Will Rust be the Gear of Embedded Systems?

2021/09/03
SWEST23 s4a
Tomoyuki Nakabayashi
Kenta Ida

Rustは組込みシステムの歯車になれるか？

2021年9月3日
SWEST23
中林智之
井田健太

発表者紹介（中林）

中林智之 @tnakabayashi

Nature株式会社 Firmware エンジニア

組込みRust本 基礎担当

シリーズ累計30万台突破！！

スマートリモコン Nature Remo
スマートエナジーハブ Nature Remo E

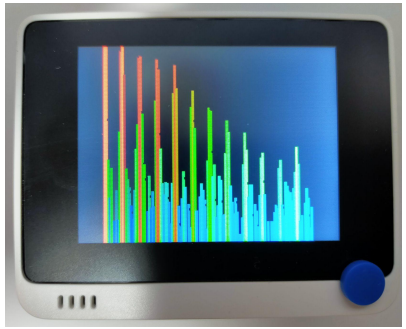
発表者紹介（井田）

井田健太 @ciniml

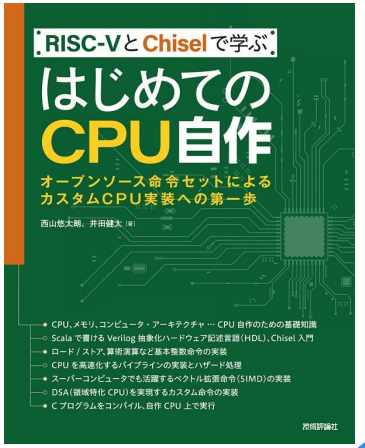
株式会社フィックスターズ FPGAエンジニア

組込みRust本 アプリ+Appendix担当

ふがぁ



こっちもちょっと
手伝いました→



Rust利用状況アンケート

アジェンダ

- Rust概要
  - 特徴, 所有権システム, etc
- 組込みRust basic
  - 組込みRustとは？, no_std, etc
- 組込みRust advanced
  - C FFI
- 組込みRust事例
  - CMSIS-DAP 実装してみた
  - ECHONET Lite ライブラリ実装してみた

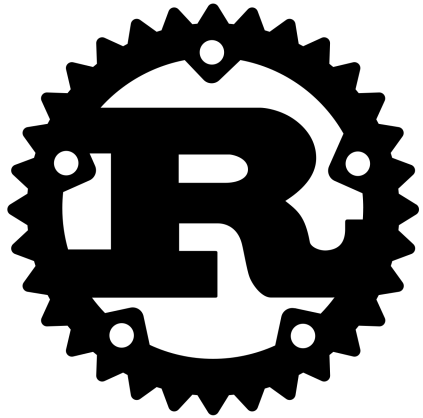
大事なこと！

- Rustの推進 ≠ C/C++の否定
- (特に)組込みRust成熟にはまだ時間がかかるし、本当に使われるようになるか、はわからない！
  - ライブラリ (BSP) とか、機能安全とか…
- ただ
  - C/C++以外の選択肢が土俵に上がった数少ないケース
  - 組込みで使う言語の議論は無駄にならない
  - 新しい言語を学ぶと新しい知見を得られる
  - 良い言語だし楽しいし、組込みの仕事でも使いたい！

Rust概要

Rust

- 2015年に version 1.0 がリリース
- システムプログラミング言語
  - OS / VM / webブラウザ / ミドルウェア
  - 組込みへの活用も期待
- 安全性とパフォーマンスを両立
- 2021年らしい言語の要素をバランス良く採用
  - 手続き型 / オブジェクト指向 / 関数型
  - ジェネリックプログラミング / 非同期 (async/await)
  - マクロ (Schemeに影響を受けている)
  - パッケージマネージャ / ビルドシステム / テストフレームワーク



歯車のロゴ

Rustの雰囲気 (コワクナイヨ

```rust
fn main() {
    let mut result: u32 = 0;
    for i in 0..10 {
        if i % 2 == 0 {
            result += i;
        }
    }
    println!("result = {}", result)
}
```

手続き型っぽく書けば
手続き型っぽいプログラムになる
(そうしないこともできる)

Rustの雰囲気 (コワクナイヨ

```
fn main() {
    let mut result: u32 = 0;
                result += i;
            }
        }
    println!("result = {}", result)
}
```

型名は変数の後ろにつける。型推論が強力なので結構省略できる。

mut をつけると変数が書き換えできる。
デフォルトでは書き換えできない。

Rustの雰囲気（関数型っぽく）

```
fn main() {
    let result: u32 = (1..10)
        .filter(|x| x % 2 == 0)
        .sum();
    println!("result = {}", result)
}
```

~~学習コストが高い~~
色々なプログラミングの要素を
学べるので1粒で何度も美味しい！

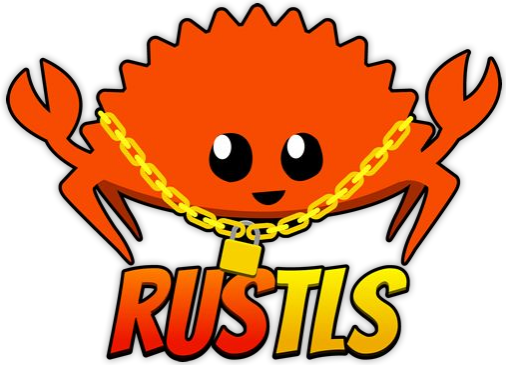
採用事例

Servo
Webエンジン

Redox
OS

firecracker
仮想マシンモニタ

rustls
TLSライブラリ

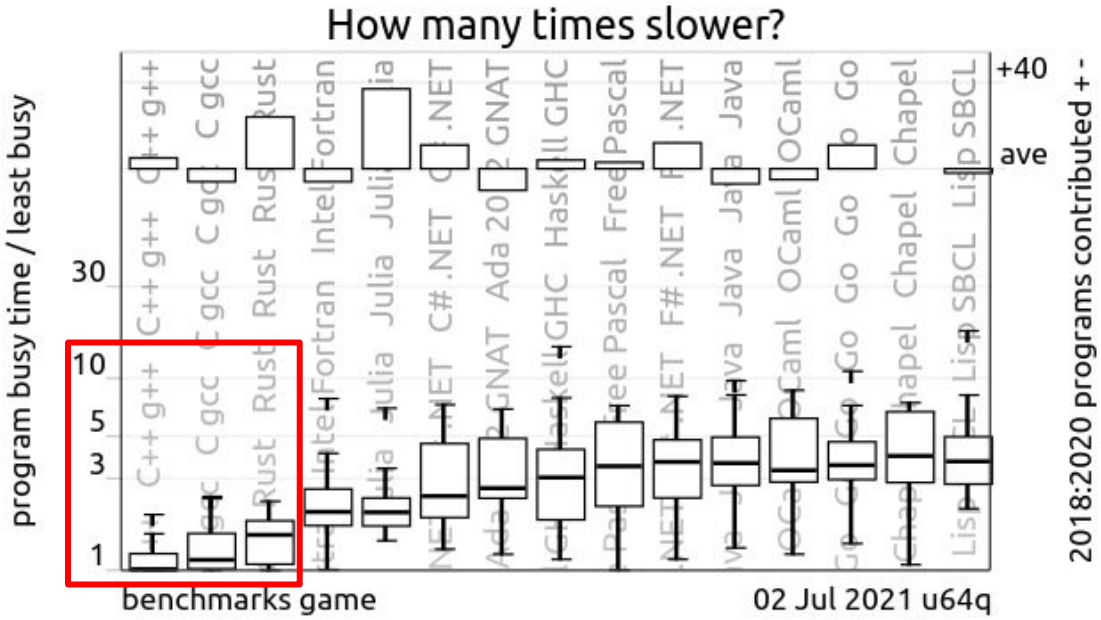


- Rustの得意分野で採用事例が増えている

Rustの特徴

- 3つを兼ね備える稀有な言語
  - **パフォーマンス**
  - **信頼性**
  - **生産性**
- さらに
  - **unsafe**な操作ができる

特徴～パフォーマンス～



C/C++と同等。ベンチマークによっては Rustの方が速い。
JavaやGoより1段階速い。

特徴～パフォーマンス～

- 機械語を出力するコンパイル型
- GC (ガベージコレクション) なしでメモリ管理
- ゼロコスト抽象化
  - 例: ポリモーフィズムを実行時コストなしで提供
- unsafeを使った人力最適化が可能
  - コンパイラが最適化不可能なメモリ操作
  - inline assembly

特徴～信頼性～

- 信頼性の高い/安全なプログラミング言語とは？
  - ✖バグゼロ
  - プログラミング言語レベルでは
    - 型安全 / メモリ安全である
  - コンパイルが通れば**わけのわからない動作（未定義動作）をしない**

コンパイルが通ったからヨシ！

例えばC/C++では

- 次のような場合わけのわからない動作になる
  - バッファオーバーラン（配列の大きさを超えた添字アクセス）
  - NULLポインタのデリファレンス
  - 安全でない暗黙の型変換や実行時の型変換
  - ダングリングポインタのアクセス（例: use after free）
  - データ競合

Rustでは

- わけのわからない動作が発生しないことを保証
  - バッファオーバーラン（配列の大きさを超えた添字アクセス）
    - →コンパイル時 / 実行時の自動検査
  - NULLポインタのデリファレンス
    - →Option型
  - 安全でない暗黙の型変換や実行時の型変換
    - 暗黙の型変換はできない
    - ダウンキャストもできない
  - ダングリングポインタのアクセス(use after free)
  - データ競合

} 所有権システムで防ぐ

所有権システムで解決できること

実行時オーバーヘッドなしに

- ダングリングポインタを作らない
  - 不正なメモリアクセスが発生しない
- 並行処理時のデータ競合を防ぐ
- (ほとんどの場合)メモリリークが発生しない
- オブジェクト同士の関係がぐちゃぐちゃになりにくい
  - 相互参照や循環参照があると急にコンパイル通すのが困難に…
  - 自然と階層構造にせざると得ない

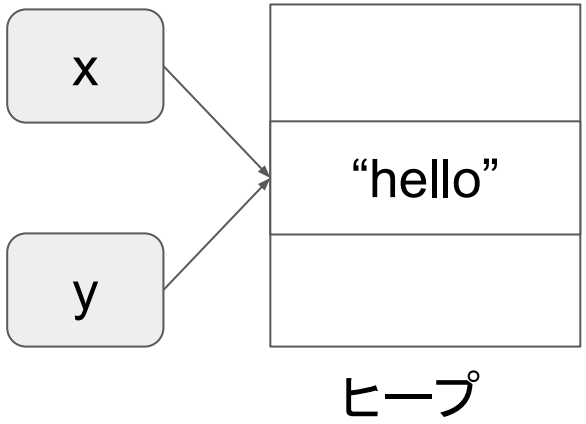
所有権システム

- 所有権
- 借用
- ライフタイム

『基礎から学ぶ組込み Rust』 Chapter3 Section11「所有権システム」参照

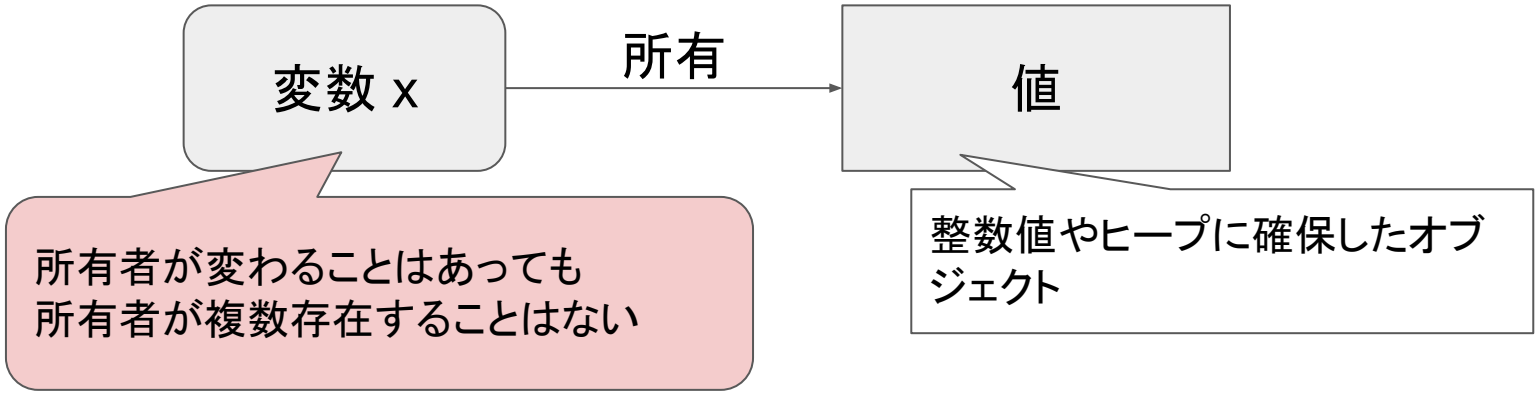
問題: 値を複数の変数からアクセスできると…

- 複数の変数から同じ値が変更される
  - データ競合！
- どこで値（メモリ）を破棄したか、して良いか、わからない
  - メモリが有効かどうかわからない、メモリリークする



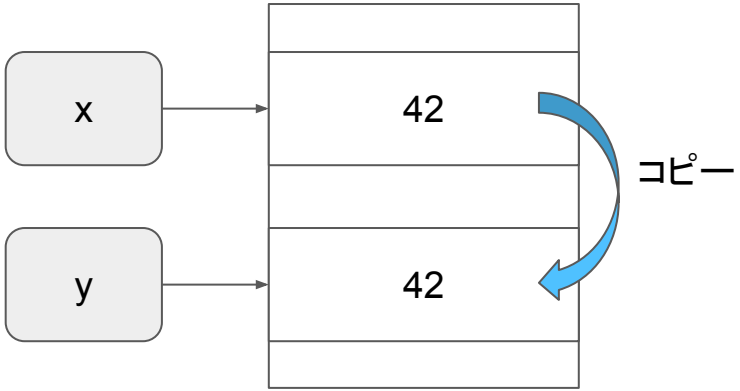
所有権

- Rustの全ての値には唯一の所有者が居る
  - データ競合が発生しない
- 所有者がスコープから外れると値は破棄される
  - ダングリングポインタを作らない、メモリリークしない



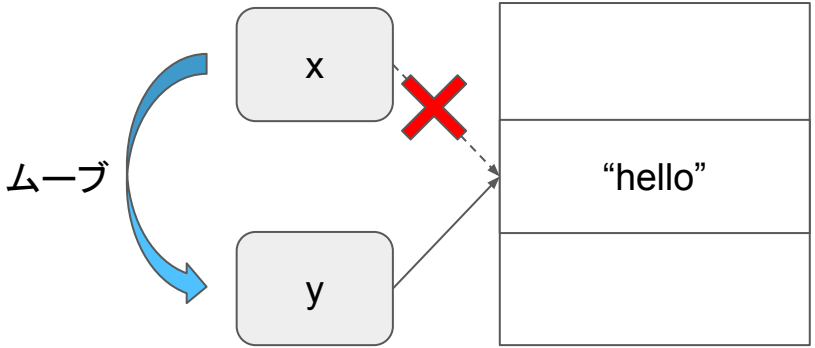
所有権のムーブ

```
let x = 42;
let y = x;
```



値を**コピー**してそれぞれの値を所有する (Copyトレイトの実装が必要)

```
let x = String::from("hello");
let y = x;
```



値はコピーせず所有権が**ムーブ**

所有者は常に唯一！

変数のスコープと値の破棄

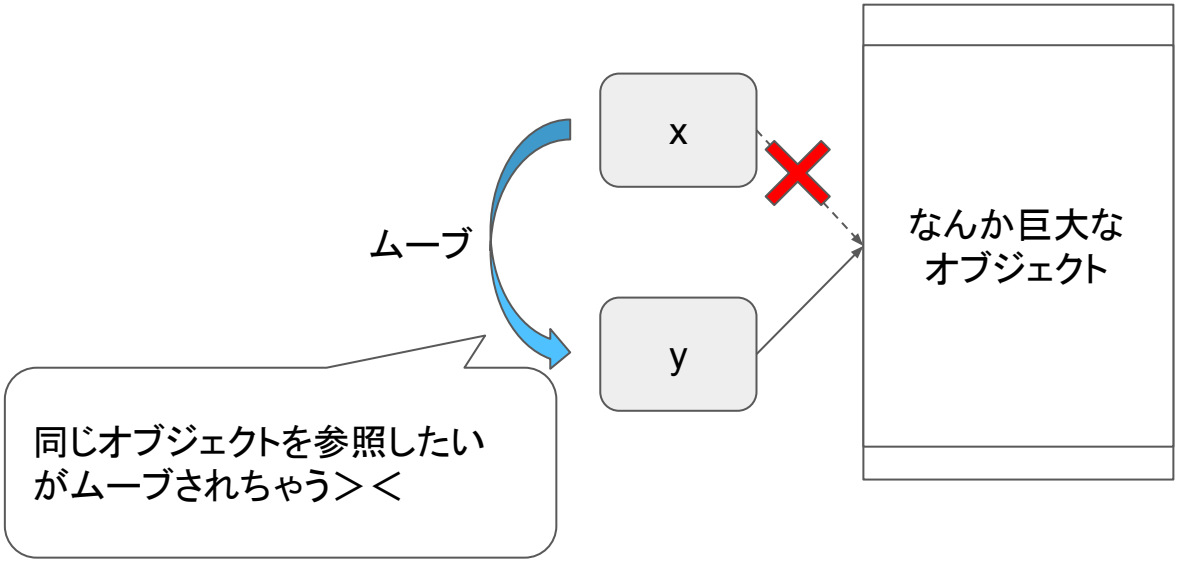
```
// xはスコープ外
{
    let x = 42;
    // xが使える
}
// xはスコープ外
// 42は破棄される
```

```
{
    let x = String::from("hello");
    // ヒープメモリ確保。xが所有者
}
// xがスコープを抜け、メモリ解放
// 以降、"hello" にはアクセスできない
```

常に唯一の所有者が存在するので値の破棄ポイントが追跡可能

借用

- 所有権だけでは値の「共有」ができない
- 大きなオブジェクトの関数引数渡しとか困る

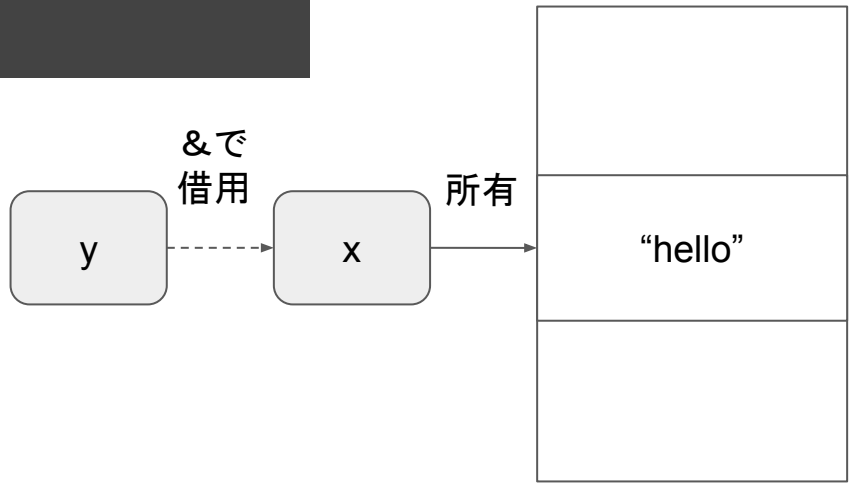


そこで
借用

借用

- 「＆」をつけると借りられる
- ただし安全性を保つために、借用ルールを守る

```
let x = String::from("hello");
let y = &x;
```

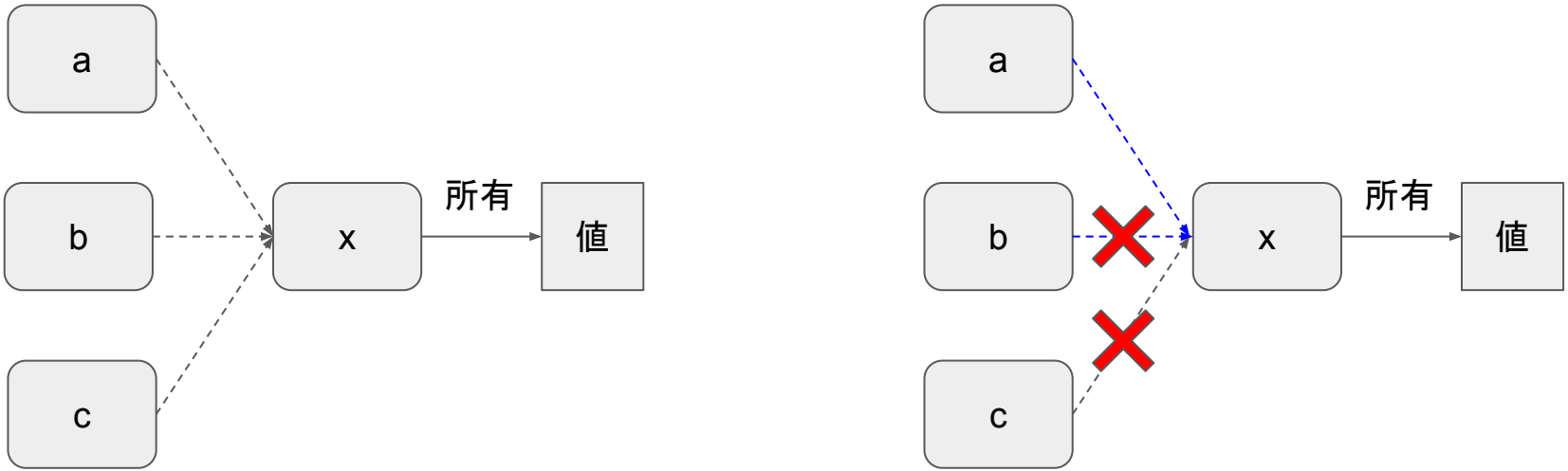


借用ルール～Shared XOR Mutable～

- 安全に値を共有するためのルール
- 共有と可変性は両立しない
  - 誰も値を変更しないなら共有しても安全！

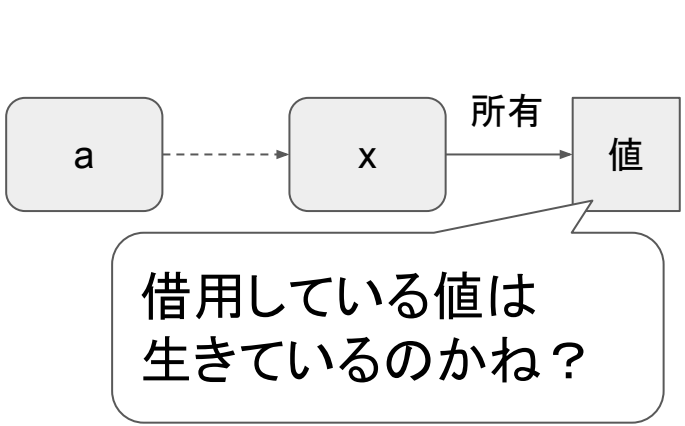
不変参照 ----------→

可変参照 ----------→



借用は良いが、値は有効なの？

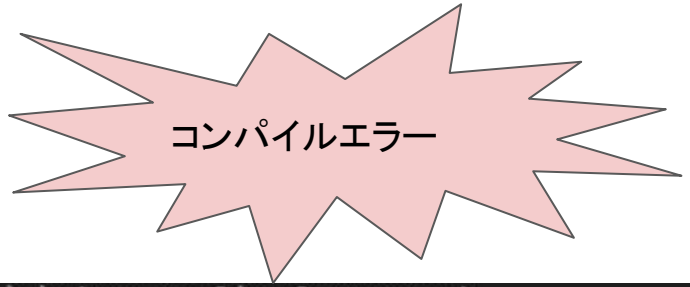
- 値を借用することで安全に値を共有できる
- 値がこっそり破棄されていたりしない？



ライフタイム

- 値が有効かどうかをコンパイラが検証する仕組み

```
let rx;
{
    let x = 42;
    rx = &x;
}
println!("rx = {}", rx);
```




```
error[E0597]: `x` does not live long enough
 --> src/main.rs:5:14
  |
5 |         rx = &x;
  |              ^^ borrowed value does not live long enough
6 |     }
  |     - `x` dropped here while still borrowed
7 |     println!("rx = {}", rx);
  |                         -- borrow later used here
```

ゆるふわ所有権システム講座おさらい

- 所有権
  - 値の所有者を唯一にすることで
    - データ競合を防ぐ
    - 所有者がスコープから外れると値を破棄する
- 借用
  - 借用ルール Shared XOR Mutable
  - 値を安全に共有できるようになる
- ライフタイム
  - 共有した値が生きていることを保証する

特徴～生産性～

- cargo
  - ビルドシステム兼パッケージマネージャ
- crate
- ジェネリクス / Trait
- 関数型
- テストフレームワーク
- ドキュメントビルダー
- コンパイル通った時の安心感

ツール周り

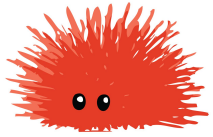
- ビルドシステムが優秀
- 親切なコンパイルエラー
- フォーマッタ: rustfmt
- lint: clippy
- language server: rust-analyzer

crate

- 3rd partyライブラリを容易に組み込める

```
[dependencies]
echonet-lite = "0.1"
```

特徴～unsafe～



- Rustコンパイラの安全保証外の操作をするとき使う
- unsafeを安全に保つのはプログラマの義務
  - Rustコンパイラとの約束！

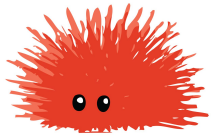
```
let num = 5;
// `生`ポインタへのキャストはsafe
let r = &num as *const i32;

unsafe {
    // 生ポインタのデリファレンスはunsafe!
    println!("{}", *r);
}
```

- ハードウェアの制御
- inline assembly
- C言語関数の呼び出し
- 高度な最適化

生ポインタ (raw pointer)
C言語のポインタ相当
- 借用ルールを無視できる
- 指し先が有効な保証はない

unsafeの誤解



- unsafeで全て台無し、は誤解
- unsafeでできることは限られている！
  - Rustのほとんどの安全性チェックはunsafeでも行われる
  - unsafe≠無保証
    - 生ポインタが割と好き放題できる、というのはある
- unsafeでできること
  - unsafe関数を呼ぶ（C言語の関数とか含む
  - 生ポインタのデリファレンス
  - 可変なグローバル変数へのアクセス
  - など

うまくunsafeとお付き合いしよう！

Rustは学習が大変？

- システムプログラミング可能な言語 & 近代的な言語機能を取り込んでいる
  - のでそういうバックグラウンドがないと大変！
- 所有権システムは困難な問題を解決している
  - 難しい問題を解いているので難しい！が恩恵が絶大
  - 実行時オーバーヘッドなしで
    - ダングリングポインタを作らず
    - ほとんどの場合不要になったメモリを最速で自動解放しつつ
    - メモリコピーを最小限に抑えることができる

組込みRust basic

話すこと

- 組込みRustとは
- no_std（ベアメタルでのRustプログラミング）
- 組込みで活きるRustの機能3選

『基礎から学ぶ組込み Rust』 Chapter5「組込みRustの基礎知識」
Chapter6「Wio Terminal搭載のデバイスを使う」

組込みRustとは

- 組込みシステム開発でRustを使うこと
- 広義: 組込みLinuxなど含む
- 狭義: OSなしのマイコン環境 (no_std)
  - 本発表では主にこちら

組込みLinux情報（2021年9月現在）

- ユーザーランドアプリケーション開発
  - 普通に使える
  - ライブラリのbindingあるかどうか、くらい
- kernel / device driver
  - 議論中
  - もうしばらく厳しい！
- Yocto
  - meta-rust / meta-rust-bin

なぜRustで組込みなのか？

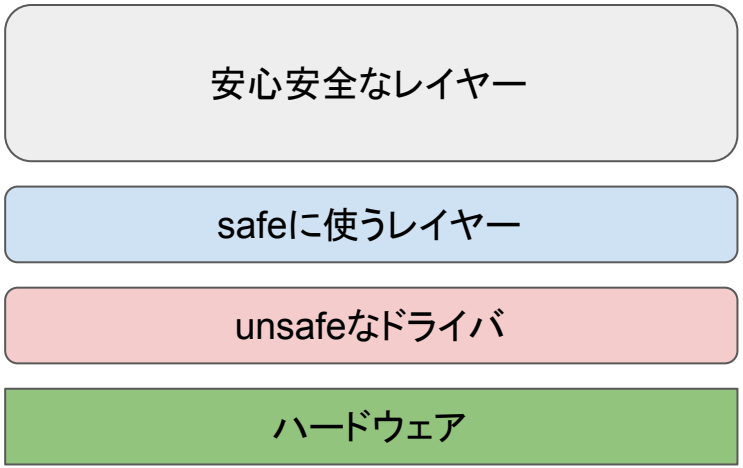
- パフォーマンス
- 信頼性
- 生産性
- unsafeなこともできる

ポテンシャルはある！資産は足りない！

- 遊べる環境が整ってきている
  - 途方に暮れる感じではなくなっている
- C/C++以外にもう1つくらい選択肢あっても良いよね
- 生きてるうちにC言語が組込みの主流でなくなる世界、見たくない？

組込みRustの疑問

- 組込みだとunsafeだらけになって旨味がないのでは？
- そうでもない
  - ハードウェア制御するところは確かに
  - unsafeコードをsafeに使えるレイヤーを用意しよう



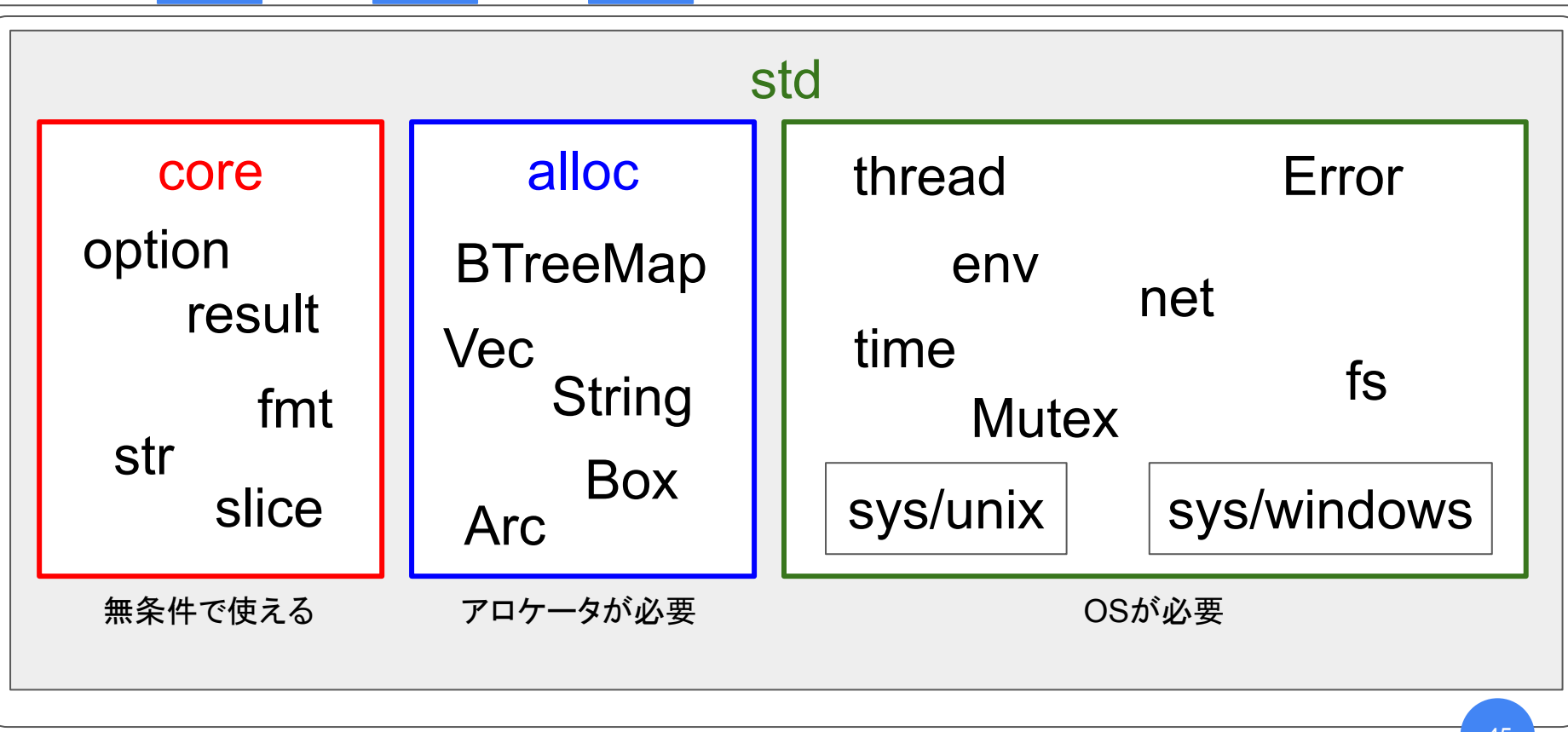
#![deny(unsafe_code)]でここの安全性は担保
大規模になるほど嬉しい

ここが危ないのはどの言語使っても同じ

no_std

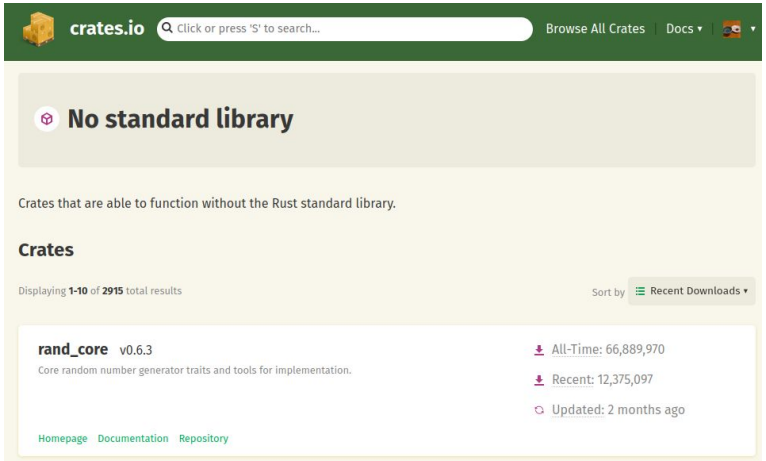
- Rustの標準ライブラリは3レベル構成
  - core
  - alloc
  - std
- stdはOSがあることを前提に実装
  - ベアメタル開発では使えない

no_std


std
core
option
result
fmt
str
slice
alloc
BTreeMap
Vec
String
Box
Arc
thread
Error
env
net
time
fs
Mutex
sys/unix
sys/windows
無条件で使える
アロケータが必要
OSが必要

意外となんとかなるno_stdプログラミング

- ベアメタルでのRustプログラミングでも
  - coreは無条件で使用可能
  - allocはメモリアロケータ実装で使用可能
- crateはno_stdで使えるものも
- gdbも普通に使える
- 組込み環境でstdの整備も
  - VxWorks
  - ESP-IDF (RISC-Vコアのみ)
    - Xtensaコアは非対応

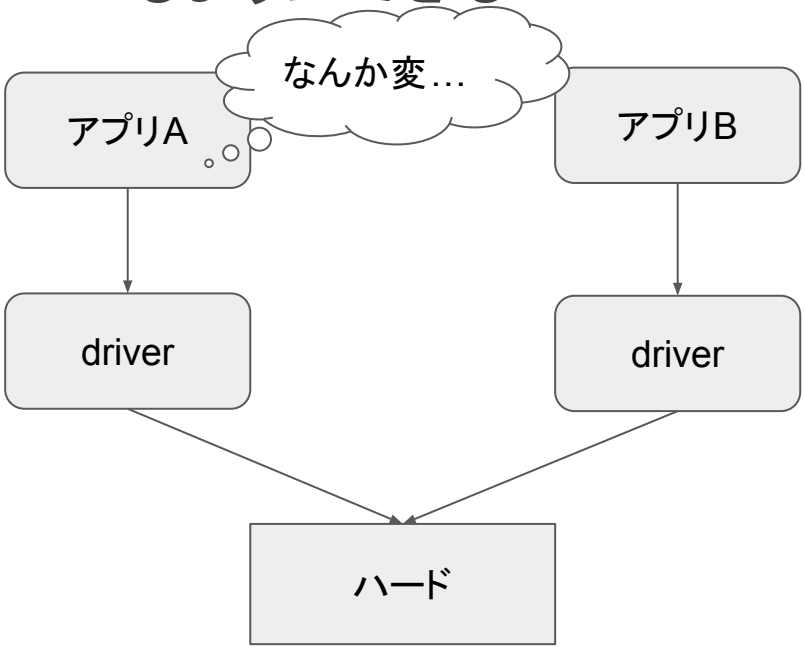


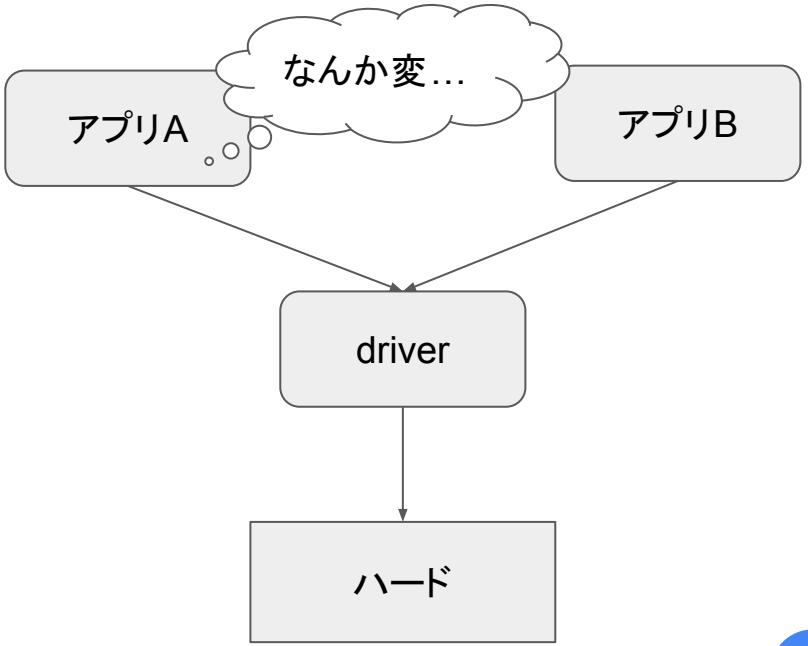
組込みで活きるRustの機能3選！

- シングルトンdriver
- 並行性
- 型状態プログラミング

シングルトンdriver

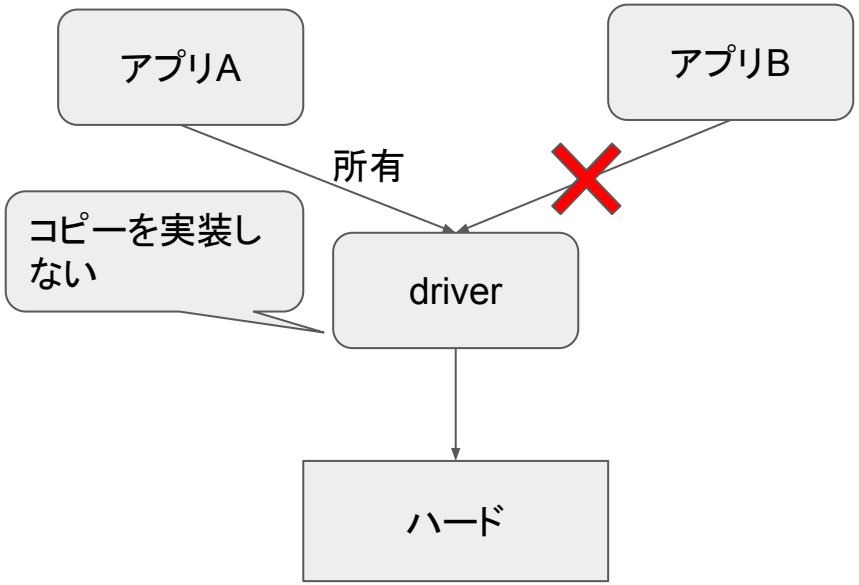
- 所有権を活用してハードウェアに対して唯一の利用者が存在するようにできる！





シングルトンdriver

- ドライバオブジェクトを所有権システムで運用



アプリBから無理な使い方する
- コンパイルエラー
- unsafeを使う

アプリBから真っ当な使い方をする
- 所有権のムーブでアプリAからは使えなくする
- 排他制御する
- うまいこと借用する

並行性

- 割り込みハンドラと資源を安全に共有
  - データの共有はバグになりやすい…
- 可変なグローバル変数へのアクセスはunsafe

```
static mut COUNTER: u32 = 0;
fn main() {
    unsafe { COUNTER += 1 };
}
```

並行性

- 安全に可変なグローバル変数を操作するには
  - アトミック命令

```
use core::sync::atomic::{AtomicUsize, Ordering};
static COUNTER: AtomicUsize = AtomicUsize::new(0);
fn main() {
    // アトミック操作命令の`fetch_add()`を使います
    COUNTER.fetch_add(1, Ordering::Relaxed);
}
```

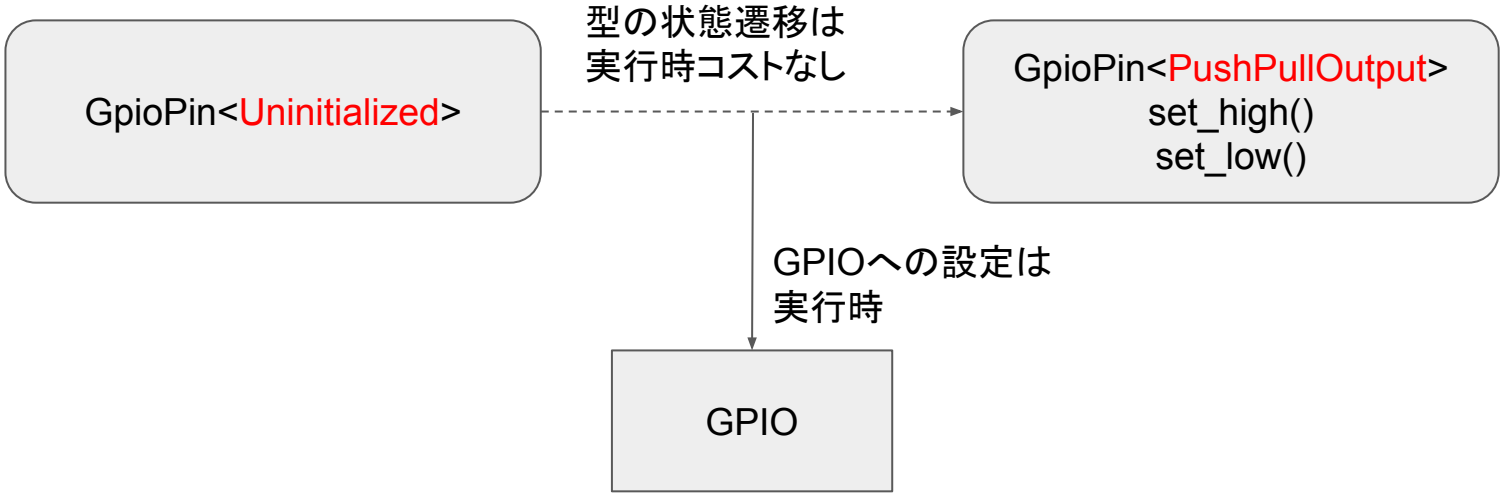
並行性

- 安全に可変なグローバル変数を操作するには
  - Mutex

```
use std::sync::Mutex;
use lazy_static::lazy_static;
lazy_static! {
    static ref COUNTER: Mutex<usize> = Mutex::new(0);
}
fn main() {
    let mut counter = COUNTER.lock().unwrap();
    *counter += 1;
}
```

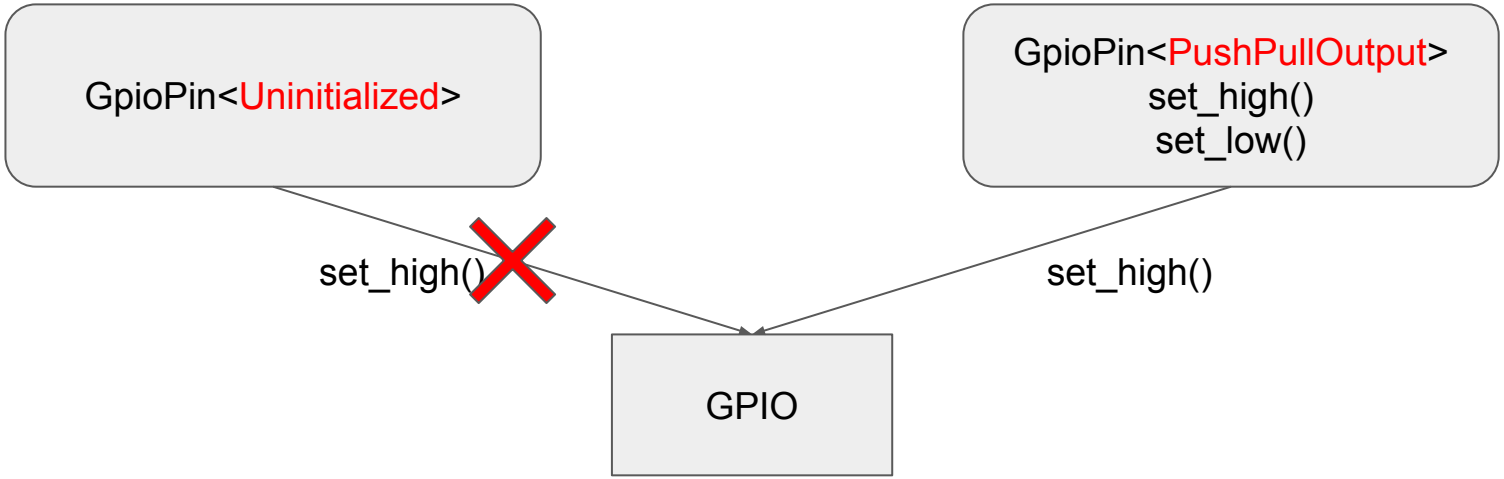
型状態プログラミング

- Rustの強い型システムを利用して状態を型で表現
  - 状態遷移 -> ある型と別の型との変換
  - 通常の状態マシンは実行時にコストがかかる



型状態プログラミング

- 操作ミスをコンパイル時に防ぐ
  - 例: 未初期化状態でGPIO出力レベルを変更するメソッド呼び出しはコンパイルエラー



型状態プログラミング

- GPIOが出力モードになっていないのに、出力レベルを変更するメソッドを呼び出すミスを防ぐ！

```
let pin: GpioPin<Uninitialized> = get_gpio_pin();
// pin.set_high(); // これはコンパイルエラー

// これは大丈夫。`PushPullOutput`な型状態にする
let mut output_pin: GpioPin<PushPullOutput> =
    pin.into_push_pull_output();
output_pin.set_high();
```

型状態プログラミング

GPIOが未初期化状態
- GPIOへの操作不可

GPIOが出力設定
- 出力レベルの変更などが可能

GPIOが未初帰化状態のときに出力レベル変更しようとするとコンパイルエラー

```
struct GpioPin<MODE> {
    pin: Pin,
    mode: MODE,
}

// GpioPinのMODEのための型状態
struct PushPullOutput;
struct Uninitialized;
```

型状態プログラミング

```
impl<MODE> GpioPin<MODE> {
    // GpioPinがPushPullOutputな型状態になる
    pub fn into_push_pull_output(self) -> GpioPin<PushPullOutput> {
        self.pin.modify(|_r, w| w.mode.out());
        GpioPin { pin: self.pin, mode: PushPullOutput }
    }
}
impl GpioPin<PushPullOutput> { // `PushPullOutput`な型状態の時だけ使用できる
    pub fn set_high(&mut self) {
        self.pin.modify(|_r, w| w.level.high());
    }
}
```

フレームワーク / OS

- embedded-hal
  - 基礎から学ぶ 組込みRustでも紹介！（ステマ
- Tock
  - 組込みOS（not RTOS）
  - BLEとか使える
- DroneOS
  - RTOS

Ferrous Systems

- (組込み)Rustのトレーニング / コンサル / ツール開発
  - https://ferrous-systems.com/
- 組込みツール開発プロジェクト knurling-rs
  - GitHub sponsorsで投げ銭できる！
  - https://github.com/knurling-rs/
- 機能安全プロジェクト Ferrocene
  - Ferrocene aims to qualify the Rust compiler at ISO 26262/ASIL-B readiness and general availability by the end of 2022.
  - https://ferrous-systems.com/ferrocene/

embedded Rust advanced

既存資産の活用

全てRustで記述するだけのリソースがあれば良いが
あまり現実的ではない

　特に組込みでは既にCで記述されている
　ライブラリなどを呼び出す必要があることが多い

無線関係の処理の場合、バイナリ提供のみの場合もある

　e.g. NordicやEspressifの無線SoCの無線部分

　当然、Cからの呼び出し前提のIFとなっている

→Cとの相互運用機能が必要

C FFI (C Foreign Function Interface)

当然ながら

RustからC言語の関数を呼び出す

C言語からRustの関数を呼び出す

の両方が可能

CからRustの関数を呼び出す

#[no_mangle] 属性とextern "C"を関数につければよい

no_mangleでマングリングの抑制

extern "C" でC ABIを使うことを指定

C側 (Rustの関数を呼び出す)

```c
void rust_main(void);
void main()
{
    //...
    rust_main (); // Rust の関数を呼び出す。
    //...
}
```

Rust側 (Cから呼び出される)

```rust
#[no_mangle]
pub extern "C" fn rust_main()
{
    // ...
}
```

RustからCの関数を呼び出す

extern "C" を関数宣言につければよい

e.g. C標準関数のwriteを呼び出す場合

Rust側 (Cの関数を呼び出す)

```
pub extern "C" fn write(fd: i32 ,
     buf: *const u8, len: usize) -> size;
```

C側 (Rustから呼び出される)

```
ssize_t write(
     int fd,
     const void* buf,
     size_t count);
```

C言語のプリミティブ型

RustとC言語のプリミティブ型は直接対応するものはない

- Rustにint型やshort型はない
- Rustのプリミティブ型は符号やサイズが明示される
- Cのプリミティブ型はサイズや符号が処理系依存

→Rustの対応する型を明示する必要がある

- e.g. Cのint = 32bit符号付き整数の環境の場合

```
pub extern "C" fn write(fd: i32 , buf: *const u8, len: usize) -> size;
ssize_t write(int fd, const void* buf,  size_t count);
```

ctyクレート

Cのプリミティブ型に対応するRustの型を定義するクレート

ctyで定義された型を使えばターゲット非依存の型定義が可能

cty::の下にCの型に対応するRustの型が定義されている

- プリミティブの場合: c_のプリフィクス + Cでの型名
- 標準ライブラリの型の場合: その型名

```
// int=c_int, void=c_void, size_t=size_t
pub extern "C" fn write(
    fd: cty::c_int, buf: *const cty::c_void, len: cty::size_t) -> size;
ssize_t write(int fd, const void* buf,  size_t count);
```

構造体

Cの構造体を扱う場合、
同等の構造を持つRustでの型定義が必要

構造体のフィールドの配置規則をCと同等にするため
#[repr(C)]属性を付けて定義する

Rustでの定義

```
#[repr(C)]
pub struct ExampleStruct {
    field0: u8,
    field1: u32,
    field2: cty::c_int,
}
```

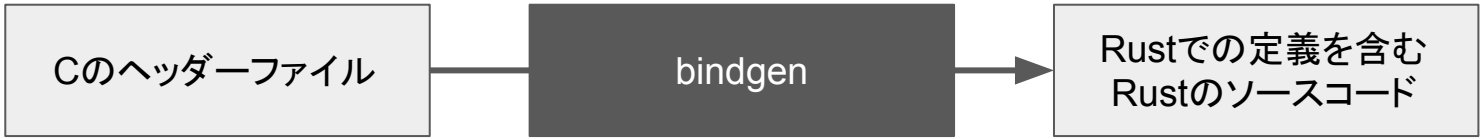
Cでの定義

```
struct ExampleStruct {
    field0: uint8_t;
    field1: uint32_t;
    field2: int;
};
```

バインディングの自動生成

RustからCの関数を呼び出すための定義(バインディング)を
手動で定義するのは手間がかかる

C言語の定義から
Rustでの定義を生成するツールとして **bindgen** がある



bindgenの使用例

bindgenをcargoのビルドスクリプト（build.rs）から呼び出して、ビルド中にバインディングのソースコードを生成

```
let bindings = bindgen::Builder::default()
    .clang_arg("-nostdinc")
    .clang_arg("-I".to_owned() +
xtensa_toolchain_path.clone().join("xtensa-esp32-elf").join("include").to_str().unwrap())
    .use_core()
    .ctypes_prefix("cty")
    .disable_untagged_union()
    .generate_comments(false)
    .rustfmt_bindings(true)
    .layout_tests(false)
    .derive_copy(true)
    .derive_debug(true)
    .derive_default(true)
    .whitelist_function(r"(esp|ESP)_.+")
    .header("wrapper.h")
    .generate()
    .expect("Unable to generate bindings");
```

bindgenの使用例

内部的にlibclangを使っているので、
対象のCソースで必要なコンパイルオプションを指定する

```
let bindings = bindgen::Builder::default()
    .clang_arg("-nostdinc")         // 標準ライブラリは含まない
    .clang_arg("-I".to_owned() +   /* 追加のインクルードパスの指定 */
xtensa_toolchain_path.clone().join("xtensa-esp32-elf").join("include").to_str().unwrap())
    .use_core()                     // stdクレートを使わずcoreクレートのみ使う (no_std)
    .ctypes_prefix("cty")           // Cの型定義を参照するときに付けるprefix (cty使う指定)
    .disable_untagged_union()       // Rustのタグ無しunionを使わない
    .generate_comments(false)       // コメントを生成しない
    .rustfmt_bindings(true)         // 生成するバインディングのソースコードをrustfmtで成形する
    .layout_tests(false)            // 構造体レイアウトのテストコードを生成しない
    .derive_copy(true)              // 構造体に#[derive(copy)]を付加してコピー可能にする
    .derive_debug(true)             // 構造体に#[derive(debug)]を付加してデバッグ出力フォーマット可能にする
    .derive_default(true)           // 構造体に#[derive(default)]を付加してデフォルト値の取得を可能にする
    .whitelist_function(r"(esp|ESP)_.+")       // バインディングを生成する関数の名前のパターン
    .header("wrapper.h")                       // 入力のCヘッダーファイルのパス
    .generate()                                // 生成を実行する
    .expect("Unable to generate bindings");    // エラーチェック
```

生成されるバインディング

```
// 関数定義の生成
extern "C" {
    pub fn esp_get_idf_version() -> *const cty::c_char;
}
// typedefでのエイリアスの変換
pub type esp_mac_type_t = cty::c_uint;
// enumの変換
pub const esp_reset_reason_t_ESP_RST_UNKNOWN: esp_reset_reason_t = 0;
pub const esp_reset_reason_t_ESP_RST_POWERON: esp_reset_reason_t = 1;
pub type esp_reset_reason_t = cty::c_uint;
// 関数ポインタ型の変換
pub type shutdown_handler_t = ::core::option::Option<unsafe extern "C" fn()>;
extern "C" {
    pub fn esp_register_shutdown_handler(handle: shutdown_handler_t) -> esp_err_t;
}
```

bindgen雑感

そこそこ複雑なフレームワークのバインディングを生成

対象：ESP-IDF（ESP32用のEspressif公式フレームワーク）

内部にlwIP（組込み向けTCP/IPスタックなどを含む）

Kconfigを使ったビルドオプションの構成システム

I2C, SPI, GPIO等の
ドライバ関数のバインディングを生成できることを確認

bindgenのオプション調整は必要だが、手書きよりはるかに楽

組込みシステムでの現実的な利用法

フレームワークから全てRust実装できるのが理想的だが、
現実的には難しい

C FFIを活用してCの既存フレームワークを利用しつつ
アプリケーションの本体など重要な部分から徐々に置き換える運用も可能

bindgenなどのツールを活用すれば
現実的なコストでCの既存資産を呼び出し可能

async/await

非同期操作を同期操作と
同じ記述順序で書けるようにする言語機能

　C#, TypeScript, Python等様々な言語に同等機能あり

1年前くらい (Rust 1.44) で no_std 環境でも使用可能に

　つまり (技術的には) 組込み環境で使用可能

コンパイラ自体の言語機能とは別に、
非同期操作をどのように実行するかを決定して実行する
非同期ランタイムが必要

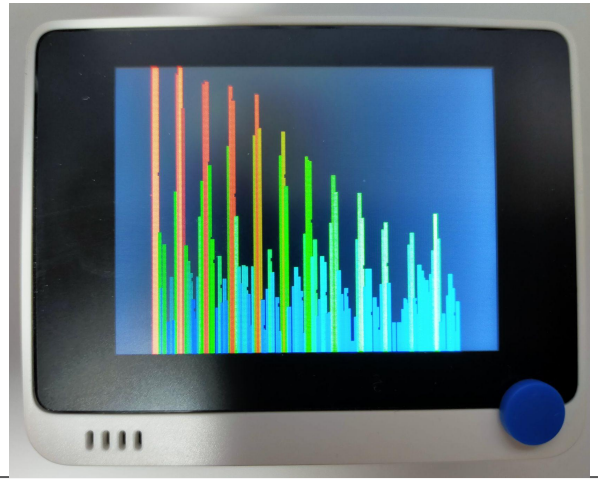
CMSIS-DAP

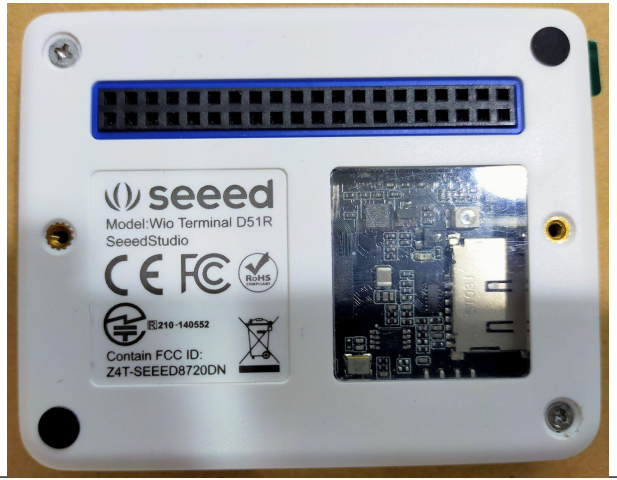
Wio Terminal

組込みRust本の解説で用いるマイコン・モジュール

液晶・ボタン・スピーカーが1筐体に収まっている

メインCPUはArm Cortex-M4F (ATSAMD51P19A)





CMSIS-DAP

Arm Cortex系プロセッサのデバッグ機能に
USB経由でアクセスするための仕様とファームウェア実装

- ホストとUSB経由で通信
- ターゲットとJTAGもしくはSWDで通信

Armが策定していて仕様とソースが公開されている

https://arm-software.github.io/CMSIS_5/DAP/html/index.html

CMSIS-DAP準拠デバッグ用アダプタを作る

USB機能付きMCUがあれば作れる

Wio Terminalのデバッグではseeeduino XIAOを使用

- ATSAMD21G18(Cortex-M0+)の
  小型モジュール
- USB2.0 FS対応

組込みRust本執筆時は既存ファームを使用
（Seeedが公開している）

Seeeduino XIAO

既存ファームウェアの問題点

既存ファームウェアは
Windowsのドライバが自動インストールできない

- ベンダ固有クラスなのでWinUSBドライバが必要
- WCIDというものを正しく返す必要がある

既存ファームをWCID対応にするのは手間がかかる

- 利用しているUSBライブラリの構造上の問題

つまり、RustでCMSIS-DAPを実装してしまえばいいのでは？

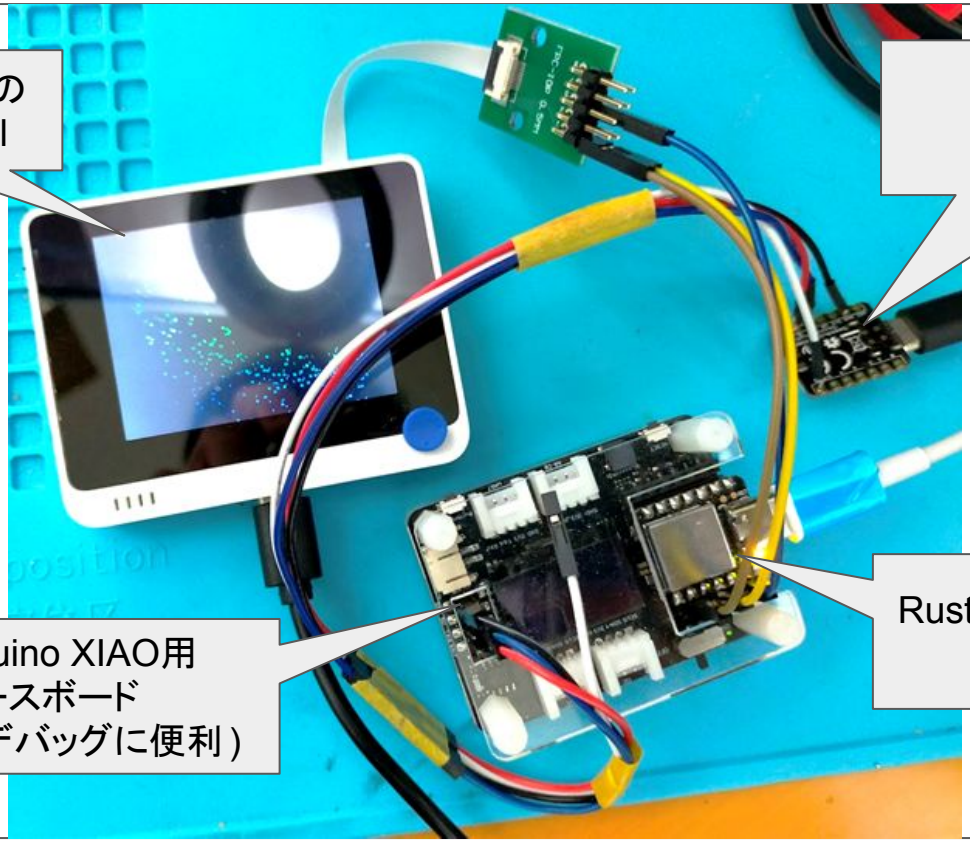
RustでCMSIS-DAPを実装する

そもそも可能か？

- Seeeduino XIAOのコードをRustで書けるか？→Yes
  - BSPとしてxiao_m0クレートがある
- USBを扱うことはできるか？→Yes
  - usb-deviceクレート
  - xiao_m0クレートが
    usb-deviceクレート内のトレイトを実装
  - Seeeduino XIAOのUSB機能を利用可能

→実装しよう！

開発中の風景


デバッグ対象の
WioTerminal
CMSIS-DAP
ファームウェア
デバッグ用XIAO
Seeeduino XIAO用
ベースボード
(XIAOのデバッグに便利)
Rust実装版CMSIS-DAP
を動かすXIAO

開発の流れ

USBデバイスとしての機能を実装していく

- usb_deviceクレートのUsbClassトレイトを実装
  - コンフィグレーション・ディスクリプタを返す
  - ストリング・ディスクリプタを返す
  - コントロール転送を行う
  - etc…

USBデバイス開発としてはかなり楽

大体3日くらいで実装完了

```
impl<B: UsbBus> UsbClass<B> for
CmsisDapInterface <'_, B> {
fn get_configuration_descriptors(...) {...}
fn get_bos_descriptors(...) {...}
fn get_string(...) {...}
fn control_in(...) {...}
...
}
```

動作確認（OpenOCD）

OpenOCDで接続…うごいた！

```
$ sudo ~/openocd-pico/bin/openocd -f interface/cmsis-dap.cfg -c
"cmsis_dap_vid_pid 0x6666 0x4444" -f target/atsame5x.cfg
...
Info : Using CMSIS-DAPv2 interface with VID:PID=0x6666:0x4444,
serial=test
Info : CMSIS-DAP: SWD  Supported
Info : CMSIS-DAP: FW Version = 2.00
Info : CMSIS-DAP: Serial# = Piyo
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : SWCLK/TCK = 0 SWDIO/TMS = 0 TDI = 0 TDO = 0 nTRST = 0
nRESET = 0
Info : CMSIS-DAP: Interface ready  // つながった！
Info : clock speed 2000 kHz
Info : SWD DPIDR 0x2ba01477  // SWDでターゲットと通信して読んだ値
Info : atsame5.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : starting gdb server for atsame5.cpu on 3333
Info : Listening on port 3333 for gdb connections // GDB待ち
```

動作確認（GDB）

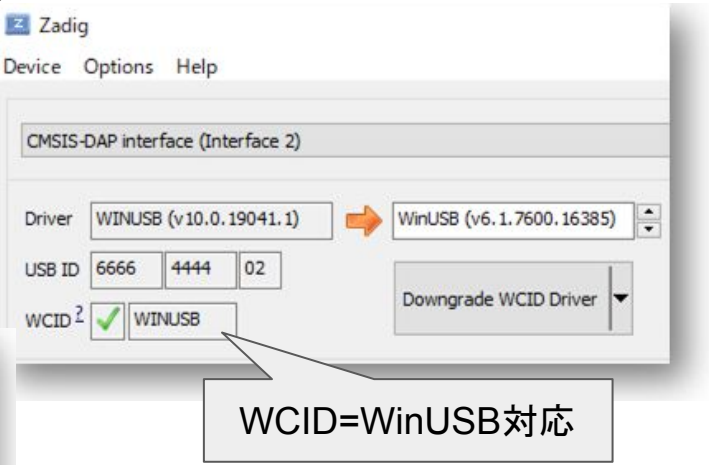
GDBを繋いで確認

```
$ gdb-multiarch 8-2-mic_fft  // 組込みRust本のスペアナサンプル
(gdb) load  // WioTerminalに書き込み
...
Transfer rate: 19 KB/sec, 8730 bytes/write.  // 19[KB/s]なのでまあまあ実用的
(gdb) monitor reset halt     // ターゲットのリセット
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0000056c msp: 0x2000d6a0
(gdb) cont // 実行開始
Continuing.
^C          // 中断
...
(gdb) info reg  // レジスタ読み出し
r0          0x6              6
r1          0x19f1           6641
...
pc          0x75fe           0x75fe
<embedded_graphics::pixelcolor::conversion::<impl
core::convert::From<embedded_graphics::pixelcolor::rgb_color::Rgb888> for
embedded_graphics::pixelcolor::rgb_color::Rgb565>::from+64>
xPSR        0x41000000       1090519040
...
```

動作確認（Windows）

Windows機に接続して
ドライバが自動インストールされるか確認

- WCID対応→OK
- WinUSB互換と認識
- WinUSBのロードを確認



実装してみて感じたRustの便利なところ

cargoが便利

- 外部ライブラリの導入が容易
- C/C++だと結構面倒…

ハードウェア抽象化が良くできている

- トレイトによる抽象化
- cargoによりライブラリ間共通トレイトの導入が容易
  - e.g. usb-deviceクレートとxiao_m0クレート
- 今回のCMSIS-DAP実装もusb-deviceを使った
  デバイス非依存の実装

実装してみて感じたRustの便利なところ

Result<T,E>によりエラー伝搬が楽に書ける

- C++だと例外だが組込みだと大体無効化されている
  - 仕方なくエラー値伝搬するが言語サポート無くて大変辛い
- ? を使ってエラー時のearly returnが簡単に書ける
- エラー処理がサクサク書けるのでサボりがちな
  エラー処理を書く気になれる

```
let result =
swd_transfer_inner_with_retry(config, swdio, swd_request, 0)?;// エラー時early return
write_u32(response, result);
*response_count += 1;
```

実装してみて感じたRustの便利なところ

範囲外アクセス時に未定義動作とならない

- 範囲外アクセス時はpanicする
- panicハンドラに飛んでいくだけなので
  バックトレースから発生源を容易に特定可能

実装してみて感じたRustの便利なところ

結局のところ、

現代的な仕様の言語

組込みで使える要求コストの低さ

（GCなし、ゼロコスト抽象化、etc…）

を両立しているところが便利

（ライフタイムなどの仕組みによる安全性はいわずもがな）

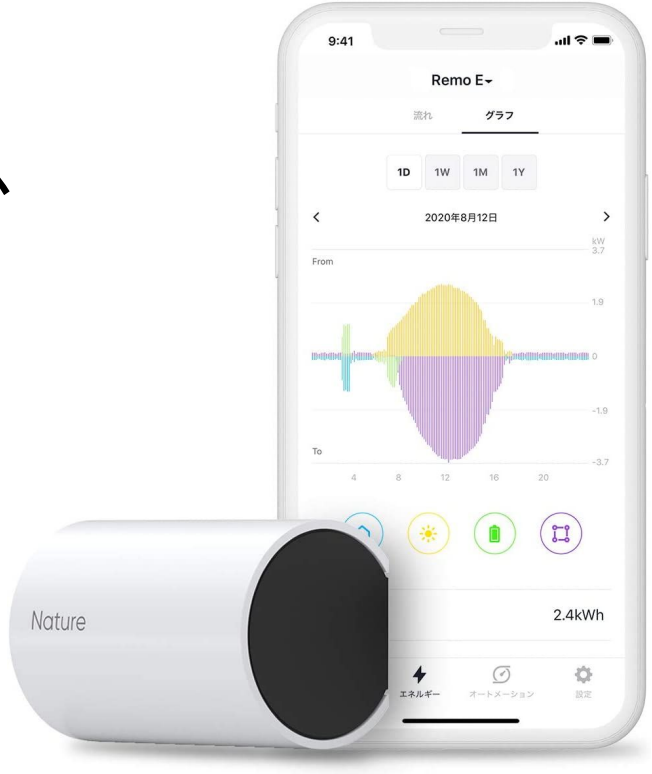
ECHONET Lite

Nature Remo E / E lite

HEMS
家庭内エネルギーマネジメントシステム
(Remo E lite はスマートメーターのみ)

- スマートメーター
- 蓄電池
- 太陽光発電システム

各機器とは **ECHONET Lite**
という通信規格でやりとり



ECHONET Lite

- スマートホームを実現する通信規格
- 主にUDPを通信経路として実装
- 機器ごとに制御可能なプロパティを定義
  - エネルギーマネジメントが念頭にあってえらい！

クラスグループ	機器例
センサ関連機器	火災センサ、人体検知センサ、温度センサ、CO2センサ、etc
空調関連機器	エアコン、扇風機、ホットカーペット、etc
住宅設備関連機器	給湯機、スマートメーター、太陽光発電、蓄電池、電気自動車充放電機、 etc
管理操作関連機器	コントローラ、etc

[ECHONET Lite規格の概要] https://echonet.jp/about/features/

ECHONET Lite

表 6-5 プロファイルオブジェクトスーパークラス構成プロパティ一覧

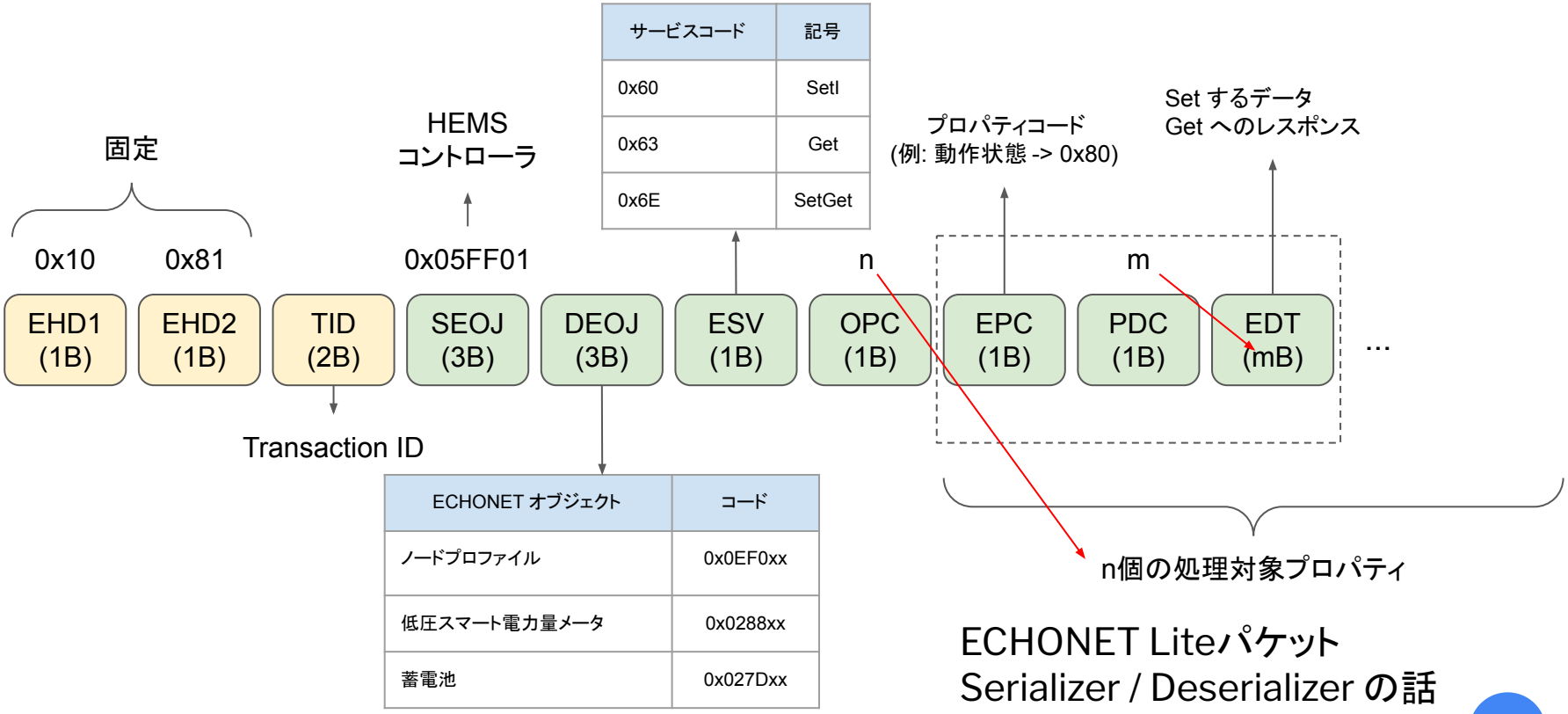
プロパティ名称	EPC	プロパティ内容 値域(10 進表記)	データ型	サイズ(Byte)	ｱｸｾｽﾙｰﾙ	必須	状変時アナウンス	備考
異常発生状態	0x88	何らかの異常の発生状況を示す。 異常発生有＝0x41、異常発生無＝0x42	unsigned char	1	Get			(1)
メーカコード	0x8A	3 バイトで指定。 (ECHONET コンソーシアムで規定。)	unsigned char×3	3	Get	○		(2)
事業場コード	0x8B	3 バイトの事業場コードで指定。 (各メーカ毎に規定。)	unsigned char×3	3	Get			(3)
商品コード	0x8C	ASCII コードで指定。 (各メーカ毎に規定。)	unsigned char×12	12	Get			(4)
製造番号	0x8D	ASCII コードで指定。 (各メーカ毎に規定。)	unsigned char×12	12	Get			(5)
製造年月日	0x8E	4 バイトで指定。 YYMD(1 文字 1 ﾊﾞｲﾄ)で示す。 YY：西暦年(1999 年の場合:0x07CF) M：月(12 月の場合＝0x0C) D：日(20 日の場合＝0x14)	unsigned char×4	4	Get			(6)
状変アナウンスプロパティマップ	0x9D	「APPENDIX ECHONET 機器オブジェクト詳細規定」付録 1. 参照	unsigned char×(MAX17)	Max. 17	Get	○		
Set プロパティマップ	0x9E	「APPENDIX ECHONET 機器オブジェクト詳細規定」付録 1. 参照	unsigned char×(MAX17)	Max. 17	Get	○		
Get プロパティマップ	0x9F	「APPENDIX ECHONET 機器オブジェクト詳細規定」付録 1. 参照	unsigned char×(MAX17)	Max. 17	Get	○		

注）状態変化時（状変時）アナウンスの○は、プロパティ実装時には、処理必須を示す。

こういう600ページくらいのPDFがある

[ECHONET Lite規格書] https://echonet.jp/spec_g/

ECHONET Lite パケットフォーマット



サービスコード	記号
0x60	SetI
0x63	Get
0x6E	SetGet

ECHONET オブジェクト	コード
ノードプロファイル	0x0EF0xx
低圧スマート電力量メータ	0x0288xx
蓄電池	0x027Dxx

ECHONET Liteパケット
Serializer / Deserializer の話

echonet-lite-rs 開発経緯

- 多数のECHONET Lite機器が存在 (大型家電も...)
- 実験設備を求めて全国行脚
- もっと便利なツールが欲しいよね、という話題に

じゃあRustで！

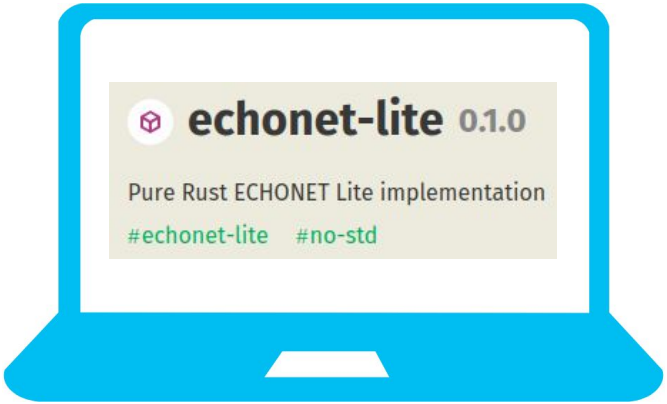
echonet-lite-rs

- https://crates.io/crates/echonet-lite
- https://github.com/tomoyuki-nakabayashi/echonet-lite-rs
- ECHONET Lite パケットの Serializer / Deserializer
- UDPで通信するところは別途必要
- あえてミニマムに切り出し

ファームウェア開発にRustを導入する戦略

- スクラッチからRustで作る
- 一部をRustで置き換える ← こっちをやりたい
  - Serializer / Deserializer は切り出しやすい

①便利ツールとして使ってもらう → ②便利ツールからの利用で完成度をあげる → ③満を持して持っていく！



echonet-lite-rs の活用事例

ECHONET Lite 機器のスキャナー

NatureはRust使ってる！

```
$ cargo run -- scan -i wlp2s0 -t 192.168.1.17 -c Pv
House Hold Solar Power: 0x0279
[動作状態]　80: 30
[設置場所]　81: 00
[規格version]　82: 00 00 4A 00
[識別番号]　83: FE 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[瞬時消費電力]　84: 00 00
[積算消費電力]　85: 00 00 00 00
...
```

ECHONET Lite 機器エミュレータ

```
$ cargo run -- emulator -o data/emulator/pv/pv.json data/emulator/pv/profile.json
```

echonet-lite-rs を試すには？

- examples に find.rs というサンプルプログラムを用意
- LAN内の ECHONET Lite 機器から動作状態を取得

```
$ cargo run --example find
got response from 192.168.1.2:3610
EHD: 1081
TID: 1
SEOJ: [0E F0 01]
DEOJ: [05 FF 01]
ESV: 72 (GetRes)
80: 30
```

ECHONET Lite 機器は
逸般的な誤家庭にはあるという噂が …

家にECHONET Lite機器がない？

そんなあなたに！
Nature Remo E / E lite が
あればお喋りできる！！！

しかもエネルギーマネジメントもできる！

買わない理由が見つからない！

Rust使ってよかったこと

- 雑に書いても（正しく）動く
  - パフォーマンス気にしないところはコピーを実装すれば所有権システムにあまり悩まない
  - 数千行くらい書き殴ってもコンパイル通れば不安なく動く
  - 圧倒的手戻りの少なさ
- ライブラリが優秀
  - serdeが優秀
  - structoptが優秀

serde

- デファクトスタンダードの Serializer / Deserializer
- 様々なフォーマットに対応
  - json
  - yaml
  - msgpack
  - cbor
  - など

```
use serde::{Serialize, Deserialize};
#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}
```

```
let point = Point { x: 10, y: 20 };
serde_json::to_string(&point)
serde_yaml::to_string(&point)
```

structopt

- コマンドラインオプションを良い感じに作れる

```
use structopt::StructOpt;
#[derive(StructOpt, Debug)]
#[structopt(name = "basic")]
struct Opt {
    /// Activate debug mode
    #[structopt(short, long)]
    debug: bool,
    /// Files to process
    #[structopt(name = "FILE", parse(from_os_str))]
    files: Vec<PathBuf>,
}
```

```
$ ./cli --help
USAGE:
    cli [FLAGS] [OPTIONS] [file]...

FLAGS:
    -d, --debug    Activate debug mode
    -h, --help     Prints help information

ARGS:
    <file>...    Files to process
```

no_stdの困りごと

- ライブラリ機能が制限される
  - stdの定番crateで使いたい機能が使えなかったり…
- ベストプラクティス迷子
  - まだ定番の方法が確立されていない

組込みRustはこういう人におすすめ！

- パフォーマンスを追求したい！
- コンパイラにたくさん叱られたい！
  - コンパイラにできるだけたくさんチェックして欲しい
- 雑に書いても正しく動いてそれなりに性能出したい！
  - プロトタイピングにも悪くないのでは？
- 型システムでパズルを楽しみたい！

more info

- 組込みRustの歩き方 Edition 2021
  - https://speakerdeck.com/tomoyuki/how-to-learn-embedded-rust-edition-2021
- 組込みRust和訳集へのコントリビュータも募集中

Rustは組込みシステムの歯車になれるか？

ポテンシャルはある！
最後はマンパワー

なので、ぜひ今日から組込み Rustを始めましょう！

今日から組込みRustを始めるのに最高の本が…

Amazon 売れ筋ランキング: - 51,830位本 (の売れ筋ランキングを見る本)
- 384位ソフトウェア開発・言語

2021/09/03 朝のランキング
明日が楽しみですねぇ！

Will Rust be the Gear of Embedded Systems?

2021/09/03
SWEST23
Tomoyuki Nakabayashi
Kenta Ida