Rust x Pico

艮電算術研究所 - ラスピコロボット開発プロジェクト 進捗報告 Vol. 1

Rust x Raspberry Pi Picoで実装する

IMUからの姿勢情報の取得と応用

大野 駿太郎 [著]

艮電算術研究所

はじめに

まえがき

Rust 言語は堅牢性・高速性・並列性を誇る言語として、現在、世界で高いパフォーマンスが求められるプログラムの開発に使用されています。そして筆者は、Rust は上記の 3 点に加え、コード資産の管理が非常に優れている点が、今後も世界で重宝されていく理由になると考えています。具体的には、Rust のライブラリ（クレート）は一括して crates.io で管理され、バージョン情報も厳密に保存されます。従来のプログラミング言語環境では、せっかく作ったライブラリが行方不明になることは日常茶飯事で、コードを人類の遺産として残すのは困難なことでした。しかし Rust の登場によりこの問題は解決され、今日書いたコードが明日以降にも活用されていくことが保証されるようになりました。これが、近年のソフトウェア開発環境における大きな変化です。

一方、ハードウェア開発環境にも嬉しい進歩がありました。英国の Raspberry Pi 財団設計の RP2040 チップを搭載した、Raspberry Pi Pico ボードの発売です。数百円で買えるこの小さなマイコンボードの中には、GPIO（01 入出力）・ADC（アナログ入出力）・SPI・I²C・UART という基本的な機能の他、Programmable Input/Output（PIO）という、任意の機能をもつ入出力を設計できる機構を備えています。そして安価かつ小型であるということは、個人開発の規模でも多くの設計や実験を行い、高度な機能を持ったメカを作ることができるということです。

この Raspberry Pi Pico を動かす言語として、Rust コミュニティも rp2040_hal 等のクレートを作り、名乗りを上げました。著者はこの、Rust と Raspberry Pi Pico の組み合わせによる開発を「ラスピコ（Rust x Pico）」と名付け、Rust の活用や電子工作の入門として推奨しています。Rust による組み込み開発、ならびに「ラスピコ」の活動は、まだ始まったばかりです。しかし、Rust 環境のコード資産管理技術を背景に、これから着実に歴史を積み重ねていくことができます。あなたの「ラスピコ」ライフの端緒として、この本がお役に立てたら幸いです。

艮電算術研究所 – ラスピコロボット開発プロジェクトとは

艮電算術研究所は著者が所属するサークルならびに Web サイト（https://ushitora.net）であり、電子計算機を用いた高度な計算技術（電算術）を用いて社会の発展に寄与することを

目標としています。従来は現象の解析をメインに活動してきたのですが、電算術が社会に影響を与えるには、コンピュータの中だけでなく、現実の 3 次元の社会と物質的な相互作用を起こすことが必要であると思い至り、それを実現するための方法として、Rust と Raspberry Pi Pico を使ったロボット開発プロジェクトを始動しました。本書はその第 1 回の進捗報告として、ロボットやドローンの姿勢制御に不可欠な IMU（慣性計測ユニット）の取り扱いについて、手法や TIPS を紹介したいと思います。

この本の目標と、目標にしないこと

タイトルの通り、この本では Raspberry Pi Pico と IMU（慣性計測ユニット）を使用した回路を作成し、Rust でソフトウェアを記述して、姿勢情報を取得・表示することを目指します。したがって、説明の都合で必要とならない限り、本書では以下の内容を詳しくは扱いません。

- Rust によるプログラミングの基本
- Rust による組込みプログラミングの総論
- Git の使い方

また、本書では IMU として BNO055 を使用します。このモジュールには姿勢推定の方法が標準で備わっているため、以下のような内容も詳しくは扱いません。

- 加速度・ジャイロセンサフュージョンによる姿勢推定の計算
- （拡張）カルマンフィルタ

これらの内容については、巻末の「参考文献リスト」にまとめた文献を参照してください。

以上より、本書の目標は以下の 3 点です。

- Rust で Raspberry Pi Pico を動かす
- Rust で Raspberry Pi Pico を操作し、情報を表示する
- Rust と Raspberry Pi Pico で IMU を適切に設定し、姿勢情報を取得・表示する

これらは各章の内容と対応しています。

注意書き

- 本書で紹介した企業名・商品名等は、一般に各企業の商標または登録商標です。本書では、一般的な商標マークの記載を省略しました。
- 本書の内容によって利用者に不利益や損害が生じる場合であっても、著者並びに発行元は一切責任を負いません。
- 著作権者や発行元の許可なく、本書の内容の一部およびすべてを複製、転載または配布、印刷など、第三者の利用に供することを禁止します。

サポートページ

本書で用いるサンプルコードはすべて、著者の Github リポジトリ

【外部リンク】rp_bno055 サンプルコードリポジトリ
https://github.com/doraneko94/rp_bno055

に公開されていますので、適宜参照してください。

検証環境

本書で記述するコードの動作確認について、著者は以下の環境で動作確認を行いました。

OS：Windows 11 Home
Rust：rustc 1.65.0

目次

第 0 章：環境構築とパーツの入手

0.1 Rust のインストール

まずは Rust がなくては話が始まりません。以下の公式サイト等を参考に、Rust をインストールしてください。

【外部リンク】Rust をインストール
https://www.rust-lang.org/ja/tools/install

0.2 Rust のアップデートとターゲットの追加

既知のバグを避けるため、Rust を最新の状態にします。また、Raspberry Pi Pico での開発を可能にするために、開発ターゲットとして thumbv6m-none-eabi を追加します。コマンドプロンプトを開き、以下のコマンドを実行してください。

リスト 0.2-1｜アップデートとターゲット追加：shell

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

0.3 ビルドツールのインストール

Rust で記述したプログラムを、Raspberry Pi Pico に書き込める UF2 という形式に変換するためのツールとして、elf2uf2-rs をインストールします。コマンドプロンプトを開き、以下のコマンドを実行してください。

リスト 0.3-1｜elf2uf2-rs のインストール：shell

```
C:¥hogehoge>cargo install elf2uf2-rs –locked
```

このコマンドの実行に失敗した場合は、「(コラム) インストールに失敗→ Rust のバージョンを落としてみる」を参照してください。

以上でソフトウェア側の準備は完了です。

0.4 電子パーツのお買い物リスト

続いて、ハードウェア側の準備を進めます。この本の中で最も複雑な回路は図 3.2-1 なので、これを作れるように電子パーツを買い集めます。以下、はんだ付けの設備等、一般的な電子工作に必要な道具は一通り揃っているものと想定します。

表 0.4-1｜本書で使用するパーツリスト

商品名	個数	大体の価格
Raspberry Pi Pico	1	770 円
BNO055 使用　9 軸センサーフュージョンモジュールキット	1	2450 円
カーボン抵抗 3.3kΩ	2	100 本で 100 円
ジャンパワイヤ	6	10 本 200 円
ブレッドボード	1	400 円
細ピンヘッダ 1×20	2	60 円
USB ケーブル（micro B）	1	150 円
積層セラミックコンデンサ 0.1μF（なくても良い）	1	30 円

上記のパーツはすべて秋月電子通商で揃えることができます。また、工作を通して USB ケーブルを何度も抜き差しすることになるため、下図のような ON-OFF スイッチ式 USB コネクタがあると便利です。

図 0.4-1｜スイッチ式 USB コネクタ

（コラム）インストールに失敗 → Rust のバージョンを落としてみる

2023 年 4 月 27 日現在、Rust の最新安定バージョンは 1.69 です。しかし、この環境下において、elf2uf2-rs のインストールに失敗するバグが報告されています。

【外部リンク】elf2uf2-rs Issue #17
https://github.com/JoNil/elf2uf2-rs/issues/17

どうやら、Rust のバージョンが 1.68 から 1.69 に上がった時の変更点にまだ対応できていないようです。このような原因でインストールに失敗してしまった場合は、以下の手順で Rust のバージョンを落としてみましょう。

現在のバージョン確認

次のコマンドで、現在の Rust のバージョンを確認します。以下のような出力が帰ってきた場合、現在のバージョンが 1.69 であることがわかります。

リスト ex0-1｜rustc のバージョン確認：shell

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

指定バージョンの Rust を追加

次のコマンドで、指定したバージョンの Rust の toolchain を追加します。ここではバージョン 1.68 を指定しました。

リスト ex0-2｜toolchain の追加：shell

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

指定バージョンが追加されたことを確認

次のコマンドで、現在インストール済みの toolchain のバージョン一覧を取得することができます。

リスト ex0-3｜toolchain 一覧の確認：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
```

上記のように表示された場合、バージョン 1.68 がインストールされていることが確認できます（末尾）。

default の変更

リスト ex0-3 の(default)の指定を見ると、現在は安定版（stable-x86_64-pc-windows-msvc; 2023 年 4 月 27 日現在でバージョン 1.69）がデフォルトで用いられていることがわかります。次のコマンドによって、デフォルトをバージョン 1.68 に変更することができます。

リスト ex0-4｜default の切り替え：shell

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

変更内容の確認

再び toolchain の一覧を確認することで、default が変更されたことが確認できます。

リスト ex0-5｜toolchain 一覧の確認：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)
```

また、最初に実行した以下のコマンドからも、Rust のバージョンが落とせていることが確認できます。

リスト ex0-6｜ 変更後の rustc のバージョン確認：shell

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

これでバージョンに基づくエラーは回避されたはずですので、再度 elf2uf2-rs をインストールしてみましょう。ただ、この時点でバージョン 1.68 に対応する開発ターゲットがインストールされていない状態になるため、一緒に再インストールします。

リスト ex0-7｜ ターゲットとツールの再インストール：shell

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

以上で安定した環境構築が完了したはずです。

第 1 章：Rust x Raspberry Pi Pico に慣れる

1.1 Rust x Raspberry Pi Pico 開発のテンプレート

組込み開発のためには、設定や用意するべきクレートが多く、この作業を行うのは、なかなか大変です。そこで、著者の Github に rp_pico_template という、開発用のテンプレートを用意しました（というか、自分用に作ってありました）ので、以下の URL から git clone コマンド等の方法でダウンロードしてください。

【外部リンク】rp_pico_template のリポジトリ
https://github.com/doraneko94/rp_pico_template

このディレクトリのプロジェクト名を書き換えることで、新しいプロジェクト開発をすぐにはじめることができます。ここでは、以下の 2 ステップでプロジェクト名を rp_bno055 に変更しましょう。

1. ディレクトリの名前を rp_bno055 に変更する
2. Cargo.toml を開き、[package]の name の値を”rp_bno055”に書き換える

リスト 1.1-1｜パッケージ名の書き換え：Cargo.toml

```
[package]
name = "rp_bno055" # <- もともと、”rp_pico_template”になっている
version = "0.1.0"
...
```

これで準備は完了です。

1.2 Raspberry Pi Pico の内蔵 LED を点滅させる（L チカ）

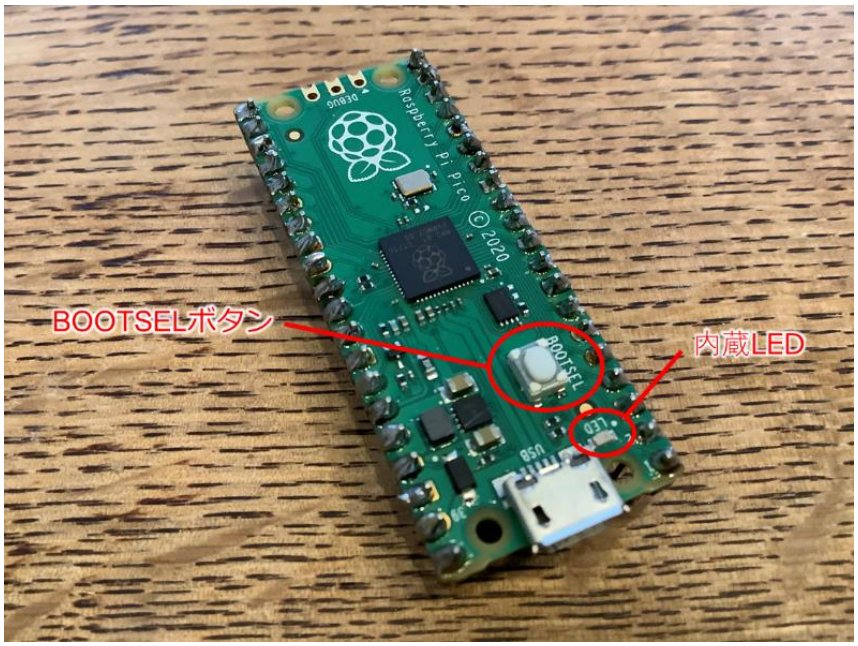
新しいプログラミング言語に触れたとき、大抵は「Hello, world!」の出力するコードを書くことからはじめます。組込み開発の場合は、LED をチカチカ点滅させる、通称「L チカ」という作業からスタートするのが王道です。Raspberry Pi Pico のボードには内蔵 LED が搭

載されているので、回路を組む必要すらありません。さっそくやってみましょう。

…とは言ったものの、実は rp_pico_template の src ディレクトリにはすでに led.rs というファイルが含まれており、これで L チカができるようになっています。そのため、何も新たに書く必要はなく、実行さえすれば入門編は終わりです。以下の手順にしたがって操作してください。

1. コマンドプロンプトで、プロジェクトフォルダ（rp_bno055）に移動
2. BOOTSEL ボタン（図 1.2-1 参照）を押しながら、Raspberry Pi Pico を PC の USB ポートに接続（Raspberry Pi Pico が認識されたことを確認してください）
3. コマンドプロンプトで、cargo run --release --bin led を実行

図 1.2-1｜Raspberry Pi Pico の内蔵 LED と BOOTSEL ボタン



リスト 1.2-1｜led.rs の実行：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
```

正常に実行されると、リスト 1.2-1 のようなメッセージが表示され、Raspberry Pi Pico 上の LED が約 1 秒周期で点滅します。
よくある失敗例としては、手順 2 の BOOTSEL ボタンを正しく押せていないケースが考えられます。このボタンを押しながら差すこと（差してから押すのはダメ）で、Raspberry Pi Pico にそれまで書き込まれていたプログラムが消去され、書き込みモードとして使用することが可能になります。ここで失敗して Raspberry Pi Pico が正常に認識されていない場合には、以下のようなエラーメッセージが出力されているはずです。

リスト 1.2-2｜BOOTSEL ボタンが正しく押せていない場合：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¥thumbv6m-none-eabi¥release¥led` (exit code: 1)
```

なお、BOOTSEL ボタンを押さずに Raspberry Pi Pico を PC に接続した場合や、VSYS ピンから電源を供給した場合は、それ以前に Raspberry Pi Pico に書き込まれていた最新のプログラムが実行されます。

1.3 L チカのコード解説

1 行もコードを書かずに入門編終了では味気ないので、led.rs の中身を簡単に解説していき

ましょう。Rust による組込み開発のコードは、main 関数の内部とそれ以前の大きく 2 つに分けることができ、さらに main 関数の中は loop の内部とそれ以前に分けて考えることができます。

main 関数以前

リスト 1.3-1｜main 関数以前の記述：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
```

main 関数の前には、①環境属性、②使用するクレート、③定数、などを記述します。①の例として、#![no_std]の記述は std クレートを使用しないことを宣言しています。これにより、このコードが OS を持たない環境(ベアメタル環境)でも実行できるようになりますが、std クレートに実装されている Vec 型や println!マクロなど、便利な機能が利用できなくなるというデメリットも生じます（そのため、この後も、異なる方法で類似の機能を実装することを考えます)。また、②の例として、embedded_hal::digital::v2::OutputPin を使用しています。これは、内蔵 LED を点灯させるために、ピンを電流の出力元として使用する際に必要になるトレイトです。

main 関数内部（loop 以前）

リスト 1.3-2｜main 関数内部、loop 以前の記述：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_clock.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();
```

ここでは、Raspberry Pi Pico 上の周辺機器のうち使用するものを宣言したり、モードを設定したりします。周辺機器とは、マイコンボード上に用意されたタイマーやピンのことを指します。このコードで実行したいことは、内部 LED を点滅させるためのピンを使用可能にすることと、「500 ミリ秒待つ」といった Delay 機能を使えるようにすることです。
それぞれ、②のブロックが Delay、③のブロックが内部 LED を設定しています。それにともない、関数の引数で必要になるものを、ドミノ式で事前に宣言しています。ここでは、以下のような依存関係があります。

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

そして、これらの最上位にくるのが pac と core です。そのため、ほとんどのケースにおいて①は忘れずに記述する必要があります。
③について、内部 LED を点滅させるピンを変数名 led_pin で取得し、mut 変数として状態を変更（点灯・消灯）できるようにしています。その際、内部 LED を点滅させるピンには Raspberry Pi Pico ボード上で GPIO25 という名前が付けられているため、すべてのピン情

報を格納した変数 pins から gpio25 を取得し、into_push_pull_output メソッドにより、出力モードに設定しています。

main 関数内部（loop 内部）

リスト 1.3-3｜main 関数内部、loop 内部の記述：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);
}
```

この記述を見るとわかるとおり、この loop は永久に繰り返されます。これは、組込みのプログラムが何らかの処理をして終了するのではなく、電源が切れるまで動き続けることを前提としているためです。そのため、fn main() -> !と、main 関数の出力に「!」が指定され、このメイン関数は終了しない（終了コードを返さない）ことを宣言しています。
src/led.rs では、

1. 出力ピンの電圧を High にして内部 LED を点灯（set_high）
2. 500 ミリ秒待つ（delay_ms(500)）
3. 出力ピンの電圧を Low にして内部 LED を消灯（set_low）
4. 500 ミリ秒待つ（delay_ms(500)）

の処理を繰り返すことで、1 秒周期で内部 LED を点滅させています。

（コラム）正確に 1 秒周期で L チカする

リスト 1.3-3 のコメントにも書かれているとおり、Delay による待機時間は、実は結構ラフなのです。そのため、正確に時間を測りたい場合には delay_ms メソッドなどの関数の使用を避けるべきです。
より正確に時間を計測するためには、Timer を使用します。Timer を使用して src/led.rs を書き換えた src/led_timer.rs を用意しましたので、著者の Github リポジトリのコードを参照してください。

【外部リンク】src/led_timer.rs サンプルコード
https://github.com/doraneko94/rp_bno055/blob/master/src/led_timer.rs

リスト ex1-1｜Timer を用いた停止時間の実装：src/led_timer.rs (Github)

```
...
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'
}
```

①で定義した Timer 構造体は、カウンタ（ticks）の値が 1 マイクロ秒に 1 つ増加します。そのため、時間を測り始める時点のカウンタの値を保存し（②）、その後、現在とスタート時点のカウンタの差が 500,000（0.5×10^6 マイクロ秒＝0.5 秒）となるまで、while ループで時間をつぶしています（③）。

第 2 章：シリアル通信

2.1 シリアル通信で Hello, world!

前章で組込みプログラミングの入門編は終了ですが、一応、Hello, world!したい人のために、文字が表示される入門コードも用意しました。Raspberry Pi Pico と USB ケーブルで接続された PC との間でシリアル通信を行い、PC の画面に Hello, world!の文字を表示します。サンプルコードは

【外部リンク】src/serial_hello.rs サンプルコード
https://github.com/doraneko94/rp_bno055/blob/master/src/serial_hello.rs

を参照してください。

新規ファイルの作成

シリアル通信による Hello, world!を実行するため、src ディレクトリに serial_hello.rs という名の新しいファイルを作成します。また、Cargo.toml の末尾に[[bin]]ブロックを追加することで、このファイルをバイナリ（実行ファイル）として登録します。これにより、cargo run コマンドからの実行が可能になります。

リスト 2.1-1 | [[bin]]ブロックの追記：Cargo.toml

```
[[bin]] # ファイルの末尾
name = "serial_hello"
path = "src/serial_hello.rs"
```

src/serial_hello.rs の内容は、サンプルコードを見ながら記述してください。以下、サンプルコードの特徴的な部分について解説を加えます。

クレートの導入

通常、src/serial_hello.rs の記述を始める前に、シリアル通信を行うために必要なクレートを Cargo.toml に導入する必要があります（ただし、rp_pico_template には既に導入されているので、参考程度に読み流してください）。追加するのは

- usb_device：UsbDevice インスタンスや UsbClass、UsbBus を実装し、これらを組み合わせて USB 複合デバイスを形成するクレート
- usb_hid：USB HID（Human Interface Device）クラスの実装
- usb_serial：CDC-ACM（Communication Device Class Abstract Control Model） USB シリアルポートの実装

の 3 つです。

サンプルコード上では、usb_device と usb_serial から構造体やトレイトを使用することを宣言しています。

リスト 2.1-2｜使用するクレートの宣言：src/serial_hello.rs

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

必要な構造体の定義

USB 経由の通信のために必要なのは、

- USB Bus
- Serial Port
- USB Device

の 3 つです。これらを 1 つずつ実装していきます。

リスト 2.1-3｜UsbBus の構築：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,
```

```
));
```

usb_device の UsbBusAllocator を通して、rp2040_hal の UsbBus を構築します。

リスト 2.1-4｜SerialPort の構築：src/serial_hello.rs

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

先程構築した UsbBus を使用し、SerialPort を構築します。

リスト 2.1-5｜UsbDevice の構築：src/serial_hello.rs

```
let mut usb_dev = UsbDeviceBuilder::new(&usb_bus, UsbVidPid(0x16c0, 0x27dd))
    .manufacturer("Fake company")
    .product("Serial port")
    .serial_number("TEST")
    .device_class(2)
    .build();
```

こちらも同様に UsbBus を使用して構築します。その際、製造者名（manufacturer）やプロダクト名（product）、シリアル番号（serial_number）等を定義することができますが、ここではそれぞれにダミーの名前を与えています。デバイスクラス（device_class）は Usb.org で定義された番号で、ここでは 0x02（Communications and CDC Control）を指定します。クラスコードの詳細は以下のページを参照してください。

【外部リンク】USB.org｜Defined Class Codes
https://www.usb.org/defined-class-codes

文字列の出力

必要な構造体を定義したら、loop に処理を記述していきます。

リスト 2.1-6｜loop 内の処理：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
}
```

このコードでは、1 秒周期で Hello, world!を出力します。
ここで、②が Hello, world!を書き込む記述です。このとき、改行コード（¥r¥n）も忘れずに付与し、クオーテーションの前に b を付けてバイト列として write メソッドに渡します。これにより、SerialPort が「Hello, world!¥r¥n」という文字列を書き込む準備ができた状態になります。
①は引数に与えた機能に新しいイベントが発生したかどうかを調査し、実行可能な処理を遂行する関数です。ここでは SerialPort の更新の有無を調べ、書き込まれた文字列が存在する場合は、それを出力します。このとき、引数の機能に更新がある場合は true、ない場合は false を返しますが、ここではその返り値を使用していません。
また、注意として、この関数はシリアル通信を維持するために、少なくとも 10 ミリ秒に 1 回実行される必要があります（それ以上長い時間実行されない場合、シリアル通信が途絶します）。すなわち、1 秒待つ処理を実行するために delay.delay_ms(1000)などとして、処理を 10 ミリ秒以上停止させることができません。そこで、5 ミリ秒×200 回=1 秒であることを利用して、5 ミリ秒待つ→usb_dev.poll()という処理を 200 回繰り返して 1 秒間を作り出しています。

コードの実行

最後に、完成したプログラムを実行してみましょう。すると、以下のような出力が 1 秒周期で表示されるはずです。

リスト 2.1-7｜Hello, world!の出力：shell

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

```
...
```

2.2 数値の出力

（この節の流れを簡単にまとめると、「シリアル通信で数値を出力するのは結構面倒です。そこで、便利なクレートを作りました」というものです。本書では、以降そのクレートを使用して数値を出力するため、技術的な背景に興味がない場合は、内容を読み飛ばし、本節末尾のクレートの追加作業のみ行ってください）

numtoa クレートを使用した整数の出力

前節では「Hello, world!」という文字列を出力しました。本書ではこの後、BNO055 から読み取った値を確認する際に、シリアル通信が活躍します。そのため、文字列ではなく数値を出力する必要があります。serial.write メソッドの引数に渡すため、数値をバイト列に変換する際には、no_std 環境では numtoa クレートを使用します。

numtoa クレートは rp_2040_template に標準で含まれてはいないため、後のサンプルコードを使用する場合は Cargo.toml を開いて、[dependencies]に以下のように追加してください。

リスト 2.2-1｜numtoa クレートの追加：Cargo.toml

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

numtoa クレートを用いたサンプルコードは

【外部リンク】src/serial_hello_numtoa.rs サンプルコード
https://github.com/doraneko94/rp_bno055/blob/master/src/serial_hello_numtoa.rs

を参考にしてください。このコードでは、1 秒周期で Hello, world!を出力するとともに、それまでに Hello, world!を出力した回数も表示します。

numtoa クレートで数値を出力するための基本的な記法は以下のとおりです。

リスト 2.2-2 | numtoa クレートによる数値出力：src/serial_hello_numtoa.rs（Github）

```
...
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
    }
}
```

まず、数値のバイト列を格納するためのバッファが必要になります（①）。次に、文字列部分をまとめて出力します（②）。数値を出力する際、numtoa::NumToA トレイトは、整数型に numtoa メソッドを付与します。引数には基数（何進数で表現するか）と、①で確保したバッファを与えます（③）。最後に、改行コードを出力します（④）。それまでの serial.write メソッドでは改行コードを出力していないので、「Hello, world! x（回数）」までが 1 行に表示されることになります（そして、この改行コードを忘れると、1 行が終了しないため何も表示されません）。

これを実行すると、以下のような出力が得られます。

リスト 2.2-3｜Hello, world! x (回数)の出力：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
...
```

（サンプルコードを実行する場合は、リスト 2.2-1 同様、Cargo.toml に[[bin]]ブロックを追加しておく）

浮動小数点数の出力

なお、numtoa クレートで浮動小数点数（小数）を変換することはできないため、例えば f32 型の 123.4567 を小数点以下 2 桁まで表示する場合、次のような工夫が必要になります。

1. 123.4567 を i32 型に変換して、整数部分を抜き出す（123）
2. 整数部分を出力する
3. 小数点を文字列「.」として出力する
4. f32 型に再変換した整数部分（123.0）を、元の数字から引き小数点以下部分を抜き出す（0.4567）
5. 小数点以下部分を 100 倍する（45.67）
6. i32 型に変換して出力する（45）

ただし上記の手順では、例えば 123.0456 が「123.4」と表示されてしまうという問題等があり、ゼロ埋めなどの細かな調整を行う必要があります。毎回この処理を記述するのは面倒で、コードの可読性も損なわれます。そこで著者は serial_write というクレートを作り、上記の処理を 1 行で書けるようにしました。

【外部リンク】serial_write クレートの crates.io ページ
https://crates.io/crates/serial_write

【外部リンク】serial_write クレートの Github リポジトリ
https://github.com/doraneko94/serial_write

このクレートは浮動小数点数をそのまま表示するだけではなく、文字の数を揃えて見やすくできる「1.23e002」のような指数形式の表示法や、配列の表示も可能です。

先程と同じ機能を果たすコードを、serial_write クレートを使用して書いたサンプルが以下のリンクから取得できます。

【外部リンク】src/serial_hello_count.rs サンプルコード
https://github.com/doraneko94/rp_bno055/blob/master/src/serial_hello_count.rs

リスト 2.2-4 | serial_write クレートによる数値出力：src/serial_hello_count.rs（Github）

```
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
    }
}
```

です。serial_write クレートではまず Writer 構造体（①）を構築し、出力するものの型に合わせた関数を使用して文字列や数値、配列を表示します。また、「write」から始まる関数を使用すると改行なし（②）、「writeln」から始まる関数を使用すると改行コードを末尾に付与します（③）。

本書では、これ以降、シリアル通信を介した出力には serial_write クレートを使用しますので、Cargo.toml に以下のとおり追加してください。

リスト 2.2-5｜serial_write クレートの追加：Cargo.toml

```
[dependencies]
cortex-m = "0.7.4"
…
panic-halt = "0.2.0"
numtoa = "0.2.4" # numtoa を使用したサンプルを実行しないなら、なくても良い
serial_write = “0.1.0” #  追加
```

（コラム）シリアル通信の開始

0.3 節でインストールした elf2uf2-rs は、Rust で書いたコードを Raspberry Pi Pico に書き込むためのツールです。これについて、.cargo/config では、以下のように設定しています。

リスト ex2-1 | elf2uf2-rs の設定：.cargo/config

```
...
runner = "elf2uf2-rs -d -s"
```

ここで、2 種類のオプションが与えられています。

- -d: マウントされた Raspberry Pi Pico に自動的にデプロイする
- -s: デプロイ後に Raspberry Pi Pico をシリアルデバイスとして開き、シリアル出力をプリントする

そのため、cargo run コマンドによりプログラムの書き込み・実行を行った後、シリアル通信を開始し、通信が生きている（10 ミリ秒以内に usb_dev.poll メソッドが呼ばれ続けている）限り、通信内容の表示を行います。
-s オプションが与えられているときの処理を、elf2uf2-rs の実装から確認してみましょう。

elf2uf2-rs の処理

リスト ex2-2 | elf2uf2-rs におけるシリアル通信：elf2uf2-rs/main.rs (Github)

```
...
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::TimedOut => (),
                        Err(e) => return Err(e.into()),
                    }
```

```
                        }
                    }
                }

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

全体の流れは、

1. 事前準備（①）
2. Raspberry Pi Pico にプログラムをデプロイ（②）
3. 通信ポートの取得（③～⑤）
4. 通信の開始・継続（⑥～⑨）

から成ります。1 つずつ見ていきましょう。
まず、プログラムのデプロイが実行される前に、現在利用可能な通信ポート一覧を取得しておきます（①）。②のブロックは elf2uf2-rs のメインとなる処理で、Raspberry Pi Pico にプログラムをデプロイするための処理が行われています。これにより、プログラムにシリアル通信を行う処理が書かれている場合、新たにポートが 1 つ開通します。

③はプログラムのオプションに-s が設定されている場合の分岐で、以下、シリアル通信により Raspberry Pi Pico から送られてくる情報を表示するための処理が行われます。
まず、デプロイ後に利用可能な通信ポート一覧を再取得し、それらを 1 つずつ検証します（④）。そして、とあるポートが①で取得した一覧に含まれていなかった場合、そのポートはデプロイによって新規作成されたポートであり、Raspberry Pi Pico が情報を送ってくる窓口であると判断することができます（⑤）。この処理は 200 ミリ秒間隔で 10 回まで実行され、その間に新規作成されたポートが取得できない（Raspberry Pi Pico に書き込んだプログラムにシリアル通信の処理が含まれていない）場合、elf2uf2-rs の処理を終了します（⑥）。

⑤で新規作成されたポートが取得できていた場合（⑥）は、そのポートに対して通信を確立

します（⑦）。通信確立処理も 200 ミリ秒間隔で 5 回まで試してみて、成功した場合は、通信内容を受け取るバッファを作成し（⑧）、そこに情報を読み取り続ける無限ループに入ります（⑨）。
以上が elf2uf2-rs によるデプロイと同時に、シリアル通信の内容がシェルに表示される理由です。

elf2uf2-rs に頼らずにシリアル通信を読み取る

BOOTSEL ボタンを押さずに Raspberry Pi Pico を PC に接続した場合、Raspberry Pi Pico は書き込みモードに入らず、ただ PC から電源が供給されただけと判断されるため、それ以前に書き込まれたプログラムの内容が実行されます。ただしこの時、Raspberry Pi Pico に書き込まれたプログラムがシリアル通信を行う処理を有していても、どの画面にも何も表示されません。これは、elf2uf2-rs がデプロイと同時に実行したような、シリアル通信の受け手側の処理が行われていないためです。
そこで、Raspberry Pi Pico が差し込まれたことにより開設されたシリアル通信のポートを、事後的に読み取るソフトウェアを作成しました。詳細は

【外部リンク】oscillo_serial Github リポジトリ
https://github.com/doraneko94/oscillo_serial

を参照してください。このソフトウェアには、text と plot の 2 つのモードがありますが、ここでは text のみ取り扱います（plot の機能は「（コラム）出力情報のグラフ表示」で説明します）。まず、上記のソフトウェアを git clone コマンド等を用いて、手元で実行可能な状態にしてください。

リスト ex2-3｜oscillo_serial の入手：shell

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

次に、シリアル通信の処理を含むプログラム（例えば、serial_hello.rs）を書き込んだ Raspberry Pi Pico を、BOOTSEL ボタンを押さずに PC に USB 接続します。そして、oscillo_serial のディレクトリに移動して、以下のコマンドでソフトウェアを実行します。

リスト ex2-4｜oscillo_serial の実行：shell

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

すると、リスト 2.1-7 と同様の出力が確認できるはずです。

oscillo_serial の実装

上記の機能を実装している oscillo_serial のコードを見てみましょう。

リスト ex2-5｜ コード全景：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：リスト ex2-6 参照 }
            _ => {}
        }
    };
    Ok(())
}
```

①で各種パラメータを取得したあと、②でシリアル通信を確立します。ここでは elf2uf2-rs でやっているのと違い、利用可能なポートに片っ端から通信を持ちかけます。その結果、既に通信が確立しているポートからは通信を拒否されてエラーが返るため、最終的には、まだ受け手がいない Raspberry Pi Pico のポートに行き着くことになります。その後、通信用のバッファを用意して、①で読み取ったモード（-mo）の指定が text である場合、③の処理に移行します。インデントが多く見にくくなるため、③の内容をリスト ex2-6 に別途表示します。

リスト ex2-6｜ リスト ex2-5-③の拡大：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()),
    }
}
```

とはいえ、ここでやっているのは、エラーハンドリングと文字列の表示のみの単純な処理です。oscillo_serial では elf2uf2-rs とは異なり、io::stdout 関数ではなく print!マクロを使用し、バッファの内容を String 型に変換して出力しています。

第 3 章：BNO055 から姿勢情報を読み取る

3.1 BNO055 の概要

さて、ここからがこの本の主題です。本章では、Raspberry Pi Pico を用いて IMU（慣性計測装置：Inertial Measurement Unit）の出力を読み取ります。そして使用する IMU には使い勝手や入手性を考慮し、ここでは BNO055 を使用します。

【外部リンク】BOSCH 社の製品ページ
https://www.bosch-sensortec.com/products/smart-sensors/bno055/

BNO055 は加速度センサ（3 軸）、ジャイロセンサ（3 軸）、地磁気センサ（3 軸）の合計 9 軸のセンサとオリエンテーションソフトウェアをひと纏めにしたパッケージです。それぞれのセンサが検出した値を個別に出力するのはもちろん、複数のセンサの出力を統合し、現在のセンサの姿勢（回転）角を自動で計算して出力してくれます。出力の通信インターフェースには I^2C と UART が利用可能ですが、ここでは I^2C を使用します。

各センサから姿勢角を計算する原理について「（コラム）センサ情報による姿勢角の計算原理」にて簡単に説明していますが、詳細な解説は[野波 2020]や[野波ほか 2022]を参照してください。
また、BNO055 が姿勢角を計算するためには適したモードに設定する必要があり、モードによって、出力される角度が相対的であったり絶対的であったりします。BNO055 のモードについては「（コラム）BNO055 の動作モード」または、

【外部リンク】BNO055 datasheet [BOSCH 2021]
https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf

を参照してください。
上記のデータシートの重要な部分は、本書でも和訳しながら引用していきます。しかし、本書ではカバーできない BNO055 の魅力についても詳細が記載されているため、ぜひ一読することを勧めます。また、本書では BNO055 をブレッドボード上で使用するために、秋月

電子がモジュール化した

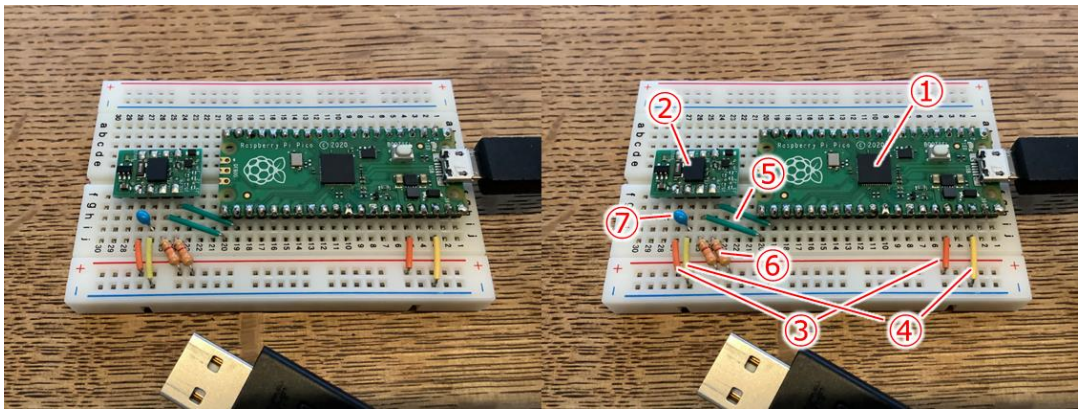
【外部リンク】ＢＮＯ０５５使用　９軸センサーフュージョンモジュールキット
https://akizukidenshi.com/catalog/g/gK-16996/

を使用しますので、このモジュールの取扱説明書[秋月電子通商 2022]も非常に参考になります。

3.2 回路の作成

「0.4 電子パーツのお買い物リスト」で購入したパーツを組み合わせ、以下のような回路を作ります。

図 3.2-1｜BNO055 の制御回路



①：Raspberry Pi Pico

ブレッドボードの 1～20 番の中央に刺します。

②：BNO055 モジュール

ブレッドボードの 24～27 番の中央に刺します。

③：3.3V 電源供給

Raspberry Pi Pico の 3V3(OUT)ピン（ブレッドボードの 5 番：後にある図 3.2-2 も参照）とブレッドボードの+行を繋いで、+行を 3.3V にします。その後、+行と BNO055 の VIN を接続して電源を供給します。

④：GND

Raspberry Pi Pico の GND ピン（ブレットボードの 3 番）とブレットボードの－行を繋いで、－行を 0V にします。その後、－行と BNO055 の GND を接続して接地します。

⑤：I2C 通信用ワイヤ

図 3.2-2 を見ると、(他にも該当するピンがありますが）GPIO16 と GPIO17 のピンがそれぞれ「I2C0 SDA」と「I2C SCL」の機能を有していることがわかります。よって、BNO055 の対応するピンに接続します。すなわち、GPIO16（ブレットボードの 20 番）を BNO055 の SDA/T ピン、GPIO17（ブレットボードの 19 番）を BNO055 の SCL/R ピンに接続します（T, R はそれぞれ、UART で接続するときのピンの名称です)。この 2 本のワイヤを介して、Raspberry Pi Pico と BNO055 の間で I^2C 通信を行います。

⑥：プルアップ抵抗（3.3kΩ）

I2C 通信を安定させるために、BNO055 の SDA/T ピン・SCL/R ピンとブレットボードの +行を抵抗で繋ぎます。この抵抗をプルアップ抵抗と呼びます。秋月電子通商の情報

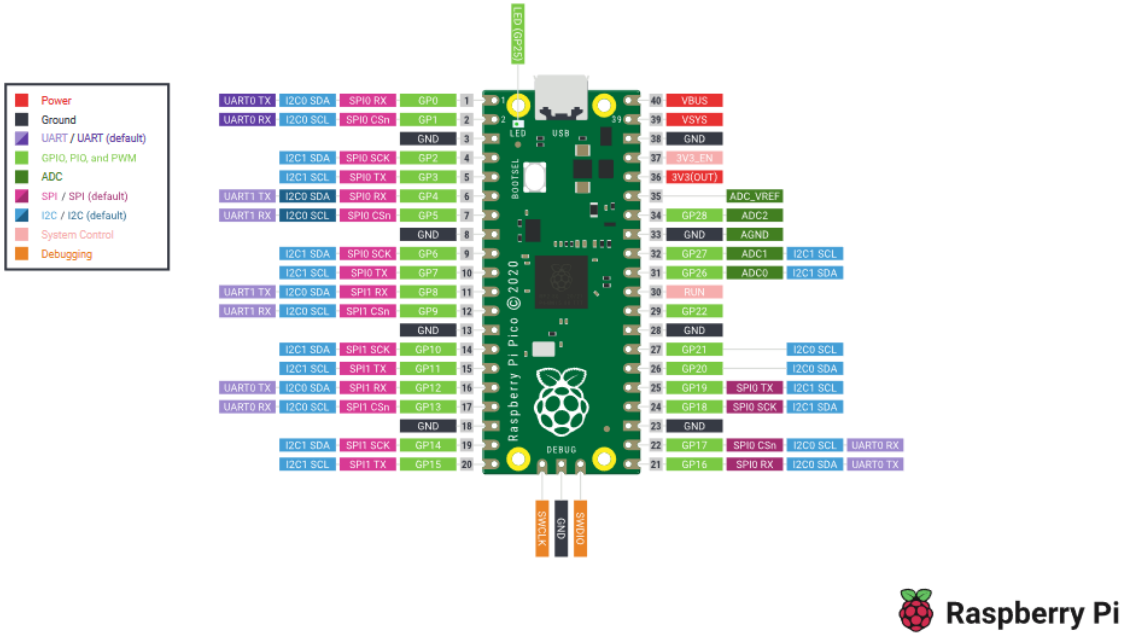
【外部リンク】ＢＮＯ０５５使用　９軸センサーフュージョンモジュールキットの質問と回答
https://akizukidenshi.com/catalog/faq/goodsfaq.aspx?class1=K&code=16996

によると、マイコンの内部弱プルアップ（数十 k～100kΩ）と比較して、1k～10kΩが安定すると推奨されているため、ここでは 3.3kΩを使用します。

⑦：バイパスコンデンサ

電源と GND の間のノイズ除去のために 0.1μF のコンデンサを配置します。これをバイパスコンデンサと言います。回路図では、BNO055 の VIN と GND ピンを繋いでいます。

図 3.2-2 | Raspberry Pi Pico のピンアサイン（raspberrypi.com より引用）



3.3 サンプルコードの実行

BNO055 を使用するためには必要な処理が複数あり、段階的にコードを書いていくのはむしろ分かりにくくなるため、ここでは先にコードの全景を提示して動作を確認し、その後、ブロックごとに解説していきます。

基本コードの動作確認

まず、rp_bno055 の src ディレクトリに imu_base.rs という名前のファイルを作成し、Cargo.toml にこれをバイナリファイルとして登録してください。

リスト 3.3-1 | [[bin]]ブロックの追記：Cargo.toml

```
[[bin]] # ファイルの末尾
name = "imu_base"
path = "src/imu_base.rs"
```

次に、必要なクレートを Cargo.toml に追加します。[dependencies]の末尾に、以下の 2 つ

のクレートについて追記してください。

リスト 3.3-2 | [dependencies]の追記：Cargo.toml

```
[dependencies]
…
fugit = “0.3.6”
bno055 = { git = "https://github.com/eupn/bno055", rev =
"58970ced667deddd09933954b1a928587226daf7" }
```

fugit は I^2C の周波数を設定する際に使う補助クレートです。そして、bno055 は BNO055 を動かすための本命のクレートです。ただし、このクレートの最新リリースは未修復のバグが多く、最新のコードと大きな乖離があるので、Github リポジトリから直接引っ張ってくることにします。その際、今後のアップデートによって破壊的な変更が生じることを避けるため、安定動作が見込まれる時点のコードを指定します。ここでは、bno055 のクレートパラメータを用いて、git でリポジトリの URL を指定し、rev で指定したコミット ID 時点のコードを使用する旨の記述を行っています[rust-lang B]。

そして、サンプルコード

【外部リンク】src/imu_base.rs サンプルコード
https://github.com/doraneko94/rp_bno055/blob/master/src/imu_base.rs

の内容を、src/imu_base.rs にコピーしてください。
その後、前節でパーツを配置したブレッドボード上の Raspberry Pi Pico を、BOOTSEL ボタンを押しながら PC に USB 接続し、src/imu_base.rs のプログラムを書き込み、実行します。

リスト 3.3-3 | src/imu_base.rs の実行：shell

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

書き込みが正常に終了すると、以下のような出力が得られます。

リスト 3.3-4 | src/imu_base.rs の較正情報出力：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)
...
```

ここでは、BNO055 から IMU としての出力を読み取る前に、3 種類のセンサの較正（キャリブレーション）を行っています。BNO055 のデータシート[BOSCH 2021]（と著者の経験則）によると、それぞれのセンサの較正は以下のような手順で行います。

加速度センサ（Accelerometer）の較正

安定した 6 種類以上の位置に、それぞれ 2 秒以上静置する。このとき、少なくとも 1 回ずつ重力が X, Y, Z 軸の正および負の方向を向くように置く。すなわち、以下のような置き方で 6 回向きを変える。（注意！：向きの変更は、ゆっくりと行う。）

1. ブレッドボードを普通に（両面テープを床向きにして）2 秒置く
2. ブレッドボードをひっくり返し、両面テープが天井を向くように 2 秒保持する
3. ブレッドボードを横に傾け、側面の長辺の一方を床につけて 2 秒置く
4. ブレッドボードを 3.の反対側に傾け、もう一方の長辺を床につけて 2 秒置く
5. ブレッドボードを縦に傾け、側面の短辺の一方を床につけて 2 秒置く
6. ブレッドボードを 5.の反対側に傾け、もう一方の短辺を床につけて 2 秒置く

すなわち、ブレッドボードの 6 面を 1 回ずつ下にして置いて、2 秒ずつ安定させます。

ジャイロセンサ（Gyroscope）の較正

ブレッドボードを任意の安定した状態で数秒間放置する。

地磁気センサ（Magnetometer）の較正

ブレッドボードを安定した状態で放置しておけば自然と較正される。周囲に磁場の発生源がないことに注意する。なかなか較正が終了しない場合は、ブレッドボードを置く場所を変えてみる。

較正の進行度は、以下の 4 つの項目について 0（未完了）〜3（較正完了）の 4 段階で評価されます。項目と較正対象の関係は以下の通りです。

表 3.3-1｜較正情報の内容一覧

項目（記号）	内容
SYS	全体
ACC	加速度センサ
GYR	ジャイロセンサ
MAG	地磁気センサ

つまり、リスト 3.3-4 の出力が、すべての項目について「3」になれば較正は完了です。較正手順の煩雑さから予想できる通り、ACC の較正に最も苦労すると思いますので、ゆっくりと向きを変えながら、6 種類の向きのうち、あまり試していなさそうなものを順次試していってください（それなりに時間がかかります）。後にこの作業をスキップする実装を 3.5 節で導入しますので、今しばらくお付き合いください。

姿勢角の表示

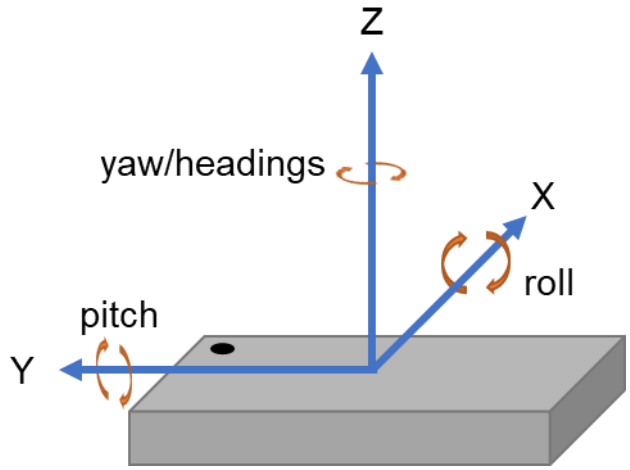
較正が完了すると、BNO055 の読み取りを開始し、シリアル通信を通して以下のように姿勢角の情報が表示されます。

リスト 3.3-5｜src/imu_base.rs の姿勢角情報出力：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
...
```

ここでは、ブレッドボードの（正確には BNO055 の）姿勢情報を 100 ミリ秒周期で表示しています。姿勢の表し方にはオイラー角を用い、単位は度（degree）です。オイラー角の構成要素であるピッチ（pitch）・ロール（roll）・ヨー（yaw/heading）と、BNO055 の XYZ 軸の関係は、以下の通りです。

図 3.3-1｜ オイラー角と XYZ 軸の関係



しかし、実はこの対応関係は、BNO055 の公式情報とは異なります。このことに関し、公式・bno055 クレート・オイラー角を保存する構造体のクレート・著者の思惑が複雑に交差している現状があるため、その詳細を「(コラム) BNO055 におけるオイラー角の取り扱い」で解説します。

姿勢角はリアルタイムに更新されるため、ブレッドボードの向きを変えると、出力は以下のように変化します。

リスト 3.3-6｜Roll 方向に回転：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
...
```

ここでは roll 方向（ブレッドボードの長軸方向）に回転を加えています。

3.4 基本コードの実装

ここからは、src/imu_base.rs の内容を解説します。まず、コードの全体から、特徴的な部分を提示します。

リスト 3.4-1｜BNO055 を使用する基本コード：src/imu_base.rs

```
...
use fugit::RateExtU32; // 補助クレートの導入
...
#[rp2040_hal::entry]
fn main() -> ! {
    ...
    let sda_pin = pins.gpio16.into_mode::<hal::gpio::FunctionI2C>(); // ①：ここから
    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：ここまで
```

```
    ...

    // Wait 1 sec after power supply to BNO055.
    for _ in 0..200 { // ②：ここから
        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：
ここまで

    let mut count = 0usize; // ③：ここから
    '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：ここまで
```

```
    loop { // ④：ここから
        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：ここまで
}
```

全体の処理は、①I²C 通信の設定、②BNO055 の初期化、③センサの較正、④姿勢情報の出力、の 4 段階で構成されています。以下、個別に詳しく見ていきます。

I2C 通信の設定

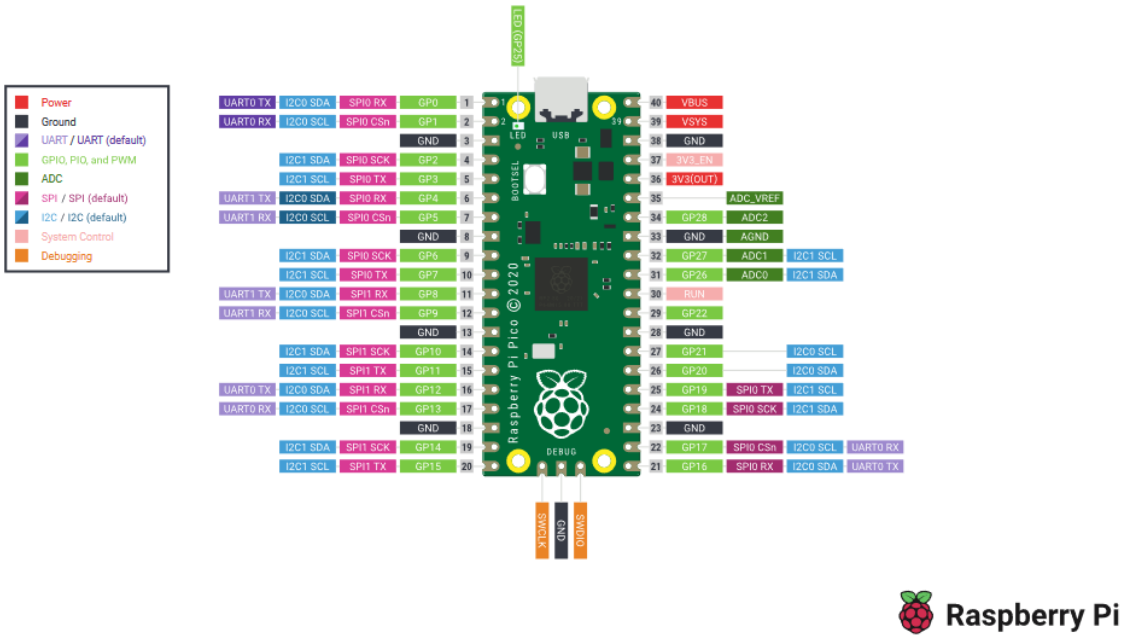
リスト 3.4-2｜I2C 通信の設定：src/imu_base.rs

```
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 クレートの活用
    &mut pac.RESETS,
    &clocks.system_clock,
); // ②
```

I²C は SCL（クロック生成）と SDA（データ送受信）の 2 本の線を用いて行う通信方法です。①では、これらの役割を担う Raspberry Pi Pico のピンを 2 本（GPIO16, GPIO17）取得しています。以下に Raspberry Pi Pico のピンアサインを再掲します。

図 3.2-2 | Raspberry Pi Pico のピンアサイン（raspberrypi.com より引用：再掲）



21 番ピン（GPIO16：ブレッドボード上では 20 番）が「I2C0 SDA」、22 番ピン（GPIO17：ブレッドボード上では 19 番）が「I2C0 SCL」の機能を有していることがわかるため、それぞれに対応した機能を割り当てています。そして、これらのピンは、Raspberry Pi Pico が持つ 2 個の I2C ブロック（I2C0, I2C1）のうち、I2C0 に対応しているため、②にて I2C0 を初期化しています。この時、400 という u32 型の整数を 400kHz という周波数の型に変換するために、fugit クレートの RateExtU32 トレイトを使用しています。

BNO055 の初期化

リスト 3.4-3 | Bno055 構造体の初期化：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
```

ここでは、BNO055 を使用するために、Bno055 構造体を初期化していきます。具体的には、I²C 通信を通して適切なアドレスに設定を書き込んでいくわけですが、この書き込み動作が行えるのは、BNO055 に電源が供給されてしばらく時間が経ち、動作が安定した後になります。そのため、まず①でシリアル通信を維持したまま、5 ミリ秒×200=1 秒間待機します。

実は、この待機時間がなくても cargo run コマンドによる書き込みと実行は成功し、正しく BNO055 を使用することができます。しかし、その後（BOOTSEL ボタンを押さずに）給電のみを行った場合、BNO055 の初期化に失敗します。その理由は、この 2 つの操作に以下のような違いが生じるためです。

BOOTSEL ボタンを押して接続し、cargo run で新規書き込みを行って実行した時

1. USB 接続と同時に BNO055 に給電が開始される
2. 書き込みモードなので、プログラムは何も実行されない
3. 画面を切り替えたり、コマンドを打ち込んだり、モタモタしたりして 1 秒くらい経つ
4. プログラムが書き込まれ、I²C 通信が開始される

BOOTSEL ボタンを押さずに接続し、給電のみを行った場合

1. USB 接続と同時に BNO055 に給電が開始される
2. それと同時に書き込み済みのプログラムが実行され、I²C 通信が開始される
3. BNO055 の動作がまだ安定しておらず、アクセスに失敗する

そういう理由で、①の待機時間が必要になります。著者はこれに気付かずに 2 日くらい無駄にしました。

BNO055 の動作が安定した後、Bno055 構造体を作ります（②）。ただしこの時に with_alternative_address メソッドを呼び出す必要があります。このメソッドを呼び出すと、Bno055 構造体の I²C デバイスアドレスには 0x28 が指定され、呼び出さなかった場合はデフォルトで 0x29 になります。今回使用するモジュールの説明書[秋月電子通商 2022]によると、モジュールでは BNO055 の COM3 ピンが LOW に設定されており、アドレスが 0x28 の状態で出荷されているので、ここではこのようにしています。その後、この構造体を初期化し（③）、動作モードを指定します（④）。ここでは NDOF という、いわゆる「全部盛り」のモードを指定しています。BNO055 の動作モードとできること・できないことの関係については、「（コラム）BNO055 の動作モード」を参照してください。

センサの較正

リスト 3.4-4｜センサの較正：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);
}
```

ここでは無限ループに入り、センサの較正が未完了の間、その途中経過を 1 秒周期で表示します（①）。センサの較正が終了したかどうかは is_fully_calibrated メソッドで調べることができ、これが true であるならば、ループを抜けます（②）。この時、ただ break;としてしまうと、5 ミリ秒×200=1 秒周期を作るための for ループを抜けるだけになってしまうので、loop の前にフラグ'label をつけて、break 'label;で一気に脱出します。1 秒待っても較正が完了していない場合は get_calibration_status メソッドを呼び出し、SYS, ACC, GYR, MAG の 4 種類の較正状態を取得し、count 変数により数えている経過秒数とともに出力します（③）。

姿勢情報の出力

リスト 3.4-5｜姿勢情報の出力：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);
}
```

較正が終了したら、BNO055 から姿勢情報を取得して出力します。ここでは eular_angles メソッドを呼び出して、ブレッドボードのオイラー角を取得しています。そして、姿勢情報はオイラー角ではなくクオータニオン（四元数）としても取得することができます。というより、BNO055 は姿勢情報を基本的にクオータニオンとして保存していて、オイラー角はそこから変換した 2 次産物に過ぎません。このあたりの議論は「（コラム）BNO055 におけるオイラー角の取り扱い」の後半部に書かれています。

姿勢情報の他に、センサ単体の出力値も取得できます。メソッド名と取得できる値の関係に

ついては、以下の対応表を参照してください。

表 3.4-1｜メソッドと取得できる値

メソッド名	取得できる値	Fusion Mode
quaternion	クオータニオン（姿勢角）	要
euler_angles	オイラー角（姿勢角）	要
linear_acceleration_fixed	直線加速度（cm/s²）	要
linear_acceleration	直線加速度（m/s²）	要
gravity_fixed	重力方向ベクトル（cm/s²）	要
gravity	重力方向ベクトル（m/s²）	要
accel_data_fixed	加速度センサの測定値（cm/s²）	不要
accel_data	加速度センサの測定値（m/s²）	不要
gyro_data_fixed	ジャイロセンサの測定値（1/16×°/s）	不要
gyro_data	ジャイロセンサの測定値（°/s）	不要
mag_data_fixed	地磁気センサの測定値（1/16×uT）	不要
mag_data	地磁気センサの測定値（uT）	不要
temperture	チップの温度（℃）	不要

ここで、Fusion Mode の項目が「要」になっている項目は、BNO055 の動作モードが Fusion Mode のいずれかに設定されていないと使用できません。また、「不要」の項目であっても、対応するセンサが利用可能なモードになっている必要があります。BNO055 の動作モードについては、「（コラム）BNO055 の動作モード」を参照してください。

3.5 較正情報の保存（フラッシュメモリへのデータ書き込み）

前節で BNO055 を制御するための基本のコードを書きましたが、毎回センサの較正に時間が取られるのは大変です。そこで、本章ではこの較正結果を保存し、次回起動時に再利用することを考えます。

較正情報の直打ち

センサの較正情報は、bno055::BNO055Calibration 構造体として取り扱われています。この構造体は全部で 22 個のフィールドから構成されています。そのため、一度較正を行った後

これらをすべて調べ、コードに１つ１つ書き込んでおくことで次回に再利用できます。
サンプルコードの src/imu_const.rs では、著者が以前に較正を行った際の結果を定数 CALIB として指定し、これを読み込んで較正作業をスキップしています。

【外部リンク】src/imu_const.rs サンプルコード
https://github.com/doraneko94/rp_bno055/blob/master/src/imu_const.rs

リスト 3.5-1｜BNO055Calibration 構造体の読み込み：src/imu_const.rs（Github）

```
...
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が、著者が以前に較正を行った結果で、これを set_calibration_profile メソッドを用いて読み込むことで、過去の較正結果を再利用できます（②）。①の値を取得するためには、較正作業が完了した後、calibration_profile メソッドを用いて BNO055Calibration 構造体を取得し、そのフィールドを１つ１つシリアル通信で表示する等のコードを書けば良いです。

較正情報を Raspberry Pi Pico に保存

前項のコードを書けば確かに較正情報を再利用することができますが、そのためには本命のコードとは別に、較正を行い→BNO055Calibration 構造体を取得し→フィールドを１つ１つ表示する特別ファイルを用意する必要があり、さらに、その出力を再度本命のコードに打ち込むという結構面倒な手順を経なければいけません。そこで、較正情報を Raspberry Pi Pico 自身に保存することを考えます。これは以下のようなイメージです。

1. 初回は普通に較正する
2. 初回の較正が終了したら、その結果を Raspberry Pi Pico に保存する
3. 電源を切る（初回の起動を終了する）
4. 再度電源を供給する（2 回目の起動）
5. 前回の較正結果が保存されているので、それを読み込んで再利用する

RP2040 のメモリ事情

Raspberry Pi Pico、というよりも、それに搭載されている RP2040 チップには、メイン RAM（一時データ＝計算結果等、の保存場所）として 264kB の SRAM（その他諸々合わせて 284kB）の他に、プログラム保存用の外部フラッシュメモリが搭載されています。RP2040 では XIP（eXecute In Place）方式を採用し、この外部フラッシュメモリをメモリ空間の

0x1000_0000 番地以降にマッピングして、内部メモリのように使用することを可能にしています。

外部フラッシュメモリの容量は 16MB（16,777,216）です。これが 32 個のブロックに分割され、各ブロックはさらに 16 個のセクタに分かれます。フラッシュの内容の消去はこのセクタ単位で行われ、書き込みは 256Byte のページ単位で実行されます。Rust からメモリを操作する際には 1Byte の情報を u8 型を用いて操作するため、以上の議論を Byte 単位で考え、さらに 16 進数を用いて表記すると

表 3.5-1 | Byte 単位の容量と 16 進数表記

項目	Byte 表記	16 進数表記
マッピングの開始アドレス	-	0x1000_0000
外部フラッシュメモリ容量（終端アドレス）	16,777,216 B ÷ 8 = 2,097,152 Byte	0x0020_0000
ブロックサイズ	2,097,152 ÷ 32 = 65,536 Byte	0xFFFF
セクタサイズ	65,536 ÷ 16 = 4,096 Byte	0x1000
ページサイズ	256 Byte	0x100

となります。外部フラッシュメモリの容量は 16MB と、大抵のプログラムの書き込みには十分すぎるものです。つまり、プログラムがフラッシュメモリの先頭から書き込まれていくと考えた場合、フラッシュメモリの末尾には余分な空間が生じるため、今回はそこに較正結果のデータを保存します。
較正結果は u8 型（1Byte）が 22 個の 22Byte から構成されるため、これは 1 個のセクタに収まります。したがって、今回はフラッシュの末尾にあるセクタの内容を消去し、その後、そこに必要な情報を書き込む、という作戦を採ります。

なお、フラッシュメモリを直接触るという行為はそれなりに危険であり、特に書き込み動作は unsafe な処理になります。そして、そのような危険な処理は、動作を安定させるためにいくつかの下準備が必要になります。ここでは、Rust x Raspberry Pi Pico（ラスピコ）の総本山とも言える、rp-rs/rp-hal リポジトリ（rp2040_hal という最重要なクレートを作っている所です）の Issue #257 での議論を参考に、フラッシュメモリを読み書きする処理を実装していきます[rp-rs A]。

【外部リンク】rp-rs/rp-hal Issue #257
https://github.com/rp-rs/rp-hal/issues/257

外部フラッシュ操作の準備

まず、Issue #257 では、以下の記述を Cargo.toml に追加することが重要であると結論付けられています。

リスト 3.5-2｜最適化情報の指定：Cargo.toml

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

この中で特に重要なのが、「# <-- HERE」で示されている lto（Link Time Optimizations）です。これを"fat"（true と同値）に設定し、LLVM（コンパイラ）によるコード生成の際に、プログラムの依存関係グラフのすべてのクレートにわたって最適化を実行させる[rust-lang B]必要があります。こうしなければ、外部フラッシュメモリの操作を含むプログラムのコンパイルに失敗します。

外部フラッシュメモリの読み取り

以下、フラッシュ操作に必要な関数を記述していきます。これらをすべてバイナリファイルに記述するとファイルが非常に長くなることや、今後別のファイルで再利用することも考慮して、src/lib.rs に書いていくことにします。

リスト 3.5-3｜フラッシュ読み取り関数：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
    }
}
```

まず、定数として、外部フラッシュメモリがマッピングされている開始アドレス（XIP_BASE）と、外部フラッシュメモリの終端のアドレス（FLASH_END）を定義します（①）。これらの値については、表 3.5-1 を参照してください。

read_flash 関数では、引数として与えた読み取り用バッファ data のサイズ分（②：size）だけ、外部フラッシュメモリの末尾の内容を読み取ります。末尾のアドレスは、外部フラッシュメモリがマッピングされている XIP_BASE からその容量分だけ進んだところ（XIP_BASE+FLASH_END）にあるので、読み取り開始アドレスにはそこから size だけ前に戻ったところを指定します（③）。そして、core クレートにある slice::from_raw_parts 関数に開始アドレスと読み取りサイズを与えて内容を取得し、1 つ 1 つ data に写し取っていきます。

外部フラッシュメモリへの書き込み

以下の定数や関数を、引き続き src/lib.rs に追記します。

リスト 3.5-4｜フラッシュ書き込み関数：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_BLOCK_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
        });
    }
}
```

①で定義される定数は表 3.5-1 を参照してください。また、FLASH_BLOCK_CMD はフラッシュ消去の際に用いる消去コマンドですが、Issue #257 によると、これ（0x20 = 32）が正常に動作する値のようです。さらに、関数定義の前に#[inline(never)], #[link_section = ".data.ram_func"]というアトリビュートが付けられていますが、これはなくても動作するそうです。しかし、著者はお守りとして消さずに残しています。

write_flash 関数は、引数に与えたデータ（data：u8 配列型で与えられる）の内容を、外部フラッシュメモリの末尾に書き込みます。このとき、データの大きさを size とすると（②)、書き込み開始のアドレスは FLASH_END − size になります（③)。
実際の書き込み操作は unsafe です。そしてこのとき、このプログラムが書き込みを行っている間に、他の処理によって妨害されないように注意する必要があります。そのため、cortex_m クレートの interrupt::free 関数を用いて、ここでは以下の処理を割り込みフリーの状態で実行しています（④)。

1. フラッシュへの接続
2. RP2040 の XIP 処理を（一旦）終了
3. フラッシュ末尾の消去
4. フラッシュ末尾への書き込み
5. XIP 処理の再開
6. XIP の構成の再設定

これらの処理は、rp2040_hal クレートの rom_data の関数群を使って実行します。rp2040_hal::rom_data では、RP2040 が標準搭載している Bootrom の関数群をそのまま利用できるようにしています。フラッシュの消去は flash_range_erase 関数、書き込みは flash_range_program 関数で行います。ここで、RP2040 におけるフラッシュの消去はセクタ（4096Byte）単位、書き込みはページ（256Byte）単位で実行する必要があったことを思い出してほしいのですが、ここでは消去と書き込みの開始アドレスに共通の addr を使用しているので、addr はより範囲の大きい、4096 の倍数として指定しなければなりません。すなわち、この関数を用いた書き込みは、セクタ単位で行うことになります。

書き込み記録の保存

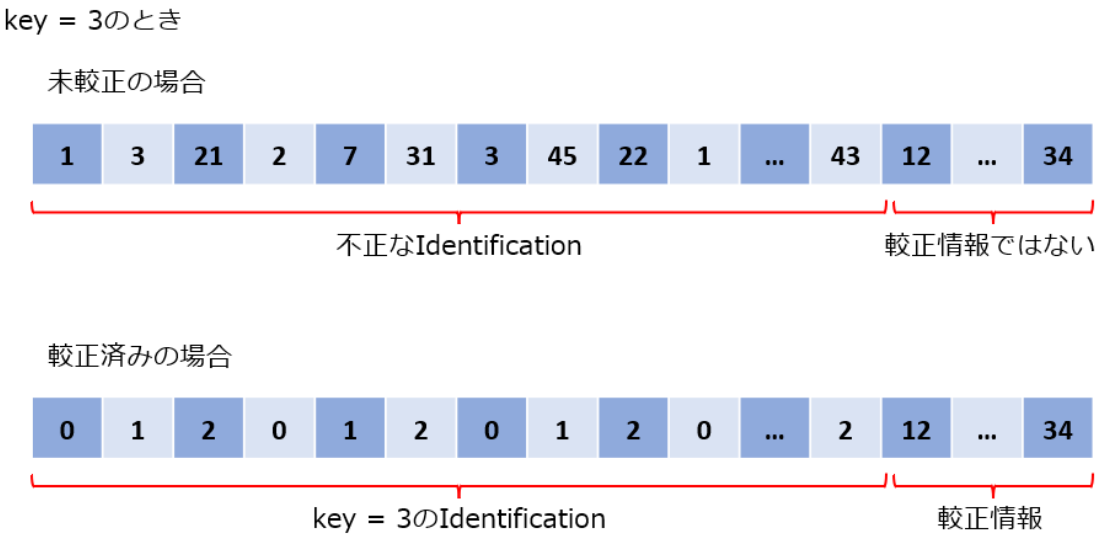
我々が目指す較正情報の保存機能の使用は、以下のようなものでした。

1. 外部フラッシュメモリに過去の較正情報が書き込まれているなら、それを読み込む
2. 書き込まれていないなら、較正を行い、その結果を外部フラッシュメモリに記録する

しかし、過去の較正情報が書き込まれていない状態でも、外部フラッシュメモリから何らかの値を読み込むことはできてしまいます。そのため、較正情報を保存したかどうかという情

報も、較正情報と同時にフラッシュに保存しておく必要があります。write_flash 関数はその使用上、最低でも 4096Byte の書き込みを行わなければならないので、較正情報の 22Byte 以前の部分を保存履歴の記録のために贅沢に使ってしまいましょう。
ここでは、セクタ前方の 4096 − 22 = 4074Byte に合い言葉や特徴的なパターンを記述し、これがあれば較正情報を記録済み、なければ未較正と判断します。パターンの作り方には色々な手法が考えられますが、ここでは 1〜255 の値を取る key の周期で数字を並べることにします。想定される挙動は以下のようになります。

図 3.5-1｜「記録済み」の判定法



これを実装したものが、以下の 2 つの関数です。

リスト 3.5-5｜identification の生成と認証：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 関数は引数 key 周期の数列（indentification）を生成します（①）。check_identification 関数は引数 data で与えられる配列の値を調べ、それらがすべて引数 key 周期の数列のルールを満たしているかどうかを判定します（②）。

較正情報保存機能を持った BNO055 制御プログラム

以上の関数を用いて、最終的なプログラムを書いていきます。src ディレクトリに imu_save.rs というファイルを作成し、Cargo.toml にバイナリファイルとして登録してください。

リスト 3.5-6 | [[bin]]ブロックの追記：Cargo.toml

```
[[bin]] # ファイルの末尾
name = "imu_save"
path = "src/imu_save.rs"
```

そして、サンプルコード

【外部リンク】src/imu_save.rs サンプルコード
https://github.com/doraneko94/rp_bno055/blob/master/src/imu_save.rs

の内容を src/imu_save.rs にコピーします。以下、src/imu_base.rs との差異を中心に、重要な部分を解説します。

リスト 3.5-7｜較正情報の保存と再利用：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: BNO055Calibration) { // 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::<()>();
}
```

まず、必要なクレートや src/lib.rs で定義した関数群を宣言します（①）。そして定数として、書き込むデータの大きさ DATA_SIZE を（ページではなく）セクタと同じ大きさに設定します（②）。その他、CALIB_SIZE は BNO055Calibration 構造体が持つフィールド数と同じ 22Byte、indentification のサイズ IDENT_SIZE は、セクタ長から CALIB_SIZE を引いた残りすべてとします。identification 生成の際に用いる IDENT_KEY には、ここでは 192 を設定していますが、好みで別の数にしてもらっても問題ありません。

外部フラッシュメモリの読み書きは、calib_data 変数に確保した、長さ DATA_SIZE バッファを介して行います（③）。このバッファに read_flash 関数で外部フラッシュメモリの末尾の

値を読み取り、先頭から IDENT_SIZE までの数列が identification として正当なものであるかを判定します（④）。正当であると判断された場合には、バッファの identification 部より後の情報を長さ CALIB_SIZE の u8 配列型に読み取り（⑤, ⑤'）、これを from_buf 関数を用いて BNO055Calibration 構造体に変換し、再利用します。正当な identification が書き込まれていない場合には、BNO055 の較正が過去に一度も行われていないと判断し、src/imu_base.rs 同様に較正のためのループに入ります（⑥）。完了後、calib_buf を一旦すべて identification で埋め（⑦）、その後、後方の CALIB_SIZE の長さだけを較正情報に書き換えます（⑧, ⑧'）。このとき、as_bytes メソッドを用いて BNO055Calibration 構造体を u8 配列型に変換して処理しています。最後に、write_flash 関数で calib_buf の内容を外部フラッシュに書き込みます。その後は src/imu_base.rs と同じく、姿勢角表示のための無限ループに入ります。

動作の確認

以上で準備完了です。早速実行して、動作を確認してみましょう。

リスト 3.5-8 | src/imu_save.rs の初回実行：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
...
```

このように、src/imu_base.rs と同様の挙動を示します。一度 USB 接続を解除し、BOOTSEL ボタンを押しながら再接続して、もう一度同じコマンドを実行してみましょう。(BOOTSEL ボタンを押さずに PC に接続し、「（コラム）シリアル通信の開始」で紹介した oscillo_serial を用いても問題ありません)。

リスト 3.5-9 | src/imu_save.rs の 2 回目の実行：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
...
```

今度はいきなり姿勢情報の表示が始まりました！これで、較正情報をスキップできていることがわかります。

較正情報の保存期間

リスト 3.5-8 の操作において src/imu_save.rs のプログラムを Raspberry Pi Pico に書き込んで実行し、その際、較正情報を外部フラッシュメモリに保存しています。その後、リスト 3.5-9 の操作においては、リスト 3.5-8 で書き込んだプログラムを一旦消去して、新しく src/imu_save.rs のプログラムを書き込んで実行しているのですが、それでも較正情報ならびに identification の情報は消えていません。すなわち、新しいプログラムを書き込んだ後でもこの情報は消えません。試しに、imu_save.rs → serial_hello.rs → imu_save.rs と異なるプログラムの書き込みと実行を間に挟んでみても、1 回目の imu_save.rs で書き込まれた情報が残っていることが確認できます。

これは、imu_save.rs や serial_hello.rs から生成されるプログラムの容量が小さく、較正情報や identification が書かれているメモリの位置まで届かないためです。実際、これらの情報が書かれている最終セクタまではほぼ 16MB の猶予があり、ここに届く（そして最大容量以内に収まる）プログラムが書き込まれることはほとんどありません。

較正のやり直し

では心機一転、これまでの較正情報を破棄し、再び較正をやり直したくなった場合はどうしたら良いでしょうか？プログラムを上書きしても保存した情報が消えないため、何らかの工夫が必要になります。回路にトグルスイッチなどのパーツを追加して、それを長時間押した場合に較正をやりなおすというのも一手ですが、identification の実装を利用して、コードのわずかな改変だけでこの問題に対応することができます。

その方法とは、identification の生成に用いる key を変えることです。以下にその例を示します。

図 3.5-2｜key の変更による較正情報の無効化

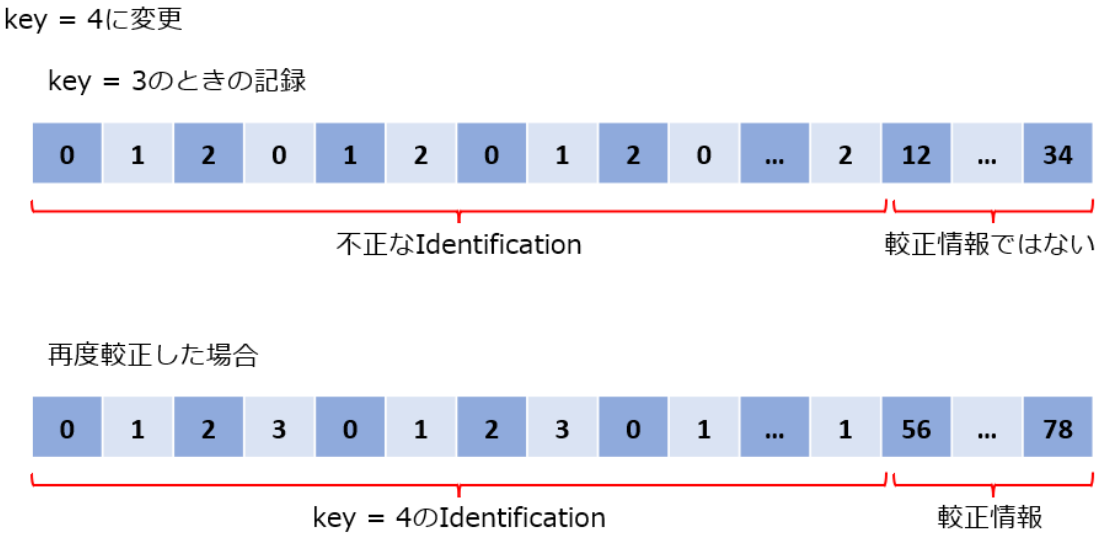


図 3.5-2 では、key が 3 の状態で保存された較正情報を、key を 4 に変更することで無効化しています。key＝4 のとき、周期 3 の identification は不正なものとみなされ、それに続く 22Byte も較正情報ではないと判断されるのです。その後、key＝4 の状態で保存された情報は、key＝4 であり続ける限り再利用されます。

実際に、src/imu_save.rs の IDENT_KEY を変更して実行してみましょう。

リスト 3.5-10｜IDENT_KEY の変更：src/imu_save.rs

```
...
const IDENT_KEY: u8 = 191; //  もともと 192 だった
...
```

リスト 3.5-11｜src/imu_save.rs の IDENT_KEY を変えて実行：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)
...
```

この通り、再び較正が始まりました。

（コラム）出力情報のグラフ表示

リスト 3.3-5 に表示されたような数字の羅列では、値の変動がわかりにくいかもしれません。そこで、値をグラフ表示してみましょう。
ここでは、「(コラム) シリアル通信の開始」で紹介した oscillo_serial の plot 機能を使って、BNO055 の出力時系列をプロットします。

【外部リンク】oscillo_serial Github リポジトリ
https://github.com/doraneko94/oscillo_serial

コードの書き換え

plot 機能にはフォーマットの制限があるため、src/imu_save.rs で表示される文字列を変更します。そのために、src ディレクトリに新しく imu_save_plot.rs というファイルを作り、Cargo.toml にバイナリファイルとして登録します。

リスト ex3-1 | [[bin]]ブロックの追記：Cargo.toml

```
[[bin]] # ファイルの末尾
name = "imu_save_plot"
path = "src/imu_save_plot.rs"
```

そして、以下のサンプルコードを参考に、内容を記述してください。

【外部リンク】src/imu_save_plot.rs サンプルコード
https://github.com/doraneko94/rp_bno055/blob/master/src/imu_save_plot.rs

このファイルは、src/imu_save.rs の無限 loop の中だけを変更しています。

リスト ex3-2 | loop での表示内容を変更：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);
}
```

これを実行すると、次のような「,」区切りで、pitch と roll の情報が出力されます（現在の IDENT_KEY での較正が完了していない場合は較正が始まります）。

リスト ex3-3｜src/imu_save_plot.rs が表示する生データ：shell

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

較正が完了した状態で一度 USB 接続を解除し、oscillo_serial を使用するための新しいコマンドプロンプトを開きましょう。そして、BOOTSEL ボタンを押さずに Raspberry Pi Pico を PC に接続した後、新しいコマンドプロンプトで以下のコマンドを実行して oscillo_serial を plot モードで起動します。

リスト ex3-4｜oscillo_serial を plot モードで実行：shell

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

ここで、各パラメータの意味は以下の通りです。

表 ex3-1｜oscillo_serial のパラメータ

パラメータ	設定項目	取りうる値
-mo	モード	text, plot
-xs	グラフの X 方向のサイズ	整数

-ys	グラフの Y 方向のサイズ	整数
-de	要素を区切る文字	文字列
-ne	表示する要素の個数	整数

すなわち、今回は plot モードで、100×20 の描画領域を持つグラフに、「,」で区切られた 2 つの要素（pitch と roll）を表示するように設定しています。

そのため、起動後には以下のような出力がリアルタイムで表示されます。

図 ex3-1｜plot モードの出力

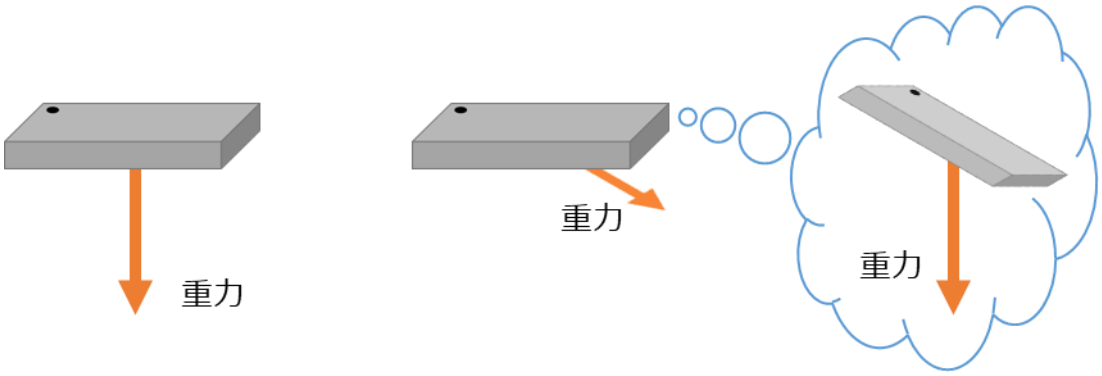
①がブレッドボードを roll 回転だけさせた時間帯で、②が pitch 回転だけさせた時間帯です。

oscillo_serial の plot モードの実装についてはリポジトリを参照してください。なお、oscillo_serial は開発中のソフトウェアなので、実装や仕様が変化する可能性があります。

（コラム）センサ情報による姿勢角の計算原理

加速度センサ

図 ex3-2｜加速度センサによる姿勢推定



加速度センサを水平面に置くと、重力加速度は Z 軸方向に検知されます（図 ex3-2・左)。逆に、検知されている加速度ベクトルが Z 軸以外の方向を向いている場合、重力方向が変わることは起こり得ないので、デバイス自体が傾いているのだと判断することができます（図 ex3-2・右)。ただし、デバイスを水平面上で回転させても検知される加速度ベクトルの方向は変化しないので、加速度センサによって水平面上での回転を区別することはできないことがわかります。

加速度センサは非常に繊細で、細かなノイズが乗りやすいことが知られています。また、加速度センサが動いている場合には重力以外の力の影響も受けるため、加速度の測定値から単純に姿勢角を求めるのは困難になります。

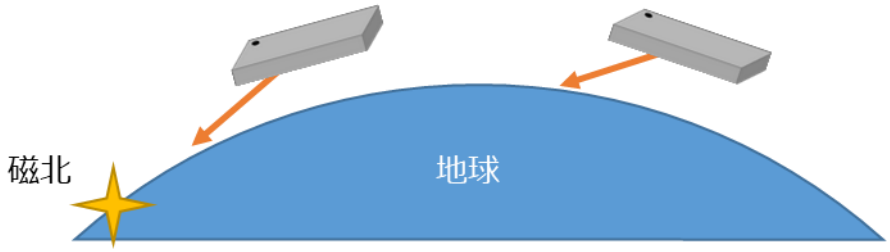
ジャイロセンサ

ジャイロセンサは XYZ それぞれの軸周りの回転速度（角速度）を計測するセンサです。そのため、計測値を積分する（計測値にサンプリング時間Δt をかけて足し合わせていく）ことにより、計測を開始した時から現在までの姿勢角の変化が計算できます。

したがって、ジャイロセンサを用いて計算できる姿勢角は相対的なものになります。また、積分の性質上、各測定値の誤差が蓄積していくことになるため、ドリフトと呼ばれる誤差が生じることが知られています。

地磁気センサ

図 ex3-3｜ 地磁気センサによる姿勢推定



磁北（真北からは少しずれる）の方向を計測し、そのベクトルから現在の姿勢角を計算します。磁北という基準があるので、姿勢角は絶対的なものになります。
しかし、図 ex3-3 にある通り、デバイスの地球上の位置によって、その計測値は変化します（図のデバイスは、ともに地上に対して同じ姿勢をとっています)。デバイスが自身の緯度経度を知ることができない場合、これにより大きな誤差が生じます。また、地磁気センサは周囲の磁場の乱れに影響されやすいため、使用できる環境には制限があります。

センサフュージョンによる姿勢角推定

以上により、単体センサでの姿勢推定は困難であることがわかりました。そのため、デバイスの姿勢角を推定する際には複数のセンサの測定値を組み合わせるという手法が採用されます。例えば、以下の式で表現される相補フィルタは、加速度センサとジャイロセンサによる姿勢角の推定法です。

式 ex3-1｜ 相補フィルタの更新式

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

ここで、θ(t)は時刻 t における姿勢角の推定値、ω_g(t)は時刻 t におけるジャイロセンサの測定値、θ_a(t)は加速度センサの測定値のみから計算した姿勢角の推定値です。見方によっては、ジャイロセンサと加速度センサ、それぞれからの推定値を重み係数 K によって平均したと考えることができます。実際には、この操作によって加速度センサのノイズ（高周波）とジャイロセンサのドリフト（低周波）を取り除くためのフィルタとして機能し、そのカットオフ周波数を f_c とすると、K は以下のように計算できます。

式 ex3-2｜K の決定法

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

以上のように、複数のセンサ出力を組み合わせて精度を上げたり、新しい情報を導いたりすることをセンサフュージョンと言います。

（コラム）BNO055 の動作モード

表 ex3-2 | BNO055 の動作モード

動作モード		利用可能な信号			方位角	
		加速度	地磁気	ジャイロ	相対的	絶対的
	CONFIGMODE					
非 Fusion Mode	ACCONLY	○				
	MAGONLY		○			
	GYROONLY			○		
	ACCMAG	○	○			
	ACCGYRO	○		○		
	MAGGYRO		○	○		
	AMG	○	○	○		
Fusion Mode	IMU	○		○	○	
	COMPASS	○	○			○
	M4G	○	○		○	
	NDOF_FMC_OFF	○	○	○		○
	NDOF	○	○	○		○

（[BOSCH 2021] Table 3-3 をもとに作成）

CONFIGMODE

電源を入れた直後はこのモードになります。いずれのセンサ情報も読み取ることはできませんが、XYZ 軸の入れ替えやセンサの範囲・帯域設定など、様々な設定を行うことができるモードです。

非 Fusion Mode

個々のセンサの測定値のみを使用するモード群です。使用可能なセンサがそのままモードの名前になっています。

Fusion Mode

複数のセンサの測定値を組み合わせ、空間におけるデバイスの向きを計算するモードです。向きを表す角度のうち、方位角について絶対的なものが出力されるか、相対的なものになる

かはモードによって異なります。絶対的な方位角とは、磁北を 0 度とした角度であり、地磁気センサを用いてこれを取得できるモードがこれを出力します。それ以外の場合は、電源を入れた時点の向きを 0 度とした相対的な方位を出力します。
さらに、Fusion Mode では、直線加速度や重力ベクトル（デバイスから見た重力の向き）を出力できます。加速度センサは重力の影響の他、デバイスが加速度運動をしている場合にはその影響も受け、これらの和を測定値として出力しています。しかし、ここでジャイロセンサや地磁気センサの測定値も考慮に入れると、これら 2 つの影響を分離することができるのです。
以下、各 Fusion Mode の性質を紹介します。

- IMU：加速度センサとジャイロセンサを用いる。動作が速い。
- COMPASS：加速度センサと地磁気センサを用いる。方位の計算には地磁気ベクトルのうち、水平面の成分（XY 軸）のみ参照する。磁場が不安定な環境での使用には注意が必要。
- M4G：地磁気センサを使用しているが、地磁気ベクトルはその変位のみを参照しジャイロセンサのように扱う。ジャイロセンサが持つドリフトが生じないというメリットがある。地磁気の変位のみを参照するため、磁北を測ることができず、方位角は相対的になる。以上の性質により、地磁気センサを較正する必要がない（というより、できない）。磁場が不安定な環境での使用には注意が必要。
- NDOF_FMC_OFF：NDOF から Fast Magnetometer Calibration の機能を除いたモード。
- NDOF：すべてのセンサを統合したモード。Fast Magnetometer Calibration により地磁気センサの較正を高速化するとともに、精度も向上させている。加速度・ジャイロセンサによる補正が効くので、COMPASS・M4G モードに比べて、磁場が不安定な環境に対する耐性がある。

NDOF・NDOF_FMC_OFF 以外のモードを使う際の注意点として、較正終了の判定に Bno055::is_fully_calibrated 関数を使わないことが挙げられます。この関数の実装を見ると

リスト ex3-5｜is_fully_calibrated 関数の実装：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.sys == 3) //
```

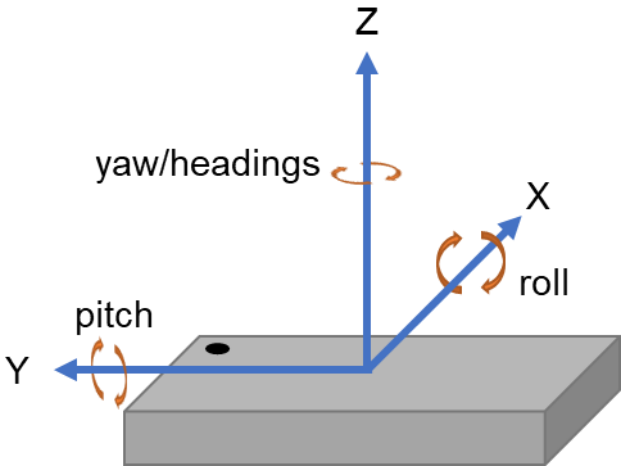
```
②
}
```

となり、SYS, ACC, GYR, MAG の全ての値が 3 になったかどうかを判定しています（②)。現在のモードにおいて使用されないセンサの較正パラメータは常に 0 のままですから、これでは較正が永久に終わりません。よって、実装の内部で行われているように、bno055::BNO055CalibrationStatus を取得し（①)、現在取り扱っているセンサについてのみ、パラメータの値を確認するコードを書くと良いでしょう。

（コラム）BNO055 におけるオイラー角の取り扱い

著者・BNO055・各クレートにおける扱いの違い

図 3.3-1｜ オイラー角と XYZ 軸の関係（再掲）

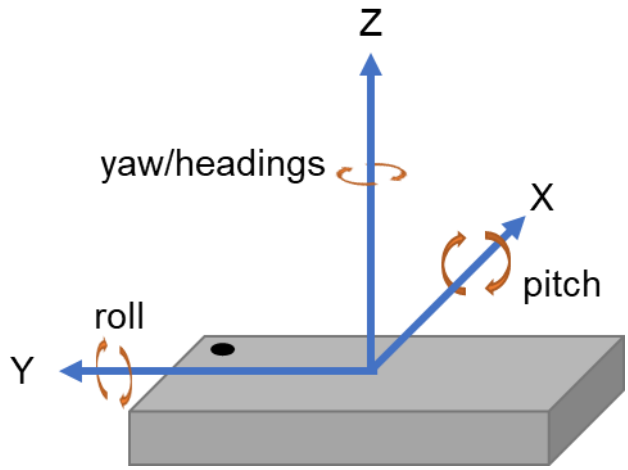


著者が考える回転角の名付け方は図 3.3-1 で示した通りで、本書ではこの対応関係でオイラー角を表現します。進行方向を X 軸とし、その周りの回転を roll と呼ぶのが一般的な解釈であると考えられるためです。

【外部リンク】ローリング – Wikipedia
https://w.wiki/6ifv

しかし、公式情報（実は[BOSCH 2021]には載っていないので、他の BNO055 搭載モジュールや関連ページの情報に基づく。例えば[MathWorks A]を参照）によると、オイラー角は次のように対応付けられています。

図 ex3-4｜ オイラー角と XYZ 軸の関係（公式情報）



そして EUL_DATA_LSB（0x1A)で始まるアドレスから、レジスタに yaw/headings, roll, pitch の順で情報が保存されています。

次に、bno055 クレートにおけるオイラー角の扱いを見てみましょう。クレートの Docs-rs によると、euler_angles メソッドには以下のようなコメントが添えられています。

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

つまり、オイラー角を取得して roll, pitch, yaw の順番で表現すると述べています。確かに実装を見てみると

リスト ex3-6｜euler_angles の実装：bno055/src/lib.rs（Github）

```
/// 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, ()>, Error<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, heading * scale]); // ②

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

①で heading, roll, pitch の順番で取得した情報を、roll, pitch, heading の順に並び替えて mint::EulerAngles 構造体に保存しています。

最後に、mint::EulerAngles 構造体でのオイラー角の扱いを見てみましょう。mint クレートの Docs-rs には以下のような記述があります。

“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).”

すなわち、pitch, yaw, roll の順番で保存されることを期待しています。つまり、bno055 クレートは mint クレートの要請を完全無視しています。また、著者は BNO055 から得られる roll と pitch の名称が逆であるべきだと思っているので、本書ではこれらを入れ替えています。以上の関係を表にすると以下のようになります。

表 ex3-3 | mint, bno055, 著者のオイラー角の扱い

mint::EulerAngles のフィールド名	mint クレートが指定するオイラー角	bno055 クレートが保存するオイラー角	本書における名称
a	pitch	roll	pitch
b	yaw/heading	pitch	roll
c	roll	yaw/heading	yaw/heading

オイラー角の名称の他にも、その値の範囲が異なる点にも注意が必要です。以下の表に mint クレートと bno055 クレートにおける範囲の違いを示します。

表 ex3-4 | オイラー角の範囲

オイラー角	mint クレートが指定する範囲	bno055 クレートが出力する範囲
pitch	$-\pi \sim \pi$	-90～90°
roll	$-\pi \sim \pi$	-180～180°
yaw/heading	$-\pi/2 \sim \pi/2$	0～360°

実は BNO055 にもオイラー角をラジアンで出力する機能はあるのですが、bno055 クレートはそれを提供していません。

その他 BNO055 のオイラー角が抱える問題点

名称や範囲の他にも、BNO055 のオイラー角出力には以下の問題が知られています。

- 精度が低い（値の刻みが大きい）
- pitch または roll が±45° を越えると、yaw の出力が不安定になる

これらの問題を一括して解決するには、特段の事情がない場合にはオイラー角ではなくクオータニオン（四元数）を使用することを勧めます。クオータニオンの性質については、例

えば著者の解説記事を参照してください[艮電算術研究所 B]。

参考文献リスト

Rust の基本

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

[rust-jp A] The Rust Programming Language 日本語版. https://doc.rust-jp.rs/book-ja/

[κeen ほか 2019] κeen, 河野達也, 小松礼人. 実践 Rust 入門 – 言語仕様から開発手法まで. 技術評論社（2019）

Cargo.toml の記法

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

Rust 用語集

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

Git

[湊川ほか 2021] 湊川あい, DQNEO. 改訂 2 版 わかばちゃんと学ぶ Git 使い方入門 <GitHub、SourceTree、コマンド操作対応>. C&R 研究所（2021）

Rust による組込みプログラミング

[中林ほか 2021] 中林智之, 井田健太. 基礎から学ぶ 組込み Rust. C&R 研究所（2021）

[初田ほか 2020] 初田直也, 山口聖人, 吉川哲史, 豊田優貴, 松本健太郎, 原将己, 中村謙弘. 実践 Rust プログラミング入門. 秀和システム（2020）

[Nakabayashi A] The Embedded Rust Book. https://tomoyuki-nakabayashi.github.io/book/

[Nakabayashi B] Embedded Rust Techniques.
https://tomoyuki-nakabayashi.github.io/embedded-rust-techniques/

Rust による Raspberry Pi Pico プログラミング（ラスピコ）

[艮電算術研究所 A] Rust x Raspberry Pi Pico で電子工作―実装例集
https://ushitora.net/archives/2617

BNO055

[BOSCH 2021] 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
[秋月電子通商 2022] BNO055 使用 9 軸センサフュージョン モジュール AE-BNO055-BO Ver.2022-4-13
https://akizukidenshi.com/download/ds/akizuki/AE-BNO055-BO_20220413.pdf

センサによる姿勢推定の原理

[野波 2020] 野波健蔵. ドローン工学入門 – モデリングから制御まで. コロナ社（2020）
[野波ほか 2022] 野波健蔵, 鈴木智, 王偉, 三輪昌史. ドローンのつくり方・飛ばし方: 構造、原理から製作・カスタマイズまで. オーム社（2022）

BNO055 におけるオイラー角

[MathWorks A] readOrientation
https://www.mathworks.com/help/supportpkg/arduinoio/ref/bno055.readorientation.readorientation.html

クオータニオン

[艮電算術研究所 B] クオータニオンの定義と性質
https://ushitora.net/archives/2961

Raspberry Pi Pico のフラッシュ読み書き

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

購入者特典パスワード 1：　#$u-$V7RMKdV
購入者特典パスワード 2：　sWpHsCu3zbfz

※これらのパスワードは、改訂版の配布などの購入者特典を利用する際に使用します。特典の情報は著者の Twitter アカウント（https://twitter.com/doraneko_b1f）または艮電算術研究所（https://ushitora.net）において配信する予定です。

Rust x Raspberry Pi Pico で実装する
IMU からの姿勢情報の取得と応用

大野 駿太郎　著

初版（電子書籍）　令和 5 年 5 月 21 日発行
発行元　艮電算術研究所（https://ushitora.net）
連絡先　sohno@ushitora.net
感想フォーム　https://ushitora.net/contact