

組み込み Rust

組み込み Rust 編

JIJINBEI

2024-07-29

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 の解説
5. And more…

組み込みとは

組み込みとは

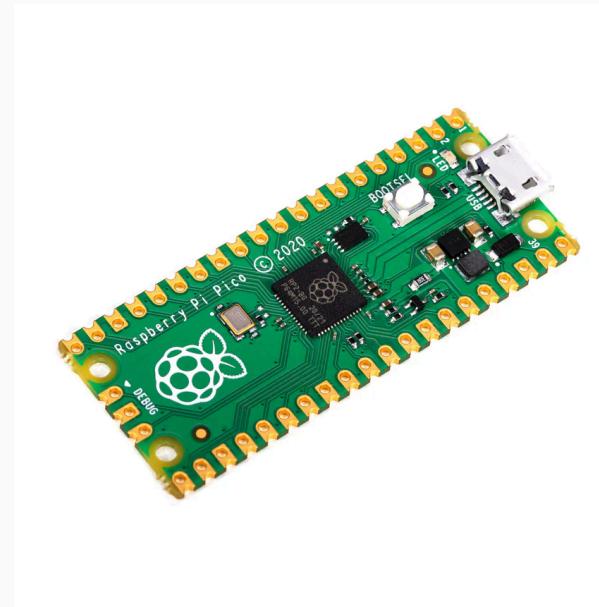


図 1: Raspberry Pi Pico

- 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

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

どういうときに unsafe か

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 の解説

組み込み Rust をやるのに知るべきこと

- 優先度 高
 - ▶ データシートを読む
 - ▶ `#![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();`
- 優先度 中
 - ▶ 組み込み用語
 - `Peripherals`
 - `WATCHDOG`
 - `clocks`
 - `cortex`

組み込み Rust をやるのに知るべきこと

- SIO
- ▶ rp_pico のクレートの中身
- ▶ defmt クレート

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？

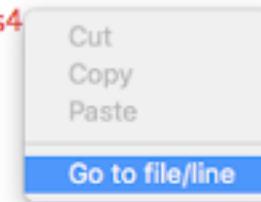
panic_halt

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
```

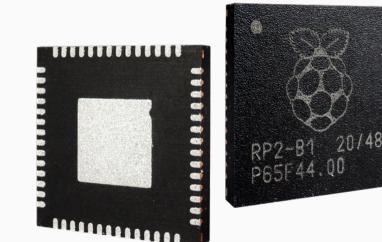
- rp_pico [link] はラズピコの **BSP(Board Support Package)** クレート
 - コードを見てみると、すごく簡単に書かれている(ほかのボードも) [link]

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

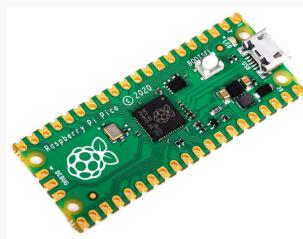
組み込み機器の抽象化

`rp_pico::hal = rp2040_hal`

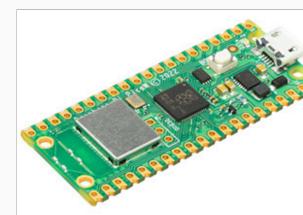


RP2400

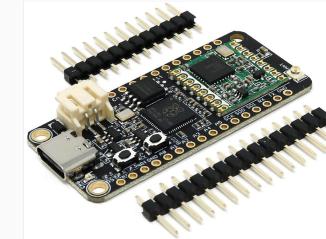
HALがハードウェア
の差分を吸収



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

rp_pico::hal と embedded_hal

rp2040_hal クレート [link] に次のような記述がある

「This is an implementation of the embedded-hal traits for the RP2040 microcontroller」

- embedded_hal は、マイコンの抽象化レイヤー

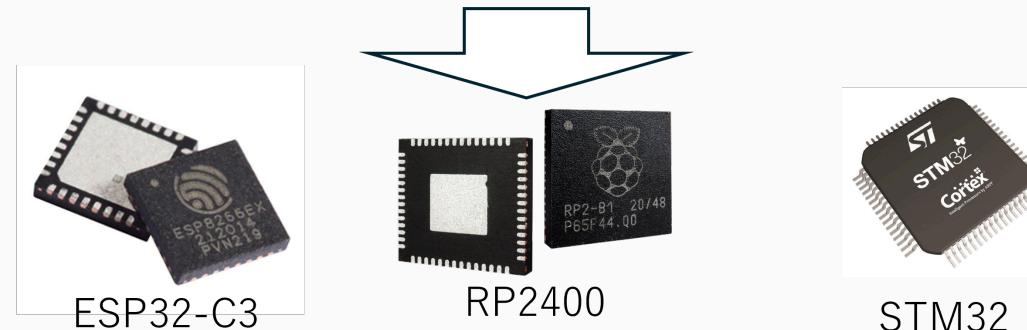
例えば、[link] には、embedded_hal に準拠したマイコンのクレートがある

embedded_hal で継承された BSP を使うことになるので、プログラミングをする際には意識することが多い。

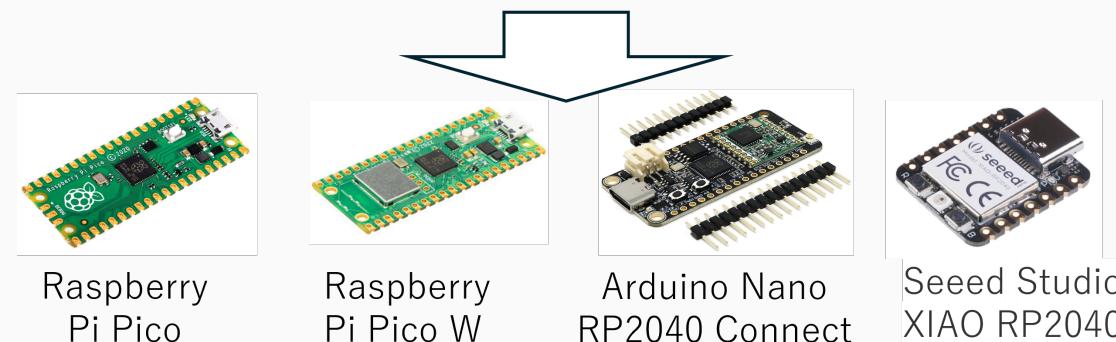
言い換えると、embedded_hal さえ理解すれば、ほかのマイコンでも同様にプログラミングができる

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

所有権をうまく使う

```
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？

所有権をうまく使う

```
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？

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

所有権をうまく使う

```
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？

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

△ 今回の場合、pac.IO_BANK0 などが、ほかのところでは使えなくなっている(pins2 は実装できない)

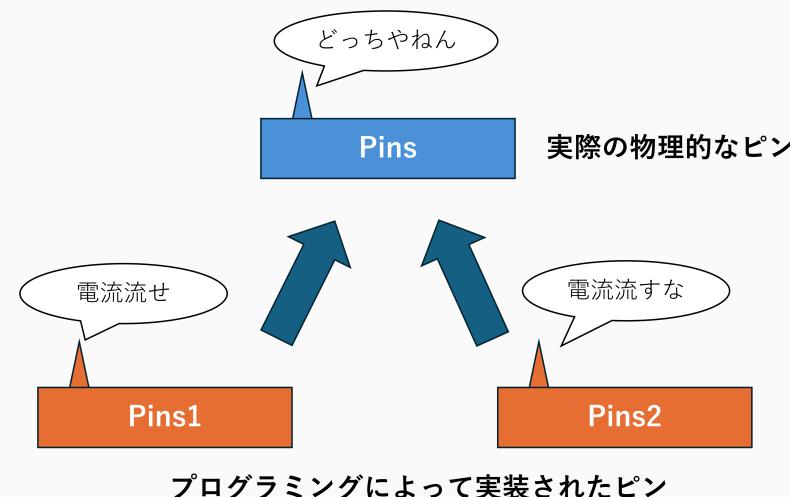
所有権をうまく使う

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

所有権システムを使って、Peripheral が使われているのが一つだけであることを保証

メリット

- ・同時アクセスによるデータ競合を防ぐ



所有権をうまく使う

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

pinsの中から led を取り出して、push_pull_output に変換しているが、
push_pull_output とは何だろうか？

所有権をうまく使う

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

`pins`の中から `led`を取り出して、`push_pull_output`に変換しているが、`push_pull_output`とは何だろうか？

LEDを点灯させるために出力モードを指定している。

- `push_pull_output`は、出力モード
 - ▶ LEDを光らす
 - ▶ モーターを動かす
 - ▶ 電流を流す
- `into_pull_up_input()`や `into_pull_down_input()`は、入力モード
 - ▶ スイッチを押したときに反応する

所有権をうまく使う

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

ここで、重要なのは型である。

```
let mut led_pin: Pin<Gpio25, FunctionSio<SioOutput>, ...> = pins.led.into_push_pull_output();
```

詳しく書くと `Pin<Gpio25, FunctionSio<SioOutput>, PullDown>` となっている。

これによって、`led_pin` は型によって出力モードであると制限される

→ 出力モード専用のメソッドしか使えない

- `led_pin.set_high()` や `led_pin.set_low()`

And more…

組み込み Rust の難しいところ

1. どのクレートにどの機能があるか？

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
- probe-rs [\[link\]](#)でデバッグ
- embedded-graphic [\[link\]](#)で display 表示 (DrawTarget, Drawable)
- cortex-m [\[link\]](#)で割り込み処理やスリープ
- std 環境でのプログラミング 「The Rust on ESP Book」 [\[link\]](#)
- 組み込み OS の TOCK や組み込み Linux など