省電力コプロセッサをsafeに使うRustライブラリ

鈴木 豪
メンター: 光成さん

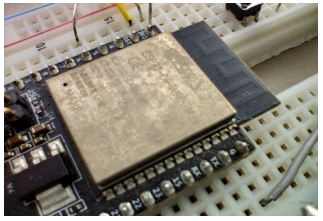
サイボウズ・ラボユース
夏合宿

自己紹介 & はじめに

鈴木 豪

興味・まあまあ知っていること:
ｺﾝﾋﾟｭｰﾀｱｰｷ，コンパイラ，電子工作，OS

ターゲットマイコン
ESP32 series, M5Stack



© M5Stack

省電力コプロセッサをsafeに使うRustライブラリ

名前: mtukai (仮)

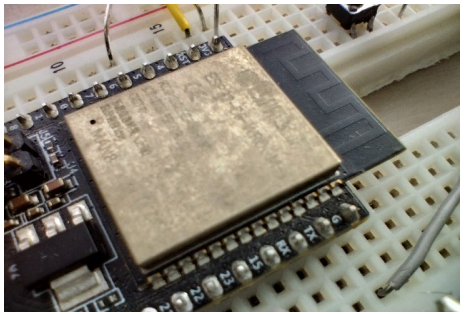
省電力コプロセッサ (ESP32-S3, C6, C5, P4, S31)
→ メインプロセッサの1/10の消費電力
※実測値

ESP32のRustサポート→ 2025/10にVer. 1.0安定版！
(安定しているわけではない)

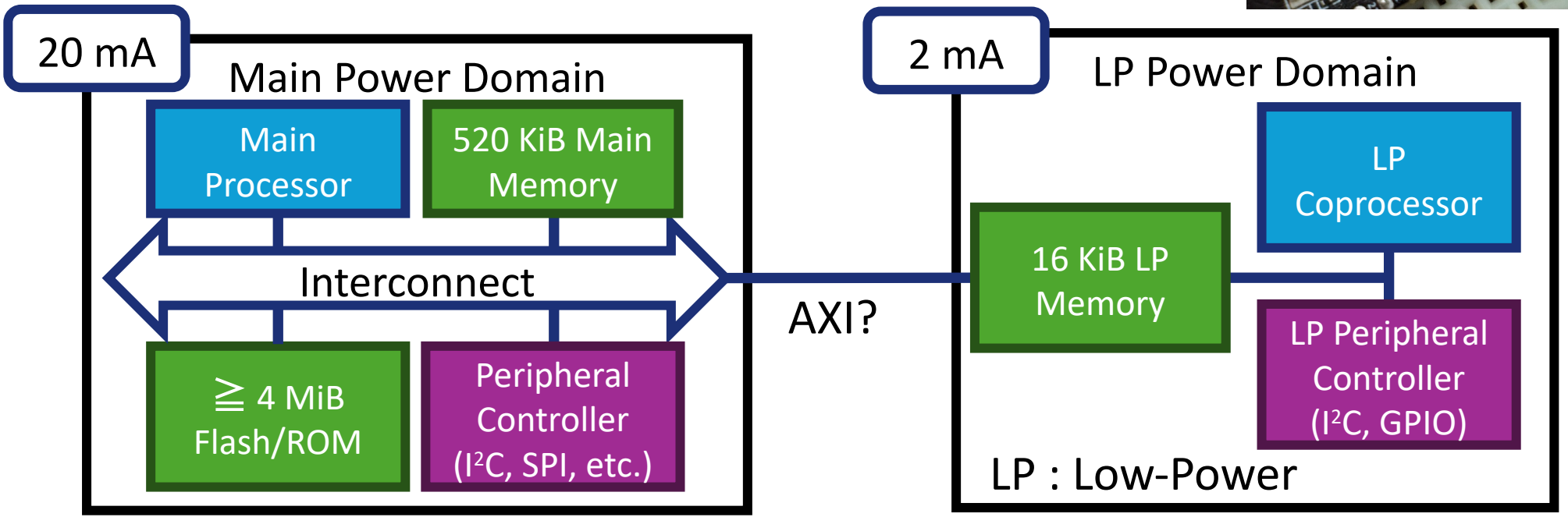
FAQ: なぜGo言語ではないのか？ → そもそもESP32-C6へ移植する必要がある
FAQ: 他に対応できそうなマイコンは？ → nordicのnRF54L15など

LPコア in ESP32-C6

ESP32-C6



ESP32-C6は2種類のコアとメインメモリを持っています．
ESP32-C6では省電力コプロセッサをLPコアと呼んでいます．

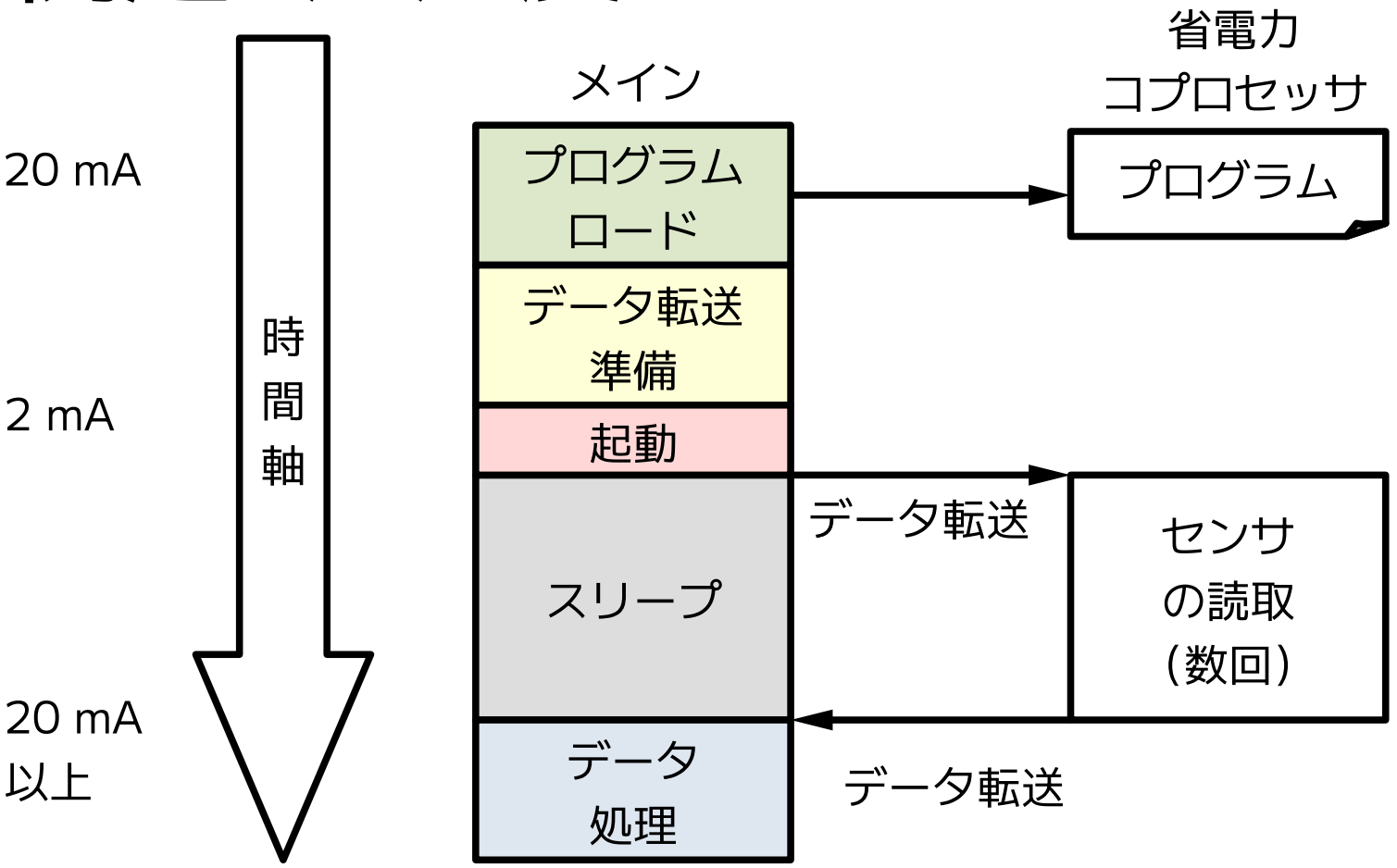


FAQ: LPコアのアーキテクチャは？無印ESP32以外はRISC-V (RV32IM(A)C)

使い道

- **省電力**
  - センサによる計測をLPに任せ，メインプロセッサは普段寝ている（より高いサンプルレート！）
- 実時間
  - 不思議なプロトコルを喋らせるとか，ロータリエンコーダみたいなことをするとか
  - In-Order，独立したバス，キャッシュなし

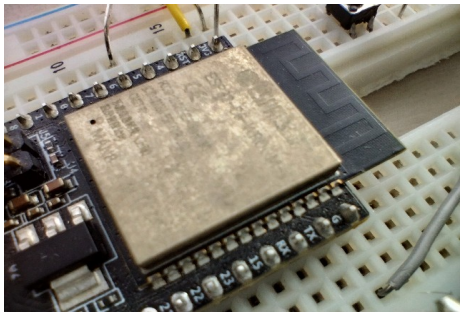
例題: 温湿度センサ

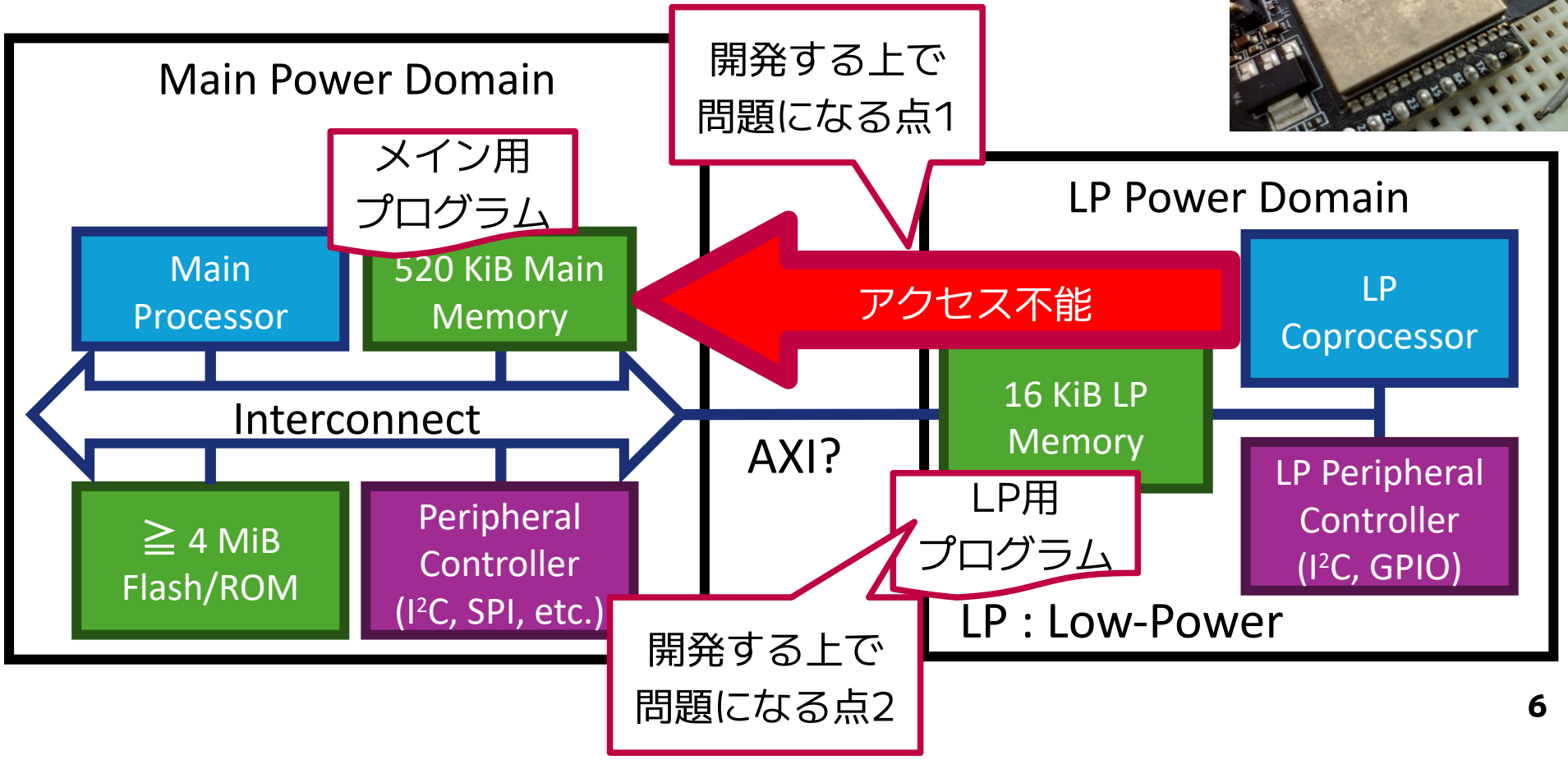

20 mA
2 mA
20 mA
以上
時
間
軸
メイン
省電力
コプロセッサ
プログラム
ロード
プログラム
データ転送
準備
起動
スリープ
データ転送
センサ
の読取
(数回)
データ
処理
データ転送


HAT
ENV II
M5STACK
T&H
SHT30
PRESSURE
BMP280
GEOMAGNETIC SENSOR
BMM150
G26
I2C-SCL
G0
I2C-SDA
I2C
INTERFACE

問題点

ESP32-C6




Main Power Domain
メイン用
プログラム
開発する上で
問題になる点1
LP Power Domain
Main
Processor
520 KiB Main
Memory
アクセス不能
LP
Coprocessor
Interconnect
AXI?
16 KiB LP
Memory
LP Peripheral
Controller
(I²C, GPIO)
≧ 4 MiB
Flash/ROM
Peripheral
Controller
(I²C, SPI, etc.)
LP用
プログラム
開発する上で
問題になる点2
LP : Low-Power

省電力コプロセッサの問題点

1.メインプロセッサが眠っている間
メインメモリにアクセスできない

→予めコピーしておく必要がある

2.プログラムが2つのコアで分かれている

→見通しが悪い．プロジェクトも2つ分かれている

標準ESP32ライブラリのLチカ例

main

```
let mut lp_core = LpCore::new(peripherals.LP_CORE);
// load code to LP core
let lp_core_code = load_lp_code!(
    "LPコプロセッサの実行バイナリへのパス"
);

// start LP core
lp_core_code.run(&mut lp_core, LpCoreWakeupSource::HpCpu,
lp_pin);
println!("lpcore run");

let data = (0x5000_2000) as *mut u32;
loop {
    print!("Current {:x}           \u{000d}",
        unsafe { data.read_volatile()}
    );
}
```

coprocessor

```
#[entry]
fn main(mut gpio1: Output<1>) -> ! {
    let mut i: u32 = 0;
  let ptr = 0x5000_2000 as *mut u32;
    loop {
        i = i.wrapping_add(1u32);
      unsafe {ptr.write_volatile(i);}

        gpio1.set_high().unwrap();
        gpio1.set_low().unwrap();
    }
}
```

ポインタ直書き，Rustの安全性ゼロ

温湿度センサ

unsafe不使用！
安全にデータ処理

main

```
let mut lp_core = LpCore::new(peripherals.LP_CORE);
// load code to LP core
let lp_core_code = load_lp_code!(
    "LPコプロセッサの実行バイナリへのパス"
);
```

LP側のELFバイナリを解析するRustマクロ

```
let mut parcel = MainLPParcel {
   i2c : LPI2C::new(i2c),
   result : LPVec::new(),
   measurement_count : 30 };
```

省電力コプロセッサで
利用するものを詰め込む

```
lp_core_code.run_light_sleep(&mut lp_core, /*略*/, &mut parcel);
```

コプロセッサ起動時に送る
メインプロセッサはスリープ

```
for tah in parcel.result.iter().flatten() {
  println!("Temp: {} C, Hum: {} %", tah.temp, tah.hum);
}
```

メインプロセッサが再開する
とデータが更新されて返る

温湿度センサ

メインプロセッサと協調動作するアロケータ

coprocessor

```
esp_rs_copro_procmacro::esp_rs_copro_statics!(4096);

fn main() -> ! {

    let v: &mut MainLPParcel = get_transfer::<MainLPParcel>().unwrap();

    for i in 0..v.measurement_count {
        v.result.push(read_sht30(&mut v.i2c).ok());
        delay_ms(1000);
    }
    wake_hp_core();
    lp_core_halt()

}
```

アロケータ・ヒープ
（ライブラリの力では送信するオブジェクトのサイズを推定できない）

メインプロセッサから受け取る関数

現時点では，ここの型引数だけは開発者が気にする必要がある

unsafeなことはなくデータの読み書き可能

キーアイテム

- **LPBox<T>構造体**　省電力コプロセッサへのアロケーションをサポートするBox<T>
- **MovableObjectトレイト**　ポインタの書き換えなどのデータの移動に必要な処理を実装
  - move_to_main, move_to_lpメソッド
  - マクロで自動実装できます！

似ているようであんまり似ていない: DRust [Haoran Ma et al. USENIX OSDI ‘24]

構造体定義

```
#[derive(MovableObject)]
pub struct TempAndHumid {
    pub humid : f32, pub temp : f32
}

#[derive(MovableObject)]
pub struct MainLPParcel{
    pub i2c : LPI2C,
    pub result : LPVec<TempAndHumid>,
    pub measurement_count : i32
}
```

なぜRustだと上手くいく？

- マネージドポインタ
  - unsafeの生ポインタ以外は逆参照などの処理をカスタマイズできる
- マクロ
  - C++だったとしたらプリプロセッサとして実装する必要があるだろう．C# (nanoFramework)やTinyGoだとランタイムを修正？

省電力コプロセッサの問題点

1.メインプロセッサが眠っている間
　メインメモリにアクセスできない

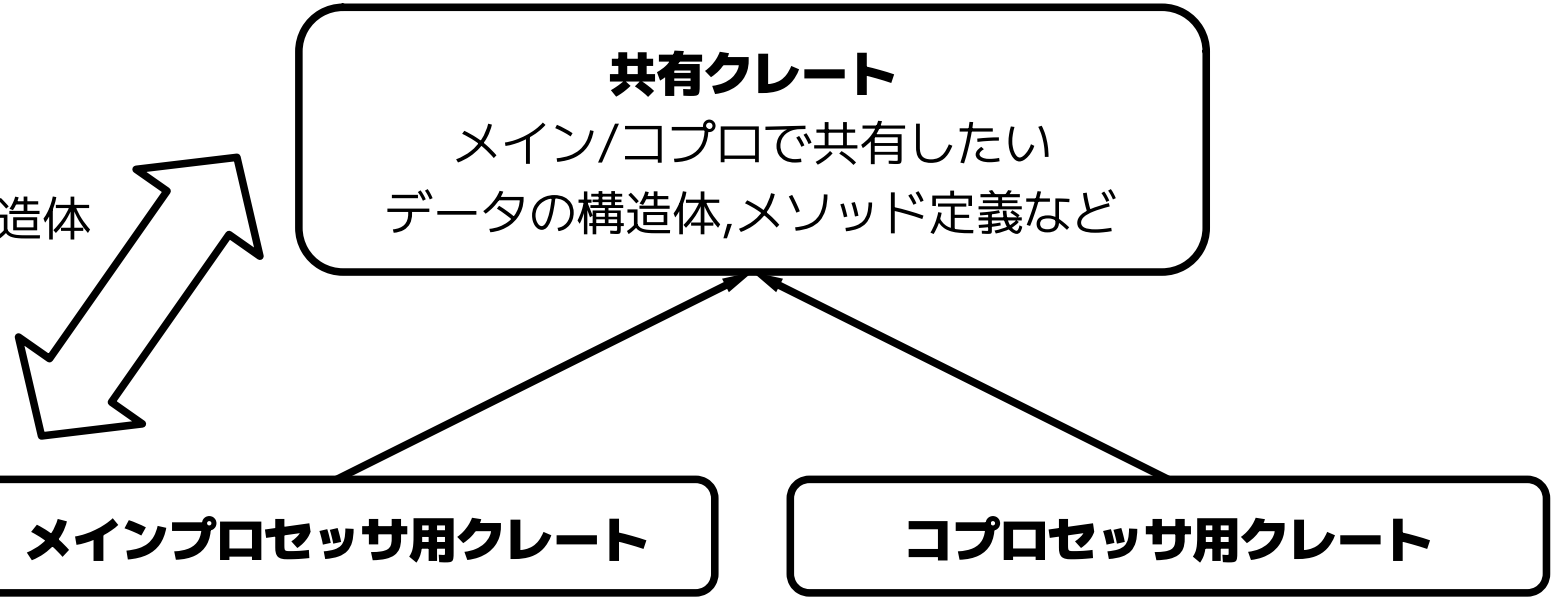
　　→予めコピーしておく必要がある

2.プログラムが2つのコアで分かれている

　　→見通しが悪い．プロジェクトも2つ分かれている

Rustの現状

問題点1
どっちに関数や構造体
を定義する？



cargoの問題点
targetの切り替えがなかなか面倒
・コマンドラインオプションをいちいち書く
・targetだけでなくfeatureも切り替えてビルドする

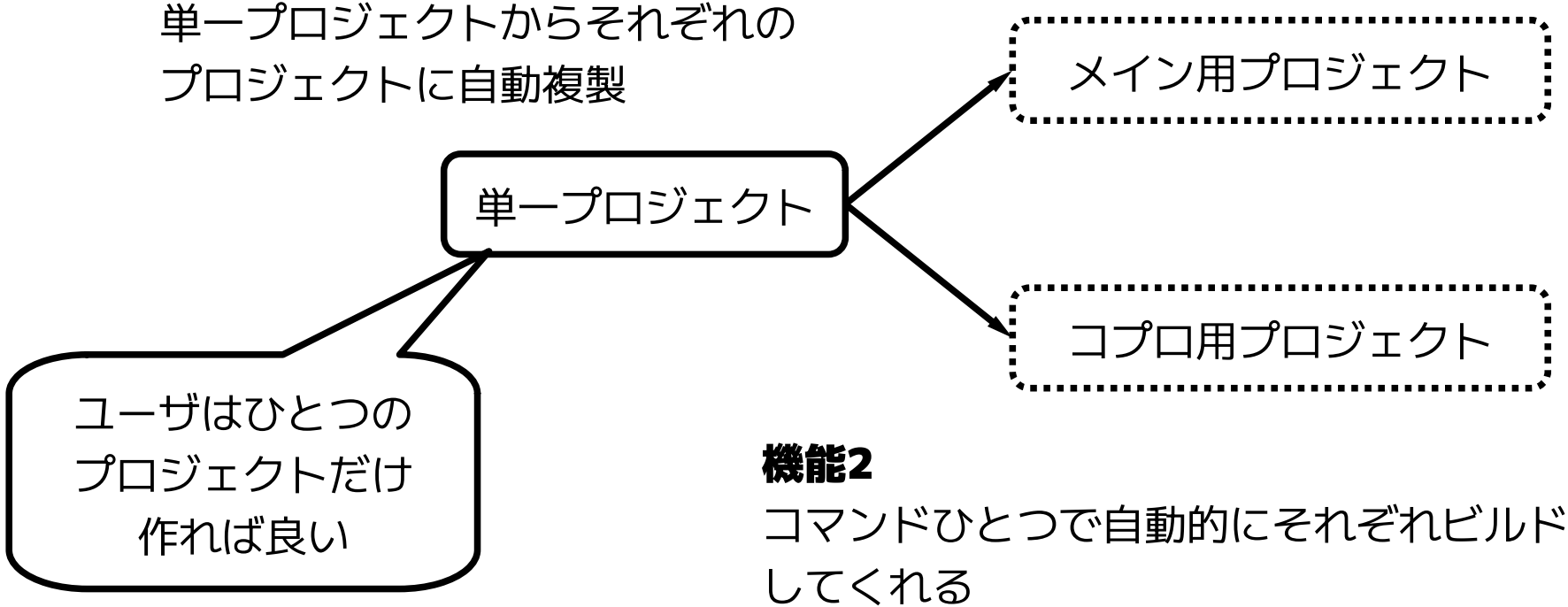
問題点2
コンパイルし忘れ

プロジェクト複製ツール

mtukai-projgen

機能1

単一プロジェクトからそれぞれの
プロジェクトに自動複製



機能2

コマンドひとつで自動的にそれぞれビルド
してくれる

単一ソース・複数プロジェクト

いくぶん自然にコプロセッサで関数を実行することができる.

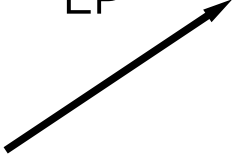
```
#[mtukai_projgen_procmacro::entry(4096)]
fn lpmain(_ : &mut LpContext, data : &mut LPBox<SimpleList>, result : &mut i32) -> ! {
    *result = data.sum();
    data.push(10000);
    wake_hp_core();
    lp_core_halt()
}


#[cfg(feature = "has-lp-core")]
#[esp_hal::main]
fn main() -> ! {
    let (list, expected_sum) = gen_list();
    let mut data = LPBox::new(list);
    let mut result = 0;
    println!("lpcore run");
    let mut lp_context = LpContext::new(&mut LpCore::new(peripherals.LP_CORE), &mut Rtc::new(peripherals.LPWR));
    if let Err(e) = lpmain(&mut lp_context, &mut data, &mut result) {
        println!("Error running LP core: {}", e);
    }

    println!("result: {} (expected: {})", result, expected_sum);
    print_list(&data);
    println!("result: {} (expected: {})", data.sum(), expected_sum + 10000)
}
```

lpmainはコプロセッサで実行される.
同じプロジェクト内に書ける.
関数呼出として書くことができる.
立役者: entryマクロ

コンパイルエラーを起こさない複製

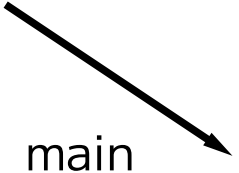
片方のプロセッサに向けにしかコンパイルできない関数などがある可能性がある.
メイン関数から間接/直接呼出される関数のみを取り出して複製する.

```
use main_dake_lib;

fn main_hoge() { ... }

fn lp_hoge() { ... }

fn main_main() { main_hoge(); }

fn lp_main() { lp_hoge(); }
```




```
/* use main_dake_lib; */
#[cfg(not(feature = "is-lp-core"))]
fn main_hoge() { ... }
fn lp_hoge() { ... }
#[cfg(not(feature = "is-lp-core"))]
fn main_main() { main_hoge(); }
fn lp_main() { lp_hoge(); }
```

それぞれで呼出される関数のみ有効化




```
use main_dake_lib;
fn main_hoge() { ... }
#[cfg(not(feature = "has-lp-core"))]
fn lp_hoge() { ... }
fn main_main() { main_hoge(); }
#[cfg(not(feature = "has-lp-core"))]
fn lp_main() { lp_hoge(); }
```

<u>実現方法</u>

rust-analyzerを用いたロバストなコールフロー解析をした.
traitについてはRapid Type Analysis風の生存解析をした.
[OOPSLA '96]

この手法の良い点/悪い点

良い点

- 変化の激しいRustコンパイラに手を入れていない
- プロジェクトが1つで済む
- LPプログラムのコンパイルし忘れを防げる

悪い点

- main/LP向けのuseを混在できると分かりづらい？
- エディタで編集しているときLSPが困惑する
- コンパイルエラーが複製先のファイル/行番号（#LINEがRustにはない）
- 保守的な解析なので，多少は開発者がアノテーションを加える必要があるかもしれない（無いよりはまし）

評価

フェアじゃないところ
- Rust: 様々なランタイム機能(**タイマーのキャリブレーション** etc.)が未実装
- Rust: Link-Time Optimization有効 (無いと6.9KiB)
- C: グローバル変数でベタにデータをやりとり

温湿度センサについて消費電力とコプロセッサ上のコードサイズを比較.
1分ごとに1回計測する．30分間

	消費電力 [J]	コードサイズ [B]	safe?
C	2.27	2480	No
mruby/copro	2.27	4932 （JITコード+ランタイム）	Yes
Rust (作ったもの)	35.00	2150	Yes
Unsafe Rust	未計測	1330	No

消費電力：他より高い

Rust用ESPライブラリが未熟だから．不要な回路に電源入ってる？→成熟を待つ

コードサイズ：妥当な増加

Vecとアロケータが増分．
memcpyが+300B！？

FAQ. 今回は計測頻度が低いのでメインプロセッサと消費電力変わらない？ → YES

※mruby/copro：JITコンパイラを搭載しコプロセッサで動作するmruby (自作)

G. Suzuki, S. Moriguchi, T. Watanabe: Co-operative JIT Compilation for Resource-Constrained Low-Power Coprocessors. ACM SIGPLAN MPLR '25

貢献: バグ修正・Eratta/バグ報告

ESP-IDF（C言語実装の標準ライブラリ）にプルリクエスト：3件

- ESP32-C5において使えないピンが定義されているバグ
- マクロの重複定義ミス
- タイムアウトのための時間計測の精度が非常に低い

ESP-RUST/ESP-HAL（Rust言語実装の標準ライブラリ）にプルリクエスト：4件

- LP関連のExample Codeの追加(2つ)
- LPのコードのロードがコンパイラの機嫌?によっておかしくなるバグ
- LP I^2Cのバッファの管理が間違えて溢れるバグ

ESP32-C6のTechnical Reference ManualにEratta報告: 2件

某中華オシロスコープ（20万円程度）のバグ報告: 3件

VSCodeにシンボリックリンクに関するPR作成（マージされていない，放置状態...）

→ **M5Stackなど多く利用されるマイコンのライブラリへの貢献！**

& まだまだ不安定なRustライブラリの礎に貢献しました

まとめ

- 課題: 省電力コプロセッサはメインプロセッサのスリープ中にメインメモリにアクセスできない
- 省電力コプロセッサをsafeに利用できるRustライブラリ
  - オブジェクトの自動移動
  - プログラムを1箇所に書けるようにする
- 次にやること
  - 両プロセッサの並列動作
    （排他動作にシームレスに移行できると嬉しい）

補足

ESP32コプロセッサの歴史

ESP8266: 最初のESPマイコン．省電力コプロセッサなし．
ESP32: 初の省電力コプロセッサ．**謎ISA**，8KiB，GPIOのピン番号違う！
ESP32-S3: 初のRISC-V省電力コプロセッサ．メインプロセッサはXtensa．
I2CやADCがまったく動かない！（ADCは今後リストラされました）
ESP32-C6: 利用可能なメモリ空間が**16 KiB**に拡張．メインプロセッサがRISC-Vに
ESP32-C5: 特に変更点なし．
ESP32-P4: 利用可能なメモリ空間が**32 KiB**に．ROMも16 KiB搭載．
メモリ保護機構（Memory Protection Unit）搭載！

P4以前は，
・NULLを参照するとランダムなアクセスが帰り，
・メインメモリにアクセスするとフリーズ
した．

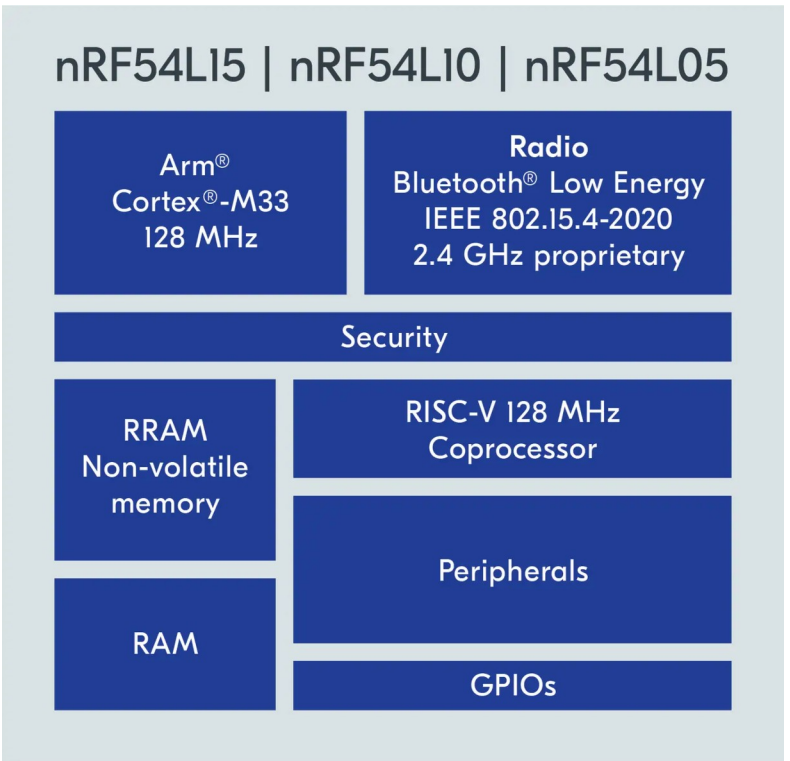
異種マルチプロセッサは他にも

Milk-V Duo

https://akizukidenshi.com/catalog/g/g118375/

nrf54L15

https://akizukidenshi.com/catalog/g/g131257/



```rust
pub struct MainLPParcel{
    pub i2c : LPI2C,
    pub result : LPBox<[TempAndHumid]>
}

impl MovableObject for MainLPParcel {
    unsafe fn move_to_main(&self, dest : *mut u8) { unsafe {
        let dest = dest as * mut MainLPParcel;
        self.i2c.move_to_main((&mut (*dest).i2c) as * mut LPI2C as * mut u8);
        self.result.move_to_main((&mut (*dest).result) as * mut LPBox<[TempAndHumid]> as * mut u8);
    }}

    unsafe fn move_to_lp(&self, dest : *mut u8) { unsafe {
        let dest = dest as * mut MainLPParcel;
        self.i2c.move_to_lp((&mut (*dest).i2c) as * mut LPI2C as * mut u8);
        self.result.move_to_lp((&mut (*dest).result) as * mut LPBox<[TempAndHumid]> as * mut u8);
    }}
}
```

仕組み: LPBox, MovableObject

- **struct LPBox<T>**: 移動関連のメソッドが付いたBox<T>
  - 実は現状では，構造体を作る意味はそこまで無い．
    が，必要に応じてデータをコピーする場合，Derefの実装を変える必要がある．
- **trait MovableObject**: 移動できるオブジェクト
  - LPBox<T>.move_to_...はアロケートした後T.move_to_...を呼ぶ．
    - LPBox→場所を確保する，MovableObject→データをコピーする役割
  - SerdeのSerializable/Deserializableに近い．しかし，1. memcpyして，2. ポインタのある場所だけ書き変えるという最適化が可能である．
    - 構造体にインラインに入っている構造体はそれぞれmemcpyする必要無し
- DRust [Ma et al. OSDI'24]を参考にしているものの，やっぱりポインタ書き換えや，メモリの節約などでいろいろ異なる．

おもしろいこと: load_lp_code

このマクロは，コプロセッサ用のelfファイルを読み取り，LP_CODEという配列のグローバル変数にバイナリを格納する．

- 省電力コプロセッサ上のヒープサイズを知りたいので，実はシンボル名にヒープサイズを含めて，それを読みとることで，メインプロセッサがヒープを初期化する．
  →コプロセッサ側にある:　`esp_rs_copro_procmacro::esp_rs_copro_statics!(4096);`
- この仕組みを使えば型情報も埋め込めるかもしれないと思ったが...

```
lp_core_code.run_light_sleep(&mut lp_core, /*起動方法など*/, &mut parcel);

let v: &mut MainLPParcel = get_transfer::<MainLPParcel>().unwrap();
```

名前衝突した場合とかに目をつむれば，もしかしたらできるかもしれない
（と書いている今思い始めた）

やっていること

ESP32の省電力コプロセッサを
safeに使えるRustライブラリ

uchan: ESP32のULPプログラミングのはまりポイント. elchika. 2021/01/17

https://elchika.com/article/3fbb17bd-5dec-4409-a063-d0b77871603f/

省電力コプロセッサ
（Ultra Low Power Coprocessor, LP Core, ULP LP Core Coprocessorなど
ESP32のドキュメントでさまざまな呼称がある）
はメインプロセッサの1/10(実測値)の消費電力！