

# LT 会

## 組み込み Rust

---

JIJINBEI

2024-07-25

HiCoder

# Table of contents

1. 組み込みとは
2. Rust とは
3. なぜ組み込みで Rust
4. 組み込み Rust のデモ
5. 組み込み Rust のデモの要点
6. And more…

# 組み込みとは

---

# 組み込みとは

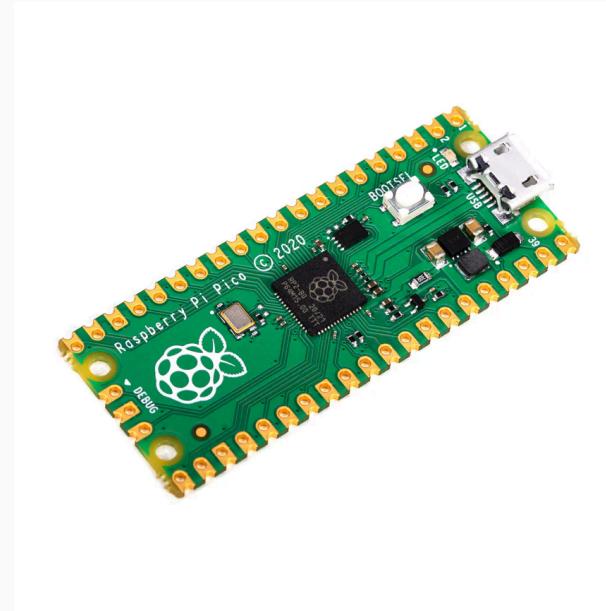


図 1: Raspberry Pi Pico

- 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
- aaaaaaa

# 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)

# Rust とは

---

# Rust の特徴

- 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
  - VM を使わない
  - 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



# 所有権

## 所有権の例(変数の所有権の移動)

```
1 fn main() {  
2     let s1 = String::from("hello");  
3     let s2 = s1; // 変数 s1 から変数 s2 に所有権の移動が発生  
4  
5     // 所有権が移動しているので、変数 s1 を利用するとコンパイルエラーが起きる  
6     // println!("{} , world!", s1); // value borrowed here after move  
7  
8     println!("{} , world!", s2);  
9 }
```

s1 は、値と所有権を持っているが、s2 に所有権を移動すると、s1 は利用できなくなる。

値は一つであることが保証される

# 所有権

## 所有権の例(関数の所有権の移動)

```
1 fn main() {  
2     let s = String::from("hello");  
3     takes_ownership(s);  
4     // println!("{}", s); // コンパイルエラーが起きる  
5 }  
6  
7 fn takes_ownership(some_string: String) {  
8     // s の所有権が関数に移動  
9     println!("{}", some_string);  
10 }
```

rust

some\_string が所有権を持っているので、s は利用できなくなる

値は一つであることが保証される<sup>1</sup>

---

<sup>1</sup> この利点は後ほど…

# 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  
コミュニティが小さい

# 組み込み Rust のデモ

---

# Rust のコード

rp-pico クレートの pico\_blinky.rs から抜粋。

Raspberry pi pico を 0.5 秒毎に LED を点滅させる

```
1  #![no_std]
2  #![no_main]
3
4  use rp_pico::entry;
5
6  use embedded_hal::digital::OutputPin;
7  use panic_halt as _;
8
9  use rp_pico::hal::prelude::*;
10
11 use rp_pico::hal::pac;
12 use rp_pico::hal;
13
14 #[entry]
15 fn main() -> ! {
16     let mut pac = pac::Peripherals::take().unwrap();
17     let core = pac::CorePeripherals::take().unwrap();
```

rust

# Rust のコード

```
18
19     let mut watchdog = hal::Watchdog::new(pac.WATCHDOG);
20
21     let clocks = hal::clocks::init_clocks_and_plls(
22         rp_pico::XOSC_CRYSTAL_FREQ,
23         pac.XOSC,
24         pac.CLOCKS,
25         pac.PLL_SYS,
26         pac.PLL_USB,
27         &mut pac.RESETS,
28         &mut watchdog,
29     )
30     .ok()
31     .unwrap();
32
33     let mut delay = cortex_m::delay::Delay::new(core.SYST,
34         clocks.system_clock.freq().to_Hz());
35
36     let sio = hal::Sio::new(pac.SI0);
37
38     let pins = rp_pico::Pins::new(
39         pac.IO_BANK0,
```

# Rust のコード

```
39     pac.PADS_BANK0,
40     sio.gpio_bank0,
41     &mut pac.RESETS,
42 );
43
44 let mut led_pin = pins.led.into_push_pull_output();
45
46 loop {
47     led_pin.set_high().unwrap();
48     delay.delay_ms(500);
49     led_pin.set_low().unwrap();
50     delay.delay_ms(500);
51 }
52 }
```

ここで、本質的な部分はどこか？

# Rust のコード

ここで、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

# Rust のコード

ここで、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？

# Rust のコード

ここで、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 のデモの要点

---

# 重要な部分

- 優先度 高
  - ▶ `#![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`
    - `SIO`

# 重要な部分

- ▶ rp\_pico のクレートの中身

`#![no_std]`

Rust の標準ライブラリを使わず、OS を使わないことを示す。

標準ライブラリー一覧の [\[link\]](#)

std で使えないもので一番困るもの

`Vec<T>` [\[link\]](#)

`Vec` が使えない、`String` 型も使えない、文字列を扱うのが難しい<sup>2</sup>

---

<sup>2</sup>使えないのは、ヒープ領域を用意する必要(`malloc`)があり、組み込みではメモリが少ないため

## `#![no_main]` と `#[entry]`

OSがない環境では、`main` 関数は使えず、エントリーポイント(プログラムが始まるところ)を自分で指定する

今回の例では、`#[entry]`で `main` 関数を指定している

# 優先度 高

```
use panic_halt as _;
```

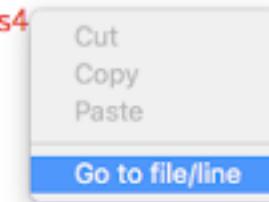
エラーとはどういうものを思い浮かべるか？

# 優先度 高

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

```
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)

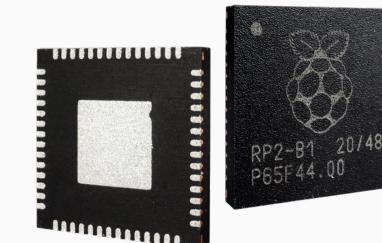
## rp\_pico::hal と embedded\_hal

HAL(Hardware Abstraction Layer) とは、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吸収

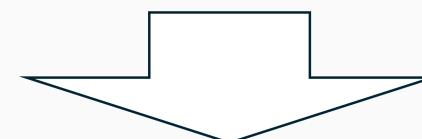
rp\_pico [link] は Raspberry Pi Pico のハードウェアのクレート

- rp\_pico のクレートに 「it re-exports the rp2040\_hal crate」 という記述がある
  - rp2040\_hal [link] は Raspberry Pi Pico のマイコンのクレート
  - RP2400 とは、Raspberry Pi Pico のマイコンのこと
  - Raspberry Pi Pico はボード全体のことで **BSP (Board Support Package)** と呼ばれる

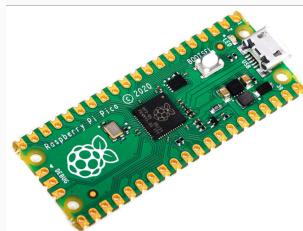
## rp\_pico::hal と embedded\_hal



RP2400



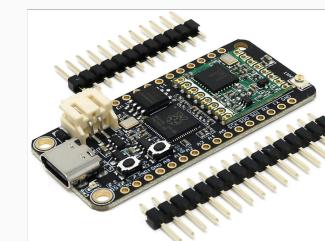
HALがハードウェア  
の差分を吸収



Raspberry Pi Pico



Raspberry Pi Pico W



Arduino Nano  
RP2040 Connect



Seeed Studio XIAO  
RP2040

## 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」

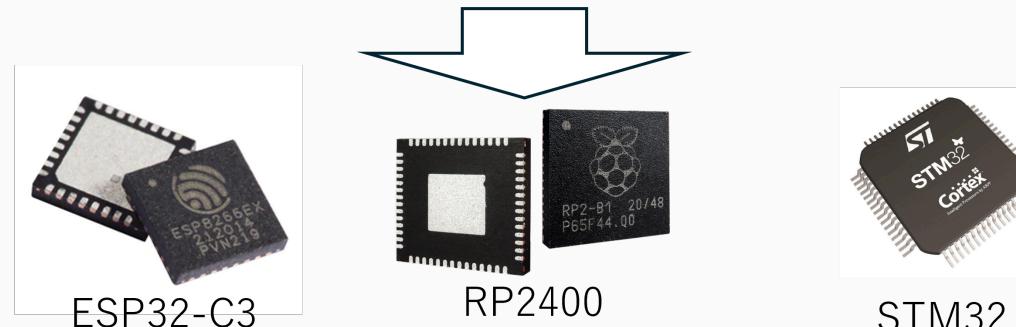
- embedded\_hal は、マイコンの抽象化レイヤー

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

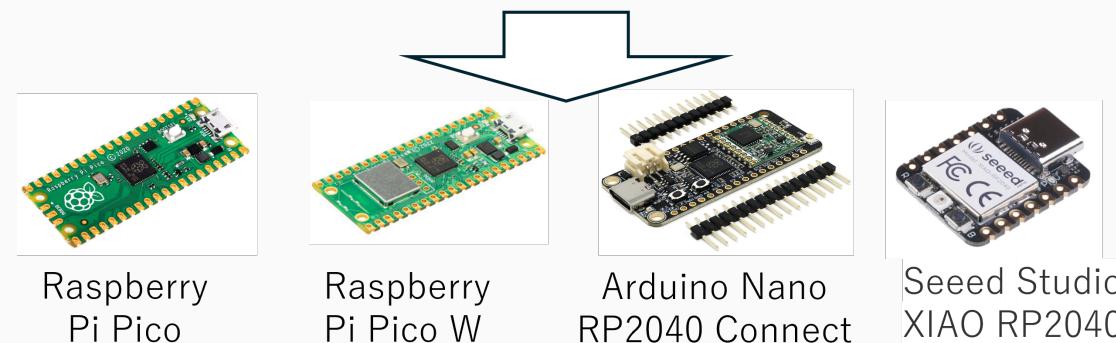
embedded\_hal で継承された BSP を使うことになるので、プログラミングをする際には意識することが多い。

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

## 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

# 優先度 高

```
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？

優先度 高

# 優先度 高

```
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

優先度 高

```
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

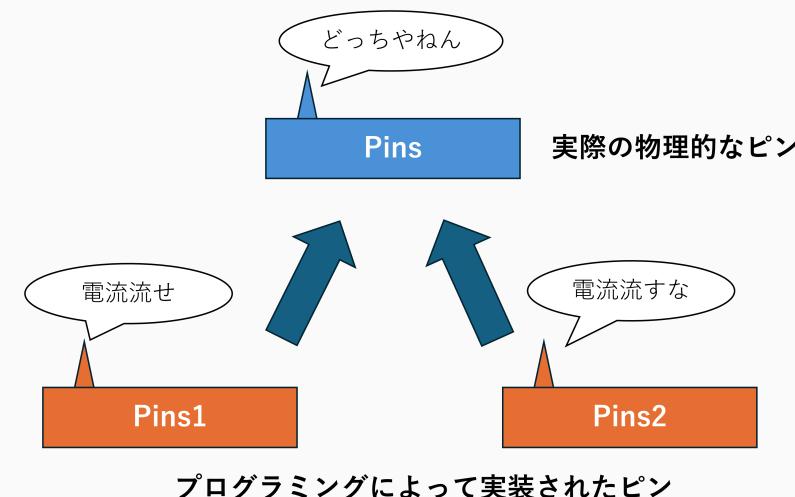
△ 今回の場合、pac.IO\_BANK0 などが、ほかのところでは使えなくなっている(pins2 は実装できない)

```
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



```
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？

```
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

```
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…

---

# 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
- データシートを読む
- std 環境でのプログラミング 「The Rust on ESP Book」 [\[link\]](#)

## 次にやること

- 組み込み OS の TOCK や組み込み Linux など