33 and Hazard3 RISC-V processors, and we'll provide instructions for both architectures.

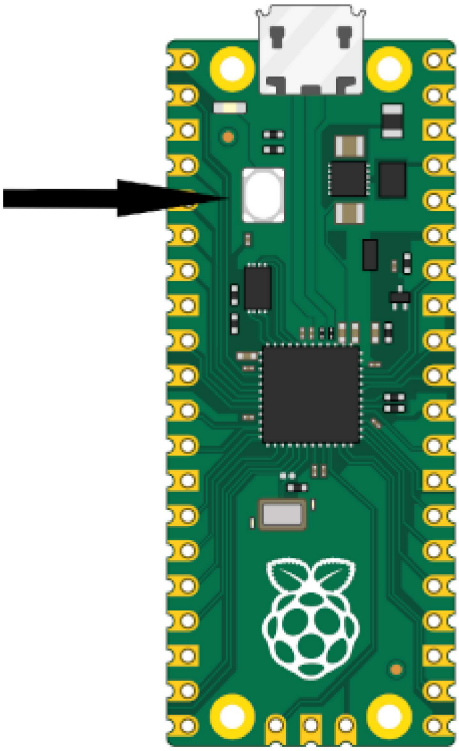
Note: These commands should be run from your project folder. This is included here as a general step to avoid repetition. If you haven't created a project yet, begin with the Quick Start or Blink LED section.

Build and Run for ARM

Use this command to build and run programs on the Raspberry Pi Pico 2 in ARM mode, utilizing the Cortex-M33 processors.

```
# build the program
cargo build --target=thumbv8m.main-none-eabihf
```

To Flash your application onto the Pico 2, press and hold the BOOTSEL button. While holding it, connect the Pico 2 to your computer using a micro USB cable. You can release the button once the USB is plugged in.



```
# Run the program
cargo run --target=thumbv8m.main-none-eabihf
```

Note: The example codes include a runner configuration in the `.cargo/config.toml` file, defined as: `runner = "picotool load -u -v -x -t elf"`. This means that when you execute `cargo run`, it actually invokes the `picotool` with the `load` subcommand to flash the program.

Build and Run for RISC-V

Use this command to build and run programs on the Raspberry Pi Pico 2 n RISC-V mode, utilizing the Hazard3 processors.

```
# build the program
cargo build --target=riscv32imac-unknown-none-elf
```

Follow the same BOOTSEL steps as described above.

```
# Run the program
cargo run --target=riscv32imac-unknown-none-elf
```

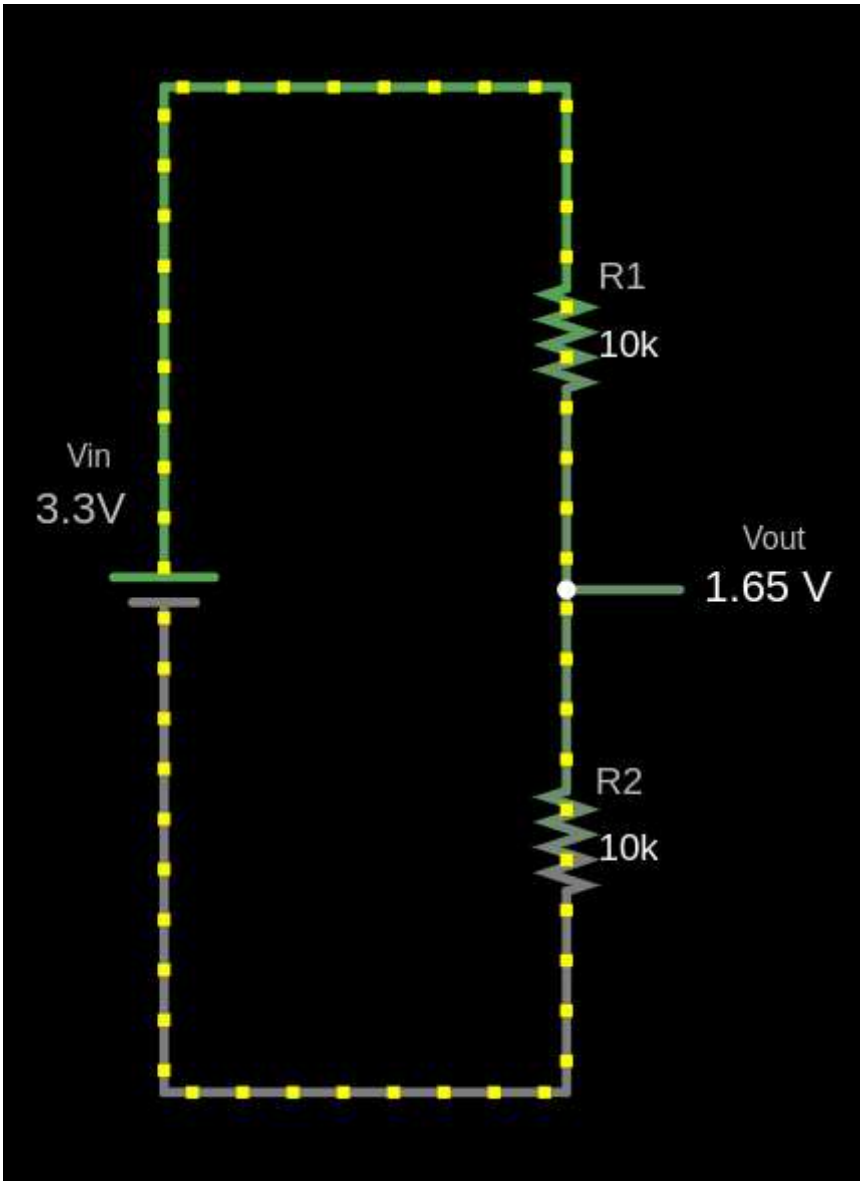
Core concepts

This section primarily covers theoretical concepts. Feel free to skip it and dive into the exercises instead. As you work through the exercises related to each concept, we'll link back to the theory. This approach may help you understand the material better by showing you exactly where and why these concepts are needed in practice.

Voltage Divider

A **voltage divider** is a simple circuit that reduces an input voltage V_{in} to a lower output voltage V_{out} using two series resistors. The resistor connected to the input voltage V_{in} is called R_1, and the other resistor is R_2. The output voltage V_{out} is taken from the junction between R_1 and R_2, producing a fraction of V_{in}.

Circuit



The output voltage (V_{out}) is calculated using this formula:

$$V_{out} = V_{in} \times \frac{R_2}{R_1 + R_2}$$

Example Calculation for V_{out}

Given:

- $V_{in} = 3.3V$
- $R_1 = 10k\Omega$
- $R_2 = 10k\Omega$

Substitute the values:

$$V_{out} = 3.3V \times \frac{10k\Omega}{10k\Omega + 10k\Omega} = 3.3V \times \frac{10}{20} = 3.3V \times 0.5 = 1.65V$$

The output voltage V_{out} is 1.65V.

```
fn main() {
    // You can edit the code
    // You can modify values and run the code
    let vin: f64 = 3.3;
    let r1: f64 = 10000.0;
    let r2: f64 = 10000.0;

    let vout = vin * (r2 / (r1 + r2));

    println!("The output voltage Vout is: {:.2} V", vout);
}
```

Use cases

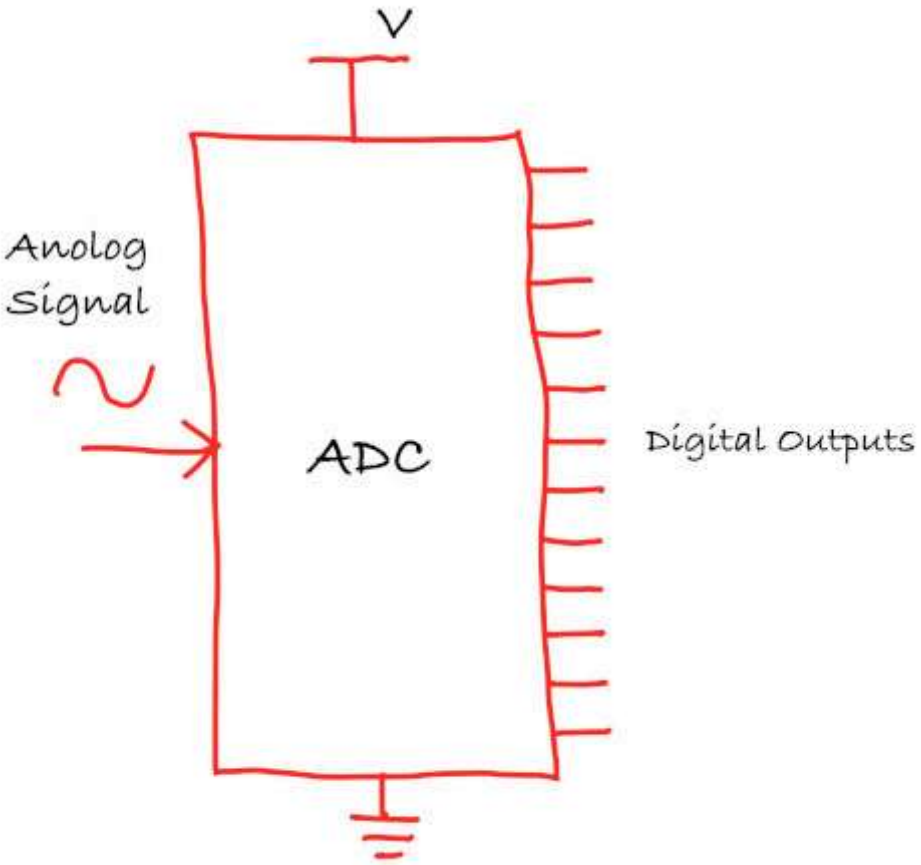
Voltage dividers are used in applications like potentiometers, where the resistance changes as the knob is rotated, adjusting the output voltage. They are also used to measure resistive sensors such as light sensors and thermistors, where a known voltage is applied, and the microcontroller reads the voltage at the center node to determine sensor values like temperature.

Simulator

I used the website https://www.falstad.com/circuit/e-voltdivide.html to create this diagram. It's a great tool for drawing circuits. You can download the file I created,
`voltage-divider.circuitjs.txt`
, and import it to experiment with the circuit.

ADC (Analog to Digital Converter)

An Analog-to-Digital Converter (ADC) is a device used to convert analog signals (continuous signals like sound, light, or temperature) into digital signals (discrete values, typically represented as 1s and 0s). This conversion is necessary for digital systems like microcontrollers (e.g., Raspberry Pi, Arduino) to interact with the real world. For example, sensors that measure temperature or sound produce analog signals, which need to be converted into digital format for processing by digital devices.



ADC Resolution

The resolution of an ADC refers to how precisely the ADC can measure an analog signal. It is expressed in bits, and the higher the resolution, the more precise the measurements.

- 8-bit ADC produces digital values between 0 and 255.
- 10-bit ADC produces digital values between 0 and 1023.
- 12-bit ADC produces digital values between 0 and 4095.

The resolution of the ADC can be expressed as the following formula:

$$\text{Resolution} = \frac{\text{Vref}}{2^{\text{bits}} - 1}$$

Pico

Based on the Pico datasheet, Raspberry Pi Pico has 12-bit 500ksps Analogue to Digital Converter (ADC). So, it provides values ranging from 0 to 4095 (4096 possible values)

$$\text{Resolution} = \frac{3.3V}{2^{12} - 1} = \frac{3.3V}{4095} \approx 0.000805\text{V} \approx 0.8\text{mV}$$

Pins

The Raspberry Pi Pico has **four accessible ADC channels** on the following GPIOs:

GPIO Pin	ADC Channel	Function
GPIO26	ADC0	Can be used to read voltage from peripherals.
GPIO27	ADC1	Can be used to read voltage from peripherals.
GPIO28	ADC2	Can be used to read voltage from peripherals.
GPIO29	ADC3	Measures the VSYS supply voltage on the board.

In pico, ADC operates with a reference voltage set by the supply voltage, which can be measured on pin 35 (ADC_VREF).

ADC Value and LDR Resistance in a Voltage Divider

In a voltage divider with an LDR and a fixed resistor, the output voltage V_{out} is given by:

$$V_{\text{out}} = V_{\text{in}} \times \frac{R_{\text{LDR}}}{R_{\text{LDR}} + R_{\text{fixed}}}$$

It is same formula as explained in the previous chapter, just replaced the R_2 with R_{LDR} and R_1 with R_{fixed}

- **Bright light** (low LDR resistance): V_{out} decreases, resulting in a lower ADC value.
- **Dim light** (high LDR resistance): V_{out} increases, leading to a higher ADC value.

Example ADC value calculation:

Bright light:

Let's say the Resistence value of LDR is $1k\Omega$ in the bright light (and we have $10k\Omega$ fixed resistor).

$$V_{\text{out}} = 3.3V \times \frac{1k\Omega}{1k\Omega + 10k\Omega} \approx 0.3V$$

The ADC value is calculated as:

$$\text{ADC value} = \left(\frac{V_{\text{out}}}{V_{\text{ref}}}\right) \times (2^{12} - 1) \approx \left(\frac{0.3}{3.3}\right) \times 4095 \approx 372$$

Darkness:

Let's say the Resistence value of LDR is $140k\Omega$ in very low light.

$$V_{\text{out}} = 3.3V \times \frac{140k\Omega}{140k\Omega + 10k\Omega} \approx 3.08V$$

The ADC value is calculated as:

$$\text{ADC value} = \left(\frac{V_{\text{out}}}{V_{\text{ref}}}\right) \times (2^{12} - 1) \approx \left(\frac{3.08}{3.3}\right) \times 4095 = 3822$$

Converting ADC value back to voltage:

Now, if we want to convert the ADC value back to the input voltage, we can multiply the ADC value by the resolution (0.8mV).

For example, let's take an ADC value of 3822:

$$\text{Voltage} = 3822 \times 0.8mV = 3057.6mV \approx 3.06V$$

Reference

- What is Analog to Digital Converter & Its Working

Pulse Width Modulation (PWM)

In this section, we will explore what is PWM and why we need it.

Digital vs Analog

In a digital circuit, signals are either high (such as 5V or 3.3V) or low (0V), with no in-between values. These two distinct states make digital signals ideal for computers and digital devices, as they're easy to store, read, and transmit without losing accuracy.

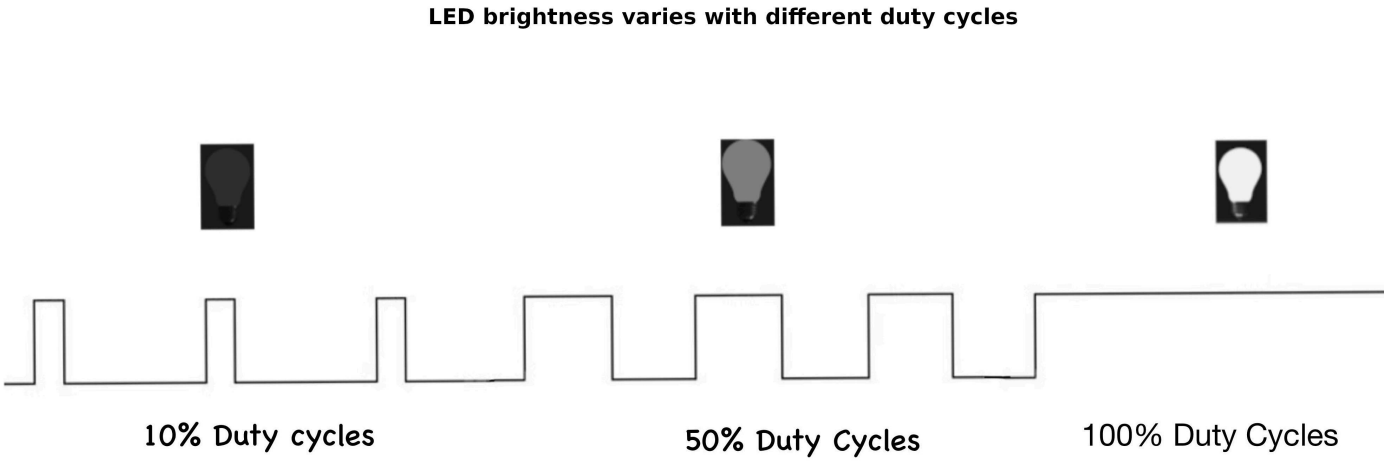
Analog signals, however, can vary continuously within a range, allowing for any value between a High and Low voltage. This smooth variation is valuable for applications requiring fine control, such as adjusting audio volume or light brightness.

Devices like servo motors and LEDs(for dimming effect) often need gradual, precise control over voltage, which analog signals provide through their continuous range.

Microcontrollers use PWM to bridge this gap.

What is PWM?

PWM stands for **Pulse Width Modulation**, creates an analog-like signal by rapidly pulsing a digital signal on and off. The average output voltage, controlled by adjusting the pulse's high duration or "duty cycle," can simulate a continuous analog level.



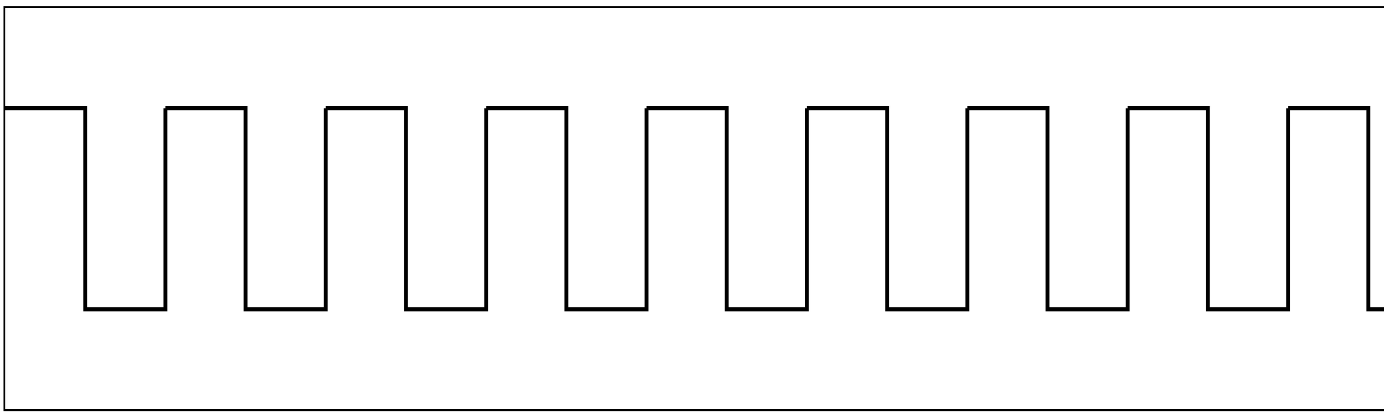
Duty Cycle

The **duty cycle** is the percentage of time a signal is "on" during one complete cycle.

For example:

- 100% duty cycle means the signal is always on.
- 50% duty cycle means the signal is on half the time and off half the time.
- 0% duty cycle means the signal is always off.

Here is the interactive simulation. Use the sliders to adjust the duty cycle and frequency, and watch how the pulse width and LED brightness change. The upper part of the square wave represents when the signal is high (on). The lower part represents when the signal is low(off)



Duty Cycle (%): 50%

Frequency (Hz):

If you change the duty cycle from "low to high" and "high to low" in the simulation, you should notice the LED kind of giving a dimming effect.

Period and Frequency

Period is the total time for one on-off cycle to complete.

The frequency of a PWM signal is the number of cycles it completes in one second, measured in Hertz (Hz). Frequency is the inverse of the period. So, a higher frequency means a shorter period, resulting in faster switching between HIGH and LOW states.

$$\text{Frequency (Hz)} = \frac{1}{\text{Period (s)}}$$

So if the period is 1 second, then the frequency will be 1HZ.

$$1\text{Hz} = \frac{1 \text{ cycle}}{1 \text{ second}} = \frac{1}{1 \text{ s}}$$

For example, if the period is 20ms(0.02s), the frequency will be 50Hz.

$$\text{Frequency} = \frac{1}{20 \text{ ms}} = \frac{1}{0.02 \text{ s}} = 50 \text{ Hz}$$

Calculating Cycle count from Frequency per second

The Formula to calculate cycle count:

$$\text{Cycle Count} = \text{Frequency (Hz)} \times \text{Total Time (seconds)}$$

If a PWM signal has a frequency of 50Hz, it means it completes 50 cycles in one second.

PWM in Depth

Timer Operation

The timer plays a key role in the PWM generator. It counts from zero to a specified maximum value (stored in a register), then resets and starts the cycle over. This counting process determines the duration of one complete cycle, called the period.

Compare Value

The timer's hardware compares its current count with a compare value (stored in a register). When the count is less than the compare value, the signal stays high; when the count exceeds the compare value, the signal goes low.

PWM Resolution

In PWM (Pulse Width Modulation), resolution refers to how precisely the duty cycle can be controlled. This is determined by the number of bits used in the PWM's compare register.

The timer counts from 0 to a maximum value based on the resolution. The higher the resolution, the more finely the duty cycle can be adjusted.

For a system with **n** bits of resolution, the timer can count from 0 to $2^n - 1$, which gives 2^n possible levels for the duty cycle.

For example:

- 8-bit resolution allows the timer to count from 0 to 255, providing 256 possible duty cycle levels.

- 10-bit resolution allows the timer to count from 0 to 1023, providing 1024 possible duty cycle levels.

Higher resolution gives more precise control over the duty cycle but also means the timer must count more values within the same period, which could lower the frequency or require more processing power. Essentially, the resolution defines how many distinct duty cycle values can be set, with more bits offering finer adjustments.

Simulation

You can modify the PWM resolution bits and duty cycle in this simulation. Adjusting the PWM resolution bits increases the maximum count but remains within the time period (it does not affect the duty cycle). Changing the duty cycle adjusts the on and off states accordingly, but it also stays within the period.

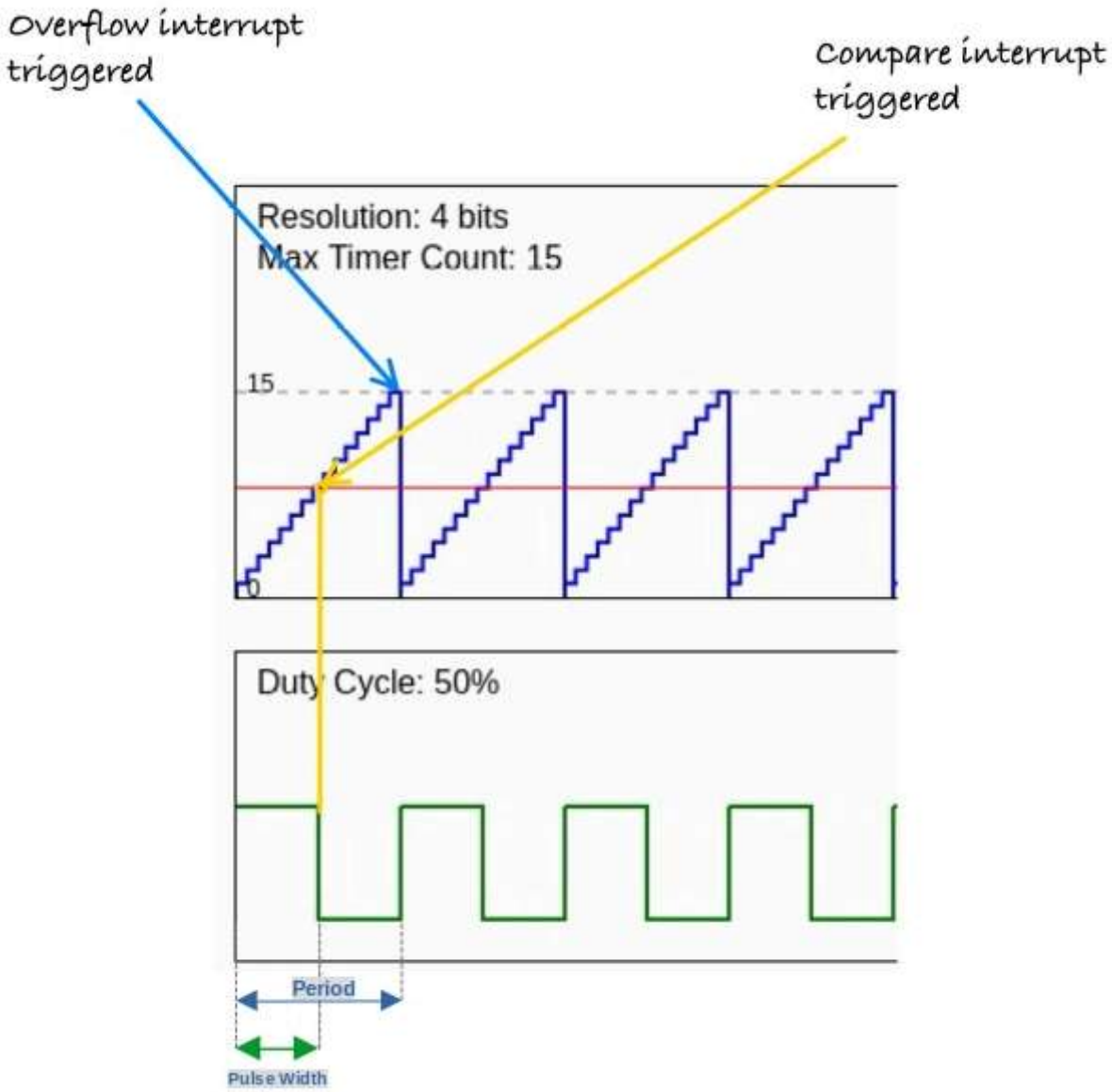
PWM Resolution (Bits): 8

Duty Cycle (%): 75

Relationship Between Duty Cycle, Frequency, and Resolution

This diagram illustrates the relationship between duty cycle, frequency, period, pulse width, and resolution. While it may seem a bit complex at first glance, breaking it down helps to clarify these concepts.

In this example, the timer resolution is 4 bits, meaning the timer counts from 0 to 15. When the timer reaches its maximum value, an overflow interrupt is triggered (indicated by the blue arrow), and the counter resets to 0. The time it takes for the timer to count from 0 to its maximum value is called as "period".



The duty cycle is configured to 50%, meaning the signal remains high for half the period. At each step in the counting process, the timer compares its current count with the duty cycle's compare value. When the timer count exceeds this

compare value (marked by the yellow arrow), the signal transitions from high to low. This triggers the compare interrupt, signaling the state change.

The time during which the signal is high is referred to as the pulse width.

I2C

I²C, also known as I2C or IIC (Inter-Integrated Circuit), is a communication protocol widely used to link microcontrollers with devices like sensors, displays, and other integrated circuits. This protocol is also called the Two-Wire Interface (TWI).

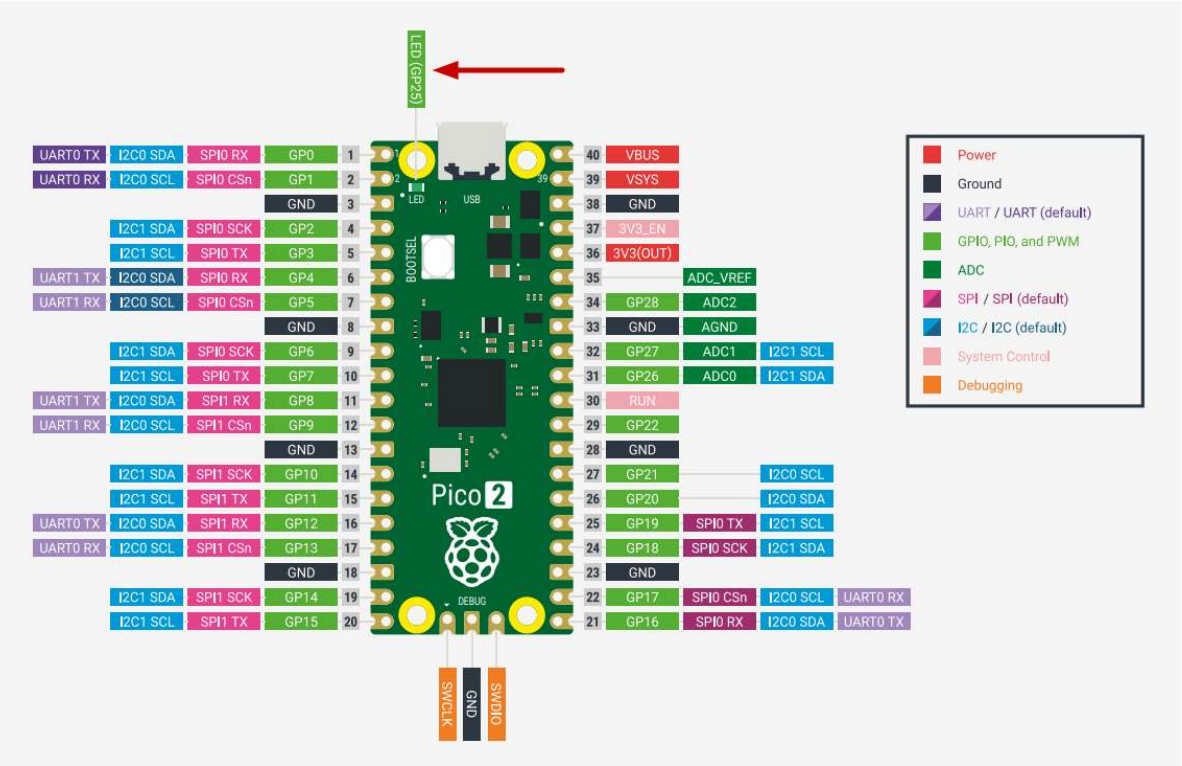
- Two Wires: Only 2 lines, SDA (Serial Data) and SCL (Serial Clock), are used to transfer data.
  - The SDA line is used by both the master and slave to send and receive data
  - The SCL line carries the clock signal
- Multi-Device Support: I2C allows multiple slave devices to be connected to a single master, and it also supports multiple masters.
- The master is responsible for generating the clock and controlling the transfer of data, while the slave responds by either transmitting or receiving data from the master.

RP2350's I2C

The RP2350 has 2 identical I2C controllers, each connected to specific GPIO pins on the Raspberry Pi Pico.

I2C Controller	GPIO Pins
I2C0 – SDA	GP0, GP4, GP8, GP12, GP16, GP20
I2C0 – SCL	GP1, GP5, GP9, GP13, GP17, GP21
I2C1 – SDA	GP2, GP6, GP10, GP14, GP18, GP26
I2C1 – SCL	GP3, GP7, GP11, GP15, GP19, GP27

Figure 2. The pinout of the Raspberry Pi Pico 2 board



Resources

- Basics of the I2C Communication Protocol
- A Basic Guide to I2C
- I2C in a Nutshell
- I2C

Watchdog

In January 1994, the Clementine spacecraft successfully mapped the moon. While traveling toward the asteroid Geographos, a floating point exception occurred on May 7, 1994, in the Honeywell 1750 processor, which was responsible for telemetry and various spacecraft functions.

The 1750 had a built-in watchdog timer but it was not utilized. The software team later regretted this decision and noted that a standard watchdog might not have been robust enough to detect the failure mode.

So, What exactly is a watchdog?

You might have already figured out its purpose.

What is watchdog?

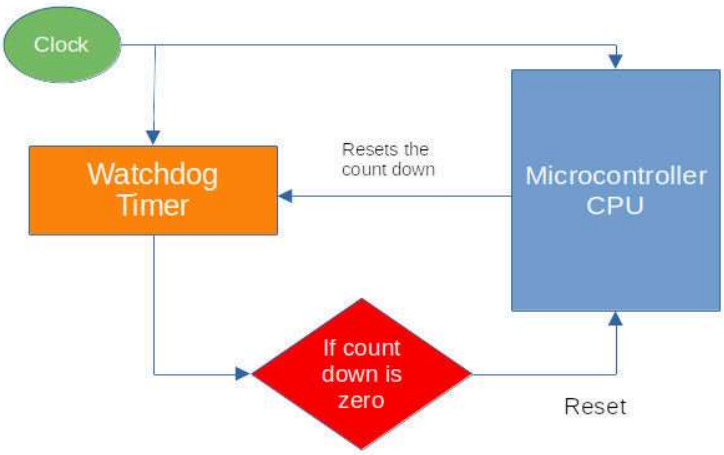
A watchdog timer (WDT) is a hardware component used in embedded systems, its primary purpose is to detect software anomalies and automatically reset the processor if a malfunction occurs, ensuring that the system can recover without human intervention.

How It Works?

The watchdog timer functions like a counter that counts down from a preset value to zero. The embedded software is responsible for periodically "feeding the dog" (also known as "kicking the dog," a term I don't like) by resetting the counter before it reaches zero. If the software fails to reset the counter (perhaps due to an infinite loop or a system hang), the watchdog timer assumes there's a problem and triggers a reset of the processor. This allows the system to restart and return to normal operation.

Feeding the dog:

Think of a watchdog timer like a dog that needs regular feeding to stay healthy and active. Just as you must feed your dog at scheduled intervals, the watchdog timer requires periodic resets to ensure that the embedded system is operating correctly. Imagine the dog's energy levels decreasing over time. If it runs out of energy, it will bark to alert you (just like the watchdog timer triggers an alert if it reaches zero). To keep the dog happy and active, you need to feed it regularly (or reset the timer) before it runs out of energy!



By implementing a watchdog timer, embedded systems can be made self-reliant, essential for devices that may be unreachable by human operators, such as space probes or other remote applications.

Code

In this code snippet, we were setting up the watchdog driver, which is essential for the clock setup process.

```
// Set up the watchdog driver - needed by the clock setup code
let mut watchdog = hal::Watchdog::new(pac.WATCHDOG);
```

References

- Great Watchdog Timers For Embedded Systems, by Jack Ganssle
- Born to fail
- A Guide to Watchdog Timers for Embedded Systems
- Proper Watchdog Timer Use

No Standard and Main in Rust

If you haven't read "The Embedded Rust Book" yet, I highly recommend you to check it out. https://docs.rust-embedded.org/book/intro/index.html

#![no_std]

The `#![no_std]` attribute disables the use of the standard library (std). This is necessary most of the times for embedded systems development, where the environment typically lacks many of the resources (like an operating system, file system, or heap allocation) that the standard library assumes are available.

Related Resources:

- Rust official doc
- The Embedded Rust Book
- Writing an OS in Rust Book

#![no_main]

The `#![no_main]` attribute is to indicate that the program won't use the standard entry point (fn main). Instead, it provides a custom entry point, usually required when working with embedded systems where the runtime environment is minimal or non-existent.

Related Resources:

- Rust official doc
- Writing an OS in Rust Book

Panic Handler

A panic handler is a function in Rust that defines what happens when your program encounters a panic. In environments without the standard library (when using no_std attribute), you need to create this function yourself using the #[panic_handler] attribute. This function must follow a specific format and can only appear once in your program. It provides details about the error, such as where it happened and why. By setting up a panic handler, you can choose how to respond to errors, like logging them for later review or stopping the program completely.

You don't have to define your own panic handler function; you can use existing crates such as panic_halt or panic_probe instead.

For example, we used the panic_halt crate to halt execution when a panic occurs.

```
use panic_halt as _;
```

The program will stop and remain in this infinite loop whenever a panic occurs.

In fact, the panic_halt crate's code implements a simple panic handler, which looks like this:

```
use core::panic::PanicInfo;
use core::sync::atomic::{self, Ordering};

#[inline(never)]
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {
        atomic::compiler_fence(Ordering::SeqCst);
    }
}
```

Related Resources:

- Rust official doc
- The Embedded Rust Book

Dimming LED

In this section, we will learn how to create a dimming effect(i.e. reducing and increasing the brightness) for an LED using the Raspberry Pi Pico 2. First, we will dim the onboard LED, which is connected to GPIO pin 25 (based on the datasheet).

To make it dim, we use a technique called PWM (Pulse Width Modulation). You can refer to the intro to the PWM section here.

We will gradually increment the PWM's duty cycle to increase the brightness, then we gradually decrement the PWM duty cycle to reduce the brightness of the LED. This effectively creates the dimming LED effect.

The Eye

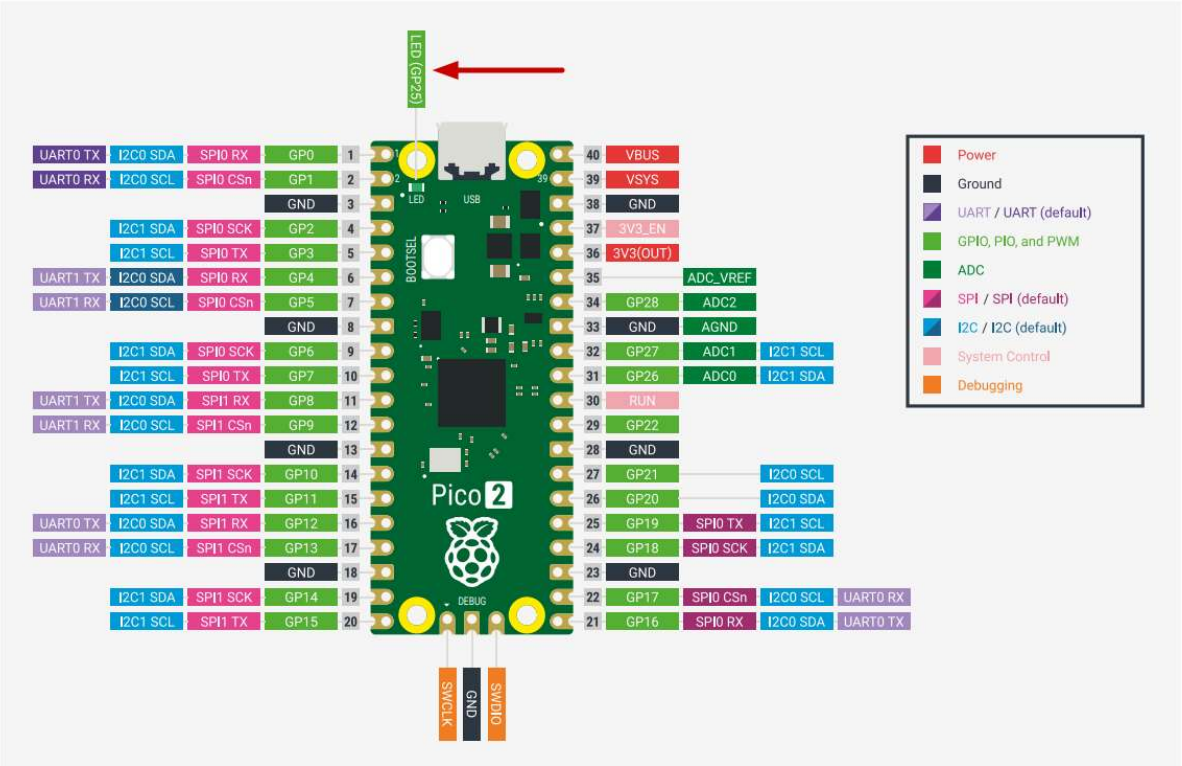
" Come in close... Closer...

Because the more you think you see... The easier it'll be to fool you...

Because, what is seeing?.... You're looking but what you're really doing is filtering, interpreting, searching for meaning... "

Now, here's the cool part: when this switching happens super quickly, our eyes can't keep up. Instead of seeing the blinking, it just looks like the brightness changes! The longer the LED stays ON, the brighter it seems, and the shorter it's ON, the dimmer it looks. It's like tricking your brain into thinking the LED is smoothly dimming or brightening. PWM: the magician of electronics!

Figure 2. The pinout of the Raspberry Pi Pico 2 board



PWM Peripheral in RP2350

The pico has a PWM peripheral with 12 PWM generators, known as slices. Each slice has two channels (A and B), providing a total of 24 PWM output channels available for use.

Refer the 1073th page of the RP2350 Datasheet for more information.

Mapping of PWM channels to GPIO Pins:

You can find the table on page 1073 of the RP2350 Datasheet for more information. You have to refer this table when using a specific GPIO, as each GPIO corresponds to a particular slice. For instance, using GP25 (for the LED) means you are working with output 4B, which is the B output of the fourth slice.

Table 1129. Mapping of PWM channels to GPIO pins on RP2350. This is also shown in the main GPIO function table, Table 645

GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
PWM Channel	8A	8B	9A	9B	10A	10B	11A	11B	8A	8B	9A	9B	10A	10B	11A	11B

Initialize the PWM slices by creating an instance using the PAC's PWM peripheral and reset control.

```
let mut pwm_slices = hal::pwm::Slices::new(pac.PWM, &mut pac.RESETS);
```

Retrieve a mutable reference to PWM4 from the initialized PWM slices for further configuration.

```
let pwm = &mut pwm_slices.pwm4;
```

Configure PWM4 to operate in phase-correct mode for smoother output transitions. (You can refer the secrets arudion PWM if you want to know what is phase-correct)

```
pwm.set_ph_correct();
```

Get a mutable reference to channel B of PWM4 and direct its output to GPIO pin 25.

```
let channel = &mut pwm.channel_b;
channel.output_to(pins.gpio25);
```

Fading Effect

Fading Up

The code below gradually increases the LED brightness by changing the duty cycle from 0 to 25,000, with a small delay between each step:

```
for i in LOW..=HIGH {
  delay.delay_us(8);
  let _ = channel.set_duty_cycle(i);
}
```

The delay ensures the LED brightens gradually. We need the delay between them; otherwise, it would quickly jump from dim to bright. The delay allows for a smooth, noticeable "fading up" effect.

Dont' believe me! Adjust the delay to 0 and observe. You can increase the delay (eg: 25) and observe the fading effect.

Note:
`set_duty_cycle`
function under the hood writes the given value into CC register(Count compare value).

Fading Down

The following code reduce the LED brightness by decrementing the duty cycle from 25,000 to 0.

```
// Here rev is to reverse the iteration. so it goes from 25_000 to 0
for i in (LOW..=HIGH).rev() {
    delay.delay_us(8);
    let _ = channel.set_duty_cycle(i);
}
```

Pause

After fading up and down, the program pauses for 500 milliseconds before repeating the cycle, allowing the LED to rest briefly.

Play around by adjusting the `delay` and observe. You can even comment out one of the for loop and observe the effect.

Blink (Dimming) LED Program with RP HAL

rp-hal is an Embedded-HAL for RP series microcontrollers, and can be used as an alternative to the Embassy framework for pico.

This example code is taken from rp235x-hal repo (It also includes additional examples beyond just the blink examples):

"https://github.com/rp-rs/rp-hal/tree/main/rp235x-hal-examples"

The main code

```
//! # PWM Blink Example
//!
//! If you have an LED connected to pin 25, it will fade the LED using
the PWM
//! peripheral.
//!
//! It may need to be adapted to your particular board layout and/or
pin assignment.
//!
//! See the `Cargo.toml` file for Copyright and license details.

#![no_std]
#![no_main]

// Ensure we halt the program on panic (if we don't mention this crate
it won't
// be linked)
use panic_halt as _;

// Alias for our HAL crate
use rp235x_hal as hal;

// Some things we need
use embedded_hal::delay::DelayNs;
use embedded_hal::pwm::SetDutyCycle;

/// Tell the Boot ROM about our application
#[link_section = ".start_block"]
#[used]
pub static IMAGE_DEF: hal::block::ImageDef =
hal::block::ImageDef::secure_exe();
/// The minimum PWM value (i.e. LED brightness) we want
const LOW: u16 = 0;

/// The maximum PWM value (i.e. LED brightness) we want
const HIGH: u16 = 25000;

/// External high-speed crystal on the Raspberry Pi Pico 2 board is 12
MHz.
/// Adjust if your board has a different frequency
const XTAL_FREQ_HZ: u32 = 12_000_000u32;

/// Entry point to our bare-metal application.
///
/// The `#[hal::entry]` macro ensures the Cortex-M start-up code calls
this function
/// as soon as all global variables and the spinlock are initialised.
///
```

```
/// The function configures the rp235x peripherals, then fades the LEI
in an
/// infinite loop.
#[hal::entry]
fn main() -> ! {
    // Grab our singleton objects
    let mut pac = hal::pac::Peripherals::take().unwrap();

    // Set up the watchdog driver - needed by the clock setup code
    let mut watchdog = hal::Watchdog::new(pac.WATCHDOG);

    // Configure the clocks
    //
    // The default is to generate a 125 MHz system clock
    let clocks = hal::clocks::init_clocks_and_plls(
        XTAL_FREQ_HZ,
        pac.XOSC,
        pac.CLOCKS,
        pac.PLL_SYS,
        pac.PLL_USB,
        &mut pac.RESETS,
        &mut watchdog,
    )
    .ok()
    .unwrap();

    // The single-cycle I/O block controls our GPIO pins
    let sio = hal::Sio::new(pac.SIO);

    // Set the pins up according to their function on this particular
board
    let pins = hal::gpio::Pins::new(
        pac.IO_BANK0,
        pac.PADS_BANK0,
        sio.gpio_bank0,
        &mut pac.RESETS,
    );

    // The delay object lets us wait for specified amounts of time (ı
    // milliseconds)
    let mut delay = hal::Timer::new_timer0(pac.TIMER0, &mut
pac.RESETS, &clocks);

    // Init PWMs
    let mut pwm_slices = hal::pwm::Slices::new(pac.PWM, &mut
pac.RESETS);

    // Configure PWM4
    let pwm = &mut pwm_slices.pwm4;
    pwm.set_ph_correct();
    pwm.enable();
```

```
    // Output channel B on PWM4 to GPIO 25
    let channel = &mut pwm.channel_b;
    channel.output_to(pins.gpio25);

    // Infinite loop, fading LED up and down
    loop {
        // Ramp brightness up
        for i in LOW..=HIGH {
            delay.delay_us(8);
            let _ = channel.set_duty_cycle(i);
        }

        // Ramp brightness down
        for i in (LOW..=HIGH).rev() {
            delay.delay_us(8);
            let _ = channel.set_duty_cycle(i);
        }

        delay.delay_ms(500);
    }
}

/// Program metadata for `picotool info`
#[link_section = ".bi_entries"]
#[used]
pub static PICOTOOL_ENTRIES: [hal::binary_info::EntryAddr; 5] = [
    hal::binary_info::rp_cargo_bin_name!(),
    hal::binary_info::rp_cargo_version!(),
    hal::binary_info::rp_program_description!(c"PWM Blinky Example"),
    hal::binary_info::rp_cargo_homepage_url!(),
    hal::binary_info::rp_program_build_attribute!(),
];

// End of file
```

Clone the existing project

You can clone the blinky project I created and navigate to the `blinky` folder to run this version of the blink program:

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/blinky
```

How to Run?

You refer the "Running The Program" section

Blinking an External LED

In this section, we'll use an external LED to blink.

You'll need some basic electronic components

Hardware Requirements

- LED
- Resistor
- Jumper wires

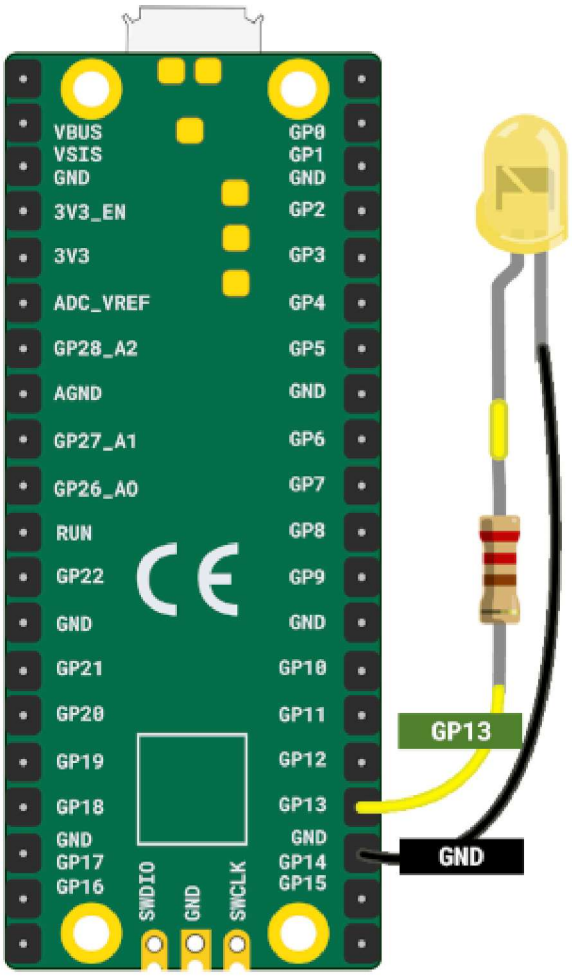
Refer the Raspberry pi guide for the hardware setup. I actually used it with breadboard instead.

Components Overview

1. LED: An LED (Light Emitting Diode) lights up when current flows through it. The longer leg (anode) connects to positive, and the shorter leg (cathode) connects to ground. We'll connect the anode to GP13 (with a resistor) and the cathode to GND.

2. Resistors: A resistor limits the current in a circuit to protect components like LEDs. Its value is measured in Ohms (Ω). We'll use a 330 ohm resistor to safely power the LED.

Pico Pin	Wire	Component
GPIO 13		Resistor
Resistor		Anode (long leg) of LED
GND		Cathode (short leg) of LED



Code

There isn't much change in the code. We showcase how to blink an external LED connected to GPIO 13. The code uses a push-pull output configuration to control the LED state. By setting the pin high, the LED is turned on, and by setting it low, the LED is turned off. This process is repeated in an infinite loop with a delay of 200 milliseconds between each state change, allowing the LED to blink at a consistent rate.

```
let mut led_pin = pins.gpio13.into_push_pull_output();

loop {
    led_pin.set_high().unwrap();
    delay.delay_ms(200);
    led_pin.set_low().unwrap();
    delay.delay_ms(200);
}
```

Clone the existing project

You can clone the blinky project I created and navigate to the `external-led` folder to run this version of the blink program:

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/external-led
```

How to Run?

You refer the "Running The Program" section

Ultrasonic

In this guide, we'll learn how to use the HC-SR04 ultrasonic sensor with the Raspberry Pi Pico 2. Ultrasonic sensors measure distances by emitting ultrasonic sound waves and calculating the time taken for them to return after bouncing off an object.

These kind of sensors you can normally find in the car parking assistance; When you reverse the car for parking, the sensor measures the distance between objects and alert you as you get close to it.

We will build a simple project that gradually increases the LED brightness using PWM, when the ultrasonic sensor detects an object distance of less than 30 cm - You can adjust this value as per your needs.

Prerequisites

Before starting, get familiar with yourself on these topics

- PWM

🛠 Hardware Requirements

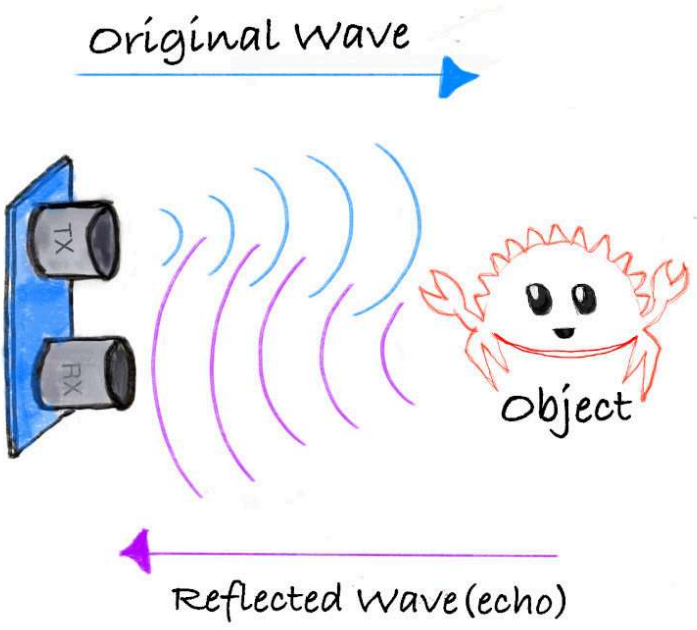
To complete this project, you will need:

- HC-SR04 Ultrasonic Sensor
- Breadboard
- Jumper wires
- External LED (You can also use the onboard LED, but you'll need to modify the code accordingly)

The HC-SR04 Sensor module has a transmitter and receiver. The module has Trigger and Echo pins which can be connected to the GPIO pins of a pico. When the receiver detects the returning sound wave, the Echo pin goes HIGH for a duration equal to the time it takes for the wave to return to the sensor.

How Does an Ultrasonic Sensor Work?

Ultrasonic sensors work by emitting sound waves at a frequency too high(40kHz) for humans to hear. These sound waves travel through the air and bounce back when they hit an object. The sensor calculates the distance by measuring how long it takes for the sound waves to return.



- **Transmitter:** Sends out ultrasonic sound waves.
- **Receiver:** Detects the sound waves that bounce back from an object.

Formula to calculate distance:

```
Distance = (Time x Speed of Sound) / 2
```

The speed of sound is approximately 0.0343 cm/μs (or 343 m/s) at normal air pressure and a temperature of 20°C.

Example Calculation:

Let's say the ultrasonic sensor detects that the sound wave took 2000 µs to return after hitting an object.

Step 1: Calculate the total distance traveled by the sound wave:

```
Total distance = Time x Speed of Sound
Total distance = 2000 µs x 0343 cm/µs = 68.6 cm
```

Step 2: Since the sound wave traveled to the object and back, the distance to the object is half of the total distance:

```
Distance to object = 68.6 cm / 2 = 34.3 cm
```

Thus, the object is 34.3 cm away from the sensor.

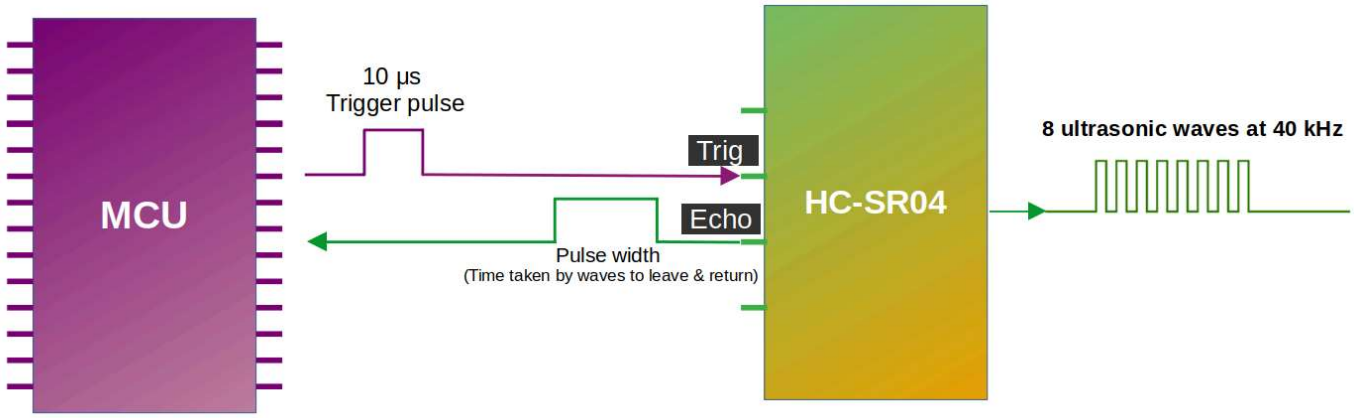
HC-SR04 Pinout

The module has four pins: VCC, Trig, Echo, and GND.

Pin	Function
VCC	Power Supply
Trig	Trigger Signal
Echo	Echo Signal
GND	Ground

Measuring Distance with the HC-SR04 module

The HC-SR04 module has a transmitter and receiver, responsible for sending ultrasonic waves and detecting the reflected waves. We will use the Trig pin to send sound waves. And read from the Echo pin to measure the distance.



As you can see in the diagram, we connect the Trig and Echo pins to the GPIO pins of the microcontroller (we also connect VCC and GND but left them out to keep the illustration simple). We send ultrasonic waves by setting the Trig pin HIGH for 10 microseconds and then setting it back to LOW. This triggers the module to send 8 consecutive ultrasonic waves at a frequency of 40 kHz. It is recommended to have a minimum gap of 50ms between each trigger.

When the sensor's waves hit an object, they bounce back to the module. As you can see in the diagram, the Echo pin changes the input sent to the microcontroller, with the length of time the signal stays HIGH (pulse width) corresponding to the distance. In the microcontroller, we measure how long the Echo pin stays HIGH; Then, we can use this time duration to calculate the distance to the object.

Pulse width and the distance:

The pulse width (amount of time it stays high) produced by the Echo pin will range from about 150µs to 25,000µs(25ms); this is only if it hits an object. If there is no object, it will produce a pulse width of around 38ms.

Circuit

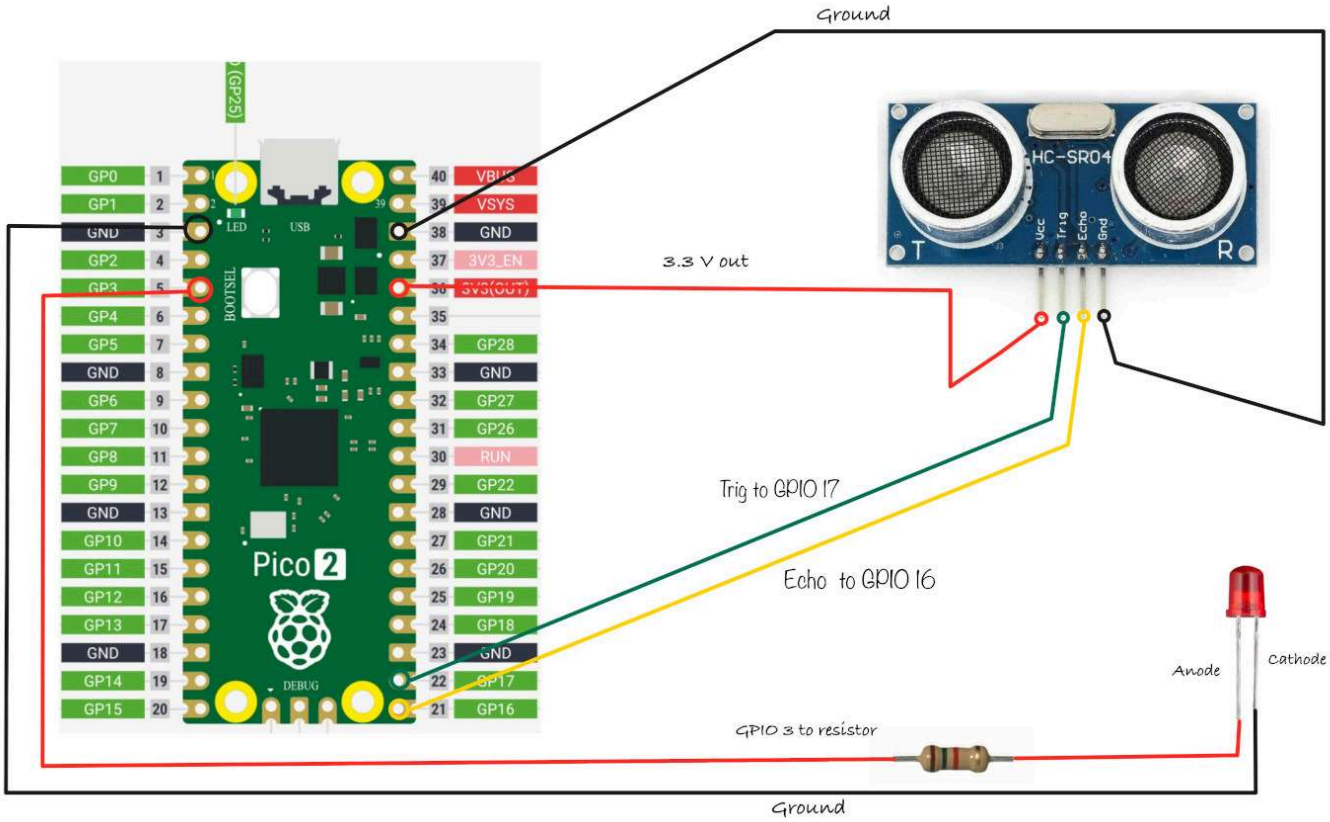
Connection for the Pico and Ultrasonic:

Pico Pin	Wire	HC-SR04 Pin
3.3V		VCC
GPIO 17		Trig
GPIO 16		Echo
GND		GND

- **VCC**: Connect the VCC pin on the HC-SR04 to the 3.3V pin on the Pico.
  - Although the HC-SR04 generally operates at 5V, using 3.3V helps protect the Pico since the Pico GPIO pins are only 3.3V tolreant.
  - Other considerations: Alternatively, you can use HCSR04+ (which can operate at both 3.3v and 5V) or use voltage divider.
- **Trig**: Connect to GPIO 17 on the Pico to start the ultrasonic sound pulses.
- **Echo**: Connect to GPIO 16 on the Pico; this pin sends a pulse when it detects the reflected signal, and the pulse length shows how long the signal took to return.
- **GND**: Connect to the ground pin on the Pico.
- **LED**: Connect the anode (long leg) of the LED to GPIO 3, as in the External LED setup.

Connection for the Pico and LED:

Pico Pin	Wire	Component
GPIO 3		Resistor
Resistor		Anode (long leg) of LED
GND		Cathode (short leg) of LED


Ground
HC-SR04
Vcc
Trig
Echo
Gnd
T
R
3.3 V out
Trig to GPIO 17
Echo to GPIO 16
Anode
Cathode
GPIO 3 to resistor
Ground
Pico 2
LED
USB
BOOTSEL
DEBUG
GP0
GP1
GND
GP2
GP3
GP4
GP5
GND
GP6
GP7
GP8
GP9
GND
GP10
GP11
GP12
GP13
GND
GP14
GP15
VBUS
VSYS
GND
3V3_EN
3V3(OUT)
GP28
GND
GP27
GP26
RUN
GP22
GND
GP21
GP20
GP19
GP18
GND
GP17
GP16

Writing Rust Code Use HC-SR04 Ultrasonic Sensor with Pico 2

We'll start by generating the project using the template, then modify the code to fit the current project's requirements.

Generating From template

Refer to the Template section for details and instructions.

To generate the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, choose a name for your project-let's go with "bat-beacon". Don't forget to select
`rp-hal`
as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "bat-beacon":
# cd bat-beacon
```

Setup the LED Pin

You should understand this code by now. If not, please complete the Blink LED section first.

Quick recap: Here, we're configuring the PWM for the LED, which allows us to control the brightness by adjusting the duty cycle.

```
let pwm = &mut pwm_slices.pwm6;  // Access PWM slice 6
pwm.set_ph_correct();            // Set phase-correct mode for
smoother transitions
pwm.enable();                    // Enable the PWM slice
let led = &mut pwm.channel_b; // Select PWM channel B
led.output_to(pins.gpio3);   // Set GPIO 3 as the PWM output pin
```

Setup the Trigger Pin

The Trigger pin on the ultrasonic sensor is used to start the ultrasonic pulse. It needs to be set as an output so we can control it to send the pulse.

```
let mut trigger = pins.gpio17.into_push_pull_output();
```

Setup the Echo Pin

The Echo pin on the ultrasonic sensor receives the returning signal, which allows us to measure the time it took for the pulse to travel to an object and back. It's set as an input to detect the returning pulse.

```
let mut echo = pins.gpio16.into_pull_down_input();
```

Light it Up

Step 1: Send the Trigger Pulse

First, we need to send a short pulse to the trigger pin to start the ultrasonic measurement.

```
// Ensure the Trigger pin is low before starting
trigger.set_low().ok().unwrap();
timer.delay_us(2);

// Send a 10-microsecond high pulse
trigger.set_high().ok().unwrap();
timer.delay_us(10);
trigger.set_low().ok().unwrap();
```

Step 2: Measure the Echo Time

Next, we will use two loops. The first loop will run as long as the echo pin state is LOW. Once it goes HIGH, we will record the current time in a variable. Then, we start the second loop, which will continue as long as the echo pin remains HIGH. When it returns to LOW, we will record the current time in another variable. The difference between these two times gives us the pulse width.

```
let mut time_low = 0;
let mut time_high = 0;

// Wait for the Echo pin to go high and note down the time
while echo.is_low().ok().unwrap() {
    time_low = timer.get_counter().ticks();
}

// Wait for the Echo pin to go low and note down the time
while echo.is_high().ok().unwrap() {
    time_high = timer.get_counter().ticks();
}

// Calculate the time taken for the signal to return
let time_passed = time_high - time_low;
```

Step 3: Calculate Distance

To calculate the distance, we need to use the pulse width. The pulse width tells us how long it took for the ultrasonic waves to travel to an obstacle and return. Since the pulse represents the round-trip time, we divide it by 2 to account for the journey to the obstacle and back.

The speed of sound in air is approximately 0.0343 cm per microsecond. By multiplying the time (in microseconds) by this value and dividing by 2, we obtain the distance to the obstacle in centimeters.

```
let distance = time_passed as f64 * 0.0343 / 2.0;
```

Step 4: PWM Duty cycle for LED

Finally, we adjust the LED brightness based on the measured distance.

The duty cycle percentage is calculated using our own logic, you can modify it to suit your needs. When the object is closer than 30 cm, the LED brightness will increase. The closer the object is to the ultrasonic module, the higher the calculated ratio will be, which in turn adjusts the duty cycle. This results in the LED brightness gradually increasing as the object approaches the sensor.

```
let duty_cycle = if distance < 30.0 {
    let step = 30.0 - distance;
    (step * 1500.) as u16 + 1000
} else {
    0
};

// Change the LED brightness
led.set_duty_cycle(duty_cycle).unwrap();
```

Complete Logic of the loop

Note: This code snippet highlights the loop section and does not include the entire code.

```
loop {
    timer.delay_ms(5);

    trigger.set_low().ok().unwrap();
    timer.delay_us(2);
    trigger.set_high().ok().unwrap();
    timer.delay_us(10);
    trigger.set_low().ok().unwrap();

    let mut time_low = 0;
    let mut time_high = 0;
    while echo.is_low().ok().unwrap() {
        time_low = timer.get_counter().ticks();
    }
    while echo.is_high().ok().unwrap() {
        time_high = timer.get_counter().ticks();
    }
    let time_passed = time_high - time_low;

    let distance = time_passed as f64 * 0.0343 / 2.0;

    let duty_cycle = if distance < 30.0 {
        let step = 30.0 - distance;
        (step * 1500.) as u16 + 1000
    } else {
        0
    };
    led.set_duty_cycle(duty_cycle).unwrap();
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `ultrasonic` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/ultrasonic
```

Your Challenge

1. Use Embassy framework instead of rp-hal
2. Use the onboard LED instead

OLED Display

In this section, we'll learn how to connect an OLED display module to the Raspberry Pi Pico 2.

We'll create simple projects like displaying text and an image (display Ferris 🦀 image) on the OLED. We'll use the I2C protocol to connect the OLED display to the Pico.

Hardware Requirements

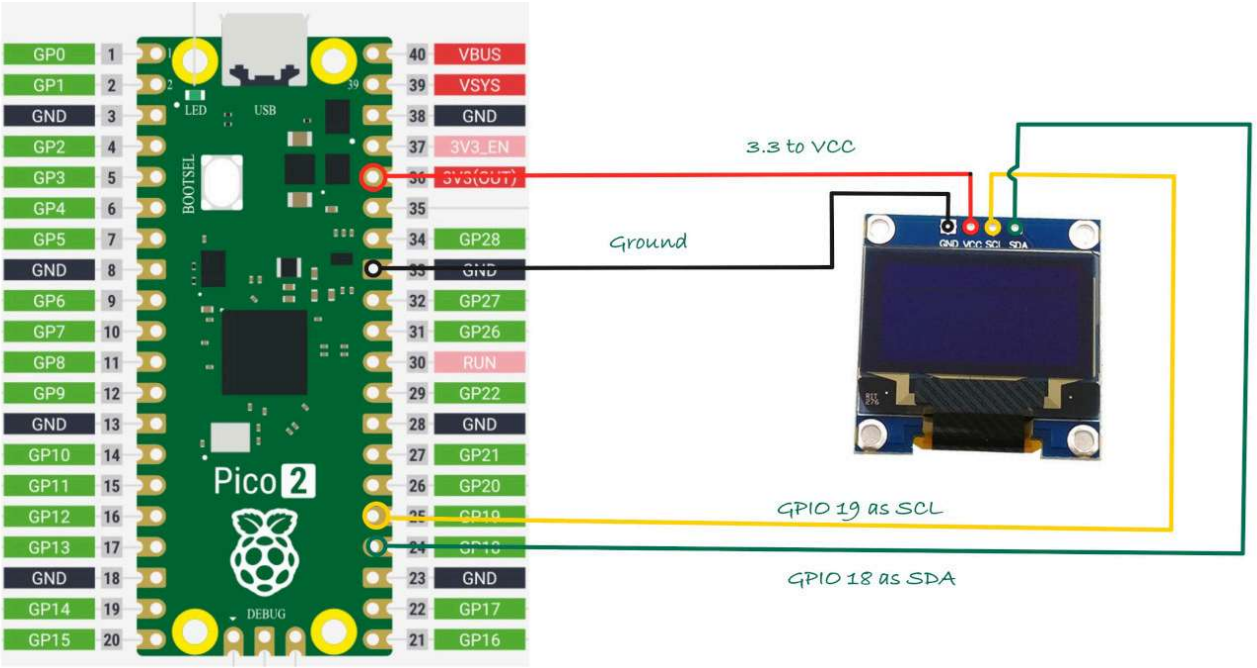
For this project, you'll need:

- An OLED display (0.96 Inch I2C/IIC 4-Pin, 128x64 resolution, SSD1306 chip)
- A breadboard
- Jumper wires

Setup

Pico Pin	Wire	OLED Pin
GPIO 18		SDA
GPIO 19		SCL
3.3V		VCC
GND		GND

We will connect the SDA to GPIO 18 and the SCL to GPIO 19. Attach VCC to 3.3V for power, and GND to GND. This setup allows the OLED display to communicate with the microcontroller using I2C.



New crates

In addition to the usual crate like `rp-hal`, we will be using these new crates necessary for the project.

- `ssd1306`
  : a driver for the SSD1306 OLED display, supporting both I2C and 4-wire SPI.
- `embedded-graphics`

: a 2D graphics library tailored for memory-constrained embedded devices, enabling text and graphic rendering.

- `tinybmp`

  : a lightweight BMP parser for embedded, no-std environments. We'll use this to directly load `.bmp` images with the `embedded-graphics` crate, avoiding the need for raw image data.

Prerequisite

- I2C protocol

Resources

- SSD1306 Datasheet

Hello Rust on OLED

We will create a simple program to display "Hello, Rust" in the OLED display.

Generating From template

Refer to the Template section for details and instructions.

To generate the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, choose a name for your project-let's go with "oh-led". Don't forget to select
`rp-hal`
as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "oh-led":
# cd oh-led
```

Add Additional Dependencies

Since we are using the SSD1306 OLED display, we need to include the SSD1306 driver. To add this dependency, use the following Cargo command:

```
cargo add ssd1306@0.9.0
```

We will use the embedded_graphics crate to handle graphical rendering on the OLED display, to draw images, shapes, and text.

```
cargo add embedded-graphics@0.8.1
```

Code

Additional imports

In addition to the imports from the template, you'll need the following additional dependencies for this task.

```
use hal::fugit::RateExtU32;
use hal::gpio::{FunctionI2C, Pin};
use ssd1306::{prelude::*, I2CDisplayInterface, Ssd1306};
use embedded_graphics::prelude::*;
use embedded_graphics::mono_font::ascii::FONT_6X10;
use embedded_graphics::mono_font::MonoTextStyleBuilder;
use embedded_graphics::pixelcolor::BinaryColor;
use embedded_graphics::text::{Baseline, Text};
```

Pin Configuration

We start by configuring the GPIO pins for the I2C communication. In this case, GPIO18 is set as the SDA pin, and GPIO19 is set as the SCL pin. We then configure the I2C peripheral to work in master mode.

```
// Configure two pins as being I²C, not GPIO
let sda_pin: Pin<_, FunctionI2C, _> = pins.gpio18.reconfigure();
let scl_pin: Pin<_, FunctionI2C, _> = pins.gpio19.reconfigure();

let i2c = hal::I2C::i2c1(
    pac.I2C1,
    sda_pin,i2c1
    scl_pin,
    400.kHz(),
    &mut pac.RESETS,
    &clocks.system_clock,
);
```

Prepare Display

We create an interface for the OLED display using the I2C.

```
//helper struct is provided by the ssd1306 crate
let interface = I2CDisplayInterface::new(i2c);
// initialize the display
let mut display = Ssd1306::new(interface, DisplaySize128x64,
DisplayRotation::Rotate0)
    .into_buffered_graphics_mode();
display.init().unwrap();
```

Set Text Style and Draw

Next, we define the text style and use it to display "Hello Rust" on the screen:

```
// Embedded graphics
let text_style = MonoTextStyleBuilder::new()
    .font(&FONT_6X10)
    .text_color(BinaryColor::On)
    .build();

Text::with_baseline("Hello, Rust!", Point::new(0, 16), text_style,
Baseline::Top)
    .draw(&mut display)
    .unwrap();
```

Here, we are writing the message at coordinates (x=0, y=16).

Write out data to a display

```
display.flush().unwrap();
```

Full logic

```
let sda_pin: Pin<_, FunctionI2C, _> = pins.gpio18.reconfigure();
let scl_pin: Pin<_, FunctionI2C, _> = pins.gpio19.reconfigure();

let i2c = hal::I2C::i2c1(
    pac.I2C1,
    sda_pin,
    scl_pin,
    400.kHz(),
    &mut pac.RESETS,
    &clocks.system_clock,
);

let interface = I2CDisplayInterface::new(i2c);

let mut display = Ssd1306::new(interface, DisplaySize128x64,
DisplayRotation::Rotate0)
    .into_buffered_graphics_mode();

display.init().unwrap();
let text_style = MonoTextStyleBuilder::new()
    .font(&FONT_6X10)
    .text_color(BinaryColor::On)
    .build();

Text::with_baseline("Hello, Rust!", Point::new(0, 16), text_style,
Baseline::Top)
    .draw(&mut display)
    .unwrap();

display.flush().unwrap();
loop {
    timer.delay_ms(500);
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `hello-oled` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/hello-oled
```

Ferris on OLED

In this task, we will display the `ferris.bmp` (Download and put it in the project folder) file on the OLED screen.

Follow the same instructions as in the "Hello Rust" program, with just a few modifications.

First, we need to add the tinybmp crate to handle BMP file loading. Use the following Cargo command to include it in your project:

```
cargo add tinybmp@0.6.0
```

Additional imports

```
use embedded_graphics::image::Image;
use tinybmp::Bmp;
```

Difference

After initializing the display, we will load the ferris.bmp file using the tinybmp crate, and then we will draw the image.

```
let bmp = Bmp::from_slice(include_bytes!("../ferris.bmp")).unwrap();
let im = Image::new(&bmp, Point::new(32, 0));
im.draw(&mut display).unwrap();
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `ferris-oled` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/ferris-oled
```

Useful tools

- Bitmap to Rust array Converter: Image2Bytes Generator

Servo Motor and PWM

In this section, we'll connect an SG90 Micro Servo Motor to the Pico 2 and control its rotation using PWM. The servo will move in a loop, rotating from 0 degrees to 90 degrees, and then to 180 degrees.

Before moving forward, make sure you've read the PWM introduction in the Blink LED section.

Hardware Requirements

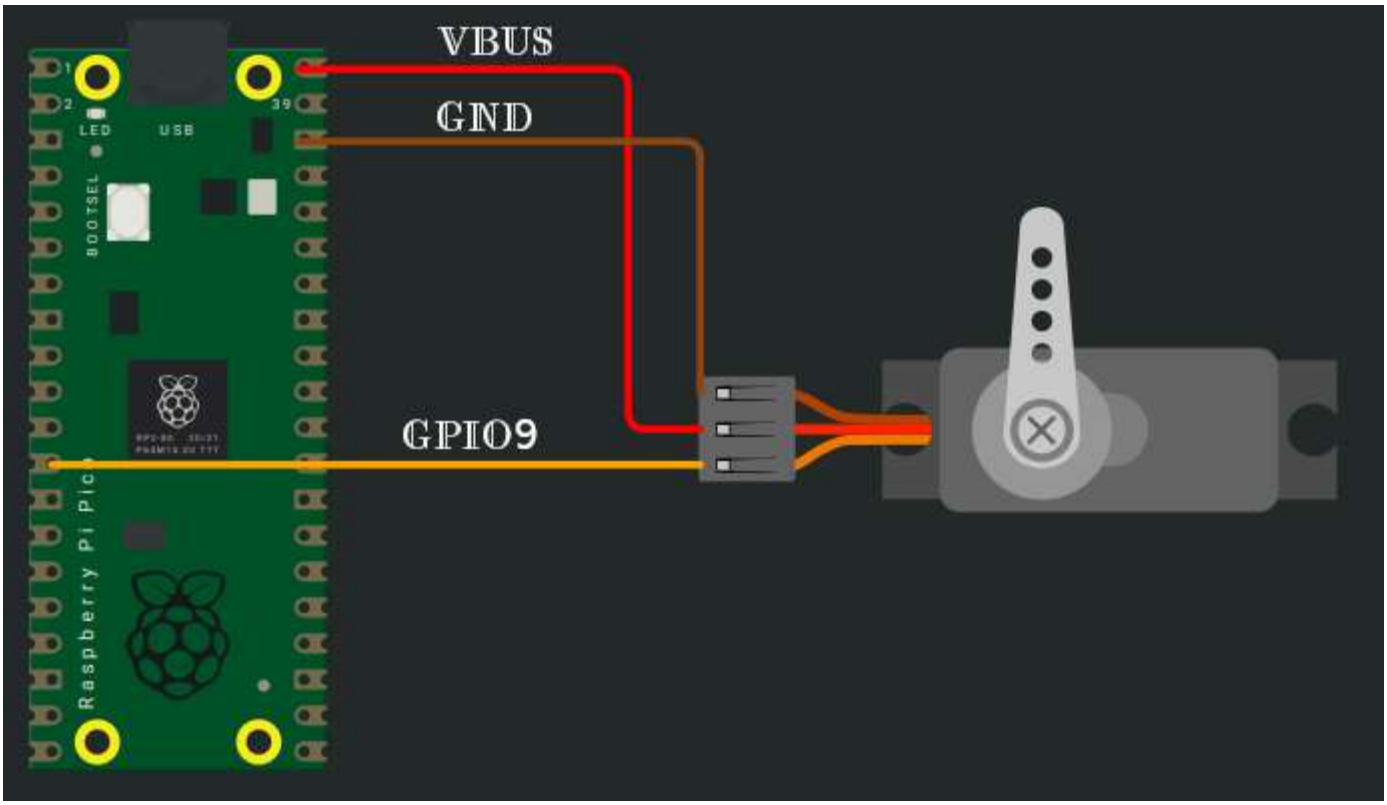
- **SG90 Micro Servo Motor**
- **Jumper Wires**:
  - **Female-to-Male**(or Male to Male depending on how you are connecting) jumper wires for connecting the Pico 2 to the servo motor pins (Ground, Power, and Signal).

Connection Overview

1. **Ground (GND)**: Connect the servo's GND pin (typically the **brown** wire, though it may vary) to any ground pin on the Pico 2.
2. **Power (VCC)**: Connect the servo's VCC pin (usually the **red** wire) to the Pico 2's 5V (or 3.3V if required by your setup) power pin.
3. **Signal (PWM)**: Connect the servo's control (signal) pin to **GPIO9** on the Pico 2, configured for PWM. This is commonly the **orange** wire (may vary).

Pico Pin	Wire	Servo Motor	Notes
VBUS		Power (Red Wire)	Supplies 5V power to the servo.

Pico Pin	Wire	Servo Motor	Notes
GND		Ground (Brown Wire)	Connects to ground.
GPIO 9		Signal (Orange Wire)	Receives PWM signal to control the servo's position.



Introduction to Servo Motors

A servo motor controls movement by adjusting its position using a feedback system. It is guided by a signal, usually Pulse Width Modulation (PWM), to reach and maintain the desired position.

They are widely used in applications requiring precise motion, such as robotics, RC vehicles, and camera systems, as well as in various projects. Hobby servos, which are often used in RC toys like cars, airplanes, are also popular for building robots.

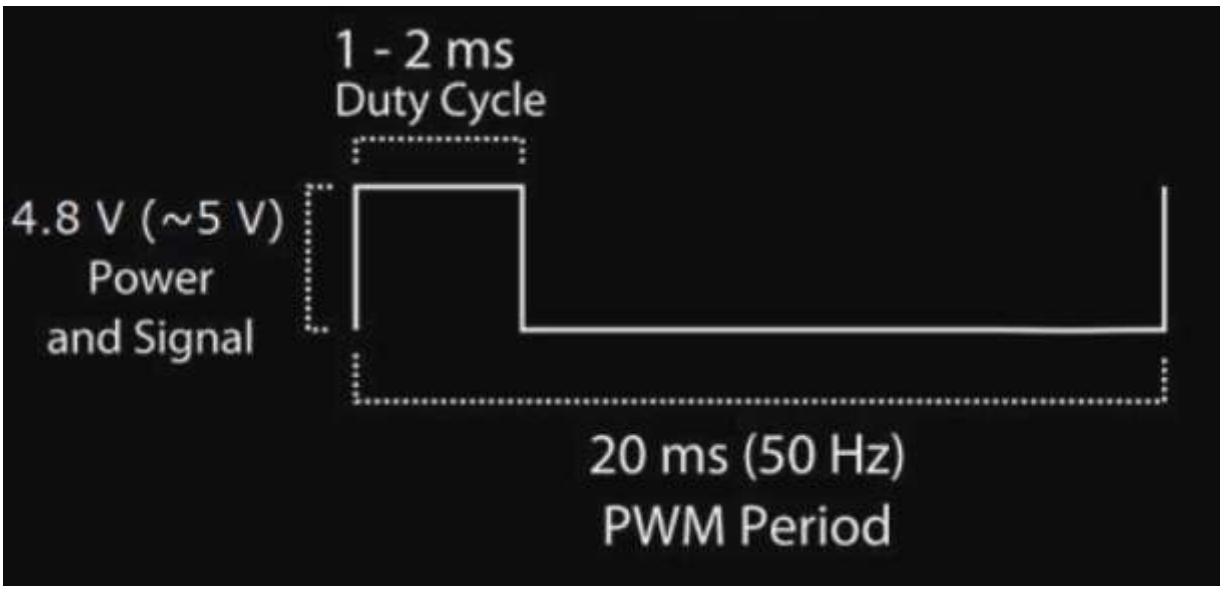
In our exercise, we'll be using the hobby server (Micro Servo SG90)

How does it work?

A servo motor is controlled by sending a series of pulses through its signal line. The signal has a frequency of 50Hz, with a pulse every 20 milliseconds. The width of the pulse determines the servo's position. Typically, a servo can rotate 180 degrees.

Controlling the position

The position of a servo motor is controlled by sending a pulse with a specific duration. The length of the pulse determines the angle of the motor. For most servos, a 1ms pulse moves the motor to 0 degrees, a 1.5ms pulse moves it to 90 degrees (neutral position), and a 2ms pulse moves it to 180 degrees.



However, from my experiment, I found that not all servos follow these exact timings. For example, with my servo, the pulse duration for 0 degrees was 0.5ms, 1.5ms for 90 degrees, and approximately 2.4ms for 180 degrees. I had to experiment and adjust to get it right. If you're unsure, you can use tools like an oscilloscope to fine-tune it, or simply test different values to find what works best for your specific servo.

The example I'll provide in this exercise is based on my servo's configuration, you might need to adjust the values depending on the servo you're using.

Reference

- Learn to Control Servo motor using PWM - Wokwi Style

More on PWM

The servo motor we're using operates at a 50Hz frequency, which means that a pulse is sent every 20 milliseconds (ms).

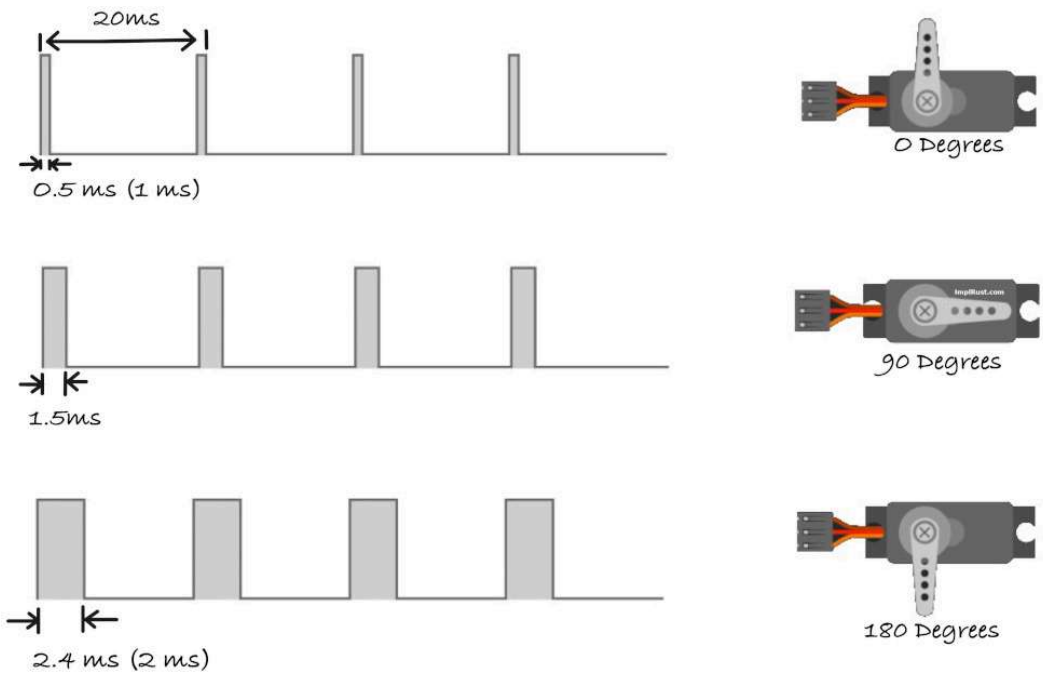
Let's break this down further:

- **50Hz Frequency:** Frequency refers to how many times an event happens in a given time period. A 50Hz frequency means that the servo expects a pulse to occur 50 times per second. In other words, the servo receives a pulse every 1/50th of a second, which is 20 milliseconds.
- **20ms Time Interval:** This 20ms is the time between each pulse. It means that every 20 milliseconds, the servo expects a new pulse to adjust its position. Within this 20ms period, the width of the pulse (how long it stays "high") determines the angle at which the servo will move.

So, when we say the servo operates at 50Hz, it means that the motor is constantly receiving pulses every 20ms to keep it in motion or adjust its position based on the width of each pulse.

Pulse Width and Duty Cycle

Let's dive deeper into how different pulse widths like 0.5ms, 1.5ms, and 2.4ms affect the servo's position.



1. **0.5ms Pulse (Position: 0 degrees)**

   - **What Happens**: A 0.5ms pulse means the signal is "high" for 0.5 milliseconds within each 20ms cycle. The servo interprets this as a command to move to the 0-degree position.

   - **Duty Cycle**: The duty cycle refers to the percentage of time the signal is "high" in one complete cycle. For a 0.5ms pulse:

     $$\text{Duty Cycle } (\%) = \frac{0.5\text{ms}}{20\text{ms}} \times 100 = 2.5\%$$

     This means that for just 2.5% of each 20ms cycle, the signal stays "high" causing the servo to rotate to the 0-degree position.

2. **1.5ms Pulse (Position: 90 degrees)**

   - **What Happens**: A 1.5ms pulse means the signal is "high" for 1.5 milliseconds in the 20ms cycle. The servo moves to its neutral position, around 90 degrees (middle position).
   - **Duty Cycle**: For a 1.5ms pulse:

$$\text{Duty Cycle (\%)} = \frac{1.5\text{ms}}{20\text{ms}} \times 100 = 7.5\%$$

Here, the signal stays "high" for 7.5% of the cycle, which positions the servo at 90 degrees (neutral).

3. **2.4ms Pulse (Position: 180 degrees)**

- **What Happens**: A 2.4ms pulse means the signal is "high" for 2.4 milliseconds in the 20ms cycle. The servo will move to its maximum position, typically 180 degrees (full rotation to one side).
- **Duty Cycle**: For a 2.4ms pulse:

$$\text{Duty Cycle (\%)} = \frac{2.4\text{ms}}{20\text{ms}} \times 100 = 12\%$$

In this case, the signal is "high" for 12% of the cycle, which causes the servo to rotate to 180 degrees.

Servo and Pico

To control a servo with the Raspberry Pi Pico, we need to set a 50Hz PWM frequency. Currently, RP-HAL doesn't allow directly setting the frequency, so we achieve this by adjusting the `top` and `div_int` values.

Refer the 1073th page of the RP2350 Datasheet to understand how `top` and `div_int` works.

Formula from datasheet

The following formula from the datasheet is used to calculate the period and determine the output frequency based on the system clock frequency.

1. **Period calculation:**

$$\text{period} = (\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \Big(\text{DIV_INT}$$

2. **PWM output frequency calculation:**

$$f_{PWM} = \frac{f_{sys}}{\text{period}} = \frac{f_{sys}}{(\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \Big(\text{DIV}}$$

Where:

- f_{PWM} is the PWM output frequency.
- f_{sys} is the system clock frequency. For the pico2, it is is 150MHZ.

Let's calculate `top`

We want the PWM frequency (f_pwm) to be 50 Hz. In order to achieve that, we are going to adjust the `top` and `div_int` values.

The top value must be within the range of 0 to 65535 (since it's a 16-bit unsigned integer). To make sure the top value fits within this range, I chose values for the divisor (div_int) in powers of 2 (such as 8, 16, 32, 64), though this isn't strictly necessary (it's just a preference). In this case, we chose `div_int = 64` to calculate a top value that fits within the u16 range.

With the chosen div_int and system parameters, we can calculate the top using the following formula:

$$\text{top} = \frac{150,000,000}{50 \times 64} - 1$$

$$\text{top} = \frac{150,000,000}{3,200} - 1$$

$$\text{top} = 46,875 - 1$$

$$\text{top} = 46,874$$

After performing the calculation, we find that the top value is `46,874`.

You can experiment with different div_int and corresponding top values. Just ensure that div_int stays within the u8 range, top fits within the u16 range, and the formula yields a 50Hz frequency.

Note:

- In case you are wondering, we are not setting the

`div_frac`

which is 0 by default. That's why it is not included in the calculation.

- We are not going to enable the phase correct for this exercise, so it also can be excluded from the calculation (since it is just multiplying by 1); if you enable phase correct, then the calculation will differ since you have to multiply by 2 (1+1)

Position calculation based on top

To calculate the duty cycle that corresponds to specific positions (0, 90, and 180 degrees), we use the following formula based on the top value:

```
const PWM_DIV_INT: u8 = 64;
const PWM_TOP: u16 = 46_874;

const TOP: u16 = PWM_TOP + 1;
// 0.5ms is 2.5% of 20ms; 0 degrees in servo
const MIN_DUTY: u16 = (TOP as f64 * (2.5 / 100.)) as u16;
// 1.5ms is 7.5% of 20ms; 90 degrees in servo
const HALF_DUTY: u16 = (TOP as f64 * (7.5 / 100.)) as u16;
// 2.4ms is 12% of 20ms; 180 degree in servo
const MAX_DUTY: u16 = (TOP as f64 * (12. / 100.)) as u16;
```

We multiply the TOP value by a duty cycle percentage to determine the appropriate pulse width for each position of the servo. You might need to adjust the percentage based on your servo.

Action

Setting Up the PWM and Servo Control

```
const PWM_DIV_INT: u8 = 64;
const PWM_TOP: u16 = 46_874;

const TOP: u16 = PWM_TOP + 1;
const MIN_DUTY: u16 = (TOP as f64 * (2.5 / 100.)) as u16;
const HALF_DUTY: u16 = (TOP as f64 * (7.5 / 100.)) as u16;
const MAX_DUTY: u16 = (TOP as f64 * (12. / 100.)) as u16;
```

1. Set Up the PWM Slice and Channel

First, initialize the PWM slice and channel. You should have already done similar in the previous blinky section.

```
let mut pwm_slices = hal::pwm::Slices::new(pac.PWM, &mut pac.RESETS);
let pwm = &mut pwm_slices.pwm4;
```

2. Adjust for 50HZ frequency

Now, set the divisor and the top value to achieve a PWM frequency of 50Hz.

```
pwm.set_div_int(PWM_DIV_INT);
pwm.set_div_frac(0);

pwm.set_top(PWM_TOP);
pwm.enable();
```

3. Set Output Pin

Next, specify the GPIO pin where the PWM signal will be sent. We will use GPIO pin 9.

```
    let servo = &mut pwm.channel_b;
    servo.output_to(pins.gpio9);
```

4. Set Servo Position in a Loop

Finally, in the loop, we adjust the duty cycle which will control the servo's position. We will move the servo to different positions (0°, 90°, and 180°) using the MIN_DUTY, HALF_DUTY, and MAX_DUTY values calculated earlier.

```
loop {
    servo.set_duty_cycle(MIN_DUTY).unwrap(); // 0 degrees
    timer.delay_ms(1000);

    servo.set_duty_cycle(HALF_DUTY).unwrap(); // 90 degrees
    timer.delay_ms(1000);

    servo.set_duty_cycle(MAX_DUTY).unwrap(); // 180 degrees
    timer.delay_ms(1000);
}
```

Full Code snippet

```
const PWM_DIV_INT: u8 = 64;
const PWM_TOP: u16 = 46_874;

const TOP: u16 = PWM_TOP + 1;
const MIN_DUTY: u16 = (TOP as f64 * (2.5 / 100.)) as u16;
const HALF_DUTY: u16 = (TOP as f64 * (7.5 / 100.)) as u16;
const MAX_DUTY: u16 = (TOP as f64 * (12. / 100.)) as u16;

#[hal::entry]
fn main() -> ! {
    // Grab our singleton objects
    let mut pac = hal::pac::Peripherals::take().unwrap();

    // Set up the watchdog driver - needed by the clock setup code
    let mut watchdog = hal::Watchdog::new(pac.WATCHDOG);

    // Configure the clocks
    // The default is to generate a 125 MHz system clock
    let clocks = hal::clocks::init_clocks_and_plls(
        XTAL_FREQ_HZ,
        pac.XOSC,
        pac.CLOCKS,
        pac.PLL_SYS,
        pac.PLL_USB,
        &mut pac.RESETS,
        &mut watchdog,
    )
    .ok()
    .unwrap();

    // The single-cycle I/O block controls our GPIO pins
    let sio = hal::Sio::new(pac.SIO);

    // Set the pins up according to their function on this particular
board
    let pins = hal::gpio::Pins::new(
        pac.IO_BANK0,
        pac.PADS_BANK0,
        sio.gpio_bank0,
        &mut pac.RESETS,
    );

    // The delay object lets us wait for specified amounts of time (i
    // milliseconds)
    let mut timer = hal::Timer::new_timer0(pac.TIMER0, &mut
pac.RESETS, &clocks);

    // Init PWMs
    let mut pwm_slices = hal::pwm::Slices::new(pac.PWM, &mut
```

```
pac.RESETS);

    // Configure PWM4
    let pwm = &mut pwm_slices.pwm4;

    pwm.set_div_int(PWM_DIV_INT);
    pwm.set_div_frac(0);

    pwm.set_top(PWM_TOP);
    pwm.enable();

    let servo = &mut pwm.channel_b;
    servo.output_to(pins.gpio9);

    loop {
        servo.set_duty_cycle(MIN_DUTY).unwrap();
        timer.delay_ms(1000);
        servo.set_duty_cycle(HALF_DUTY).unwrap();
        timer.delay_ms(1000);
        servo.set_duty_cycle(MAX_DUTY).unwrap();
        timer.delay_ms(1000);
    }
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `servo` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/servo
```

Buzzinga

In this section, we'll explore some fun activities using the `buzzer`. I chose the title "Buzzinga" just for fun (a nod to Sheldon's "Bazinga" in *The Big Bang Theory*); it's not a technical term.

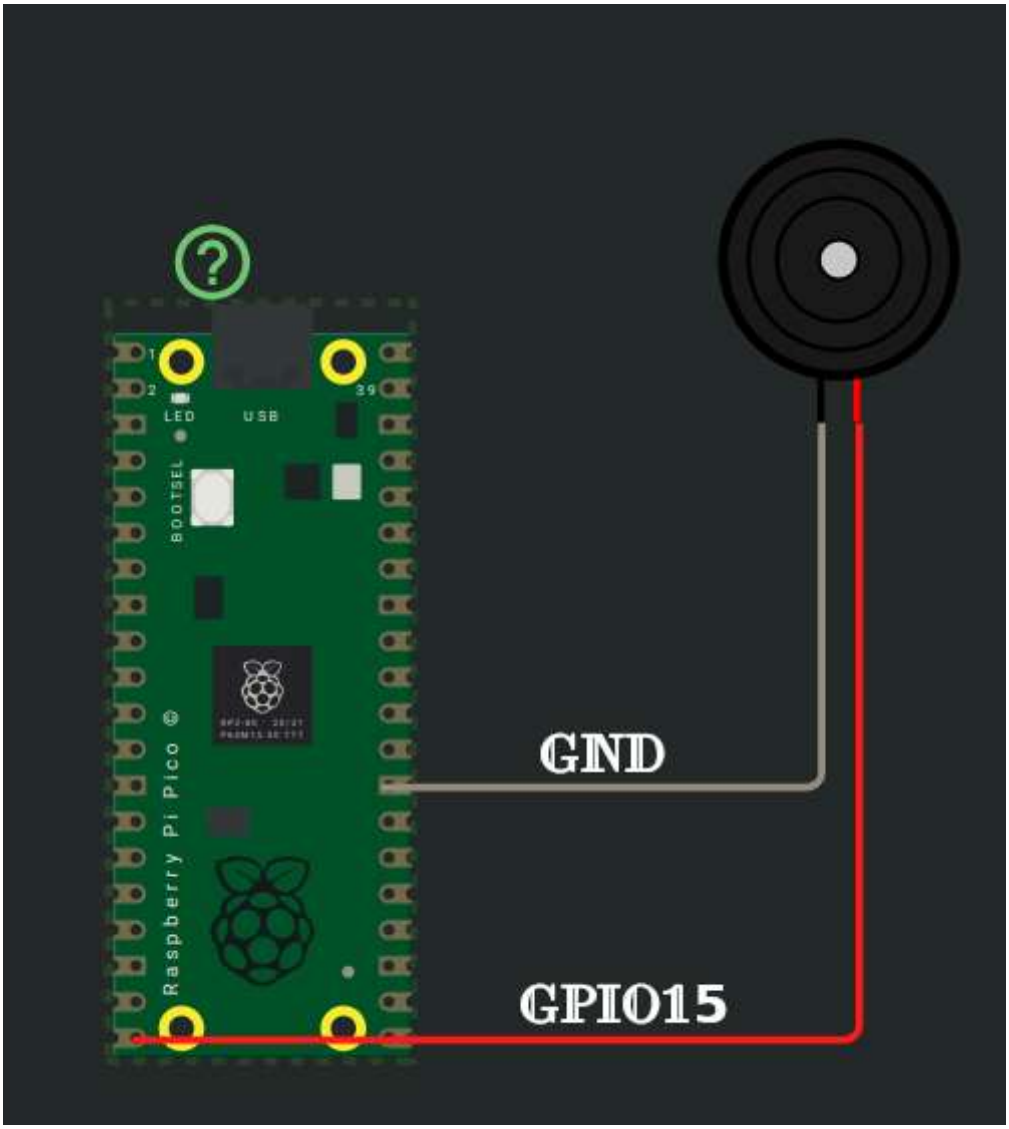
- **Passive Buzzer**
- **Jumper Wires**:
  - **Female-to-Male** jumper wires for connecting the Pico 2 to the buzzer pins (Positive and Ground).

The buzzer has two pins: Positive(Signal), Ground; The positive side of the buzzer is typically marked with a **+** symbol and is the longer pin, while the negative side (ground) is the shorter pin, similar to an LED. However, some passive buzzers may allow for either pin to be connected to ground or signal, depending on the specific model.

By the way, I used an active buzzer in my experiment. A passive buzzer is recommended if you plan to play different sounds, as it provides a better tone.

Connection Overview

Pico Pin	Wire	Buzzer Pin	Notes
GPIO 15		Positive Pin	Receives PWM signals to produce sound.
GND		Ground Pin	Connects to ground.



Before moving forward, make sure you've read the following sections and understood the concepts.

- PWM introduction in the Blink LED section
- More on PWM in the servo section
- Calculating top in the servo section

Reference

- Pico official guide on buzzer

Introduction to Buzzer

A buzzer is an electronic device used to generate sound, beeps, or even melodies, and is commonly found in alarm systems, timers, computers, and for confirming user inputs, such as mouse clicks or keystrokes. Buzzers serve as audio signaling devices, providing audible feedback for various actions.

Active Buzzer vs Passive Buzzer

Active Buzzer:

- **Built-in Oscillator**: An active buzzer has an internal oscillator that generates the tone automatically when power is applied. You can identify whether you have active buzzer or not by connecting the buzzer directly to the battery and it will make a sound.
- **Simpler Usage**: No need to worry about generating specific frequencies since the buzzer does it internally.
- **Tone**: Typically produces a single tone or a fixed frequency.

- **How to identify**: Usually has a white covering on top and a black smooth finish at the bottom. It produces sound when connected directly to a battery.

Passive Buzzer:

- **External Signal Required**: A passive buzzer requires an external signal (usually a square wave) to generate sound. It does not have an internal oscillator, so it relies on a microcontroller to provide a frequency.
- **Flexible Tones**: You can control the frequency and create different tones, melodies, or alarms based on the input signal.

- **How to identify**: Typically has no covering on the top and looks like a PCB-style blue or green covering at the bottom.

Which one ?

Choose **Active Buzzer** if:

- You need a simple, fixed tone or beep. It's ideal for basic alerts, alarms, or user input confirmation.

Choose **Passive Buzzer** if:

- You want to generate different tones, melodies, or sound patterns.

It is recommended to use a passive buzzer for our exercises. However, if you only have an active buzzer, don't worry; you can still use it. In fact, I personally used an active buzzer for this.

Beep

In this exercise, we will generate a beeping sound with a 50% duty cycle, followed by a 0% duty cycle, creating a looping pattern of sound and silence. We will use the A4 note (440Hz frequency) for this. (If you're not familiar with the A4 note, please look up more information on musical notes.)

Get Top function

In the previous exercise (servo motor), we manually calculated and hardcoded the top value. In this exercise, we create a small function to calculate the top value based on the target frequency and div_int:

```
const fn get_top(freq: f64, div_int: u8) -> u16 {
    let result = 150_000_000. / (freq * div_int as f64);
    result as u16 - 1
}
```

`div_int` value

We will be using 64 as `div_int`.

```
const PWM_DIV_INT: u8 = 64;
```

Configure the GPIO 15 pin

Next, we need to configure the GPIO pin (GPIO 15) to output the PWM signal.

```
let pwm = &mut pwm_slices.pwm7;
pwm.enable();
pwm.set_div_int(PWM_DIV_INT);
pwm.channel_b.output_to(pins.gpio15);
```

To Set a frequency 440Hz(A4 Note)

Now we calculate the top value required to generate the 440Hz frequency (A4 note) and set it for the PWM:

```
let top = get_top(440., PWM_DIV_INT);
pwm.set_top(top);
```

Loop

Finally, we create a loop to alternate between a 50% duty cycle (beep) and a 0% duty cycle (silence). The loop repeats with a delay of 500 milliseconds between each change:

```
loop {
    pwm.channel_b.set_duty_cycle_percent(50).unwrap();
    timer.delay_ms(500);
    pwm.channel_b.set_duty_cycle(0).unwrap();
    timer.delay_ms(500);
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `beep` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/beep
```

GOT Buzzer?

We are going to play the Game of Thrones (GOT) background music (BGM) on the buzzer, thanks to the awesome arduino-songs repository. It also has other bgms.

If you're unsure about musical notes and sheet music, feel free to check out the quick theory I've provideded here.

I've splitted the code into rust module(you can do it in single file as we have done so far):

`music`

,

`got`

.

Reference

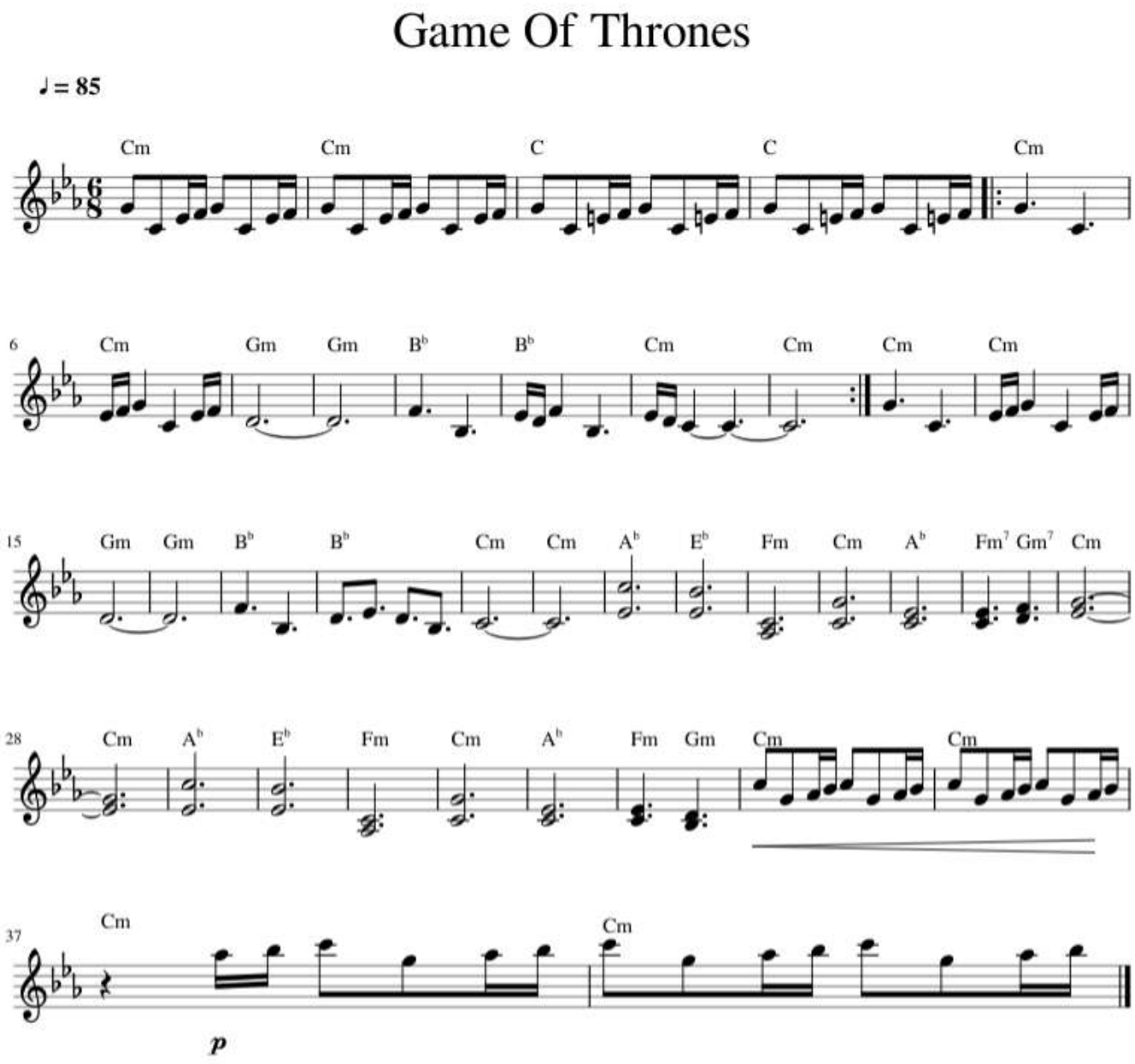
- arduino-songs repo

Introduction to Music Notes and Sheet Music

This is a brief guide to music notes and sheet music. While it may not cover everything, it provides a quick reference for key concepts.

Music Sheet

The notes for the music are based on the following sheet. You can refer to this Musescore link for more details.



In music, note durations are represented by the following types, which define how long each note is played:

- **Whole note**: The longest note duration, lasting for 4 beats.
- **Half note**: A note that lasts for 2 beats.
- **Quarter note**: A note that lasts for 1 beat.
- **Eighth note**: A note that lasts for half a beat, or 1/8th of the duration of a whole note.
- **Sixteenth note**: A note that lasts for a quarter of a beat, or 1/16th of the duration of a whole note.

Dotted Notes

A dotted note is a note that has a dot next to it. The dot increases the note's duration by half of its original value. For example:

- **Dotted half note**: A half note with a dot lasts for 3 beats (2 + 1).
- **Dotted quarter note**: A quarter note with a dot lasts for 1.5 beats (1 + 0.5).

Tempo and BPM (Beats Per Minute)

Tempo refers to the speed at which a piece of music is played. It is usually measured in **beats per minute** (BPM), indicating how many beats occur in one minute.

Music module

In the music module, we define constants for common notes and their corresponding frequency values.

```rust
// Note frequencies in Hertz as f64
pub const NOTE_B0: f64 = 31.0;
pub const NOTE_C1: f64 = 33.0;
pub const NOTE_CS1: f64 = 35.0;
pub const NOTE_D1: f64 = 37.0;
pub const NOTE_DS1: f64 = 39.0;
pub const NOTE_E1: f64 = 41.0;
pub const NOTE_F1: f64 = 44.0;
pub const NOTE_FS1: f64 = 46.0;
pub const NOTE_G1: f64 = 49.0;
pub const NOTE_GS1: f64 = 52.0;
pub const NOTE_A1: f64 = 55.0;
pub const NOTE_AS1: f64 = 58.0;
pub const NOTE_B1: f64 = 62.0;
pub const NOTE_C2: f64 = 65.0;
pub const NOTE_CS2: f64 = 69.0;
pub const NOTE_D2: f64 = 73.0;
pub const NOTE_DS2: f64 = 78.0;
pub const NOTE_E2: f64 = 82.0;
pub const NOTE_F2: f64 = 87.0;
pub const NOTE_FS2: f64 = 93.0;
pub const NOTE_G2: f64 = 98.0;
pub const NOTE_GS2: f64 = 104.0;
pub const NOTE_A2: f64 = 110.0;
pub const NOTE_AS2: f64 = 117.0;
pub const NOTE_B2: f64 = 123.0;
pub const NOTE_C3: f64 = 131.0;
pub const NOTE_CS3: f64 = 139.0;
pub const NOTE_D3: f64 = 147.0;
pub const NOTE_DS3: f64 = 156.0;
pub const NOTE_E3: f64 = 165.0;
pub const NOTE_F3: f64 = 175.0;
pub const NOTE_FS3: f64 = 185.0;
pub const NOTE_G3: f64 = 196.0;
pub const NOTE_GS3: f64 = 208.0;
pub const NOTE_A3: f64 = 220.0;
pub const NOTE_AS3: f64 = 233.0;
pub const NOTE_B3: f64 = 247.0;
pub const NOTE_C4: f64 = 262.0;
pub const NOTE_CS4: f64 = 277.0;
pub const NOTE_D4: f64 = 294.0;
pub const NOTE_DS4: f64 = 311.0;
pub const NOTE_E4: f64 = 330.0;
pub const NOTE_F4: f64 = 349.0;
pub const NOTE_FS4: f64 = 370.0;
pub const NOTE_G4: f64 = 392.0;
pub const NOTE_GS4: f64 = 415.0;
pub const NOTE_A4: f64 = 440.0;
pub const NOTE_AS4: f64 = 466.0;
pub const NOTE_B4: f64 = 494.0;
pub const NOTE_C5: f64 = 523.0;
```

```
pub const NOTE_CS5: f64 = 554.0;
pub const NOTE_D5: f64 = 587.0;
pub const NOTE_DS5: f64 = 622.0;
pub const NOTE_E5: f64 = 659.0;
pub const NOTE_F5: f64 = 698.0;
pub const NOTE_FS5: f64 = 740.0;
pub const NOTE_G5: f64 = 784.0;
pub const NOTE_GS5: f64 = 831.0;
pub const NOTE_A5: f64 = 880.0;
pub const NOTE_AS5: f64 = 932.0;
pub const NOTE_B5: f64 = 988.0;
pub const NOTE_C6: f64 = 1047.0;
pub const NOTE_CS6: f64 = 1109.0;
pub const NOTE_D6: f64 = 1175.0;
pub const NOTE_DS6: f64 = 1245.0;
pub const NOTE_E6: f64 = 1319.0;
pub const NOTE_F6: f64 = 1397.0;
pub const NOTE_FS6: f64 = 1480.0;
pub const NOTE_G6: f64 = 1568.0;
pub const NOTE_GS6: f64 = 1661.0;
pub const NOTE_A6: f64 = 1760.0;
pub const NOTE_AS6: f64 = 1865.0;
pub const NOTE_B6: f64 = 1976.0;
pub const NOTE_C7: f64 = 2093.0;
pub const NOTE_CS7: f64 = 2217.0;
pub const NOTE_D7: f64 = 2349.0;
pub const NOTE_DS7: f64 = 2489.0;
pub const NOTE_E7: f64 = 2637.0;
pub const NOTE_F7: f64 = 2794.0;
pub const NOTE_FS7: f64 = 2960.0;
pub const NOTE_G7: f64 = 3136.0;
pub const NOTE_GS7: f64 = 3322.0;
pub const NOTE_A7: f64 = 3520.0;
pub const NOTE_AS7: f64 = 3729.0;
pub const NOTE_B7: f64 = 3951.0;
pub const NOTE_C8: f64 = 4186.0;
pub const NOTE_CS8: f64 = 4435.0;
pub const NOTE_D8: f64 = 4699.0;
pub const NOTE_DS8: f64 = 4978.0;
pub const REST: f64 = 0.0; // No sound, for pauses
```

Next, we create small helper `struct` to to represent a musical Song and provide some functions to calculate note durations based on tempo.

This struct has a single field whole_note, which will store the duration of a whole note in milliseconds. The reason we store the duration in milliseconds is

that musical timing is often based on tempo (beats per minute, BPM), and we need to calculate how long each note lasts in terms of time.

```
pub struct Song {
    whole_note: u32,
}
```

The formula (60_000 * 4) / tempo as u32 calculates the duration of a whole note in milliseconds. We use 60_000 because there are 60,000 milliseconds in a minute, and we multiply by 4 because a whole note is typically equivalent to four beats.

```
impl Song {
    pub fn new(tempo: u16) -> Self {
        let whole_note = (60_000 * 4) / tempo as u32;
        Self { whole_note }
    }
```

`calc_note_duration`

The `calc_note_duration` function calculates the duration of a musical note based on its division relative to a whole note. It takes in a `divider` parameter, which can be positive or negative, and returns the duration of the note in milliseconds.

```
    pub fn calc_note_duration(&self, divider: i16) -> u32 {
        if divider > 0 {
            self.whole_note / divider as u32
        } else {
            let duration = self.whole_note / divider.unsigned_abs() as
u32;
            (duration as f64 * 1.5) as u32
        }
    }
}
```

Logic:

1. **When** `divider > 0`:
   - If the `divider` is positive, the function calculates the note's duration by dividing the duration of a whole note by the `divider`.
   - For example, if `divider = 4`, the function calculates the duration of a quarter note, which is 1/4 of a whole note.
2. **When** `divider <= 0`:
   - If the `divider` is negative, the function first converts the `divider` to a positive value using `unsigned_abs()`.
   - It divides the whole note's duration by this absolute value, then multiplies the result by **1.5** to account for **dotted notes** (e.g., dotted quarter note, dotted eighth note), which last 1.5 times the duration of a regular note.

This positive and negative logic is a custom approach (based on an Arduino example I referred to) to differentiate dotted notes. It is not related to standard musical logic.

Melody Example: Game of Thrones Theme

These section contains code snippets for the rust module `got`.

Tempo

we declare the tempo for the song(you can also change and observe the result).

```
pub const TEMPO: u16 = 85;
```

Melody Array

We define the melody of the Game of Thrones theme using the notes and durations in an array. The melody consists of tuple of note frequencies and their corresponding durations. The duration of each note is represented by an integer, where positive values represent normal notes and negative values represent dotted notes.

```rust
pub const MELODY: [(f64, i16); 92] = [
    // Game of Thrones Theme
    (NOTE_G4, 8),
    (NOTE_C4, 8),
    (NOTE_DS4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, 8),
    (NOTE_C4, 8),
    (NOTE_DS4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, 8),
    (NOTE_C4, 8),
    (NOTE_DS4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, 8),
    (NOTE_C4, 8),
    (NOTE_DS4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, 8),
    (NOTE_C4, 8),
    (NOTE_E4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, 8),
    (NOTE_C4, 8),
    (NOTE_E4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, 8),
    (NOTE_C4, 8),
    (NOTE_E4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, 8),
    (NOTE_C4, 8),
    (NOTE_E4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, -4),
    (NOTE_C4, -4),
    (NOTE_DS4, 16),
    (NOTE_F4, 16),
    (NOTE_G4, 4),
    (NOTE_C4, 4),
    (NOTE_DS4, 16),
    (NOTE_F4, 16),
    (NOTE_D4, -1),
    (NOTE_F4, -4),
    (NOTE_AS3, -4),
    (NOTE_DS4, 16),
    (NOTE_D4, 16),
    (NOTE_F4, 4),
    (NOTE_AS3, -4),
    (NOTE_DS4, 16),
    (NOTE_D4, 16),
```

```
        (NOTE_C4, -1),
        // Repeat
        (NOTE_G4, -4),
        (NOTE_C4, -4),
        (NOTE_DS4, 16),
        (NOTE_F4, 16),
        (NOTE_G4, 4),
        (NOTE_C4, 4),
        (NOTE_DS4, 16),
        (NOTE_F4, 16),
        (NOTE_D4, -1),
        (NOTE_F4, -4),
        (NOTE_AS3, -4),
        (NOTE_DS4, 16),
        (NOTE_D4, 16),
        (NOTE_F4, 4),
        (NOTE_AS3, -4),
        (NOTE_DS4, 16),
        (NOTE_D4, 16),
        (NOTE_C4, -1),
        (NOTE_G4, -4),
        (NOTE_C4, -4),
        (NOTE_DS4, 16),
        (NOTE_F4, 16),
        (NOTE_G4, 4),
        (NOTE_C4, 4),
        (NOTE_DS4, 16),
        (NOTE_F4, 16),
        (NOTE_D4, -2),
        (NOTE_F4, -4),
        (NOTE_AS3, -4),
        (NOTE_D4, -8),
        (NOTE_DS4, -8),
        (NOTE_D4, -8),
        (NOTE_AS3, -8),
        (NOTE_C4, -1),
        (NOTE_C5, -2),
        (NOTE_AS4, -2),
        (NOTE_C4, -2),
        (NOTE_G4, -2),
        (NOTE_DS4, -2),
        (NOTE_DS4, -4),
        (NOTE_F4, -4),
        (NOTE_G4, -1),
    ];
```

Playing the Game of Thrones Melody

This section demonstrates how to play the Game of Thrones melody using PWM (Pulse Width Modulation) for generating the tones and timing the note durations. The code calculates the duration of each note, sets the PWM duty cycle, and controls the timing to ensure proper pauses between notes.

Song object

This creates a new Song object using the tempo value from got::TEMPO, which is set to 85 BPM. The Song object will manage the note durations based on the tempo.

```
let song = Song::new(got::TEMPO);
```

Playing the notes

This loop iterates over the muisc-notes array. Each item in the melody is a tuple containing a note and its duration type (e.g., quarter note, eighth note).

```
for (note, duration_type) in got::MELODY {
        let top = get_top(note, PWM_DIV_INT);
        pwm.set_top(top);

        let note_duration = song.calc_note_duration(duration_type);
        let pause_duration = note_duration / 10; // 10% of
note_duration

        pwm.channel_b.set_duty_cycle_percent(50).unwrap(); // Set duty
cycle to 50% to play the note

        timer.delay_ms(note_duration - pause_duration); // Play 90%
        pwm.channel_b.set_duty_cycle(0).unwrap(); // Stop tone
        timer.delay_ms(pause_duration); // Pause for 10%
    }
```

First, we calculate the top value based on the note frequency. This sets the PWM frequency to match the target note.

Next, the `calc_note_duration` function is used to determine how long each note should be played. We also calculate the pause duration as 10% of the note duration. The 90% duration ensures that the note is played clearly, while the 10% pause creates a small gap between notes, resulting in a cleaner and more distinct melody.

Keeping the Program Running

This loop keeps the program running indefinitely, as required by the main function's signature. The main function has a -> ! return type, meaning it doesn't return.

```
loop {
    // Keep the program running
    timer.delay_ms(500);
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `got-buzzer` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/got-buzzer
```

Wokwi Project

An Arduino version of this exercise is available on the Wokwi site with the Pico board. Unfortunately, the site currently does not support coding in Rust for the

Pico. However, you can refer to this project to understand how it works.

Project Link

Beeping with an Active Buzzer

Since you already know that an active buzzer is simple to use, you can make it beep just by powering it. In this exercise, we'll make it beep with just a little code.

Hardware Requirements

- **Active Buzzer**
- **Female-to-Male** or **Male-to-Male** (depending on your setup)

We'll use the Embassy HAL for this project.

Project from template

To set up the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, give your project a name, like "active-beep" and select `embassy` as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "active-beep":
# cd active-beep
```

All you need to do is change the output pin from 25 to 15 in the template code.

```
// Active Buzzer
#[embassy_executor::main]
async fn main(_spawner: Spawner) {
    let p = embassy_rp::init(Default::default());
    let mut buzzer = Output::new(p.PIN_15, Level::Low); // Changed PIN
number to 15

    loop {
        buzzer.set_high();
        Timer::after_millis(500).await;

        buzzer.set_low();
        Timer::after_millis(500).await;
    }
}
```

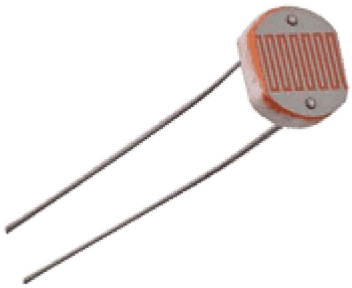
Clone the existing project

You can clone (or refer) project I created and navigate to the `active-beep` folder.

```
git clone https://github.com/ImplFerris/pico2-embassy-projects
cd pico2-embassy-projects/active-beep
```

LDR (Light Dependent Resistor)

In this section, we will use an LDR (Light Dependent Resistor) with the Raspberry Pi Pico 2. An LDR changes its resistance based on the amount of light falling on it. The brighter the light, the lower the resistance, and the dimmer the light, the higher the resistance. This makes it ideal for applications like light sensing, automatic lighting, or monitoring ambient light levels.



Components Needed:

- LDR (Light Dependent Resistor)
- Resistor (typically 10kΩ); needed to create voltage divider
- Jumper wires (as usual)

Prerequisite

To work with this, you should get familiar with what a voltage divider is and how it works. You also need to understand what ADC is and how it functions.

- Voltage divider section.
- ADC

How LDR works?

We have already given an introduction to what an LDR is. Let me repeat it again: an LDR changes its resistance based on the amount of light falling on it. The brighter the light, the lower the resistance, and the dimmer the light, the higher the resistance.

Dracula: Think of the LDR as Dracula. In sunlight, he gets weaker (just like the resistance gets lower). But in the dark, he gets stronger (just like the resistance gets higher).

We will not cover what kind of semiconductor materials are used to make an LDR, nor why it behaves this way in depth. I recommend you read this article and do further research if you are interested.

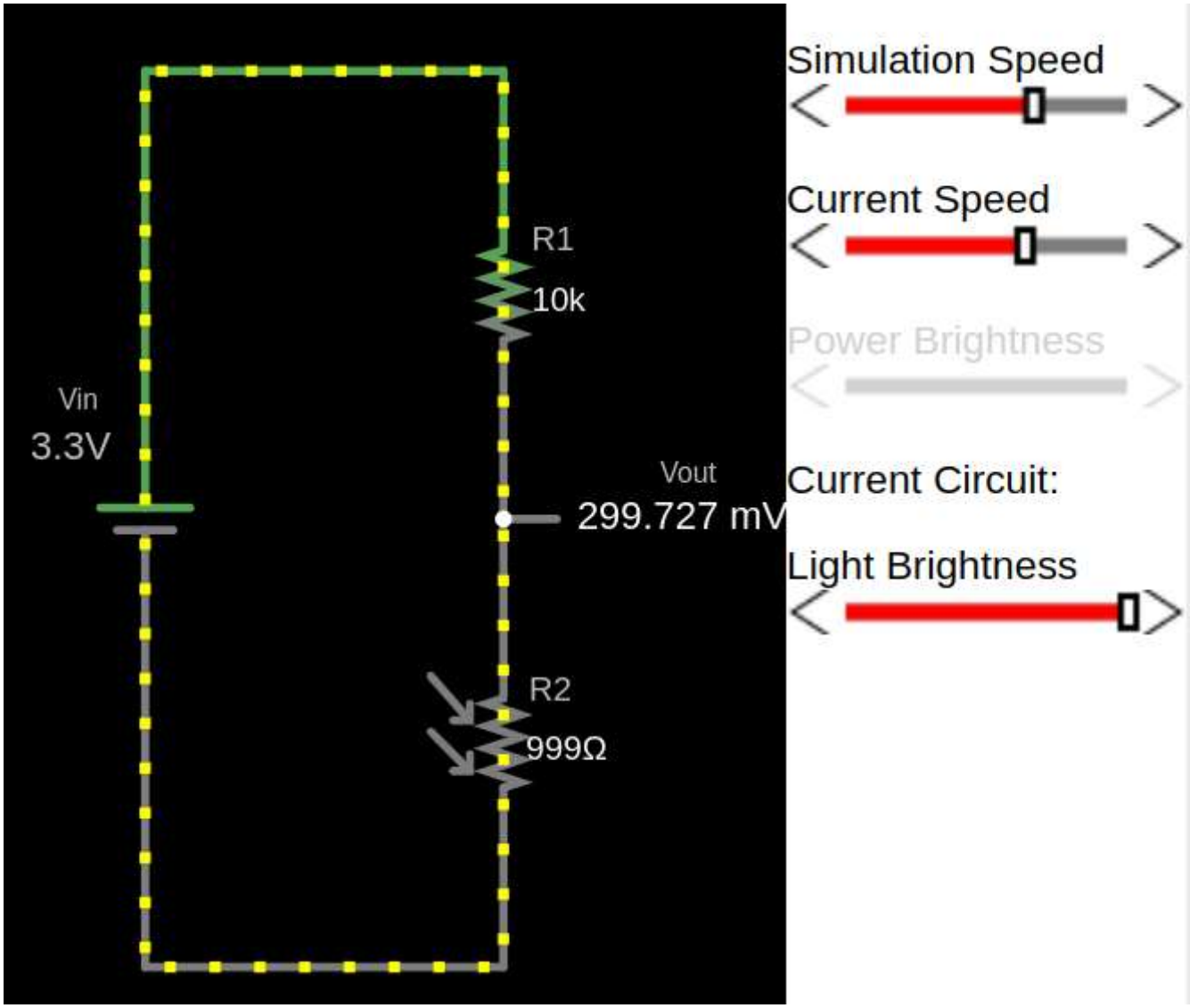
Simulation of LDR in Voltage Divider

I have created a voltage divider circuit with an LDR(a resistor symbol with arrows, kind of indicating light shining on it) in Falstad . You can import the circuit file I created,
`voltage-divider-ldr.circuitjs.txt`
, import into the Falstad site and play around.

You can adjust the brightness value and observe how the resistance of R2 (which is the LDR) changes. Also, you can watch how the V_{out} voltage changes as you increase or decrease the brightness.

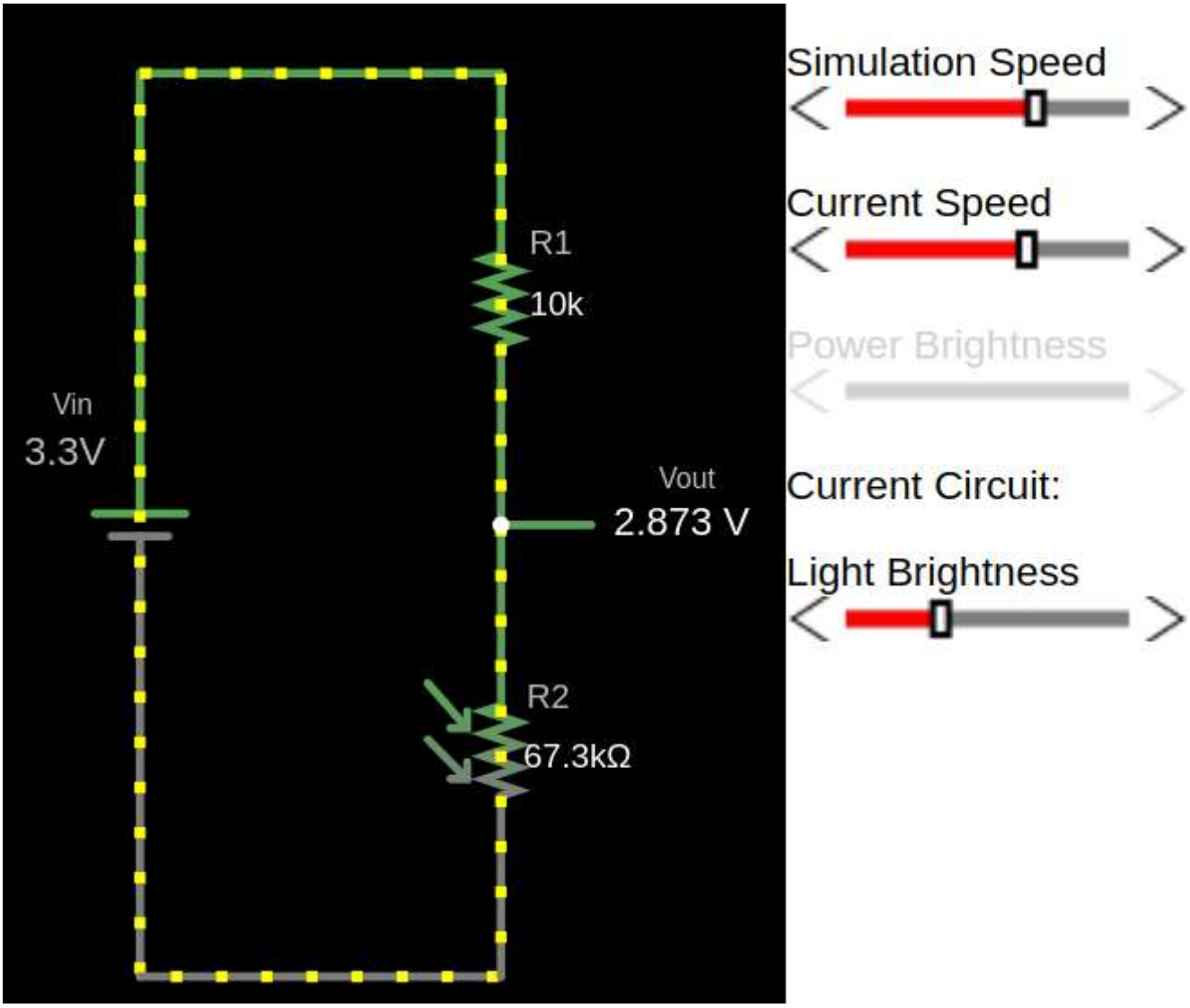
Example output for full brightness

The resistance of the LDR is low when exposed to full brightness, causing the output voltage(V_{out}) to be significantly lower.



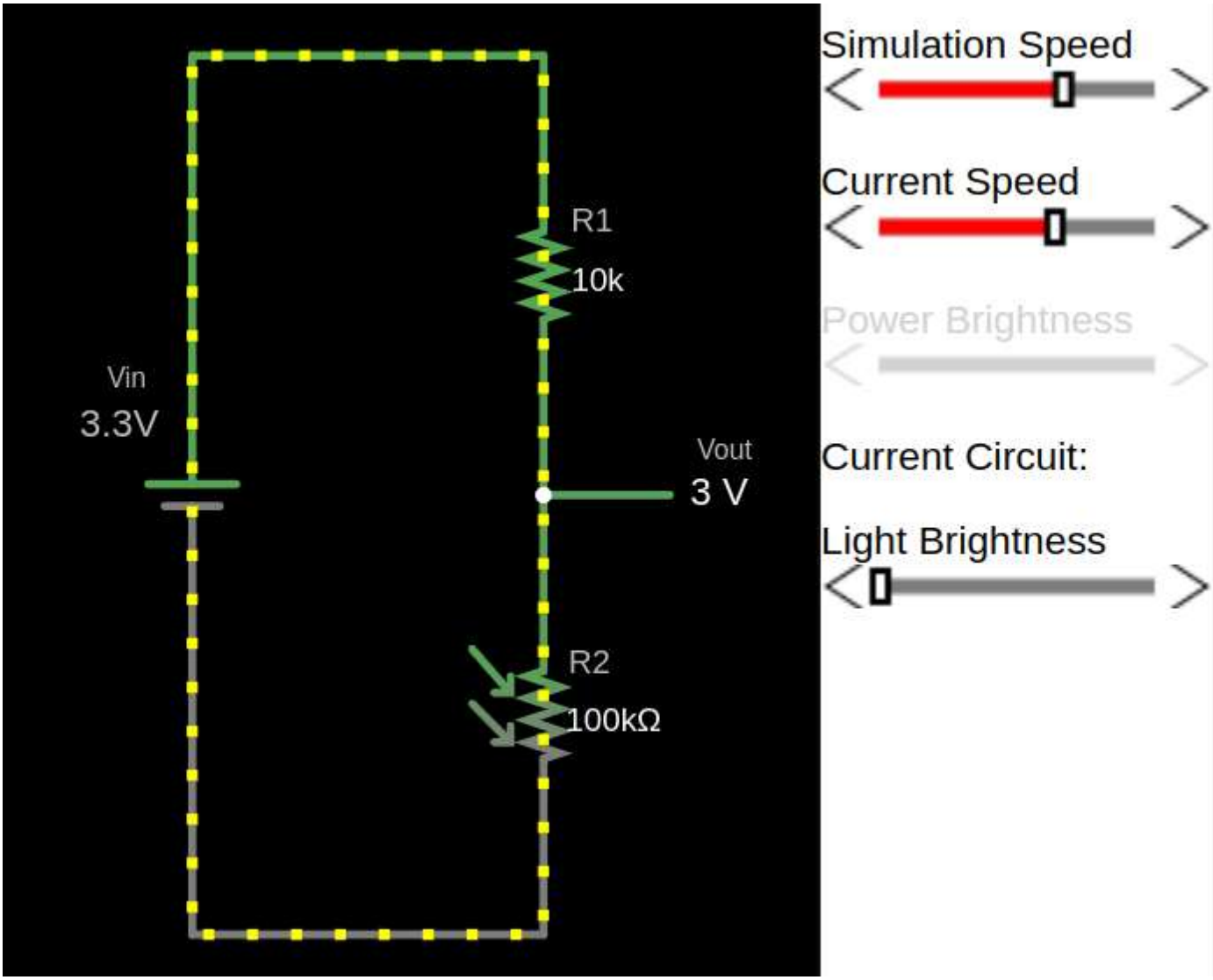
Example output for low light

With less light, the resistance of the LDR increases and the output voltage increase.



Example output for full darkness

In darkness, the LDR's resistance is high, resulting in a higher output voltage (V_{out}).


R1
10k
Vin
3.3V
Vout
3 V
R2
100kΩ
Simulation Speed
Current Speed
Power Brightness
Current Circuit:
Light Brightness

Turn on LED(or Lamp) in low Light with Pico

In this exercise, we'll control an LED based on ambient light levels. The goal is to automatically turn on the LED in low light conditions.

You can try this in a closed room by turning the room light on and off. When you turn off the room-light, the LED should turn on, given that the room is dark enough, and turn off again when the room-light is switched back on. Alternatively, you can adjust the sensitivity threshold or cover the light sensor (LDR) with your hand or some object to simulate different light levels.

Note: You may need to adjust the ADC threshold based on your room's lighting conditions and the specific LDR you are using.

Setup

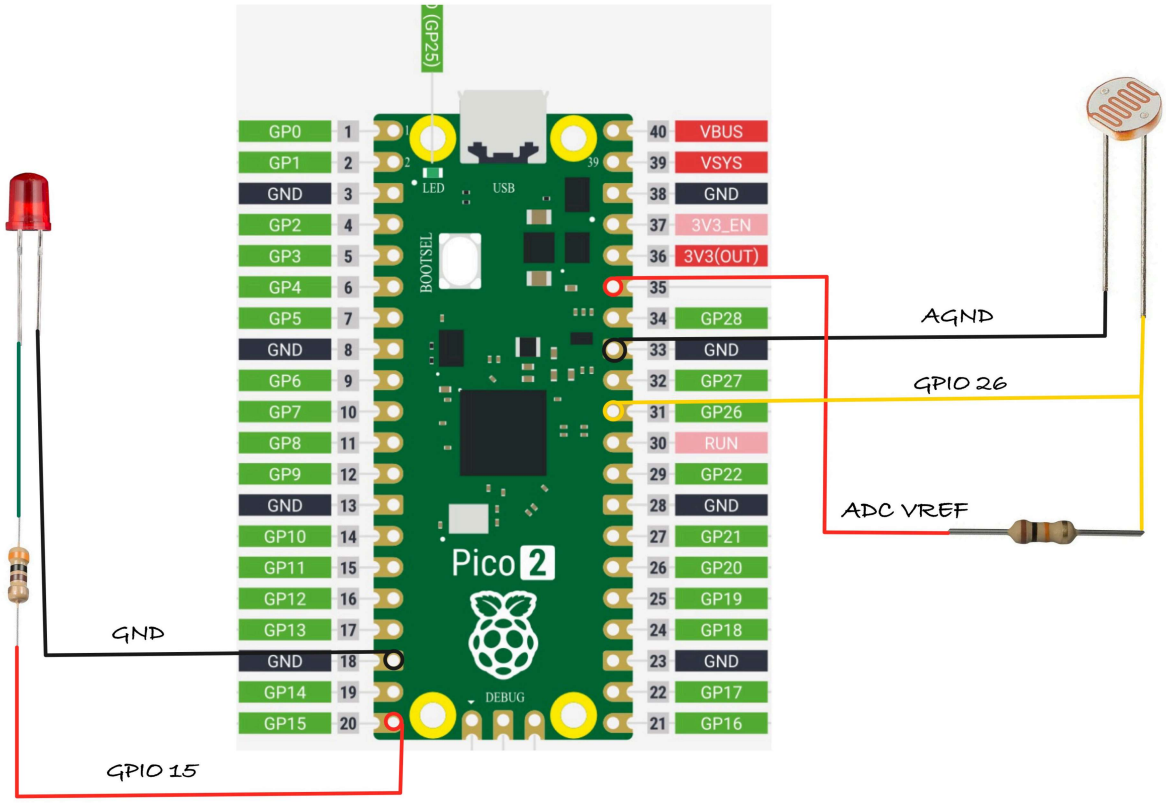
Hardware Requirements

- **LED** – Any standard LED (choose your preferred color).
- **LDR (Light Dependent Resistor)** – Used to detect light intensity.
- **Resistors**
  - **330Ω** – For the LED to limit current and prevent damage. (You might have to choose based on your LED)
  - **10kΩ** – For the LDR, forming a voltage divider in the circuit. (You might have to choose based on your LDR)
- **Jumper Wires** – For connecting components on a breadboard or microcontroller.

Circuit to connect LED, LDR with Pico

1. **One side of the LDR** is connected to **AGND** (Analog Ground).

2. The **other side of the LDR** is connected to **GPIO26 (ADC0)**, which is the analog input pin of the pico2
3. A **resistor** is connected in series with the LDR to create a voltage divider between the LDR and **ADC_VREF** (the reference voltage for the ADC).
    - From the datasheet: "ADC_VREF is the ADC power supply (and reference) voltage, and is generated on Pico 2 by filtering the 3.3V supply. This pin can be used with an external reference if better ADC performance is required"



Action

We'll use the Embassy HAL for this exercise.

Project from template

To set up the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, give your project a name, like "dracula-ldr" and select `embassy` as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "dracula-ldr":
# cd dracula-ldr
```

Interrupt Handler

Let's set up interrupt handling for the ADC.

```
use embassy_rp::adc::InterruptHandler;

bind_interrupts!(struct Irqs {
    ADC_IRQ_FIFO => InterruptHandler;
});
```

In simple terms, when the ADC completes a conversion and the result is ready, it triggers an interrupt. This tells the pico that the new data is available, so it can process the ADC value. The interrupt ensures that the pico doesn't need to constantly check the ADC, allowing it to respond only when new data is ready.

Read more about RP2350 interreupts in the datasheet (82th page).

Initialize the Embassy HAL

```
let p = embassy_rp::init(Default::default());
```

Initialize the ADC

```
let mut adc = Adc::new(p.ADC, Irqs, Config::default());
```

Configuring the ADC Pin and LED

We set up the ADC input pin (PIN_26) for reading an analog signal. Then we set up an output pin (PIN_15) to control an LED. The LED starts in the low state (Level::Low), meaning it will be off initially.

```
let mut p26 = Channel::new_pin(p.PIN_26, Pull::None);
let mut led = Output::new(p.PIN_15, Level::Low);
```

Main loop

The logic is straightforward: read the ADC value, and if it's greater than 3800, turn on the LED; otherwise, turn it off.

```
    loop {
        let level = adc.read(&mut p26).await.unwrap();
        if level > 3800 {
            led.set_high();
        } else {
            led.set_low();
        }
        Timer::after_secs(1).await;
    }
```

The full code

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

use embassy_executor::Spawner;
use embassy_rp::adc::{Adc, Channel, Config, InterruptHandler};
use embassy_rp::bind_interrupts;
use embassy_rp::block::ImageDef;
use embassy_rp::gpio::{Level, Output, Pull};
use embassy_time::Timer;
use {defmt_rtt as _, panic_probe as _};

#[link_section = ".start_block"]
#[used]
pub static IMAGE_DEF: ImageDef = ImageDef::secure_exe();

bind_interrupts!(struct Irqs {
    ADC_IRQ_FIFO => InterruptHandler;
});

#[embassy_executor::main]
async fn main(_spawner: Spawner) {
    let p = embassy_rp::init(Default::default());
    let mut adc = Adc::new(p.ADC, Irqs, Config::default());

    let mut p26 = Channel::new_pin(p.PIN_26, Pull::None);
    let mut led = Output::new(p.PIN_15, Level::Low);

    loop {
        let level = adc.read(&mut p26).await.unwrap();
        if level > 3800 {
            led.set_high();
        } else {
            led.set_low();
        }
        Timer::after_secs(1).await;
    }
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `dracula-ldr` folder.

```
git clone https://github.com/ImplFerris/pico2-embassy-projects
cd pico2-embassy-projects/dracula-ldr/
```

Thermistor

In this section, we'll be using a thermistor with the Raspberry Pi Pico. A thermistor is a variable resistor that changes its resistance based on the temperature. The amount of change in resistance depends on its composition. The term comes from combining "thermal" and "resistor.".

Thermistors are categorized into two types:

- NTC (Negative Temperature Coefficient):
  - Resistance decreases as temperature increases.
  - They are primarily used for temperature sensing and inrush current limiting.
  - We'll be using the NTC thermistor to measure temperature in our exercise.

- PTC (Positive Temperature Coefficient):
  - Resistance increases as temperature rises.
  - They primarily protect against overcurrent and overtemperature conditions as resettable fuses and are commonly used in air conditioners, medical devices, battery chargers, and welding equipment.

Reference

- Thermistor Basics
- Thermistors

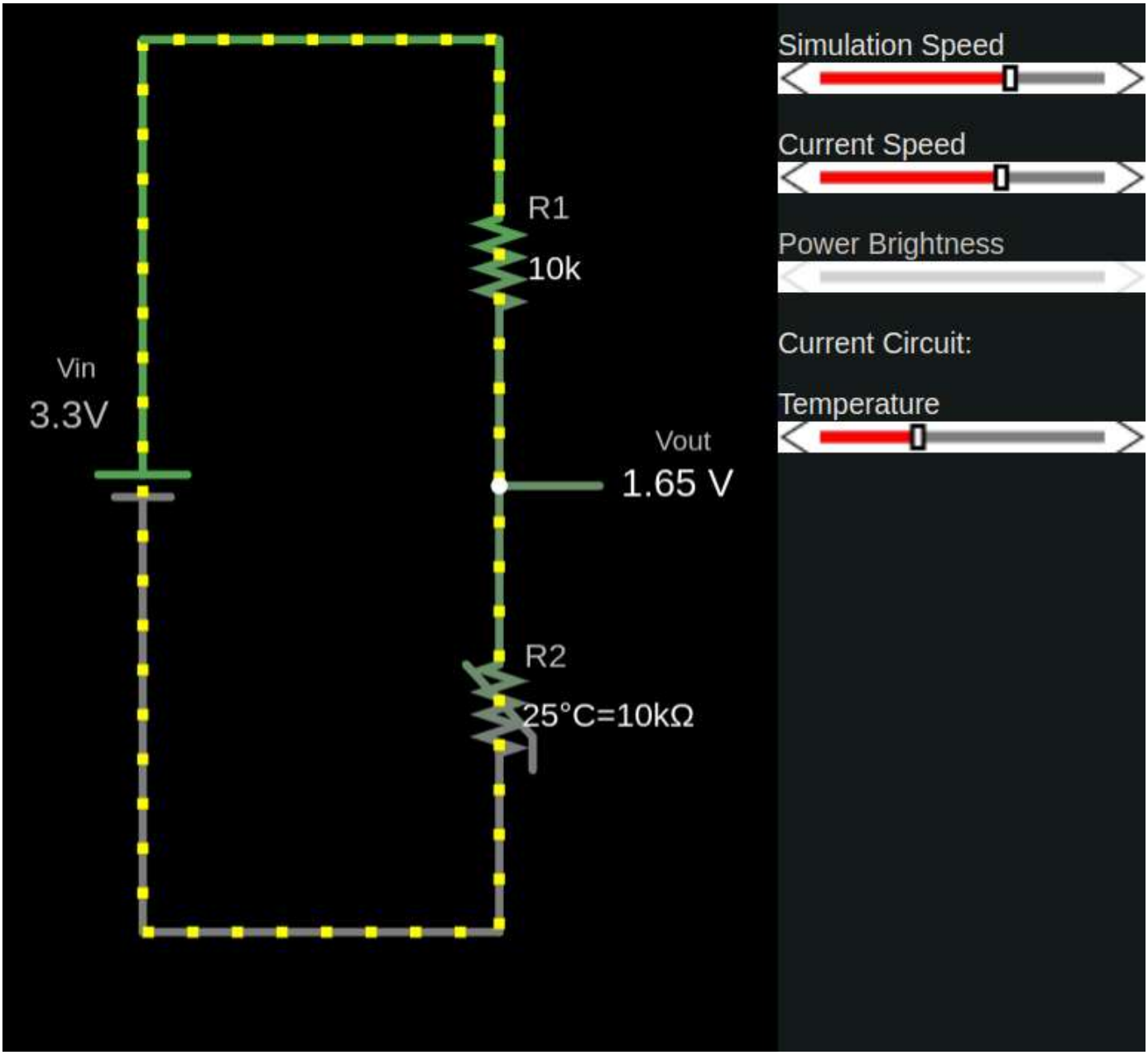
NTC and Voltage Divider

I have created a circuit on the Falstad website, and you can download the voltage-divider-thermistor.circuitjs.txt ile to import and experiment with. This setup is similar to what we covered in the voltage divider chapter of the LDR section. If you haven't gone through that section, I highly recommend completing the theory there before continuing.

This circuit includes a 10kΩ thermistor with a resistance of 10kΩ at 25°C. The input voltage V_{in} is set to 3.3V.

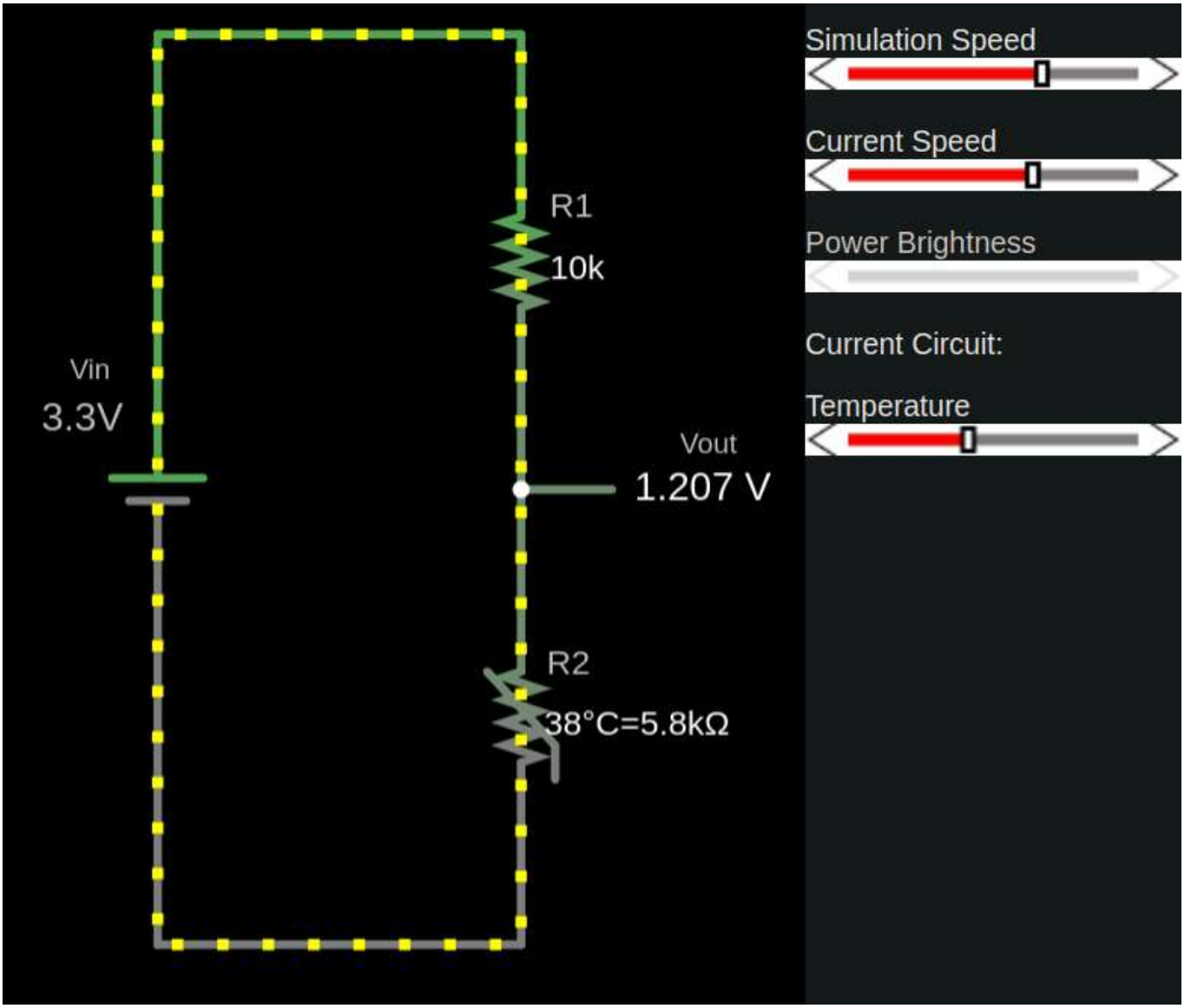
Themistor at 25°C

The thermistor has a resistance of 10kΩ at 25°C, resulting in an output voltage (V_{out}) of 1.65V.



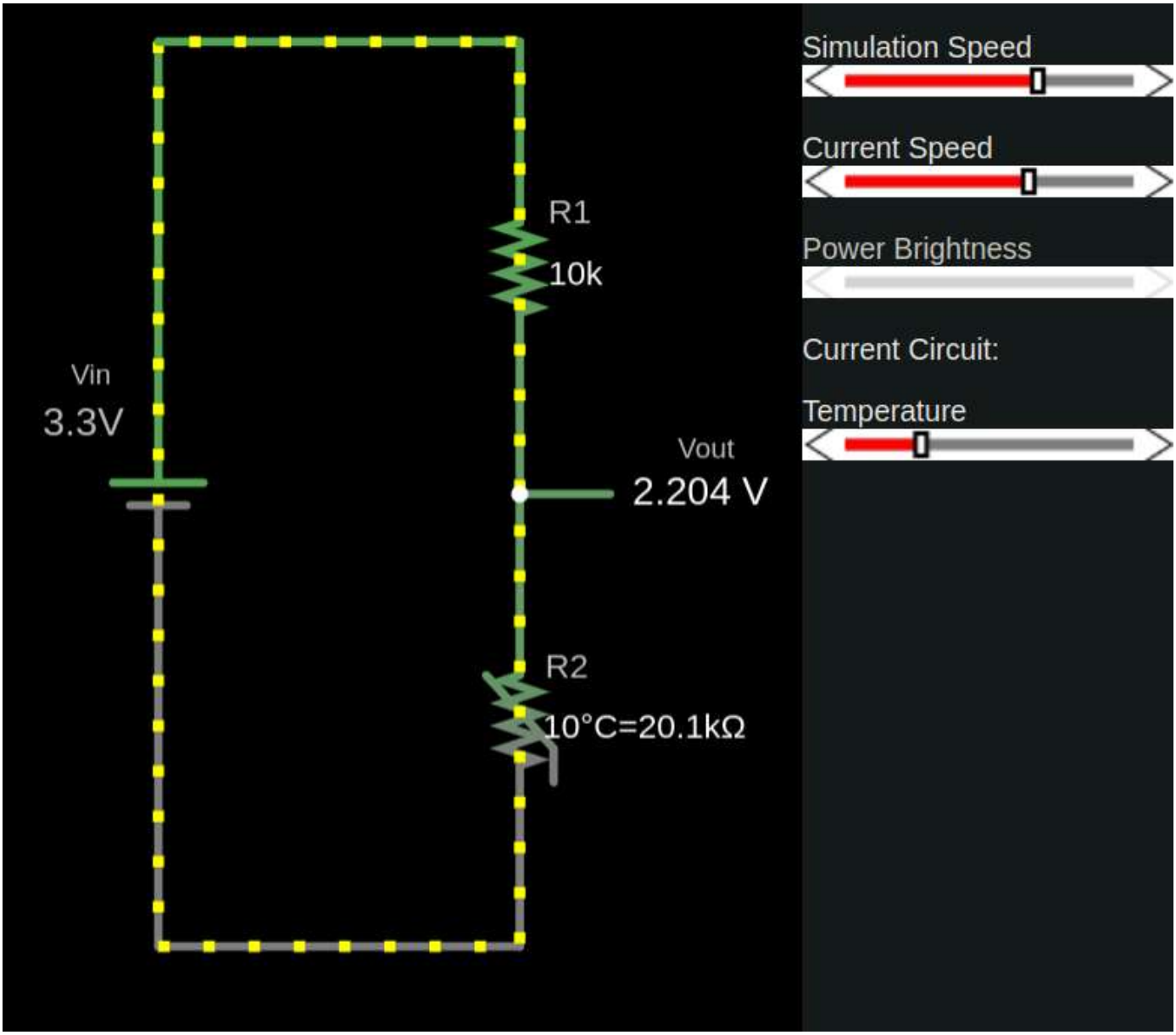
Thermistor at 38°C

The thermistor's resistance decreases due to its negative temperature coefficient, altering the voltage divider's output.



Thermistor at 10°C

The thermistor's resistance increases, resulting in a higher output voltage (V_{out}).


Simulation Speed
Current Speed
Power Brightness
Current Circuit:
Temperature
R1
10k
Vin
3.3V
Vout
2.204 V
R2
10°C=20.1kΩ

ADC

When setting up the thermistor with the Pico, we don't get the voltage directly. Instead, we receive an ADC value (refer to the ADC explanation in the LDR section). In the LDR exercise, we didn't calculate the resistance corresponding to the ADC value since we only needed to check whether the ADC value increased. However, in this exercise, to determine the temperature, we must convert the ADC value into resistence.

ADC to Resistance

We need resistance value from the adc value for the thermistor temperature calculation(that will be discussed in the next chapters).

We will use this formula to calculate the resistance value from the ADC reading. If you need how it is derived, refer the Deriving Resistance from ADC Value.

$$R_2 = \frac{R_1}{\left(\frac{\text{ADC_MAX}}{\text{adc_value}} - 1\right)}$$

Note: If you connected the thermistor to power supply instead of GND. You will need opposite. since thermistor becomes R1.

$$R_1 = R_2 \times \left(\frac{\text{ADC_MAX}}{\text{adc_value}} - 1\right)$$

Where:

- R2: The resistance based on the ADC value.
- R1: Reference resistor value (typically 10kΩ)
- ADC_MAX: The maximum ADC value is 4095 (2^{12} -1) for a 12-bit ADC
- adc_value: ADC reading (a value between 0 and ADC_MAX).

Rust Function

```rust
const ADC_MAX: u16 = 4095;
const REF_RES: f64 = 10_000.0;

fn adc_to_resistance(adc_value: u16, ref_res:f64) -> f64 {
    let x: f64 = (ADC_MAX as f64/adc_value as f64)  - 1.0;
    // ref_res * x // If you connected thermistor to power supply
    ref_res / x
}

fn main() {
    let adc_value = 2000; // Our example ADC value;

    let r2 = adc_to_resistance(adc_value, REF_RES);
    println!("Calculated Resistance (R2): {} Ω", r2);
}
```

Derivations

You can skip this section if you'd like. It simply explains the math behind deriving the resistance from the ADC value.

ADC to Voltage

The formula to convert an ADC value to voltage is:

$$V_{\text{out}} = V_{in} \times \frac{\text{adc_value}}{\text{adc_max}}$$

Where:

- **adc_value**: The value read from the ADC.
- **v_in**: The reference input voltage (3.3V for the Pico).
- **adc_max**: The maximum ADC value is 4095 (2^{12} -1) for a 12-bit ADC.

Deriving Resistance from ADC Value

We combine the voltage divider formula with ADC Resolution formula to find the Resistance(R2).

Note: It is assumed here that one side of the thermistor is connected to Ground (GND). I noticed that some online articles do the opposite, connecting one side of the thermistor to the power supply instead, which initially caused me some confusion.

Votlage Divider Formula

$$V_{out} = V_{in} \times \frac{R_2}{R_1 + R_2}$$

Step 1:

We can substitue the Vout and make derive it like this

$$V_{in} \times \frac{\text{adc_value}}{\text{adc_max}} = V_{in} \times \frac{R_2}{R_1 + R_2}$$

$$\cancel{V_{in}} \times \frac{\text{adc_value}}{\text{adc_max}} = \cancel{V_{in}} \times \frac{R_2}{R_1 + R_2}$$

Step 2:

Lets temperoarily assign the adc_value/adc_max to x for ease of derivation and finally subsitue

$$x = \frac{\text{adc_value}}{\text{adc_max}}$$

Substituting x into the equation:

$$x = \frac{R_2}{R_1 + R_2}$$

Rearrange to Solve R_2

$$R_2 = x \times (R_1 + R_2)$$

Expand the right-hand side:

$$R_2 = x \times R_1 + x \times R_2$$

Rearrange to isolate R_2 terms:

$$R_2 - x \times R_2 = R_1 \times x$$

$$R_2 \times (1 - x) = R_1 \times x$$

$$R_2 = R_1 \times \frac{x}{1 - x}$$

$$R_2 = R_1 \times \frac{1}{\left(\frac{1}{x} - 1\right)}$$

Step 3

Let's subsitute the x value back. We need 1/x, lets convert it.

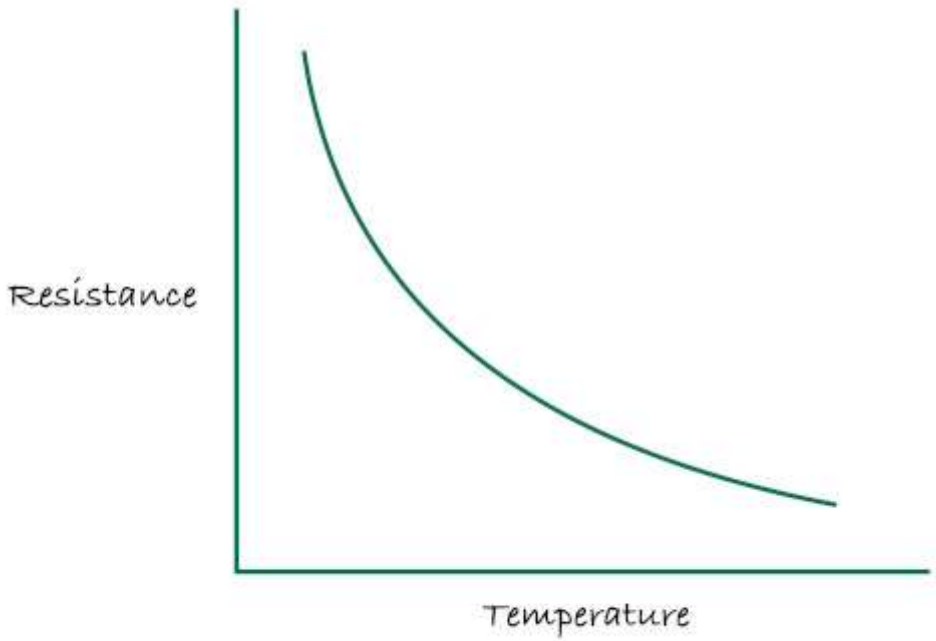
$$\frac{1}{x} = \frac{\text{adc_max}}{\text{adc_value}}$$

Final Formula

$$R_2 = R_1 \times \frac{1}{\left(\frac{\text{adc_max}}{\text{adc_value}} - 1\right)}$$

Non-Linear

Thermistors have a non-linear relationship between resistance and temperature, meaning that as the temperature changes, the resistance doesn't change in a straight-line pattern. The behavior of thermistors can be described using the Steinhart-Hart equation or the B equation.



The B equation is simple to calculate using the B value, which you can easily find online. On the other hand, the Steinhart equation uses A, B, and C coefficients. Some manufacturers provide these coefficients, but you'll still need to calibrate and find them yourself since the whole reason for using the Steinhart equation is to get accurate temperature readings.

In the next chapters, we will see in detail how to use B equation and Steinhart-Hart equation to determine the temperature.

Referemce

- The B parameter vs. Steinhart-Hart equation

- Characterising Thermistors – A Quick Primer, Beta Value & Steinhart-Hart Coefficients

B Equation

The B equation is simpler but less precise.

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{B}\ln\left(\frac{R}{R_0}\right)$$

Where:

- T is the temperature in **Kelvin**.
- T_0 is the reference temperature (usually 298.15K or 25°C), where the thermistor's resistance is known (typically 10kΩ).
- R is the **resistance** at temperature T.
- R_0 is the **resistance** at the reference temperature T_0 (often 10kΩ).
- B is the **B-value** of the thermistor.

The B value is a constant usually provided by the manufacturers, changes based on the material of a thermistor. It describes the gradient of the resistive curve over a specific temperature range between two points(i.e T_0 vs R_0 and T vs R). You can even rewrite the above formula to get B value yourself by calibrating the resistance at two temperatures.

Example Calculation:

Given:

- Reference temperature $T_0 = 298.15K$ (i.e., 25°C + 273.15 to convert to Kelvin)
- Reference resistance $R_0 = 10k\Omega$
- B-value B = 3950 (typical for many thermistors)
- Measured resistance at temperature T: 10475Ω

Step 1: Apply the B-parameter equation

Substitute the given values:

$$\frac{1}{T} = \frac{1}{298.15} + \frac{1}{3950}\ln\left(\frac{10,475}{10,000}\right)$$

$$\frac{1}{T} = 0.003354016 + \frac{1}{3950}\ln(1.0475)$$

$$\frac{1}{T} = 0.003354016 + (0.000011748)$$

$$\frac{1}{T} = 0.003365764$$

Step 2: Calculate the temperature (T)

$$T = \frac{1}{0.003365764} = 297.10936358(Kelvin)$$

Convert to Celsius:

$$T_{Celsius} = 297.10936358 - 273.15 \approx 23.95936358^{\circ}C$$

Result:

The temperature corresponding to a resistance of 10475Ω is approximately **23.96°C**.

Rust function

```rust
fn calculate_temperature(current_res: f64, ref_res: f64, ref_temp:
f64, b_val: f64) -> f64 {
    let ln_value = (current_res/ref_res).ln();
    // let ln_value = libm::log(current_res / ref_res); // use this
crate for no_std
    let inv_t = (1.0 / ref_temp) + ((1.0 / b_val) * ln_value);
    1.0 / inv_t
}

fn kelvin_to_celsius(kelvin: f64) -> f64 {
    kelvin -  273.15
}

fn celsius_to_kelvin(celsius: f64) -> f64 {
    celsius + 273.15
}

const B_VALUE: f64 = 3950.0;
const V_IN: f64 = 3.3; // Input voltage
const REF_RES: f64 = 10_000.0; // Reference resistance in ohms (10kΩ)
const REF_TEMP: f64 = 25.0;  // Reference temperature 25°C

fn main() {
    let t0 = celsius_to_kelvin(REF_TEMP);
    let r = 9546.0; // Measured resistance in ohms

    let temperature_kelvin = calculate_temperature(r, REF_RES, t0,
B_VALUE);
    let temperature_celsius = kelvin_to_celsius(temperature_kelvin);
    println!("Temperature: {:.2} °C", temperature_celsius);
}
```

Steinhart Hart equation

The Steinhart-Hart equation provides a more accurate temperature-resistance relationship over a wide temperature range.

$$\frac{1}{T} = A + B \ln R + C(\ln R)^3$$

Where:

- T is the temperature in **Kelvins**. (Formula to calculate kelvin from degree Celsius, K = °C + 273.15)
- R is the resistance at temperature T in **Ohms**.
- A, B, and C are constants specific to the thermistor's material, often provided by the manufacturer. For better accuracy, you may need to calibrate and determine these values yourself. Some datasheets provide resistance values at various temperatures, which can also be used to calculate this.

Calibration

To determine the accurate values for A, B, and C, place the thermistor in three temperature conditions: room temperature, ice water, and boiling water. For each condition, measure the thermistor's resistance using the ADC value and use a reliable thermometer to record the actual temperature. Using the resistance values and corresponding temperatures, calculate the coefficients:

- Assign A to the ice water temperature,
- B to the room temperature, and
- C to the boiling water temperature.

Calculating Steinhart-Hart Coefficients

With three resistance and temperature data points, we can find the A, B and C.

$$\begin{bmatrix} 1 & \ln R_1 & \ln^3 R_1 \\ 1 & \ln R_2 & \ln^3 R_2 \\ 1 & \ln R_3 & \ln^3 R_3 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \frac{1}{T_1} \\ \frac{1}{T_2} \\ \frac{1}{T_3} \end{bmatrix}$$

Where:

- R_1, R_2, R_3 are the resistance values at temperatures T_1, T_2, T_3.

Let's calculate the coefficients

Compute the natural logarithms of resistances:

$$L_1 = \ln R_1, \quad L_2 = \ln R_2, \quad L_3 = \ln R_3$$

Intermediate calculations:

$$Y_1 = \frac{1}{T_1}, \quad Y_2 = \frac{1}{T_2}, \quad Y_3 = \frac{1}{T_3}$$

$$\gamma_2 = \frac{Y_2 - Y_1}{L_2 - L_1}, \quad \gamma_3 = \frac{Y_3 - Y_1}{L_3 - L_1}$$

So, finally:

$$C = \left(\frac{\gamma_3 - \gamma_2}{L_3 - L_2} \right) (L_1 + L_2 + L_3)^{-1}$$

$$B = \gamma_2 - C \left(L_1^2 + L_1 L_2 + L_2^2 \right)$$

$$A = Y_1 - \left(B + L_1^2 C \right) L_1$$

Good news, Everyone! You don't need to calculate the coefficients manually. Simply provide the resistance and temperature values for cold, room, and hot environments, and use the form below to determine A, B and C

ADC value and Resistance Calculation

Note: if you already have the temperature and corresponding resistance, you can directly use the second table to input those values.

If you have the ADC value and want to calculate the resistance, use this table to find the corresponding resistance at different temperatures. As you enter the ADC value for each temperature, the calculated resistance will be automatically updated in the second table.

To perform this calculation, you'll need the base resistance of the thermistor, which is essential for determining the resistance at a given temperature based on the ADC value.

Please note that the ADC bits may need to be adjusted if you're using a different microcontroller. In our case, for the the Raspberry Pi Pico, the ADC resolution is 12 bits.

Base Resistance (Ω): 10000
ADC Bits: 12

Environment	ADC value
Cold Water	
Room Temperature	
Boiling Water	

Coefficients Finder

Adjust the temperature by entering a value in either Fahrenheit or Celsius; the form will automatically convert it to the other format. Provide the resistance corresponding to each temperature, and then click the "Calculate Coefficients" button.

Environment	Resistance (Ohms)	Temperature (°F)	Tempe
Cold Water	25000	41,00	5
Room Temperature	10000	77,00	25
Boiling Water	4000	113,00	45

Results

A: 2.10850817e-3 (0.0021085081731127)

B: 7.97920473e-5 (0.0000797920472678)

C: 6.53507631e-7 (0.0000006535076315)

Calculate Coefficients

Calculate Temperature from Resistance

Now, with these coefficients, you can calculate the temperature for any given resistance:

R (Ohms): 10000

Calculate Temperature

Result (°C):

Result (°F):

Rust function

```
fn steinhart_temp_calc(
    resistance: f64, // Resistance in Ohms
    a: f64,          // Coefficient A
    b: f64,          // Coefficient B
    c: f64,          // Coefficient C
) -> Result<(f64, f64), String> {
    if resistance <= 0.0 {
        return Err("Resistance must be a positive
number.".to_string());
    }

    // Calculate temperature in Kelvin using Steinhart-Hart equation:
    // 1/T = A + B*ln(R) + C*(ln(R))^3
    let ln_r = resistance.ln();
    let inverse_temperature = a + b * ln_r + c * ln_r.powi(3);

    if inverse_temperature == 0.0 {
        return Err("Invalid coefficients or resistance leading to
division by zero.".to_string());
    }

    let temperature_kelvin = 1.0 / inverse_temperature;

    let temperature_celsius = temperature_kelvin - 273.15;
    let temperature_fahrenheit = (temperature_celsius * 9.0 / 5.0) +
32.0;

    Ok((temperature_celsius, temperature_fahrenheit))
}

fn main() {
    // Example inputs
     let a = 2.10850817e-3;
    let b = 7.97920473e-5;
    let c = 6.53507631e-7;
    let resistance = 10000.0;


    match steinhart_temp_calc(resistance, a, b, c) {
        Ok((celsius, fahrenheit)) => {
            println!("Temperature in Celsius: {:.2}", celsius);
            println!("Temperature in Fahrenheit: {:.2}", fahrenheit);
        }
        Err(e) => println!("Error: {}", e),
    }
}
```

Referemce

- Thermistor Calculator
- Thermistor Steinhart-Hart Coefficients for Calculating Motor Temperature
- Calibrate Steinhart-Hart Coefficients for Thermistors
- Cooking Thermometer With Steinhart-Hart Correction

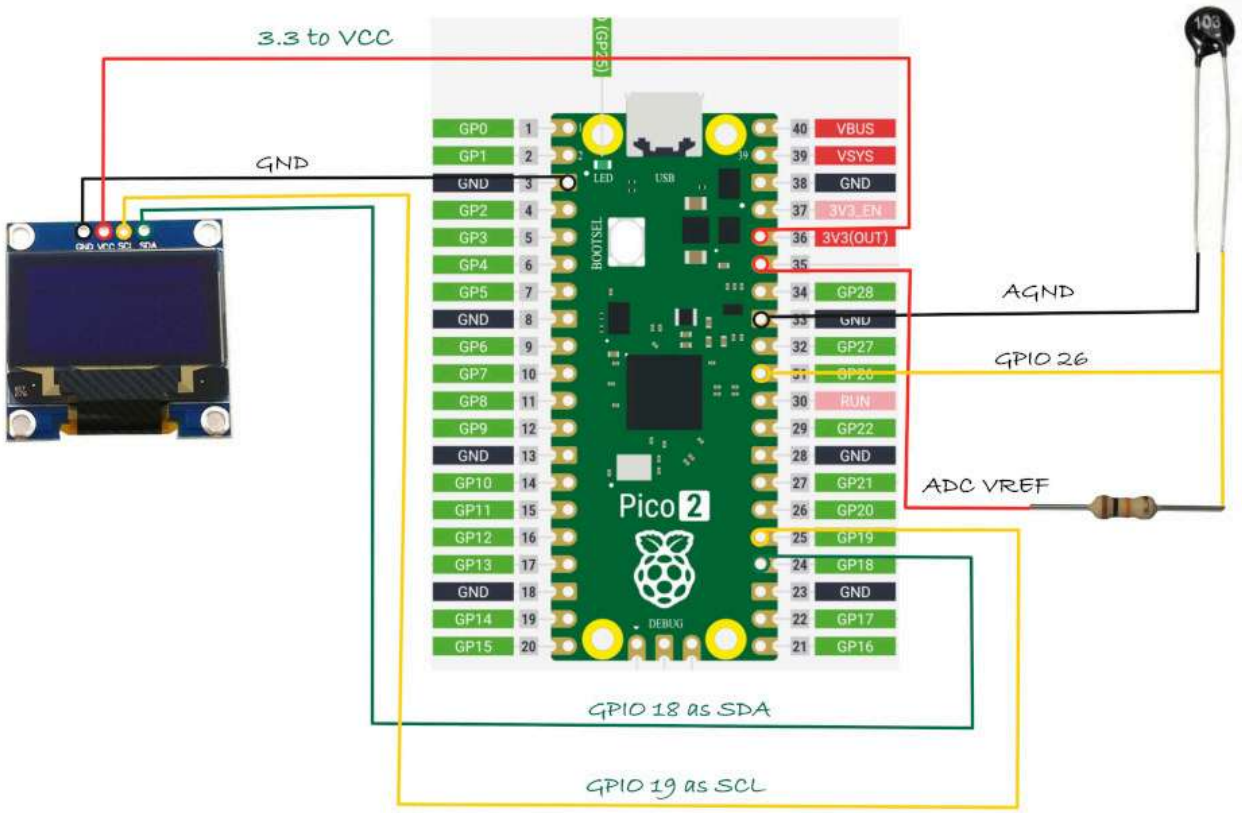
Temperature on OLED

In this section, we will measure the temperature in your room and display it on the OLED screen.

Hardware Requirments

- An OLED display: (0.96 Inch I2C/IIC 4-Pin, 128x64 resolution, SSD1306 chip)
- Jumper wires
- NTC 103 Thermistor: 10K OHM, 5mm epoxy coated disc
- 10kΩ Resistor: Used with the thermistor to form a voltage divider

Circuit to connect OLED, Thermistor with Raspberry Pi Pico



1. **One side of the Thermistor** is connected to **AGND** (Analog Ground).
2. The **other side of the Thermistor** is connected to **GPIO26 (ADC0)**, which is the analog input pin of the pico2
3. A **resistor** is connected in series with the Thermistor to create a voltage divider between the Thermistor and **ADC_VREF** (the reference voltage for the ADC).

Note:Here, one side of the thermistor is connected to ground, as shown. If you've connected it to the power supply instead, you'll need to use the alternate formula mentioned earlier.

The Flow

- We read the ADC value
- Get resisance value from ADC value
- Calculate temperature using B parameter equation
- Display the ADC, Resistance, Temperature(in Celsius) in the OLED

Action

We'll use the Embassy HAL for this exercise.

Project from template

To set up the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, give your project a name, like "thermistor" and select `embassy` as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "thermistor":
# cd thermistor
```

Additional Crates required

Update your Cargo.toml to add these additional crate along with the existing dependencies.

```
ssd1306 = "0.9.0"
heapless = "0.8.0"
libm = "0.2.11"
```

- `ssd1306`: Driver for controlling SSD1306 OLED display.
- `heapless`: In a `no_std` environment, Rust's standard

`String`
type (which requires heap allocation) is unavailable. This provides stack-allocated, fixed-size data structures. We will be using to store dynamic text, such as ADC, resistance, and temperature values, for display on the OLED screen

- `libm`
  : Provides essential mathematical functions for embedded environments. We need this to calculate natural logarithm.

Additional imports

```
use heapless::String;
use ssd1306::mode::DisplayConfig;
use ssd1306::prelude::DisplayRotation;
use ssd1306::size::DisplaySize128x64;
use ssd1306::{I2CDisplayInterface, Ssd1306};

use embassy_rp::adc::{Adc, Channel};
use embassy_rp::peripherals::I2C1;
use embassy_rp::{adc, bind_interrupts, i2c};

use embassy_rp::gpio::Pull;

use core::fmt::Write;
```

Interrupt Handler

We have set up only the ADC interrupt handler for the LDR exercises so far. For this exercise, we also need to set up an interrupt handler for I2C to enable communication with the OLED display.

```
bind_interrupts!(struct Irqs {
    ADC_IRQ_FIFO => adc::InterruptHandler;
    I2C1_IRQ => i2c::InterruptHandler<I2C1>;
});
```

ADC related functions

We can hardcode 4095 for the Pico, but here's a simple function to calculate ADC_MAX based on ADC bits:

```
const fn calculate_adc_max(adc_bits: u8) -> u16 {
    (1 << adc_bits) - 1
}
const ADC_BITS: u8 = 12; // 12-bit ADC in Pico
const ADC_MAX: u16 = calculate_adc_max(ADC_BITS); // 4095 for 12-bit
ADC
```

Thermistor specific values

The thermistor I'm using has a 10kΩ resistance at 25°C and a B value of 3950.

```
const B_VALUE: f64 = 3950.0;
const REF_RES: f64 = 10_000.0; // Reference resistance in ohms (10kΩ)
const REF_TEMP: f64 = 25.0; // Reference temperature 25°C
```

Helper functions

```
// We have already covered about this formula in ADC chpater
fn adc_to_resistance(adc_value: u16, ref_res: f64) -> f64 {
    let x: f64 = (ADC_MAX as f64 / adc_value as f64) - 1.0;
    // ref_res * x // If you connected thermistor to power supply
    ref_res / x
}

// B Equation to convert resistance to temperature
fn calculate_temperature(current_res: f64, ref_res: f64, ref_temp:
f64, b_val: f64) -> f64 {
    let ln_value = libm::log(current_res / ref_res); // Use libm for
`no_std`
    let inv_t = (1.0 / ref_temp) + ((1.0 / b_val) * ln_value);
    1.0 / inv_t
}

fn kelvin_to_celsius(kelvin: f64) -> f64 {
    kelvin - 273.15
}

fn celsius_to_kelvin(celsius: f64) -> f64 {
    celsius + 273.15
}
```

Base setups

First, we set up the Embassy HAL, configure the ADC on GPIO 26, and prepare the I2C interface for communication with the OLED display

```
let p = embassy_rp::init(Default::default());
// ADC to read the Vout value
let mut adc = Adc::new(p.ADC, Irqs, adc::Config::default());
let mut p26 = Channel::new_pin(p.PIN_26, Pull::None);

// Setting up I2C send text to OLED display
let sda = p.PIN_18;
let scl = p.PIN_19;
let i2c = i2c::I2c::new_async(p.I2C1, scl, sda, Irqs,
i2c::Config::default());
let interface = I2CDisplayInterface::new(i2c);
```

Setting Up an SSD1306 OLED Display in Terminal Mode

Next, create a display instance, specifying the display size and orientation. And enable terminal mode.

```
let mut display =
    Ssd1306::new(interface, DisplaySize128x64,
DisplayRotation::Rotate0).into_terminal_mode();
display.init().unwrap();
```

Heapless String

This is a heapless string set up with a capacity of 64 characters. The string is allocated on the stack, allowing it to hold up to 64 characters. We use this variable to display the temperature, ADC, and resistance values on the screen.

```
let mut buff: String<64> = String::new();
```

Convert the Reference Temperature to Kelvin

We defined the reference temperature as 25°C for the thermistor. However, for the equation, we need the temperature in Kelvin. To handle this, we use a helper function to perform the conversion. Alternatively, you could directly hardcode the Kelvin value (298.15 K, which is 273.15 + 25°C) to skip using the function.

```
let ref_temp = celsius_to_kelvin(REF_TEMP);
```

Loop

In a loop that runs every 1 second(adjust as you require), we read the ADC value, calculate the resistance from ADC, then derive the temperature from resistance, and display the results on the OLED.

Read ADC

We read the ADC value; we also put into the buffer.

```
let adc_value = adc.read(&mut p26).await.unwrap();
writeln!(buff, "ADC: {}", adc_value).unwrap();
```

ADC To Resistance

We convert the ADC To resistance; we put this also into the buffer.

```
let current_res = adc_to_resistance(adc_value, REF_RES);
writeln!(buff, "R: {:.2}", current_res).unwrap();
```

Calculate Temperature from Resistance

We use the measured resistance to calculate the temperature in Kelvin using the B-parameter equation.Afterward, we convert the temperature from Kelvin to Celsius.

```
let temperature_kelvin = calculate_temperature(current_res, REF_RES,
ref_temp, B_VALUE);
let temperature_celsius = kelvin_to_celsius(temperature_kelvin);
```

Write the Buffer to Display

```
writeln!(buff, "Temp: {:.2} °C", temperature_celsius).unwrap();
display.write_str(&buff).unwrap();
Timer::after_secs(1).await;
```

Clear the Buffer and Screen

```
buff.clear();
display.clear().unwrap();
```

Final code

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

use embassy_executor::Spawner;
use embassy_rp as hal;
use embassy_rp::block::ImageDef;
use embassy_rp::gpio::Pull;
use embassy_time::Timer;
use heapless::String;
use ssd1306::mode::DisplayConfig;
use ssd1306::prelude::DisplayRotation;
use ssd1306::size::DisplaySize128x64;
use ssd1306::{I2CDisplayInterface, Ssd1306};
use {defmt_rtt as _, panic_probe as _};

use embassy_rp::adc::{Adc, Channel};
use embassy_rp::peripherals::I2C1;
use embassy_rp::{adc, bind_interrupts, i2c};

use core::fmt::Write;

/// Tell the Boot ROM about our application
#[link_section = ".start_block"]
#[used]
pub static IMAGE_DEF: ImageDef = hal::block::ImageDef::secure_exe();

bind_interrupts!(struct Irqs {
    ADC_IRQ_FIFO => adc::InterruptHandler;
    I2C1_IRQ => i2c::InterruptHandler<I2C1>;
});
const fn calculate_adc_max(adc_bits: u8) -> u16 {
    (1 << adc_bits) - 1
}
const ADC_BITS: u8 = 12; // 12-bit ADC in Pico
const ADC_MAX: u16 = calculate_adc_max(ADC_BITS); // 4095 for 12-bit
ADC

const B_VALUE: f64 = 3950.0;
const REF_RES: f64 = 10_000.0; // Reference resistance in ohms (10kΩ)
const REF_TEMP: f64 = 25.0; // Reference temperature 25°C
                            // We have already covered about this
formula in ADC chpater
fn adc_to_resistance(adc_value: u16, ref_res: f64) -> f64 {
    let x: f64 = (ADC_MAX as f64 / adc_value as f64) - 1.0;
    // ref_res * x // If you connected thermistor to power supply
    ref_res / x
}

// B Equation to convert resistance to temperature
```

```
fn calculate_temperature(current_res: f64, ref_res: f64, ref_temp:
f64, b_val: f64) -> f64 {
    let ln_value = libm::log(current_res / ref_res); // Use libm for
`no_std`
    let inv_t = (1.0 / ref_temp) + ((1.0 / b_val) * ln_value);
    1.0 / inv_t
}

fn kelvin_to_celsius(kelvin: f64) -> f64 {
    kelvin - 273.15
}

fn celsius_to_kelvin(celsius: f64) -> f64 {
    celsius + 273.15
}

#[embassy_executor::main]
async fn main(_spawner: Spawner) {
    let p = embassy_rp::init(Default::default());
    // ADC to read the Vout value
    let mut adc = Adc::new(p.ADC, Irqs, adc::Config::default());
    let mut p26 = Channel::new_pin(p.PIN_26, Pull::None);

    // Setting up I2C send text to OLED display
    let sda = p.PIN_18;
    let scl = p.PIN_19;
    let i2c = i2c::I2c::new_async(p.I2C1, scl, sda, Irqs,
i2c::Config::default());
    let interface = I2CDisplayInterface::new(i2c);

    let mut display =
        Ssd1306::new(interface, DisplaySize128x64,
DisplayRotation::Rotate0).into_terminal_mode();
    display.init().unwrap();
    let mut buff: String<64> = String::new();
    let ref_temp = celsius_to_kelvin(REF_TEMP);
    loop {
        buff.clear();
        display.clear().unwrap();

        let adc_value = adc.read(&mut p26).await.unwrap();
        writeln!(buff, "ADC: {}", adc_value).unwrap();

        let current_res = adc_to_resistance(adc_value, REF_RES);
        writeln!(buff, "R: {:.2}", current_res).unwrap();

        let temperature_kelvin = calculate_temperature(current_res,
REF_RES, ref_temp, B_VALUE);
        let temperature_celsius =
kelvin_to_celsius(temperature_kelvin);

        writeln!(buff, "Temp: {:.2} °C",
```

```
temperature_celsius).unwrap();
        display.write_str(&buff).unwrap();
        Timer::after_secs(1).await;
    }
}

// Program metadata for `picotool info`.
// This isn't needed, but it's recomended to have these minimal
entries.
#[link_section = ".bi_entries"]
#[used]
pub static PICOTOOL_ENTRIES: [embassy_rp::binary_info::EntryAddr; 4] =
[
    embassy_rp::binary_info::rp_program_name!(c"Blinky Example"),
    embassy_rp::binary_info::rp_program_description!(
        c"This example tests the RP Pico on board LED, connected to
gpio 25"
    ),
    embassy_rp::binary_info::rp_cargo_version!(),
    embassy_rp::binary_info::rp_program_build_attribute!(),
];

// End of file
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `thermistor` folder.

```
git clone https://github.com/ImplFerris/pico2-embassy-projects
cd pico2-embassy-projects/thermistor/
```

USB Serial Communication

In this section, we'll explore how to establish communication between our device (Pico) and a computer(Linux). We'll demonstrate how to send a simple string from the device(Pico) to the computer, as well as how to send input from the computer to the device.

CDC ACM

The Communication Device Class (CDC) is a standard USB device class defined by the USB Implementers Forum (USB-IF). The Abstract Control Model (ACM) in CDC allows a device to act like a traditional serial port (like old COM ports). It's commonly used for applications that previously relied on serial COM or UART communication.

Tools for Linux

When you flash the code in this exercise, the device will appear as /dev/ttyACM0 in your computer. To interact with the USB serial port on Linux, you can use tools like minicom, tio (or cat) to read and send data to and from the device

- minicom: Minicom is a text-based serial port communications program. It is used to talk to external RS-232 devices such as mobile phones, routers, and serial console ports.
- tio: tio is a serial device tool which features a straightforward command-line and configuration file interface to easily connect to serial TTY devices for basic I/O operations.

Rust Crates

We will be using the example taken from the RP-HAL repository. It use two crates: usb-device, an USB stack for embedded devices in Rust, and usbd-serial, which implements the USB CDC-ACM serial port class. The SerialPort class in usbd-serial implements a stream-like buffered serial port and can be used in a similar way to UART.

References

- CDC: Communication Device Class (ACM)
- USB Device CDC ACM Class
- What is the difference between /dev/ttyUSB and /dev/ttyACM?
- Defined Class Codes

Pico to PC

The example provided in the RP-HAL repository sends a simple "Hello, World!" message from the Pico to the computer once the timer ticks reach 2,000,000. To ensure the message is only sent once, we add a check that sends it only on the first occurrence. Also, it polls for any incoming data to the device (Pico). If data is received, it converts it to uppercase and send it back(This is just show communication is working, not just echoing).

We'll slightly modify the code to make it more fun. Instead of sending "Hello, World!", we'll send "Hello, Rust!" to the computer. Wait, I know that's not the fun part. Here it comes: if you type 'r' in the terminal connected via USB serial, the onboard LED will turn on. Type anything else, and the LED will turn off.

Project from template

To set up the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, give your project a name, like "usb-fun" and select `RP-HAL` as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "usb-fun":
# cd usb-fun
```

Additional Crates required

Update your Cargo.toml to add these additional crate along with the existing dependencies.

```
usbd-serial = "0.2.2"
usb-device = "0.3.2"
```

Additional imports

```
// USB Device support
use usb_device::{class_prelude::*, prelude::*};
// USB Communications Class Device support
use usbd_serial::SerialPort;
```

Set up the USB driver

```
let usb_bus = UsbBusAllocator::new(hal::usb::UsbBus::new(
    pac.USB,
    pac.USB_DPRAM,
    clocks.usb_clock,
    true,
    &mut pac.RESETS,
));
```

Set up the USB Communications Class Device driver

```
let mut serial = SerialPort::new(&usb_bus);
```

Create a USB device with a fake VID and PID

```
let mut usb_dev = UsbDeviceBuilder::new(&usb_bus, UsbVidPid(0x16c0,
0x27dd))
    .strings(&[StringDescriptors::default()
        .manufacturer("implRust")
        .product("Ferris")
        .serial_number("TEST")])
    .unwrap()
    .device_class(2) // 2 for the CDC, from:
https://www.usb.org/defined-class-codes
    .build();
```

Sending Message to PC

This part sends "Hello, Rust!" to the PC when the timer count exceeds 2,000,000 by writing the text to the serial port. We ensure the message is sent only once.

```
if !said_hello && timer.get_counter().ticks() >= 2_000_000 {
    said_hello = true;
    // Writes bytes from `data` into the port and returns the number
of bytes written.
    let _ = serial.write(b"Hello, Rust!\r\n");
}
```

Polling for data

Here is the fun part. When you type characters on your computer, they are sent to the Pico via USB serial. On the Pico, we check if the received character matches the letter 'r'. If it matches, the onboard LED turns on. For any other character, the LED turns off.

```
    if usb_dev.poll(&mut [&mut serial]) {
        let mut buf = [0u8; 64];
        if let Ok(count) = serial.read(&mut buf) {
            for &byte in &buf[..count] {
                if byte == b'r' {
                    led.set_high().unwrap();
                } else {
                    led.set_low().unwrap();
                }
            }
        }
    }
```

The Full code

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

use embedded_hal::digital::OutputPin;
use hal::block::ImageDef;
use panic_halt as _;
use rp235x_hal as hal;

use usb_device::{class_prelude::*, prelude::*};
use usbd_serial::SerialPort;

#[link_section = ".start_block"]
#[used]
pub static IMAGE_DEF: ImageDef = hal::block::ImageDef::secure_exe();

const XTAL_FREQ_HZ: u32 = 12_000_000u32;

#[hal::entry]
fn main() -> ! {
    let mut pac = hal::pac::Peripherals::take().unwrap();
    let mut watchdog = hal::Watchdog::new(pac.WATCHDOG);

    let clocks = hal::clocks::init_clocks_and_plls(
        XTAL_FREQ_HZ,
        pac.XOSC,
        pac.CLOCKS,
        pac.PLL_SYS,
        pac.PLL_USB,
        &mut pac.RESETS,
        &mut watchdog,
    )
    .ok()
    .unwrap();
    let timer = hal::Timer::new_timer0(pac.TIMER0, &mut pac.RESETS,
&clocks);

    let sio = hal::Sio::new(pac.SIO);
    let pins = hal::gpio::Pins::new(
        pac.IO_BANK0,
        pac.PADS_BANK0,
        sio.gpio_bank0,
        &mut pac.RESETS,
    );
    let mut led = pins.gpio25.into_push_pull_output();

    let usb_bus = UsbBusAllocator::new(hal::usb::UsbBus::new(
        pac.USB,
        pac.USB_DPRAM,
```

```
        clocks.usb_clock,
        true,
        &mut pac.RESETS,
    ));

    let mut serial = SerialPort::new(&usb_bus);

    let mut usb_dev = UsbDeviceBuilder::new(&usb_bus,
UsbVidPid(0x16c0, 0x27dd))
        .strings(&[StringDescriptors::default()
            .manufacturer("implRust")
            .product("Ferris")
            .serial_number("TEST")])
        .unwrap()
        .device_class(2) // 2 for the CDC, from:
https://www.usb.org/defined-class-codes
        .build();

    let mut said_hello = false;
    loop {
        // Send data to the PC
        if !said_hello && timer.get_counter().ticks() >= 2_000_000 {
            said_hello = true;
            // Writes bytes from `data` into the port and returns the
number of bytes written.
            let _ = serial.write(b"Hello, Rust!\r\n");
        }

        // Read data from PC
        if usb_dev.poll(&mut [&mut serial]) {
            let mut buf = [0u8; 64];
            if let Ok(count) = serial.read(&mut buf) {
                for &byte in &buf[..count] {
                    if byte == b'r' {
                        led.set_high().unwrap();
                    } else {
                        led.set_low().unwrap();
                    }
                }
            }
        }
    }
}

#[link_section = ".bi_entries"]
#[used]
pub static PICOTOOL_ENTRIES: [hal::binary_info::EntryAddr; 5] = [
    hal::binary_info::rp_cargo_bin_name!(),
    hal::binary_info::rp_cargo_version!(),
    hal::binary_info::rp_program_description!(c"USB Fun"),
    hal::binary_info::rp_cargo_homepage_url!(),
```

```
    hal::binary_info::rp_program_build_attribute!(),
];
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `usb-fun` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/usb-fun/
```

How to Run ?

The method to flash (run the code) on the Pico is the same as usual. However, we need to set up tio to interact with the Pico through the serial port (/dev/ttyACM0). This allows us to read data from the Pico or send data to it.

tio

Make sure you have tio installed on your system. If not, you can install it using:

```
apt install tio
```

Connecting to the Serial Port

Run the following command to connect to the Pico's serial port:

```
tio /dev/ttyACM0
```

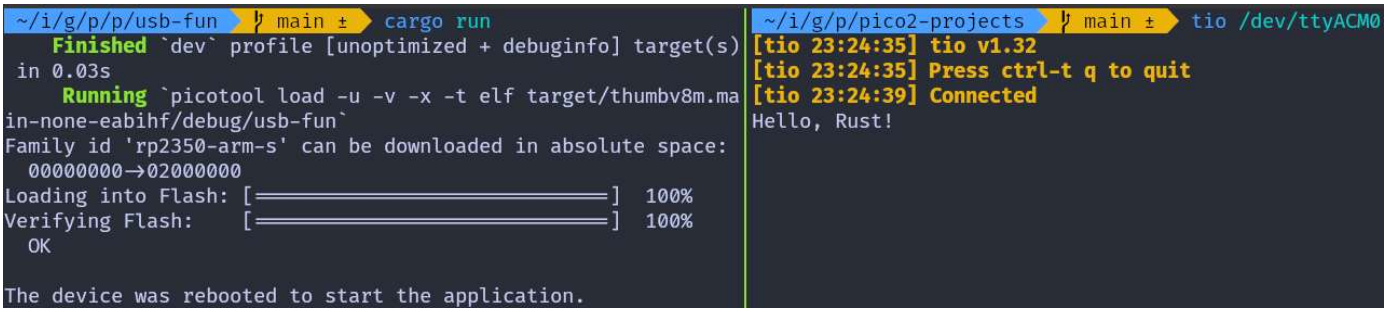
This will open a terminal session for communicating with the Pico.

Flashing and Running the Code

Open another terminal, navigate to the project folder, and flash the code onto the Pico as usual:

```
cargo run
```

If everything is set up correctly, you should see a "Connected" message in the tio terminal, followed by the "Hello, Rust!" message sent from the Pico.



Send data to Pico

In the terminal where tio is running, you type that will be sent to the Pico. You won't see what you type (since we're not echoing back the input).

If you press the letter 'r', the onboard LED will be turned on. If you press any other character, the LED will be turned off.

Embassy version

You can also refer to this project, which demonstrates using USB Serial with the Embassy framework.

```
git clone https://github.com/ImplFerris/pico2-embassy-projects
cd pico2-embassy-projects/usb-serial/
```

RFID

In this section, we will use the RFID Card Reader (RC522) module to read data from RFID tags and key fob tags.

What is RFID?

You've probably used them without even realizing it; on your apartment key, at the office, in parking lots, or with a contactless credit card. If you've got a toll pass in your car or used a hotel keycard, then yep, you've already seen them in action.

RFID (Radio Frequency Identification) is a technology that uses radio waves to identify and track objects, animals. It wirelessly transmits the stored data from a tag (containing a chip and antenna) to a reader when in range.

Categories By Range

RFID systems can be categorized by their operating frequency. The three main types are:

- **Low Frequency (LF)**: Operates at ~125 kHz with a short range (up to 10cm). It's slower and commonly used in access control and livestock tracking.
- **High Frequency (HF)**: Operates at 13.56 MHz with a range of 10cm to 1m. It offers moderate speed and is widely used in access control systems, such as office spaces, apartments, hotel keycards, as well as in ticketing, payments, and data transfer. We are going to use this one (RC522 module which operates at 13.56MHz)
- **Ultra-High Frequency (UHF)**: Operates at 860–960 MHz with a range of up to 12m. It's faster and commonly used in retail inventory management, anti-counterfeiting, and logistics.

Categories By Power source

RFID tags can either be active or passive, depending on how they are powered.

- **Active tags**: They have their own battery and can send signals on their own. These are typically used on large objects like rail cars, big reusable containers, and assets that need to be tracked over long distances.
- **Passive tags:** Unlike active tags, passive tags don't have a battery. They rely on the electromagnetic fields emitted by the RFID reader to power up. Once energized, they transmit data using radio waves. These are the most common type of RFID tags and are likely the ones you've encountered in everyday life. If you guessed it correctly, yes the RC522 is the passive tags.

Components:

RFID systems consist of an RFID Reader, technically referred to as the PCD (Proximity Coupling Device). In passive RFID tags, the reader powers the tag using an electromagnetic field. The tags themselves are called RFID Tags or, in technical terms, PICCs (Proximity Integrated Circuit Cards). It is good to know its technical terms also, it will come in handy if you want to refer the datasheet and other documents.

Reader typically include memory components like FIFO buffers and EEPROM. They also incorporate cryptographic features to ensure secure communication with Tags, allowing only authenticated RFID readers to interact with them. For example, RFID readers from NXP Semiconductors use the Crypto-1 cipher for authentication.

Each RFID tag has a hardcoded UID (Unique Identifier), which can be 4, 7, or 10 bytes in size.

References

- What Is A RFID Antenna?
- Types of RFID Systems

Meet the module

We will be using the RC522 RFID Card Reader Module, which is built on the MFRC522 IC (designed by NXP), operates at 13.56 MHz . This module is widely available online at an affordable price and typically comes with an RFID tag (MIFARE Classic 1K) and key fob, each containing 1KB of memory. MFRC522 Datasheet can be found here.

The microcontroller can communicate with the reader using SPI, UART, I2C. It also has an IRQ (Interrupt Request) pin that can trigger interrupts, so the microcontroller(pico) knows when the tag is nearby, instead of constantly asking the reader (kind of like "Are we there yet?").

Unfortunately, the library we're going to use doesn't support this feature yet, so we won't be using it for now. We'll update this section once support is added. So, are we there yet?

Additional Information about the Module:

- **Supported Standards**: ISO/IEC 14443 A / MIFARE
- **Card Reading Distance**: 0~50 mm
- **Idle Current**: 10–13 mA
- **Operating Current**: 13–26 mA
- **Operating Voltage**: DC 3.3V (⚠️ Do not use 5V or higher, it will cause damage).

MIFARE

MIFARE is a series of integrated circuit (IC) chips used in contactless smart cards and proximity cards, developed by NXP Semiconductors. MIFARE cards follow ISO/IEC 14443A standards and use encryption methods such as Crypto-1 algorithm. The most common family is MIFARE Classic, with a subtype called MIFARE Classic EV1.

Memory Layout

The MIFARE Classic 1K card is divided into 16 sectors, with each sector containing 4 blocks. Each block can hold up to 16 bytes, resulting in a total memory capacity of 1KB.

16 sectors × 4 blocks/sector × 16 bytes/block = 1024 bytes = 1KB

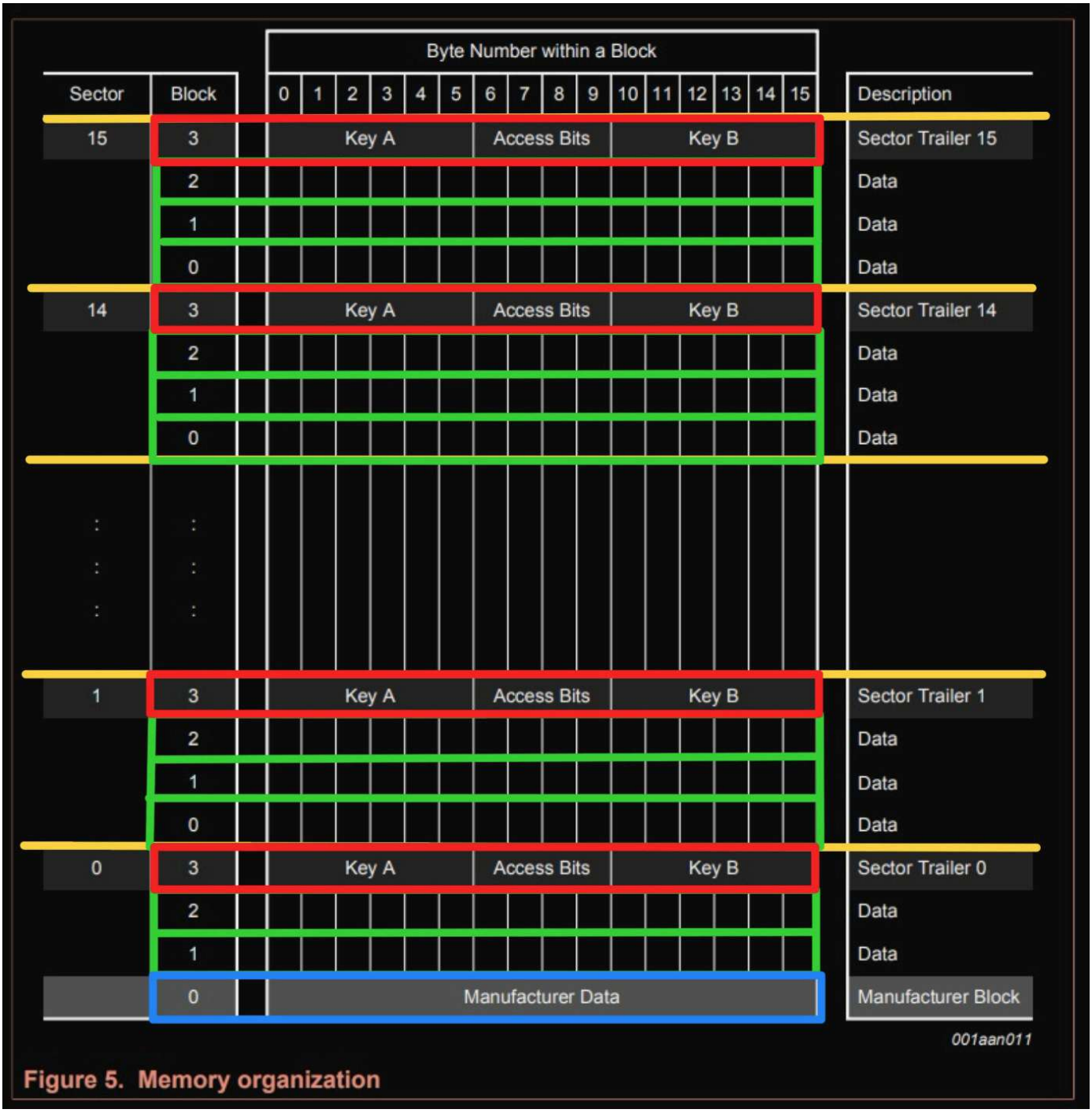


Figure 5. Memory organization

Sector Trailer

The last block of each sector, known as the "trailer" holds two secret keys and programmable access conditions for the blocks within that sector. Each sector has its own pair of keys (KeyA and KeyB), enabling support for multiple applications with a key hierarchy.

Default Keys

The MIFARE Classic 1K card is pre-configured with the default key FF FF FF FF FF FF for both KeyA and KeyB. When reading the trailer block, KeyA

values are returned as all zeros (00 00 00 00 00 00), while KeyB returned as it is.

By default, the access bytes (6, 7, and 8 of the trailer) are set to FF 07 80h. You can refer the 10th page for the datasheet for more information. And the 9th byte can be used for storing data.

Byte Number	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Description	KEY A						Access Bits		
Default Data	FF	FF	FF	FF	FF	FF	FF	07	80

Manufacturer Block

The first block (block 0) of the first sector(sector 0) contains IC manufacturer's data including the UID. This block is write-protected.

Data Block

Each sector has a trailer block, so only 3 blocks can be used for data storage in each sector. However, the first sector only has 2 usable blocks because the first block stores manufacturer data.

To read or write the data, you first need to authenticate with either Key A or Key B of that sector.

The data blocks can be further classified into two categories based on the access bits(we will explain about it later).

- read/write block: These are standard data blocks that allow basic operations such as reading and writing data.
- value block: These blocks are ideal for applications like electronic purses, where they are commonly used to store numeric values, such as account balances. So, you can perform incrementing (e.g., adding $10 to a balance) or decrementing (e.g., deducting $5 for a transaction).

Reference

- Datasheet: MIFARE Classic EV1 1K - Mainstream contactless smart card IC for fast and easy solution development

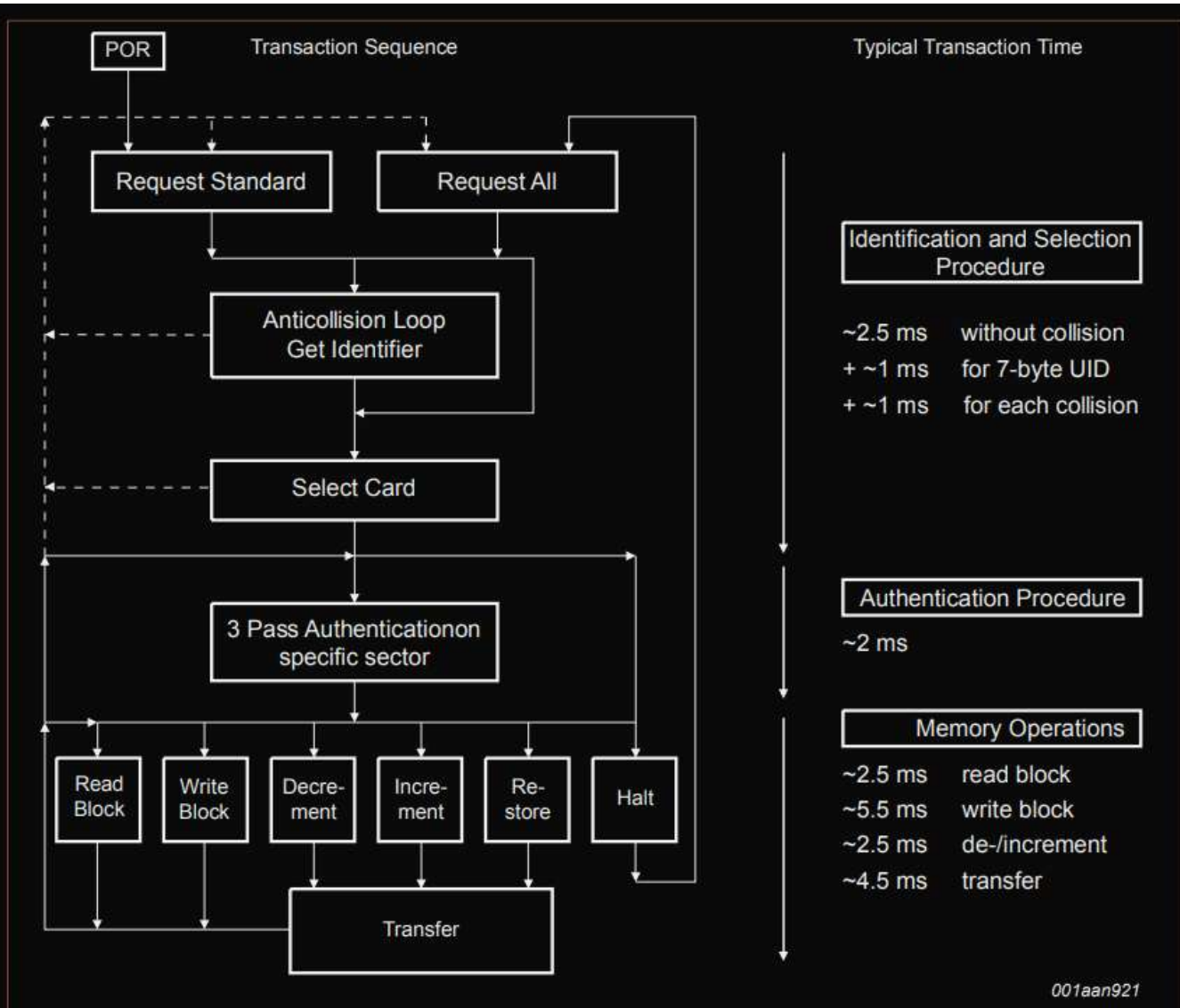
Flow

When you bring the tag near the reader, it goes into a state where it waits for either a REQA (Request) or WUPA (Wake Up) command.

To check if any tag is nearby, we send the REQA command in a loop. If the tag is nearby, it responds with an ATQA (Answer to Request).

Once we get the response, we select the card, and it sends back its UID (we won't dive into the full technical details involved in this process). After that, we authenticate the sector we want to read or write from. Once we're done with our operation, we send a HLTA command to put the card in the HALT state.

Note: Once the card is in the HALT state, only the WUPA command (reminds me of Chandler from Friends saying "WOOPAAH") can wake it up and let us do more operations.



1. the command flow diagram does not include the Personalize UID Usage and the SET_MOD_TYPE command, for details on those commands please see Section 10.1.1 and Section 11

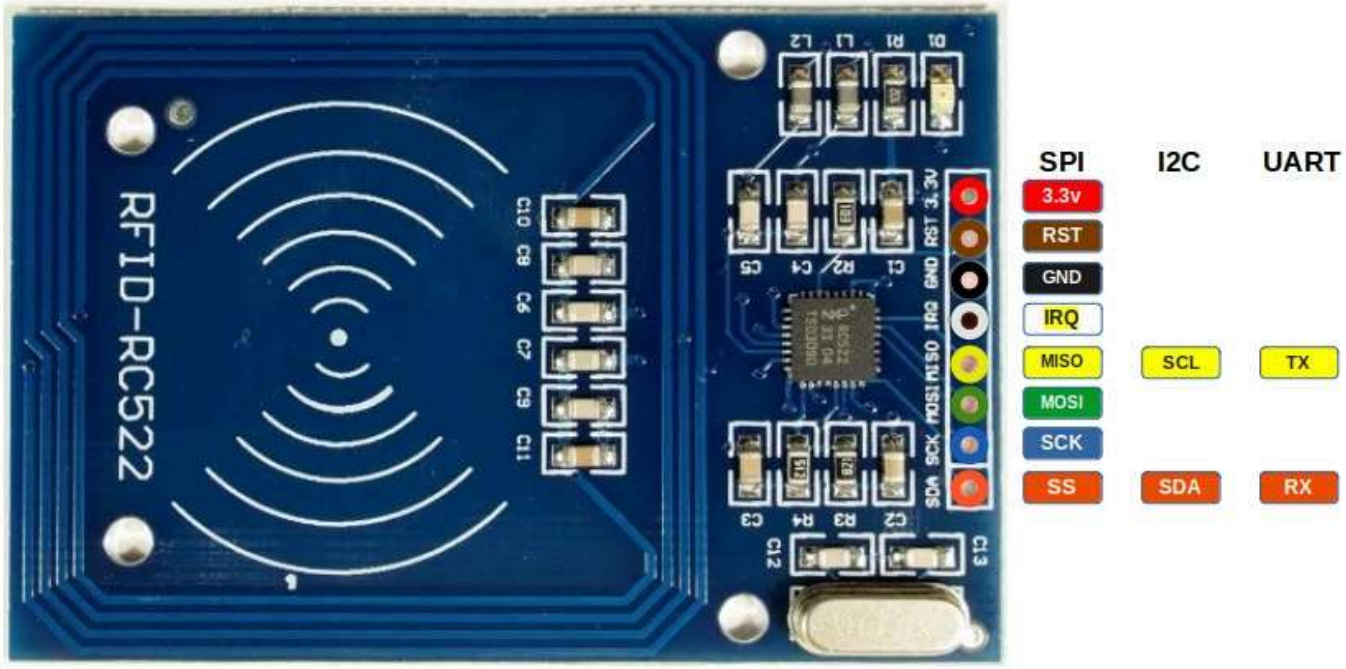
Figure 4. MIFARE Classic command flow diagram

Circuit

The introduction has become quite lengthy, so we will move the circuit diagram for connecting the Pico to the RFID reader to a separate page. Additionally, there are more pins that involved in this than any of the previous components we've used so far.

Pinout diagram of RC522

There are 8 pins in the RC522 RFID module.



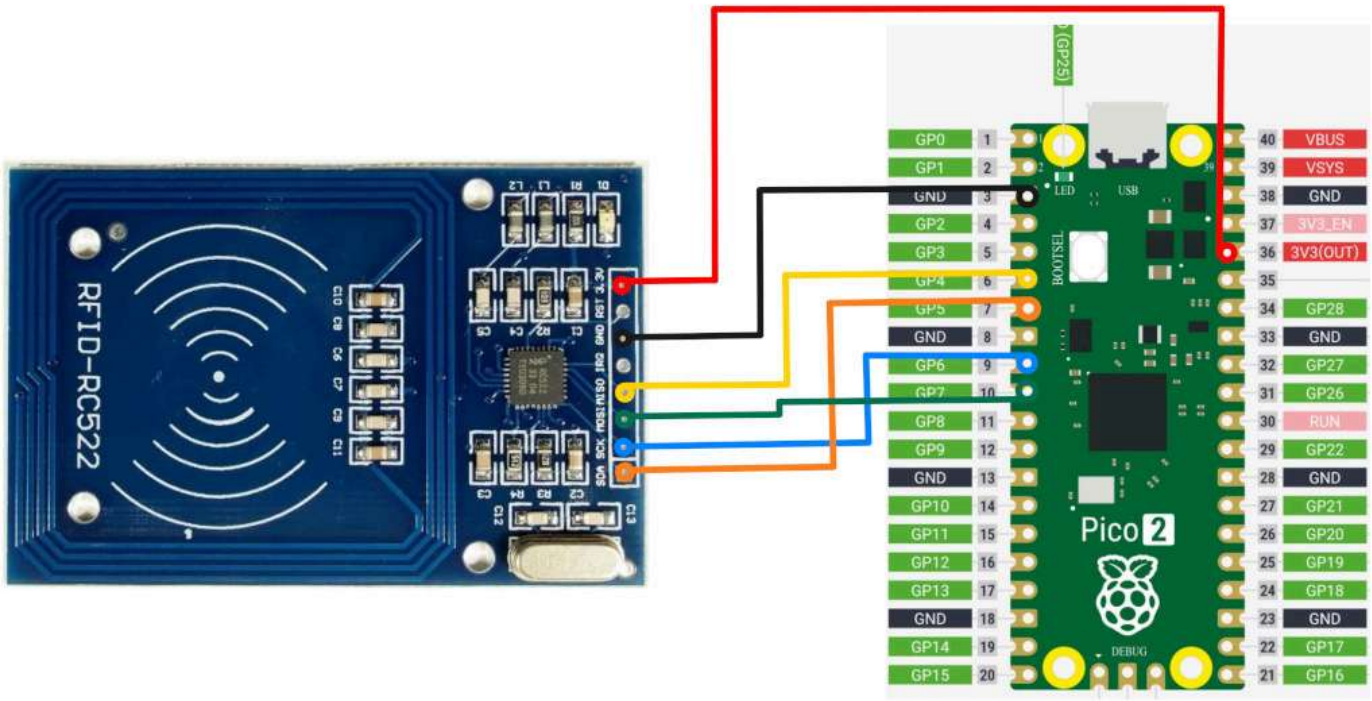
Pin	SPI Function	I²C Function	UART Function	Description
3.3V	Power	Power	Power	Power supply (3.3V).
GND	Ground	Ground	Ground	Ground connection.
RST	Reset	Reset	Reset	Reset the module.
IRQ	Interrupt (optional)	Interrupt (optional)	Interrupt (optional)	Interrupt Request (IRQ) informs the microcontroller when an RFID tag is detected. Without

				using IRQ, the microcontroller would need to constantly poll the module.
MISO	Master-In-Slave-Out	SCL	TX	In SPI mode, it acts as Master-In-Slave-Out (MISO). In I²C mode, it functions as the clock line (SCL). In UART mode, it acts as the transmit pin (TX).
MOSI	Master-Out-Slave-In	-	-	In SPI mode, it acts as Master-Out-Slave-In (MOSI).
SCK	Serial Clock	-	-	In SPI mode, it acts as the clock line that synchronizes data transfer.
SDA	Slave Select (SS)	SDA	RX	In SPI mode, it acts as the Slave select (SS, also referred as Chip Select). In I²C mode, it serves as the data line (SDA). In UART mode, it acts as the receive pin (RX).

Connecting the RFID Reader to the Raspberry Pi Pico

To establish communication between the Raspberry Pi Pico and the RFID Reader, we will use the SPI (Serial Peripheral Interface) protocol. The SPI interface can handle data speed up to 10 Mbit/s. We wont be utilizing the following Pins: RST, IRQ at the moment.

Pico Pin	Wire	RFID Reader Pin
3.3V		3.3V
GND		GND
GPIO 4		MISO
GPIO 5		SDA
GPIO 6		SCK
GPIO 7		MOSI



Read UID

Alright, let's get to the fun part and dive into some action! We'll start by writing a simple program to read the UID of the RFID tag.

mfrc522 Driver

We will be using the awesome crate "mfrc522". It is still under development. However, it has everything what we need for purposes.

USB Serial

To display the tag data, we'll use USB serial, which we covered in the last chapter. This will allow us to read from the RFID tag and display the UID on the computer.

Project from template

To set up the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, give your project a name, like "rfid-uid" and select `RP-HAL` as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "rfid-uid":
# cd rfid-uid
```

Additional Crates required

Update your Cargo.toml to add these additional crate along with the existing dependencies.

```
usbd-serial = "0.2.2"
usb-device = "0.3.2"
heapless = "0.8.0"
mfrc522 = "0.8.0"
embedded-hal-bus = "0.2.0"
```

We have added embedded-hal-bus, which provides the necessary traits for SPI and I2C buses. This is required for interfacing the Pico with the RFID reader.

Additional imports

```
use hal::fugit::RateExtU32;
use core::fmt::Write;

// to prepare buffer with data before writing into USB serial
use heapless::String;

// for setting up USB Serial
use usb_device::{class_prelude::*, prelude::*};
use usbd_serial::SerialPort;

// Driver for the MFRC522
use mfrc522::{comm::blocking::spi::SpiInterface, Mfrc522};

use embedded_hal_bus::spi::ExclusiveDevice;
```

Make sure to check out the USB serial tutorial for setting up the USB serial. We won't go over the setup here to keep it simple.

Helper Function to Print UID in Hex

We'll use this helper function to convert the u8 byte array (in this case UID) into a printable hex string. You could also just use raw bytes and enable hex mode in tio(requires latest version) or minicom, but I find this approach easier. In hex mode, it prints everything in hex, including normal text.

```
fn print_hex_to_serial<B: UsbBus>(data: &[u8], serial: &mut
SerialPort<B>) {
    let mut buff: String<64> = String::new();
    for &d in data.iter() {
        write!(buff, "{:02x} ", d).unwrap();
    }
    serial.write(buff.as_bytes()).unwrap();
}
```

Setting Up the SPI for the RFID Reader

Now, let's configure the SPI bus and the necessary pins to communicate with the RFID reader.

```
let spi_mosi = pins.gpio7.into_function::<hal::gpio::FunctionSpi>();
let spi_miso = pins.gpio4.into_function::<hal::gpio::FunctionSpi>();
let spi_sclk = pins.gpio6.into_function::<hal::gpio::FunctionSpi>();
let spi_bus = hal::spi::Spi::<_, _, _, 8>::new(pac.SPI0, (spi_mosi,
spi_miso, spi_sclk));
let spi_cs = pins.gpio5.into_push_pull_output();
let spi = spi_bus.init(
    &mut pac.RESETS,
    clocks.peripheral_clock.freq(),
    1_000.kHz(),
    embedded_hal::spi::MODE_0,
);
```

Getting the `SpiDevice` from SPI Bus

To work with the `mfrc522` crate, we need an `SpiDevice`. Since we only have the SPI bus from RP-HAL, we'll use the `embedded_hal_bus` crate to get the `SpiDevice`

from the SPI bus.

```
let spi = ExclusiveDevice::new(spi, spi_cs, timer).unwrap();
```

Initialize the mfrc522

```
let itf = SpiInterface::new(spi);
let mut rfid = Mfrc522::new(itf).init().unwrap();
```

Read the UID and Print

The main logic for reading the UID is simple. We continuously send the REQA command. If a tag is present, it send us the ATQA response. We then use this response to select the tag and retrieve the UID.

Once we have the UID, we use our helper function to print the UID bytes in hex format via USB serial.

```
loop {
    // to estabilish USB serial
    let _ = usb_dev.poll(&mut [&mut serial]);

    if let Ok(atqa) = rfid.reqa() {
        if let Ok(uid) = rfid.select(&atqa) {
            serial.write("\r\nUID: \r\n".as_bytes()).unwrap();
            print_hex_to_serial(uid.as_bytes(), &mut serial);
            timer.delay_ms(500);
        }
    }
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `rfid-uid` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/rfid-uid/
```

How to Run ?

The method to flash (run the code) on the Pico is the same as usual. However, we need to set up tio to interact with the Pico through the serial port (/dev/ttyACM0). This allows us to read data from the Pico or send data to it.

tio

Make sure you have tio installed on your system. If not, you can install it using:

```
apt install tio
```

Connecting to the Serial Port

Run the following command to connect to the Pico's serial port:

```
tio /dev/ttyACM0
```

This will open a terminal session for communicating with the Pico.

Flashing and Running the Code

Open another terminal, navigate to the project folder, and flash the code onto the Pico as usual:

```
cargo run
```

If everything is set up correctly, you should see a "Connected" message in the tio terminal.

Reading the UID

Now, bring the RFID tag near the reader. You should see the UID bytes displayed in hex format on the USB serial terminal.

```
~/i/g/p/p/rfid-uid  main ±  cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running `picotool load -u -v -x -t elf target/thumbv8m.main-none-eabihf/debug/rfid
-uid`
Family id 'rp2350-arm-s' can be downloaded in absolute space:
  00000000→02000000
Loading into Flash: [==============================]  100%
Verifying Flash:    [==============================]  100%
  OK

The device was rebooted to start the application.
```

```
~/i/g/p/p/rfid-uid  main ±  tio /dev/ttyACM0
[tio 09:25:55] tio v1.32
[tio 09:25:55] Press ctrl-t q to quit
[tio 09:26:03] Connected

UID:
13 37 73 31
[tio 09:26:07] Disconnected
```

Turn on LED on UID Match

In this section, we'll use the UID obtained in the previous chapter and hardcode it into our program. The LED will turn on only when the matching RFID tag is nearby; otherwise, it will remain off. When you bring the RFID tag close, the LED will light up. If you bring a different tag, like a key fob or any other RFID tag, the LED will turn off.

Logic

It is very simple straightforward logic.

```
let mut led = pins.gpio25.into_push_pull_output();

// Replace the UID Bytes with your tag UID
const TAG_UID: [u8; 4] = [0x13, 0x37, 0x73, 0x31];

loop {
    led.set_low().unwrap();

    if let Ok(atqa) = rfid.reqa() {
        if let Ok(uid) = rfid.select(&atqa) {
            if *uid.as_bytes() == TAG_UID {
                led.set_high().unwrap();
                timer.delay_ms(500);
            }
        }
    }
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `rfid-led` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/rfid-led/
```

Light it Up

Lets flash the pico with our program.

```
cargo run
```

Now bring the RFID tag near the RFID reader, the onboard LED on the Pico should turn on. Next, try bringing the key fob closer to the reader, and the LED will turn off. Alternatively, you can first read the key fob UID and hardcode it into the program to see the opposite behavior.

Read the data

In this section, we'll read all the blocks from the first sector (sector 0). As we mentioned earlier, to read or write to a specific block on the RFID tag, we first need to authenticate with the corresponding sector.

Authentication

Most tags come with a default key, typically 0xFF repeated six times. You may need to check the documentation to find the default key or try other common keys. For the RFID reader we are using, the default key is 0xFF repeated six times.

For authentication, we need:

- The tag's UID (obtained using the REQA and Select commands).
- The block number within the sector.
- The key (hardcoded in this case).

Read the block

After successful authentication, we can read data from each block using the `mf_read` function from the mfrc522 crate. If the read operation succeeds, the function returns 16 bytes of data from the block. This data will then be converted into a hex string and sent to the USB serial output.

The first sector (sector 0) consists of 4 blocks, with absolute block numbers ranging from 0 to 3. For higher sectors, the absolute block numbers increase accordingly (e.g., for sector 1, the blocks are 4, 5, 6, 7).

```
fn read_sector<E, COMM: mfrc522::comm::Interface<Error = E>, B:
UsbBus>(
    uid: &mfrc522::Uid,
    sector: u8,
    rfid: &mut Mfrc522<COMM, mfrc522::Initialized>,
    serial: &mut SerialPort<B>,
) -> Result<(), &'static str> {
    const AUTH_KEY: [u8; 6] = [0xFF; 6];

    let block_offset = sector * 4;
    rfid.mf_authenticate(uid, block_offset, &AUTH_KEY)
        .map_err(|_| "Auth failed")?;

    for abs_block in block_offset..block_offset + 4 {
        let data = rfid.mf_read(abs_block).map_err(|_| "Read
failed")?;
        print_hex_to_serial(&data, serial);
        serial
            .write("\r\n".as_bytes())
            .map_err(|_| "Write failed")?;
    }
    Ok(())
}
```

The main loop

The main loop operates similarly to what we covered in the previous chapter. After selecting a tag, we proceed to read its blocks. Once the block data is read, the loop sends the HLTA and stop_crypto1 commands to put the card in HALT state.

```
    loop {
        let _ = usb_dev.poll(&mut [&mut serial]);

        if let Ok(atqa) = rfid.reqa() {
            if let Ok(uid) = rfid.select(&atqa) {
                if let Err(e) = read_sector(&uid, 0, &mut rfid, &mut
serial) {
                    serial.write(e.as_bytes()).unwrap();
                }
                rfid.hlta().unwrap();
                rfid.stop_crypto1().unwrap();
            }
        }
    }
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `rfid-read` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/rfid-read/
```

How to Run ?

The method to flash (run the code) on the Pico is the same as usual. However, we need to set up tio to interact with the Pico through the serial port (/dev/ttyACM0). This allows us to read data from the Pico or send data to it.

Connecting to the Serial Port

Run the following command to connect to the Pico's serial port:

```
tio /dev/ttyACM0
```

This will open a terminal session for communicating with the Pico.

Flashing and Running the Code

Open another terminal, navigate to the project folder, and flash the code onto the Pico as usual:

```
cargo run
```

If everything is set up correctly, you should see a "Connected" message in the tio terminal.

Reading the UID

Bring the RFID tag close to the reader, and the USB serial terminal will display the data bytes read from the blocks of the first sector (sector 0).

```
~/i/g/p/p/rfid-read  main ±  cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running `picotool load -u -v -x -t elf target/thumbv8m.main-none-eabihf/debug/rfid
-read`
Family id 'rp2350-arm-s' can be downloaded in absolute space:
  00000000→02000000
Loading into Flash: [==============================]  100%
Verifying Flash:    [==============================]  100%
  OK

The device was rebooted to start the application.
```

```
~/i/g/p/p/rfid-read  main ±  tio /dev/ttyACM0
[tio 00:26:41] tio v1.32
[tio 00:26:41] Press ctrl-t q to quit
[tio 00:26:47] Connected
13 37 73 31 d6 08 04 00 62 63 64 65 66 67 68 69
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 ff 07 80 69 ff ff ff ff ff ff
```

Dump Entire Memory

You've learned how to read the data from each block of the first sector(sector 0) by authenticating into it. Now, we will loop through each sector. Re-Authentication is required every time we move to a new sector. For each sector, we will display the 16-byte data from every 4 blocks.

To make it clearer, we'll add some formatting and labels, indicating which sector and block we're referring to (both absolute and relative block numbers to the sector), as well as whether the block is a sector trailer or a data block.

Loop through the sector

We will create a separate function to loop through all 16 sectors (sectors 0 to 15), read all the blocks within each sector, and print their data.

```
fn dump_memory<E, COMM: mfrc522::comm::Interface<Error = E>, B:
UsbBus>(
    uid: &mfrc522::Uid,
    rfid: &mut Mfrc522<COMM, mfrc522::Initialized>,
    serial: &mut SerialPort<B>,
) -> Result<(), &'static str> {
    let mut buff: String<64> = String::new();
    for sector in 0..16 {
        // Printing the Sector number
        write!(buff, "\r\n-----------SECTOR {}-----------\r\n",
sector).unwrap();
        serial.write(buff.as_bytes()).unwrap();
        buff.clear();

        read_sector(uid, sector, rfid, serial)?;
    }
    Ok(())
}
```

Labels

The

`read_sector`

function follows the same logic as before, but with added formatting and labels. It now prints the absolute block number, the block number relative to the sector, and labels for the manufacturer data (MFD) block and sector trailer blocks.

```
fn read_sector<E, COMM: mfrc522::comm::Interface<Error = E>, B:
UsbBus>(
    uid: &mfrc522::Uid,
    sector: u8,
    rfid: &mut Mfrc522<COMM, mfrc522::Initialized>,
    serial: &mut SerialPort<B>,
) -> Result<(), &'static str> {
    const AUTH_KEY: [u8; 6] = [0xFF; 6];

    let mut buff: String<64> = String::new();

    let block_offset = sector * 4;
    rfid.mf_authenticate(uid, block_offset, &AUTH_KEY)
        .map_err(|_| "Auth failed")?;

    for abs_block in block_offset..block_offset + 4 {
        let rel_block = abs_block - block_offset;
        let data = rfid.mf_read(abs_block).map_err(|_| "Read
failed")?;

        // Prining the Block absolute and relative numbers
        write!(buff, "\r\nBLOCK {} (REL: {}) | ", abs_block,
rel_block).unwrap();
        serial.write(buff.as_bytes()).unwrap();
        buff.clear();

        // Printing the block data
        print_hex_to_serial(&data, serial);

        // Printing block type
        let block_type = get_block_type(sector, rel_block);
        write!(buff, "| {} ", block_type).unwrap();
        serial.write(buff.as_bytes()).unwrap();
        buff.clear();
    }
    serial
        .write("\r\n".as_bytes())
        .map_err(|_| "Write failed")?;
    Ok(())
}
```

We will create a small helper function to determine the block type based on the sector and its relative block number.

```
fn get_block_type(sector: u8, rel_block: u8) -> &'static str {
    match rel_block {
        0 if sector == 0 => "MFD",
        3 => "TRAILER",
        _ => "DATA",
    }
}
```

The main loop

There isn't much change in the main loop. We just call the `dump_memory` function instead of `read_sector`.

```
loop {
    let _ = usb_dev.poll(&mut [&mut serial]);
    if let Ok(atqa) = rfid.reqa() {
        if let Ok(uid) = rfid.select(&atqa) {
            if let Err(e) = dump_memory(&uid, &mut rfid, &mut serial)
{
                serial.write(e.as_bytes()).unwrap();
            }
            rfid.hlta().unwrap();
            rfid.stop_crypto1().unwrap();
        }
    }
}
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `rfid-dump` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/rfid-dump/
```

Dump

When you run the program and bring your tag or key fob close, you should see output like this. If you notice the 0x40..0x43 bytes in the block 18 (the block 2 of the sector 4) and wonder why it's there; good catch! That's the custom data I wrote to the tag.

```
~/i/g/p/p/rfid-dump  main ±  cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running `picotool load -u -v -x -t elf target/thumbv8m.main-none-eabihf/debug/rfid-dump`
Family id 'rp2350-arm-s' can be downloaded in absolute space:
  00000000->02000000
Loading into Flash: [==============================]  100%
Verifying Flash:    [==============================]  100%
  OK

The device was rebooted to start the application.
~/i/g/p/p/rfid-dump  main ±
```

```
~/i/g/p/p/rfid-dump  main ±  tio /dev/ttyACM0
[tio 10:10:01] tio v1.32
[tio 10:10:01] Press ctrl-t q to quit
[tio 10:10:27] Connected

──────────SECTOR 0──────────

BLOCK 0 (REL: 0) | 13 31 73 31 d6 08 04 00 62 63 64 65 66 67 68 69 | MFD
BLOCK 1 (REL: 1) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 2 (REL: 2) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 3 (REL: 3) | 00 00 00 00 00 00 ff 07 80 69 ff ff ff ff ff ff | TRAILER

──────────SECTOR 1──────────

BLOCK 4 (REL: 0) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 5 (REL: 1) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 6 (REL: 2) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 7 (REL: 3) | 00 00 00 00 00 00 ff 07 80 69 ff ff ff ff ff ff | TRAILER

──────────SECTOR 2──────────

BLOCK 8 (REL: 0) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 9 (REL: 1) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 10 (REL: 2) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 11 (REL: 3) | 00 00 00 00 00 00 ff 07 80 69 ff ff ff ff ff ff | TRAILER

──────────SECTOR 3──────────

BLOCK 12 (REL: 0) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 13 (REL: 1) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 14 (REL: 2) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 15 (REL: 3) | 00 00 00 00 00 00 ff 07 80 69 ff ff ff ff ff ff | TRAILER

──────────SECTOR 4──────────

BLOCK 16 (REL: 0) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 17 (REL: 1) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 18 (REL: 2) | 40 41 42 43 40 41 42 43 40 41 42 43 40 41 42 43 | DATA
BLOCK 19 (REL: 3) | 00 00 00 00 00 00 ff 07 80 69 ff ff ff ff ff ff | TRAILER

──────────SECTOR 5──────────

BLOCK 20 (REL: 0) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 21 (REL: 1) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 22 (REL: 2) | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | DATA
BLOCK 23 (REL: 3) | 00 00 00 00 00 00 ff 07 80 69 ff ff ff ff ff ff | TRAILER
```

Write Data

We will write data into block 2 of sector 4. First, we will print the data in the block before writing to it, and then again after writing. To perform the write operation, we will use the `mf_write` function from the mfrc522 crate.

Writing trailer block

Accidentally writing to the wrong block and overwriting the trailer block may alter the authentication key or access bits, which could make the sector unusable.

Write function

We will use this function to write data to the block. The mf_write function requires the absolute block number, which we will calculate using the sector number and its relative block number.

```
fn write_block<E, COMM: mfrc522::comm::Interface<Error = E>>(
    uid: &mfrc522::Uid,
    sector: u8,
    rel_block: u8,
    data: [u8; 16],
    rfid: &mut Mfrc522<COMM, mfrc522::Initialized>,
) -> Result<(), &'static str> {
    const AUTH_KEY: [u8; 6] = [0xFF; 6];

    let block_offset = sector * 4;
    let abs_block = block_offset + rel_block;

    rfid.mf_authenticate(uid, block_offset, &AUTH_KEY)
        .map_err(|_| "Auth failed")?;

    rfid.mf_write(abs_block, data).map_err(|_| "Write failed")?;

    Ok(())
}
```

The main loop

The main loop begins by reading and printing the current content of a specified block before writing new data to it. The `write_block` function is used to write the constant DATA, which must fill the entire 16-byte block. Any unused bytes are padded with null bytes (0x00).

```
    let target_sector = 4;
    let rel_block = 2;
    const DATA: [u8; 16] = [
        b'i', b'm', b'p', b'l', b'R', b'u', b's', b't', // "implRust"
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // Remaining
bytes as 0x00
    ];

    loop {
        let _ = usb_dev.poll(&mut [&mut serial]);

        if let Ok(atqa) = rfid.reqa() {
            if let Ok(uid) = rfid.select(&atqa) {
                serial
                    .write("\r\n----Before Write----\r\n".as_bytes())
                    .unwrap();
                if let Err(e) = read_sector(&uid, target_sector, &mut
rfid, &mut serial) {
                    serial.write(e.as_bytes()).unwrap();
                }

                if let Err(e) = write_block(&uid, target_sector,
rel_block, DATA, &mut rfid) {
                    serial.write(e.as_bytes()).unwrap();
                }

                serial
                    .write("\r\n----After Write----\r\n".as_bytes())
                    .unwrap();
                if let Err(e) = read_sector(&uid, target_sector, &mut
rfid, &mut serial) {
                    serial.write(e.as_bytes()).unwrap();
                }
                rfid.hlta().unwrap();
                rfid.stop_crypto1().unwrap();
            }
        }
    }
```

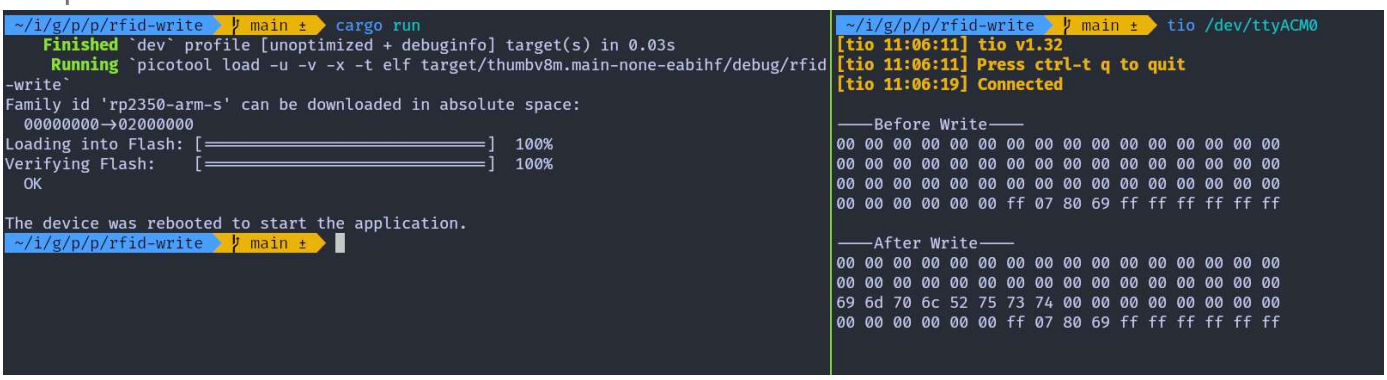
Clone the existing project

You can clone (or refer) project I created and navigate to the `rfid-write` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/rfid-write/
```

Output

When you run the program, the output will display the hex representation of "implRust" visible in the third row.



Changing the Authentication Key

Let's change the authentication key (KeyA) for sector 1. By default, it is set to `FF FF FF FF FF FF`. We'll update it to `52 75 73 74 65 64` which is hex for "Rusted." To do this, we need to modify the trailer block (block 3) of sector 1 while leaving the rest of the sector untouched.

Before proceeding, it is a good idea to verify the current contents of this block. Run the Dump Memory or Read Data program to check.

> **Default Keys**
>
> The MIFARE Classic 1K card is pre-configured with the default key FF FF FF FF FF FF for both KeyA and KeyB. When reading the trailer block, KeyA values are returned as all zeros (00 00 00 00 00 00), while KeyB returned as it is.

We'll also modify the KeyB contents to verify that the write was successful. We'll set KeyB to the hex bytes of "Ferris" (46 65 72 72 69 73).

Before writing, the access bytes and KeyB values in your block should mostly match what I have, but double-checking is always better than guessing.

Here's the plan:

1. In the program, we hardcode the default key (`FF FF FF FF FF FF`) into a variable named `current_key`.
2. Set the `new_key` to `Rusted`

(in hex bytes). This is necessary to print the block content after writing; otherwise, we'll get an auth error.

3. The program will print the block's contents both before and after writing.

Once the key is updated, bring the tag nearby again. You will likely see an "Auth failed" error. If you're wondering why, congrats-you figured it out! The new key was successfully written, so the hardcoded `current_key` no longer works. To verify, modify the `read-data` program to use the new key (`Rusted`) and try again.

Key and Data

The DATA array contains the new KeyA ("Rusted" in hex), access bits, and KeyB ("Ferris" in hex). The current_key is set to the default FF FF FF FF FF FF, and new_key is the first 6 bytes of DATA, which is "Rusted".

```
let target_sector = 1;
let rel_block = 3;
const DATA: [u8; 16] = [
    0x52, 0x75, 0x73, 0x74, 0x65, 0x64, // Key A: "Rusted"
    0xFF, 0x07, 0x80, 0x69, // Access bits and trailer byte
    0x46, 0x65, 0x72, 0x72, 0x69, 0x73, // Key B: "Ferris"
];
let current_key = &[0xFF; 6];
let new_key: &[u8; 6] = &DATA[..6].try_into().unwrap();
```

Write Block function

We have slighly modified the write_block function to accept key as argument.

```rust
fn write_block<E, COMM: mfrc522::comm::Interface<Error = E>>(
    uid: &mfrc522::Uid,
    sector: u8,
    rel_block: u8,
    data: [u8; 16],
    key: &[u8; 6],
    rfid: &mut Mfrc522<COMM, mfrc522::Initialized>,
) -> Result<(), &'static str> {

    let block_offset = sector * 4;
    let abs_block = block_offset + rel_block;

    rfid.mf_authenticate(uid, block_offset, key)
        .map_err(|_| "Auth failed")?;

    rfid.mf_write(abs_block, data).map_err(|_| "Write failed")?;

    Ok(())
}
```

Read Sector function

We have done similar modification for the read_sector function also.

```rust
fn read_sector<E, COMM: mfrc522::comm::Interface<Error = E>, B:
UsbBus>(
    uid: &mfrc522::Uid,
    sector: u8,
    key: &[u8; 6],
    rfid: &mut Mfrc522<COMM, mfrc522::Initialized>,
    serial: &mut SerialPort<B>,
) -> Result<(), &'static str> {
    let block_offset = sector * 4;
    rfid.mf_authenticate(uid, block_offset, key)
        .map_err(|_| "Auth failed")?;

    for abs_block in block_offset..block_offset + 4 {
        let data = rfid.mf_read(abs_block).map_err(|_| "Read
failed")?;
        print_hex_to_serial(&data, serial);
        serial
            .write("\r\n".as_bytes())
            .map_err(|_| "Write failed")?;
    }
    Ok(())
}
```

The main loop

There's nothing new in the main loop. All the read and write functions are ones you've already seen. We're just printing the sector content before and after changing the key.

```
    loop {
        let _ = usb_dev.poll(&mut [&mut serial]);

        if let Ok(atqa) = rfid.reqa() {
            if let Ok(uid) = rfid.select(&atqa) {
                serial
                    .write("\r\n----Before Write----\r\n".as_bytes())
                    .unwrap();
                if let Err(e) = read_sector(&uid, target_sector,
current_key, &mut rfid, &mut serial) {
                    serial.write(e.as_bytes()).unwrap();
                }

                if let Err(e) =
                    write_block(&uid, target_sector, rel_block, DATA,
current_key, &mut rfid)
                {
                    serial.write(e.as_bytes()).unwrap();
                }

                serial
                    .write("\r\n----After Write----\r\n".as_bytes())
                    .unwrap();
                if let Err(e) = read_sector(&uid, target_sector,
new_key, &mut rfid, &mut serial) {
                    serial.write(e.as_bytes()).unwrap();
                }
                rfid.hlta().unwrap();
                rfid.stop_crypto1().unwrap();
            }
        }
    }
```

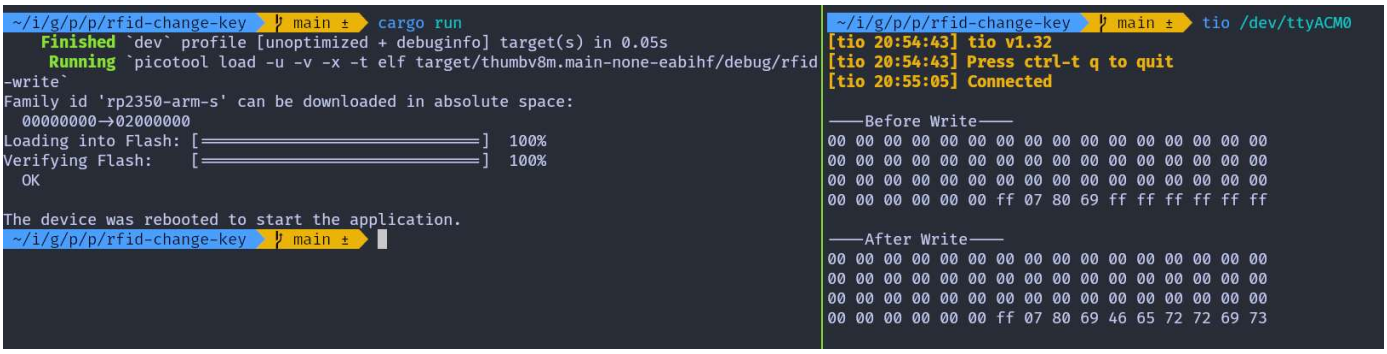
Clone the existing project

You can clone (or refer) project I created and navigate to the `rfid-change-key` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/rfid-change-key/
```

Output

As you can see in the output, when you run the program, it will display the contents of the target block before and after writing. After we change the key, bringing the tag back to the reader will result in an "auth failed" message because the current_key has been changed; The new key is 52 75 73 74 65 64 (Rusted).



You can also modify the read data program we used earlier with the new key to verify it.

Access Control

The tag includes access bits that enable access control for the data stored in the tag. This chapter will explore how these access bits function. This section might feel a bit overwhelming, so I'll try to make it as simple and easy to understand as possible.

Modifying Access Bits

Be careful when writing the access bits, as incorrect values can make the sector unusable.

Permissions

These are the fundamental permissions that will be used to define access conditions. The table explains each permission operation and specifies the blocks to which it is applicable: normal data blocks (read/write), value blocks, or sector trailers.

Operation	Description	Applicable for Block Type
Read	Reads one memory block	Read/Write, Value, Sector Trailer
Write	Writes one memory block	Read/Write, Value, Sector Trailer
Increment	Increments the contents of a block and stores the result in the internal Transfer Buffer	Value
Decrement	Decrements the contents of a block and stores the result in the internal Transfer Buffer	Value

Operation	Description	Applicable for Block Type
Restore	Reads the contents of a block into the internal Transfer Buffer	Value
Transfer	Writes the contents of the internal Transfer Buffer to a block	Value, Read/Write

Access conditions

Let's address the elephant in the room: The access conditions. During my research, I found that many people struggled to make sense of the access condition section in the datasheet. Here is my attempt to explain it for easy to understand 🤞 .

You can use just 3 bit-combinations per block to control its permissions. In the official datasheet, this is represented using a notation like CX_Y ($C1_0$, $C1_2$... $C3_3$) for the access bits. The first number (X) in this notation refers to the access bit number, which ranges from 1 to 3, each corresponding to a specific permission type. However, the meaning of these permissions varies depending on whether the block is a data block or a trailer block. The second number (Y) in the subscript denotes the relative block number, which ranges from 0 to 3.

Table 1: Access conditions for the sector trailer

In the original datasheet, the subscript number is not specified in the table. I have added the subscript "3", as the sector trailer is located at Block 3.

Readable Key

If you can read the key, it cannot be used as an authentication key. Therefore, in this table, whenever Key B is readable, it cannot serve as the authentication key. If you've noticed, yes, the Key A can never be read.

Access Bits			Access Condition for					
			Key A		Access Bits		Key B	
$C1_3$	$C2_3$	$C3_3$	Read	Write	Read	Write	Read	W
0	0	0	never	key A	key A	never	key A	k
0	1	0	never	never	key A	never	key A	n
1	0	0	never	key B	key A|B	never	never	k
1	1	0	never	never	key A|B	never	never	n
0	0	1	never	key A	key A	key A	key A	k
0	1	1	never	key B	key A|B	key B	never	k
1	0	1	never	never	key A|B	key B	never	n
1	1	1	never	never	key A|B	never	never	n

How to make sense out of this table?

It is a simple table showing the correlation between bit combinations and permissions.

For example: Let's say you select "1 0 0" (3rd row in the table), then you can't read KeyA, KeyB. However, you can modify the KeyA as well as KeyB value with KeyB. You can Read Access Bits with either KeyA or KeyB. But, you can never modify the Access Bits.

Now, where should these bits be stored? We will place them in the 6th, 7th, and 8th bytes at a specific location, which will be explained shortly.

Table 2: Access conditions for data blocks

This applies to all data blocks. The original datasheet does not include the subscript "Y", I have added it for context. Here, "Y" represents the block number (ranging from 0 to 2).

The default config here indicates that both Key A and Key B can perform all operations. However, as seen in the previous table, Key B is readable (in default config), making it unusable for authentication. Therefore, only Key A can be used.

Access Bits			Access Condition for			
$C1_Y$	$C2_Y$	$C3_Y$	Read	Write	Increment	Decrement,Tra
0	0	0	key A|B	key A|B	key A|B	key A|B
0	1	0	key A|B	never	never	never
1	0	0	key A|B	key B	never	never
1	1	0	key A|B	key B	key B	key A|B
0	0	1	key A|B	never	never	key A|B
0	1	1	key B	key B	never	never
1	0	1	key B	never	never	never
1	1	1	never	never	never	never

Note: "If KeyB can be read in the Sector Trailer, it can't be used for authentication. As a result, if the reader uses KeyB to authenticate a block with access conditions that uses KeyB, the card will refuse any further memory access after authentication."

How to make sense out of this table?

It's similar to the previous one; it shows the relationship between bit combinations and permissions.

For example: If you select "0 1 0" (2nd row in the table) and use this permission for block 1, you can use either KeyA or KeyB to read block 1. However, no other operations can be performed on block 1.

The notation for this is as follows: the block number is written as a subscript to the bit labels (e.g., $C1_1$, $C2_1$, $C3_1$). Here, the subscript "1" represents block 1. For the selected combination "0 1 0", this means:

- $C1_1 = 0$
- $C2_1 = 1$
- $C3_1 = 0$

These bits will also be placed in the 6th, 7th, and 8th bytes at a specific location, which will be explained shortly.

Table 3: Access conditions table

Let's colorize the original table to better visualize what each bit represents. The 7th and 3rd bits in each byte are related to the sector trailer. The 6th and 2nd bits correspond to Block 2. The 5th and 1st bits are associated with Block 1. The 4th and 0th bits are related to Block 0.

The overline on the notation indicates inverted values. This means that if the CX_y value is 0, then $\overline{CX_y}$ becomes 1.

Byte	7	6	5	4	3	2	1	0
Byte 6	$C2_3$	$C2_2$	$C2_1$	$C2_0$	$C1_3$	$C1_2$	$C1_1$	$C1_0$
Byte 7	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$C3_3$	$C3_2$	$C3_1$	$C3_0$
Byte 8	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$

The default access bit "FF 07 80". Let's try to understand what it means.

Byte	7	6	5	4	3	2	1	0
Byte 6	1	1	1	1	1	1	1	1
Byte 7	0	0	0	0	0	1	1	1
Byte 8	1	0	0	0	0	0	0	0

We can derive the CX_Y values from the table above. Notice that only $C3_3$ is set to 1, while all other values are 0. Now, refer to Table 1 and Table 2 to understand which permission this corresponds to.

Block	$C1_Y$	$C2_Y$	$C3_Y$	Access
Block 0	0	0	0	All permissions with Key A
Block 1	0	0	0	All permissions with Key A
Block 2	0	0	0	All permissions with Key A
Block 3 (Trailer)	0	0	1	You can write Key A using Key A. Access Bits and Key B can only be read and written using Key A.

Since Key B is readable, you cannot use it for authentication.

Calculator on next page

Still confused? Use the calculator on the next page to experiment with different combinations. Adjust the permissions for each block and observe how the Access Bits values change accordingly.

Reference

- 11th page of the datasheet

MIFARE Classic 1K Access Bits Calculator

Decode: You can modify the "Access bits" and the Data Block and Sector Trailer tables will automatically update.

Encode: Click the "Edit" button in each row of the table to select your preferred access conditions. This will update the Access Bits.

Warning

Writing an incorrect value to the access condition bits can make the sector inaccessible.

Access Bits

FF0780

Data Block Access Conditions:

Block	C1Y	C2Y	C3Y	Read	Write	Increment	Dec
Block 0	0	0	0	key A|B	key A|B	key A|B	key
Block 1	0	0	0	key A|B	key A|B	key A|B	key
Block 2	0	0	0	key A|B	key A|B	key A|B	key

Sector Trailer (Block 3) Access Conditions:

$C1_3$	$C2_3$	$C3_3$	Read Key A	Write Key A	Read Access Bits	Write Access Bits	Read Key B
0	0	1	never	key A	key A	key A	key A

$C1_3$	$C2_3$	$C3_3$	Read Key A	Write Key A	Read Access Bits	Write Access Bits	Read Key B

References

- This UI is inspired from this calculator: Mifarecalc
- MIFARE-Classic-1K-Access-Bits-Calculator

SD Card (SDC/MMC)

In this section, we will explore how to use the SD Card reader module. Depending on your project, you can use the SD card to store collected data from sensors, save game ROMs and progress, or store other types of information.

MMC

The MultiMediaCard (MMC) was introduced as an early type of flash memory storage, preceding the SD Card. It was commonly used in devices such as camcorders, digital cameras, and portable music players. MMCs store data as electrical charges in flash memory cells, unlike optical disks, which rely on laser-encoded data on reflective surfaces.

SD (Secure Digital) Card

The Secure Digital Card (SDC), commonly referred to as an SD Card, is an evolution of the MMC. SD Cards are widely used as external storage in electronic devices such as cameras, smartphones. A smaller variant, the microSD card, is commonly used in smartphones, drones, and other devices.

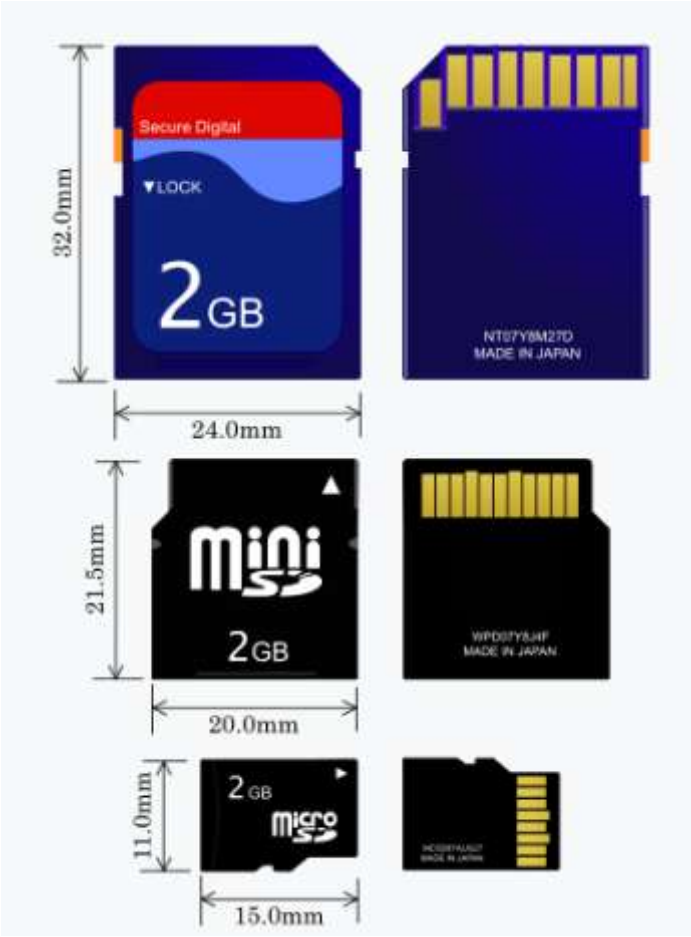


Image credit: Based on SD card by Tkgd2007, licensed under the GFDL and CC BY-SA 3.0, 2.5, 2.0, 1.0.

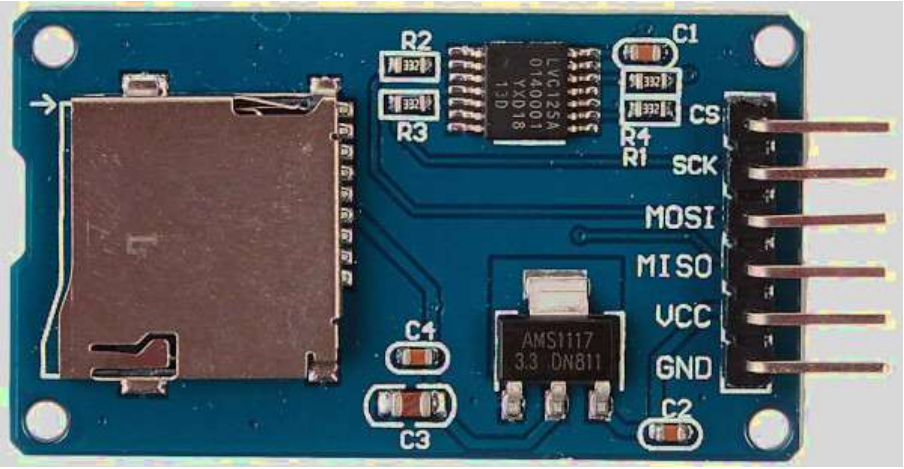
SD cards read and write data in blocks, typically 512 bytes in size, allowing them to function as block devices; this makes SD cards behave much like hard drives.

Protocol

To communicate with an SD card, we can use the SD Bus protocol, SPI protocol, or UHS-II Bus protocol. The Raspberry Pi (but not the Raspberry Pi Pico) uses the SD Bus protocol, which is more complex than SPI. The full specs of the SD Bus protocol are not accessible to the public and are only available through the SD Association. We will be using the SPI protocol, as the Rust driver we will be using is designed to work with it.

Hardware Requirements

We'll be using the Micro SD Card adapter module. You can search for either "Micro SD Card Reader Module" or "Micro SD Card Adapter" to find them.



And of course, you'll need a microSD card. The SD card should be formatted with FAT32; Depending on your computer's hardware, you might need a separate SD card adapter (not the one mentioned above) to format the microSD card. Some laptops comes with direct microSD card support.

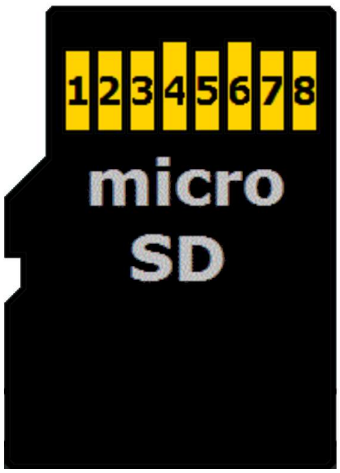
References:

- I highly recommend watching Jonathan Pallant's talk at Euro Rust 2024 on writing an SD card driver in Rust. He wrote the driver we are going to use (originally he created it to run MS-DOS on ARM). It is not intended for production systems.
- If you want to understand how it works under the hood in SPI mode, you can refer to this article: How to Use MMC/SDC
- Wikipedia

Circuit

microSD Card Pin Mapping for SPI Mode

We'll focus only on the microSD card since that's what we're using. The microSD has 8 pins, but we only need 6 for SPI mode. You may have noticed that the SD card reader module we have also has only 6 pins, with markings for the SPI functions. The table below shows the microSD card pins and their corresponding SPI functions.

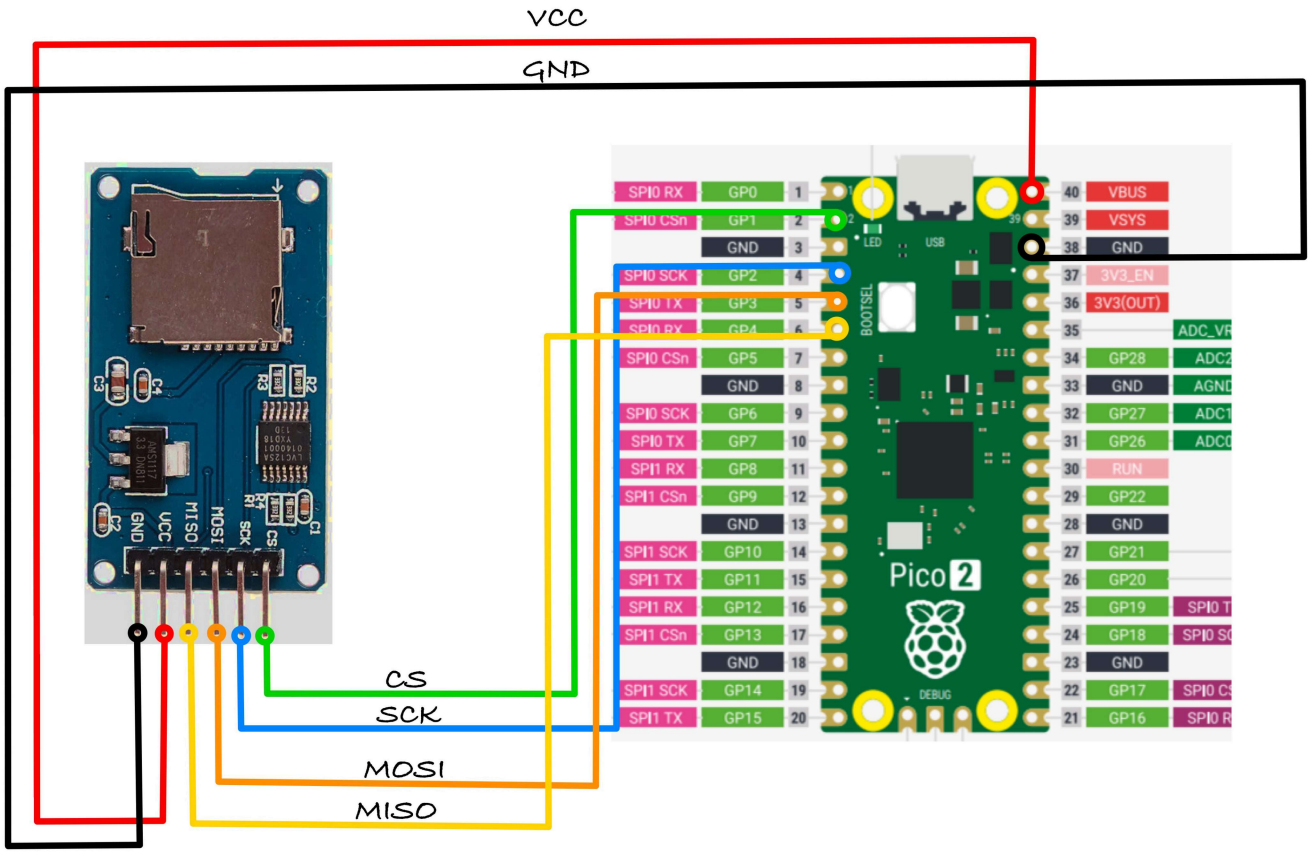


microSD Card Pin	SPI Function
1	-
2	Chip Select (CS); also referred as Card Select
3	Data Input (DI) - corresponds to MOSI. To receive data from the microcontroller.
4	VDD - Power supply (3.3V)
5	Serial Clock (SCK)
6	Ground (GND)
7	Data Output (DO) - corresponds to MISO. To send data from the microSD card to the microcontroller.
8	-

Connecting the Raspberry Pi Pico to the SD Card Reader

The microSD card operates at 3.3V, so using 5V to power it could damage the card. However, the reader module comes with an onboard voltage regulator and logic shifter, allowing it to safely be connected to the 5V power supply of the Pico.

Pico Pin	Wire	SD Card Pin
GPIO 1		CS
GPIO 2		SCK
GPIO 3		MOSI
GPIO 4		MISO
5V		VCC
GND		GND



Read SD Card with Raspberry Pi Pico

Let's create a simple program that reads a file from the SD card and outputs its content over USB serial. Make sure the SD card is formatted with FAT32 and contains a file to read (for example, "RUST.TXT" with the content "Ferris").

Project from template

To set up the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, give your project a name, like "read-sdcard" and select `RP-HAL` as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "read-sdcard":
# cd read-sdcard
```

Additional Crates required

Update your Cargo.toml to add these additional crate along with the existing dependencies.

```
// USB serial communication
usbd-serial = "0.2.2"
usb-device = "0.3.2"
heapless = "0.8.0"

// To convert Spi bus to SpiDevice
embedded-hal-bus = "0.2.0"

// sd card driver
embedded-sdmmc = "0.8.1"
```

Except for the embedded-sdmmc crate, we have already used all these crates in previous exercises.

- The usbd-serial and usb-device crates are used for sending or receiving data to and from a computer via USB serial. The heapless crate acts as a helper, providing a buffer before printing data to USB serial.
- The embedded-hal-bus crate offers the necessary traits for SPI and I²C buses, which are essential for interfacing the Pico with the SD card reader.
- The embedded-sdmmc crate is a driver for reading and writing files on FAT-formatted SD cards.

Additional imports

```
use usb_device::{class_prelude::*, prelude::*};
use usbd_serial::SerialPort;

use hal::fugit::RateExtU32;
use heapless::String;

use core::fmt::Write;

use embedded_hal_bus::spi::ExclusiveDevice;
use embedded_sdmmc::{SdCard, TimeSource, Timestamp, VolumeIdx,
VolumeManager};
```

Make sure to check out the USB serial tutorial for setting up the USB serial. We won't go over the setup here to keep it simple.

Dummy Timesource

The TimeSource is needed to retrieve timestamps and manage file metadata. Since we won't be using this functionality, we'll create a DummyTimeSource that implements the TimeSource trait. This is necessary for compatibility with the embedded-sdmmc crate.

```rust
/// Code from https://github.com/rp-rs/rp-hal-boards/blob/main/boards/rp-pico/examples/pico_spi_sd_card.rs
/// A dummy timesource, which is mostly important for creating files.
#[derive(Default)]
pub struct DummyTimesource();

impl TimeSource for DummyTimesource {
    // In theory you could use the RTC of the rp2040 here, if you had
    // any external time synchronizing device.
    fn get_timestamp(&self) -> Timestamp {
        Timestamp {
            year_since_1970: 0,
            zero_indexed_month: 0,
            zero_indexed_day: 0,
            hours: 0,
            minutes: 0,
            seconds: 0,
        }
    }
}
```

Setting Up the SPI for the SD Card Reader

Now, let's configure the SPI bus and the necessary pins to communicate with the SD Card reader.

```rust
let spi_cs = pins.gpio1.into_push_pull_output();
let spi_sck = pins.gpio2.into_function::<hal::gpio::FunctionSpi>();
let spi_mosi = pins.gpio3.into_function::<hal::gpio::FunctionSpi>();
let spi_miso = pins.gpio4.into_function::<hal::gpio::FunctionSpi>();
let spi_bus = hal::spi::Spi::<_, _, _, 8>::new(pac.SPI0, (spi_mosi, spi_miso, spi_sck));

let spi = spi_bus.init(
    &mut pac.RESETS,
    clocks.peripheral_clock.freq(),
    400.kHz(), // card initialization happens at low baud rate
    embedded_hal::spi::MODE_0,
);
```

Getting the `SpiDevice` from SPI Bus

To work with the embedded-sdmmc crate, we need an `SpiDevice`. Since we only have the SPI bus from RP-HAL, we'll use the `embedded_hal_bus` crate to get the `SpiDevice` from the SPI bus.

```
let spi = ExclusiveDevice::new(spi, spi_cs, timer).unwrap();
```

Setup SD Card driver

```
let sdcard = SdCard::new(spi, timer);
let mut volume_mgr = VolumeManager::new(sdcard,
DummyTimesource::default());
```

Print the size of the SD Card

```
match volume_mgr.device().num_bytes() {
    Ok(size) => {
        write!(buff, "card size is {} bytes\r\n", size).unwrap();
        serial.write(buff.as_bytes()).unwrap();
    }
    Err(e) => {
        write!(buff, "Error: {:?}", e).unwrap();
        serial.write(buff.as_bytes()).unwrap();
    }
}
```

Open the directory

Let's open the volume with the volume manager then open the root directory.

```
let Ok(mut volume0) = volume_mgr.open_volume(VolumeIdx(0)) else {
    let _ = serial.write("err in open_volume".as_bytes());
    continue;
};

let Ok(mut root_dir) = volume0.open_root_dir() else {
    serial.write("err in open_root_dir".as_bytes()).unwrap();
    continue;
};
```

Open the file in read-only mode

```
let Ok(mut my_file) =  root_dir.open_file_in_dir("RUST.TXT",
embedded_sdmmc::Mode::ReadOnly) else {
    serial.write("err in open_file_in_dir".as_bytes()).unwrap();
    continue;
};
```

Read the file content and print

```
while !my_file.is_eof() {
    let mut buffer = [0u8; 32];
    let num_read = my_file.read(&mut buffer).unwrap();
    for b in &buffer[0..num_read] {
        write!(buff, "{}", *b as char).unwrap();
    }
}
serial.write(buff.as_bytes()).unwrap();
```

Full code

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

use embedded_hal::delay::DelayNs;
use hal::block::ImageDef;
use panic_halt as _;
use rp235x_hal::{self as hal, Clock};

use usb_device::{class_prelude::*, prelude::*};
use usbd_serial::SerialPort;

use hal::fugit::RateExtU32;
use heapless::String;

use core::fmt::Write;

use embedded_hal_bus::spi::ExclusiveDevice;
use embedded_sdmmc::{SdCard, TimeSource, Timestamp, VolumeIdx,
VolumeManager};

#[link_section = ".start_block"]
#[used]
pub static IMAGE_DEF: ImageDef = hal::block::ImageDef::secure_exe();

const XTAL_FREQ_HZ: u32 = 12_000_000u32;

/// A dummy timesource, which is mostly important for creating files.
#[derive(Default)]
pub struct DummyTimesource();

impl TimeSource for DummyTimesource {
    // In theory you could use the RTC of the rp2040 here, if you had
    // any external time synchronizing device.
    fn get_timestamp(&self) -> Timestamp {
        Timestamp {
            year_since_1970: 0,
            zero_indexed_month: 0,
            zero_indexed_day: 0,
            hours: 0,
            minutes: 0,
            seconds: 0,
        }
    }
}

#[hal::entry]
fn main() -> ! {
    let mut pac = hal::pac::Peripherals::take().unwrap();
```

```
    let mut watchdog = hal::Watchdog::new(pac.WATCHDOG);

    let clocks = hal::clocks::init_clocks_and_plls(
        XTAL_FREQ_HZ,
        pac.XOSC,
        pac.CLOCKS,
        pac.PLL_SYS,
        pac.PLL_USB,
        &mut pac.RESETS,
        &mut watchdog,
    )
    .ok()
    .unwrap();
    let mut timer = hal::Timer::new_timer0(pac.TIMER0, &mut
pac.RESETS, &clocks);

    let sio = hal::Sio::new(pac.SIO);
    let pins = hal::gpio::Pins::new(
        pac.IO_BANK0,
        pac.PADS_BANK0,
        sio.gpio_bank0,
        &mut pac.RESETS,
    );

    let usb_bus = UsbBusAllocator::new(hal::usb::UsbBus::new(
        pac.USB,
        pac.USB_DPRAM,
        clocks.usb_clock,
        true,
        &mut pac.RESETS,
    ));

    let mut serial = SerialPort::new(&usb_bus);

    let mut usb_dev = UsbDeviceBuilder::new(&usb_bus,
UsbVidPid(0x16c0, 0x27dd))
        .strings(&[StringDescriptors::default()
            .manufacturer("implRust")
            .product("Ferris")
            .serial_number("TEST")])
        .unwrap()
        .device_class(2) // 2 for the CDC, from:
https://www.usb.org/defined-class-codes
        .build();

    let spi_cs = pins.gpio1.into_push_pull_output();
    let spi_sck = pins.gpio2.into_function::<hal::gpio::FunctionSpi>
();
    let spi_mosi = pins.gpio3.into_function::<hal::gpio::FunctionSpi>
();
    let spi_miso = pins.gpio4.into_function::<hal::gpio::FunctionSpi>
();
```

```
    let spi_bus = hal::spi::Spi::<_, _, _, 8>::new(pac.SPI0,
(spi_mosi, spi_miso, spi_sck));

    let spi = spi_bus.init(
        &mut pac.RESETS,
        clocks.peripheral_clock.freq(),
        400.kHz(), // card initialization happens at low baud rate
        embedded_hal::spi::MODE_0,
    );

    let spi = ExclusiveDevice::new(spi, spi_cs, timer).unwrap();
    let sdcard = SdCard::new(spi, timer);
    let mut buff: String<64> = String::new();

    let mut volume_mgr = VolumeManager::new(sdcard,
DummyTimesource::default());

    let mut is_read = false;
    loop {
        let _ = usb_dev.poll(&mut [&mut serial]);
        if !is_read && timer.get_counter().ticks() >= 2_000_000 {
            is_read = true;
            serial
                .write("Init SD card controller and retrieve card
size...".as_bytes())
                .unwrap();
            match volume_mgr.device().num_bytes() {
                Ok(size) => {
                    write!(buff, "card size is {} bytes\r\n",
size).unwrap();
                    serial.write(buff.as_bytes()).unwrap();
                }
                Err(e) => {
                    write!(buff, "Error: {:?}", e).unwrap();
                    serial.write(buff.as_bytes()).unwrap();
                }
            }
            buff.clear();

            let Ok(mut volume0) = volume_mgr.open_volume(VolumeIdx(0))
else {
                let _ = serial.write("err in open_volume".as_bytes());
                continue;
            };

            let Ok(mut root_dir) = volume0.open_root_dir() else {
                serial.write("err in
open_root_dir".as_bytes()).unwrap();
                continue;
            };

            let Ok(mut my_file) =
```

```
                root_dir.open_file_in_dir("RUST.TXT",
embedded_sdmmc::Mode::ReadOnly)
            else {
                serial.write("err in
open_file_in_dir".as_bytes()).unwrap();
                continue;
            };

            while !my_file.is_eof() {
                let mut buffer = [0u8; 32];
                let num_read = my_file.read(&mut buffer).unwrap();
                for b in &buffer[0..num_read] {
                    write!(buff, "{}", *b as char).unwrap();
                }
            }
            serial.write(buff.as_bytes()).unwrap();
        }
        buff.clear();

        timer.delay_ms(50);
    }
}

#[link_section = ".bi_entries"]
#[used]
pub static PICOTOOL_ENTRIES: [hal::binary_info::EntryAddr; 5] = [
    hal::binary_info::rp_cargo_bin_name!(),
    hal::binary_info::rp_cargo_version!(),
    hal::binary_info::rp_program_description!(c"USB Fun"),
    hal::binary_info::rp_cargo_homepage_url!(),
    hal::binary_info::rp_program_build_attribute!(),
];
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `read-sdcard` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/read-sdcard/
```

How to Run ?

The method to flash (run the code) on the Pico is the same as usual. However, we need to set up tio to interact with the Pico through the serial port (/dev/ttyACM0). This allows us to read data from the Pico or send data to it.

tio

Make sure you have tio installed on your system. If not, you can install it using:

```
apt install tio
```

Connecting to the Serial Port

Run the following command to connect to the Pico's serial port:

```
tio /dev/ttyACM0
```

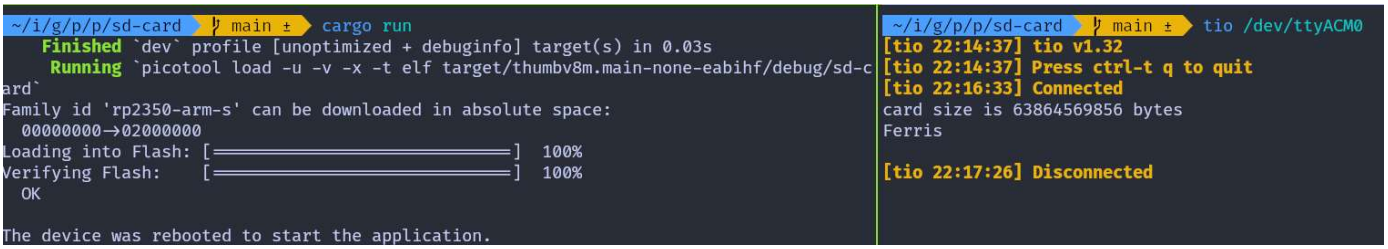
This will open a terminal session for communicating with the Pico.

Flashing and Running the Code

Open another terminal, navigate to the project folder, and flash the code onto the Pico as usual:

```
cargo run
```

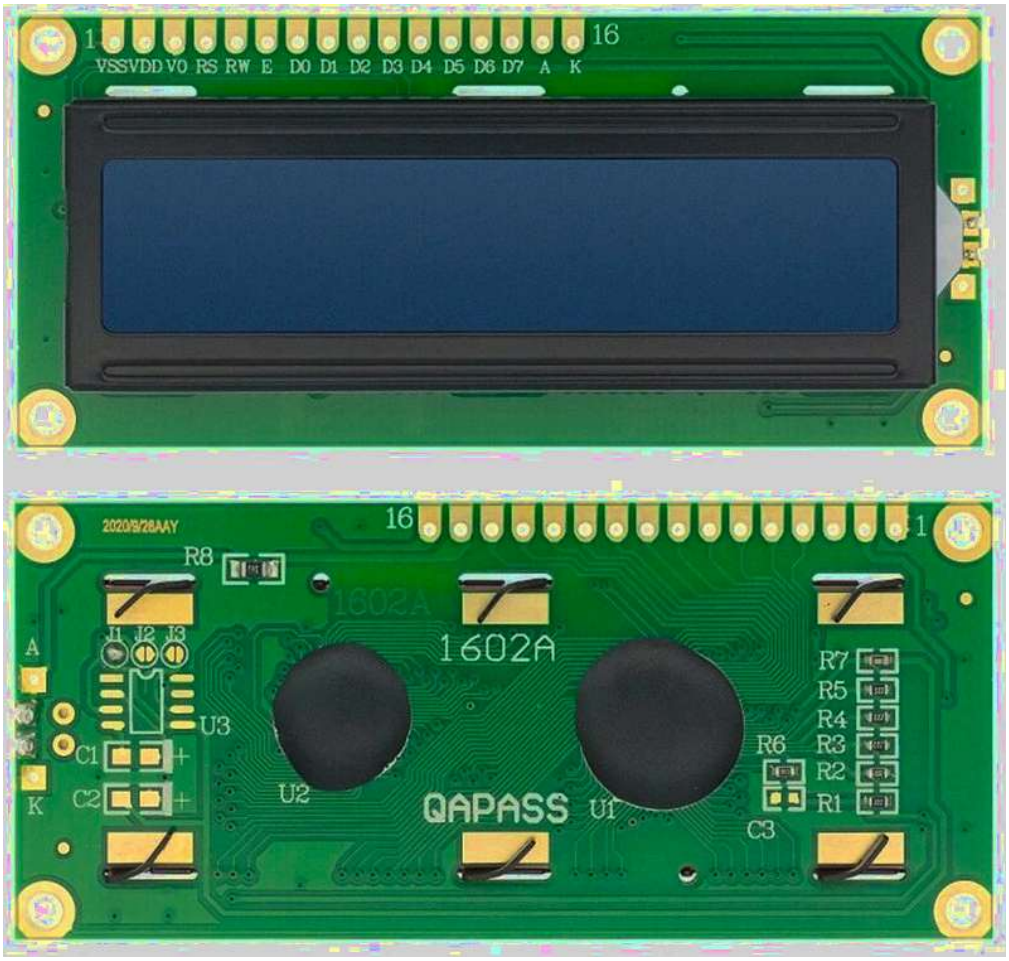
If everything is set up correctly, you should see a "Connected" message in the tio terminal. It will then print the card size and the content of the file once the timer's ticks reach 2,000,000.



LCD Display

In this section, we will be using Hitachi HD44780 compatible LCD (Liquid Crystal Display) displays. You might have seen them in devices like printers, digital clocks, microwaves, washing machines, air conditioners, and other home appliances. They're also used in equipment like copiers, fax machines, and routers.

You can display ASCII character and up to 8 custom characters.



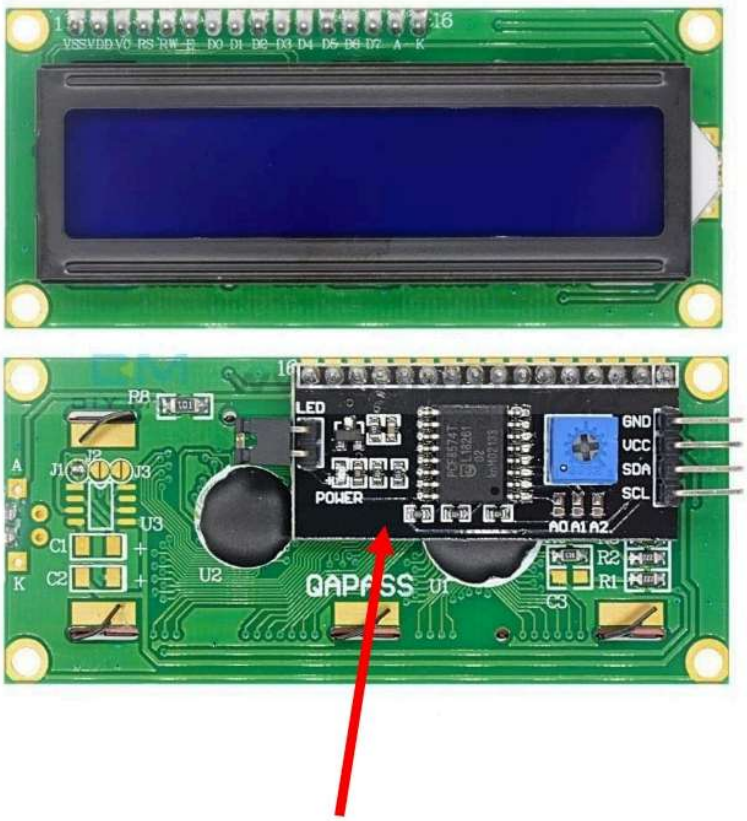
Variants

It comes in various variants, such as 16x2 (16 columns, 2 rows) and 20x4 (20 columns, 4 rows), and also based on backlight color (blue, yellow, or green). The one I have displays white characters with a blue backlight. However, you can choose any variant as it won't significantly affect the code. Most of these variants will have 16 pins.

I2C variants

Some variants come with an I2C interface adapter, so you can use I2C for communication. The main advantage of I2C variant is that it reduces the number of pin connections. However, we'll be working with the parallel interface instead.

You can also buy the I2C interface adapter separately and solder it later.



Hardware Requirements

- **LCD Display (LCD1602):** I would recommend getting a 16x2 so you can follow along.
- **Potentiometer (Optional):** This is used to control the LCD contrast. I didn't have one at the moment, so I used resistors instead.
- **Resistors (Optional):** Two 1K resistors. If you have a potentiometer, you can ignore this. I used these to control the contrast.
- **Jump Wires:** A lot! We'll need around 15+ jump wires since we're using a parallel interface (not I2C), which requires connecting many pins.

Datasheet

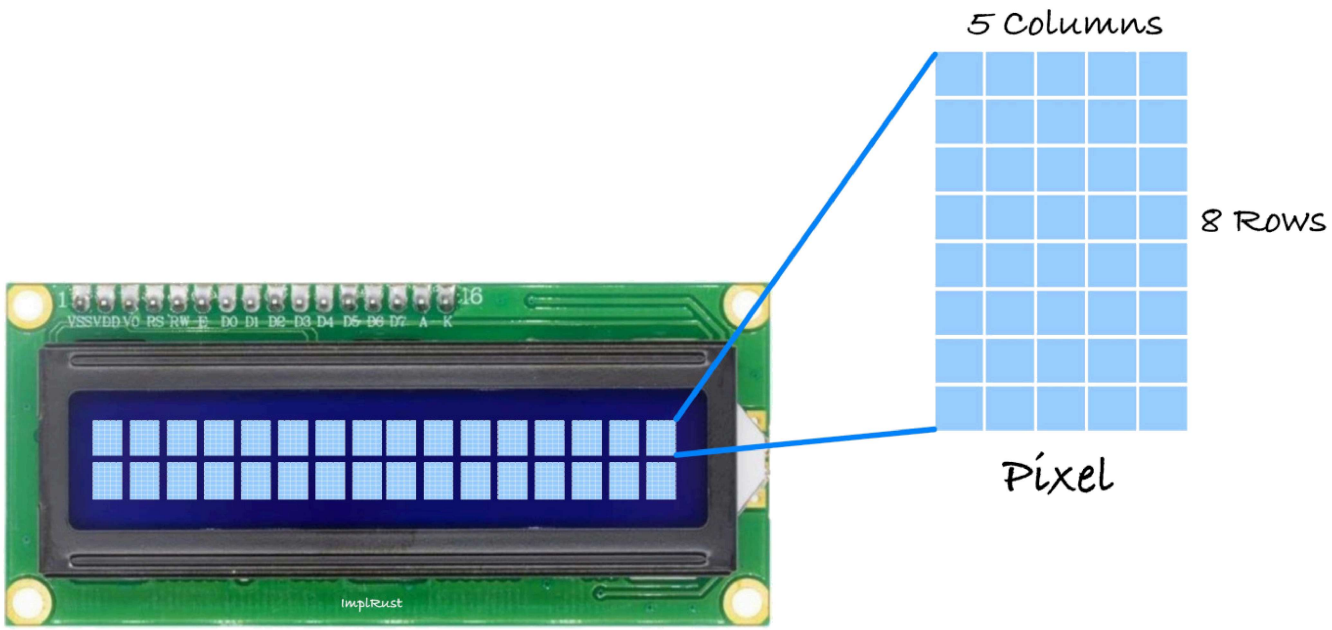
- You can access the datasheet for the HD44780 from Sparkfun or MIT site
- LCD Driver Data Book
- LCD Module 1602A Datasheet

How it works?

A Liquid Crystal Display (LCD) uses liquid crystals to control light. When electricity is applied, the crystals change orientation, either allowing light to pass through or blocking it, creating images or text. A backlight illuminates the screen, and colored sub-pixels (red, green, and blue) combine to form various colors. The crystals can also turn opaque in specific areas, blocking the backlight and creating dark regions to display characters.

16x2 LCD Display and 5x8 Pixel Matrix

A 16x2 LCD has 2 rows and 16 columns, allowing it to display 32 characters in total. Each character is made up of a 5x8 pixel grid, where 5 columns and 8 rows of pixels form the shape of the character. This grid is used to display text and simple symbols on the screen.



Displaying Text and Custom Characters on 16x2 LCD

We don't have to manually draw the pixels; This is taken care of by the HD44780 IC, which automatically maps ASCII characters to the 5x8 pixel grid.

However, if you'd like to create custom characters or symbols, you will need to define the 5x8 pixel pattern yourself. This pattern is saved in the LCD's memory, and once it's defined, you can use the custom character. Keep in mind, only up to 8 custom characters can be stored at a time.

Data transfer mode

The LCM (Liquid Crystal Module) supports two types of data transfer modes: 8-bit and 4-bit. In 8-bit mode, data is sent as a full byte using all the data pins. In 4-bit mode, only the higher-order data bits are used, sending data in nibbles. While 8-bit mode is faster, it comes with a trade-off;using too many wires, which can quickly exhaust the GPIO pins on a microcontroller. To minimize wiring, we'll use 4-bit mode.

Reference:

- 16x2 character LCD dot matrix module

Pin Layout

The LCD has a total of 16 pins for the parallel interface.

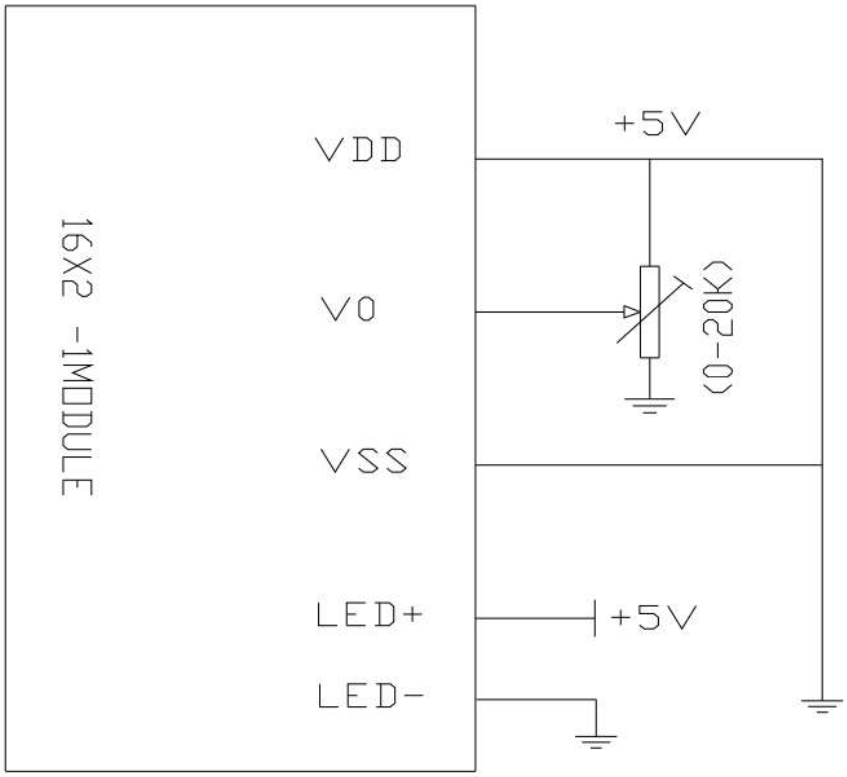
Pin Position	LCD Pin	Details
1	***VSS***	Should be connected to the Ground.
2	***VSS***	Power supply (5V) for the logic
3	***V_O***	Contrast adjustment: - If you use a potentiometer (10k), connect the middle pin to adjust the contrast. Other pins of the potentiometer should be connected to 5V (or 3.3V) and GND. - I used two 1k resistors instead, which was sufficient for this exercise.
4	***RS***	Register select pin: - Set LOW (RS = 0) to send commands to the LCD. - Set HIGH (RS = 1) to send data to the LCD.
5	***RW***	Read/Write pin: - Set LOW (RW = 0) to write to the LCD, which is what we will mostly do. - Set HIGH (RW = 1) to read from the LCD (rarely used).

Pin Position	LCD Pin	Details
		- We will connect this to Ground since we're only writing.
6	***E***	The Enable pin is pulsed high and then brought back to low (ground) to trigger the LCD to accept and process data.
7-10	***D0 - D3***	These are the 4 lower-order data bits, used only in 8-bit mode. If you are using 4-bit mode, leave these pins unconnected.
11-14	***D4 - D7***	These are the 4 higher-order data bits, used to represent the data in 4-bit mode.
15	***A***	Anode of the backlight. Should be connected to 5V.
16	***K***	Cathode of the backlight. Should be connected to Ground.

Contrast Adjustment

The V_O pin controls the contrast of the LCD.

According to the datasheet of the LCD1602A, the **Vo** pin controls the contrast of the LCD by adjusting the operating voltage for the LCD, which is the difference between the power supply for the logic (**VDD**) and the contrast control pin (**Vo**). When **Vo** is closer to ground, the voltage difference (**VDD - Vo**) is larger, resulting in a higher contrast, making the text on the screen more distinct and readable. When **Vo** is closer to **VDD**, the voltage difference decreases, resulting in a lower contrast, causing the text to appear faded or less visible.



You can use the potentiometer to adjust the contrast on the fly. You have to connect the middle pin of the potentiometer to Vo, and the other two pins to VCC and Ground.

You can also use resistors to adjust the contrast, which is what I did. You need to adjust the contrast one way or another. The first time I ran the program, I couldn't see the text clearly. I placed two 1k resistors(when I added only one 1k resistor, it didn't look that great) between Ground and the Vo, and then the text became visible.

Register Select Pin (RS)

The Register Select (RS) pin determines whether the LCD is in command mode or data mode.

When it is in Low(RS = 0), the LCD is in command mode, where the input is interpreted as a command, such as clearing the display or setting the cursor position (e.g., sending a command to clear the display).

When it is in High(RS = 1), the LCD is in data mode, where the input is interpreted as data to be displayed on the screen (e.g., sending text to display).

Enable Pin (E)

It is used to control when data is transferred to the LCD display. The enable pin is typically kept low (E=0) but is set high (E=1) for a specific period of time to initiate a data transfer, and then returned to low.. The data is latched into the LCD on the transition from high to low.

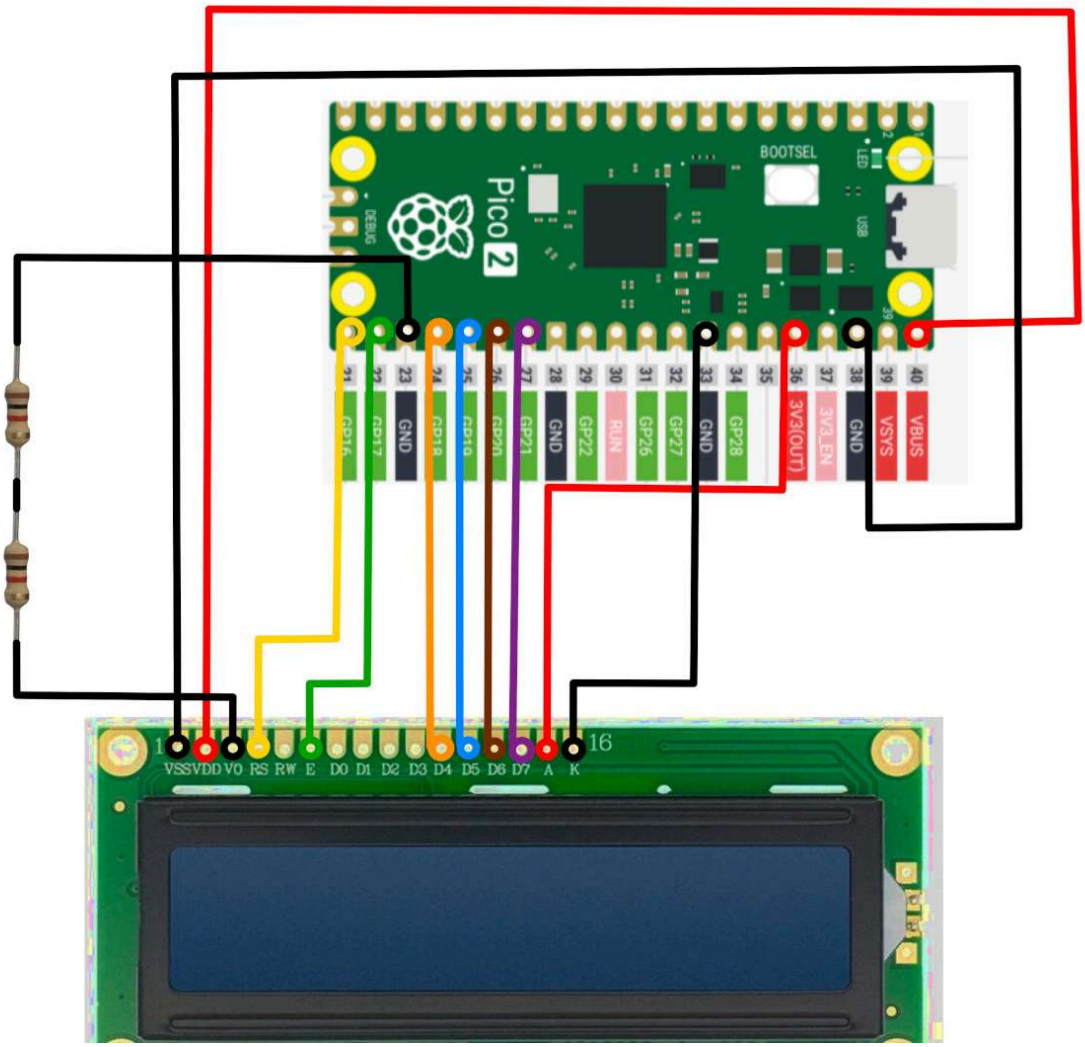
Circuit

Connecting LCD Display (LCD1602) to the Raspberry Pi Pico

We will be using parallel interface in 4bit mode. Remaining Pins like D0 to D3 won't be connected.

LCD Pin	Wire	Pico Pin	Notes
VSS		GND	Ground
VDD		VBUS (5V)	Power Supply
V_O		GND or Potentiometer	You can use Potentiometer or resistors to adjust the contrast. I placed two 1K resistors in between Ground and V_O
RS		GPIO 16	Register Select (0 = command, 1 = data)
RW		GND	Read/Write. Set '0' to write to display. If you want to read from display, set '1'
EN		GPIO 17	Enable
D4		GPIO 18	Data Bit 4
D5		GPIO 19	Data Bit 5

LCD Pin	Wire	Pico Pin	Notes
D6		GPIO 20	Data Bit 6
D7		GPIO 21	Data Bit 7
A		3V3(OUT)	LED Backlight +
K		GND	LED Backlight -



"Hello, Rust!" in LCD Display

In this program, we will just print "Hello, Rust!" text in the LCD display.

HD44780 Drivers

During my research, I came across many Rust crates for controll the LCD Display, but these two stood out as working well. In this program, we will start by using the `hd44780-driver` crate.

- hd44780-driver
- liquid_crystal

Project from template

To set up the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, give your project a name, like "lcd-hello" and select `RP-HAL` as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "lcd-hello":
# cd lcd-hello
```

Additional Crates required

Update your Cargo.toml to add these additional crate along with the existing dependencies.

```
hd44780-driver = "0.4.0"
```

Additional imports

```
use hd44780_driver::HD44780;
```

Mapping Pico and LCD Pins

We connect GPIO16 to the RS pin, GPIO17 to the Enable (E) pin, and GPIO18-21 to the D4-D7 data pins of the LCD. We're using only 4 data pins since we will be working on 4-bit mode.

```
// Read Select Pin
let rs = pins.gpio16.into_push_pull_output();

// Enable Pin
let en = pins.gpio17.into_push_pull_output();

// Data Pins
let d4 = pins.gpio18.into_push_pull_output();
let d5 = pins.gpio19.into_push_pull_output();
let d6 = pins.gpio20.into_push_pull_output();
let d7 = pins.gpio21.into_push_pull_output();
```

Write Text to the LCD

Here, we initialize the LCD module, clear the screen, and then write the text "Hello, Rust!".

```rust
// LCD Init
let mut lcd = HD44780::new_4bit(rs, en, d4, d5, d6, d7, &mut
timer).unwrap();

// Clear the screen
lcd.reset(&mut timer).unwrap();
lcd.clear(&mut timer).unwrap();

// Write to the top line
lcd.write_str("Hello, Rust!", &mut timer).unwrap();
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `lcd-hello` folder.

```bash
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/lcd-hello/
```

Supported Characters

When referring to the HD44780 datasheet, you'll find two character set tables corresponding to two different ROM versions(A00 and A02). To determine which ROM your display uses, try unique characters from both tables. The one that displays correctly indicates the ROM version. Once identified, you only need to refer to the relevant table.

In my case, the LCD module I'm using is based on ROM version A00. I'll present the A00 table and explain how to interpret it, though the interpretation logic is the same for both versions.

Table 4 Correspondence between Character Codes and Character Patterns (ROM Code: A00)

Lower 4 Bits \ Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	ß	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	Ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	q	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	×	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		−	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	■

Note: The user can specify any pattern for character-generator RAM.

It's an 8-bit character, where the upper 4 bits come first, followed by the lower 4 bits, to form the complete character byte. In the reference table, the upper 4 bits correspond to the columns, while the lower 4 bits correspond to the rows.

For example, to get the binary representation of the character "#," the upper 4 bits are 0010, and the lower 4 bits are 0011. Combining them gives the full

binary value 00100011. In Rust, you can represent this value either in binary (0b00100011) or as a hexadecimal (0x23).

hd44780-driver crate

In the `hd44780-driver` crate we are using, we can write characters directly as a single byte or a sequence of bytes.

Write single byte

```
lcd.write_byte(0x23, &mut timer).unwrap();
lcd.write_byte(0b00100011, &mut timer).unwrap();
```

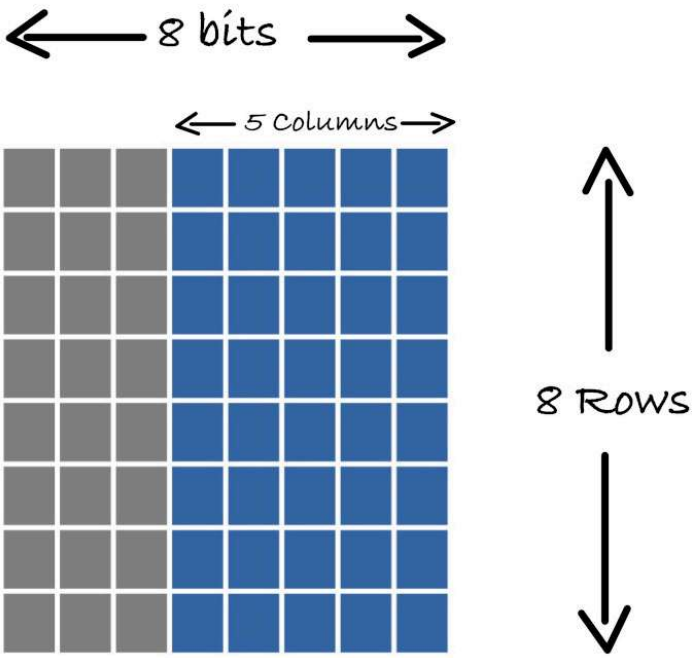
Write multiple bytes

```
lcd.write_bytes(&[0x23, 0x24], &mut timer).unwrap();
```

Custom Characters

Besides the supported characters, you can create your own custom ones, like smileys or heart symbols. The module includes 64 bytes of Character Generator RAM (CGRAM), allowing up to 8 custom characters.

Each character is an 8x8 grid, where each row is represented by a single 8-bit value (
`u8`
). This makes it 8 bytes per character (8 rows × 1 byte per row). That's why, with a total of 64 bytes, you can only store up to 8 custom characters (8 chars × 8 bytes = 64 bytes).



Note: If you recall, in our LCD module, each character is represented as a 5x8 grid. But wait, didn't we say we need an 8x8 grid for the characters? Yes, that's correct-we need 8 x 8 (8 bytes) memory, but we only use 5 bits in each row. The 3 high-order bits in each row are left as zeros.

LCD Custom Character Generator (5x8 Grid)

Select the grids to create a symbol or character. As you select the grid, the corresponding bits in the byte array will be updated.

Generated Array

```
[
    0b00000,
    0b00000,
    0b00000,
    0b00000,
    0b00000,
    0b00000,
    0b00000,
    0b00000,
]
```

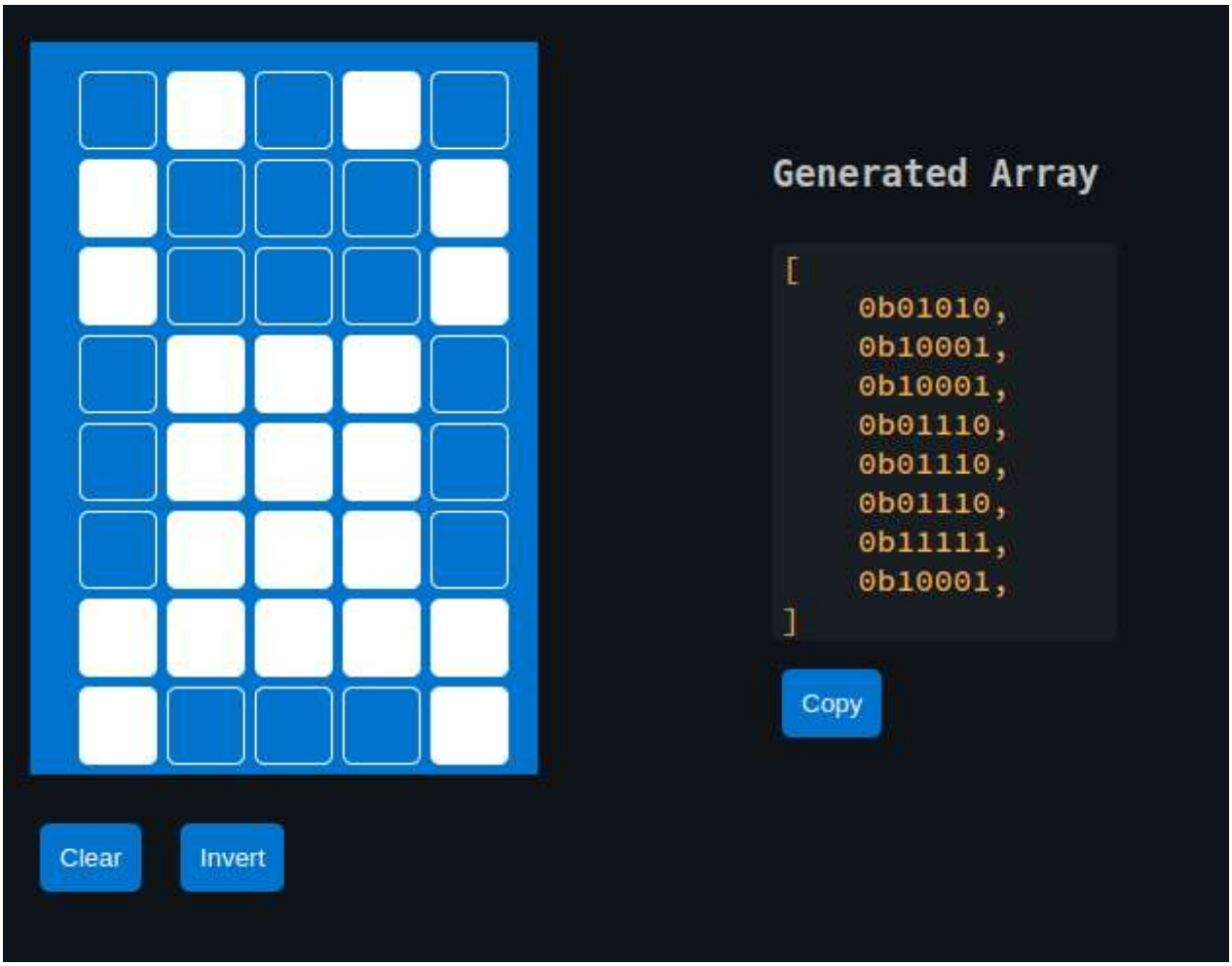
Copy

Clear Invert

Ferris on LCD Display

Let's create Ferris (my attempt to make it look like a crab; if you have a better design, feel free to send a pull request) using a single character. In fact, we can combine 4 or 6 adjacent grids to display a single symbol. Creativity is up to you, and you can improve it however you like.

We'll use the custom character generator from the previous page to create this symbol. This will give us the byte array that we can use.



Note that the previous crate hd44780-driver doesn't support custom characters. To handle this, we can use the liquid_crystal crate, which allows us to work with custom characters.

Initialize the LCD interface

```
let mut lcd_interface = Parallel::new(d4, d5, d6, d7, rs, en,
lcd_dummy);
let mut lcd = LiquidCrystal::new(&mut lcd_interface, Bus4Bits,
LCD16X2);
lcd.begin(&mut timer);
```

Our generated byte array for the custom character

```
    const FERRIS: [u8; 8] = [
        0b01010, 0b10001, 0b10001, 0b01110, 0b01110, 0b01110, 0b11111,
0b10001,
    ];
    // Define the character
    lcd.custom_char(&mut timer, &FERRIS, 0);
```

Displaying

Displaying the character is straightforward. You just need to use the CustomChar enum and pass the index of the custom character. We've defined only one custom character, which is at position 0.

```
    lcd.write(&mut timer, CustomChar(0));
    lcd.write(&mut timer, Text(" implRust!"));
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `lcd-custom` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/lcd-custom/
```

Multi Custom Character Generator

This is used when you want to combine multiple grids to create a symbol. You can utilize adjacent grids on the 16x2 LCD display to design a custom symbol or character. You can view the example symbol created with this generator and how to use in Rust in the next page.

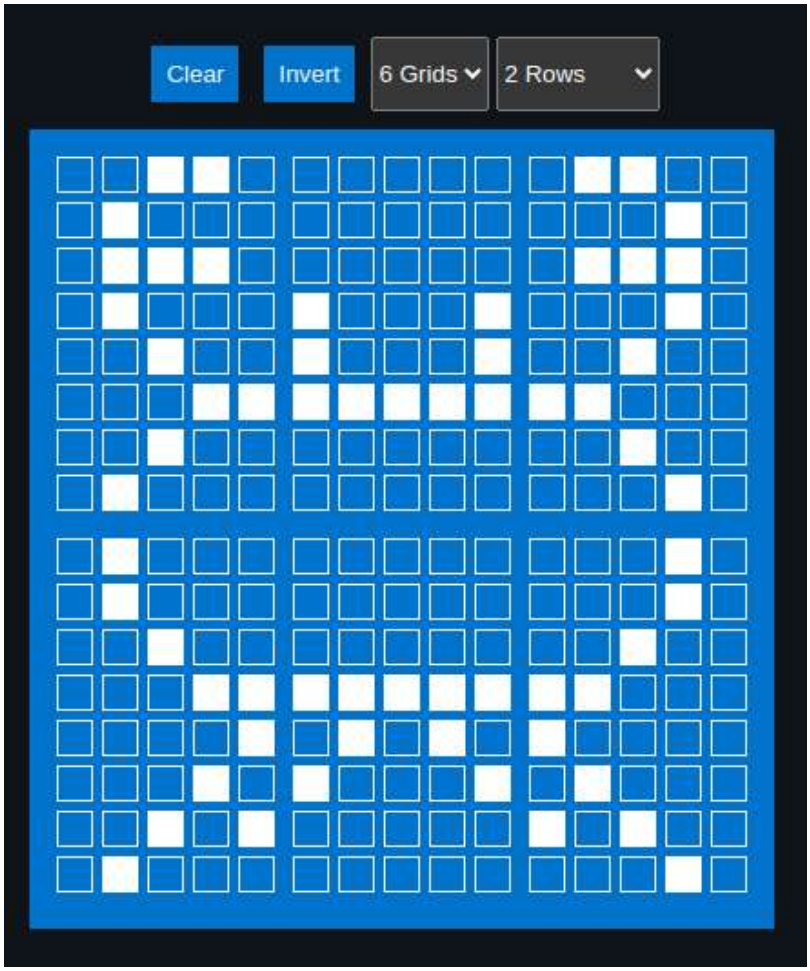
Clear Invert 2 Grids Single Row

Generated Array

Copy

Multi Custom character

I attempted to create the Ferris image using 6 adjacent grids with the generator from the previous page. Here's the Rust code to utilize those byte arrays.



Generated Byte array for the characters

```
const SYMBOL1: [u8; 8] = [
    0b00110, 0b01000, 0b01110, 0b01000, 0b00100, 0b00011, 0b00100,
0b01000,
];

const SYMBOL2: [u8; 8] = [
    0b00000, 0b00000, 0b00000, 0b10001, 0b10001, 0b11111, 0b00000,
0b00000,
];

const SYMBOL3: [u8; 8] = [
    0b01100, 0b00010, 0b01110, 0b00010, 0b00100, 0b11000, 0b00100,
0b00010,
];

const SYMBOL4: [u8; 8] = [
    0b01000, 0b01000, 0b00100, 0b00011, 0b00001, 0b00010, 0b00101,
0b01000,
];

const SYMBOL5: [u8; 8] = [
    0b00000, 0b00000, 0b00000, 0b11111, 0b01010, 0b10001, 0b00000,
0b00000,
];

const SYMBOL6: [u8; 8] = [
    0b00010, 0b00010, 0b00100, 0b11000, 0b10000, 0b01000, 0b10100,
0b00010,
];
```

Declare them as character

```
lcd.custom_char(&mut timer, &SYMBOL1, 0);
lcd.custom_char(&mut timer, &SYMBOL2, 1);
lcd.custom_char(&mut timer, &SYMBOL3, 2);
lcd.custom_char(&mut timer, &SYMBOL4, 3);
lcd.custom_char(&mut timer, &SYMBOL5, 4);
lcd.custom_char(&mut timer, &SYMBOL6, 5);
```

Display

Let's write the first 3 grids into the first row, then the second half into the second row of the LCD display.

```
lcd.set_cursor(&mut timer, 0, 4)
    .write(&mut timer, CustomChar(0))
    .write(&mut timer, CustomChar(1))
    .write(&mut timer, CustomChar(2));

lcd.set_cursor(&mut timer, 1, 4)
    .write(&mut timer, CustomChar(3))
    .write(&mut timer, CustomChar(4))
    .write(&mut timer, CustomChar(5));
```

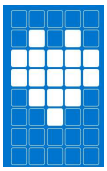
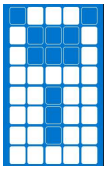
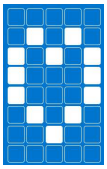
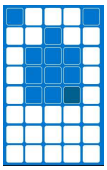
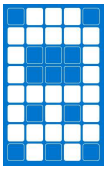
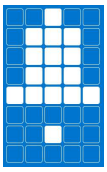
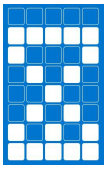
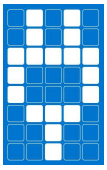
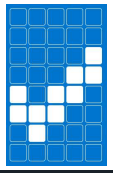
Clone the existing project

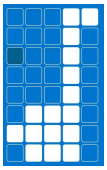
You can clone (or refer) project I created and navigate to the `lcd-custom-multi` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/lcd-custom-multi/
```

Symbols Index

Here is a list of custom symbols with their corresponding byte arrays. If you've designed an interesting symbol and want to add to this list, feel free to submit a pull request. Please use the custom character generator provided here to ensure consistency.

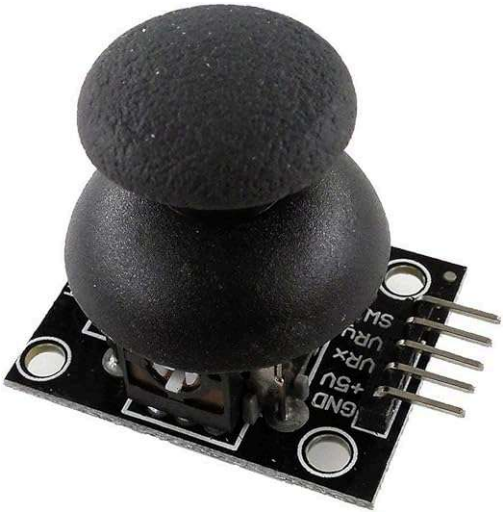
Title	Preview	Byte Array
Heart		[0b00000, 0b01010, 0b11111, 0b11111, 0b01110, 0b00100, 0b00000, 0b00000,]
Lock		[0b01110, 0b10001, 0b10001, 0b11111, 0b11011, 0b11011, 0b11011, 0b11111,]
Hollow Heart		[0b00000, 0b01010, 0b10101, 0b10001, 0b10001, 0b01010, 0b00100, 0b00000,]
Battery		[0b01110, 0b11011, 0b10001, 0b10001, 0b10001, 0b11111, 0b11111, 0b11111,]
Bus		[0b01110, 0b11111, 0b10001, 0b10001, 0b11111, 0b10101, 0b11111, 0b01010,]
Bell		[0b00100, 0b01110, 0b01110, 0b01110, 0b11111, 0b00000, 0b00100, 0b00000,]
Hour Glass		[0b00000, 0b11111, 0b10001, 0b01010, 0b00100, 0b01010, 0b10101, 0b11111,]
Charger		[0b01010, 0b01010, 0b11111, 0b10001, 0b10001, 0b01110, 0b00100, 0b00100,]
Tick Mark		[0b00000, 0b00000, 0b00001, 0b00011, 0b10110, 0b11100, 0b01000, 0b00000,]

Title	Preview	Byte Array
Music Note		[0b00011, 0b00010, 0b00010, 0b00010, 0b00010, 0b01110, 0b11110, 0b01110,]

Joystick

In this section, we'll explore how to use the Joystick Module. It is similar to the joysticks found on PS2 (PlayStation 2) controllers. They are commonly used in gaming, as well as for controlling drones, remote-controlled cars, robots, and other devices to adjust position or direction.

Meet the hardware - Joystick module

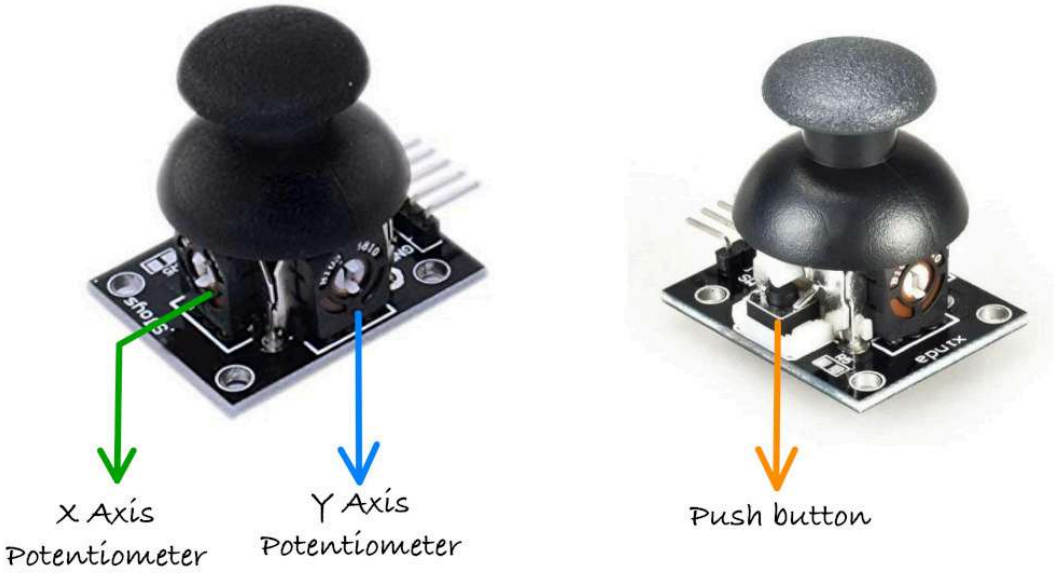


You can move the joystick knob vertically and horizontally, sending its position (X and Y axes) to the MCU (e.g., Pico). Additionally, the knob can be pressed down like a button. The joystick typically operates at 5V, but it can also be connected to 3.3V.

How it works?

The joystick module has two 10K potentiometers: one for the X-axis and another for the Y-axis. It also includes a push button, which is visible.

When you move the joystick from right to left or left to right(X axis), you can observe one of the potentiometers moving accordingly. Similarly, when you move it up and down(Y-axis), you can observe the other potentiometer moving along.



You can also observe the push-button being pressed when you press down on the knob.

Joystick Movement and Corresponding ADC Values

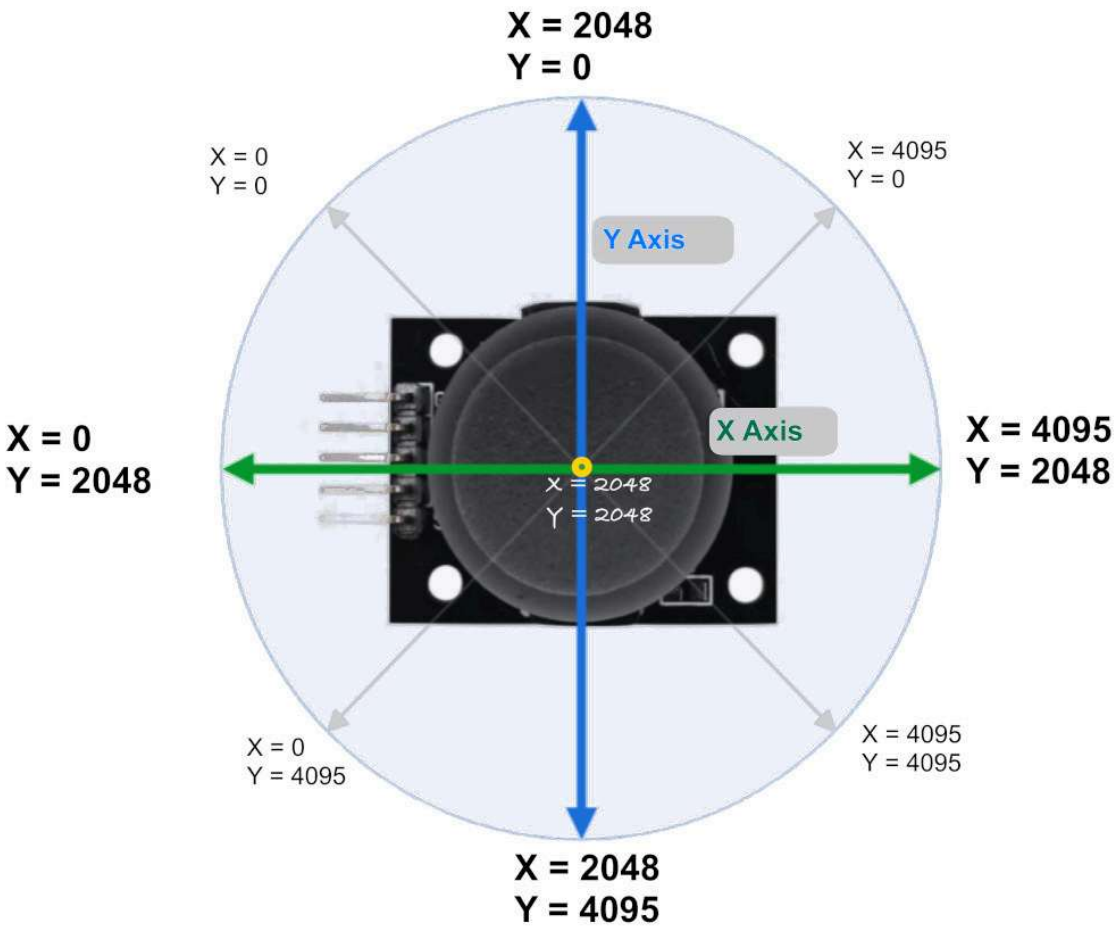
When you move the joystick along the X or Y axis, it produces an analog signal with a voltage that varies between 0 and 3.3V(or 5V if we connect it to 5V supply). When the joystick is in its center (rest) position, the output voltage is approximately 1.65V, which is half of the VCC(VCC is 3.3V in our case).

Voltage Divider

The reason it is 1.65V in the center position is that the potentiometer acts as a voltage divider. When the potentiometer is moved, its resistance changes, causing the voltage divider to output a different voltage accordingly. Refer the voltate divider section.

The joystick has a total of 5 pins, and we will shortly discuss what each of them represents. Out of these, two pins are dedicated to sending the X and Y axis positions, which should be connected to the ADC pins of the microcontroller.

As you may already know, the Raspberry Pi Pico has a 12-bit SAR-type ADC, which converts analog signals (voltage differences) into digital values. Since it is a 12-bit ADC, the analog values will be represented as digital values ranging from 0 to 4095. If you're not familiar with ADC, refer to the ADC section that we covered earlier.



Note:

The ADC values in the image are just approximations to give you an idea and won't be exact. For example, I got around 1850 for X and Y at the center position. When I moved the knob toward the pinout side, X went to 0, and when I moved it to the opposite side, it went to 4095. The same applies to the Y axis.So, You might need to calibrate your joystick.

Pin layout

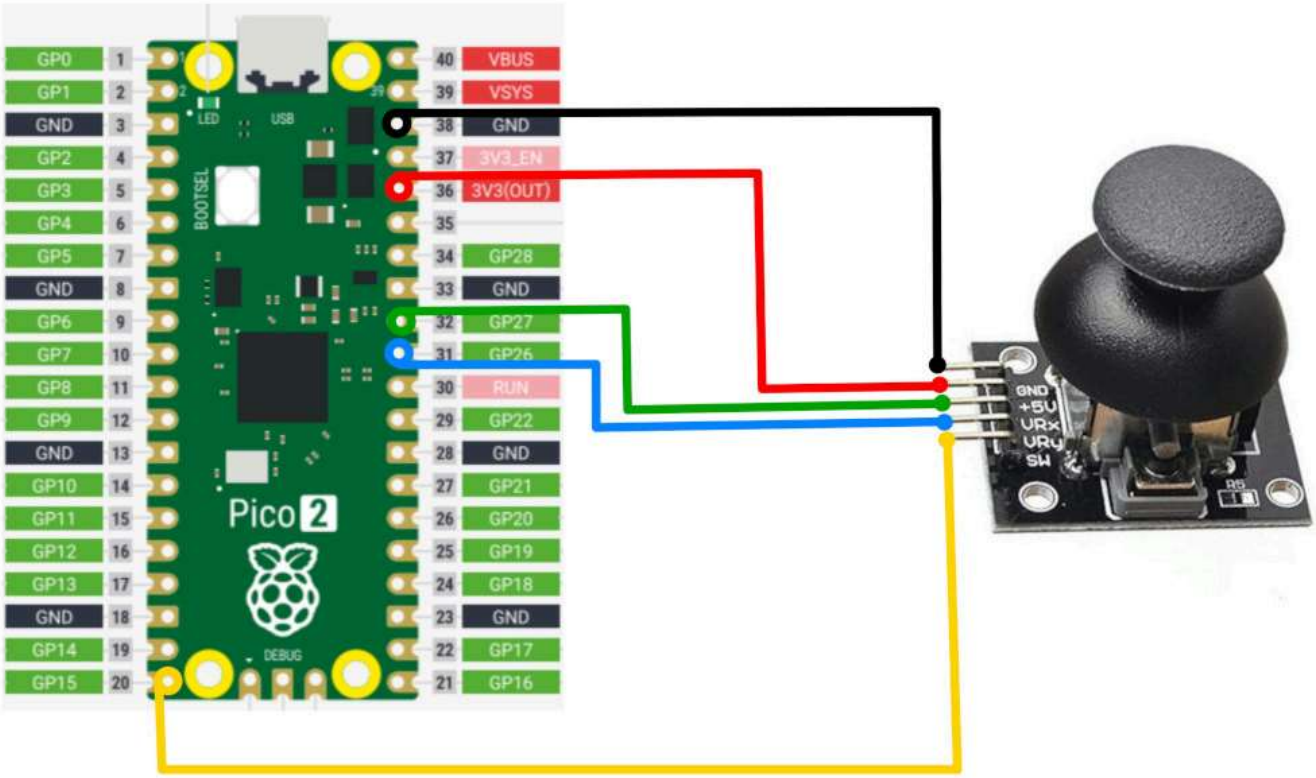
The joystick has a total of 5 pins: power supply, ground, X-axis output, Y-axis output, and switch output pin.

Joystick Pin	Details
GND	Ground pin. Should be connected to the Ground of the circuit.
VCC	Power supply pin (typically 5V or 3.3V).
VRX	The X-axis analog output pin varies its voltage based on the joystick's horizontal position, ranging from 0V to VCC as the joystick is moved left and right.
VRY	The Y-axis analog output pin varies its voltage based on the joystick's vertical position, ranging from 0V to VCC as the joystick is moved up and down.
SW	Switch pin. When the joystick knob is pressed, this pin is typically pulled LOW (to GND).

Connecting the Joystick to the Raspberry Pi Pico

Let's connect the joystick to the Raspberry Pi Pico. We need to connect the VRX and VRY pins to the ADC pins of the Pico. The joystick will be powered with 3.3V instead of 5V because the Pico's GPIO pins are only 3.3V tolerant. Connecting it to 5V could damage the Pico's pins. Thankfully, the joystick can operate at 3.3V as well.

Pico Pin	Wire	Joystick Pin
GND		GND
3.3V		VCC
GPIO 27 (ADC1)		VRX
GPIO 26 (ADC0)		VRY
GPIO 15		SW



Sending Joystick Movement ADC Values to USB Serial

In this program, we'll observe how joystick movement affects ADC values in real time. We will connect the Raspberry Pi Pico with the joystick and set up USB serial communication. If you're not sure how to set up USB Serial, check the USB Serial section.

As you move the joystick, the corresponding ADC values will be printed in the system. You can compare these values with the previous Movement and ADC Diagram;they should approximately match the values shown. Pressing the joystick knob will print **"Button Pressed"** along with the current coordinates.

Project from template

To set up the project, run:

```
cargo generate --git https://github.com/ImplFerris/pico2-template.git
```

When prompted, give your project a name, like "joystick-usb" and select `RP-HAL` as the HAL.

Then, navigate into the project folder:

```
cd PROJECT_NAME
# For example, if you named your project "joystick-usb":
# cd joystick-usb
```

Additional Crates required

Update your Cargo.toml to add these additional crate along with the existing dependencies.

```
usb-device = "0.3.2"
usbd-serial = "0.2.2"
heapless = "0.8.0"

embedded_hal_0_2 = { package = "embedded-hal", version = "0.2.5",
features = [
  "unproven",
] }
```

The first three should be familiar by now; they set up USB serial communication so we can send data between the Pico and the computer. heapless is a helper function for buffers.

embedded_hal_0_2 is the new crate. You might already have embedded-hal with version "1.0.0" in your Cargo.toml. So, you may wonder why we need this version. The reason is that Embedded HAL 1.0.0 doesn't include an ADC trait to read ADC values, and the RP-HAL uses the one from version 0.2. (Don't remove the existing embedded-hal 1.0.0; just add this one along with it.)

Additional imports

```
/// This trait is the interface to an ADC that is configured to read a
specific channel at the time
/// of the request (in contrast to continuous asynchronous sampling).
use embedded_hal_0_2::adc::OneShot;

// for USB Serial
use usb_device::{class_prelude::*, prelude::*};
use usbd_serial::SerialPort;
use heapless::String;
```

USB Serial

Make sure you've completed the USB serial section and added the boilerplate code from there into your project.

```
    let usb_bus = UsbBusAllocator::new(hal::usb::UsbBus::new(
        pac.USB,
        pac.USB_DPRAM,
        clocks.usb_clock,
        true,
        &mut pac.RESETS,
    ));

    let mut serial = SerialPort::new(&usb_bus);

    let mut usb_dev = UsbDeviceBuilder::new(&usb_bus,
UsbVidPid(0x16c0, 0x27dd))
        .strings(&[StringDescriptors::default()
            .manufacturer("implRust")
            .product("Ferris")
            .serial_number("12345678")])
        .unwrap()
        .device_class(2) // 2 for the CDC, from:
https://www.usb.org/defined-class-codes
        .build();

    let mut buff: String<64> = String::new();
```

Pin setup

Let's set up the ADC and configure GPIO 27 and GPIO 26, which are mapped to the VRX and VRY pins of the joystick:

```
let mut adc = hal::Adc::new(pac.ADC, &mut pac.RESETS);

//VRX Pin
let mut adc_pin_1 = hal::adc::AdcPin::new(pins.gpio27).unwrap();
// VRY pin
let mut adc_pin_0 = hal::adc::AdcPin::new(pins.gpio26).unwrap();
```

We also configure GPIO15 as a pull-up input for the button:

```
let mut btn = pins.gpio15.into_pull_up_input();
```

Printing Co-ordinates

We want to print the coordinates only when the vrx or vry values change beyond a certain threshold. This avoids continuously printing unnecessary values.

To achieve this, we initialize variables to store the previous values and a flag to determine when to print:

```
let mut prev_vrx: u16 = 0;
let mut prev_vry: u16 = 0;
let mut print_vals = true;
```

Reading ADC Values:

First, read the ADC values for vrx and vry. If there's an error during the read operation, we ignore it and continue the loop:

```
let Ok(vry): Result<u16, _> = adc.read(&mut adc_pin_0) else {
    continue;
};
let Ok(vrx): Result<u16, _> = adc.read(&mut adc_pin_1) else {
    continue;
};
```

Checking for Threshold Changes:

Next, we check if the absolute difference between the current and previous values of vrx or vry exceeds a threshold (e.g., 100). If so, we update the previous values and set the print_vals flag to true:

```
if vrx.abs_diff(prev_vrx) > 100 {
    prev_vrx = vrx;
    print_vals = true;
}

if vry.abs_diff(prev_vry) > 100 {
    prev_vry = vry;
    print_vals = true;
}
```

Using a threshold filters out small ADC fluctuations, avoids unnecessary prints, and ensures updates only for significant changes.

Printing the Coordinates

If print_vals is true, we reset it to false and print the X and Y coordinates via the USB serial:

```
if print_vals {
    print_vals = false;

    buff.clear();
    write!(buff, "X: {} Y: {}\r\n", vrx, vry).unwrap();
    let _ = serial.write(buff.as_bytes());
}
```

Button Press Detection with State Transition

The button is normally in a high state. When you press the knob button, it switches from high to low. However, since the program runs in a loop, simply checking if the button is low could lead to multiple detections of the press. To avoid this, we only register the press once by detecting a high-to-low transition, which indicates that the button has been pressed.

To achieve this, we track the previous state of the button and compare it with the current state before printing the "button pressed" message. If the button is currently in a low state (pressed) and the previous state was high (not pressed), we recognize it as a new press and print the message. Then, we update the previous state to the current state, ensuring the correct detection of future transitions.

```
let btn_state = btn.is_low().unwrap();
if btn_state && !prev_btn_state {
    let _ = serial.write("Button Pressed\r\n".as_bytes());
    print_vals = true;
}
prev_btn_state = btn_state;
```

The Full code

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

use core::fmt::Write;
use embedded_hal::{delay::DelayNs, digital::InputPin};
use embedded_hal_0_2::adc::OneShot;
use hal::block::ImageDef;
use heapless::String;
use panic_halt as _;
use rp235x_hal as hal;

use usb_device::{class_prelude::*, prelude::*};
use usbd_serial::SerialPort;

#[link_section = ".start_block"]
#[used]
pub static IMAGE_DEF: ImageDef = hal::block::ImageDef::secure_exe();

const XTAL_FREQ_HZ: u32 = 12_000_000u32;

#[hal::entry]
fn main() -> ! {
    let mut pac = hal::pac::Peripherals::take().unwrap();
    let mut watchdog = hal::Watchdog::new(pac.WATCHDOG);

    let clocks = hal::clocks::init_clocks_and_plls(
        XTAL_FREQ_HZ,
        pac.XOSC,
        pac.CLOCKS,
        pac.PLL_SYS,
        pac.PLL_USB,
        &mut pac.RESETS,
        &mut watchdog,
    )
    .ok()
    .unwrap();
    let mut timer = hal::Timer::new_timer0(pac.TIMER0, &mut
pac.RESETS, &clocks);

    let sio = hal::Sio::new(pac.SIO);
    let pins = hal::gpio::Pins::new(
        pac.IO_BANK0,
        pac.PADS_BANK0,
        sio.gpio_bank0,
        &mut pac.RESETS,
    );
    // let mut led = pins.gpio25.into_push_pull_output();

    let usb_bus = UsbBusAllocator::new(hal::usb::UsbBus::new(
```

```
        pac.USB,
        pac.USB_DPRAM,
        clocks.usb_clock,
        true,
        &mut pac.RESETS,
    ));

    let mut serial = SerialPort::new(&usb_bus);

    let mut usb_dev = UsbDeviceBuilder::new(&usb_bus,
UsbVidPid(0x16c0, 0x27dd))
        .strings(&[StringDescriptors::default()
            .manufacturer("implRust")
            .product("Ferris")
            .serial_number("12345678")])
        .unwrap()
        .device_class(2) // 2 for the CDC, from:
https://www.usb.org/defined-class-codes
        .build();

    let mut btn = pins.gpio15.into_pull_up_input();

    let mut adc = hal::Adc::new(pac.ADC, &mut pac.RESETS);

    //VRX Pin
    let mut adc_pin_1 = hal::adc::AdcPin::new(pins.gpio27).unwrap();
    // VRY pin
    let mut adc_pin_0 = hal::adc::AdcPin::new(pins.gpio26).unwrap();

    let mut prev_vrx: u16 = 0;
    let mut prev_vry: u16 = 0;
    let mut prev_btn_state = false;
    let mut buff: String<64> = String::new();
    let mut print_vals = true;
    loop {
        let _ = usb_dev.poll(&mut [&mut serial]);

        let Ok(vry): Result<u16, _> = adc.read(&mut adc_pin_0) else {
            continue;
        };
        let Ok(vrx): Result<u16, _> = adc.read(&mut adc_pin_1) else {
            continue;
        };

        if vrx.abs_diff(prev_vrx) > 100 {
            prev_vrx = vrx;
            print_vals = true;
        }

        if vry.abs_diff(prev_vry) > 100 {
            prev_vry = vry;
            print_vals = true;
```

```
        }

        let btn_state = btn.is_low().unwrap();
        if btn_state && !prev_btn_state {
            let _ = serial.write("Button Pressed\r\n".as_bytes());
            print_vals = true;
        }
        prev_btn_state = btn_state;

        if print_vals {
            print_vals = false;

            buff.clear();
            write!(buff, "X: {} Y: {}\r\n", vrx, vry).unwrap();
            let _ = serial.write(buff.as_bytes());
        }

        timer.delay_ms(50);
    }
}

#[link_section = ".bi_entries"]
#[used]
pub static PICOTOOL_ENTRIES: [hal::binary_info::EntryAddr; 5] = [
    hal::binary_info::rp_cargo_bin_name!(),
    hal::binary_info::rp_cargo_version!(),
    hal::binary_info::rp_program_description!(c"JoyStick USB"),
    hal::binary_info::rp_cargo_homepage_url!(),
    hal::binary_info::rp_program_build_attribute!(),
];
```

Clone the existing project

You can clone (or refer) project I created and navigate to the `joystick-usb` folder.

```
git clone https://github.com/ImplFerris/pico2-rp-projects
cd pico2-projects/joystick-usb/
```

How to Run ?

The method to flash (run the code) on the Pico is the same as usual. However, we need to set up tio to interact with the Pico through the serial port (/dev/ttyACM0). This allows us to read data from the Pico or send data to it.

tio

Make sure you have tio installed on your system. If not, you can install it using:

```
apt install tio
```

Connecting to the Serial Port

Run the following command to connect to the Pico's serial port:

```
tio /dev/ttyACM0
```

This will open a terminal session for communicating with the Pico.

Flashing and Running the Code

Open another terminal, navigate to the project folder, and flash the code onto the Pico as usual:

```
cargo run
```

If everything is set up correctly, you should see a "Connected" message in the tio terminal. As you move the joystick, the coordinates will be printed. Pressing the knob downwards will also display a "Button pressed" message.

```
~/i/g/p/p/joystick-usb  main  cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running `picotool load -u -v -x -t elf target/thumbv8m.main-none-eabihf/debug/joys
tick`
Family id 'rp2350-arm-s' can be downloaded in absolute space:
  00000000→02000000
Loading into Flash: [==============================]  100%
Verifying Flash:    [==============================]  100%
  OK

The device was rebooted to start the application.
~/i/g/p/p/joystick-usb  main
```

```
[tio 17:03:51] Press ctrl-t q to quit
[tio 17:04:11] Connected
X: 4 Y: 1828
X: 791 Y: 1764
X: 1883 Y: 1786
Button Pressed
X: 1888 Y: 1842
Button Pressed
X: 1866 Y: 1798
X: 2122 Y: 1790
X: 3706 Y: 1822
X: 4095 Y: 1869
X: 1869 Y: 1850
X: 1657 Y: 1773
X: 1278 Y: 1827
Button Pressed
X: 1858 Y: 1801
```

Curated List of Projects Written in Rust for Raspberry Pi Pico 2

Here is a curated list of projects I found online that are interesting and related to Pico 2 and Rust. If you have some interesting projects to showcase, please send a PR :)

- Pico Rex: Dinosaur Game written in Rust for the Raspberry Pi Pico 2 (RP2350) with an OLED display, using the Embassy framework.
- GB-RP2350 A Game Boy emulator for the Pi Pico 2 written in Rust: You can also find the reddit post by the OP here.
- simple-robot: A very simple robot with HC-SR04 distance sensor and autonomous as well as remote controlled movement written in Rust(Embassy)

Useful resources

This section will include a list of resources I find helpful along the way.

Blog Posts

- RP2350 Launch Blog by theJPster

Tutorials

- Official Raspberry Pi Pico guide

Other resources

- Curated list of resources for Embedded Rust
- Writing an OS in Rust : many useful concepts explained here
- Embassy Book