

組み込み Rust

組み込み Rust 編

JIJINBEI

2024-10-15

HiCoder

Table of contents

1. 組み込みとは
2. なぜ組み込みで Rust
3. 組み込み Rust を試してみよう
4. 組み込み Rust の重要な概念
5. 組み込み Rust で開発する際に
6. PWM を使ってみよう
7. And more…

組み込みとは

組み込みとは

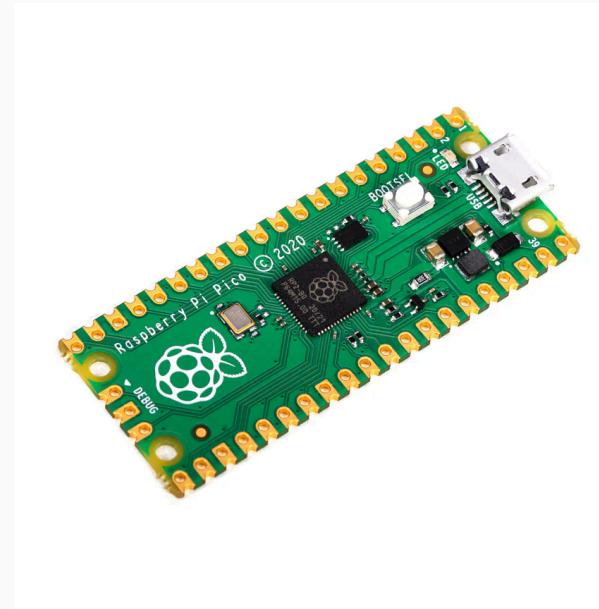


図 1: Raspberry Pi Pico

- マイコンを使ったシステム
- LED を光らせたり、モーターを動かすところから、車載システム、IoT まで

組み込みとは



図 2: 組み込みの例 (組込みシステム産業振興機構)

なぜ組み込みで Rust

Rust のメリット

- 安全性
 - ▶ 型安全性
 - ▶ メモリ安全性
 - 所有権
 - ライフタイム
- 処理速度の速さ
 - ▶ メモリ管理を自動で行わない
- 並行処理
 - ▶ 所有権によるデータ競合の防止
- バージョンやパッケージ管理
 - ▶ Cargo

Rust の言語の特徴が使え、C や C++よりも安全にプログラミングができる

難しすぎる

Rust のデメリット

組み込み Rust 特有の機能の理解

情報が少ない

型でコンパイルが通らない

難しそう

基本的に英語の文献

Rust の所有権システムやライフタイムが理解しづらい

コミュニティが小さい

組み込み Rust を試してみよう

「チカを試してみよう!!

<https://github.com/rp-rs/rp2040-project-template> のテンプレートを試し、ラズピコを 0.5 秒毎に LED を点滅させるコードを書いてみよう

方法

1. リンクを参考にライブラリをインストール
2. cargo install cargo-generate で cargo-generate をインストール
3. rp-pico のテンプレートをダウンロード

1 cargo generate <https://github.com/rp-rs/rp2040-project-template>

terminal

4. config.toml が runner = "elf2uf2-rs -d" となっていることを確認¹
5. cargo run でビルド

¹elf2uf2-rs は、uf2 ファイルを作成するツールで、probe-rs はデバックまで行えるツール(後述予定)

「チカコードの解説

先程のコードで本質的な部分はどこか？

「チカコード」の解説

先程のコードで本質的な部分はどこか？

```
1 loop {  
2     led_pin.set_high().unwrap();  
3     delay.delay_ms(500);  
4     led_pin.set_low().unwrap();  
5     delay.delay_ms(500);  
6 }
```

rust

「チカコード」の解説

先程のコードで本質的な部分はどこか？

```
1 loop {  
2     led_pin.set_high().unwrap();  
3     delay.delay_ms(500);  
4     led_pin.set_low().unwrap();  
5     delay.delay_ms(500);  
6 }
```

rust

それでは、それ以外のところは重要でないのか？

「チカコード」の解説

先程のコードで本質的な部分はどこか？

```
1 loop {  
2     led_pin.set_high().unwrap();  
3     delay.delay_ms(500);  
4     led_pin.set_low().unwrap();  
5     delay.delay_ms(500);  
6 }
```

rust

それでは、それ以外のところは重要でないのか？

とても重要

組み込み Rust の重要な概念

組み込み Rust をやるのに知るべきこと

組み込み Rust をやる上で重要な概念

- データシートの情報
- `#![no_std]`
- `#![no_main]` と `#[entry]`
- `use panic_halt as _;`
- `rp_pico::hal` と `embedded_hal`
- `let mut pac = pac::Peripherals::take().unwrap();`
- `let mut led_pin = pins.led.into_push_pull_output();` のような型について

データシートを読む

ラズピコは、RP2040 というマイコンを使っている

- RP2400 のデータシート: [\[link\]](#)
- ボードのデータシート: [\[link\]](#)

どの部品がどのマイコンのピンに接続されているかを理解する

問題

- 内蔵 LED の GPIO 番号を調べて、どのようにしたら点灯できるかを考えよう!!
- RP2040 に接続されている部品はどのようなものがあるか調べよう!!

`#![no_std]` とは

`#![no_std]`

- Rust の標準ライブラリは `core`², `alloc`³, `std` の三階層構造
- `no_std` は、Rust の標準ライブラリ[\[link\]](#) を使わず、OS を使わないことを示す。
 - `println!` は OS を使っているので使えない
 - 外部ファイルの読み書きもできない

²`core` クレートはプリミティブ型やアトミック操作

³`alloc` クレートはヒープメモリを利用するが、組み込みはメモリが小さいので `Vec` が簡単に使えない。使い方は `embedded-alloc` クレート [\[link\]](#)

`#![no_main] と #[entry]`

`#![no_main] と #[entry]`

`no_std` 環境では、`main` 関数は使えず、自分でエントリーポイント(プログラムが始まるところ)を指定

今回の例では、`#[entry]`で `main` 関数を指定している

panic_halt

```
use panic_halt as _;
```

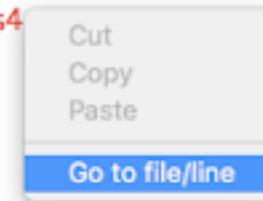
エラーとはどういうものを思い浮かべるか？

panic_halt

```
use panic_halt as _;
```

エラーとはどういうものを思い浮かべるか？

```
ok call to invminus4 next:  
0.5  
bad call to invminus4 next:  
Traceback (most recent call last):  
  File "/Users/anh/comp150/showtraceback.py", line 13, in <module>  
    main()  
  File "/Users/anh/comp150/showtraceback.py", line 10, in main  
    print(invminus4(4))  
  File "/Users/anh/comp150/showtraceback.py", line 4, in invminus4  
    return 1.0/(x-4)  
ZeroDivisionError: float division by zero  
>>>
```



エラー文が出てきて、プログラムが止まること

プログラム的には異常な動作だが、エラー文を出してプログラムを止めているので正常な動作

panic_halt

`use panic_halt as _;`

組み込みでは、

- 画面に表示できないハードウェアが多い
- エラーの表示にもメモリを使う(組み込みではメモリは少ない)

そのため、

- エラーが出たら、プログラムを止める(無限ループ)

組み込み機器の抽象化

```
rp_pico::hal = rp2040_hal
```

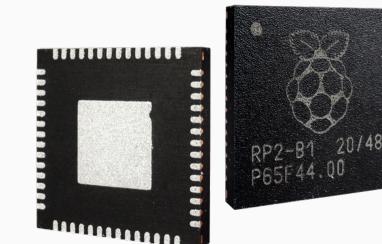
- rp_pico [link] はラズピコの **BSP(Board Support Package)** クレート
 - コードを見てみると、すごく簡単に書かれている(ほかのボードも) [link]

なぜ簡単に書かれているのか？

- **HAL(Hardware Abstraction Layer)**: ハードウェアの抽象化レイヤー
- 抽象化レイヤーを使うことで、ハードウェアの差異をうまく吸収してくれる
- rp2040_hal クレートを用いると、簡単に RP2040 のボードを作ることができるはず

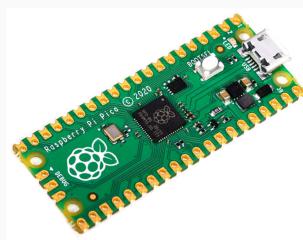
組み込み機器の抽象化

`rp_pico::hal = rp2040_hal`

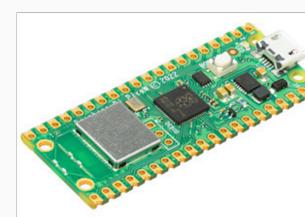


RP2400

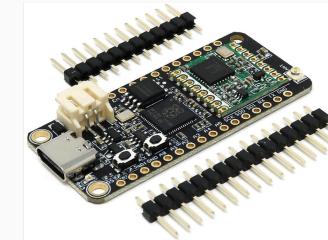
HALがハードウェア
の差分を吸収



Raspberry Pi Pico



Raspberry Pi Pico W



Arduino Nano
RP2040 Connect



Seeed Studio XIAO
RP2040

組み込み機器の抽象化

embedded_hal

問題意識

ボードやマイコンが違ってもどれも似たような機能を持っているはずなのに、コードの書き方が変わってしまう

⇒ Embedded devices Working Group (WG)が書き方を統一するために embedded_hal クレートを作成⁴

- OutputPin の例 [\[link\]](#)
- embedded_hal に準拠したクレート一覧 [\[link\]](#)

embedded_hal のおかげで複雑なクレートの関係をシンプルに

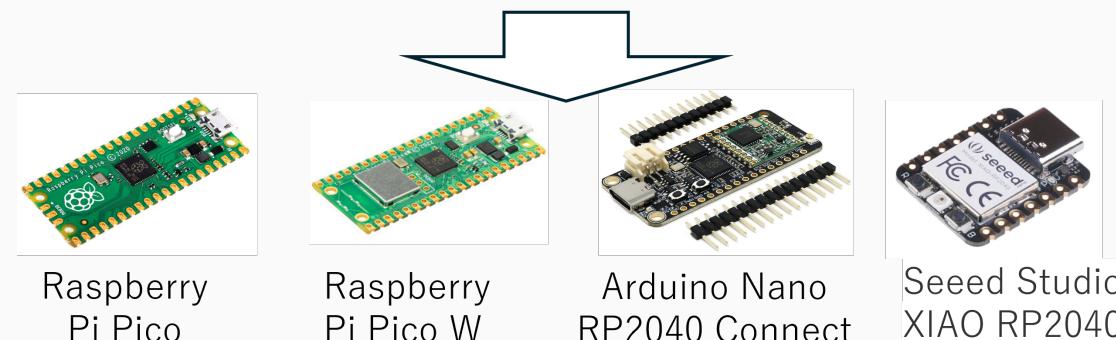
⁴トレイトを思い出そう！！

組み込み機器の抽象化

embedded-halで抽象化



マイコンの抽象化されたクレート



BSPクレート

組み込みで所有権を賢く使う

```
let mut pac = pac::Peripherals::take().unwrap();
```

```
1 let pins = rp_pico::Pins::new(  
2     pac.IO_BANK0,  
3     pac.PADS_BANK0,  
4     sio.gpio_bank0,  
5     &mut pac.RESETS,  
6 );
```

rust

で何度も出てくる pac とは何か？⁵

⁵Pins::new の引数に恐怖を覚えなくともよい。なぜなら型推測があるから

組み込みで所有権を賢く使う

```
let mut pac = pac::Peripherals::take().unwrap();
```

```
1 let pins = rp_pico::Pins::new(  
2     pac.IO_BANK0,  
3     pac.PADS_BANK0,  
4     sio.gpio_bank0,  
5     &mut pac.RESETS,  
6 );
```

rust

で何度も出てくる pac とは何か？⁶

Peripheral Access Crate の略で

マイコンの Peripheral を使うための許可証で、一個しか使われていないことを保証するもの

⁶Pins::new の引数に恐怖を覚えなくともよい。なぜなら型推測があるから

組み込みで所有権を賢く使う

```
let mut pac = pac::Peripherals::take().unwrap();
```

```
1 let pins = rp_pico::Pins::new(  
2     pac.IO_BANK0,  
3     pac.PADS_BANK0,  
4     sio.gpio_bank0,  
5     &mut pac.RESETS,  
6 );
```

rust

で何度も出てくる pac とは何か？⁷

Peripheral Access Crate の略で

マイコンの Peripheral を使うための許可証で、一個しか使われていないことを保証するもの

所有権のおかげで一度しか使われていないことを保証している！！

⁷Pins::new の引数に恐怖を覚えなくともよい。なぜなら型推測があるから

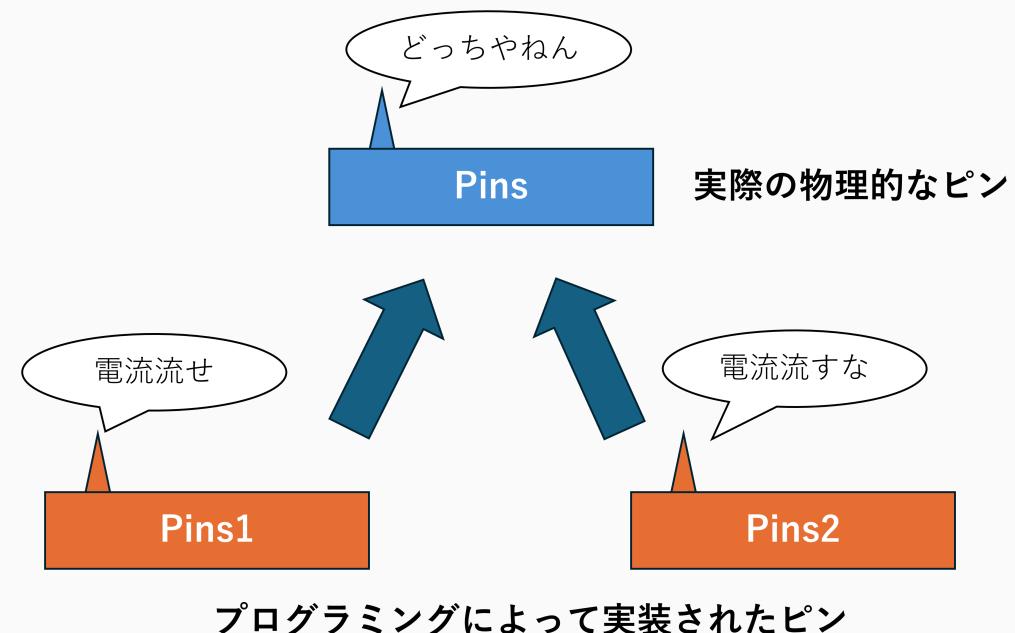
所有権をうまく使う

```
let mut pac = pac::Peripherals::take().unwrap();
```

所有権システムで Peripheral が使われているのが一つだけであることを保証する

メリット

- ・同時アクセスによるデータ競合を防ぐ



組み込み Rust で開発する際に

Rust で組み込みの情報を得るには

1. どこで情報を得るか？

- クレートはどのように使うか？
 - 使いたいクレートの example
 - 例: rp2040-hal[link] や rp-pico[link] の github の example を見る
- 使いたいメソッドでエラーがでたら
 - Docs.rs
 - これの読み方を覚えることが大事（次のスライド）
- RP2040 の機能がわからない
 - データシートを読む

2. 開発が進まないなら

- 僕に聞いてくれ。そして一緒に考えよう。
- Rust の過度な神格化をやめ、C や Python で書く

Docs.rs の読み方（型）

考え方（L チカの場合）

1. L チカでは、`set_high()` のメソッドを使いたい
2. docs.rs で `rp2040-hal` を検索
3. `set_high()` を検索し、型を確認

```
1 impl<I, P> OutputPin for Pin<I, FunctionSio<SioOutput>, P>
2 where
3     I: PinId, // Gpio0, Gpio1, ...
4     P: PullType, // PullUp, PullDown, ...
```

rust

4. Pin の型を満たすものに変換するものがないかを探す⁸

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

```
let mut led_pin = pins.led.reconfigure::<FunctionSio<SioOutput>, PullDown>();
```

⁸型変換に関するものなので、From トレイトや Into トレイトを使っていることが多い → into から絞る

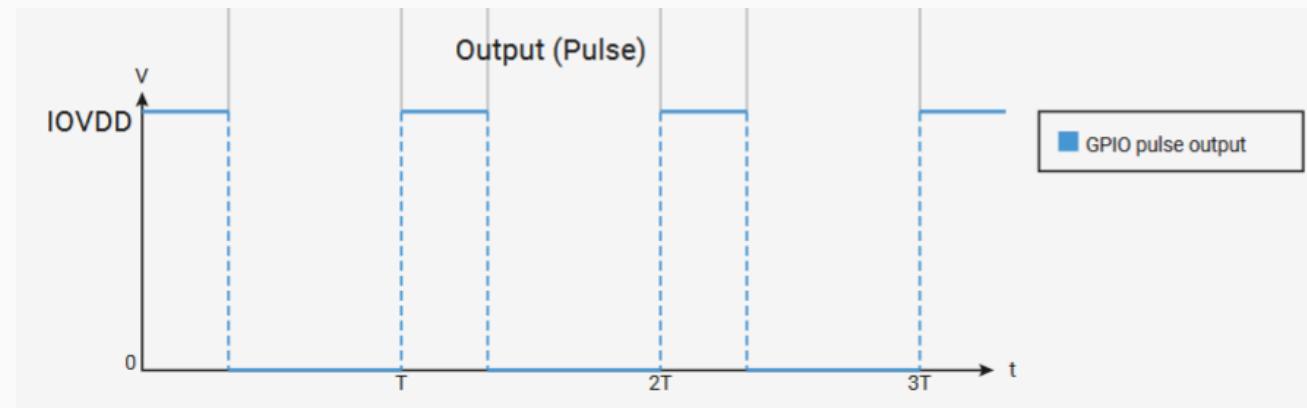
おめでとう

組み込み Rust の学習がおわりました

PWM を使ってみよう

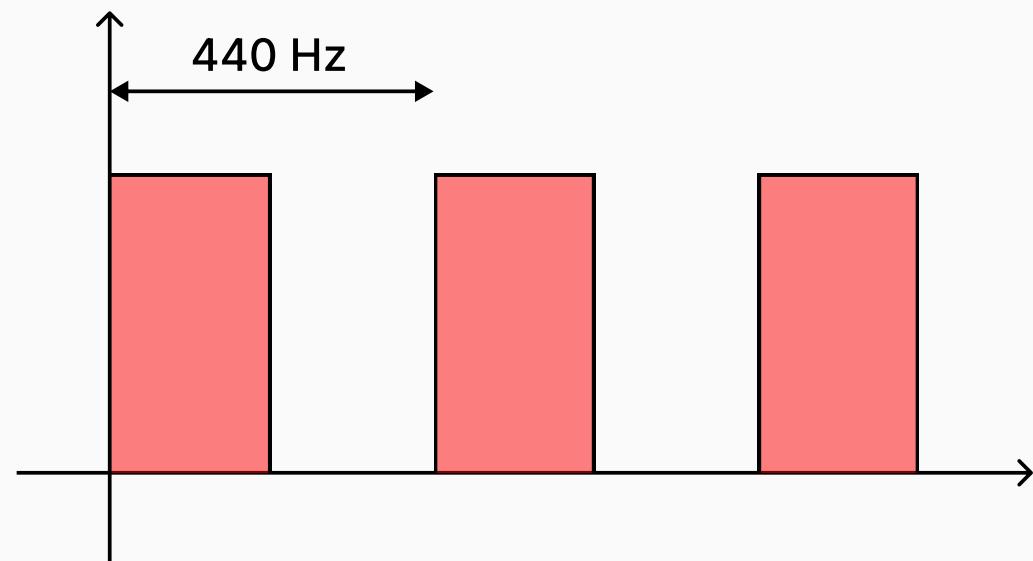
PWM とは

- Pulse Width Modulation (PWM)
 - オンとオフの繰り返しスイッチングを行い、出力される電力を制御
 - 使用例
 - LED の明るさを調整
 - 人間には高速点滅していても気づかず、暗く見える
 - モーターの回転数を調整
 - サーボモーターの角度を調整



PWM で音を鳴らそう

- PWM で音を鳴らすには、周波数とデューティ比を調整する
 - 周波数: 音の高さ
 - デューティ比: 音の長さ
- ラ(440 Hz)を鳴らす
- RP2040 のデータシートとコードと `rp2040_hal` クレート[\[link\]](#) を見ながら理解しよう!!



RP2040 の PWM の構成

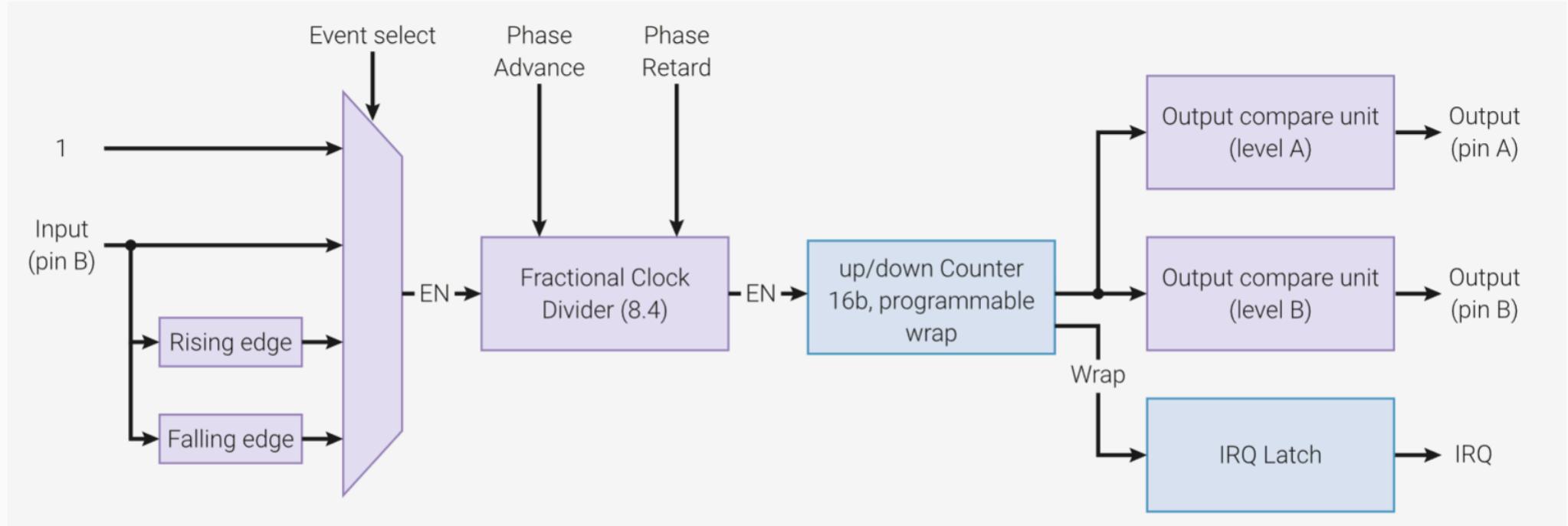


図 10: RP2040 の PWM の構成

PWM である GPIO を使うには

GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B		

図 11: RP2040 の PWM のチャンネルと GPIO の関係

- GPIO 0 を使う場合。CHANNEL 0A を使えばよい

PWM の周波数を設定する

- The **TOP** register
- Whether phase-correct mode is enabled (**CSR_PH_CORRECT**)
- The **DIV** register

図 12: PWM で設定できるパラメータ

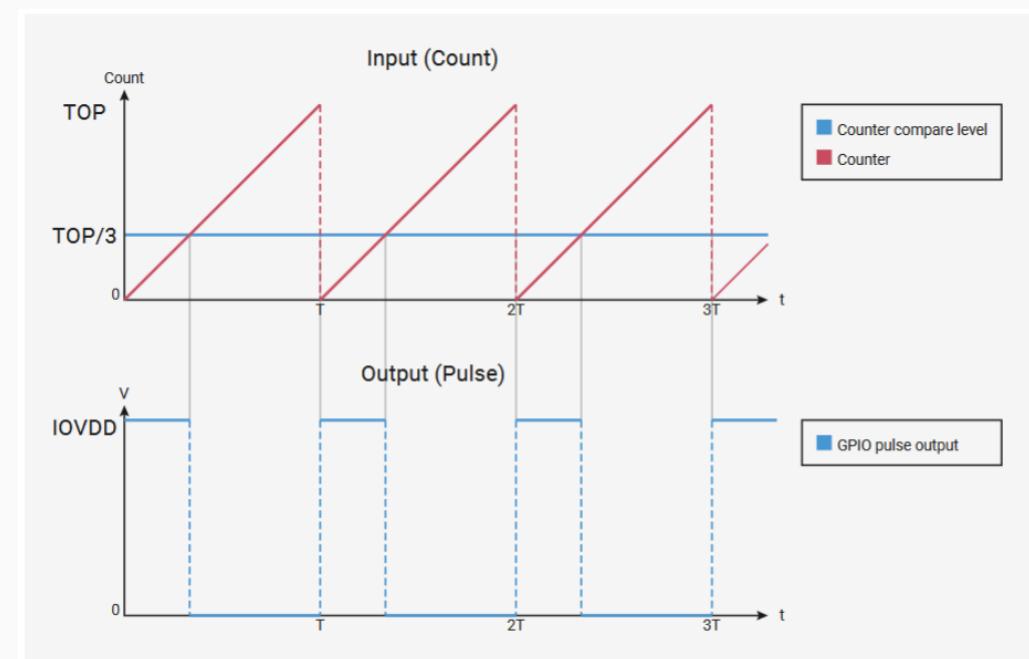


図 13: TOP の設定

PWM の周波数を設定する

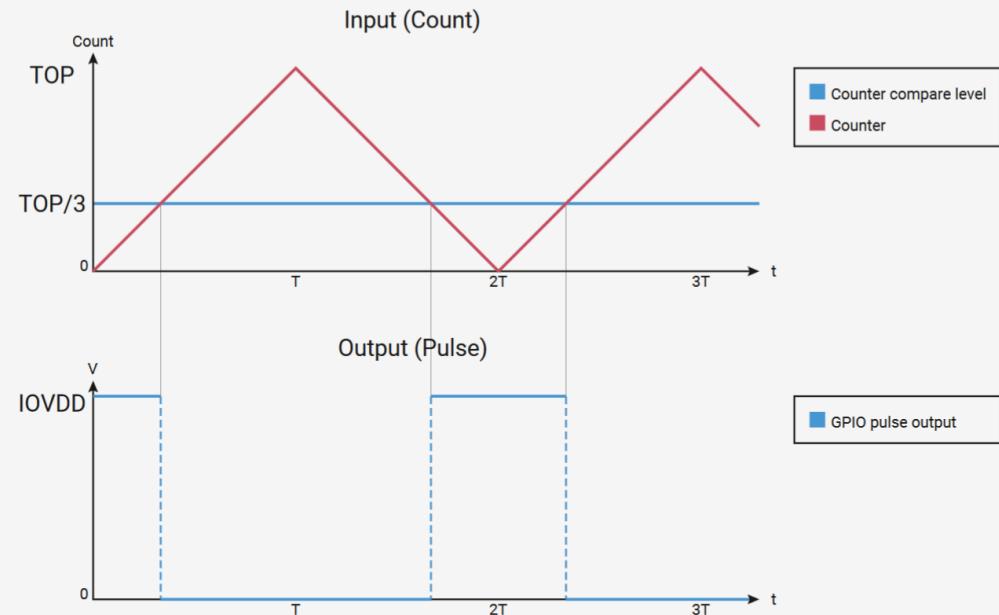


図 14: CSR_PH_CORRECT の設定

$$\text{period} = (\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \left(\text{DIV_INT} + \frac{\text{DIV_FRAC}}{16} \right)$$

The output frequency can then be determined based on the system clock frequency:

$$f_{\text{PWM}} = \frac{f_{\text{sys}}}{\text{period}} = \frac{f_{\text{sys}}}{(\text{TOP} + 1) \times (\text{CSR_PH_CORRECT} + 1) \times \left(\text{DIV_INT} + \frac{\text{DIV_FRAC}}{16} \right)}$$

図 15: 周波数と周期

PWM の周波数を設定する

$$f_{\text{PWM}} = 440 \text{ Hz}$$

$$f_{\text{sys}} = 125 \text{ MHz}$$

ここから最適なパラメーターの値は、

TOP = 28488

CSR_PH_CORRECT = false

DIV_INT = 10

DIV_FRAC = 0

And more…

debug をするには

elf2uf2-rs では、print デバッグは簡単に使えない

- 使うには空いているピンや USB などを用いて通信を行う

probe-rs [link]を使うと便利⁹

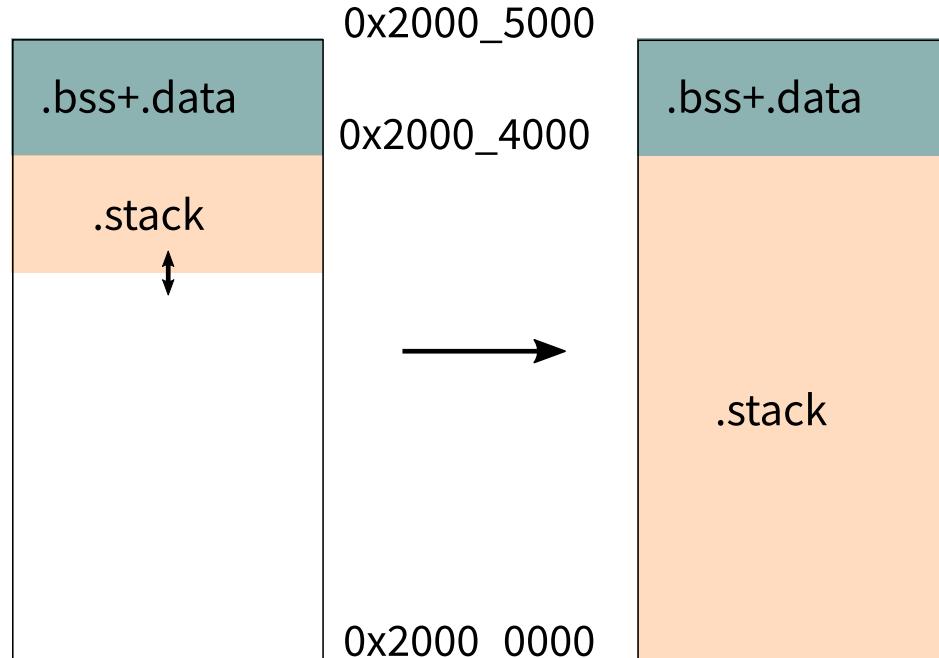
⁹github のスターが少ないのでなぜ

さらば no_std、こんにちは std

組み込み Rust では高度なことをやりたいなら、マイコンは RP2400 より **ESP32**

- std 環境でのプログラミングができるため
 - 例) 簡易的な温度計サーバーを建てる
 - 例) bluetooth 通信をする
- std 環境でのプログラミング 「The Rust on ESP Book」 [\[link\]](#)

スタックとヒープの話をしたので、flip-link を語りたい



flip-link を使わないと

1. 普通は stack は下から積み上がる
2. heap 領域を破壊
3. どのように動作するのか謎になる

flip-link を使うと

1. stack は上から積み上がる
2. メモリの端までいくと **エラーが出る**

おすすめの書籍やサイト

- 基礎から学ぶ組込み Rust (著者：中林 智之／井田 健太)
 - 聖書です。読んでください。
- The Embedded Rust Book(日本語版) [\[link\]](#)
 - 組み込み Rust ならではの機能がまとまっている
- Interface 2023 年 5 月号特集 質実剛健 Rust 言語
 - 最新の情報が載っているし、`std` のことが書かれている貴重な文献
- awesome-embedded-rust [\[link\]](#)

次にやること

- UART 通信 rp2040-hal [[link](#)]
- core::fmt::Write [[link](#)]で UART 通信で文字列を送信
- I2C 通信 rp2040-hal [[link](#)]
- SPI 通信 rp2040-hal [[link](#)]
- PWM 出力 rp2040-hal [[link](#)]
- タイマー rp2040-hal [[link](#)]
- USB rp2040-hal [[link](#)] usb_device [[link](#)]
- ADC rp2040-hal [[link](#)]
- heapless [[link](#)] で Vec が使える
- embedded-graphic [[link](#)]で display 表示 (DrawTarget, Drawable)
- cortex-m [[link](#)]で割り込み処理やスリープ
- 組み込み OS の TOCK や組み込み Linux など