Embedded development with Rust and Raspberry Pi Pico

— Obtaining posture information with an IMU! —

[Author] Shuntaro OHNO

Apr. 30, 2025 ver 1.0

■ **Disclaimer**

This book is intended solely for informational purposes.
Whatever happens as a result of executing, applying, or operating based on the contents of this book is the sole responsibility of the individual who undertakes such actions. We, the authors and all related parties, accept no responsibility whatsoever.

■ **Trademarks**

The system names and product names mentioned in this book are trademarks or registered trademarks of their respective companies.
In addition, marks such as ™, ®, and © have been omitted throughout this book.

Introduction

Preface

The Rust programming language, known for its robustness, speed, and concurrency, is currently being used worldwide for the development of programs requiring high performance. Additionally, we believe that beyond these three strengths, Rust's exceptional capability for managing code assets will continue to make it highly valued globally. Specifically, Rust libraries (called "crates") are centrally managed on crates.io, with version information rigorously preserved. In traditional programming environments, it was common for valuable libraries to become lost over time, making it difficult to leave code as a lasting legacy for humanity. However, with the advent of Rust, this problem has been solved, and it is now possible to ensure that code written today can continue to be utilized far into the future. This represents a major shift in modern software development environments.

Meanwhile, there has also been an exciting advancement in the hardware development environment: the release of the Raspberry Pi Pico board, featuring the RP2040 chip designed by the Raspberry Pi Foundation in the United Kingdom. Despite costing only a few dollers, this small microcontroller board is equipped not only with fundamental functions such as GPIO (digital input/output), ADC (analog input/output), SPI, I2C, and UART, but also with Programmable Input/Output (PIO), a mechanism that allows the design of custom input/output functionalities. Its low cost and compact size mean that even individual developers can perform a wide range of designs and experiments, ultimately enabling the creation of sophisticated mechanical systems.

The Rust community has also stepped up to support the Raspberry Pi Pico by developing crates such as `rp2040_hal`. We have named the combination of Rust and Raspberry Pi Pico development "Raspico" (Rust x Pico) and recommend it both as an introduction to Rust programming and to electronics projects. The field of embedded development using Rust—and the "Raspico" movement—is still in its infancy. However, backed by Rust' s powerful code asset management infrastructure, it is poised to steadily build a rich history over time. We hope this book can serve as a starting point for your journey into the world of "Raspico."

About J-IMPACT

At J-IMPACT, we are a collective of engineers dedicated to providing practical hardware and software technologies in anticipation of the coming era where individuals will create industries.

We are building and disseminating a real technological foundation that enables individuals

to understand, implement, and apply advanced technologies—domains that were once the exclusive privilege of large corporations.

Our vision is a world where everyone has access to the technical capabilities needed to oversee the entire process, from blueprints to real-world application. From the front lines of implementation, J-IMPACT is committed to nurturing and supporting individuals who will drive the industries of the future.

Objectives and Scope of This Book

As the title suggests, the goal of this book is to create a circuit using a Raspberry Pi Pico and an IMU (Inertial Measurement Unit), write software in Rust, and acquire and display posture information. Accordingly, unless necessary for the flow of explanation, we will not cover the following topics in detail:

- Basics of programming in Rust
- General overview of embedded programming with Rust
- How to use Git

In addition, we will use the BNO055 module as the IMU. Since this module includes built-in posture estimation functionality, we will also not go into detail on the following topics:

- Posture estimation calculations using sensor fusion of accelerometers and gyroscopes
- (Extended) Kalman filters

For readers interested in these subjects, we recommend referring to the H listed at the end of the book. Therefore, the objectives of this book are summarized as the following three points:

1. Operate the Raspberry Pi Pico using Rust
2. Control the Raspberry Pi Pico and display information using Rust
3. Configure the IMU appropriately with Rust and the Raspberry Pi Pico, and acquire and display posture information

Each of these objectives corresponds to the contents of individual chapters.

Notes

- The company names, product names, and other names mentioned in this book are generally trademarks or registered trademarks of their respective companies. Trademark symbols have been omitted throughout this book for readability.
- We, the authors and publisher, are not liable for any disadvantages or damages incurred by users as a result of using the information provided in this book.

Support Page

All sample code used in this book is available in the author's GitHub repository:

[External Link] `rp_bno055` Sample Code Repository
`https://github.com/doraneko94/rp_bno055`

Please refer to it as needed.

Validation Environment

The author has verified the operation of the code described in this book under the following environment:

- OS: Windows 11 Home
- Rust: rustc 1.65.0

Table of Contents

Chapter 1

Setting Up the Environment and Obtaining Parts

1.1 Installing Rust

First and foremost, we need Rust to get started. Please refer to the following official website to install Rust:

[External Link] Install Rust https://www.rust-lang.org/ja/tools/install

1.2 Updating Rust and Adding a Target

To avoid known bugs, we update Rust to the latest stable version. Additionally, to enable development for the Raspberry Pi Pico, we add the target `thumbv6m-none-eabi`. Open a command prompt and run the following commands:

▼ List 1.1: Update and Add Target: shell

```
C:\hogehoge>rustup self update
C:\hogehoge>rustup update stable
C:\hogehoge>rustup target add thumbv6m-none-eabi
```

1.3 Installing the Build Tool

To convert Rust programs into the UF2 format required for writing to the Raspberry Pi Pico, we install the tool elf2uf2-rs. Open a command prompt and run the following command:

▼ List 1.2: Install elf2uf2-rs: shell

```
C:\hogehoge>cargo install elf2uf2-rs --locked
```

If this installation fails, please refer to the appendix Appendix A "Installation Failed → Try Downgrading Rust.".

With this, the software setup is complete.

1.4 Shopping List for Electronic Parts

Next, we move on to preparing the hardware. The most complex circuit covered in this book is shown in Figure 4.1, so we will gather electronic parts sufficient to build it. We assume that you already have general tools necessary for basic electronics work, such as soldering equipment.

▼ Table 1.1: Parts List Used in This Book

Item	Quantity	Approximate Price
Raspberry Pi Pico	1	$7
9-axis Sensor Fusion Module Kit with BNO055	1	$24
Carbon Resister 3.3k Ω	2	$1
Jumper Wires	6	$2
Breadboard	1	$4
Narrow Pin Header 1 × 20	2	$1
USB Cable (micro B)	1	$2
Multilayer Ceramic Capacitor 0.1uF (optional)	1	$1

Additionally, because we will frequently connect and disconnect the USB cable during the project, it is convenient to use a USB connector with an ON-OFF switch, as shown in Figure 1.1.

▲ Figure 1.1: USB connector with an ON-OFF switch

Chapter 2

Getting Familiar with Rust x Raspberry Pi Pico

2.1 Development Template for Rust x Raspberry Pi Pico

Embedded development involves a large number of settings and required crates, and handling all of these preparations can be quite challenging. To make this easier, we have prepared a development template called `rp_pico_template` available on the author's GitHub (originally created for personal use). Please download it using `git clone` or another method from the following URL:

[External Link] rp_pico_template Repository
https://github.com/doraneko94/rp_pico_template

By renaming the project name in this directory, you can quickly start developing new projects. Here, we will change the project name to `rp_bno055` following these two steps:

1. Rename the directory to `rp_bno055`
2. Open `Cargo.toml` and change the value of `name` under `[package]` to `"rp_bno055"`

▼ List 2.1: Changing the Package Name: Cargo.toml

```
[package]
name = "rp_bno055" # <- Originally, this is set to "rp_pico_template"
version = "0.1.0"
...
```

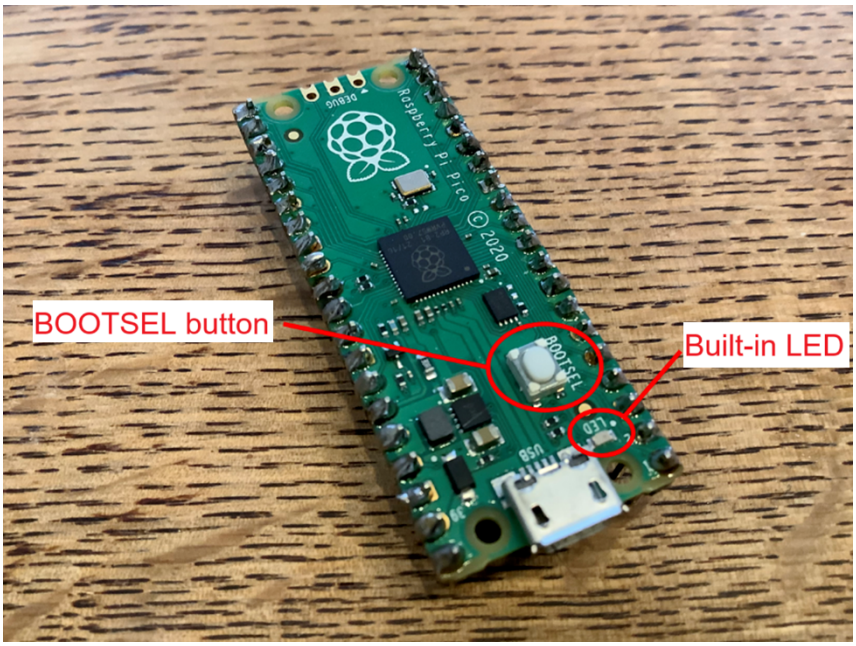
With this, the setup is complete.

2.2 Blinking the Built-in LED on the Raspberry Pi Pico

When encountering a new programming language, we typically start by writing a program that outputs "Hello, world!". In embedded development, the equivalent first step is "blinking an LED". Since the Raspberry Pi Pico board comes with a built-in LED, there is no need to assemble any circuits. Let's dive right in.

...That said, in fact, the `src` directory of the `rp_pico_template` already includes a file named `led.rs` that performs an LED blink. Thus, there is no need to write any new code; you simply need to run it to complete the introductory exercise. Please follow the steps below:

1. Open a command prompt and navigate to the project folder (`rp_bno055`)
2. While holding down the BOOTSEL button (see Figure 2.1), connect the Raspberry Pi Pico to your PC's USB port (make sure that the Raspberry Pi Pico is properly recognized)
3. In the command prompt, execute the following command: `cargo run --release --bin led`



▲ Figure 2.1: Built-in LED and BOOTSEL Button on the Raspberry Pi Pico

▼ List 2.2: Running led.rs: shell

```
C:\hogehoge>cd rp_bno055
C:\hogehoge\rp_bno055>cargo run --release --bin led
```

```
Compiling ···
···
Compiling ···
    Finished release [optimized] target(s) in 25.27s
     Running `elf2uf2-rs -d -s target\thumbv6m-none-eabi\release\led`
Found pico uf2 disk G:\
Transfering program to pico
9.00 KB / 9.00 KB [=========================] 100.00 % 1.86 MB/s
```

If everything runs successfully, a message like that shown in List 2.2 will appear, and the LED on the Raspberry Pi Pico will blink at approximately one-second intervals.

A common mistake is failing to correctly press the BOOTSEL button in Step 2. You must hold down the BOOTSEL button **while** connecting the device (pressing it **after** connecting will **not** work). Holding the button during connection erases the previously written program on the Raspberry Pi Pico and enables it to enter write mode. If you fail at this step and the Raspberry Pi Pico is not properly recognized, an error message like the following will be displayed:

▼ List 2.3: If the BOOTSEL Button Is Not Pressed Correctly: shell

```
C:\hogehoge\rp_bno055>cargo run --release --bin led
Compiling ···
···
Compiling ···
    Finished release [optimized] target(s) in 25.27s
     Running `elf2uf2-rs -d -s target\thumbv6m-none-eabi\release\led`
Error: “Unable to find mounted pico”
error: process didn’t exit successfully: `elf2uf2-rs -d -s target\thumbv›
›6m-none-eabi\release\led` (exit code: 1)
```

Note that if you connect the Raspberry Pi Pico to your PC **without** pressing the BOOTSEL button, or if you supply power through the VSYS pin, the latest program previously written to the Raspberry Pi Pico will run automatically.

2.3 Code Explanation for LED Blink

Ending the introductory chapter without writing a single line of code would feel a bit unsatisfying, so let’s walk through the contents of `led.rs` briefly. In embedded development using Rust, the code structure can generally be divided into two major sections:

- the portion **before** the `main` function, and
- the body **inside** the `main` function, which itself can be further divided into:

- the setup **before** the `loop`, and
- the repetitive operations **inside** the `loop`.

♣ Before the `main` Function

▼ List 2.4: Code Before the main Function: src/led.rs

```
#![no_std] // ①
#![no_main]
…
use embedded_hal::digital::v2::OutputPin; // ②
use rp2040_hal::clocks::Clock;
…
const XTAL_FREQ_HZ: u32 = 12_000_000u32; // ③
```

Before the `main` function, we describe three types of elements: ① **Environmental attributes**, ② **Used crates**, and ③ **Constants**.

As an example of ①, the notation `#![no_std]` declares that the `std` crate will not be used. This allows the code to run even in environments without an operating system (bare-metal environments). However, it also brings disadvantages: convenient features implemented in the `std` crate, such as the `Vec` type and the `println!` macro, become unavailable. (For this reason, we will consider alternative ways to implement similar functionality later.)

As an example of ②, we use `embedded_hal::digital::v2::OutputPin`. This is a trait necessary for controlling a pin as a current source to light up the onboard LED.

♣ Inside the `main` Function (Before the `loop`)

▼ List 2.5: Code Inside the main Function, Before the loop: src/led.rs

```
let mut pac = pac::Peripherals::take().unwrap(); // ①
let core = pac::CorePeripherals::take().unwrap();

let mut watchdog = hal::Watchdog::new(pac.WATCHDOG); // ②
let clocks = hal::clocks::init_clocks_and_plls(
XTAL_FREQ_HZ
    …
    &mut watchdog,
)
.ok()
.unwrap();
let mut delay = cortex_m::delay::Delay::new(core.SYST, clocks.system_cloc›
›k.freq().to_Hz());

let sio = hal::Sio::new(pac.SIO); // ③
```

```
let pins = hal::gpio::Pins::new(
…
    sio.gpio_bank0,
    …
);
let mut led_pin = pins.gpio25.into_push_pull_output();
```

Here, we declare the peripherals on the Raspberry Pi Pico that we will use and configure their modes. `Peripherals` refer to components like timers and pins that are available on the microcontroller board.

What we want to achieve in this code is:

- Making the pin that controls the onboard LED available
- Enabling the `Delay` functionality (such as "wait 500 milliseconds")

Specifically,

- ② block configures the `Delay`
- ③ block configures the onboard LED.

To support these operations, we declare the necessary peripherals ahead of time through a domino-style dependency. The dependency relationships are as follows:

- ②: `Delay` ← `clocks` ← `Watchdog`
- ③: `led_pin` ← `Pins` ← `Sio`

At the top of these dependency chains are `pac` and `core`. Therefore, in almost all cases, ① must be included.

Regarding ③: We acquire the pin for blinking the onboard LED as a variable named `led_pin`, and set it as a `mut` variable so that we can change its state (turning it on and off). The pin for the onboard LED is labeled `GPIO25` on the Raspberry Pi Pico board. Thus, we retrieve `gpio25` from the variable `pins`, which holds all pin information, and set it to output mode using the `into_push_pull_output` method.

♣ Inside the `main` Function (Inside the `loop`)

▼ List 2.6: Code Inside the main Function, Inside the loop: src/led.rs

```
loop {
    led_pin.set_high().unwrap();
    // TODO: Replace with proper 1s delays once we have clocks working
    delay.delay_ms(500);
    led_pin.set_low().unwrap();
    delay.delay_ms(500);
}
```

As you can see from this code, the `loop` continuously repeats forever. This is because embedded programs are designed not to terminate after completing a task, but to continue running until the power is turned off. For this reason, the function signature is written as `fn main() -> !`, where the `!` indicates that the `main` function does not return (i.e., it never terminates or returns an exit code).

In `src/led.rs`, the program performs the following sequence:

1. Set the output pin voltage to High and turn on the onboard LED (`set_high`)
2. Wait for 500 milliseconds (`delay_ms(500)`)
3. Set the output pin voltage to Low and turn off the onboard LED (`set_low`)
4. Wait for 500 milliseconds (`delay_ms(500)`)

By repeating this sequence, the onboard LED blinks with a cycle of approximately one second. If you want to blink the LED at a more precise interval, see Appendix B "Blinking the LED at an Accurate 1-Second Interval".

Chapter 3
Serial Communication

3.1 Hello, world! via Serial Communication

With the previous chapter, we completed the basics of embedded programming. However, for those who still wish to experience a "Hello, world!" moment, we have also prepared an introductory example that displays text. We will perform serial communication between the Raspberry Pi Pico and a PC connected via a USB cable, displaying the text "Hello, world!" on the PC screen. Please refer to the sample code at the following link:

[External Link] Sample Code for `src/serial_hello.rs`
https://github.com/doraneko94/rp_bno055/blob/master/src/serial_hello.rs

♣ Creating a New File

To run the "Hello, world!" example via serial communication, create a new file named `serial_hello.rs` in the `src` directory. Additionally, add a new `[[bin]]` block at the end of `Cargo.toml` to register this file as a binary (executable). This allows you to execute it using the `cargo run` command.

▼ List 3.1: Adding a bin Block: Cargo.toml

```
[[bin]] # End of File
name = "serial_hello"
path = "src/serial_hello.rs"
```

Please write the contents of `src/serial_hello.rs` while referring to the sample code. Below, we will explain some of the key features of the sample.

♣ Introducing Crates

Normally, before writing `src/serial_hello.rs`, you would need to add crates required for serial communication to `Cargo.toml`. (However, since these crates are already included in `rp_pico_template`, you can skim this section for reference.)

The crates to be added are:

- `usb_device`: A crate that implements `UsbDevice` instances, `UsbClass`, and `UsbBus`, allowing you to form composite USB devices.
- `usb_hid`: An implementation of the USB HID (Human Interface Device) class.
- `usb_serial`: An implementation of CDC-ACM (Communication Device Class Abstract Control Model) USB serial ports.

In the sample code, we declare the use of structures and traits from `usb_device` and `usb_serial`.

▼ List 3.2: Declaring the Crates to Use: src/serial_hello.rs

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

♣ Defining the Required Structures

For communication over USB, we need the following three components:

- USB Bus
- Serial Port
- USB Device

We will implement each of these step by step.

▼ List 3.3: Building the UsbBus: src/serial_hello.rs

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

We build the `UsbBus` from `rp2040_hal` using the `UsbBusAllocator` provided by `usb_device`.

▼ List 3.4: Building the SerialPort: src/serial_hello.rs

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

Using the `UsbBus` that we just constructed, we build the `SerialPort`.

▼ List 3.5: Building the UsbDevice: src/serial_hello.rs

```
let mut usb_dev = UsbDeviceBuilder::new(&usb_bus, UsbVidPid(0x16c0, 0x27d
›d))
    .manufacturer("Fake company")
    .product("Serial port")
    .serial_number("TEST")
    .device_class(2)
    .build();
```

Similarly, we use the `UsbBus` to construct the `UsbDevice`. At this point, you can define attributes such as the manufacturer name (`manufacturer`), product name (`product`), and serial number (`serial_number`). In this sample, dummy names are assigned to each. The device class (`device_class`) is a number defined by USB.org; here, we specify `0x02` (Communications and CDC Control). For details on class codes, refer to the following page:

[External Link] USB.org | Defined Class Codes
https://www.usb.org/defined-class-codes

♣ Outputting the Text

After defining the necessary structures, we write the main process inside the `loop`.

▼ List 3.6: Process Inside the loop: src/serial_hello.rs

```
loop {
    for _ in 0..200 {
        delay.delay_ms(5);
        let _ = usb_dev.poll(&mut [&mut serial]); // ①
    }
    let _ = serial.write(b"Hello, world!\r\n"); // ②
}
```

In this code, "Hello, world!" is output at a one-second interval.

② handles writing "Hello, world!" to the serial port. Here, be sure to include the line break code (`\r\n`) and prefix the string with `b` to treat it as a byte array when passing it to the `write` method. This prepares the `SerialPort` to write the text "Hello, world!\r\n".

① polls the given device feature to check if any new events have occurred and processes any available actions. In this case, it checks for updates to the `SerialPort`, and if any written data exists, it outputs the data. The function returns `true` if there is an update and `false` otherwise, though the return value is not used here.

Important: This polling function must be called at least once every 10 milliseconds to maintain stable serial communication. If it is not called for longer than that, the serial connection may be disrupted.

Therefore, it is not possible to perform a simple wait of 1000 milliseconds using `delay.delay_ms(1000)`. Instead, we achieve a 1-second wait by repeating a 5-millisecond wait followed by `usb_dev.poll()` 200 times (5 ms × 200 = 1000 ms).

♣ Running the Code

Finally, let's run the completed program. You should see the following output displayed on the screen at approximately one-second intervals:

▼ List 3.7: Output of "Hello, world!": shell

```
C:\hogehoge\rp_bno055>cargo run --release --bin serial_hello
...
Found pico serial on COM6
Hello, world!
Hello, world!
...
```

3.2 Outputting Numbers

(To summarize this section briefly: "Outputting numbers via serial communication is quite tedious. Therefore, we created a convenient crate." In this book, we will use that crate from now on for number output. If you are not interested in the technical background, you may skip this section and only perform the crate addition at the end.)

♣ Outputting Integers Using the `numtoa` Crate

In the previous section, we output the string "Hello, world!". Moving forward in this book, serial communication will play an important role in verifying values read from the BNO055 sensor. Thus, we need to output not just text but also numeric values. In `no_std` environments, we use the `numtoa` crate to convert numbers into byte arrays for passing to the `serial.write` method.

Because `numtoa` is not included by default in `rp_pico_template`, if you plan to use later sample codes, please open `Cargo.toml` and add the following entry under `[dependencies]`:

▼ List 3.8: Adding the numtoa Crate: Cargo.toml

```
[dependencies]
cortex-m = "0.7.4"
…
panic-halt = "0.2.0"
numtoa = "0.2.4" # Add
```

You can refer to the sample code here:

[External Link] Sample Code for `src/serial_hello_numtoa.rs`
https://github.com/doraneko94/rp_bno055/blob/master/src/serial_hello_numtoa.rs

In this sample, the code outputs "Hello, world!" every second along with the number of times it has been output.

The basic procedure for outputting numbers using `numtoa` is as follows:

▼ List 3.9: Number Output with numtoa: src/serial_hello_numtoa.rs

```
…
use numtoa::NumToA;

#[rp2040_hal::entry]
fn main() -> ! {
    …
    let mut count = 0;

    let mut buf = [0u8; 20]; // ①
    loop {
        for _ in 0..200 {
            delay.delay_ms(5);
            let _ = usb_dev.poll(&mut [&mut serial]);
        }
        count += 1;
        let _ = serial.write(b"Hello, world! x "); // ②
        let _ = serial.write(count.numtoa(10, &mut buf)); // ③
        let _ = serial.write(b"\r\n"); // ④
    }
}
```

- Prepare a buffer to hold the byte sequence for the number (①).
- Output the text string portion together (②).
- Use the `numtoa::NumToA` trait to attach the `numtoa` method to integer types, providing the radix (number base) and the buffer (③).

- Output a line break code (④).

Since previous uses of the `serial.write` method did not output line breaks, the text up to "Hello, world! x (count)" is displayed on the same line. (If you forget to add a line break, nothing will appear because the line will not be finalized.)

Running this code results in output like the following:

▼ List 3.10: Output of "Hello, world! x (count)": shell

```
C:\hogehoge\rp_bno055>cargo run --release --bin serial_hello_numtoa
...
Found pico serial on COM6
Hello, world! x 1
Hello, world! x 2
...
```

(When running the sample, remember to add a `[[bin]]` block to `Cargo.toml` just like in List 3.1.)

♣ Outputting Floating-Point Numbers

The `numtoa` crate cannot convert floating-point numbers (decimals). Therefore, to display a value such as 123.4567 (of type `f32`) up to two decimal places, the following manual steps are needed:

1. Convert 123.4567 to an `i32` to extract the integer part (123).
2. Output the integer part.
3. Output a period (".") as a string.
4. Convert the integer part back to `f32` (123.0) and subtract it from the original number to extract the decimal part (0.4567).
5. Multiply the decimal part by 100 (to get 45.67).
6. Convert to `i32` and output (45).

However, this manual approach can cause problems—for example, 123.0456 might be displayed incorrectly as "123.4". Adjustments like zero-padding are needed to handle such cases properly. Writing this process manually every time is tedious and reduces code readability. Thus, we created the `serial_write` crate, which allows this entire procedure to be written in a single line.

[External Link] `serial_write` Crate Page on crates.io
https://crates.io/crates/serial_write

[External Link] `serial_write` GitHub Repository

https://github.com/doraneko94/serial_write

The `serial_write` crate not only displays floating-point numbers easily but also offers features such as:

- Consistent width formatting (e.g., scientific notation like "1.23e002")
- Array formatting

You can see a sample that accomplishes the same functionality using `serial_write` here:

[External Link] Sample Code for `src/serial_hello_count.rs`
https://github.com/doraneko94/rp_bno055/blob/master/src/serial_hello_count.rs

▼ List 3.11: Number Output with serial_write: src/serial_hello_count.rs

```
use serial_write::Writer;

#[rp2040_hal::entry]
fn main() -> ! {
    ...
    let mut count = 0;

    let mut writer = Writer::new(); // ①
    loop {
        for _ in 0..200 {
            delay.delay_ms(5);
            let _ = usb_dev.poll(&mut [&mut serial]);
        }
        count += 1;
        let _ = writer.write_str("Hello, world! x ", &mut serial); // ②
        let _ = writer.writeln_i32(count, &mut serial); // ③
    }
}
```

Using `serial_write`:

- First, build a `Writer` structure (①).
- Then, use functions that match the type of data to be output.
- Functions starting with `write` output without a line break (②), while functions starting with `writeln` output with a line break (③).

From this point on, this book will use the `serial_write` crate for all serial communication outputs. Please add it to `Cargo.toml` as follows:

▼ List 3.12: Adding the serial_write Crate: Cargo.toml

```
[dependencies]
cortex-m = "0.7.4"
…
panic-halt = "0.2.0"
numtoa = "0.2.4" # If you don't want to run the sample using numtoa, you ›
›don't need this.
serial_write = "0.1.0" # Add
```

In the above examples, serial communication is performed simultaneously with the build on `elf2uf2-rs`. For details on this communication, see Appendix C "Starting Serial Communication".

Chapter 4

Reading Posture Information from the BNO055

4.1 Overview of the BNO055

Now we arrive at the main topic of this book. In this chapter, we will use the Raspberry Pi Pico to read output from an IMU (Inertial Measurement Unit). Considering usability and availability, we will use the BNO055 for this purpose (If you want to use another sensor, please see Appendix H "If You Want to Use a Sensor other than BNO055").

[External Link] BOSCH BNO055 Product Page
https://www.bosch-sensortec.com/products/smart-sensors/bno055/

The **BNO055** is a package that combines a 9-axis sensor (3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer) with orientation software. It can output raw sensor values individually, or it can automatically calculate and output the current orientation (rotation angles) by fusing the sensor outputs. The available communication interfaces are I2C and UART, and we will use I2C in this project.

The basic principle of calculating posture angles from sensor data is briefly explained in Appendix E "Principle of Posture Angle Estimation Using Sensor Information". For detailed explanations, please refer to specialized book.

Also, to have the BNO055 calculate posture angles correctly, it must be configured to the appropriate mode. Depending on the mode, the output angles may be relative or absolute. For details on the BNO055 modes, see Appendix F "Operating Modes of the BNO055" or:

[External Link] BNO055 Datasheet [6]

https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf

Important portions of the datasheet will be cited and translated in this book, but the datasheet also contains many details not covered here, and we highly recommend reading it.

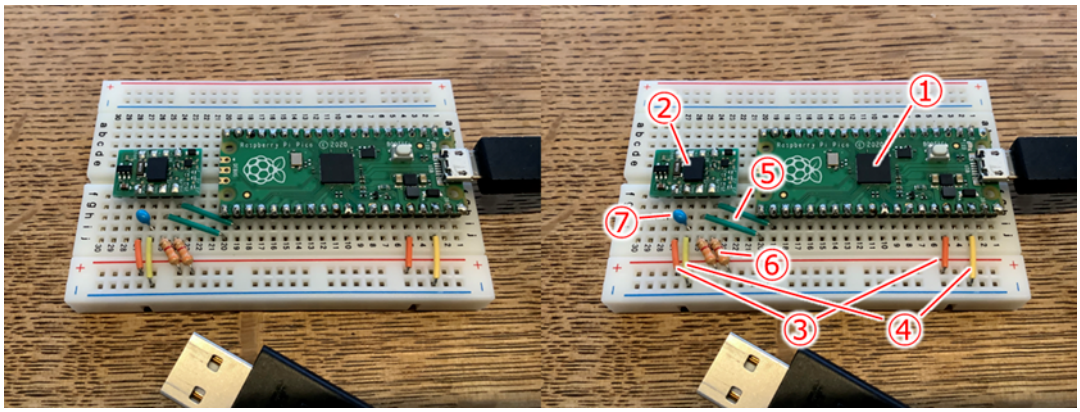
For use on a breadboard, we will use the BNO055 module kit distributed by Akizuki Denshi (Japanese shop):

[External Link] BNO055 9-Axis Sensor Fusion Module Kit
https://akizukidenshi.com/catalog/g/gK-16996/

Its user manual [7] is also a valuable reference.

4.2 Building the Circuit

Using the parts purchased in "1.4 Shopping List for Electronic Parts" we will build the circuit shown in Figure 4.1:



▲ Figure 4.1: Control Circuit for the BNO055

♣ ①: Raspberry Pi Pico

Insert into the center of the breadboard, spanning rows 1 – 20.

♣ ②: BNO055 Module

Insert into the center of the breadboard, spanning rows 24 – 27.

♣ ③: 3.3V Power Supply

Connect the Pico's `3V3(OUT)` pin (breadboard row 5, also see Figure 4.2) to the "+" power rail. Then, connect the "+" rail to the BNO055's `VIN` pin.

♣ ④: Ground (GND)

Connect the Pico's `GND` pin (breadboard row 3) to the "− " rail, and then connect the "− " rail to the BNO055's `GND` pin.

♣ ⑤: I2C Communication Wires

Based on Figure 4.2, `GPIO16` and `GPIO17` correspond to `I2C0 SDA` and `I2C0 SCL`. Connect `GPIO16` (breadboard row 20) to the BNO055's `SDA/T` pin, and `GPIO17` (row 19) to the `SCL/R` pin.

♣ ⑥: Pull-Up Resistors (3.3k Ω)

Connect pull-up resistors between the `SDA/T` and `SCL/R` pins and the "+" rail to stabilize I2C communication. According to information from Akizuki Denshi:

[External Link] BNO055 Module Q&A
https://akizukidenshi.com/catalog/faq/goodsfaq.aspx?class1=K&code=16996

a resistance of 1k Ω to 10k Ω is recommended; we use 3.3k Ω here.

♣ ⑦: Bypass Capacitor

Place a 0.1uF capacitor between `VIN` and `GND` for power supply noise reduction.

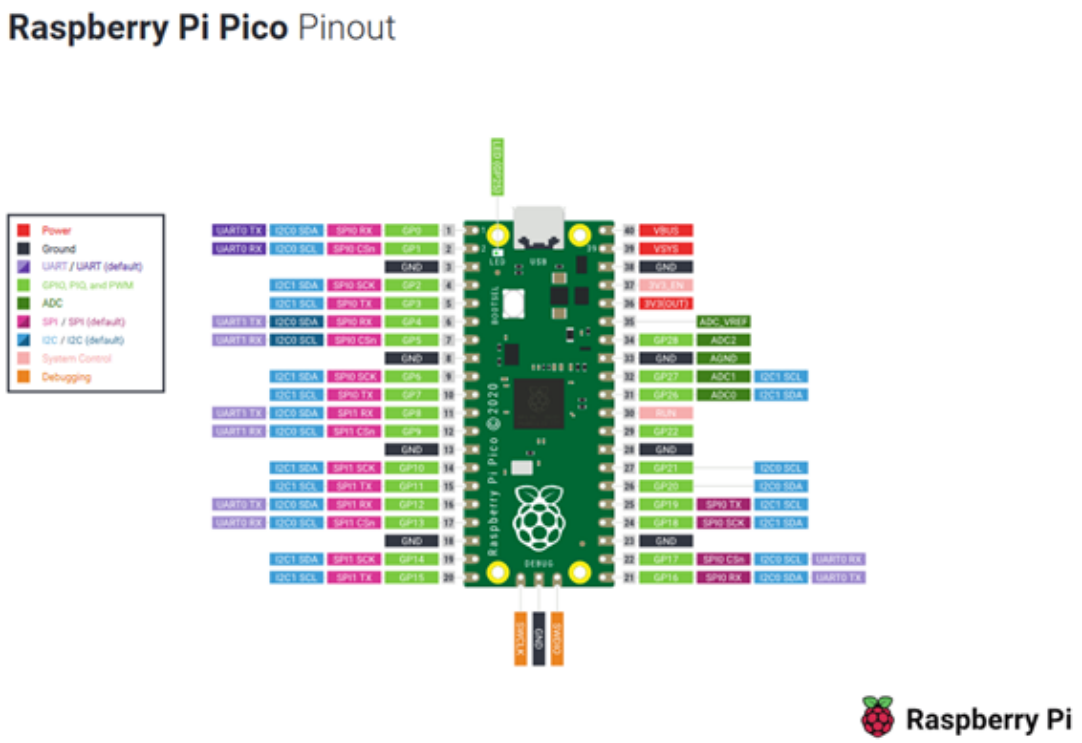
4.3 Running the Sample Code

Since multiple steps are needed to operate the BNO055, we will first present the complete code, verify its operation, and then explain it block by block.

♣ Verifying the Basic Code Operation

First, create a file named `imu_base.rs` in the `src` directory of the `rp_bno055` project, and register it as a binary in `Cargo.toml`.

▼ List 4.1: Adding the bin Block: Cargo.toml



▲ Figure 4.2: Raspberry Pi Pico Pinout (quoted from raspberrypi.com)

```
[[bin]] # End of File
name = "imu_base"
path = "src/imu_base.rs"
```

Next, add the following crates to the end of the `[dependencies]` section in `Cargo.toml`:

▼ List 4.2: Adding dependencies: Cargo.toml

```
[dependencies]
...
fugit = "0.3.6"
bno055 = { git = "https://github.com/eupn/bno055", rev = "58970ced667dedd
›d09933954b1a928587226daf7" }
```

- fugit: A helper crate for setting I2C frequencies
- bno055: The main crate for operating the BNO055

Since the latest release of `bno055` contains many unresolved bugs, we will fetch it directly from the GitHub repository, pinning a specific commit to ensure stable operation. To avoid disruptive changes from future updates, we specify a stable version of the code. In this case, we use the `bno055` crate parameters to specify the Git repository URL and the commit ID

using the `git` and `rev` fields. This ensures that we are using the code at a known stable state [2].

Next, copy the sample code from the following link into `src/imu_base.rs`:

[External Link] Sample Code for `src/imu_base.rs`
https://github.com/doraneko94/rp_bno055/blob/master/src/imu_base.rs

Afterward, connect the Raspberry Pi Pico (with the parts assembled as described in the previous section) to your PC while holding the BOOTSEL button, then write and run the `src/imu_base.rs` program.

▼ List 4.3: Running src/imu_base.rs: shell

```
C:\hogehoge\rp_bno055>cargo run --release --bin imu_base
```

If writing succeeds, you should see the following output:

▼ List 4.4: Output of Calibration Information: shell

```
sys: 0 acc: 0 gyr: 0 mag: 0 (elasped time: 1 sec)
sys: 0 acc: 0 gyr: 3 mag: 0 (elasped time: 2 sec)
sys: 0 acc: 0 gyr: 3 mag: 0 (elasped time: 3 sec)
...
```

At this point, before reading the IMU output from the BNO055, we perform calibration of three types of sensors. According to the BNO055 datasheet [6] (and based on our experience), the calibration procedures are as follows:

♣ Calibrating the Accelerometer

- Place the breadboard in six or more stable orientations, holding each for at least two seconds.
- Ensure that the gravitational force points at least once in the positive and negative directions of each X, Y, and Z axis.

Specifically, perform the following six placements (move slowly!):

1. Place the breadboard normally (double-sided tape facing down) and hold for 2 seconds.
2. Flip it upside down (double-sided tape facing up) and hold for 2 seconds.
3. Tilt the breadboard sideways so that one long side touches the floor, and hold for 2 seconds.
4. Tilt it to the opposite side, touching the other long side to the floor, and hold for 2 seconds.
5. Tilt it vertically so that one short side touches the floor, and hold for 2 seconds.

6. Tilt it to the opposite short side, and hold for 2 seconds.

In other words, stabilize each of the six faces of the breadboard one by one for two seconds each.

♣ Calibrating the Gyroscope

- Leave the breadboard in any stable position for several seconds.

♣ Calibrating the Magnetometer

- Leave the breadboard stationary. Calibration will occur naturally.
- Make sure there are no magnetic field sources nearby.
- If calibration does not complete easily, try changing the placement location.

♣ Progress of Calibration

The calibration progress is evaluated for four items on a scale from 0 (uncalibrated) to 3 (fully calibrated). The items and corresponding sensors are:

▼ Table 4.1: Calibration Information Overview

Item (Code)	Meaning
SYS	System as a whole
ACC	Accelerometer
GYR	Gyroscope
MAG	Magnetometer

Thus, calibration is considered complete when all outputs in List 4.4 display "3". As you might expect from the complexity, the **ACC** (accelerometer) calibration will likely be the most exhausting. Please patiently try various orientations, especially ones you haven't yet attempted. (This process can take a fair amount of time.)

(Note: In "4.5 Saving Calibration Data to Flash Memory", we will introduce a method to skip this calibration step.)

♣ Displaying Posture Angles

After calibration is complete, we start reading from the BNO055. Posture angle information is displayed via serial communication as follows:

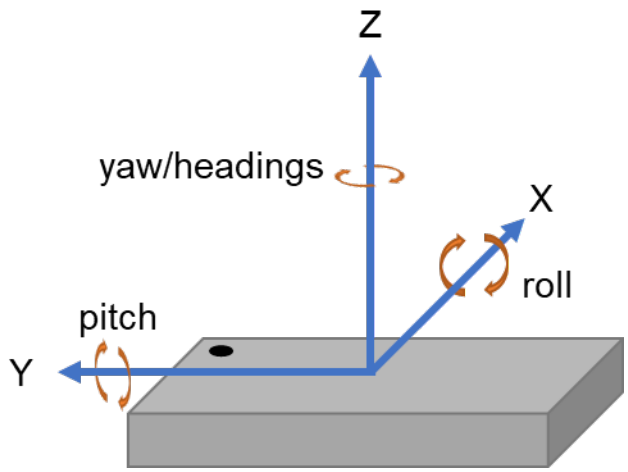
▼ List 4.5: Output of Posture Angles from src/imu_base.rs: shell

```
...
sys: 2 acc: 3 gyr: 3 mag: 3 (elasped time: 61 sec)
sys: 3 acc: 3 gyr: 3 mag: 3 (elasped time: 62 sec)
```

```
Pitch: 9.81, Roll: 38.81, Yaw: 276.75
Pitch: 10.43, Roll: 31.31, Yaw: 276.00
...
```

Here, the posture (more precisely, the posture of the BNO055) is updated and displayed every 100 milliseconds. The representation uses **Euler angles**, and the unit is degrees.

The correspondence between pitch, roll, yaw (heading) and the BNO055's XYZ axes is shown in Figure 4.3:



▲ Figure 4.3: Relationship Between Euler Angles and XYZ Axes

However, this correspondence actually differs from the BNO055's official information. The reasons for this discrepancy involve a complex intersection of factors: official specifications, the `bno055` crate, the structure used to store Euler angles, and our design intentions. We explain these details in Appendix G "Handling of Euler Angles in the BNO055"

Since the posture angles are updated in real time, changing the breadboard's orientation results in output like the following:

▼ List 4.6: Rotation Along the Roll Direction: shell

```
...
Pitch: 9.81, Roll: 38.81, Yaw: 276.75
Pitch: 10.43, Roll: 31.31, Yaw: 276.00
Pitch: 10.56, Roll: 23.81, Yaw: 273.93
Pitch: 10.56, Roll: 16.87, Yaw: 273.31
Pitch: 11.56, Roll: 9.75, Yaw: 273.37
Pitch: 12.50, Roll: 1.68, Yaw: 273.56
Pitch: 13.12, Roll: -7.12, Yaw: 273.37
Pitch: 12.18, Roll: -16.68, Yaw: 273.18
```

```
Pitch: 10.56, Roll: -25.75, Yaw: 272.93
Pitch: 6.06, Roll: -39.56, Yaw: 271.87
Pitch: 4.12, Roll: -47.68, Yaw: 272.18
Pitch: 3.00, Roll: -56.62, Yaw: 272.37
Pitch: 2.06, Roll: -69.31, Yaw: 272.31
Pitch: 2.18, Roll: -82.31, Yaw: 270.43
Pitch: 2.31, Roll: -89.31, Yaw: 269.12
...
```

Here, a rotation has been applied in the **roll** direction (the breadboard's long axis).

4.4 Implementing the Basic Code

From here, we will explain the content of `src/imu_base.rs`. First, let's highlight the key features from the overall code.

▼ List 4.7: Basic Code Using the BNO055: src/imu_base.rs

```
...
use fugit::RateExtU32; // use helper crate
...
#[rp2040_hal::entry]
fn main() -> ! {
    ...
    let sda_pin = pins.gpio16.into_mode::<hal::gpio::FunctionI2C>(); // ›
›①: begin
    let scl_pin = pins.gpio17.into_mode::<hal::gpio::FunctionI2C>();

    let i2c = hal::I2C::i2c0(
        pac.I2C0,
        sda_pin,
        scl_pin,
        400.kHz(),
        &mut pac.RESETS,
        &clocks.system_clock,
    ); // ①: end

    ...

    // Wait 1 sec after power supply to BNO055.
    for _ in 0..200 { // ②: begin
        delay.delay_ms(5);
        let _ = usb_dev.poll(&mut [&mut serial]);
```

```
    }

    let mut imu = bno055::Bno055::new(i2c).with_alternative_address();
    imu.init(&mut delay).unwrap();
    imu.set_mode(bno055::BNO055OperationMode::NDOF, &mut delay).unwrap();
// ②: end

    let mut count = 0usize; // ③: begin
    'label: loop {
        for _ in 0..200 {
            if imu.is_fully_calibrated().unwrap() {
                break 'label;
            }
            delay.delay_ms(5);
            let _ = usb_dev.poll(&mut [&mut serial]);
        }
        count += 1;
        let status = imu.get_calibration_status().unwrap();
        let _ = writer.write_str("sys: ", &mut serial);
        let _ = writer.write_u8(status.sys, &mut serial);
        …
        let _ = writer.write_str(" (elasped time: ", &mut serial);
        let _ = writer.write_usize(count, &mut serial);
        let _ = writer.writeln_str(" sec)", &mut serial);
    } // ③: end

    loop { // ④: begin
        for _ in 0..20 {
            delay.delay_ms(5);
            let _ = usb_dev.poll(&mut [&mut serial]);
        }
        let eular = imu.euler_angles().unwrap();
        let _ = writer.write_str("Pitch: ", &mut serial);
        let _ = writer.write_f32(eular.a, 2, &mut serial);
        …
        let _ = writer.write_str(", Yaw: ", &mut serial);
        let _ = writer.writeln_f32(eular.c, 2, &mut serial);
    } // ④: end
}
```

The overall processing is structured into four stages:

- ①: I2C communication setup
- ②: BNO055 initialization

- ③: Sensor calibration
- ④: Posture information output

Let's go through each stage in detail.

♣ I2C Communication Setup

▼ List 4.8: Setting Up I2C Communication: src/imu_base.rs

```
let sda_pin = pins.gpio16.into_mode::<hal::gpio::FunctionI2C>(); // ①
let scl_pin = pins.gpio17.into_mode::<hal::gpio::FunctionI2C>();

let i2c = hal::I2C::i2c0(
    pac.I2C0,
    sda_pin,
    scl_pin,
    400.kHz(), // fugit crate
    &mut pac.RESETS,
    &clocks.system_clock,
); // ②
```

I2C communication uses two lines: **SCL** (clock signal) and **SDA** (data transmission). In ①, we retrieve two GPIO pins on the Raspberry Pi Pico (GPIO16 and GPIO17) assigned for these roles (see Figure 4.2).

Pin 21 (`GPIO16`, breadboard row 20) functions as `I2C0 SDA`, and Pin 22 (`GPIO17`, breadboard row 19) functions as `I2C0 SCL`. Thus, we assign the appropriate functions to these pins.

Since these pins correspond to the Pico's I2C0 block, in ② we initialize `I2C0`. Here, to convert the integer 400 into a 400kHz frequency type, we use the `RateExtU32` trait from the `fugit` crate.

♣ Initializing the BNO055

▼ List 4.9: Initializing the Bno055 Structure: src/imu_base.rs

```
// Wait 1 sec after power supply to BNO055.
for _ in 0..200 { // ①
    delay.delay_ms(5);
    let _ = usb_dev.poll(&mut [&mut serial]);
}

let mut imu = bno055::Bno055::new(i2c).with_alternative_address(); // ②
imu.init(&mut delay).unwrap(); // ③
imu.set_mode(bno055::BNO055OperationMode::NDOF, &mut delay).unwrap(); //
```

```
④
```

Here, we initialize the `Bno055` structure to operate the sensor. Specifically, we perform write operations to the correct addresses via I2C. However, these writes can only succeed after the BNO055 has stabilized after power-up.

Thus, in ①, we maintain serial communication and wait for 5ms × 200 = 1 second.

Interestingly, even without this waiting time, writing and running via `cargo run` usually succeeds. However, if you simply connect without pressing the BOOTSEL button (power only), initialization can fail. This happens due to differences between the two scenarios:

Using BOOTSEL and cargo run

1. Power is supplied to BNO055 via USB.
2. The Pico enters write mode; no program runs.
3. Manual steps (e.g., switching screens) consume several milliseconds.
4. The program is written and then starts I2C communication.

Connecting without BOOTSEL (power only)

1. Power is supplied to BNO055 via USB.
2. The already-written program starts immediately.
3. BNO055 is not yet stable, causing access failures.

Thus, the 1-second wait in ① is essential. (The author spent about two days before noticing this issue.)

After stabilization, we create the `Bno055` structure (②). At this point, it is necessary to call the `with_alternative_address` method. Calling this method sets the I2C device address of the **Bno055** structure to `0x28`. If you do not call it, the default address remains `0x29`.

According to the module manual provided by Akizuki Denshi [7], the `COM3` pin of the BNO055 module is set to LOW, meaning the address is configured as `0x28` at shipment. Therefore, we use `with_alternative_address` here.

After that, we initialize the structure (③) and specify the operating mode (④). Here, we select the `NDOF` mode, commonly referred to as the "everything-enabled" mode.

For more about the relationship between BNO055 operating modes and their capabilities and limitations, please refer to Appendix F "Operating Modes of the BNO055".

♣ Sensor Calibration

▼ List 4.10: Sensor Calibration: src/imu_base.rs

```
let mut count = 0usize;
'label: loop { // ①
    for _ in 0..200 {
```

```
        if imu.is_fully_calibrated().unwrap() { // ②
            break 'label;
        }
        delay.delay_ms(5);
        let _ = usb_dev.poll(&mut [&mut serial]);
}
    count += 1;
    let status = imu.get_calibration_status().unwrap(); // ③
    let _ = writer.write_str("sys: ", &mut serial);
    let _ = writer.write_u8(status.sys, &mut serial);
    …
    let _ = writer.write_str(" (elasped time: ", &mut serial);
    let _ = writer.write_usize(count, &mut serial);
    let _ = writer.writeln_str(" sec)", &mut serial);
}
```

Here, we enter an infinite loop. While calibration is incomplete, we output calibration progress every second (①).

We use `is_fully_calibrated` to check if calibration is complete. If true, we break out of the loop (②).

Since `break;` alone would only exit the inner `for`-loop, we label the outer loop (`'label: loop`) and break using `break 'label;`.

If calibration isn't completed within a second, we call `get_calibration_status` and output the SYS, ACC, GYR, and MAG statuses along with elapsed time (③).

♣ Outputting Posture Information

▼ List 4.11: Outputting Posture Information: src/imu_base.rs

```
loop {
    for _ in 0..20 {
        delay.delay_ms(5);
        let _ = usb_dev.poll(&mut [&mut serial]);
    }
    let eular = imu.euler_angles().unwrap(); // ①
    let _ = writer.write_str("Pitch: ", &mut serial);
    let _ = writer.write_f32(eular.a, 2, &mut serial);
    …
    let _ = writer.write_str(", Yaw: ", &mut serial);
    let _ = writer.writeln_f32(eular.c, 2, &mut serial);
}
```

Once calibration completes, we begin acquiring and outputting posture information from

the BNO055.

Here, we call `euler_angles` to retrieve the breadboard's Euler angles. However, the BNO055 internally stores posture data as a **quaternion**. Euler angles are derived secondarily from the quaternion.

We will discuss this more deeply in "7.3 Other Known Issues with BNO055 Euler Angles".

In addition to posture, you can retrieve raw sensor outputs. The correspondence between methods and retrievable values is summarized below:

▼ Table 4.2: Methods and Their Retrieved Values

Method	Retrieved Value	Requires Fusion Mode
`quaternion`	Quaternion (posture)	Yes
`euler_angles`	Euler angles (posture)	Yes
`linear_acceleration_fixed`	Linear acceleration (cm/s^2)	Yes
`linear_acceleration`	Linear acceleration (m/s^2)	Yes
`gravity_fixed`	Gravity vector (cm/s^2)	Yes
`gravity`	Gravity vector (m/s^2)	Yes
`accel_data_fixed`	Accelerometer reading (cm/s^2)	No
`accel_data`	Accelerometer reading (m/s^2)	No
`gyro_data_fixed`	Gyroscope reading ($1/16 \times °$ /s)	No
`gyro_data`	Gyroscope reading (° /s)	No
`mag_data_fixed`	Magnetometer reading ($1/16 \times \mu$ T)	No
`mag_data`	Magnetometer reading (μ T)	No
`temperature`	Chip temperature (° C)	No

For entries marked "Yes" under "Requires Fusion Mode," the BNO055 must be operating in one of its Fusion Modes. For more about operating modes, see Appendix F "Operating Modes of the BNO055".

4.5 Saving Calibration Data to Flash Memory

In the previous section, we wrote basic code to control the BNO055, but spending time on sensor calibration every time can be cumbersome. Thus, in this chapter, we consider saving the calibration results for reuse upon future startups.

♣ Hardcoding Calibration Information

Sensor calibration information is managed using the `bno055::BNO055Calibration` structure. This structure consists of 22 fields. Thus, after performing calibration once, you can retrieve these values and hardcode them into your program for reuse.

In the sample code `src/imu_const.rs`, we define the calibration result as a constant `CALIB`

and skip calibration by loading it:

[External Link] Sample Code for `src/imu_const.rs`
https://github.com/doraneko94/rp_bno055/blob/master/src/imu_const.rs

▼ List 4.12: Loading BNO055Calibration: src/imu_const.rs

```
...
const CALIB: bno055::BNO055Calibration = bno055::BNO055Calibration { // ①
    acc_offset_x_lsb: 237,
    acc_offset_x_msb: 255,
    acc_offset_y_lsb: 212,
    acc_offset_y_msb: 255,
    acc_offset_z_lsb: 227,
    acc_offset_z_msb: 255,
    mag_offset_x_lsb: 72,
    mag_offset_x_msb: 0,
    mag_offset_y_lsb: 252,
    mag_offset_y_msb: 255,
    mag_offset_z_lsb: 156,
    mag_offset_z_msb: 253,
    gyr_offset_x_lsb: 255,
    gyr_offset_x_msb: 255,
    gyr_offset_y_lsb: 255,
    gyr_offset_y_msb: 255,
    gyr_offset_z_lsb: 1,
    gyr_offset_z_msb: 0,
    acc_radius_lsb: 232,
    acc_radius_msb: 3,
    mag_radius_lsb: 12,
    mag_radius_msb: 2
};
...
#[rp2040_hal::entry]
fn main() -> ! {
    ...
    imu.set_calibration_profile(CALIB, &mut delay).unwrap(); // ②
    ...
}
```

① holds a previously calibrated result.

By using the `set_calibration_profile` method, we can reuse this result (②).

To obtain the values in ① by yourself, you can write a program that, after calibration, calls the `calibration_profile` method and prints each field via serial communication.

♣ Saving Calibration Information to Raspberry Pi Pico

While the above method works, it requires:

- Performing calibration
- Retrieving the calibration profile
- Manually coding the values into the program

This is tedious. Thus, we propose saving calibration information directly to the Raspberry Pi Pico itself:

1. Perform calibration normally on the first run
2. Save the calibration results to Pico's flash memory
3. Power off the device
4. Power on again later
5. Load the saved calibration information

♣ Memory Layout of the RP2040

The RP2040 chip (inside the Raspberry Pi Pico) includes:

- 264kB of SRAM for main RAM (totaling 284kB with other areas)

* External flash memory for program storage

Using the XIP (eXecute In Place) mechanism, the external flash is mapped starting from the address `0x1000_0000`.

The external flash memory has a capacity of 16MB (16,777,216 bytes). It is divided into 32 blocks, and each block is further divided into 16 sectors. Flash memory is erased on a sector basis and written in 256-byte page units. Since memory operations in Rust involve handling 1-byte information via `u8` types, we express the structure in bytes and hexadecimal Table 4.3.

▼ Table 4.3: Flash Memory Capacities in Bytes and Hexadecimal

Item	Byte Size	Hexadecimal Size
Mapping Start Address	-	0x1000_0000
Flash Memory Capacity (End)	16,777,216 ÷ 8 = 2,097,152	0x0020_0000
Block Size	2,097,152 ÷ 32 = 65,536	0xFFFF
Sector Size	65,536 ÷ 16 = 4,096	0x1000
Page Size	256	0x100

Since the flash memory capacity is a very generous 16MB, even after writing a typical program, there will be significant unused space at the end.

Thus, we plan to store the calibration data there.

The calibration result consists of 22 `u8` fields (22 bytes), which easily fits into a single sector. Therefore, our strategy is to erase the last sector of flash memory and then write the calibration data there.

Caution: Directly manipulating flash memory is risky. Especially, writing operations require `unsafe` processing. To ensure stability, specific preparations are necessary.

Following the discussions in the `rp-rs/rp-hal` repository Issue #257 [9], we will implement flash memory reading and writing procedures.

[External Link] rp-rs/rp-hal Issue #257
https://github.com/rp-rs/rp-hal/issues/257

♣ Preparing for External Flash Operations

First, Issue #257 concludes that it is crucial to add the following settings to `Cargo.toml`:

▼ List 4.13: Optimization Settings: Cargo.toml

```
...
[profile.release]
codegen-units = 1
debug = 2
debug-assertions = false
incremental = false
lto = 'fat' # <-- HERE
opt-level = 3
overflow-checks = false
```

The most important setting is `lto` (Link Time Optimization), indicated with "# <-- HERE." It should be set to "fat" (equivalent to true), ensuring that LLVM (Rust compiler) performs optimizations across all crates during code generation [2].

Without this, programs involving flash memory operations will fail to compile.

♣ Reading from External Flash Memory

We will define functions required for flash operations. Since embedding all functions directly into a binary file would make it lengthy, and considering future reuse, we place them in `src/lib.rs`.

▼ List 4.14: Flash Read Function: src/lib.rs

```
#![no_std]
```

```
pub const XIP_BASE: u32 = 0x1000_0000; // ①
pub const FLASH_END: u32 = 0x0020_0000;

pub fn read_flash(data: &mut [u8]) {
    let size = data.len(); // ②
    let addr = XIP_BASE + FLASH_END - size as u32; // ③
    unsafe {
        let _ = core::slice::from_raw_parts(addr as *const u8, size)
            .iter()
            .zip(
                data.iter_mut()
            )
            .map(|(&elem, save)| {
                *save = elem;
            })
            .collect::<()>(); // ④
    }
}
```

① define constants for the start address (`XIP_BASE`) and end address (`FLASH_END`) of the external flash memory (Refer to Table 4.3).

In the `read_flash` function, we read `size` (②) bytes into the `data` buffer from the flash memory's end.

Since the flash content is mapped from `XIP_BASE`, the address to read from is (`XIP_BASE + FLASH_END - size`) (③).

We use `core::slice::from_raw_parts` to obtain the data and individually copy it into `data`.

♣ Writing to External Flash Memory

Next, add the following constants and functions to `src/lib.rs`:

▼ List 4.15: Flash Write Function: src/lib.rs

```
pub const FLASH_BLOCK_SIZE: u32 = 0xFFFF; // ①
pub const FLASH_SECTOR_SIZE: usize = 0x1000;
pub const FLASH_PAGE_SIZE: u32 = 0x100;
pub const FLASH_BLOCK_CMD: u8 = 0x20;
…
#[inline(never)]
#[link_section = ".data.ram_func"]
pub fn write_flash(data: &[u8]) {
    let size = data.len(); // ②
```

```
    let addr = FLASH_END - size as u32; // ③
    unsafe {
        cortex_m::interrupt::free(|_cs| { // ④
            rom_data::connect_internal_flash();
            rom_data::flash_exit_xip();
            rom_data::flash_range_erase(addr, FLASH_SECTOR_SIZE, FLASH_BL›
›OCK_SIZE, FLASH_BLOCK_CMD);
            rom_data::flash_range_program(addr, data.as_ptr(), size);
            rom_data::flash_flush_cache(); // Get the XIP working again
            rom_data::flash_enter_cmd_xip(); // Start XIP back up
        });
    }
}
```

The constants defined in ① are based on Table 4.3.

`FLASH_BLOCK_CMD` (0x20) is the erase command, as suggested in Issue #257.

Although the attributes `#[inline(never)]` and `#[link_section = ".data.ram_func"]` are technically optional, we retain them for safety.

The `write_flash` function writes the contents of the `data` array (type `u8`) to the end of the flash memory.

Given the data size `size` (②), the write start address is calculated as (`FLASH_END - size`) (③).

Writing operations are `unsafe`, and to prevent interference from interrupts, we wrap the writing process inside `cortex_m::interrupt::free` (④).

Inside `interrupt::free`, the following operations are performed:

1. Connect to flash memory
2. Temporarily stop XIP processing
3. Erase the last sector of flash memory
4. Write to the last sector
5. Resume XIP processing
6. Reconfigure XIP settings

These operations use Bootrom-provided functions available via `rp2040_hal::rom_data`:

- `flash_range_erase` for flash sector erasure
- `flash_range_program` for page-wise flash writing

Note: In the RP2040, flash erasure must occur in sector units (4,096 bytes), and writing must occur in page units (256 bytes).

Thus, the starting address `addr` must be aligned to a multiple of 4096. Effectively, all writes are performed on a sector basis.

♣ Saving Write Records

Our goal for saving calibration information involves the following behavior:

1. If calibration data already exists in external flash memory, load it.
2. If no calibration data exists, perform calibration and save the result.

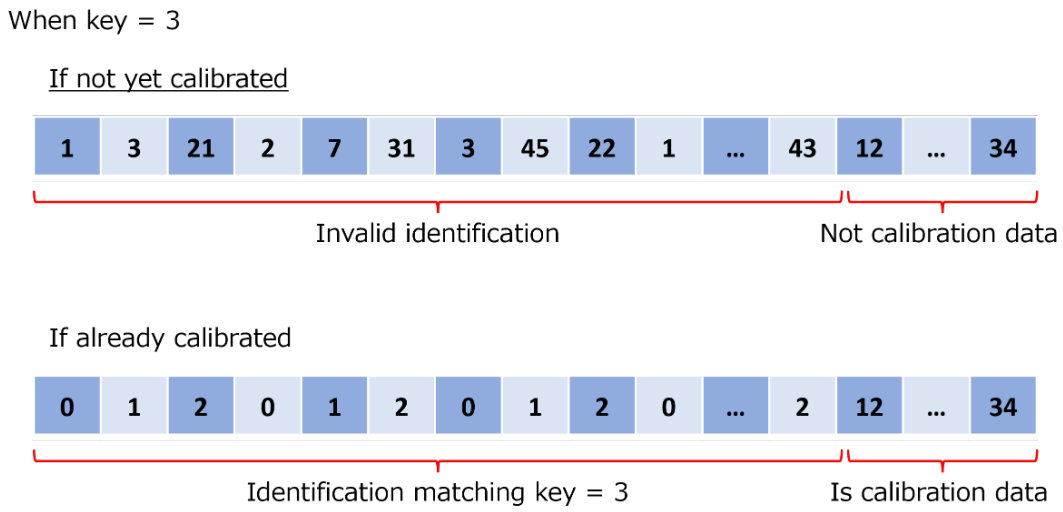
However, even if calibration data has not been saved, it is still technically possible to read arbitrary values from flash memory. Thus, we must also save a separate "record" indicating whether calibration data was intentionally saved.

Since the `write_flash` function always writes in sector-sized units (4096 bytes), we will use the space preceding the 22 bytes of calibration data to store the record.

Specifically, we dedicate 4096 - 22 = 4074 bytes at the beginning of the sector to store a unique pattern or "identification." If this pattern is detected, we consider calibration as saved; otherwise, it is considered unsaved.

As for generating the pattern, many methods are possible, but here, we simply create a repeating sequence based on a chosen key value (1 to 255).

The expected behavior is illustrated in Figure 4.4.



▲ Figure 4.4: Determining "Saved" Status

Two functions implement this logic:

▼ List 4.16: Generating and Verifying Identification: src/lib.rs

```
pub fn gen_identification(data: &mut [u8], key: u8) { // ①
    let key_usize = key as usize;
    let _ = data.iter_mut()
        .enumerate()
```

```
        .map(|(i, save)| {
            *save = (i % key_usize) as u8;
        })
        .collect::<()>();
}

pub fn check_identification(data: &[u8], key: u8, size: usize) -> bool { 
// ②
    let key_usize = key as usize;
    let mut valid = true;
    for (i, &elem) in data.iter().enumerate() {
        if (i % key_usize) as u8 != elem {
            valid = false;
            break;
        }
    }

    valid
}
```

- `gen_identification`: Generates a repeating sequence based on the `key` (①).
- `check_identification`: Checks whether an array matches the expected sequence rules based on the `key` (②).

♣ BNO055 Control Program with Calibration Storage

Using the above functions, we now build the final program.

Create a new file called `imu_save.rs` inside the `src` directory, and register it as a binary target in `Cargo.toml`.

▼ List 4.17: Adding bin Block: Cargo.toml

```
[[bin]] # End of File
name = "imu_save"
path = "src/imu_save.rs"
```

And, copy the contents from the sample code below:

[External Link] Sample Code for `src/imu_save.rs`
https://github.com/doraneko94/rp_bno055/blob/master/src/imu_save.rs

Major differences compared to `imu_base.rs` are explained below:

▼ List 4.18: Saving and Reusing Calibration: src/imu_save.rs

```
…
use bno055::BNO055Calibration; // ①
use rp_bno055::{
    read_flash, write_flash,
    gen_identification, check_identification,
    FLASH_SECTOR_SIZE
};
…
const DATA_SIZE: usize = FLASH_SECTOR_SIZE as usize; // ②
const CALIB_SIZE: usize = 22;
const IDENT_KEY: u8 = 192;
const IDENT_SIZE: usize = DATA_SIZE - CALIB_SIZE;
…
#[rp2040_hal::entry]
fn main() -> ! {
    …
    let mut calib_data = [0u8; DATA_SIZE]; // ③
    read_flash(&mut calib_data);
    if check_identification(&calib_data[..IDENT_SIZE], IDENT_KEY) { // ④
        let calib = read_calib_from_buf(&calib_data); // ⑤
        imu.set_calibration_profile(calib, &mut delay).unwrap();
    } else {
        let mut count = 0usize;
        'label: loop { // ⑥
            …
        }
        gen_identification(&mut calib_data, IDENT_KEY); // ⑦
        write_calib_to_buf(&mut calib_data, imu.calibration_profile(&mut ›
›delay).unwrap()); // ⑧
        write_flash(&calib_data);
    }
    …
}

fn read_calib_from_buf(data: &[u8; DATA_SIZE]) -> BNO055Calibration { // ›
›⑤'
    let mut calib_buf = [0u8; CALIB_SIZE];
    let _ = calib_buf.iter_mut()
        .zip(data[IDENT_SIZE..DATA_SIZE].iter())
        .map(|(bi, &di)| { *bi = di; } )
        .collect::<()>();
    BNO055Calibration::from_buf(&calib_buf)
```

```
}

fn write_calib_to_buf(data: &mut [u8; DATA_SIZE], calib: BNO055Calibratio
n) { // ⑧'
    let calib_buf = calib.as_bytes();
    let _ = calib_buf.iter()
        .zip(data[IDENT_SIZE..DATA_SIZE].iter_mut())
        .map(|(&bi, di)| { *di = bi; } )
        .collect::<()>();
}
```

- Import necessary crates and flash functions (①).
- Set constants (②):
  - `DATA_SIZE`: Sector size
  - `CALIB_SIZE`: 22 bytes (number of fields in `BNO055Calibration`)
  - `IDENT_SIZE`: Sector size minus 22
  - `IDENT_KEY`: Set to 192 (can be changed freely)
- Prepare a `calib_data` buffer of length `DATA_SIZE` (③).
- Read flash content into the buffer and verify the identification pattern (④).
- If the identification is valid:
  - Extract calibration data (⑤) and reconstruct the `BNO055Calibration` structure (⑤, ⑤').
- If the identification is invalid:
  - Enter calibration loop as usual (⑥).
- After calibration:
  - Fill buffer with identification (⑦).
  - Overwrite the last 22 bytes with calibration data (⑧, ⑧').

Finally, save the buffer into flash memory using `write_flash`.

Afterward, the infinite loop for displaying posture angles proceeds as usual.

♣ Verifying Operation

Once complete, execute the program:

▼ List 4.19: First Run of src/imu_save.rs: shell

```
C:\hogehoge\rp_bno055>cargo run --release --bin imu_save
...
sys: 0 acc: 0 gyr: 0 mag: 0 (elasped time: 1 sec)
...
sys: 3 acc: 3 gyr: 0 mag: 3 (elasped time: 53 sec)
Pitch: -17.50, Roll: 15.81, Yaw: 4.43
```

```
Pitch: -17.50, Roll: 15.81, Yaw: 4.43
...
```

Behavior will mirror that of `imu_base.rs`.

Now disconnect and reconnect the Raspberry Pi Pico while pressing the BOOTSEL button, and execute the command again:

▼ List 4.20: Second Run of src/imu_save.rs: shell

```
C:\hogehoge\rp_bno055>cargo run --release --bin imu_save
...
Pitch: 0.00, Roll: 0.12, Yaw: 0.00
Pitch: -3.50, Roll: 7.37, Yaw: 359.93
Pitch: -3.50, Roll: 7.50, Yaw: 359.93
Pitch: -3.50, Roll: 7.62, Yaw: 359.93
...
```

This time, posture information will immediately start being displayed! Thus, calibration was successfully skipped.

♣ Persistence of Calibration Data

In List 4.19, calibration data was saved into external flash memory.

Even after erasing the program and rewriting `src/imu_save.rs` (as in List 4.20), the calibration data and identification remain intact.

This happens because:

- Programs like `imu_save.rs` or `serial_hello.rs` are too small to overwrite the end of flash memory.
- There is almost 16MB of space before reaching the sector holding the calibration data.

Thus, unless a significantly large program overwrites the entire flash, the calibration data persists.

♣ Resetting Calibration

What if you want to discard the old calibration data and redo calibration?

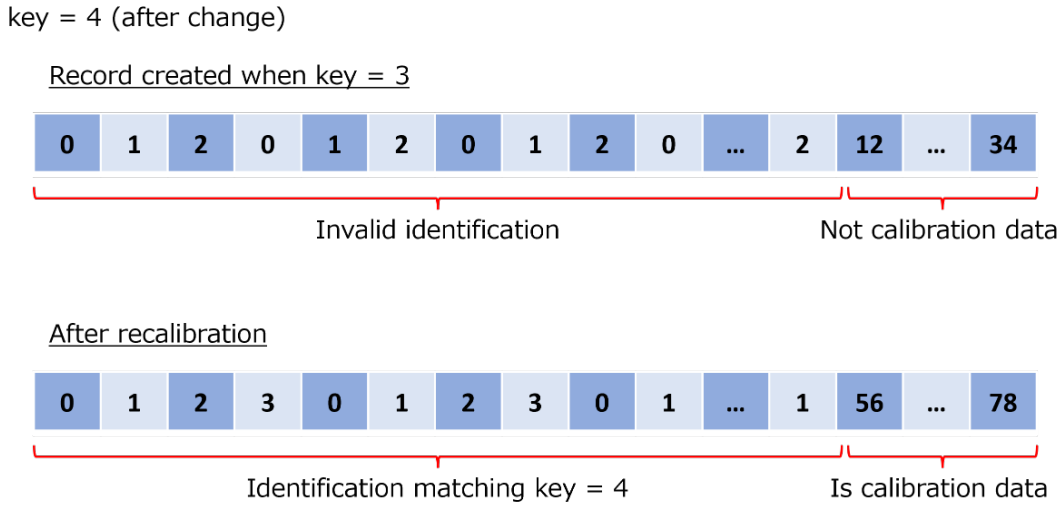
Since flashing a new program does not erase saved data, we need a strategy.

One simple approach is to change the `IDENT_KEY` used to generate the identification pattern. Changing the key invalidates old records.

Illustrated in Figure 4.5:

When the key changes (e.g., from 192 to 191), the old identification pattern becomes invalid, forcing recalibration.

Try changing `IDENT_KEY`:



▲ Figure 4.5: Invalidating Saved Data by Changing Key

▼ List 4.21: Changing IDENT_KEY: src/imu_save.rs

```
...
const IDENT_KEY: u8 = 191; // Previously 192
...
```

▼ List 4.22: Running After Changing IDENT_KEY: shell

```
C:\hogehoge\rp_bno055>cargo run --release --bin imu_save
...
sys: 0 acc: 0 gyr: 0 mag: 0 (elasped time: 1 sec)
sys: 0 acc: 0 gyr: 0 mag: 0 (elasped time: 2 sec)
...
```

Calibration will start again, as expected.

Appendix A

Installation Failed → Try Downgrading Rust.

As of April 27, 2023, the latest stable version of Rust is 1.69. However, there have been reports of a bug that causes the installation of `elf2uf2-rs` to fail under this environment.

[External Link] elf2uf2-rs Issue #17
https://github.com/JoNil/elf2uf2-rs/issues/17

It appears that `elf2uf2-rs` has not yet been updated to accommodate the changes introduced when Rust was upgraded from version 1.68 to 1.69. If you encounter installation failures due to this issue, try downgrading your Rust version by following the steps below.

A.1 Checking Your Current Version

First, check your current Rust version by running the following command. If you receive an output similar to the example below, it means that your Rust version is 1.69.

▼ List 1.1: Checking Rust Version: shell

```
C:\hogehoge>rustc -V
rustc 1.69.0 (84c898d65 2023-04-16)
```

A.2 Adding a Specific Version of Rust

Next, add the toolchain for the specified version of Rust using the following command. Here, we specify version 1.68.

▼ List 1.2: Add toolchain: shell

```
C:\hogehoge>rustup toolchain add 1.68
```

A.3 Verifying the Added Version

You can retrieve a list of all installed toolchain versions by running the following command.

▼ List 1.3: Checking the Toolchain List: shell

```
C:\hogehoge>rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
nightly-2020-01-02-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
1.68-x86_64-pc-windows-msvc
```

If the output appears as shown above, you can confirm that version 1.68 has been successfully installed (see the end of the list).

A.4 Changing the Default Version

From the (default) notation in List 1.3, we can see that the current default is the stable version (stable-x86_64-pc-windows-msvc; version 1.69 as of April 27, 2023). You can change the default to version 1.68 by executing the following command.

▼ List 1.4: Default Switching: shell

```
C:\hogehoge>rustup default 1.68-x86_64-pc-windows-msvc
}

== Verifying the Changes

Check the list of toolchains again to verify that the default has been up›
›dated.

//list[toolchain_list_after][Checking the Toolchain List: shell]{
C:\hogehoge>rustup toolchain list
stable-x86_64-pc-windows-msvc
nightly-2020-01-02-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
1.68-x86_64-pc-windows-msvc (default)
```

Additionally, you can confirm that the Rust version has been downgraded by running the same command we executed at the beginning.

▼ List 1.6: Checking the rustc Version After the Change: shell

```
C:\hogehoge>rustc -V
rustc 1.68.2 (9eb3afe9e 2023-03-27)
```

At this point, version-related errors should have been resolved, so let's try reinstalling `elf2uf2-rs`. However, note that the development target corresponding to version 1.68 may not yet be installed, so we will reinstall it together.

▼ List 1.7: Reinstalling the Target and Tool: shell

```
C:\hogehoge>rustup target add thumbv6m-none-eabi
C:\hogehoge>cargo install elf2uf2-rs – locked
```

With this, the setup of a stable development environment should be complete.

Appendix B

Blinking the LED at an Accurate 1-Second Interval

As noted in the comments within List 2.6, the wait time provided by `Delay` is actually quite rough. Therefore, if you need precise timing, you should avoid using functions like the `delay_ms` method.

To measure time more accurately, you should use the `Timer`. We have prepared `src/led_timer.rs`, which rewrites `src/led.rs` to utilize a `Timer` instead. Please refer to the sample code in the author's GitHub repository:

[External Link] Sample Code for `src/led_timer.rs`
https://github.com/doraneko94/rp_bno055/blob/master/src/led_timer.rs

▼ List 2.1: Implementing Wait Times Using a Timer: src/led_timer.rs

```
...
let timer = hal::timer::Timer::new(pac.TIMER, &mut pac.RESETS); // ①
...
loop {
    led_pin.set_high().unwrap();
    let start = timer.get_counter().ticks(); // ②
    while timer.get_counter().ticks() - start < 500_000 {} // ③
    led_pin.set_low().unwrap();
    let start = timer.get_counter().ticks(); // ②'
    while timer.get_counter().ticks() - start < 500_000 {} // ③'
}
```

The `Timer` structure defined in ① increments its counter (`ticks`) by one every microsecond. Thus, we first save the counter value at the moment when we start measuring time (②), and then, we use a `while` loop to wait until the difference between the current counter value and

the starting value reaches 500,000 microseconds (0.5×10^6 microseconds = 0.5 seconds) (③).

Appendix C
Starting Serial Communication

The `elf2uf2-rs` tool, which we installed in "1.3 Installing the Build Tool", is used to write Rust programs to the Raspberry Pi Pico. In `.cargo/config`, it is configured as follows:

▼ List 3.1: elf2uf2-rs Configuration: .cargo/config

```
…
runner = "elf2uf2-rs -d -s"
```

Here, two options are specified:

- `-d`: Automatically deploy to the mounted Raspberry Pi Pico
- `-s`: After deployment, open the Raspberry Pi Pico as a serial device and print its serial output

Thus, after writing and executing the program via the `cargo run` command, serial communication is started, and as long as the communication remains alive (i.e., `usb_dev.poll` continues to be called within 10 milliseconds), the transmitted data will be displayed.

Let's check how the `-s` option is handled inside the implementation of `elf2uf2-rs`.

C.1 How `elf2uf2-rs` Handles Serial Communication

▼ List 3.2: Serial Communication in elf2uf2-rs: elf2uf2-rs/main.rs

```
…
let serial_ports_before = serialport::available_ports()?; // ①
…
{
… // ②
}
if Opts::global().serial { // ③
let mut counter = 0;
```

```
    let serial_port_info = 'find_loop: loop {
        for port in serialport::available_ports()? { // ④
            if !serial_ports_before.contains(&port) { // ⑤
                println!("Found pico serial on {}", &port.port_name);
                break 'find_loop Some(port);
            }
}

counter += 1;

        if counter == 10 {
            break None;
        }

        thread::sleep(Duration::from_millis(200));
    };

    if let Some(serial_port_info) = serial_port_info { // ⑥
        for _ in 0..5 {
            if let Ok(mut port) = serialport::new(&serial_port_info.port_›
›name, 115200)
                .timeout(Duration::from_millis(100))
                .flow_control(FlowControl::Hardware)
                .open() // ⑦
            {
                if port.write_data_terminal_ready(true).is_ok() {
                    let mut serial_buf = [0; 1024]; // ⑧
                    loop { // ⑨
                        match port.read(&mut serial_buf) {
                            Ok(t) => io::stdout().write_all(&serial_buf[.›
›.t])?,
                            Err(ref e) if e.kind() == io::ErrorKind::Time›
›dOut => (),
                            Err(e) => return Err(e.into()),
                        }
                    }
                }
            }

            thread::sleep(Duration::from_millis(200));
        }
    }
}
```

The overall flow consists of:

1. Preparation (①)
2. Deploying the program to the Raspberry Pi Pico (②)
3. Obtaining communication ports (③–⑤)
4. Starting and maintaining communication (⑥–⑨)

Let's go through each step:

- Before deployment, a list of currently available communication ports is acquired (①).
- In ②, the main processing block, the program is deployed to the Raspberry Pi Pico. If the program contains serial communication processing, a new port will open.
- In ③, if the `-s` option is set, processing for receiving information from the Raspberry Pi Pico via serial communication begins.
- After deployment, the list of available ports is acquired again (④), and each is checked.
- If a port was not included in the pre-deployment list, it is considered a newly opened port for communication (⑤).
- This check is repeated up to 10 times at 200 ms intervals. If no new port is found, the process ends (⑥).
- If a new port is found (⑥), communication is established with it (⑦).
- Connection attempts are made up to 5 times at 200 ms intervals.
- Upon success, a communication buffer is prepared (⑧) and the program enters an infinite loop to continuously read incoming data (⑨).

This is why serial output appears automatically in the shell after deploying with `elf2uf2-rs`.

C.2 Serial Communication Without `elf2uf2-rs`

If you connect the Raspberry Pi Pico to your PC without pressing the BOOTSEL button, it will not enter write mode; instead, the previously written program will execute. However, even if the program includes serial communication, nothing will be displayed on the PC screen because there is no receiver handling the serial data.

Therefore, we created software that can read from the serial port after the Pico has been connected:

[External Link] `oscillo_serial` GitHub Repository
https://github.com/doraneko94/oscillo_serial

This software has two modes: `text` and `plot`. Here, we will focus only on `text` mode. (The

plot feature will be explained in Appendix D "Graph Display of Output Information")

First, clone and set up the software locally:

▼ List 3.3: Obtaining oscillo_serial: shell

```
C:\hogehoge>git clone https://github.com/doraneko94/oscilli_serial
```

Next, connect a Raspberry Pi Pico (with a serial communication program, such as serial_hello.rs written) to the PC without pressing the BOOTSEL button. Move into the oscillo_serial directory and execute the following command:

▼ List 3.4: Running oscillo_serial: shell

```
C:\hogehoge>cd oscillo_serial
C:\hogehoge\oscillo_serial>cargo run -- -mo text
```

You should then see output similar to List 3.7.

C.3 Implementation of oscillo_serial

Let's look at how oscillo_serial implements these features:

▼ List 3.5: Overview of Code: oscillo_serial/src/main.rs

```
…
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let args: Vec<String> = std::env::args().collect(); // ①
    let params = Params::from_args(&args);
    …
    for port in serialport::available_ports()? {
        match serialport::new(port.port_name, 115200)
            .timeout(std::time::Duration::from_millis(10))
            .flow_control(serialport::FlowControl::Hardware)
            .open() // ②
        {
            Ok(mut port) => {
                let mut buf = [0; 1024];
                match params.mode {
                    Mode::Dev => {}
                    Mode::Plot => { … }
                    Mode::Text => { … // ③: refer to next list }
            _ => {}
        }
    };
```

```
    Ok(())
}
```

- First, various parameters are retrieved (①).
- Then, serial communication is established (②).

Unlike `elf2uf2-rs`, this software aggressively attempts to connect to all available ports. Already active ports reject the connection, and eventually, the correct port (the Pico's port) is found.

After preparing a communication buffer, if the `-mo` parameter indicates `text` mode, processing transitions to ③.

Due to the deep indentation, the contents of ③ are shown separately in:

▼ List 3.6: Expanded View of ③: oscillo_serial/src/main.rs

```
loop {
match port.read(&mut buf) {
        Ok(t) => {
            print!("{}", String::from_utf8(buf[..t].to_vec()).unwrap()); 
// ④
            //io::stdout().write_all(&mut buf[..t]);
        }
        Err(ref e) if e.kind() == io::ErrorKind::TimedOut => (),
        Err(e) => return Err(e.into()),
    }
}
```

That said, the processing here is simple: It consists of basic error handling and text display. Unlike `elf2uf2-rs`, `oscillo_serial` uses the `print!` macro rather than `io::stdout` and converts the buffer contents into a `String` before outputting.

Appendix D
Graph Display of Output Information

When viewing raw numerical output like that shown in List 4.5, it can be difficult to intuitively grasp value fluctuations. Therefore, let us display the output as a graph.

Here, we use the `plot` feature of `oscillo_serial` introduced in Appendix C "Starting Serial Communication" to plot the time series output from the BNO055.

[External Link] `oscillo_serial` GitHub Repository
https://github.com/doraneko94/oscillo_serial

D.1 Code Modification

Because the `plot` feature imposes certain formatting constraints, we must adjust the output format in `src/imu_save.rs`.

Create a new file `imu_save_plot.rs` inside the `src` directory, and register it as a binary target in `Cargo.toml`.

▼ List 4.1: Adding bin Block: Cargo.toml

```
[[bin]] # End of File
name = "imu_save_plot"
path = "src/imu_save_plot.rs"
```

Refer to the following sample code when writing `src/imu_save_plot.rs`:

[External Link] Sample Code for `src/imu_save_plot.rs`

https://github.com/doraneko94/rp_bno055/blob/master/src/imu_save_plot.rs

This file modifies only the infinite loop section compared to `src/imu_save.rs`.

List 4.2: Changing Display Content in the Loop: src/imu_save_plot.rs

```
loop {
    for _ in 0..20 {
        delay.delay_ms(5);
        let _ = usb_dev.poll(&mut [&mut serial]);
    }
    let eular = imu.euler_angles().unwrap();
    let _ = writer.write_f32(eular.a, 2, &mut serial);
    let _ = writer.write_str(",", &mut serial);
    let _ = writer.writeln_f32(eular.b, 2, &mut serial);
}
```

D.2 Running the Program

When executed, the output will now consist of pitch and roll values separated by commas, as shown below:

List 4.3: Raw Data Output by src/imu_save_plot.rs: shell

```
C:\hogehoge\rp_bno055>cargo run --release --bin imu_save_plot
...
0.00,0.12
-5.62,2.31
-5.62,2.43
...
```

If calibration has not yet been completed with the current `IDENT_KEY`, the calibration process will start first.

After confirming calibration is complete, disconnect the USB once, then open a new Command Prompt to use `oscillo_serial`.

Reconnect the Raspberry Pi Pico to the PC without pressing the BOOTSEL button, and execute the following command to launch `oscillo_serial` in `plot` mode:

List 4.4: Running oscillo_serial in Plot Mode: shell

```
C:\hogehoge\oscillo_serial>cargo run -- -mo plot -xs 100 -ys 20 -de , -ne
› 2
```

Each parameter in the command corresponds to:

▼ Table D.1: oscillo_serial Parameters

Parameter	Setting Target	Possible Values
-mo	Mode	text, plot
-xs	X-axis size	Integer
-ys	Y-axis size	Integer
-de	Element delimiter	String
-ne	Number of elements	Integer

In this case, we set it to `plot` mode, with a graph area of 100 × 20, displaying two elements (pitch and roll) separated by commas.

Thus, after launch, the output will look like this in real time (Figure D.1):

▲ Figure D.1: Output in Plot Mode

- ① represents a period where the breadboard was rotated along the roll axis only.
- ② represents a period where the breadboard was rotated along the pitch axis only.

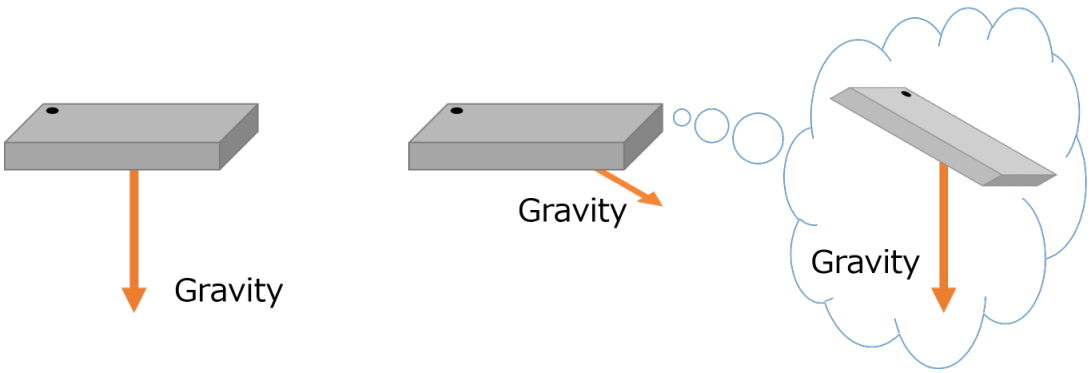
For details about the implementation of `plot` mode in `oscillo_serial`, please refer to the GitHub repository.

Note that `oscillo_serial` is under active development, and both its implementation and specifications are subject to change.

Appendix E

Principle of Posture Angle Estimation Using Sensor Information

E.1 Accelerometer



▲ Figure E.1: Posture Estimation Using an Accelerometer

When an accelerometer is placed horizontally, gravitational acceleration is detected along the Z-axis (Figure E.1, left). Conversely, if the detected acceleration vector points in a direction other than the Z-axis, since gravity's direction itself cannot change, we can infer that the device itself must be tilted (Figure E.1, right).

However, even if the device is rotated within the horizontal plane, the direction of the detected acceleration vector remains unchanged, meaning that rotation along the horizontal plane cannot be distinguished by the accelerometer.

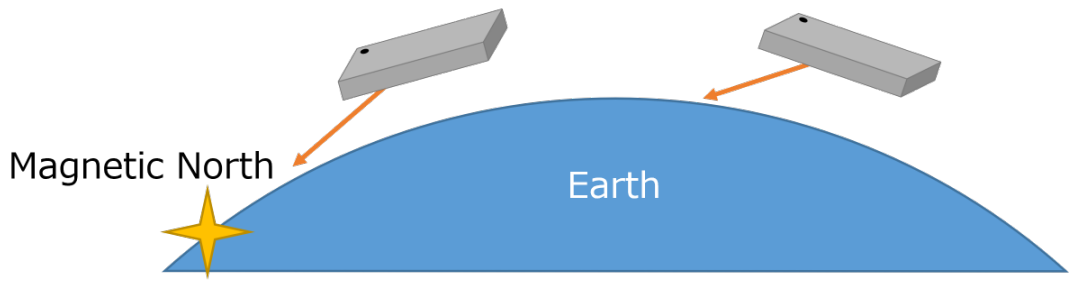
Accelerometers are known to be highly sensitive and prone to fine noise. Moreover, if the accelerometer is moving, it will be affected by forces other than gravity, making it difficult to accurately estimate posture angles based purely on acceleration measurements.

E.2 Gyroscope

A gyroscope measures rotational velocity (angular velocity) around each of the XYZ axes. Thus, by integrating the measured values (i.e., multiplying each by the sampling time Δt and summing), we can calculate how much the posture angle has changed since the beginning of measurement.

As a result, the posture angle estimated by a gyroscope is always relative. Also, because integration accumulates small errors over time, a phenomenon known as drift occurs.

E.3 Magnetometer



▲Figure E.2: Posture Estimation Using a Magnetometer

A magnetometer measures the direction of magnetic north (which slightly deviates from true north) and estimates the device's posture angle based on that vector.

Because it references a fixed standard (magnetic north), the resulting posture angles are absolute.

However, as shown in Figure E.2, the measurement values can vary depending on the device's position on Earth. (The devices in the figure maintain the same orientation relative to the ground.)

If the device cannot determine its own latitude and longitude, this positional dependency introduces large errors.

Furthermore, magnetometers are highly sensitive to local magnetic field disturbances, meaning that their usage environment is significantly limited.

E.4 Posture Angle Estimation

From the above, it is clear that using a single sensor alone is insufficient for reliable posture estimation. Thus, in practice, posture estimation devices combine readings from multiple sensors—a method called **sensor fusion**.

For example, the complementary filter expressed by Equation5.1 combines accelerometer and gyroscope measurements:

Equation5.1: Complementary Filter Update Formula

$$\theta(t) = K \cdot [\theta(t-1) + \omega_g(t) \cdot \Delta t] + (1 - K) \cdot \theta_a(t)$$

where:

- $\theta(t)$: Estimated posture angle at time t
- $\omega_g(t)$: Gyroscope measurement at time t
- $\theta_a(t)$: Estimated posture angle from accelerometer measurements only

You can think of it as a weighted average of the gyroscope and accelerometer estimations, with weight coefficient K.

In practice, this filter acts to suppress:

- High-frequency noise from the accelerometer
- Low-frequency drift from the gyroscope

The coefficient K can be calculated from the filter's cutoff frequency f_c as follows:

Equation5.2: Calculation of K

$$K = \frac{\frac{1}{2\pi f_c}}{\frac{1}{2\pi f_c} + \Delta t}$$

Thus, by combining multiple sensor outputs, we can both improve accuracy and derive new, more robust information—this is the essence of sensor fusion.

Appendix F

Operating Modes of the BNO055

Oprerating Modes		Available Sensor			Heading	
		ACC	MAG	GYRO	Relative	Absolute
	CONFIGMODE					
Non-Fusion Modes	ACCONLY	✓				
	MAGONLY		✓			
	GYROONLY			✓		
	ACCMAG	✓	✓			
	ACCGYRO	✓		✓		
	MAGGYRO		✓	✓		
	AMG	✓	✓	✓		
Fusion Modes	IMU	✓		✓	✓	
	COMPASS	✓	✓			✓
	M4G	✓	✓		✓	
	NDOF_FMC_OFF	✓	✓	✓		✓
	NDOF	✓	✓	✓		✓

▲ Figure F.1: Operating Modes of the BNO055

(Figure F.1 is based on [6] Table 3-3)

F.1 CONFIGMODE

Immediately after powering on, the device enters this mode. In CONFIGMODE, no sensor readings are available, but various settings such as axis remapping, sensor ranges, and

bandwidths can be configured.

F.2 Non-Fusion Modes

This group of modes uses only the raw readings from individual sensors. Each mode is named directly after the sensor(s) being used.

F.3 Fusion Modes

Fusion modes combine measurements from multiple sensors to calculate the device's orientation in space.

Among the orientation angles, whether the heading (azimuth) output is absolute or relative depends on the specific fusion mode.

- Absolute heading: Measured as the angle from magnetic north (0 degrees), and requires use of the magnetometer.
- Relative heading: Measured as the angle relative to the orientation at power-on, without reference to magnetic north.

Additionally, in fusion modes, the BNO055 can output:

- Linear acceleration (with the gravitational component removed)
- Gravity vector (the direction of gravity relative to the device)

Normally, the accelerometer's output includes both gravitational effects and actual movement-induced accelerations. However, by incorporating gyroscope and magnetometer data, the BNO055 can separate these two influences.

Below are brief descriptions of each fusion mode:

- **IMU**: Uses accelerometer and gyroscope. Fast response.
- **COMPASS**: Uses accelerometer and magnetometer. Heading is calculated using only the horizontal (XY) components of the magnetic field. Caution is needed in magnetically unstable environments.
- **M4G**: Uses the magnetometer but treats the magnetic field as displacement (like a gyroscope), not as an absolute heading reference.
  - Advantages: No gyroscope drift.
  - Limitations: Cannot determine magnetic north, resulting in a relative heading.
  - Therefore, no need (nor ability) to calibrate the magnetometer. Again, caution is needed in unstable magnetic environments.
- **NDOF_FMC_OFF**: Same as NDOF but without Fast Magnetometer Calibration.

- **NDOF**: Fully integrated mode using all sensors. Fast Magnetometer Calibration accelerates and improves magnetometer accuracy. Thanks to accelerometer and gyroscope correction, NDOF mode is more tolerant to unstable magnetic environments than COMPASS or M4G.

F.4 Important Caution for Non-NDOF Modes

When using modes other than **NDOF** and **NDOF_FMC_OFF**, do not use the `Bno055::is_fully_calibrated` function to judge calibration completion.

Looking at its implementation:

▼ List 6.1: Implementation of is_fully_calibrated: bno055/src/lib.rs (GitHub)

```
pub fn is_fully_calibrated(&mut self) -> Result<bool, Error<E>> {
    let status = self.get_calibration_status()?; // ①
    Ok(status.mag == 3 && status.gyr == 3 && status.acc == 3 && status.sy
s == 3) // ②
}
```

we see that it checks whether all calibration values (SYS, ACC, GYR, MAG) have reached 3 (②).

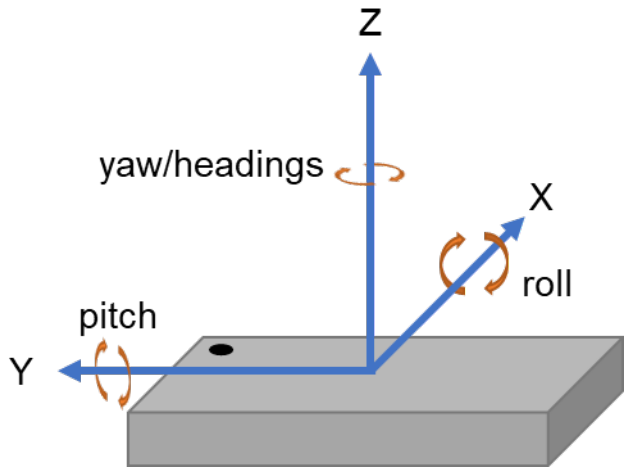
However, if a mode does not use certain sensors, those unused sensors' calibration values will remain at 0 indefinitely, meaning that calibration would never complete.

Thus, as done internally by the library, you should instead:

- Retrieve the `bno055::BNO055CalibrationStatus` structure (①)
- Check only the calibration values relevant to the currently active sensors.

Appendix G
Handling of Euler Angles in the BNO055

G.1 Differences: This Book, BNO055, and Crates



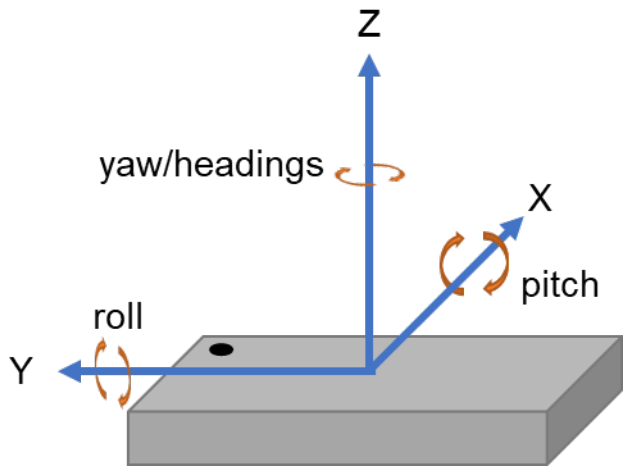
▲Figure G.1: Relationship between Euler Angles and XYZ Axes (reposted)

In this book, we define the names of the rotation angles based on Figure G.1. We follow the general interpretation that considers the X-axis as the forward direction, and the rotation around it as "roll."

【External Link】Aircraft principal axes – Wikipedia

https://en.wikipedia.org/wiki/Aircraft_principal_axes

However, according to official information (which is actually not documented in [6], but can be found in other BNO055-related resources such as [8]), Euler angles are mapped differently:



▲ Figure G.2: Official Relationship between Euler Angles and XYZ Axes

Data is stored in the registers starting from `EUL_DATA_LSB` (`0x1A`) in the order of yaw/heading, roll, and pitch.

Next, let us look at how the `bno055` crate handles Euler angles.

According to the Docs-rs documentation for the `euler_angles` method:

"Get Euler angles representation of heading in degrees. Euler angles is represented as (roll, pitch, yaw/heading)."

Indeed, inspecting the implementation:

▼ List 7.1: Implementation of euler_angles: bno055/src/lib.rs

```
/// Get Euler angles representation of heading in degrees.
/// Euler angles is represented as (`roll`, `pitch`, `yaw/heading`).
pub fn euler_angles(&mut self) -> Result<mint::EulerAngles<f32, ()>, Erro›
›r<E>> {
    self.set_page(BNO055RegisterPage::PAGE_0)?;

    // Device should be in fusion mode to be able to produce quaternions
    if self.is_in_fusion_mode()? {
        let mut buf: [u8; 6] = [0; 6];

        self.read_bytes(regs::BNO055_EUL_HEADING_LSB, &mut buf)
            .map_err(Error::I2c)?;
```

```
        let heading = LittleEndian::read_i16(&buf[0..2]) as f32; // ①
        let roll = LittleEndian::read_i16(&buf[2..4]) as f32;
        let pitch = LittleEndian::read_i16(&buf[4..6]) as f32;

        let scale = 1f32 / 16f32; // 1 degree = 16 LSB

        let rot = mint::EulerAngles::from([roll * scale, pitch * scale, h
›eading * scale]); // ②

        Ok(rot)
    } else {
        Err(Error::InvalidMode)
    }
}
```

we find that:

It retrieves the angles in the order of heading, roll, pitch (①) , then rearranges them into the order roll, pitch, heading before storing them in a `mint::EulerAngles` structure (②).

Finally, let's examine how the `mint::EulerAngles` structure itself expects Euler angles to be organized.

According to the `mint` crate's Docs-rs documentation:

"Fields

a: T

[-] First angle of rotation in range [-pi, pi] (pitch).

b: T

[-] Second angle of rotation around in range [-pi/2, pi/2] (yaw).

c: T

[-] Third angle of rotation in range [-pi, pi] (roll)."

Thus, `mint::EulerAngles` expects the order to be pitch, yaw, and roll. Clearly, the `bno055` crate completely disregards this ordering.

Furthermore, the author believes that the names roll and pitch should be swapped compared to what the BNO055 provides.

Therefore, in this book, we manually reassign them. Summarizing the relationship:

▼ Table G.1: Handling of Euler Angles across mint, bno055, and This Book

Field in mint::EulerAngles	Expected by mint	Stored by bno055	Used in this book
a	pitch	roll	pitch
b	yaw/heading	pitch	roll
c	roll	yaw/heading	yaw/heading

G.2 Differences in Angle Ranges

Besides naming differences, the range of values also differs between `mint` and `bno055`:

▼ Table G.2: Differences in Euler Angle Ranges

Euler Angle	mint crate expected range	bno055 output range
pitch	-π to π	-90° to 90°
roll	-π to π	-180° to 180°
yaw/heading	-π /2 to π /2	0° to 360°

The BNO055 technically supports outputting Euler angles in radians, but the `bno055` crate does not expose this feature.

G.3 Other Known Issues with BNO055 Euler Angles

Beyond naming and range mismatches, the BNO055 Euler angle outputs are also known to suffer from:

- Low precision (coarse quantization)
- Unstable yaw values when pitch or roll exceeds $\pm$ 45°

To avoid these problems, we recommend using quaternions rather than Euler angles whenever possible.

Appendix H

If You Want to Use a Sensor other than BNO055

If you find it difficult to obtain a BNO055 module, you will need to consider using another IMU sensor.

There might be no problem on the hardware side, as the Raspberry Pi Pico can control a wide range of sensors. However, when developing in Rust, the difficulty of development can vary significantly depending on the sensor you choose.

When selecting a sensor, we recommend searching for its name on **crates.io** first.

[External Link] crates.io
https://crates.io

If a crate corresponding to that sensor is available, development will proceed much more smoothly.

If no existing crate is published, you will need to create your own control crate based on the sensor's datasheet.

References

Rust Basics

[1] The Rust Programming Language
https://doc.rust-lang.org/stable/book/

Cargo.toml Syntax

[2] The Cargo Book
https://doc.rust-lang.org/cargo/reference/

Cargo Glossary

[3] The Rust Reference
https://doc.rust-lang.org/stable/reference/

Embedded Rust

[4] The Embedded Rust Book
https://docs.rust-embedded.org/book/index.html
[5] The Embedonomicon
https://docs.rust-embedded.org/embedonomicon/index.html

BNO055

[6] Bosch Sensortec. Data sheet: BNO055 - Intelligent 9-axis absolute orientation sensor. (2021)
https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf
[7] 9-axis sensor fusion module with BNO055 AE-BNO055-BO Ver.2022-4-13
https://akizukidenshi.com/download/ds/akizuki/AE-BNO055-BO_20220413.pdf

Euler Angles of BNO055

[8] readOrientation - MATLAB Help Center

https://www.mathworks.com/help/matlab/supportpkg/bno055.readorientation.readorientation.html

Reading/Writing Flash Memory on Raspberry Pi Pico

[9] rp-rs/rp_hal Issue #257
https://github.com/rp-rs/rp-hal/issues/257