RustのEmbassyの紹介

低レイヤーズさっぽろ #1

自己紹介

- kitaharata (Takuro Kitahara)
- kitaharata.github.io
- JavaScript, TypeScript, Python, Go, PHP, C#, Ruby, Java, Kotlin, Scala, Rust, SQL
- Rust: Actix, Tokio, Hyper, Axum

What Embassy

- https://github.com/embassy-rs/embassy.git
- Rustと非同期処理(async/await)を利用した、最新の組み込みフレームワーク。
- embassy.dev

The next-generation framework for embedded applications

Write safe, correct and energy-efficient embedded code faster, using the Rust programming language, its async facilities, and the Embassy libraries.

Get started

Rust + async 💔 embedded

The Rust programming language is blazingly fast and memory-efficient, with no runtime, garbage collector or OS. It catches a wide variety of bugs at compile time, thanks to its full memory- and thread-safety, and expressive type system.

Rust's async/await allows for unprecedently easy and efficient multitasking in embedded systems. Tasks get transformed at compile time into state machines that get run cooperatively. It requires no dynamic memory allocation, and runs on a single stack, so no per-task stack size tuning is required. It obsoletes the need for a traditional RTOS with kernel context switching, and is faster and smaller than one!

```rust
use defmt::info;
use embassy_executor::Spawner;
use embassy_nrf::gpio::{AnyPin, Input, Level, Output, OutputDrive, Pin, Pull};
use embassy_nrf::Peripherals;
use embassy_time::{Duration, Timer};

// Declare async tasks
#[embassy_executor::task]
async fn blink(pin: AnyPin) {
    let mut led = Output::new(pin, Level::Low, OutputDrive::Standard);

    loop {
        // Timekeeping is globally available, no need to mess with hardware timers.
        led.set_high();
        Timer::after_millis(150).await;
        led.set_low();
        Timer::after_millis(150).await;
    }
}

// Main is itself an async task as well.
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    // Initialize the embassy-nrf HAL.
    let p = embassy_nrf::init(Default::default());

    // Spawned tasks run in the background, concurrently.
    spawner.spawn(blink(p.P0_13.degrade())).unwrap();

    let mut button = Input::new(p.P0_11, Pull::Up);
    loop {
        // Asynchronously wait for GPIO events, allowing other tasks
        // to run, or the core to sleep.
        button.wait_for_low().await;
        info!("Button pressed!");
        button.wait_for_high().await;
```

Why Embassy

- 効率性: async/awaitにより、CPUがアイドル状態のときはスリープモードに入り、低消費電力を実現。
- 応答性: 割り込みベースでタスクがウェイクアップされるため、イベントに対して迅速に応答可能。

Why Embassy

- コードの簡潔性: 複雑な状態管理やコールバック地獄を避け、直線的なコードで非同期処理を記述可能。
- 安全性: Rustの型システムと所有権モデルにより、データ競合やメモリ安全性の問題を防ぐ。

Why Embassy

- 再利用性: embedded-halトレイトの実装により、多くのサードパーティドライバとの互換性を持つ。
- 豊富なエコシステム: HAL、ネットワークスタック、USBスタック、ブートローダーなど、組込み開発に必要な多くのコンポーネントを提供。

効率性

async/awaitによる省電力

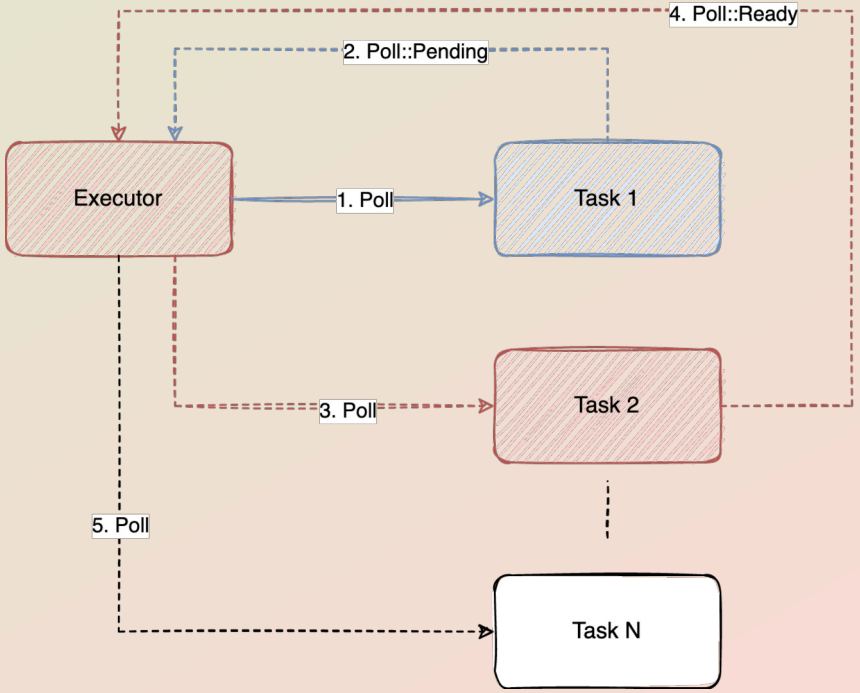
効率性

- CPUがアイドル状態のとき、Embassyのエグゼキュータは効率的にスリープモードへ移行する。
- async/await構文により、タスクが必要な処理を行うまでCPUリソースを解放。

効率性

- 結果として、大幅な低消費電力を実現し、特にバッテリー駆動のデバイスに適している。

#[embassy_executor::main]



応答性

割り込みベースの迅速なイベント処理

応答性

- Embassyは、ハードウェア割り込みをトリガーとして非同期タスクをウェイクアップ（再開）させる。
- イベント発生からタスクの処理開始までの遅延を最小限に抑える。

応答性

- リアルタイム性が要求されるアプリケーションにおいても、外部イベントに対して迅速かつ的確に応答可能。

コードの簡潔性

async/awaitによる直感的な記述

コードの簡潔性

- async/awaitを使用することで、非同期処理を同期処理のような直線的で理解しやすいコードで記述できる。
- 従来のコールバック関数を多用する複雑な状態管理や、「コールバック地獄」を回避できる。

コードの簡潔性

- コードの可読性と保守性が向上し、開発効率が高まる。

#[embassy_executor::task]

```
use embassy_executor::Spawner;
use embassy_time::Timer;
use log::*;

#[embassy_executor::task]
async fn run() {
    loop {
        info!("tick");
        Timer::after_secs(1).await;
    }
}

#[embassy_executor::main]
async fn main(spawner: Spawner) {
    env_logger::builder()
        .filter_level(log::LevelFilter::Debug)
        .format_timestamp_nanos()
        .init();

    spawner.spawn(run()).unwrap();
}
```

安全性

Rust言語によるメモリ安全とデータ競合防止

安全性

- EmbassyはRustで書かれており、Rustの強力な型システムと所有権モデルの恩恵を最大限に受ける。
- コンパイル時にデータ競合や多くのメモリ関連のバグ（ヌルポインタ、ダングリングポインタなど）を検出・防止する。

安全性

- これにより、特にリソースが限られ、安定性が重視される組込みシステムにおいて、信頼性の高いソフトウェア開発が可能。

再利用性

embedded-halによるドライバ互換性

再利用性

- EmbassyのHAL（ハードウェア抽象化レイヤ）は、標準的なembedded-halトレイトを実装している。

再利用性

- これにより、embedded-halに準拠した多くのサードパーティ製デバイスドライバ（センサー、アクチュエーター、ディスプレイなど）を容易に組み込むことができる。

再利用性

- 特定のハードウェアに依存しない、移植性の高いコード作成と、既存資産の再利用を促進する。

豊富なエコシステム

包括的な開発コンポーネント

豊富なエコシステム

- Embassyは、組込み開発に必要な様々なコンポーネントを提供し、迅速な開発を支援する。

豊富なエコシステム

- HAL (Hardware Abstraction Layers): STM32, nRF, RP2040など主要なMCUをサポート。
- embassy-net: TCP/IPネットワークスタック（Ethernet, IP, TCP, UDP, DHCPなど）。

豊富なエコシステム

- embassy-usb: デバイス側USBスタック（CDC, HIDなど）。
- embassy-boot: 電源断対応のセキュアなブートローダー。

豊富なエコシステム

- その他、Bluetooth (BLE)、LoRaWANなどのサポートも。

How Embassy

To Be Continued...?