# The Embedded Rust Book (Japanese)

# Contents

# 1. 導入

*The Embedded Rust Book* へようこそ。Rust をマイクロコントローラのような、「ベアメタル」の組込みシステムで使うための入門書です。

#### 1.0.1. 組込み Rust は誰のためのもの

組込み Rust は、Rust の高い抽象度と安全性のもと、組込みプログラミングをしたい人のためのものです。(https://doc.rust-lang.org/book/ch00-00-introduction.html)[Rust は誰のためのもの]も合わせて見て下さい)

#### 1.0.2. スコープ

この本の目的は、以下の通りです。

- 組込み Rust をできる限り速く開始できるようにします。すなわち、開発環境のセットアップ方法です。
- 組込み開発における Rust の現在のベストプラクティスを共有します。つまり、より正しい組込みソフトウェアを書くための、Rust の最善な利用方法です。
- いくつかのケースに対するマニュアルを提供します。例えば、1 つのプロジェクト内で、C 言語と Rust とを混在する方法です。

本書は出来る限り一般的な事項を取り扱います。ただし、説明を簡単にするために、全ての例で、ARM Cortex-M アーキテクチャを利用します。 読者は、このアーキテクチャに詳しい必要はありません。本書では、アーキテクチャ固有の詳細について、必要に応じて説明をします。

#### 1.0.3. この本は誰のためのもの

本書は、組込み開発か、Rust かのバックグラウンドを持つ人々に向けたものです。しかし、組込み Rust に興味がある人なら、誰でも、この本から何かを得られると思います。本書による学習効果を高めるために、事前知識が不足している読者は、「仮定と前提条件」のセクションを読み、不足している知識を補うことをお勧めします。不足知識を補うリソースを見つけるために、「その他のリソース」セクションをチェックすることができます。

##### 1.0.3.1. 仮定と前提条件

- Rust でのプログラミングを楽しんでおり、デスクトップ環境で Rust アプリケーションを書いたり、実行したり、デバッグしたりしたことがあることを前提とします。また、本書では Rust 2018 を対象とするため、2018 edition のイディオムに慣れ親しんでいる必要があります。
- C, C++, Ada といった言語で組込みシステムを開発、デバッグすることに慣れており、次の概念になじみがあることを想定します。
  - クロスコンパイル
  - メモリマップ方式のペリフェラル
  - 割り込み
  - I2C、SPI、シリアルといった一般的なインタフェース

##### 1.0.3.2. その他のリソース

もしあなたが上述した何らかの事項をよく知らない場合、もしくは、本書内の特定トピックに関して、より詳細な情報を知りたい場合、これらのリソースが役に立つでしょう。

| トピック | リソース | 説明 |
|---|---|---|
| Rust | Rust Book | もし Rust に親しんでいない場合、この本を読むことを強くお勧めします。 |
| Rust, Embedded | Discovery Book | 未翻訳 |
| Rust, Embedded | Embedded Rust Bookshelf | Rust 組込みワーキンググループによるいくらかのリソースがあります。 |
| Rust, Embedded | Embedonomicon | Rust で組込みプログラミングを行うときのより深い詳細が記載されています。 |
| Rust, Embedded | embedded FAQ | 組込みで Rust を使う際のよくある質問と回答です。 |
| Rust, Embedded | Comprehensive Rust 🦀: Bare Metal | 未翻訳 |
| Interrupts | Interrupt | - |
| Memory-mapped IO/Peripherals | Memory-mapped I/O | - |
| SPI, UART, RS232, USB, I2C, TTL | Stack Exchange about SPI, UART, and other interfaces | - |

#### 1.0.3.3. 未翻訳

未翻訳

#### 1.0.3.4. この本はどう使う

この本は、前から順番に読んでいくことを想定しています。後半の章は、前半の章で説明する概念に基づいて成り立っています。前半の章では、トピックの詳細に深入りせず、後半の章で再訪問します。

この本は、ほとんどの例で、ST マイクロエレクトロニクスの STM32F3DISCOVERY 開発ボードを使用します。このボードは、ARM Cortex-M アーキテクチャをベースとしています。基本機能はこのアーキテクチャベースの CPU では共通です。一方、ペリフェラルとマイクロコントローラ実装の詳細は、他のベンダーと異なります。同じ ST マイクロエレクトロニクスのマイクロコントローラファミリでも、違いがあります。

上記の理由から、本書内の例を理解するために、STM32F3DISCOVERY 開発ボードを購入することをお勧めします。

#### 1.0.3.5. 未翻訳

未翻訳

未翻訳

未翻訳

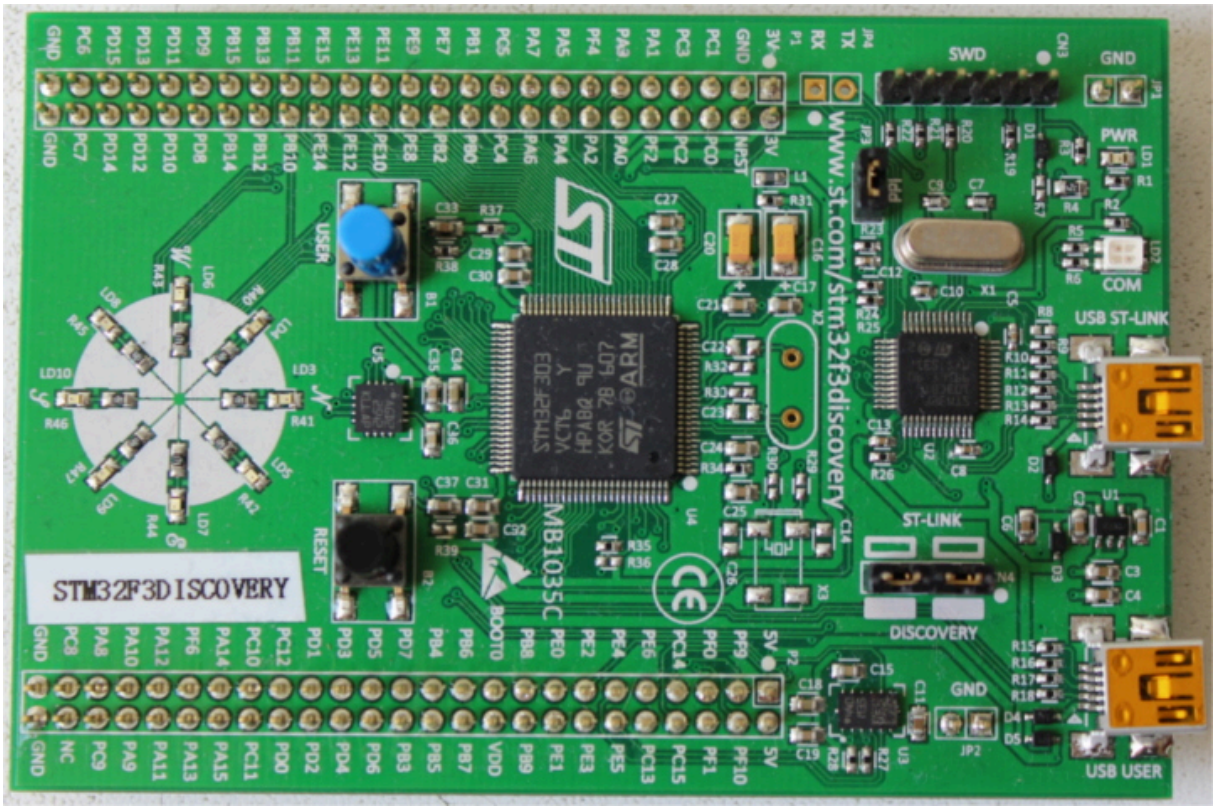
### 1.0.4. 未翻訳

未翻訳未翻訳

未翻訳

## 1.1. ハードウェアとの出会い

これから作業するハードウェアに詳しくなりましょう。




### 1.1.1. STM32F3DISCOVERY (“F3”)

私たちは、本書内でこのボードを“F3”と呼びます。

このボードには何が搭載されているか見てみましょう。

- STM32F303VCT6 マイクロコントローラが 1 つ。このマイクロコントローラは、次のものを搭載しています。
  - 単精度浮動小数点演算をハードウェアサポートし、最大 72MHz のクロック周波数で動作するシングルコアの ARM Cortex-M4F プロセッサ
  - 256 KiB の“フラッシュ”メモリ (1 KiB = 1024 bytes)
  - 48 KiB の RAM
  - 多くの“ペリフェラル”: タイマ、GPIO、I2C、SPI、USART、他
  - 両側面の“ヘッダ”に配置された多数の“ピン”
  - **重要** このマイクロコントローラは、約 3.3 ボルトで動作します。
- 加速度センサと磁気センサが 1 つずつ (1 つのパッケージにまとめられています)
- ジャイロセンサが 1 つ
- 円形に配置された 8 個のユーザ LED
- 第 2 のマイクロコントローラ: STM32F103CBT。このマイクロコントローラは、実際には、ST-LINK というオンボードプログラマおよびデバッガの一部であり、“USB ST-LINK”という名前の USB ポートに接続されています。
- “USB USER”というラベルが付いている第 2 の USB ポート。この USB ポートは、メインマイクロコントローラ (STM32F303VCT6)に接続されており、アプリケーションで利用できます。

未翻訳

未翻訳

## 1.2. Rust の `no_std` 環境

組込みプログラミングという用語は、様々な分野のプログラミングに使用されます。 たった数キロバイトの RAM か ROM が付随する 8 ビット MCU (例えば、ST72325xx)から、32/64 ビットの 4 コア Cortex-A53 @ 1.4GHz と 1GB の RAM が搭載された Raspberry Pi(Modell B 3+)のようなシステムまで、幅広いです。 どのような種類の目的とユースケースがあるか、によって、コードを書くときに異なる制限/限界が課されます。

2 つの一般的な組込みプログラミングの分類があります。

### 1.2.1. ホストされた環境

この分類の環境は、普通の PC の環境に近いです。これの意味するところは、POSIX のようなシステムインタフェースが提供されている、ということです。システムインタフェースは、ファイルシステムやネットワーク、メモリ管理、スレッドといった多様なシステムとやりとりするための基本要素を提供します。通常、標準ライブラリは、その機能を実装するために、これらの基本要素に依存します。また、sysroot や、RAM/ROM 利用の制限、そしておそらく特別なハードウェアや IO があるかもしれません。 全体としては、特殊な用途の PC 環境でコーディングをするようなものです。

### 1.2.2. ベアメタル環境

翻訳が古くなっています ベアメタル環境では、高機能な OS が動作していて、私たちのコードをホスティングしてくれる、ということはありません。これは、基本要素がないことを意味しており、それ故に、デフォルトでは標準ライブラリもありません。コードに `no_std` のマーキングをすることで、そのコードが、ベアメタル環境で実行できることを示します。no_std なコードからは、Rust の libstd とメモリの動的確保が使えません。しかしながら、no_std なコードでも libcore を使うことができます。libcore は、ほんの数種類のシンボルを提供することで、いかなる環境でも容易に利用することができます (詳細は、libcore を参照して下さい)。

#### 1.2.2.1. libstd ランタイム

上述の通り、libstd の利用には、いくらかのシステムインテグレーションが必要です。しかし、これは libstd が OS の抽象にアクセスするための共通の方法を提供しているだけでなく、ランタイムも提供しているためです。ランタイムは、とりわけ、スタックオーバーフロープロテクションの準備、コマンドライン引数の処理、メインスレッドの生成、をプログラムのメイン関数が呼び出される前に処理します。 このランタイムも、`no_std` 環境では利用できません。

### 1.2.3. まとめ

`#![no_std]`は、クレートレベルの属性で、そのクレートが std クレートの代わりに core クレートとリンクすることを意味します。libcore クレートは、プラットフォームに依存しない std クレートのサブセットです。libcore クレートは、プログラムが動作するシステムについて前提を置きません。 libcore クレートは、浮動小数点、文字列やスライスといった言語の基本要素となるAPI と、アトミック操作や SIMD 命令といったプロセッサの機能を公開する API とを、提供します。一方、プラットフォームインテグレーションを伴うような API は欠如しています。これらの特性のため、no_std と libcore のコードは、ブートローダー、ファームウェア、カーネルといったあらゆるブートストラップ (ステージ 0)のコードにも利用できます。

#### 1.2.3.1. 概略

| 機能 | no_std | std |
|---|---|---|
| ヒープ (動的メモリ) | * | ✓ |
| コレクション (Vec, HashMap, など) | ** | ✓ |
| スタックオーバーフロープロテクション | ✘ | ✓ |
| main 関数前の初期化コード実行 | ✘ | ✓ |
| libstd の利用 | ✘ | ✓ |
| libcore の利用 | ✓ | ✓ |
| ファームウェア、カーネル、ブートローダーのコードを書く | ✓ | ✘ |

* `alloc` クレートを使い、[alloc-cortex-m]のような適切なアロケータを使った場合のみ

** `collections` クレートを使い、グローバルなデフォルトアロケータを設定した場合のみ

未翻訳

### 1.2.4. 参照

- RFC-1184

## 1.3. ツール

マイクロコントローラを扱う際は、いくつかの異なるツールを利用することになります。あなたのノート PC とは異なるアーキテクチャを扱うことになるでしょうし、リモートデバイス上でプログラムを実行しデバッグする必要があります。

下記リストのツールを利用します。最小バージョンが指定されていない場合、新しいバージョンであれば機能するはずです。私たちがテストしたバージョンをリストに示しています。

- ARM Cortex-M コンパイルサポートを追加した Rust 1.31、1.31-beta 以上のツールチェイン。
- `cargo-binutils`~0.1.4
- `qemu-system-arm`。テストしたバージョン: 3.0.0
- OpenOCD >=0.8.テストしたバージョン: v0.9.0 と v0.10.0
- ARM サポートの GDB。バージョン 7.12 以上を強く推奨。テストしたバージョン: 7.10、7.11、7.12、8.1
- [任意] `git` または `cargo-generate`。どちらもインストールしていないのであれば、どちらをインストールしてもかまいません。

下記に、なぜこれらのツールを利用するのか、を説明します。インストール方法は、次のページにあります。

### 1.3.1. `cargo-generate` か `git`

ベアメタルプログラムは、非標準 (no_std)な Rust プログラムであり、プログラムのメモリレイアウトを正しくするために、リンクプロセスをいじる必要があります。これには、独特なファイル(リンカスクリプトなど)と設定(リンカフラグ)が必要です。 プロジェクト名やターゲットハードウェアの特徴などを、空白に入力するだけで済むように、テンプレートを用意しています。

このテンプレートは、`cargo-generate` との互換性があります。`cargo-generate` は、テンプレートから Cargo プロジェクトを作成するための Cargo のサブコマンドです。 このテンプレートは、`git` や `curl`、`wget`、ウェブブラウザを使ってダウンロードできます。

#### 1.3.2. `cargo-binutils`

`cargo-binutils` は、Rust ツールチェインとともに配布されている LLVM ツールを簡単に使うための Cargo サブコマンド一式です。 これらのツールは、LLVM の `objdump` や `nm`、`size` を含んでおり、バイナリを調査するために使われます。

GNU binutils ではなく、これらのツールを利用する利点は次の通りです。(a) LLVM ツールのインストールは、OS に関わらず、共通のコマンド 1 つ(cmd)で済みます。 (b) `objdump` などのツールは、`rustc` がサポートする全てのアーキテクチャ(ARM から x86_64 まで)をサポートします。これは、同じ LLVM バックエンドを共有しているためです。

#### 1.3.3. `qemu-system-arm`

QEMU はエミュレータです。今回は、ARM システムを完全にエミュレートできるものを使います。 私たちは、ホスト上で組込みプログラムを実行するために QEMU を利用します。 このおかげで、ハードウェアを持っていなくても、この本のいくつかの部分を試すことができます。

## 1.4. 未翻訳

#### 1.4.1. 未翻訳

未翻訳

#### 1.4.2. 未翻訳

##### 1.4.2.1. Probe-rs

未翻訳

##### 1.4.2.2. OpenOCD (Open On-Chip Debugger)

未翻訳

#### 1.4.3. 未翻訳

未翻訳

未翻訳

##### 1.4.3.1. Probe-rs Visual Studio Code Extension

未翻訳

##### 1.4.3.2. TRACE32

未翻訳

##### 1.4.3.3. GDB (GNU Debugger)

未翻訳

#### 1.4.4. 未翻訳

未翻訳

##### 1.4.4.1. Rusty-probe

未翻訳

##### 1.4.4.2. ST-Link

未翻訳

##### 1.4.4.3. J-Link

未翻訳

#### 1.4.4.4. MCU-Link

未翻訳

## 1.5. ツールのインストール

このページには、いくつかのツールの OS に依存しないインストール手順を掲載します。

#### 1.5.0.1. Rust ツールチェイン

https://rustup.rs の手順に従って、rustup をインストールします。

**注意** コンパイラのバージョンが `1.31` 以上であることを確認して下さい。`rustc -v` は下記に示す日付より新しい日付を返すべきです。

```
$ rustc -V
rustc 1.31.1 (b6c32da9b 2018-12-18)
```

翻訳が古くなっています バンド幅とディスク使用量に関する懸念から、デフォルトインストールではネイティブコンパイルのみをサポートします。ARM Cortex-M アーキテクチャのクロスコンパイラを追加するために、下記のコンパイルターゲットをインストールします。

未翻訳

```
rustup target add thumbv6m-none-eabi
```

未翻訳

```
rustup target add thumbv7m-none-eabi
```

未翻訳

```
rustup target add thumbv7em-none-eabi
```

未翻訳

```
rustup target add thumbv7em-none-eabihf
```

未翻訳

```
rustup target add thumbv8m.base-none-eabi
```

未翻訳

```
rustup target add thumbv8m.main-none-eabi
```

未翻訳

```
rustup target add thumbv8m.main-none-eabihf
```

#### 1.5.0.2. `cargo-binutils`

```
cargo install cargo-binutils

rustup component add llvm-tools
```

未翻訳

#### 1.5.0.3. `cargo-generate`

未翻訳

```
cargo install cargo-generate
```

未翻訳

#### 1.5.0.4. 未翻訳

使用している OS に特有の手順に従って下さい。

- Linux
- Windows
- macOS

### 1.5.1. Linux

いくつかの Linux ディストリビューションのインストールコマンドを示します。

#### 1.5.1.1. Packages

- Ubuntu 18.04 以上 / Debian stretch 以降

> **注記** `gdb-multiarch` は、ARM Cortex-M プログラムをデバッグするために使用する GDB のコマンドです。

```
sudo apt install gdb-multiarch openocd qemu-system-arm
```

- Ubuntu 14.04 と 16.04

> **注記** `arm-none-eabi-gdb` は、ARM Cortex-M プログラムをデバッグするために使用する GDB のコマンドです。

```
sudo apt install gdb-arm-none-eabi openocd qemu-system-arm
```

- Fedora 27 以上

```
sudo dnf install gdb openocd qemu-system-arm
```

- Arch Linux

> **注記** `arm-none-eabi-gdb` は、ARM Cortex-M プログラムをデバッグするために使用する GDB のコマンドです。

```
sudo pacman -S arm-none-eabi-gdb qemu-system-arm openocd
```

#### 1.5.1.2. udev ルール

このルールにより、ルート権限なしで、OpenOCD を Discovery ボードに対して使えるようにします。

翻訳が古くなっています 下記の内容で、`/etc/udev/rules.d` ディレクトリにファイルを作成します。

```
# STM32F3DISCOVERY rev A/B - ST-LINK/V2
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="3748", TAG+="uaccess"

# STM32F3DISCOVERY rev C+ - ST-LINK/V2-1
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", TAG+="uaccess"
```

その後、全ての udev ルールをリロードします。

```
sudo udevadm control --reload-rules
```

既にボードをノート PC に接続している場合、一度抜いてから、もう一度接続します。

これらのコマンド実行することで、パーミッションを確認できます。

```
lsusb
```

未翻訳

```
(..)
Bus 001 Device 018: ID 0483:374b STMicroelectronics ST-LINK/V2.1
(..)
```

未翻訳

```
ls -l /dev/bus/usb/001/018
```

```
crw-------+ 1 root root 189, 17 Sep 13 12:34 /dev/bus/usb/001/018
```

```
getfacl /dev/bus/usb/001/018 | grep user
```

```
user::rw-
user:you:rw-
```

翻訳が古くなっています パーミッションに追加された+は、パーミッションが拡張されたことを意味しています。

それでは、次のセクションに進んで下さい。

#### 1.5.2. macOS

翻訳が古くなっています 全てのツールは、Homebrew を使ってインストールできます。

##### 1.5.2.1. 未翻訳

```
$ # GDB
$ brew install arm-none-eabi-gdb

$ # OpenOCD
$ brew install openocd

$ # QEMU
$ brew install qemu
```

未翻訳

```
$ brew install --HEAD openocd
```

##### 1.5.2.2. 未翻訳

```
$ # GDB
$ sudo port install arm-none-eabi-gcc

$ # OpenOCD
$ sudo port install openocd

$ # QEMU
$ sudo port install qemu
```

以上です！次のセクションに進んで下さい。

### 1.5.3. Windows

#### 1.5.3.1. `arm-none-eabi-gdb`

ARM は Windows 向けに.exe インストーラを提供しています。here から 1 つを入手して、手順に従って下さい。 インストールプロセスが終了する直前に“環境変数にパスを追加”オプションを選択します。 その後、ツールが `%PATH%` にあることを確認します。

```
$ arm-none-eabi-gdb -v
GNU gdb (GNU Tools for Arm Embedded Processors 7-2018-q2-update) 8.1.0.20180315-git
(..)
```

#### 1.5.3.2. OpenOCD

未翻訳

未翻訳

```
$ openocd -v
Open On-Chip Debugger 0.10.0
(..)
```

#### 1.5.3.3. QEMU

QEMU 公式サイトから QEMU を入手します。

#### 1.5.3.4. ST-LINK USB ドライバ

USB ドライバもインストールする必要があります。そうでなければ OpenOCD は動きません。インストーラの手順に従って下さい。そして、正しいドライバのバージョン(32 ビットか 64 ビット)をインストールすることを確認して下さい。

以上です！次のセクションに進んで下さい。

### 1.5.4. インストールの確認

このセクションでは、必要となるツールとドライバが正しくインストールされ、設定されていることを確認します。

マイクロ USB ケーブルを使って、ノート PC / PC を discovery ボードに接続して下さい。discovery ボードは 2 つの USB コネクタを搭載しています。 ボード端の中央にある“USB ST-LINK”とラベルが付いたものを使用して下さい。

ST-LINK ヘッダが装着されていることも確認します。下の写真の赤丸で囲った部分が ST-LINK ヘッダです。

それでは、次のコマンドを実行して下さい。

```
openocd -f interface/stlink.cfg -f target/stm32f3x.cfg
```

未翻訳

次の出力が得られ、プログラムはコンソールをブロックするはずです。

```
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use
'transport select <transport>'.
adapter speed: 1000 kHz
```

```
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control. The results might
differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.919881
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

確認作業とは直接関係しませんが、ブレイクポイントとウォッチポイントに関する最後の行を取得したはずです。取得できた場合、OpenOCD プロセスを停止し、次のセクションへ進んで下さい。

“breakpoints”の行が取得できなかった場合、次のコマンドを試してく下さい。

```
openocd -f interface/stlink-v2.cfg -f target/stm32f3x.cfg
```

```
openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg
```

このコマンドが機能した場合、古いハードウェアリビジョンの discovery ボードを入手したことを意味します。これは問題になりませんが、後で少し設定を変える必要があるので、そのことを覚えておいて下さい。 次のセクションに進むことができます。

どちらのコマンドも通常ユーザとしてうまく動かなかった場合、root パーミッションで実行してみて下さい(例えば、`sudo openocd ..`)。コマンドが root パーミッションで機能した場合、udev ルールが正しく設定されているか確認して下さい。

ここまで到着していまい、OpenOCD が動いていないならば、issue を作って下さい。私たちがあなたを支援します。

# 2. 入門

このセクションでは、組込みプログラムを書いて、ビルドして、フラッシュに書き込み、デバッグする、という一連のプロセスを説明します。ほとんどの例を特別なハードウェアなしで試すことができます。有名なオープンソースハードウェアエミュレータである QEMU を使うからです。 ハードウェアが必要となる唯一のセクションは、当然ながら、OpenOCD を使って STM32F3DISCOVERY にプログラムする Hardware セクションです。

## 2.1. QEMU

Cortex-M3 マイクロコントローラの LM3S6965 用にプログラムを書くところから始めましょう。この LM3S6965 を最初のターゲットとして選んだ理由は、QEMU を使ってエミュレーションできるからです。このセクションでは、ハードウェアをいじる必要がなく、ツールと開発プロセスに集中できます。

未翻訳

### 2.1.1. 標準ライブラリを使わない Rust プログラム

未翻訳

#### 2.1.1.1. 未翻訳

未翻訳

```
cargo install cargo-generate
```

未翻訳

```
cargo generate --git https://github.com/knurling-rs/app-template
```

```
 Project Name: app
 Creating project called `app`...
 Done! New project created /tmp/app
```

```
cd app
```

#### 2.1.1.2. 未翻訳

未翻訳

```
git clone https://github.com/rust-embedded/cortex-m-quickstart app
cd app
```

未翻訳

```
[package]
authors = ["{{authors}}"] # "{{authors}}" -> "John Smith"
edition = "2018"
name = "{{project-name}}" # "{{project-name}}" -> "app"
version = "0.1.0"

# ..

[[bin]]
name = "{{project-name}}" # "{{project-name}}" -> "app"
test = false
bench = false
```

#### 2.1.1.3. 未翻訳

未翻訳

```
curl -LO https://github.com/rust-embedded/cortex-m-quickstart/archive/master.zip
unzip master.zip
mv cortex-m-quickstart-master app
cd app
```

未翻訳

未翻訳

### 2.1.2. 未翻訳

未翻訳

```
#![no_std]
#![no_main]

use panic_halt as _;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {
        // あなたのコードはここに書きます
    }
}
```

未翻訳

未翻訳

未翻訳

未翻訳

未翻訳

未翻訳

### 2.1.3. 未翻訳

未翻訳

```
MEMORY
{
  /* 未翻訳
  FLASH : ORIGIN = 0x00000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 64K
}

/* 未翻訳
/* _stack_start = ORIGIN(RAM) + LENGTH(RAM); */

/* 未翻訳
/* _stext = ORIGIN(FLASH) + 0x400; */

/* 未翻訳
/* SECTIONS {
     .ram2bss (NOLOAD) : ALIGN(4) {
       *(.ram2bss);
       . = ALIGN(4);
```

```
     } > RAM2
   } INSERT AFTER .bss;
*/
```

未翻訳

```
tail -n6 .cargo/config.toml
```

```
[build]
# 未翻訳
# target = "thumbv6m-none-eabi"    # Cortex-M0 and Cortex-M0+
target = "thumbv7m-none-eabi"    # Cortex-M3
# target = "thumbv7em-none-eabi"   # Cortex-M4 and Cortex-M7 (no FPU)
# target = "thumbv7em-none-eabihf" # Cortex-M4F and Cortex-M7F (with FPU)
```

未翻訳

```
rustup target add thumbv7m-none-eabi
```

未翻訳

```
cargo build --target thumbv7m-none-eabi
cargo build
```

#### 2.1.4. 未翻訳

未翻訳

未翻訳

```
cargo readobj --bin app -- --file-headers
```

未翻訳

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0x0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x405
  Start of program headers:          52 (bytes into file)
  Start of section headers:          153204 (bytes into file)
  Flags:                             0x5000200
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         19
  Section header string table index: 18
```

未翻訳

```
cargo size --bin app --release -- -A
```

未翻訳

```
app  :
section             size        addr
```

```
.vector_table       1024         0x0
.text                 92       0x400
.rodata                0       0x45c
.data                  0  0x20000000
.bss                   0  0x20000000
.debug_str          2958         0x0
.debug_loc            19         0x0
.debug_abbrev        567         0x0
.debug_info         4929         0x0
.debug_ranges         40         0x0
.debug_macinfo         1         0x0
.debug_pubnames     2035         0x0
.debug_pubtypes     1892         0x0
.ARM.attributes       46         0x0
.debug_frame         100         0x0
.debug_line          867         0x0
Total              14570
```

未翻訳

未翻訳

未翻訳

```
cargo objdump --bin app --release -- --disassemble --no-show-raw-insn --print-imm-hex
```

未翻訳

未翻訳

```
app:  file format ELF32-arm-little

Disassembly of section .text:
main:
     400: bl  #0x256
     404: b #-0x4 <main+0x4>

Reset:
     406: bl  #0x24e
     40a: movw  r0, #0x0
     < .. truncated any more instructions .. >

DefaultHandler_:
     656: b #-0x4 <DefaultHandler_>

UsageFault:
     657: strb  r7, [r4, #0x3]

DefaultPreInit:
     658: bx  lr

__pre_init:
     659: strb  r7, [r0, #0x1]
```

```
__nop:
     65a: bx  lr

HardFaultTrampoline:
     65c: mrs r0, msp
     660: b #-0x2 <HardFault_>

HardFault_:
     662: b #-0x4 <HardFault_>

HardFault:
     663: <unknown>
```

#### 2.1.5. 未翻訳

未翻訳

未翻訳

未翻訳

未翻訳

```
cargo remove defmt-rtt
cargo add defmt-semihosting
```

未翻訳

未翻訳

```
cargo build --bin hello
```

未翻訳

未翻訳

```
qemu-system-arm \
  -cpu cortex-m3 \
  -machine lm3s6965evb \
  -nographic \
  -semihosting-config enable=on,target=native \
  -kernel target/thumbv7m-none-eabi/debug/hello
```

未翻訳

```
git clone git@github.com:knurling-rs/defmt.git
cd defmt/qemu-run/
cargo run -- --machine lm3s6965evb ../qemu-rs/target/thumbv7m-none-eabi/debug/hello
```

```
Hello, world!
```

未翻訳

```
echo $?
```

```
0
```

未翻訳

未翻訳

```
head -n3 .cargo/config.toml
```

```
[target.thumbv7m-none-eabi]
# `cargo run`で、プログラムをQEMUで実行するため、コメントアウトを外して下さい。
runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel"
```

このランナーは、デフォルトのコンパイルターゲットである `thumbv7m-none-eabi` のみに適用されます。 これで、`cargo run` はプログラムをコンパイルして QEMU で実行します。

```
cargo ru(
  level: 2 + whole
)--release
```

```
   Compiling app v0.1.0 (file:///tmp/app)
    Finished release [optimized + debuginfo] target(s) in 0.26s
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel target/thumbv7m-none-eabi/release/
examples/hello`
Hello, world!
```

### 2.1.6. デバッグ

デバッグは組込み開発にとって非常に重要です。どのように行うのか、見てみましょう。

組込みデバイスのデバッグは、リモートデバッグを伴います。デバッグしたいプログラムは、デバッガプログラム（GDB または LLDB）を実行しているマシン上で実行されないためです。

リモートデバッグは、クライアントとサーバからなります。QEMU のセットアップで、 クライアントは GDB（または LLDB）プロセスとなり、サーバは組込みプログラムを実行している QEMU プロセスとなります。

このセクションでは、コンパイル済みの `hello` の例を使用します。

最初のデバッグステップは、QEMU をデバッグモードで起動することです。

```
qemu-system-arm \
  -cpu cortex-m3 \
  -machine lm3s6965evb \
  -nographic \
  -semihosting-config enable=on,target=native \
  -gdb tcp::3333 \
  -S \
  -kernel target/thumbv7m-none-eabi/debug/examples/hello
```

このコマンドは、コンソールに何も表示せず、端末をブロックします。ここでは 2 つの追加フラグを渡しています。

- `-gdb tcp::3333`。QEMU が TCP ポート 3333 番で、GDB コネクションを待つようにします。
- `-S`。QEMU が、起動時に、マシンをフリーズします。このフラグがないと、デバッガを起動する前に、プログラムが main 関数の終わりに到達してしまいます。

次に別の端末で GDB を起動し、`hello` の例のデバッグシンボルをロードします。

```
gdb-multiarch -q target/thumbv7m-none-eabi/debug/examples/hello
```

未翻訳

未翻訳

```
target remote :3333
```

```
Remote debugging using :3333
Reset () at $REGISTRY/cortex-m-rt-0.6.1/src/lib.rs:473
473     pub unsafe extern "C" fn Reset() -> ! {
```

未翻訳

未翻訳

未翻訳

```
core::num::bignum::Big32x40::mul_small () at src/libcore/num/bignum.rs:254 src/
libcore/num/bignum.rs: No such file or directory.
```

未翻訳

未翻訳

```
list main
```

未翻訳

```
6       use panic_halt as _;
7
8       use cortex_m_rt::entry;
9       use cortex_m_semihosting::{debug, hprintln};
10
11      #[entry]
12      fn main() -> ! {
13          hprintln!("Hello, world!").unwrap();
14
15          // exit QEMU
```

未翻訳

```
break 13
```

未翻訳

```
continue
```

```
Continuing.

Breakpoint 1, hello::__cortex_m_rt_main () at examples\hello.rs:13
13          hprintln!("Hello, world!").unwrap();
```

「Hello, world!」を表示するコードに近づいてきました。 `next` コマンドを使って、先へ進みましょう。

```
next
```

```
16          debug::exit(debug::EXIT_SUCCESS);
```

この時点で、`qemu-system-arm` を実行している端末に「Hello, world」が表示されるはずです。

```
$ qemu-system-arm (..)
Hello, world!
```

もう 1 度 `next` を実行すると、QEMU プロセスが終了します。

```
next
```

```
[Inferior 1 (Remote target) exited normally]
```

これで GDB セッションを終了できます。

```
quit
```

# 3. ハードウェア

ここまでで、ツールと開発プロセスにある程度慣れたはずです。このセクションでは、実際のハードウェアに切り替えます。 開発プロセスは、ほとんど同じままです。飛び込みましょう。

### 3.0.1. ハードウェアを知る

始める前に、プロジェクトの設定に利用するターゲットデバイスのいくつかの特徴を確認する必要があります。

- ARM コア、例えば Cortex-M3 です。
- その ARM コアは FPU を搭載していますか？Cortex-M4**F** と Cortex-M7**F** は、搭載しています。
- ターゲットデバイスに搭載されているフラッシュメモリと RAM の容量はいくらですか？例えば、フラッシュは 256KiB で RAM は 32KiB です。
- フラッシュメモリと RAM は、アドレス空間のどこにマッピングされていますか？ 例えば、RAM は、通常 `0x2000_0000` 番地に位置します。

これらの情報は、デバイスのデータシートかリファレンスマニュアルに掲載されています。

このセクションでは、私たちのリファレンスハードウェアである STM32F3DISCOVERY を使用します。 このボードは、STM32F303VCT6 マイクロコントローラを 1 つ搭載しています。このマイクロコントローラは以下のものを持っています。

- 単精度 FPU を含む Cortex-M4F コアが 1 つ
- 0x0800_0000 番地に配置された 256KiB のフラッシュメモリ
- 0x2000_0000 番地に配置された 40KiB の RAM。（別の RAM 領域もありますが、説明の簡単化のため、取り扱

### 3.0.2. 設定

テンプレートの新しいインスタンスを使って、スクラッチから書いていきましょう。 `cargo-generate` を使用しない方法については、前セクションの QEMU を参照して下さい。

```
$ cargo generate --git https://github.com/rust-embedded/cortex-m-quickstart
 Project Name: app
 Creating project called `app`...
 Done! New project created /tmp/app

$ cd app
```

第一ステップは、`.cargo/config` にデフォルトコンパイルターゲットを設定することです。

```
tail -n5 .cargo/config.toml
```

```
# 以下のコンパイルターゲットから1つを選びます
# target = "thumbv6m-none-eabi"    # Cortex-M0およびCortex-M0+
# target = "thumbv7m-none-eabi"    # Cortex-M3
# target = "thumbv7em-none-eabi"   # Cortex-M4およびCortex-M7 (no FPU)
target = "thumbv7em-none-eabihf" # Cortex-M4FおよびCortex-M7F (with FPU)
```

Cortex-M4F コアを対象とするものとして、`thumbv7em-none-eabihf` を使います。

未翻訳

第二ステップは、`memory.x` ファイルにメモリ領域の情報を入力することです。

```
$ cat memory.x
/* Linker script for the STM32F303VCT6 */
MEMORY
```

```
{
  /* NOTE 1 K = 1 KiBi = 1024 bytes */
  FLASH : ORIGIN = 0x08000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 40K
}
```

未翻訳

未翻訳

debug::exit()の呼び出しが、コメントアウトされているか削除されていることを確認して下さい。 これは、QEMU で実行する時のみ、使用します。

```
#[entry]
fn main() -> ! {
    hprintln!("Hello, world!").unwrap();

    // QEMUを終了する
    // 注記、ハードウェア上で実行しないで下さい。OpenOCDの状態を破壊する可能性があります。
    // debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

これまでやってきた通り、cargo build でプログラムをクロスコンパイルし、 cargo-binutils でバイナリを調べることができます。cortex-m-rt クレートは、チップを動作させるために必要な、全てのおまじないを処理します。 便利なことに、ほとんど全ての Cortex-M CPU が同じ方法で起動します。

```
cargo build --example hello
```

### 3.0.3. デバッグ

デバッグ方法は少し違います。実際、最初のステップは、ターゲットデバイスによって異なります。 このセクションでは、STM32F3DISCOVERY 上で実行しているプログラムをデバッグするために必要となる手順を説明します。これは、参考の役目を果たします。デバイス固有のデバッグ情報は、 the Debugonomicon を参照して下さい。

以前と同様に、リモートデバッグを行います。クライアントが GDB プロセスであることも同様です。 しかし、今回、サーバは OpenOCD になります。

インストールの確認セクションでやったように、ノート PC または PC を discovery ボードに接続し、 ST-LINK ヘッダが設定されていることを確認して下さい。

discovery ボードの ST-LINK に接続するために、端末で openocd を実行して下さい。 このコマンドは、テンプレートプロジェクトのルートディレクトリから実行して下さい。 openocd は、どのインタフェースファイルとターゲットファイルを使うか、が記述されている openocd.cfg ファイルを見つけます。

```
cat openocd.cfg
```

```
# STM32F3DISCOVERY開発ボード用のOpenOCD設定サンプル

# 持っているハードウェアのリビジョンに応じて、これらのインタフェースのうち、1つを選んで下さい。
# 常に、1つのインタフェースがコメントアウトされているべきです。

# リビジョンC （新しいリビジョン）
```

```
source [find interface/stlink.cfg]

# リビジョンAとB（古いリビジョン）
# source [find interface/stlink-v2.cfg]

source [find target/stm32f3x.cfg]
```

> **注記** インストールの確認セクションで、古いバージョンの discovery ボードを持っていることが判明している場合、 `interface/stlink-v2.cfg` を使うように `openocd.cfg` ファイルを修正する必要があります。

```
$ openocd
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use
'transport select <transport>'.
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control. The results might
differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.913879
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

別の端末で、GDB を実行します。こちらも、テンプレートプロジェクトのルートディレクトから実行して下さい。

```
gdb-multiarch -q target/thumbv7em-none-eabihf/debug/examples/hello
```

未翻訳

次に、TCP 3333 ポートで接続待ちしている OpenOCD に、GDB を接続します。

```
(gdb) target remote :3333
Remote debugging using :3333
0x00000000 in ?? ()
```

それでは、`load` コマンドを使って、マイクロコントローラにプログラムを書き込んで下さい。

```
(gdb) load
Loading section .vector_table, size 0x400 lma 0x8000000
Loading section .text, size 0x1518 lma 0x8000400
Loading section .rodata, size 0x414 lma 0x8001918
Start address 0x08000400, load size 7468
Transfer rate: 13 KB/sec, 2489 bytes/write.
```

プログラムがロードされました。このプログラムはセミホスティングを使用します。そこで、セミホスティングを呼び出して何かを行う前に、OpenOCD にセミホスティングを有効にするように、 指示する必要があります。

```
(gdb) monitor arm semihosting enable
semihosting is enabled
```

`monitor help` コマンドを実行することで、全ての OpenOCD コマンドを見ることができます。

以前のように、ブレイクポイントと `continue` コマンドを使用することで、`main` までスキップすることができます。

```
(gdb) break main
Breakpoint 1 at 0x8000490: file examples/hello.rs, line 11.
Note: automatically using hardware breakpoints for read-only addresses.

(gdb) continue
Continuing.

Breakpoint 1, hello::__cortex_m_rt_main_trampoline () at examples/hello.rs:11
11      #[entry]
```

未翻訳

未翻訳

```
(gdb) step
halted: PC: 0x08000496
hello::__cortex_m_rt_main () at examples/hello.rs:13
13          hprintln!("Hello, world!").unwrap();
```

翻訳が古くなっています この時点で、OpenOCD コンソールに、他のものと入り混じって「Hello, world!」と表示されるはずです。

```
$ openocd
(..)
Info : halted: PC: 0x08000502
Hello, world!
Info : halted: PC: 0x080004ac
Info : halted: PC: 0x080004ae
Info : halted: PC: 0x080004b0
Info : halted: PC: 0x080004b4
Info : halted: PC: 0x080004b8
Info : halted: PC: 0x080004bc
```

未翻訳

未翻訳

```
(gdb) quit
A debugging session is active.

        Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n)
```

未翻訳

```
cat openocd.gdb
```

```
target extended-remote :3333

# print demangled symbols
set print asm-demangle on

# detect unhandled exceptions, hard faults and panics
break DefaultHandler
break HardFault
break rust_begin_unwind

monitor arm semihosting enable

load

# start the process but immediately halt the processor
stepi
```

`<gdb> -x openocd.gdb target/thumbv7em-none-eabihf/debug/examples/hello` を実行することで、GDB はすぐに OpenOCD に接続し、セミホスティングを有効化し、プログラムをロードした上で、プロセスを開始します。

別の方法として、`<gdb> -x openocd.gdb` をカスタムランナーにして、`cargo run` でプログラムをビルドし、さらに GDB セッションを開始することもできます。このランナーは、`.cargo/config` に含まれていますが、 コメントアウトされています。

```
head -n10 .cargo/config.toml
```

```
[target.thumbv7m-none-eabi]
# ここのコメントアウトを外すと、`cargo run`はQEMUでプログラムを実行します
# runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel"

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
# 3つの選択肢のうち、1つのコメントアウトを外すと、`cargo run`はGDBセッションを開始します。
# どの選択肢を使うか、は対象システムによって異なります。
runner = "arm-none-eabi-gdb -x openocd.gdb"
# runner = "gdb-multiarch -x openocd.gdb"
# runner = "gdb -x openocd.gdb"
```

```
$ cargo run --example hello
(..)
Loading section .vector_table, size 0x400 lma 0x8000000
Loading section .text, size 0x1e70 lma 0x8000400
Loading section .rodata, size 0x61c lma 0x8002270
Start address 0x800144e, load size 10380
Transfer rate: 17 KB/sec, 3460 bytes/write.
(gdb)
```

## 3.1. メモリマップドレジスタ

組込みシステムでは、通常の Rust コードを実行し、データを RAM 内で移動させるだけではたいしたことはできません。 LED の点滅やボタンの押下検出、もしくは、バス上のオフチップペリフェラルとの通信など、システムが情報を入出力するには、ペリフェラルとその「メモリマップドレジスタ」の世界に足を踏み入れる必要があります。

マイクロコントローラのペリフェラルにアクセスするためのコードが、次のいずれかのレベルで、既に書かれています。

- マイクロアーキテクチャクレート。この種のクレートは、マイクロコントローラに搭載されているプロセッサコアで共通となる便利なルーチンを扱っています。また、特定のプロセッサコアを使用する全てのマイクロコントローラに共通のペリフェラルも取り扱います。 例えば、cortex-m クレートは、割り込みの有効化と無効化を行う関数を提供しています。これは全ての Cortex-M ベースマイクロコントローラで同じものです。 cortex-m クレートは、「SysTick」ペリフェラルへのアクセスも提供しています。このペリフェラルは、全ての Cortex-M ベースマイクロコントローラに搭載されています。
- ペリフェラルアクセスクレート（PAC）。この種のクレートは、薄いラッパーです。特定の型番のマイクロコントローラに対して定義されている、 様々なメモリマップドレジスタのラッパーを提供します。例えば、テキサスインスツルメンツの Tiva-C TM4C123 シリーズ向けの tm4c123x クレートや、ST マイクロの STM32F30x シリーズ向けの stm32f30x クレートです。マイクロコントローラのテクニカルリファレンスマニュアルに記載されている各ペリフェラルの操作手順に従って、 レジスタと直接やり取りします。
- HAL クレート。これらのクレートは、特定のプロセッサに対して、よりユーザフレンドリな API を提供しています。embedded-hal で定義されている共通のトレイトを使って実装されていることが多いです。例えば、このクレートは、`Serial` 構造体を提供しているでしょう。そのコンストラクタは、適切な GPIO ピンの一式とボーレートを引数に取ります。そして、データを送信するための `write_byte` 関数一式を提供します。 embedded-hal に関する詳細は、移植性の章を参照して下さい。
- ボードクレート。これらのクレートは、HAL クレートのさらに一歩先を進んでいます。これらは、STM32F3DISCOVERY ボード向けの stm32f3-discovery のように、 特定の開発キットやボード向けに、様々なペリフェラルと GPIO ピンを事前に設定してあります。

### 3.1.1. 未翻訳

未翻訳

未翻訳

未翻訳

### 3.1.2. 未翻訳

全ての Cortex-M マイクロコントローラで共通の SysTick ペリフェラルから見ていきましょう。cortex-m クレートにはかなり低レベルな API があり、次のように使うことができます。

```
#![no_std]
#![no_main]
use cortex_m::peripheral::{syst, Peripherals};
use cortex_m_rt::entry;
use panic_halt as _;

#[entry]
fn main() -> ! {
    let peripherals = Peripherals::take().unwrap();
    let mut systick = peripherals.SYST;
    systick.set_clock_source(syst::SystClkSource::Core);
    systick.set_reload(1_000);
    systick.clear_current();
    systick.enable_counter();
    while !systick.has_wrapped() {
        // ループ
    }
```

```
    loop {}
}
```

SYST 構造体の関数は、ARM テクニカルリファレンスマニュアルにおいて、このペリフェラルに定義されている機能と非常によく似ています。「X ミリ秒遅延」といった具合の API はありません。`while` ループを使って愚直に実装する必要があります。`Peripherals::take()`を呼び出すまでは、 SYST 構造体にアクセスできないことに注意して下さい。これは、プログラム全体で唯一の SYST 構造体が存在することを保証する特別な手順です。詳しくは、ペリフェラルセクションをご覧下さい。

### 3.1.3. ペリフェラルアクセスクレート（PAC）の使用

全ての Cortex-M に搭載されている基本的なペリフェラルのみに限定するのであれば、組込みソフトウェア開発はあまり進まないでしょう。どこかの時点で、使用している特定のマイクロコントローラ固有のコードを書く必要があります。今回の例では、テキサスインスツルメンツの TM4C123 があるとしましょう。TM4C123 はミドルレンジのマイクロコントローラで、80MHz の Cortex-M4 と 256 KiB のフラッシュメモリが搭載されています。このチップを利用するために、tm4c123x クレートを取得します。

```
#![no_std]
#![no_main]

use panic_halt as _; // パニックハンドラ

use cortex_m_rt::entry;
use tm4c123x;

#[entry]
pub fn init() -> (Delay, Leds) {
    let cp = cortex_m::Peripherals::take().unwrap();
    let p = tm4c123x::Peripherals::take().unwrap();

    let pwm = p.PWM0;
    pwm.ctl.write(|w| w.globalsync0().clear_bit());
    // モード1は カウントアップ/ダウンモード
    pwm._2_ctl.write(|w| w.enable().set_bit().mode().set_bit());
    pwm._2_gena.write(|w| w.actcmpau().zero().actcmpad().one());
    // 528サイクル（264カウントアップとカウントダウン）は、ビデオラインごとに4ループ（2112サイクル）
    pwm._2_load.write(|w| unsafe { w.load().bits(263) });
    pwm._2_cmpa.write(|w| unsafe { w.compa().bits(64) });
    pwm.enable.write(|w| w.pwm4en().set_bit());
}
```

先ほど SYST にアクセスした時と全く同じ方法で、`PWM0` ペリフェラルにアクセスします。違う点は、`tm4c123x::Peripherals::take()`を呼ぶことです。このクレートは、svd2rust を使って自動生成されたものです。レジスタフィールドのアクセス関数は、数値の引数ではなく、クロージャを取ります。このコードは量が多いように見えますが、Rust コンパイラは一連のチェックを実行し、手書きのアセンブラに近いマシンコードを生成します。自動生成されたコードが、特定のアクセサ関数への全引数が有効であることを判断できない場合、その関数は `unsafe` とマークされます。例えば、SVD がレジスタを 32 ビットと定義しているが、それらの 32 ビット値の一部が特別な意味を持つかどうか、記述していない場合です。上記の例では、`bits()`関数を使って `load` と `compa` サブフィールドを設定する時に、`unsafe` をマークしています。

#### 3.1.3.1. 読み込み

`read()`関数は、メーカーの SVD ファイルで定義されている通り、レジスタ内の様々なサブフィールドに対して、読み込み専用のアクセスオブジェクトを返します。特定チップ上にある、特定ペリフェラルの、特定レジスタに対して、固有の返り値 R 型があり、この R 型で使える全ての関数は、tm4c123x ドキュメントで見ることができます。

```
if pwm.ctl.read().globalsync0().is_set() {
    // 処理をする
}
```

#### 3.1.3.2. 書き込み

`write()`関数は、単一引数のクロージャを取ります。通常は、この引数を `w` と呼びます。この引数は、チップメーカーが SVD ファイルで定義している通り、様々なレジスタのサブフィールドへの読み書きアクセスを許可します。特定チップ上にある、特定ペリフェラルの、特定レジスタに対して、`w` 型で使える全ての関数も、tm4c123x ドキュメントで見ることができます。設定していない全てのサブフィールドは、デフォルト値に設定されます。レジスタの既存の内容は失われます。

```
pwm.ctl.write(|w| w.globalsync0().clear_bit());
```

#### 3.1.3.3. 修正

レジスタの特定のサブフィールドだけを変更して、残りのサブフィールドは変更したくない場合、`modify` 関数を使えます。この関数は 2 引数のクロージャを取ります。1 つは読み込み用で、もう 1 つは書き込み用です。通常、これらの引数をそれぞれ、`r` と `w` と呼びます。 `r` 引数は、レジスタの現在の内容を調べるために使用されます。そして、`w` 引数は、レジスタの内容を修正するために使用されます。

```
pwm.ctl.modify(|r, w| w.globalsync0().clear_bit());
```

`modify` 関数は、クロージャの本領を発揮します。C 言語では、一時変数に読み込み、正しいビットを修正してから、その値を書き戻す必要があります。これは、エラーが発生するかなりの余地があることを示しています。

```
uint32_t temp = pwm0.ctl.read();
temp |= PWM0_CTL_GLOBALSYNC0;
pwm0.ctl.write(temp);
uint32_t temp2 = pwm0.enable.read();
temp2 |= PWM0_ENABLE_PWM4EN;
pwm0.enable.write(temp); // ああ！間違った変数です！
```

### 3.1.4. HAL クレートの使用

あるチップ用の HAL クレートは、典型的には、PAC によって公開されている生の構造体に対して、カスタムトレイトを実装することで機能しています。大抵、このトレイトは、単独のペリフェラルには `constrain()`関数を定義し、複数ピンを利用する GPIO ポートのようなものには `split()`関数を定義します。この関数は、下層の生のペリフェラル構造体オブジェクトを消費し、より高レベルな API を備える新しいオブジェクトを返します。 この API は、シリアルポートの `new` 関数が、`Clock` 構造体オブジェクトの借用を必要とするようなことをするかもしれません。Clock 構造体オブジェクトは、PLL と全てのクロック周波数とを設定する関数呼び出しによってのみ、生成することが可能です。この方法では、最初にクロックレートを設定しないでシリアルポートオブジェクトを作成したり、 シリアルポートオブジェクトがボーレートをクロック数に誤って変換するようなことは、静的に起こり得ません。 一部のクレートでは、各 GPIO が取り得る状態のための特別なトレイトを定義することさえあります。このトレ

イトは、ペリフェラルにピンを渡す前に、ユーザがピンを正しい状態（例えば、適切な Alternate Function モードを選択することによって）にすることを求めます。これらは全て、ランタイムのコストを必要としません。

例を見てみましょう。

```
#![no_std]
#![no_main]

use panic_halt as _; // panic handler

use cortex_m_rt::entry;
use tm4c123x_hal as hal;
use tm4c123x_hal::prelude::*;
use tm4c123x_hal::serial::{NewlineMode, Serial};
use tm4c123x_hal::sysctl;

#[entry]
fn main() -> ! {
    let p = hal::Peripherals::take().unwrap();
    let cp = hal::CorePeripherals::take().unwrap();

    // SYSCTL構造体をより高レイヤなAPIオブジェクトでラップします
    let mut sc = p.SYSCTL.constrain();
    // オシレータの設定値を選択します
    sc.clock_setup.oscillator = sysctl::Oscillator::Main(
        sysctl::CrystalFrequency::_16mhz,
        sysctl::SystemClock::UsePll(sysctl::PllOutputFrequency::_80_00mhz),
    );
    // PLLをそれらの設定値で設定します
    let clocks = sc.clock_setup.freeze();

    // GPIO_PORTA構造体をより高レイヤなAPIオブジェクトでラップします。
    // GPIOペリフェラルに自動的に電源を入れるために、
    // `sc.power_control`の借用が必要なことに留意して下さい。
    let mut porta = p.GPIO_PORTA.split(&sc.power_control);

    // UARTを起動します。
    let uart = Serial::uart0(
        p.UART0,
        // 送信ピン
        porta
            .pa1
            .into_af_push_pull::<hal::gpio::AF1>(&mut porta.control),
        // 受信ピン
        porta
            .pa0
            .into_af_push_pull::<hal::gpio::AF1>(&mut porta.control),
        // RTSとCTSは必要としません
        (),
        (),
        // ボーレート
        115200_u32.bps(),
        // 出力制御
        NewlineMode::SwapLFtoCRLF,
        // ボーレートの除数を計算するためにクロックレートが必要です
        &clocks,
```

```
        // UARTペリフェラルの電源を入れるために必要です
        &sc.power_control,
    );

    loop {
        writeln!(uart, "Hello, World!\r\n").unwrap();
    }
}
```

## 3.2. セミホスティング

セミホスティングは、組込みデバイスがホスト上で I/O を行う仕組みです。主に、ホストのコンソールにログ出力するために使われます。セミホスティングには、デバッグセッションが必要ですが、他には何も必要としません（追加の配線は不要です）。そのため、非常に便利です。欠点は、非常に低速であることです。ハードウェアデバッガ（例えば、ST-Link）によっては、書き込み操作が数ミリ秒かかります。

`cortex-m-semihosting` クレートは、Cortex-M デバイス上でセミホスティング操作をするための API を提供します。下のプログラムは、セミホスティングバージョンの「Hello, world!」です。

```
#![no_main]
#![no_std]

use panic_halt as _;

use cortex_m_rt::entry;
use cortex_m_semihosting::hprintln;

#[entry]
fn main() -> ! {
    hprintln!("Hello, world!").unwrap();

    loop {}
}
```

このプログラムをハードウェア上で実行すると、OpenOCD のログに、「Hello world!」のメッセージが表示されます。

```
$ openocd
(..)
Hello, world!
(..)
```

最初に、GDB から OpenOCD のセミホスティングを有効化する必要があります。

```
(gdb) monitor arm semihosting enable
semihosting is enabled
```

QEMU はセミホスティング操作を理解しているため、上のプログラムは、デバッグセッションを開始していない `qemu-system-arm` でも動作します。セミホスティングサポートを有効化するため、QEMU に `-semihosting-config` フラグを渡す必要があることに注意して下さい。これらのフラグは、テンプレートの `.cargo/config` ファイルに既に含まれています。

```
$ # このプログラムは端末をブロックします
$ cargo run
     Running `qemu-system-arm (..)
Hello, world!
```

exit セミホスティング操作もあり、QEMU プロセスを終了するために使われます。重要：ハードウェア上で debug::exit を**使用しない**で下さい。この関数は、OpenOCD セッションを破壊する可能性があり、OpenOCD を再起動しない限り、それ以上のプログラムのデバッグができなくなります。

```
#![no_main]
#![no_std]

use panic_halt as _;

use cortex_m_rt::entry;
use cortex_m_semihosting::debug;

#[entry]
fn main() -> ! {
    let roses = "blue";

    if roses == "red" {
        debug::exit(debug::EXIT_SUCCESS);
    } else {
        debug::exit(debug::EXIT_FAILURE);
    }

    loop {}
}
```

```
$ cargo run
     Running `qemu-system-arm (..)

$ echo $?
1
```

最後のヒント：パニック時の挙動を、exit(EXIT_FAILURE)に設定することができます。これで、QEMU 上で実行できる no_std ランパステストを書くことができます。

利便性のために、panic-semihosting クレートは、「exit」フィーチャを持っています。 このフィーチャが有効化されていると、ホストの標準エラーにパニックメッセージをログ出力した後、exit(EXIT_FAILURE)を呼び出します。

```
#![no_main]
#![no_std]

use panic_semihosting as _; // features = ["exit"]

use cortex_m_rt::entry;
use cortex_m_semihosting::debug;

#[entry]
fn main() -> ! {
    let roses = "blue";

    assert_eq!(roses, "red");

    loop {}
}
```

```
$ cargo run
     Running `qemu-system-arm (..)
```

```
panicked at 'assertion failed: `(left == right)`
  left: `"blue"`,
 right: `"red"`', examples/hello.rs:15:5

$ echo $?
1
```

未翻訳

```
panic-semihosting = { version = "VERSION", features = ["exit"] }
```

未翻訳

### 3.3. パニック

パニックはRustのコア部分です。インデックス操作のような言語組込みの操作は、メモリ安全性をランタイム時に検査されます。範囲外のインデックスにアクセスしようとすると、パニックが発生します。

標準ライブラリでは、パニックは定義された動作です。ユーザがパニック発生時にプログラムをアボートする選択をしない限り、パニックを起こしたスレッドのスタックを巻き戻します。

しかし、非標準のプログラムでは、パニック時の挙動は、未定義のままです。`#[panic_handler]`関数を宣言することにより、挙動を選択することができます。この関数は、プログラムの依存関係グラフに、**1回だけ**現れる必要があります。 そして、 `fn(&PanicInfo) -> !`のシグネチャを持つ必要があります。 ここで、`PanicInfo` は、パニックした位置情報を含む構造体です。

組込みシステムは、ユーザとやり取りするものから、安全性が重要な（クラッシュできない）ものまであります。そのため、全てのパニック時動作に対応できる唯一のものはありませんが、よく利用される挙動がたくさんあります。これらの一般的な挙動が、`#[panic_handler]`関数を定義するクレートにまとめられています。 いくつか、例を挙げます。

- `panic-abort`。パニックが発生すると、アボート命令を実行します。
- `panic-halt`。パニックが発生すると、プログラム、または、現在のスレッドは、無限ループに入ることで停止します。
- `panic-itm`。パニック発生時のメッセージは、ARM Cortex-M固有のペリフェラルであるITMを使ってログ出力されます。
- `panic-semihosting`。パニック発生時のメッセージは、セミホスティングを使ってログ出力されます。

crates.ioで`panic-handler`をキーワードに検索することで、さらにクレートを見つけることができます。

プログラムは、対応するクレートとリンクすることで、これらの挙動の中から1つを選びます。パニック時の挙動がアプリケーションソースコードの中で単一行で表現されていることは、ドキュメントとして有用なだけでなく、 パニック時の挙動をコンパイル時のプロファイルで変更にする時にも利用できます。 例えば

```
#![no_main]
#![no_std]

// 開発プロファイル：パニックのデバッグを容易にします。`rust_begin_unwind`にブレイクポイントを置くことを可能にします。
#[cfg(debug_assertions)]
use panic_halt as _;
```

```
// リリースプロファイル：アプリケーションのバイナリサイズを最小化します。
#[cfg(not(debug_assertions))]
use panic_abort as _;

// ..
```

翻訳が古くなっています この例では、開発プロファイルでビルド（cargo build）した時は、panic-halt クレートとリンクします。しかし、リリースプロファイルでビルド（cargo build --release）した時は、panic-abort クレートとリンクします。

未翻訳

#### 3.3.1. 例

配列の長さを超えてアクセスしようとする例を示します。この操作はパニックを引き起こします。

```
#![no_main]
#![no_std]

use panic_semihosting as _;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    let xs = [0, 1, 2];
    let i = xs.len();
    let _y = xs[i]; // 範囲外アクセス

    loop {}
}
```

この例では、panic-semihosting の挙動を選択しており、パニックメッセージは、セミホスティングを使ってホストコンソールに出力されます。

```
$ cargo run
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb (..)
panicked at 'index out of bounds: the len is 3 but the index is 4', src/main.rs:12:13
```

挙動を panic-halt に変更し、その場合にメッセージが出力されないことを確認することができます。

### 3.4. 例外

例外と割り込みは、プロセッサが非同期イベントと致命的なエラー（例えば、不正な命令の実行）を扱うためのハードウェアの仕組みです。例外はプリエンプションを意味し、例外ハンドラを呼び出します。例外ハンドラは、イベントを引き起こした信号に応答して実行されるサブルーチンです。

cortex-m-rt クレートは、例外ハンドラを宣言するために、exception アトリビュートを提供しています。

```
// SysTick（システムタイマ）例外のための例外ハンドラ
#[exception]
fn SysTick() {
```

```
    // ..
}
```

exception 属性の他は、例外ハンドラは普通の関数のように見えます。しかし、もう 1 つ違いがあります。exception ハンドラはソフトウェアから呼び出すことができません。前述の例では、SysTick();というステートメントは、 コンパイルエラーになります。

この動作は、非常に意図的なものです。これは exception ハンドラ内で宣言された static mut 変数の利用を安全にする、という機能を提供するためのものです。

```
#[exception]
fn SysTick() {
    static mut COUNT: u32 = 0;
    // `COUNT`は`&mut u32`の型をもっており、その利用は安全です
    *COUNT += 1;
```

ご存知かもしれませんが、static mut 変数を関数内で使うことは、その関数を再入不可能にします。直接的または間接的に、複数の例外・割り込みハンドラから、もしくは、main と 1 つ以上の例外・割り込みハンドラから、再進入不可能な関数を呼び出すことは、未定義動作です。

翻訳が古くなっています 安全な Rust は、決して未定義動作になりません。そのため、再入不可能な関数は、unsafe とマークされなければなりません。 それでも、exception ハンドラは static mut な変数を安全に使える、と述べました。これが可能なのは、どうしてでしょうか。exception ハンドラはソフトウェアから呼び出すことができないため、再入する可能性はありません。だから、安全に使えるのです。

未翻訳

未翻訳

未翻訳

#### 3.4.1. 完全な例

SysTick 例外を大体 1 秒毎に発生させるシステムタイマの例を使います。 SysTick 例外ハンドラは、呼び出された回数を COUNT 変数に記録し、セミホスティングを使ってホストコンソールに COUNT の値を出力します。

**注記**：この例は、どの Cortex-M デバイスでも実行できます。QEMU 上でも実行可能です。

```
#![deny(unsafe_code)]
#![no_main]
#![no_std]

use panic_halt as _;

use core::fmt::Write;

use cortex_m::peripheral::syst::SystClkSource;
use cortex_m_rt::{entry, exception};
use cortex_m_semihosting::{
    debug,
    hio::{self, HostStream},
};
```

```
#[entry]
fn main() -> ! {
    let p = cortex_m::Peripherals::take().unwrap();
    let mut syst = p.SYST;

    // 毎秒SysTick例外を起こすためのシステムタイマを設定します
    syst.set_clock_source(SystClkSource::Core);
    // デフォルトのCPUクロックが12MHzのLM3S6965向けの設定です
    syst.set_reload(12_000_000);
    syst.clear_current();
    syst.enable_counter();
    syst.enable_interrupt();

    loop {}
}

#[exception]
fn SysTick() {
    static mut COUNT: u32 = 0;
    static mut STDOUT: Option<HostStream> = None;

    *COUNT += 1;

    // 遅延初期化
    if STDOUT.is_none() {
        *STDOUT = hio::hstdout().ok();
    }

    if let Some(hstdout) = STDOUT.as_mut() {
        write!(hstdout, "{}", *COUNT).ok();
    }

    // 重要。実際のハードウェアで実行するときは`if`ブロックを削除して下さい。そうでなければ、
    // デバッガが不整合な状態に陥るでしょう。
    if *COUNT == 9 {
        // QEMUプロセスを終了します
        debug::exit(debug::EXIT_SUCCESS);
    }
}
```

```
tail -n5 Cargo.toml
```

```
[dependencies]
cortex-m = "0.5.7"
cortex-m-rt = "0.6.3"
panic-halt = "0.2.0"
cortex-m-semihosting = "0.3.1"
```

```
$ cargo run --release
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb (..)
123456789
```

Discovery ボードでこのコードを実行すると、OpenOCD コンソールに出力を確認できるでしょう。 プログラムは、カウントが 9 に到達しても停止しません。

### 3.4.2. デフォルト例外ハンドラ

`exception` アトリビュートが実際に行っていることは、特定の例外を処理するデフォルト例外ハンドラのオーバーライドです。 特定の例外について、ハンドラをオーバーライドしない場合、`DefaultHandler` 関数がその例外を処理します。 DefaultHandler 関数は下記の通りです。

```
fn DefaultHandler() {
    loop {}
}
```

この関数は、`cortex-m-rt` クレートによって提供されており、`#[no_mangle]`とマークされています。 そのため、「DefaultHandler」にブレイクポイントを設定することができ、未処理の例外を捕捉することができます。

`exception` アトリビュートを使うことで、`DefaultHandler` をオーバーライドできます。

```
#[exception]
fn DefaultHandler(irqn: i16) {
    // カスタムデフォルトハンドラ
}
```

`irqn` 引数は、どの例外が処理されているかを示します。負の値は、Cortex-M の例外が処理されていることを意味します。ゼロまたは正の値は、デバイス固有の例外、すなわち、割り込みが処理されていること、を示しています。

### 3.4.3. ハードフォールトハンドラ

`HardFault` 例外は、少し特別です。この例外は、プログラムが不正な状態になった場合に発生します。そのため、このハンドラはリターンすることができず、未定義動作を引き起こす可能性があります。 ランタイムクレートは、デバッグ性を向上するために、ユーザ定義の `HardFault` ハンドラが呼び出される前に、少し仕事をします。

その結果、`HardFault` ハンドラは、`fn(&ExceptionFrame) -> !`のシグネチャを持つ必要があります。 ハンドラの引数は、例外によってスタックにプッシュされたレジスタへのポインタです。これらのレジスタは、例外が発生した瞬間のプロセッサステートのスナップショットで、ハードフォールトの原因を突き止めるのに便利です。

不正な操作を行う例を示します。存在しないメモリ位置への読み込みです。

> **注記**：このプログラムは、QEMU 上ではうまく動きません。つまり、クラッシュしません。`qemu-system-arm -machine lm3s6965evb` はメモリの読み込みをチェックしないため、無効なメモリを読み込むと、幸いにも、`0` を返します。

```
#![no_main]
#![no_std]

use panic_halt as _;

use core::fmt::Write;
use core::ptr;

use cortex_m_rt::{entry, exception, ExceptionFrame};
use cortex_m_semihosting::hio;

#[entry]
fn main() -> ! {
```

```
    // 存在しないメモリ位置を読み込みます
    unsafe {
        ptr::read_volatile(0x3FFF_0000 as *const u32);
    }

    loop {}
}

#[exception]
fn HardFault(ef: &ExceptionFrame) -> ! {
    if let Ok(mut hstdout) = hio::hstdout() {
        writeln!(hstdout, "{:#?}", ef).ok();
    }

    loop {}
}
```

HardFault ハンドラは、ExceptionFrame の値を表示します。実行すると、 OpenOCD コンソールに次のような表示が見えるでしょう。

```
$ openocd
(..)
ExceptionFrame {
    r0: 0x3fff0000,
    r1: 0x00000003,
    r2: 0x080032e8,
    r3: 0x00000000,
    r12: 0x00000000,
    lr: 0x080016df,
    pc: 0x080016e2,
    xpsr: 0x61000000,
}
```

pc の値は、例外発生時のプログラムカウンタの値で、例外を引き起こした命令を指しています。

プログラムのディスアセンブル結果を見ます。

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
(..)
ResetTrampoline:
 8000942:       movw    r0, #0xfffe
 8000946:       movt    r0, #0x3fff
 800094a:       ldr     r0, [r0]
 800094c:       b       #-0x4 <ResetTrampoline+0xa>
```

翻訳が古くなっています ロード命令（ldr r0, [r0]）が例外を発生させたことがわかります。そして、この時の r0 レジスタの値は、0x3fff_fffe です。この値は、ExceptionFrame の r0 フィールドと一致します。

### 3.5. 割り込み

割り込みは様々な点で例外と違いますが、その動作と使用方法は、ほとんど同じで、同じ割り込みコントローラによって処理されます。 例外が Cortex-M アーキテクチャで定義されているのに対し、割り込みは、命名と機能との両方において、常にベンダ（もっと言うとチップ）固有の実装です。

割り込みは、高度な使い方をしようとする時に必要とされる様々な柔軟性を考慮に入れています。本書では、そのような高度な使い方は対象外です。しかし、次の点に留意することをお勧めします。

- 割り込みは、ハンドラの実行順序を決めるプログラム可能な優先度を持ちます。
- 割り込みは、ネストとプリエンプションが可能です。つまり、割り込みハンドラの実行は、より優先度の高い割り込みに割り込まれる場合があります。
- 通常、割り込み要因は、割り込みハンドラが無限に再呼び出しされないようにするため、クリアされる必要があります。

ランタイムでの一般的な初期化手順は、常に同じです。

- 必要な時に割り込み要求を起こすように、ペリフェラルを設定します
- 割り込みコントローラで割り込みハンドラの優先度をセットします
- 割り込みコントローラで割り込みハンドラを有効化します

例外と同様に、例外ハンドラを宣言するために、`cortex-m-rt` クレートは、`interrupt` 属性を提供しています。 利用可能な割り込み（そして割り込みハンドラテーブルでの配置）は、通常、`svd2rust` を使って SVD から自動生成されます。

未翻訳

```
rust
use lm3s6965::interrupt; // 未翻訳

// タイマ2割り込みの割り込みハンドラ
#[interrupt]
fn TIMER2A() {
    // ..
    // 発生した割り込み要求の原因をクリアします
}
```

割り込みハンドラは、通常の関数のように見え、例外ハンドラに似ています（引数がないことを除いて）。 しかし、割り込みハンドラは、特別な呼び出し規約のため、ファームウェアの他の部分から直接呼び出すことができません。 ソフトウェアで割り込み要求を起こし、割り込みハンドラへの転送を発生させることは可能です。

例外ハンドラと同様に、割り込みハンドラ内で `static mut` 変数を宣言し、状態を安全に保持することができます。

```
#[interrupt]
fn TIMER2A() {
    static mut COUNT: u32 = 0;

    // `COUNT`は`&mut u32`の型を持っており、その使用は安全です
    *COUNT += 1;
}
```

ここで示した仕組みの詳細については、例外セクションを参照して下さい。

# 4. ペリフェラル

## 4.1. ペリフェラルとは何か？

ほとんどのマイクロコントローラはCPUやRAM、フラッシュメモリ以外のものを持っています。マイクロコントローラはシリコン上に専用のセクションを持っており、そのセクションは、マイクロコントローラの外のシステムとやり取りしたり、センサーやモーターコントローラ、またはディスプレイもしくはキーボードのようなヒューマンインタフェースによって世界中の周囲環境と直接的または間接的にやり取りするために使用されます。それらのコンポーネントはまとめてペリフェラルと呼ばれています。

これらのペリフェラルは有用です。なぜならば、開発者はペリフェラルに処理をオフロードすることが可能になるため、全ての処理をソフトウェアで行う必要がなくなります。デスクトップ開発者がグラフィック処理をビデオカードにオフロードするのと同じように、組込み開発者は一部のタスクをペリフェラルにオフロードして、CPUの時間を他の重要なことに使ったり、電力を節約するために何もしないようにすることができます。

1970年代か1980年代の旧式の家庭用コンピュータのメイン回路基板を見れば（実際に、旧式のデスクトップPCは今日の組込みシステムとさほど違いません）、次のものを目にするはずです。

- プロセッサ
- RAMチップ
- ROMチップ
- I/Oコントローラ

RAMチップ、ROMチップ、I/Oコントローラ（このシステムのペリフェラル）は「バス」として知られる一連の並列な配線を通してプロセッサに接続されているでしょう。アドレスバスは、プロセッサがバス上のどのデバイスと通信したいかを選択するアドレス情報を運び、データバスは、実際のデータを運びます。組込みマイクロコントローラにおいても、同じ原理が適用されます。それは全てが１つのシリコン片に詰め込まれているということです。

しかしながら、VulkanやMetal、OpenGLなどのソフトウェアのAPIを通常持つグラフィックカードとは異なり、ペリフェラルはメモリチャンクにマッピングされたハードウェアインターフェースとしてマイクロコントローラに公開されています。

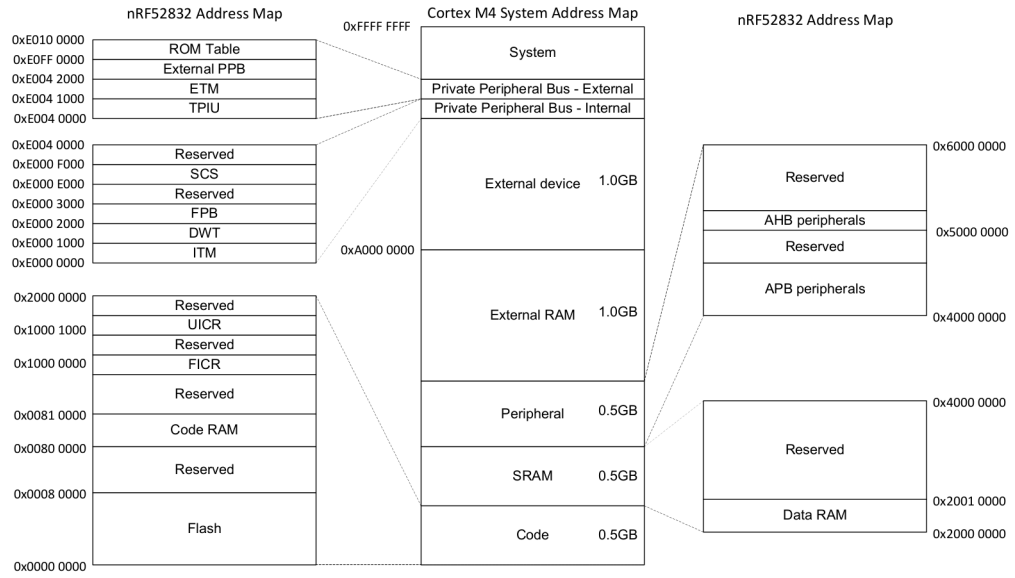
## 4.2. 線形な実メモリ空間

マイクロコントローラでは、`0x4000_0000`や`0x0000_0000`のような任意のアドレスにデータを書き込むことは、完全に正当な行為でしょう。

デスクトップシステムでは、メモリアクセスはMMU（メモリ管理ユニット）によって厳密に制御されています。このコンポーネントは２つの主な役割を持っています。メモリのセクションへのアクセス権限の強制（あるプロセスが別のプロセスのメモリを読み出したり変更したりできないようにする）、そして物理メモリのセグメントをソフトウェアで使用される仮想メモリ範囲に再マッピングすることです。マイクロコントローラは通常はMMUを持たず、代わりにソフトウェアで物理アドレスのみを使用します。

32ビットマイクロコントローラは`0x0000_0000`から`0xFFFF_FFFF`の線形な実アドレス空間を持ちますが、それらは大抵の場合、実際のメモリのためにはその範囲の数百キロバイトしか使用しません。これにより、かなりの量のアドレス空間が残ります。前の章では、RAMが`0x2000_0000`のアドレスに配置されていることについて話しました。もしもRAMが64KiBの長さなら（すなわち、最大アドレスが0xFFFF）、`0x2000_0000`から`0x2000_FFFF`がRAMのアドレ

スに対応します。0x2000_1234 のアドレスにある変数に書き込むと、内部ではアドレスの上位部分（この例では 0x2000）を検出し、アドレスの下位部分（この例では 0x1234）に作用できるように RAM をアクティブにします。Cortex-M においては、フラッシュ ROM も 0x0000_0000 から 0x0007_FFFF のアドレスにマッピングされています（512KiB のフラッシュ ROM が載っている場合）。これら２つの領域の間に残るスペースを全て無視するのではなく、代わりにマイクロコントローラの設計者は特定のメモリ配置にペリフェラルのインターフェースをマッピングしました。これは次のようなものになります。




Nordic nRF52832 Datasheet (pdf)

## 4.3. メモリマップド・ペリフェラル

一見すると、これらのペリフェラルとのやり取りは簡単です。正しいデータを正しいアドレスに書き込むだけです。例えば、32 ビットワードをシリアルポート上で送信することは、32 ビットワードを特定のメモリアドレスに書き込むことと同じくらい直接的になり得ます。シリアルポート・ペリフェラルは自動的にデータを引き受けて送信します。

これらのペリフェラルの設定についても同じように動作します。ペリフェラルの設定をするための関数を呼ぶ代わりに、ハードウェア API として機能するメモリチャンクが公開されます。0x8000_0000 を SPI 周波数の設定レジスタに書き込むと、SPI ポートは 8Mbps でデータを送信するようになります。0x0200_0000 を同じアドレスに書き込むと、SPI は 125Kbps でデータを送信するようになります。これらの設定レジスタをちょっとだけ見てみます。

### 31.6.8 FREQUENCY

Address offset: 0x524

SPI frequency

| Bit number | | | | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Id | | | | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| **Reset 0x04000000** | | | | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

| Id | RW | Field | Value Id | Value | Description |
|---|---|---|---|---|---|
| A | RW | FREQUENCY | | | SPI master data rate |
| | | | K125 | 0x02000000 | 125 kbps |
| | | | K250 | 0x04000000 | 250 kbps |
| | | | K500 | 0x08000000 | 500 kbps |
| | | | M1 | 0x10000000 | 1 Mbps |
| | | | M2 | 0x20000000 | 2 Mbps |
| | | | M4 | 0x40000000 | 4 Mbps |
| | | | M8 | 0x80000000 | 8 Mbps |

Nordic nRF52832 Datasheet (pdf)

アセンブリ言語や C 言語、Rust など、どの言語が使われようとも、このインターフェースがどのように作用するかはハードウェアによって定められています。

## 4.4. 最初の試み

### 4.4.1. レジスタ

SysTick ペリフェラルを見ていきましょう。SysTick は Cortex-M プロセッサ・コアに搭載されているシンプルなタイマーです。通常は、チップメーカーのデータシートやテクニカルリファレンスマニュアルでこれらのペリフェラルの情報を調べることができるのですが、この例においては全ての ARM Cortex-M コアで共通のものですので、今回は ARMリファレンスマニュアルを見てみましょう。 そこには 4 つのレジスタが載っています。

| オフセット | 名前 | 説明 | 幅 |
|---|---|---|---|
| 0x00 | SYST_CSR | 制御およびステータスレジスタ | 32 ビット |
| 0x04 | SYST_RVR | リロード値レジスタ | 32 ビット |
| 0x08 | SYST_CVR | 現在値レジスタ | 32 ビット |
| 0x0C | SYST_CALIB | キャリブレーション値レジスタ | 32 ビット |

### 4.4.2. C アプローチ

Rust でも、struct を使って C と同じ方法でレジスタの集合を正確に表現することができます。

```
#[repr(C)]
struct SysTick {
    pub csr: u32,
    pub rvr: u32,
    pub cvr: u32,
    pub calib: u32,
}
```

#[repr(C)]修飾子は Rust コンパイラ対して、この構造体を C コンパイラと同じようにメモリにレイアウトするように指示します。 これはとても重要なことです。なぜなら Rust では、C においては行われない構造体のフィールドの並び替えが許されているためです。 コンパイラによって暗黙のうちに構造体のフィールドが並び替えられることにより、デバッグをするはめになることは想像できるでしょう！ この修飾子を置くことで、4 つの 32 ビットの各フィールドは上記のテーブルに対応付けられます。 ただしもちろん、この struct はそれ自体では何の役にも立ちません。次のように変数として使う必要があります。

```
let systick = 0xE000_E010 as *mut SysTick;
let time = unsafe { (*systick).cvr };
```

#### 4.4.3. volatile アクセス

上記のやり方には、いくつか問題があります。

1. ペリフェラルにアクセスするためには毎回アンセーフを使わなくてはなりません。
2. どのレジスタが読み取り専用でどのレジスタが読み書き可能かを指定する方法がありません。
3. プログラム内のどのコードからでもこの構造体を通してハードウェアにアクセスできてしまいます。
4. 最も大事なことは、このコードは実際には動作しないということです...

ここで問題となるのは、コンパイラが賢いということです。同じ RAM に相次いで２回書き込むと、コンパイラはこれに気づき、最初の書き込みを完全にスキップします。C では、全ての読み書きが意図した通りに行われることを保証するために、`volatile` 型修飾子を変数につけることができます。 Rust では、変数ではなくアクセスに対して volatile をつけます。

```
let systick = unsafe { &mut *(0xE000_E010 as *mut SysTick) };
let time = unsafe { core::ptr::read_volatile(&mut systick.cvr) };
```

これで 4 つの問題のうち 1 つを直せました。しかし、さらに `unsafe` なコードがあります！ 幸いなことに、これに対処できるサードパーティ製のクレート `volatile_register` があります。

```
use volatile_register::{RW, RO};

#[repr(C)]
struct SysTick {
    pub csr: RW<u32>,
    pub rvr: RW<u32>,
    pub cvr: RW<u32>,
    pub calib: RO<u32>,
}

fn get_systick() -> &'static mut SysTick {
    unsafe { &mut *(0xE000_E010 as *mut SysTick) }
}

fn get_time() -> u32 {
    let systick = get_systick();
    systick.cvr.read()
}
```

これで `read` と `write` メソッドを通して volatile アクセスが自動的に行われるようになりました。書き込みを実行するのはまだ `unsafe` です。しかし、公平を期するために言うと、ハードウェアは変更可能な状態の集まりであるため、それらへの書き込みが実際に安全なのかどうか、をコンパイラが知る方法はないのです。そのため、これは基本姿勢としては悪くないでしょう。

#### 4.4.4. Rust のラッパ

ユーザが安全に呼び出せるように、この `struct` を高レイヤーの API でラップする必要があります。ドライバの作者として、アンセーフなコードが正しいことを手動で検証し、ユーザがそのドライバを使用する上で心配することがないように安全な API として提供します。（ユーザは提供されたものが正しいと信頼しています ‽）

一例を挙げます。

```
use volatile_register::{RW, RO};

pub struct SystemTimer {
    p: &'static mut RegisterBlock
}

#[repr(C)]
struct RegisterBlock {
    pub csr: RW<u32>,
    pub rvr: RW<u32>,
    pub cvr: RW<u32>,
    pub calib: RO<u32>,
}

impl SystemTimer {
    pub fn new() -> SystemTimer {
        SystemTimer {
            p: unsafe { &mut *(0xE000_E010 as *mut RegisterBlock) }
        }
    }

    pub fn get_time(&self) -> u32 {
        self.p.cvr.read()
    }

    pub fn set_reload(&mut self, reload_value: u32) {
        unsafe { self.p.rvr.write(reload_value) }
    }
}

pub fn example_usage() -> String {
    let mut st = SystemTimer::new();
    st.set_reload(0x00FF_FFFF);
    format!("Time is now 0x{:08x}", st.get_time())
}
```

このやり方の問題は、次のコードがコンパイラに完全に受け入れられることです。

```
fn thread1() {
    let mut st = SystemTimer::new();
    st.set_reload(2000);
}

fn thread2() {
    let mut st = SystemTimer::new();
    st.set_reload(1000);
}
```

`set_reload` 関数に `&mut self` 引数を渡すことで、そのインスタンスの `SystemTimer` 構造体に対する他の参照がないことを確認しますが、全く同じペリフェラルを指す２つ目の `SystemTimer` 構造体をユーザが作ることは止められません！このように書かれたコードは、作者がこれらの「重複した」ドライバのインスタンスを全て見つけるのに十分に熱心であれば動作するでしょうが、一度コードが複数のモジュール、ドライバ、開発者、そして日に渡って分散すると、この種の間違いがどんどん入り込みやすくなっていきます。

## 4.5. ミュータブルでグローバルな状態

残念ながら、ハードウェアは基本的にミュータブルでグローバルな状態に他なりません。これは Rust の開発者にとって非常に恐ろしいことです。 ハードウェアは書かれたコードの構造とは独立して存在しており、現実の世界からいつでも変更される可能性があります。

### 4.5.1. 何をルールとするべきか？

どうすればこれらのペリフェラルと確実にやり取りできるのでしょう？

1. いつ変化するかわからないペリフェラルメモリの読み書きには、常に `volatile` メソッドを使用してください
2. ソフトウェアでは、これらのペリフェラルへの読み取り専用アクセスをいくつでも共有できるでしょう
3. あるソフトウェアがあるペリフェラルに読み書きのアクセスをするならば、そのソフトウェアは、ペリフェラルへの唯一の参照を持つべきです

### 4.5.2. 借用チェッカ

さきほどのルールの最後の２つは、借用チェッカが行っていることに怪しいくらいよく似ています。

ペリフェラルの所有権を譲渡したり、ペリフェラルへのイミュータブルまたはミュータブルな参照を提供したりできるのか、を想像してみてください。

それは可能です。ただし、借用チェッカが正しくペリフェラルの所有権や参照を扱うためには、各ペリフェラルが持つインスタンスはただ１つにする必要があります。そうすれば、Rust はこれを正しく扱えます。幸いなことにハードウェアにおいて、任意のペリフェラルのインスタンスは１つだけしかありません。しかし、どうすればそれをコードの構造として明確にできるでしょうか？

## 4.6. シングルトン

> ソフトウェア工学において、シングルトン・パターンはクラスのインスタンス化を１つのオブジェクトに制限するデザインパターンです。
>
> *Wikipedia: Singleton Pattern*

### 4.6.1. なぜグローバル変数は使えないのか？

このように、全てをパブリックかつスタティックにすることができます。

```
static mut THE_SERIAL_PORT: SerialPort = SerialPort;

fn main() {
    let _ = unsafe {
        THE_SERIAL_PORT.read_speed();
    };
}
```

しかし、これにはいくつか問題があります。 これはミュータブルなグローバル変数であり、Rust においては、これらとやり取りするのは常にアンセーフです。これらの変数はプログラムの全体を通して見えることになり、つまりそれは借用チェッカがこれらの変数の参照や所有権を追跡するのに役立たなくなることを意味します。

### 4.6.2. Rust ではどうするか？

単にペリフェラルをグローバル変数にする代わりに、各ペリフェラル毎に `Option<T>`を含む `PERIPHERALS` と呼ばれるグローバル変数を作ることにします。

```
struct Peripherals {
    serial: Option<SerialPort>,
}
impl Peripherals {
    fn take_serial(&mut self) -> SerialPort {
        let p = replace(&mut self.serial, None);
        p.unwrap()
    }
}
static mut PERIPHERALS: Peripherals = Peripherals {
    serial: Some(SerialPort),
};
```

この構造体によって、ペリフェラルの唯一のインスタンスが取得できるようになります。もしも `take_serial()`を複数回呼び出そうとすれば、コードはパニックするでしょう。

```
fn main() {
    let serial_1 = unsafe { PERIPHERALS.take_serial() };
    // これはパニックします！
    // let serial_2 = unsafe { PERIPHERALS.take_serial() };
}
```

この構造体とのやり取りは `unsafe` にはなりますが、一度この構造体に含まれる `SerialPort` を取得してしまえばもう `unsafe` や `PERIPHERALS` 構造体を使う必要は全くありません。

この方法では、オプション型の中に `SerialPort` 構造体をラップし、`take_serial()`を一度コールする必要があるため、実行時に小さなオーバーヘッドとなります。しかし、この少々のコストを前払いすることで、残りのプログラムで借用チェッカを利用できるようになるのです。

### 4.6.3. 既存のライブラリによるサポート

上記のコードでは `Peripherals` 構造体を作りましたが、あなたのコードでも同じようにする必要はありません。 `cortex_m` クレートはこれと同様のことをしてくれる `singleton!()`と呼ばれるマクロを含んでいます。

```
use cortex_m::singleton;

fn main() {
    // `main`が一度だけしか実行されなければOKです
    let x: &'static mut bool =
        singleton!(: bool = false).unwrap();
}
```

未翻訳

未翻訳

```
// cortex-m-rtic v0.5.x
#[rtic::app(device = lm3s6965, peripherals = true)]
const APP: () = {
    #[init]
    fn init(cx: init::Context) {
        static mut X: u32 = 0;
```

```
        // 未翻訳
        let core: cortex_m::Peripherals = cx.core;

        // 未翻訳
        let device: lm3s6965::Peripherals = cx.device;
    }
}
```

#### 4.6.4. しかしなぜ？

しかし、これらのシングルトンが Rust のコードの動作にどのような顕著な違いをもたらすのでしょうか？

```
impl SerialPort {
    const SER_PORT_SPEED_REG: *mut u32 = 0x4000_1000 as _;

    fn read_speed(
        &self // <------ これは本当に、本当に重要です。
    ) -> u32 {
        unsafe {
            ptr::read_volatile(Self::SER_PORT_SPEED_REG)
        }
    }
}
```

ここには２つの重要な要素があります。

- シングルトンを使用しているため、`SerialPort` 構造体の取得手段や場所は１つだけになります。
- `read_speed()`メソッドを呼ぶためには、`SerialPort` 構造体の所有権もしくは参照を持つ必要があります。

これらの２つの要素をまとめると、借用チェッカを適切に満たしている場合のみハードウェアにアクセスできることを意味します。つまり、同じハードウェアに対して複数のミュータブルな参照を持つことはありません。

```
fn main() {
    // `self`への参照が見つかりません。これはうまく動きません。
    // SerialPort::read_speed();

    let serial_1 = unsafe { PERIPHERALS.take_serial() };

    // アクセスできるものだけ読み出すことができます。
    let _ = serial_1.read_speed();
}
```

#### 4.6.5. ハードウェアをデータのように扱う

加えて、いくつかの参照はミュータブルで、またいくつかはイミュータブルなため、関数またはメソッドがハードウェアの状態を変更できるかどうかを確認することが可能になります。

この関数はハードウェアの設定を変更できます。

```
fn setup_spi_port(
    spi: &mut SpiPort,
    cs_pin: &mut GpioPin
) -> Result<()> {
    // ...
}
```

このメソッドは変更できません。

```
fn read_button(gpio: &GpioPin) -> bool {
    // ...
}
```

これにより、実行時ではなく**コンパイル時に**コードがハードウェアを変更するかどうかを強制できます。注意点としては、通常この強制はひとつのアプリケーション内でのみ機能します。ベアメタルシステムにおいては、ソフトウェアはひとつのアプリケーションにコンパイルされるため、これは一般的には制約にはなりません。

## 5. 静的な保証

コンパイル時にデータ競合を防ぐのは、Rust の型システムです（Send と Sync トレイトを参照）。この型システムは、コンパイル時に他のプロパティをチェックするためにも使用できます。その結果、実行時チェックの必要性を減らせる場合があります。

組込みプログラムに適用する場合、これらの静的なチェックは、例えば、入出力インタフェースが正しく設定されていることを強制することができます。 例えば、使用されるピンを最初に設定することによってのみ、シリアルインタフェースを初期化できるような API 設計が可能です。

正しく設定されたペリフェラルでのみ、ピンをローレベルにするというような操作ができることを、静的にチェックすることも可能です。例えば、フローティング入力モードに設定されたピンの出力状態を変更しようとすると、 コンパイルエラーが発生します。

以前の章で見た通り、所有権の概念はペリフェラルにも適用できます。所有権は、プログラムの特定部分のみがペリフェラルを変更することを保証します。このアクセスコントロールは、ペリフェラルをグローバルでミュータブルな状態として扱う代替案と比較して、 ソフトウェアの解析をより簡単にします。

### 5.1. 型状態プログラミング

型状態の概念は、オブジェクトの現在の状態に関する情報を、そのオブジェクトの型にエンコードすることを説明しています。これは、少し難解に思えますが、Rust のビルダーパターンを使ったことがあるならば、既に型状態プログラミングを使い始めています。

```
pub mod foo_module {
    #[derive(Debug)]
    pub struct Foo {
        inner: u32,
    }

    pub struct FooBuilder {
        a: u32,
        b: u32,
    }

    impl FooBuilder {
        pub fn new(starter: u32) -> Self {
            Self {
                a: starter,
                b: starter,
            }
        }

        pub fn double_a(self) -> Self {
            Self {
                a: self.a * 2,
                b: self.b,
            }
        }

        pub fn into_foo(self) -> Foo {
            Foo {
                inner: self.a + self.b,
            }
```

```
        }
    }
}

fn main() {
    let x = foo_module::FooBuilder::new(10)
        .double_a()
        .into_foo();

    println!("{:#?}", x);
}
```

この例では、Foo オブジェクトを直接作る方法はありません。必要な Foo オブジェクトを取得する前に、FooBuilder を作り、正しく初期化しなければなりません。

この最小限の例は、2 つの状態をエンコードしています。
- FooBuilder は、「未設定」もしくは「設定中」の状態を表現します。
- Foo は、「設定済み」もしくは「使用準備完了」の状態を意味します。

#### 5.1.1. 強い型

Rust は強い型付けシステムを持っています。Foo のインスタンスを魔法のように作成したり、into_foo()メソッドを呼び出すことなしに FooBuilder から Foo に変換したりする、簡単な方法はありません。 さらに、into_foo()メソッドは、オリジナルの FooBuilder 構造体を消費します。 これは、新しいインスタンスを作成しないと、再利用ができないことを意味します。

このことにより、システムの状態を型として表現することが可能になります。そして、状態遷移に必要なアクションを、ある型と別の型とを交換するメソッドに取り入れることができます。FooBuilder を作成し、Foo オブジェクトに交換することで、基本的なステートマシンのステップを見てきました。

### 5.2. ステートマシンとしてのペリフェラル

マイクロコントローラのペリフェラルは、一連のステートマシンとして考えることができます。 例えば、簡略化された GPIO ピンの設定は、次の状態ツリーとして表すことができます。
- 無効
- 有効
  - 出力として設定
    - 出力：ハイ
    - 出力：ロー
  - 入力として設定
    - 入力：高抵抗（訳注：ハイインピーダンス）
    - 入力：プルダウン
    - 入力：プルアップ

このペリフェラルが無効モードから始まるとすると、入力：高抵抗モードに移行するには、次のステップを踏みます。
1. 無効
2. 有効
3. 入力として設定
4. 入力：高抵抗

入力：高抵抗から入力：プルダウンへと移る場合、次のステップを実行しなければなりません。
1. 入力：高抵抗

2. 入力：プルダウン

同じように、入力：プルダウンと設定されている GPIO ピンを、出力：ハイにするには、次のステップを実行しなければなりません。
1. 入力：プルダウン
2. 入力として設定
3. 出力として設定
4. 出力：ハイ

#### 5.2.1. ハードウェアの表現

通常、上記の状態は、GPIO ペリフェラルに割り当てられたレジスタに値を書き込むことで設定できます。 これを説明するために、架空の GPIO 設定レジスタを定義しましょう。

| 名前 | ビットフィールド | 値 | 意味 | 説明 |
|---|---|---|---|---|
| enable | 0 | 0 | disabled | GPIO を無効にする |
| | | 1 | enabled | GPIO を有効にする |
| direction | 1 | 0 | input | 方向を入力に設定する |
| | | 1 | output | 方向を出力に設定する |
| input_mode | 2..3 | 00 | hi-z | 入力を高抵抗に設定する |
| | | 01 | pull-low | 入力ピンはプルダウンになる |
| | | 10 | pull-high | 入力ピンはプルアップになる |
| | | 11 | n/a | 無効な状態。設定しないこと。 |
| output_mode | 4 | 0 | set-low | 出力ピンをローにする |
| | | 1 | set-high | 出力ピンをハイにする |
| input_status | 5 | x | in-val | 入力が 1.5V より低ければ 0、1.5V 以上であれば 1 |

この GPIO を制御するために、次のような Rust の構造体を公開することが できます。

```
/// GPIOインタフェース
struct GpioConfig {
    /// svd2rustで生成されたGPIO設定構造体
    periph: GPIO_CONFIG,
}

impl GpioConfig {
    pub fn set_enable(&mut self, is_enabled: bool) {
        self.periph.modify(|_r, w| {
            w.enable().set_bit(is_enabled)
        });
    }

    pub fn set_direction(&mut self, is_output: bool) {
        self.periph.modify(|_r, w| {
            w.direction().set_bit(is_output)
        });
    }

    pub fn set_input_mode(&mut self, variant: InputMode) {
        self.periph.modify(|_r, w| {
```

```
            w.input_mode().variant(variant)
        });
    }

    pub fn set_output_mode(&mut self, is_high: bool) {
        self.periph.modify(|_r, w| {
            w.output_mode.set_bit(is_high)
        });
    }

    pub fn get_input_status(&self) -> bool {
        self.periph.read().input_status().bit_is_set()
    }
}
```

しかし、この実装では、筋が通らないレジスタの修正が可能になってしまいます。例えば、GPIOを入力に設定している時に、出力モードを設定すると、どうなるのでしょう？

通常、この構造体を利用すると、上のステートマシンで定義されていない状態に到達できてしまいます。 例えば、プルダウンの出力や、ハイに設定された入力、です。

このインタフェースは書き込みには便利ですが、ハードウェア実装によって定められた設計の契約を強制しません。

### 5.3. 設計契約

前回、設計契約を強制しないインタフェースを書きました。架空の GPIO 設定レジスタをもう一度見てみましょう。

| 名前 | ビットフィールド | 値 | 意味 | 説明 |
|---|---|---|---|---|
| enable | 0 | 0 | disabled | GPIO を無効にする |
| | | 1 | enabled | GPIO を有効にする |
| direction | 1 | 0 | input | 方向を入力に設定する |
| | | 1 | output | 方向を出力に設定する |
| input_mode | 2..3 | 00 | hi-z | 入力を高抵抗に設定する |
| | | 01 | pull-low | 入力ピンはプルダウンになる |
| | | 10 | pull-high | 入力ピンはプルアップになる |
| | | 11 | n/a | 無効な状態。設定しないこと。 |
| output_mode | 4 | 0 | set-low | 出力ピンをローにする |
| | | 1 | set-high | 出力ピンをハイにする |
| input_status | 5 | x | in-val | 入力が 1.5V より低ければ 0、1.5V 以上であれば 1 |

ハードウェアを使う前に状態をチェックし、実行時に設計契約を強制すると、コードは次のように書くことができます。

```
/// GPIOインタフェース
struct GpioConfig {
    /// svd2rustによって生成されたGPIO設定構造体
    periph: GPIO_CONFIG,
}
```

```
impl GpioConfig {
    pub fn set_enable(&mut self, is_enabled: bool) {
        self.periph.modify(|_r, w| {
            w.enable().set_bit(is_enabled)
        });
    }

    pub fn set_direction(&mut self, is_output: bool) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // 方向を設定するには、有効化されてなければなりません
            return Err(());
        }

        self.periph.modify(|r, w| {
            w.direction().set_bit(is_output)
        });

        Ok(())
    }

    pub fn set_input_mode(&mut self, variant: InputMode) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // 入力モードを設定するには、有効化されてなければなりません
            return Err(());
        }

        if self.periph.read().direction().bit_is_set() {
            // 方向は入力でなければなりません
            return Err(());
        }

        self.periph.modify(|_r, w| {
            w.input_mode().variant(variant)
        });

        Ok(())
    }

    pub fn set_output_status(&mut self, is_high: bool) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // 方向は入力でなければなりません
            return Err(());
        }

        if self.periph.read().direction().bit_is_clear() {
            // 方向は出力でなければなりません
            return Err(());
        }

        self.periph.modify(|_r, w| {
            w.output_mode.set_bit(is_high)
        });

        Ok(())
    }
```

```
    pub fn get_input_status(&self) -> Result<bool, ()> {
        if self.periph.read().enable().bit_is_clear() {
            // 状態を取得するには、有効化されてなければなりません
            return Err(());
        }

        if self.periph.read().direction().bit_is_set() {
            // 方向は入力でなければなりません
            return Err(());
        }

        Ok(self.periph.read().input_status().bit_is_set())
    }
}
```

ハードウェアの制約を強制する必要があるため、時間とリソースを浪費する多くの実行時チェックを行うこととなり、開発者にとってこのコードは好ましくないです。

#### 5.3.1. 型状態

しかし、代わりに、状態遷移の規則を強制するために、Rust の型システムを使うとどうなるでしょうか？この例を見て下さい。

```
/// GPIOインタフェース
struct GpioConfig<ENABLED, DIRECTION, MODE> {
    /// svd2rustによって生成されたGPIO設定構造体
    periph: GPIO_CONFIG,
    enabled: ENABLED,
    direction: DIRECTION,
    mode: MODE,
}

// GpioConfigのMODEのための型状態
struct Disabled;
struct Enabled;
struct Output;
struct Input;
struct PulledLow;
struct PulledHigh;
struct HighZ;
struct DontCare;

/// これらの関数はどのGPIOピンにも使えます
impl<EN, DIR, IN_MODE> GpioConfig<EN, DIR, IN_MODE> {
    pub fn into_disabled(self) -> GpioConfig<Disabled, DontCare, DontCare> {
        self.periph.modify(|_r, w| w.enable.disabled());
        GpioConfig {
            periph: self.periph,
            enabled: Disabled,
            direction: DontCare,
            mode: DontCare,
        }
    }

    pub fn into_enabled_input(self) -> GpioConfig<Enabled, Input, HighZ> {
        self.periph.modify(|_r, w| {
            w.enable.enabled()
```

```
                .direction.input()
                .input_mode.high_z()
        });
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: HighZ,
        }
    }

    pub fn into_enabled_output(self) -> GpioConfig<Enabled, Output, DontCare> {
        self.periph.modify(|_r, w| {
            w.enable.enabled()
             .direction.output()
             .input_mode.set_high()
        });
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Output,
            mode: DontCare,
        }
    }
}

/// この関数はOutputピンに使用できます
impl GpioConfig<Enabled, Output, DontCare> {
    pub fn set_bit(&mut self, set_high: bool) {
        self.periph.modify(|_r, w| w.output_mode.set_bit(set_high));
    }
}

/// これらのメソッドは、有効化された入力GPIOに使えます
impl<IN_MODE> GpioConfig<Enabled, Input, IN_MODE> {
    pub fn bit_is_set(&self) -> bool {
        self.periph.read().input_status.bit_is_set()
    }

    pub fn into_input_high_z(self) -> GpioConfig<Enabled, Input, HighZ> {
        self.periph.modify(|_r, w| w.input_mode().high_z());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: HighZ,
        }
    }

    pub fn into_input_pull_down(self) -> GpioConfig<Enabled, Input, PulledLow> {
        self.periph.modify(|_r, w| w.input_mode().pull_low());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: PulledLow,
```

```
        }
    }

    pub fn into_input_pull_up(self) -> GpioConfig<Enabled, Input, PulledHigh> {
        self.periph.modify(|_r, w| w.input_mode().pull_high());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: PulledHigh,
        }
    }
}
```

それでは、これを使うコードがどのようになるか、見てみましょう。

```
/*
 * 例1：未設定から高抵抗入力
 */
let pin: GpioConfig<Disabled, _, _> = get_gpio();

// これはできません、ピンが有効になっていません！
// pin.into_input_pull_down();

// 今度は、未設定から高抵抗入力に変えます
let input_pin = pin.into_enabled_input();

// ピンから値を読みます
let pin_state = input_pin.bit_is_set();

// これはできません、入力ピンはこのインタフェースを持っていません！
// input_pin.set_bit(true);

/*
 * 例2：高抵抗入力からプルダウン入力
 */
let pulled_low = input_pin.into_input_pull_down();
let pin_state = pulled_low.bit_is_set();

/*
 * 例3：プルダウン入力から出力、ハイを設定
 */
let output_pin = pulled_low.into_enabled_output();
output_pin.set_bit(true);

// これはできません、出力ピンはこのインタフェースを持っていません！
// output_pin.into_input_pull_down();
```

これは間違いなく、ピンの状態を保存するのに便利な方法ですが、なぜこのようにするのでしょうか？なぜ、`GpioConfig` 構造体の中で、状態を `enum` として保存するより良い方法なのでしょうか？

#### 5.3.2. コンパイル時の機能の安全性

コンパイル時に、設計契約を完全に強制しているため、実行時コストはかかりません。入力方向のピンに対して、出力モードを設定することは不可能です。代わりに、そのピンを出力ピン

に変換してから、出力モードを設定することで、状態を辿る必要があります。 このおかげで、関数実行前に現在の状態をチェックすることによる実行時ペナルティは、ありません。

型システムによってこれらの状態が強制されるため、このインタフェースの利用者によるエラーの余地はもはや残っていません。もし利用者が不正な状態遷移をしようとすると、そのコードはコンパイルできません！

## 5.4. ゼロコスト抽象化

型状態はゼロコスト抽象化の優れた例でもあります。特定の動作を、コンパイル時の実行もしくは解析に移動する機能です。これらの型状態は、実際のデータを含んでおらず、代わりにマーカとして使われています。型状態はデータを含んでいないため、実行時、メモリ内に実際のデータはありません。

```
use core::mem::size_of;

let _ = size_of::<Enabled>();    // == 0
let _ = size_of::<Input>();      // == 0
let _ = size_of::<PulledHigh>(); // == 0
let _ = size_of::<GpioConfig<Enabled, Input, PulledHigh>>(); // == 0
```

### 5.4.1. ゼロサイズの型

```
struct Enabled;
```

上記のように定義された構造体をゼロサイズの型、と呼びます。これは、実際のデータを含んでいません。これらの型は、コンパイル時には「実際に」機能します。例えば、コピーも、ムーブも、参照を取ることもできます。 しかし、最適化はこれらを完全に取り除きます。

次のコードスニペットを見てください。

```
pub fn into_input_high_z(self) -> GpioConfig<Enabled, Input, HighZ> {
    self.periph.modify(|_r, w| w.input_mode().high_z());
    GpioConfig {
        periph: self.periph,
        enabled: Enabled,
        direction: Input,
        mode: HighZ,
    }
}
```

実行時、返り値の GpioConfig は、存在しません。この関数を呼び出すと、通常、1 つのアセンブリ命令にまとめられます。そのアセンブリ命令は、定数のレジスタ値を、レジスタの位置へ格納します。これは、開発した型状態インタフェースが、ゼロコスト抽象化であることを意味します。GpioConfig の状態を追跡するために、余分な CPU、RAM、コード領域を使用せず、直接レジスタアクセスするのと同じ機械語を表します。

### 5.4.2. ネスト

通常、これらの抽象化は、望み通りの深さでネストされます。使用される全てのコンポーネントが、ゼロサイズの型である限り、実行時には、構造体全体が存在しません。

複雑な構造体や深くネストした構造体については、全ての取り得る状態の組み合わせを定義することは、面倒です。このような場合、全ての実装を生成するために、マクロが利用できます。

# 6. 移植性

組込み環境においては、移植性は非常に重要なトピックです。1 つのメーカから、それぞれのベンダや各プロダクトファミリが、異なるペリフェラルや機能を提供します、異なるペリフェラルでは、ペリフェラルとやり取りする方法も異なります。

このような違いを吸収する一般的な方法は、ハードウェア抽象化レイヤまたは **HAL** と呼ばれるレイヤを導入することです。

> ハードウェア抽象化は、プラットフォーム固有の細部をエミュレーションして、プログラムにハードウェアリソースへ直接アクセスする方法を提供する、ソフトウェアの一連のルーチンです。
>
> それらのルーチンがハードウェアへのオペレーティングシステム (OS) コールを提供することで、プログラマは、デイバスに依存せず、高性能なアプリケーションを書くことができます。
>
> *Wikipedia:* ハードウェア抽象化レイヤ

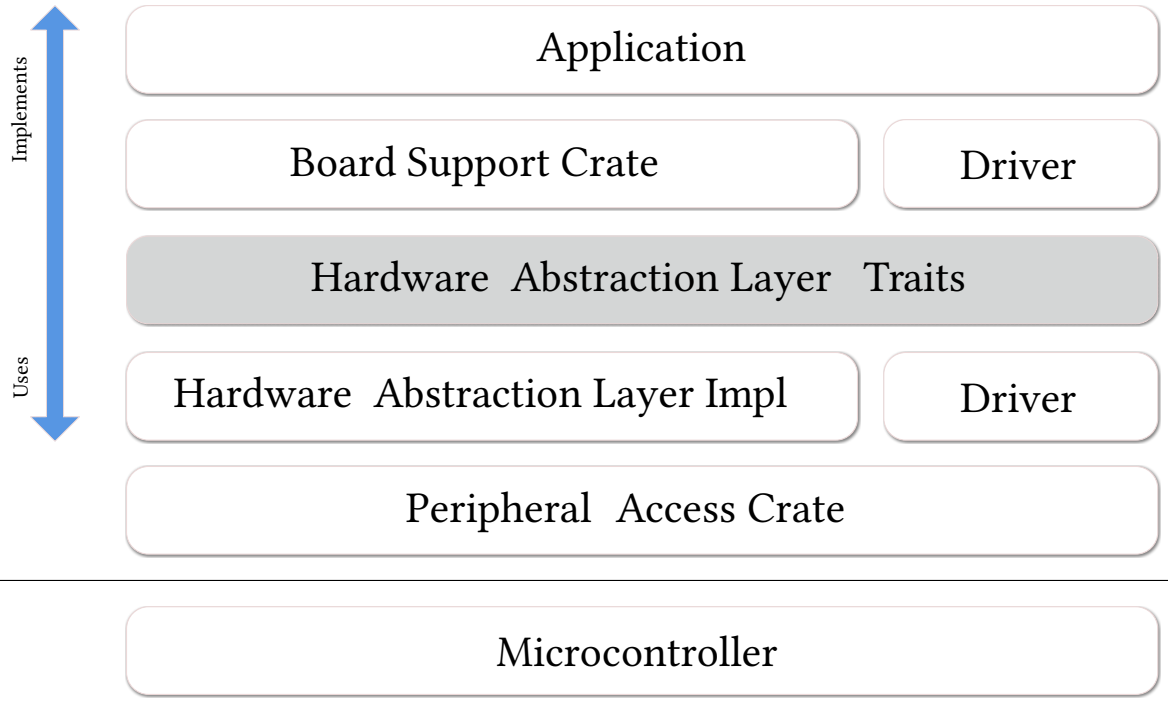
組込みシステムは、通常オペレーティングシステムを使わず、ユーザがインストールできるソフトウェアもありません。 全体として 1 つにコンパイルされたファームウェアイメージであり、様々な制約を持つ、という点が特殊です。 そのため、Wikipedia で定義されている従来のアプローチでもうまく機能する可能性はありますが、移植性を確保するための最も有効なアプローチではない可能性があります。

Rust ではどうするのでしょうか？**embedded-hal** を見ていきましょう。

### 6.0.1. embedded-hal とは？

一言で言えば、**HAL 実装**、**ドライバ**、**アプリケーションまたはファームウェア**間の実装規約を定義するトレイトの集まりのことです。それらの規約は、機能（ある型にトレイトが実装されていれば、**HAL 実装**はそのトレイトの機能を提供します）とメソッド（あるトレイトを実装している型のオブジェクトを作成できれば、そのトレイトに含まれるメソッドが利用可能であることが保証されます）を含んでいます。

典型的なレイヤ構造は、次のようになります。




**embedded-hal** で定義されているいくつかのトレイトを示します。

- GPIO（入出力ピン）
- シリアル通信
- I2C
- SPI
- タイマ/カウントダウン
- アナログデジタル変換

**embedded-hal** トレイトと、それらを実装して使用するクレートを持つ主な理由は、複雑さを抑えることです。アプリケーションがハードウェアのペリフェラルを使うコードだけでなく、追加するハードウェアコンポーネントのドライバを実装しなければならない場合を考えると、再利用性は非常に制限されます。数学的に表現すると、**M** がペリフェラル HAL 実装の数で、**N** がドライバの数とするならば、全てのアプリケーションで車輪の再発明を行うことになります。その結果、**M*N** の実装が必要になります。一方、**embedded-hal** トレイトで提供される API を使うことで、実装の複雑さは **M+N** になるでしょう。もちろん、明確に定義されたすぐに利用可能な API によって試行錯誤を減らすなど、他にも利点があります。

#### 6.0.2. embedded-hal のユーザ

上記のように、HAL には主に 3 つのユーザがいます。

##### 6.0.2.1. HAL 実装

HAL 実装は、ハードウェアと HAL トレイトのユーザとの間のインタフェースを実装します。典型的な実装では、3 つの部分から構成されます。

- 1 つ以上のハードウェア固有の型
- そのような型のオブジェクトを作成し、初期化する関数。多くの場合、様々な設定オプションを提供しています（速度、動作モード、使用ピンなど）
- 1 つ以上のその型に対する **embedded-hal** の `trait impl`

このような **HAL 実装**は、様々な種類があります。

- 低レベルのハードウェアアクセスによるもの、例えばレジスタ

- オペレーティングシステムによるもの、例えば Linux の `sysfs` の使用
- アダプタによるもの、例えばユニットテストのための型のモック
- ハードウェアアダプタ用のドライバによるもの、例えば I2C マルチプレクサや GPIO エキスパンダ

#### 6.0.2.2. ドライバ

ドライバは、embedded-hal トレイトを実装しているペリフェラルに接続されている、内部または外部コンポーネントに対するカスタム機能を実装します。 そのようなドライバの典型的な例は、様々なセンサ（温度、磁気、加速度、輝度）、ディスプレイデバイス（LED アレイ、LCD ディスプレイ）や、アクチュエータ（モータ、トランスミッタ）を含みます。

ドライバは、embedded-hal の特定のトレイトを実装する型のインスタンスと共に初期化する必要があります。そのトレイトは、トレイト境界により保証され、駆動するデバイスとやり取りできるインスタンスをカスタムメソッドと共に提供します。

#### 6.0.2.3. アプリケーション

アプリケーションは、様々な部品を結合し、必要な機能が確実に実現できるようにします。別システムに移植する際、アプリケーションは最も適合作業が求められる部分です。アプリケーションは、HAL 実装を通じて、実際のハードウェアを初期化する必要があるからです。異なるハードウェアの初期化は、極端に異なる場合があります。ユーザの選択は、多くの場合、大きな影響があります。コンポーネントが別の端子に物理的に接続されたり、ハードウェアバスが機器構成を満たすために外部ハードウェアを必要とする場合があったり、内部ペリフェラルを使えるようにするための異なるトレードオフがあったりします。例えば、異なる機能を持つ複数のタイマが利用可能であること、や、ペリフェラルが他のペリフェラルと競合すること、です。

# 7. 並行性

プログラムの異なる部分が様々なタイミングで実行されたり、アウトオブオーダに実行されると、並行性が発生します。 組込みでは、次のものが該当します。

- 割り込みが発生するたびに実行される割り込みハンドラ
- マイクロプロセッサがプログラムの一部を定期的にスワップする様々な形式のマルチスレッド
- システムによっては、各コアがプログラムの異なる部分を同時に独立して実行できるマルチコアマイクロプロセッサ

多くの組込みプログラムは割り込みを処理する必要があるため、早かれ遅かれ、並行性は発生します。 割り込みは、捉えにくく、難しいバグが数多く発生し得る場所でもあります。 幸運なことに、Rust は正しいコードを書く助けになる抽象化と安全性保証とを、いくつか提供しています。

### 7.0.1. 並行性なし

組込みプログラムの最も簡単な並行性は、並行性がないことです。ソフトウェアは 1 つの動作し続けるメインループからなり、割り込みも発生しません。時には、これが手元の問題の最適解かもしれません。通常、ループは何か入力を受け付け、何らかの処理を行い、何かを出力します。

```
#[entry]
fn main() {
    let peripherals = setup_peripherals();
    loop {
        let inputs = read_inputs(&peripherals);
        let outputs = process(inputs);
        write_outputs(&peripherals, outputs);
    }
}
```

並行性がないため、プログラム間でのデータ共有や、ペリフェラルへのアクセス同期に悩む必要はありません。 このような単純なアプローチに逃れることができるのであれば、素晴らしい解決策かもしれません。

### 7.0.2. グローバルでミュータブルなデータ

組込みでない Rust と異なり、通常、ヒープ領域を作成し、そのデータへの参照を新しく作成したスレッドに渡す、というような贅沢はできません。代わりに、割り込みハンドラはいつでも呼び出される可能性があり、使用する共有メモリにアクセスする方法を知っていなければなりません。最も低いレベルでは、静的に割り当てられた ミュータブルなメモリを持つ必要があることを意味します。 このメモリは、割り込みハンドラとメインコードの両方が参照できます。

Rust では、このような `static mut` 変数への読み書きは、常にアンセーフです。 特別な注意を払わないと、レースコンディションを引き起こす可能性があります。つまり、その変数へのアクセスが、さらにその変数にアクセスする割り込みによって、中断されるということです。

この動作によって、コード内に分かりにくいエラーが発生する可能性があります。 例えば、1 秒毎に入力信号の立ち上がりエッジをカウントする組込みプログラム（周波数カウンタ）を考えてみましょう。

```
static mut COUNTER: u32 = 0;
```

```
#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // 危険。実際に安全ではありません。データ競合を引き起こす可能性があります。
            unsafe { COUNTER += 1 };
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    unsafe { COUNTER = 0; }
}
```

毎秒、タイマ割り込みはカウンタを 0 に戻します。同時に、メインループは信号を継続的に測定し、信号がローからハイに変わった時にカウンタをインクリメントします。 `static mut` な `COUNTER` にアクセスするためには、`unsafe` を使う必要があります。これは、未定義動作を引き起こさないことを、コンパイラに約束するということです。レースコンディションがどこにあるかわかりますか？ `COUNTER` のインクリメントは、アトミックであることが保証されて いません 。 実際、ほとんどの組込みプラットフォームにおいて、この操作は、ロードし、インクリメントし、ストアする、という動作に分割されます。割り込みがロードの後からストアの前に発生した場合、0 に戻すリセットは、割り込みから復帰した後に無視されます。 そして、その期間では、2 倍の遷移をカウントすることになります。

### 7.0.3. クリティカルセクション

それでは、データ競合についてどうすれば良いのでしょうか。 単純な方法は、割り込みが無効なコンテキストである クリティカルセクション を使うことです。`main` 中の `COUNTER` へのアクセスを、クリティカルセクションでラッピングします。 そうすることで、`COUNTER` のインクリメントが完了するまで、タイマ割り込みが発生しないようにできます。

```
static mut COUNTER: u32 = 0;

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // 新しいクリティカルセクションは、COUNTERへの同期アクセスを保証します
            cortex_m::interrupt::free(|_| {
                unsafe { COUNTER += 1 };
            });
        }
        last_state = state;
    }
}

#[interrupt]
```

```
fn timer() {
    unsafe { COUNTER = 0; }
}
```

この例では cortex_m::interrupt::free を使いました。他のプラットフォームでもクリティカルセクションのコードを実行するための、類似の方法があります。これは、割り込みを無効にして、コードを実行し、再び割り込みを有効にすることと同じです。

タイマ割り込み内クリティカルセクションを置く必要がないことに注意して下さい。これは次の 2 つの理由からです。

- 読み込みをしないため、COUNTER に 0 を書くことは、競合の影響を受けません
- いずれにせよ、main スレッドによって割り込まれることはありえません

COUNTER がお互いに プリエンプション する複数の割り込みハンドラから共有される場合、 それぞれにクリティカルセクションが必要になるでしょう。

クリティカルセクションは、当面の問題を解決しますが、慎重に検討しなければならない unsafe なコードをまだたくさん書いています。その結果、必要以上にクリティカルセクションを使用することになり、オーバーヘッドと割り込みレイテンシおよびジッタをもたらします。

注目すべき点は、クリティカルセクションでは、割り込みが発生しないことが保証されますが、マルチコアシステムでは、排他性の保証は提供されないことです。他のコアは、割り込みでなくても、とあるコアと同じメモリにアクセスできてしまいます。 マルチコアを使う場合、より強力な同期プリミティブが必要になります。

#### 7.0.4. アトミックアクセス

プラットフォームによっては、アトミック命令が利用できます。アトミック命令は、リードモディファイライト操作の保証を提供します。特に Cortex-M の場合、thumbv6（Cortex-M0）はアトミック命令を提供しませんが、thumbv7（Cortex-M3 以上）は提供します。 これらの命令は、全ての割り込みを無効化する手荒な方法の代替手段を提供します。 インクリメントを試みる時、ほとんどの場合は成功しますが、割り込まれた場合はインクリメント操作全体を自動的にやり直します。 このようなアトミック操作は、複数のコアにまたがっても安全です。

```
use core::sync::atomic::{AtomicUsize, Ordering};

static COUNTER: AtomicUsize = AtomicUsize::new(0);

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // 自動的にCOUNTERに1を加えるために`fetch_add`を使います
            COUNTER.fetch_add(1, Ordering::Relaxed);
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // COUNTERに直接0を書くために`store`を使います
```

```
    COUNTER.store(0, Ordering::Relaxed)
}
```

ここで、COUNTER は安全な static 変数です。AtomicUsize のおかげで COUNTER の型は、割り込みを無効化することなく、 割り込みハンドラとメインスレッドの両方から安全に修正できます。可能であれば、これはより良い解決方法です。しかし、あなたのプラットフォームではサポートされていないかもしれません。

Ordering の注釈：これは、コンパイラとハードウェアがどのように命令の順番を入れ替えるか、に影響を与えます。また、キャッシュの可視性にも影響します。ターゲットがシングルコアプラットフォームだと仮定すると、Relaxed で十分であり、このケースでは最も効率の良い選択です。 より厳密なオーダリングでは、コンパイラはアトミック操作の前後にメモリバリアを発行します。使用するアトミック操作によって、必要かもしれませんし、必要でないかもしれません！ アトミックモデルの正確な詳細は複雑であり、他の文書内でしっかりと説明されています。

詳細は、ノミコンのアトミックとオーダリングを参照して下さい。

#### 7.0.5. 抽象化、Send と Sync

上記の解決方法のいずれも、これと言って満足いくものではありません。 どの解決方法も unsafe ブロックを必要とし、非常に注意深くチェックしなければならず、人間工学的ではありません。 Rust ではもっとうまくやれるはずです！

カウンタを、コード内のどこからでも安全に使えるインタフェースに抽象化することができます。次の例では、クリティカルセクションカウンタを使いますが、アトミックと非常に良く似たことが実現できます。

```
use core::cell::UnsafeCell;
use cortex_m::interrupt;

// カウンタはUnsafeCell<u32>の単なるラッパです。UnsafeCellはRustの内部可変性の重要要素です。
// 内部可変性を使用することで、COUNTERを`static mut`の代わりに`static`として持つことができます。
// しかし、依然として、カウンタの値は変更することができます。
struct CSCounter(UnsafeCell<u32>);

const CS_COUNTER_INIT: CSCounter = CSCounter(UnsafeCell::new(0));

impl CSCounter {
    pub fn reset(&self, _cs: &interrupt::CriticalSection) {
        // クリティカルセクションを引数として要求することで、クリティカルセクション内で
        // 実行されなければならないことがわかります。そのため、このアンセーフブロックを
        // 自信を持って使用できます（UnsafeCell::getの呼び出しに必要です）。
        unsafe { *self.0.get() = 0 };
    }

    pub fn increment(&self, _cs: &interrupt::CriticalSection) {
        unsafe { *self.0.get() += 1 };
    }
}

// 静的なCSCounterを許可するために必要です。以下の説明を参照して下さい。
unsafe impl Sync for CSCounter {}

// 内部可変性を使用するため、COUNTERは、もはや`mut`ではありません。
// 従って、アクセスの際に、アンセーフなブロックも必要なくなりました。
```

```
static COUNTER: CSCounter = CS_COUNTER_INIT;

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // アンセーフはここでは必要ありません！
            interrupt::free(|cs| COUNTER.increment(cs));
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // 有効なcsトークンを得るため、ここでクリティカルセクションに入る必要があります。
    // 他の割り込みがプリエンプションを起こさないと分かっていても必要です。
    interrupt::free(|cs| COUNTER.reset(cs));

    // オーバーヘッドを避けるために、本当に必要であれば、偽のクリティカルセクションを生成する
    // アンセーフなコードを使うことができます。
    // let cs = unsafe { interrupt::CriticalSection::new() };
}
```

`unsafe` コードを慎重に検討された抽象の内側に移動しました。そして、アプリケーションコードは、`unsafe` ブロックを含んでいません。

この設計は、アプリケーションが `CriticalSection` トークンを渡すことを要求します。トークンは、`interrupt::free` によってのみ、安全に生成することができます。そのため、このトークンが渡されることを要求することで、自分自身でロックを実際にかけることなしに、クリティカルセクション内で動作していることを保証します。この保証は、静的にコンパイラによって提供されます。cs による実行時のオーバーヘッドはありません。カウンタが複数ある場合、複数の入れ子になったクリティカルセクションなしに、同じ cs を与えることができます。

これは、Rust の並行性についても重要なトピックを提起します。`Send` と `Sync` トレイトです。the Rust book を要約すると、安全に別のスレッドに移動できるとき、型は Send です。 一方、複数のスレッド間で安全に共有できるとき、型は Sync です。 組込みでは、割り込みがアプリケーションコードとは異なるスレッドで動作すると考えます。 そのため、割り込みとメインコードとの両方からアクセスされる変数は、Sync でなければなりません。

Rust のほとんどの型では、コンパイラによって Send と Sync の両方のトレイトが自動的に継承されます。 しかし、`CSCounter` は `UnsafeCell` を含んでいるため、Sync ではありません。 従って、`static CSCounter` を作ることはできません。`static` 変数は、複数のスレッドからアクセスされるため、Sync でなければなりません。

`CSCounter` が実はスレッド間で共有しても安全なように処理していることを、コンパイラに伝えるため、Sync トレイトを明示的に実装します。 これまでのクリティカルセクションの使用と同様に、シングルコアのプラットフォームでのみ安全です。マルチコアのプラットフォームでは、安全性を確保するためにさらなる取り組みが必要です。

### 7.0.6. ミューテックス

カウンタの問題に特有の便利な抽象化を行いましたが、並行性のために利用されるいくつかの抽象化が存在します。

そのような 同期プリミティブ の 1 つはミューテックス (mutex) です。mutex は mutual exclusion の略です。 ミューテックスは、私達のカウンタのような変数への排他アクセスを保証します。あるスレッドは、ミューテックスの ロック（または 獲得）を試みます。 すると、すぐに成功するか、ロックが獲得されるのを待ってブロックするか、ミューテックスをロックできなかったエラーを返します。そのスレッドがロックを保持している間、保護されたデータへのアクセスが許可されます。 そのスレッドが実行を完了すると、ミューテックスを アンロック（または 解放）することで、他のスレッドがミューテックスをロックできるようにします。 Rust では、通常、アンロックを実装するために `Drop` トレイトを使用します。 これは、ミューテックスがスコープの外に到達すると、常に解放されることを保証するためです。

割り込みハンドラでミューテックスを使用するのはトリッキーです。割り込みハンドラ内でブロックすることは、通常、好ましくありません。割り込みハンドラ内で、メインスレッドがロックを解放するのを待ってブロックすると、特に悲惨です。なぜならば。デッドロック になるからです。(割り込みハンドラ内に実行がとどまるため、メインスレッドがロックを解放することは決してありません) デッドロックはアンセーフとは考えられていません。安全な Rust でも発生する可能性があります。

この動作を完全に避けるため、カウンタの例で示すように、ロックのためにクリティカルセクションを必要とするミューテックスを実装できます。 クリティカルセクションがロックしている間続く限り、ミューテックスのロック/アンロックの状態を追跡することなしに、 ラップされた変数に排他的にアクセスできます。

これは実際に `cortex_m` クレートで行われています！ それを使ってカウンタを書くことができます。

```
use core::cell::Cell;
use cortex_m::interrupt::Mutex;

static COUNTER: Mutex<Cell<u32>> = Mutex::new(Cell::new(0));

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            interrupt::free(|cs|
                COUNTER.borrow(cs).set(COUNTER.borrow(cs).get() + 1));
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // ミューテックスを満たすために、ここでもクリティカルセクションに入る必要があります。
    interrupt::free(|cs| COUNTER.borrow(cs).set(0));
}
```

今回は Cell を使っています。これは、RefCell の同類で安全な内部可変性を提供するために使用されます。既に、UnsafeCell が、Rust の内部可変性の最下層であることを見てきました。UnsafeCell は、値への複数のミュータブル参照の取得を可能としますが、アンセーフなコードでのみ使用できます。Cell は UnsafeCell と似ていますが、安全なインタフェースを提供します。Cell は参照を取得せず、現在値のコピーを取得するか、置き換えることだけを許可します。Cell は Sync でないため、スレッド間で共有できません。これらの制約は安全に使えることを意味しますが、static 変数として直接使用できません。static は Sync である必要があるからです。

では、なぜ上記の例はうまく動くのでしょうか？Mutex<T>は、Cell のような Send な T に対してSync を実装します。このことが、Cell を static で使うことを安全にします。なぜなら、クリティカルセクションの間だけ、その中身へのアクセスを提供するからです。従って、全くアンセーフなコードなしに、安全なカウンタを手に入れることができます。

この方法は、カウンタの u32 のような単純な型に適しています。しかし、もっと複雑な Copy でない型についてはどうでしょうか？ 組込みにおいて非常に一般的な例は、ペリフェラル構造体です。これは、通常 Copy ではありません。そのためには、RefCell に頼ることができます。

#### 7.0.7. ペリフェラルの共有

svd2rust および同様の抽象化を使って生成されるデバイスクレートは、ペリフェラルへの安全なアクセスを提供します。これは、同時に 1 つのペリフェラル構造体インスタンスしか存在できないように強制することによって、もたらされます。このことは、安全性を保証しますが、メインスレッドと割り込みハンドとの両方からペリフェラルにアクセスすることを難しくします。

ペリフェラルアクセスを安全に共有するため、上で見たように Mutex を使うことができます。また、RefCell も必要です。RefCell は、実行時チェックを使って、同時に 1 つのペリフェラルへの参照だけが渡されるようにします。実行時チェックは、普通の Cell よりもオーバーヘッドが大きくなりますが、 コピーではなく参照を受け渡しするため、同時に存在するのが 1 つだけであることを確認する必要があります。

最後に、メインコード内でペリフェラルを初期化した後、なんとかしてペリフェラルを共有変数に移動する方法が必要です。 これを実現するために、Option 型を使います。None で初期化し、後でペリフェラルのインスタンスを設定します。

```
use core::cell::RefCell;
use cortex_m::interrupt::{self, Mutex};
use stm32f4::stm32f405;

static MY_GPIO: Mutex<RefCell<Option<stm32f405::GPIOA>>> =
    Mutex::new(RefCell::new(None));

#[entry]
fn main() -> ! {
    // ペリフェラルのシングルトンを取得し、設定します。
    // この例は、svd2rustで生成されたクレートから持ってきたものですが、
    // ほとんどの組込みデバイスクレートは同様になります。
    let dp = stm32f405::Peripherals::take().unwrap();
    let gpioa = &dp.GPIOA;

    // 一連の設定をする関数です。
    // PA0を入力、PA1を出力に設定すると仮定して下さい。
    configure_gpio(gpioa);
```

```
    // GPIOAをミューテックスに格納し、ムーブします。
    interrupt::free(|cs| MY_GPIO.borrow(cs).replace(Some(dp.GPIOA)));
    // もはや`gpioa`や`dp.GPIOA`は使いません。
    // 代わりに、ミューテックス経由でアクセスする必要があります。

    // MY_GPIOを設定した後にのみ、割り込みを有効にするように注意して下さい。
    // そうしなければ、まだNoneが含まれている間に、割り込みが発生する可能性があります。
    //  (`unwrap()`を使用して) 書き込まれると、パニックになるでしょう。
    set_timer_1hz();
    let mut last_state = false;
    loop {
        // ミューテックス経由で、デジタル入力としての状態を読み込みます。
        let state = interrupt::free(|cs| {
            let gpioa = MY_GPIO.borrow(cs).borrow();
            gpioa.as_ref().unwrap().idr.read().idr0().bit_is_set()
        });

        if state && !last_state {
            // PA0の立ち上がりエッジを検出した場合、PA1をハイに設定します。
            interrupt::free(|cs| {
                let gpioa = MY_GPIO.borrow(cs).borrow();
                gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().set_bit());
            });
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // 今回は、割り込み内では単純にPA0をクリアするだけです。
    interrupt::free(|cs| {
        // `unwrap()`を使うことができます。割り込みはMY_GPIOが設定されるまで有効化されないことを
        // 知っているためです。そうでなければ、Noneを処理しなければならないでしょう。
        let gpioa = MY_GPIO.borrow(cs).borrow();
        gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().clear_bit());
    });
}
```

非常に多くのことを取り入れています。重要な部分を詳細に見ていきましょう。

```
static MY_GPIO: Mutex<RefCell<Option<stm32f405::GPIOA>>> =
    Mutex::new(RefCell::new(None));
```

ここでは、共有変数は `RefCell` を内部に含む `Mutex` です。さらに、RefCell は `Option` を含んでいます。`Mutex` はクリティカルセクションの間だけ、アクセスできるようにします。その結果、普通の `RefCell` は Sync でないにも関わらず、変数は Sync になります。`RefCell` は、`GPIOA` を使うのに必要となる参照によって内部可変性を提供します。`Option` を使用すると、この変数を空の値に初期化できます。後で実際に変数を移動します。ペリフェラルのシングルトンには静的にアクセスすることはできません。実行時のみアクセスできるため、Option が必要とされます。

```
interrupt::free(|cs| MY_GPIO.borrow(cs).replace(Some(dp.GPIOA)));
```

クリティカルセクションの内部で、ミューテックスの borrow()を呼んでいます。borrow()は RefCell の参照を提供します。 その後、replace()を呼び出して、RefCell に新しい値をムーブします。

```
interrupt::free(|cs| {
    let gpioa = MY_GPIO.borrow(cs).borrow();
    gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().set_bit());
});
```

最後に、MY_GPIO を安全で並行なやり方で使います。クリティカルセクションは、通常通り割り込みの発生を防ぎ、ミューテックスを借用できます。 RefCell は &Option<GPIOA>を提供し、その借用がいつまで続くかを追跡します。その参照がスコープ外になると、RefCell が借用されなくなったことを示すため、更新されます。

&Option の外に GPIOA をムーブすることはできないので、as_ref()を使って &Option<&GPIOA>に変換します。そうすると、最終的に、unwrap()で &GPIOA を取得できます。&GPIOA により、ペリフェラルを修正することができます。

未翻訳

```
use core::cell::RefCell;
use core::ops::DerefMut;
use cortex_m::interrupt::{self, Mutex};
use cortex_m::asm::wfi;
use stm32f4::stm32f405;

static G_TIM: Mutex<RefCell<Option<Timer<stm32::TIM2>>>> =
    Mutex::new(RefCell::new(None));

#[entry]
fn main() -> ! {
    let mut cp = cm::Peripherals::take().unwrap();
    let dp = stm32f405::Peripherals::take().unwrap();

    // 未翻訳
    let tim = configure_timer_interrupt(&mut cp, dp);

    interrupt::free(|cs| {
        G_TIM.borrow(cs).replace(Some(tim));
    });

    loop {
        wfi();
    }
}

#[interrupt]
fn timer() {
    interrupt::free(|cs| {
        if let Some(ref mut tim)) =  G_TIM.borrow(cs).borrow_mut().deref_mut() {
            tim.start(1.hz());
        }
    });
}
```

ヒューッ！これは安全ですが、少し大げさすぎて扱いにくいです。他に方法はないのでしょうか？

#### 7.0.8. RTIC

代替手段の１つは、RTFM フレームワークです。RTFM は、Real Time For the Masses の略です。RTFM は、共有リソースが常に安全にアクセスされることを保証するために、静的な優先度を適用し、 `static mut` 変数（「リソース」）へのアクセスを追跡します。 この方法は、（`RefCell` のように）常にクリティカルセクションに入り、参照カウントを使うというオーバーヘッドを必要としません。 デッドロックがないことを保証したり、時間とメモリのオーバーヘッドを極めて小さくするといった、多くの利点があります。

未翻訳

このフレームワークは他の機能も含んでいます。例えば、メッセージパッシングは明示的な共有状態の必要性を減らします。また、タスクを指定した時間に実行するスケジュールする機能もあります。これは、周期タスクの実装に使えます。詳しくはドキュメントを参照して下さい。

#### 7.0.9. Embassy

未翻訳

未翻訳

未翻訳

#### 7.0.10. リアルタイムオペレーティングシステム

翻訳が古くなっています 組込み向け並行性の異なる一般的なモデルとして、リアルタイムオペレーティングシステム（RTOS）があります。 現在、Rust ではあまり検証されていませんが、従来の組込み開発では広く使用されています。 オープンソースの例として、FreeRTOS と ChibiOS があります。 これらの RTOS は、複数のアプリケーションスレッドを動作させる機能を提供しています。 スレッドが制御を明け渡す時（コオペレーティブマルチタスク）か、 周期タイマまたは割り込みに基づく時（プリエンプティブマルチタスク）に、CPU で実行するスレッドを切り替えます。RTOS は、通常ミューテックスや他の同期プリミティブを提供します。また、DMA エンジンようなハードウェア機能を同時に使えることも多いです。

この本を書いている時点では、Rust で書かれた RTOS の例はそれほど多くありません。 しかし、興味深い分野ですので、この分野にご注目下さい！

#### 7.0.11. マルチコア

組込みプロセッサにおいても、2 個以上のコアを持つことがより一般的になってきています。このことは、並行性をさらに複雑にします。（`cortex_m::interrupt::Mutex` を含む）クリティカルセクションで使っている全ての例は、 他の実行スレッドは、割り込みスレッドだけであることを前提にしています。しかし、マルチコアシステムにおいては、これは当てはまりません。代わりに、マルチコア（SMP; symmetric multi-proccesing とも呼ばれます）向けに設計した同期プリミティブが必要になります。

通常、これまでに見たアトミック命令を使用します。アトミック命令は、処理システムが全てのコア間でのアトミック性を維持してくれるためです。

これらのトピックを詳細に説明することは、この本のスコープ範囲外ですが、一般的なパターンはシングルコアの場合と同じです。

# 8. コレクション

いずれは、プログラム内で動的なデータ構造（別名コレクション）を使いたいでしょう。 `std` は、`Vec` や `String`、`HashMap` といった、一般的なコレクションを提供しています。`std` で実装されている全てのコレクションは、グローバルな動的アロケータ（別名ヒープ）を使用します。

`core` は、定義上、メモリアロケーションがないためコレクションの実装を使うことができません。しかし、コンパイラと共に配布されている安定化していない `alloc` クレートの中にコレクションの実装があります。

もしコレクションが必要であれば、ヒープに割り当てる実装だけが選択肢ではありません。サイズが固定されたコレクションを使うことができます。そのような実装は `heapless` クレートの中にあります。

`alloc` クレートは、標準の Rust 配布物に同梱されています。このクレートをインポートするには、 `Cargo.toml` ファイルに依存関係を宣言することなしに直接 `use` します。

### 8.0.1. `alloc` を使用

`alloc` クレートは、標準の Rust 配布物に同梱されています。このクレートをインポートするには、 `Cargo.toml` ファイルに依存関係を宣言することなしに直接 `use` します。

```
#![feature(alloc)]

extern crate alloc;

use alloc::vec::Vec;
```

コレクションを使うには、まず最初に、プログラム中のグローバルアロケータを宣言する `global_allocator` アトリビュートを使う必要があります。選択したアロケータが `GlobalAlloc` トレイトを実装することが求められます。

このセクションを可能な限り自己完結させるため、グローバルアロケータとして、単純にポインタを増加するだけのアロケータを実装します。しかしながら、あなたのプログラムではこのアロケータでなく、crates.io から歴戦のアロケータを使用することを強くお勧めします。

```
// ポインタを増加するだけのアロケータ実装

use core::alloc::{GlobalAlloc, Layout};
use core::cell::UnsafeCell;
use core::ptr;

use cortex_m::interrupt;

// *シングル*コアシステム用のポインタを増加するだけのアロケータ
struct BumpPointerAlloc {
    head: UnsafeCell<usize>,
    end: usize,
}

unsafe impl Sync for BumpPointerAlloc {}

unsafe impl GlobalAlloc for BumpPointerAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        // `interrupt::free`は、割り込み内でアロケータを安全に使用するための
        // クリティカルセクションです。
        interrupt::free(|_| {
```

```
            let head = self.head.get();
            let size = layout.size();
            let align = layout.align();
            let align_mask = !(align - 1);

            // ヌルポインタはメモリ不足の状態を知らせます
            let start = (*head + align - 1) & align_mask;

            if start + size > self.end {
                // このアロケータはメモリを解放しません
                ptr::null_mut()
            } else {
                *head = start + size;
                start as *mut u8
            }
        })
    }

    unsafe fn dealloc(&self, _: *mut u8, _: Layout) {
        // このアロケータはメモリを解放しません
    }
}

// グローバルメモリアロケータの宣言
// ユーザはメモリ領域の`[0x2000_0100, 0x2000_0200]`がプログラムの他の部分で使用されないことを
// 保証しなければなりません
#[global_allocator]
static HEAP: BumpPointerAlloc = BumpPointerAlloc {
    head: UnsafeCell::new(0x2000_0100),
    end: 0x2000_0200,
};
```

グローバルアロケータの選択とは別に、ユーザはメモリ不足 (OOM) エラーの処理方法を、安定化していない `alloc_error_handler` アトリビュートを使って定義する必要があります。

```
#![feature(alloc_error_handler)]

use cortex_m::asm;

#[alloc_error_handler]
fn on_oom(_layout: Layout) -> ! {
    asm::bkpt();

    loop {}
}
```

全ての準備が整うと、ユーザはついに `alloc` のコレクションを使うことができます。

```
#[entry]
fn main() -> ! {
    let mut xs = Vec::new();

    xs.push(42);
    assert!(xs.pop(), Some(42));

    loop {
        // ..
```

```
    }
}
```

std クレートのコレクションを使ったことがあれば、実装が全く同じものであるため、これらのコレクションはお馴染みでしょう。

#### 8.0.2. heapless の使用

heapless のコレクションはグローバルメモリアロケータに依存しないため、準備は不要です。単にコレクションを use して、インスタンスを作成するだけです。

```
// heapless version: v0.4.x
use heapless::Vec;
use heapless::consts::*;

#[entry]
fn main() -> ! {
    let mut xs: Vec<_, U8> = Vec::new();

    xs.push(42).unwrap();
    assert_eq!(xs.pop(), Some(42));
    loop {}
}
```

alloc のコレクションとは違う点が 2 つあることに留意して下さい。

1 つ目は、コレクションの容量を最初に宣言しなければならないことです。 heapless コレクションは、再割り当てが発生せず、固定の容量になります。この容量はコレクションの型シグネチャの一部になります。上記の例では、xs は 8 要素の容量を持つように宣言しています。このベクタは最大で 8 つの要素を保持することができます。型シグネチャの U8（typenum を参照）がこのことを表しています。

2 つ目は、push メソッドおよび他の多くのメソッドが Result を返すことです。 heapless コレクションは固定の容量を持つため、コレクションに要素を挿入する全ての操作は、失敗する可能性があります。 API は、操作が成功したかどうかを示すための Result を返すことで、この問題に対処しています。一方、alloc コレクションは、ヒープ上で再割り当てするため、容量を増やすことができます。

v0.4.x 以降、全ての heapless コレクションは、全ての要素をインラインで格納しています。つまり、 `let x = heapless::Vec::new();`のような操作は、スタック上にコレクションを割り当てます。また、コレクションを static 変数や、ヒープ上（Box<Vec<_, _>>）にさえ、に割り当てることが可能です。

#### 8.0.3. トレードオフ

ヒープに割り当てられる再配置可能なコレクションと固定容量のコレクションとを選定する時は、次のことに留意して下さい。

##### 8.0.3.1. メモリ不足とエラー処理

ヒープアロケーションでは、メモリ不足は常に発生する可能性があり、コレクションが拡大する場所であれば、どこでも発生する可能性があります。例えば、全ての alloc::Vec.push 呼び出しは、OOM 状態を引き起こす可能性があります。 そのため、一部の操作は暗黙的に失敗する可能性があります。 一部の alloc コレクションは try_reserve メソッドを提供しています。このメソッドにより、コレクションを拡大する時に OOM 状態が発生するかどうかを確認できますが、先を見越して使用する必要があります。

heapless コレクションだけを使っていて、メモリアロケータを使用しないのであれば、OOM 状態は発生しません。 その代わりに、コレクションの容量オーバーを個別に処理しなければなりません。つまり、Vec.push のようなメソッドが返す全ての Result を処理することになります。

OOM 障害は、heapless::Vec.push が返す全ての Result を unwrap するより、デバッグが難しいでしょう。 なぜなら、障害が発生した場所は、問題の原因となる場所と一致しない可能性があるからです。例えば、他のコレクションがメモリリークを起こしているせいでアロケータが枯渇しそうな場合、vec.reserve(1)が OOM を発生させる可能性があります（メモリリークは安全な Rust でも発生します）。

#### 8.0.3.2. メモリ使用量

長期間使われるコレクションの容量は、実行時に変わる可能性があるため、ヒープ割り当てされたコレクションのメモリ使用量を推測することは難しいです。一部の操作は、暗黙的にコレクションを再割り当てし、メモリ使用量が増加します。一部のコレクションは、shrink_to_fit のようなメソッドを持っており、コレクションが使用しているメモリを減らすこともあります。最終的に、実際にメモリアロケーションを縮小するかどうかは、アロケータ次第です。さらに、アロケータは、メモリフラグメンテーションを扱う必要があります。このことは見かけ上のメモリ使用量を増やす可能性があります。

一方で、固定容量のコレクションだけを使用して、そのほとんどを static 変数に格納し、コールスタックの最大サイズを設定すると、 リンカは、物理的に利用可能なメモリより大きな容量を使おうとしたかどうか検出します。

その上、スタックに割り当てられた固定容量のコレクションは、-Z emit-stack-sizes フラグによって報告されます。 このフラグは、（stack-sizes のような）スタック使用量を解析するツールがスタック使用量を解析することを意味します。

しかし、固定容量のコレクションは、縮小することができません。 再配置可能なコレクションよりも負荷率（コレクションのサイズとその容量の比率）が低くなる可能性があります。

#### 8.0.3.3. 最悪実行時間（WCET; Worst Case Execution Time）

時間制約のあるアプリケーションやハードリアルタイムアプリケーションを作成している場合、 プログラムの様々な部分で最悪実行時間が気になるでしょう。

alloc コレクションは再割り当てする可能性があるため、コレクションが拡大する操作の最悪実行時間は、 コレクションが再割り当てされるのにかかる時間も含みます。 コレクションが再割り当てされるかどうかは、実行時のコレクションの容量に依存します。このことは、alloc::Vec.push といった操作の最悪実行時間の決定を難しくします。この操作の最悪実行時間は、使用するアロケータとコレクションの実行時容量との両方に依存するためです。

一方、固定容量のコレクションは再割り当てが発生しないため、全ての操作の実行時間が予測可能です。 例えば、heapless::Vec.push は定数時間で実行します。

#### 8.0.3.4. 使いやすさ

alloc はグローバルアロケータの準備が必要ですが、heapless はそうではありません。しかし、heapless は、インスタンスを作成する各コレクションの容量を指定する必要があります。

alloc API は、事実上、全ての Rust 開発者がなじみのあるものです。 heapless API は、alloc API に似せてはいますが、明示的なエラー処理のため、全く同じにはなりません。一部の開発者はこの明示的なエラー処理を、度が過ぎていたり、面倒すぎる、と感じるかもしれません。

# 9. 未翻訳

未翻訳

## 9.1. 未翻訳

未翻訳

### 9.1.1. 未翻訳

未翻訳

### 9.1.2. 未翻訳

#### 9.1.2.1. 未翻訳 (C-CRATE-NAME)

未翻訳

### 9.1.3. 未翻訳

#### 9.1.3.1. 未翻訳 (C-FREE)

未翻訳

未翻訳

未翻訳

未翻訳

```
# pub struct TIMER0;
pub struct Timer(TIMER0);

impl Timer {
    pub fn new(periph: TIMER0) -> Self {
        Self(periph)
    }

    pub fn free(self) -> TIMER0 {
        self.0
    }
}
```

#### 9.1.3.2. 未翻訳 (C-REEXPORT-PAC)

未翻訳

未翻訳

#### 9.1.3.3. 未翻訳 (C-HAL-TRAITS)

未翻訳

未翻訳

### 9.1.4. 未翻訳

#### 9.1.4.1. 未翻訳 (C-CTOR)

未翻訳

未翻訳

#### 9.1.4.2. 未翻訳 (C-INLINE)

未翻訳

### 9.1.5. 未翻訳

#### 9.1.5.1. 未翻訳 (C-ZST-PIN)

未翻訳

未翻訳

未翻訳

```
pub struct PA0;
pub struct PA1;
// ...

pub struct PortA;

impl PortA {
    pub fn split(self) -> PortAPins {
        PortAPins {
            pa0: PA0,
            pa1: PA1,
            // ...
        }
    }
}

pub struct PortAPins {
    pub pa0: PA0,
    pub pa1: PA1,
    // ...
}
```

#### 9.1.5.2. 未翻訳 (C-ERASED-PIN)

未翻訳

未翻訳

```
/// 未翻訳
pub struct PA0;

impl PA0 {
    pub fn erase_pin(self) -> PA {
        PA { pin: 0 }
    }
}

/// 未翻訳
pub struct PA {
    /// 未翻訳
    pin: u8,
}

impl PA {
    pub fn erase_port(self) -> Pin {
        Pin {
```

```
            port: Port::A,
            pin: self.pin,
        }
    }
}

pub struct Pin {
    port: Port,
    pin: u8,
    // 未翻訳
}

enum Port {
    A,
    B,
    C,
    D,
}
```

#### 9.1.5.3. 未翻訳 (C-PIN-STATE)

未翻訳

未翻訳

未翻訳

未翻訳

未翻訳

- `pub fn into_input<N: InputState>(self, input: N) -> Pin<N>`
- `pub fn into_output<N: OutputState>(self, output: N) -> Pin<N>`
- ```
  pub fn with_input_state<N: InputState, R>(
      &mut self,
      input: N,
      f: impl FnOnce(&mut PA1<N>) -> R,
  ) -> R
  ```
- ```
  pub fn with_output_state<N: OutputState, R>(
      &mut self,
      output: N,
      f: impl FnOnce(&mut PA1<N>) -> R,
  ) -> R
  ```

未翻訳

未翻訳

```
# use std::marker::PhantomData;
mod sealed {
    pub trait Sealed {}
}

pub trait PinState: sealed::Sealed {}
pub trait OutputState: sealed::Sealed {}
pub trait InputState: sealed::Sealed {
    // ...
}

pub struct Output<S: OutputState> {
```

```
    _p: PhantomData<S>,
}

impl<S: OutputState> PinState for Output<S> {}
impl<S: OutputState> sealed::Sealed for Output<S> {}

pub struct PushPull;
pub struct OpenDrain;

impl OutputState for PushPull {}
impl OutputState for OpenDrain {}
impl sealed::Sealed for PushPull {}
impl sealed::Sealed for OpenDrain {}

pub struct Input<S: InputState> {
    _p: PhantomData<S>,
}

impl<S: InputState> PinState for Input<S> {}
impl<S: InputState> sealed::Sealed for Input<S> {}

pub struct Floating;
pub struct PullUp;
pub struct PullDown;

impl InputState for Floating {}
impl InputState for PullUp {}
impl InputState for PullDown {}
impl sealed::Sealed for Floating {}
impl sealed::Sealed for PullUp {}
impl sealed::Sealed for PullDown {}

pub struct PA1<S: PinState> {
    _p: PhantomData<S>,
}

impl<S: PinState> PA1<S> {
    pub fn into_input<N: InputState>(self, input: N) -> PA1<Input<N>> {
        todo!()
    }

    pub fn into_output<N: OutputState>(self, output: N) -> PA1<Output<N>> {
        todo!()
    }

    pub fn with_input_state<N: InputState, R>(
        &mut self,
        input: N,
        f: impl FnOnce(&mut PA1<N>) -> R,
    ) -> R {
        todo!()
    }

    pub fn with_output_state<N: OutputState, R>(
        &mut self,
        output: N,
```

```
        f: impl FnOnce(&mut PA1<N>) -> R,
    ) -> R {
        todo!()
    }
}

// 未翻訳
```

# 10. 組込み C 開発者へのヒント

この章では、組込み C 開発の経験者が、Rust を書き始める時に役に立つ様々なヒントをまとめます。 特に、既に C 言語で慣れ親しんでいることが、Rust ではどう違うのかを強調します。

### 10.0.1. プリプロセッサ

組込み C では、次のような様々な目的でプリプロセッサを使うことが一般的です。

- `#ifdef` を使ったコンパイル時のコードブロック選択
- コンパイル時の配列サイズやコンパイル時計算
- （関数呼び出しのオーバーヘッドを避けるための）共通パターンを簡単化するマクロ

Rust にはプリプロセッサはありません。上記のユースケースは異なる方法で解決されます。セクションの残りの部分では、プリプロセッサの様々な代替手段について説明します。

#### 10.0.1.1. コンパイル時コード選択

`#ifdef ... #endif` に最も近い Rust の機能は、Cargo フィーチャです。Cargo フィーチャは、C プリプロセッサよりももう少し秩序だったものです。フィーチャの候補は、クレートごとに明示的にリスト化されており、オンまたはオフのいずれかになります。依存関係としてクレートを記載すると、フィーチャが有効になります。またこのフィーチャは追加式です。依存ツリー内の何らかのクレートが、別クレートのフィーチャを有効化した場合、そのフィーチャは、そのクレートを使う全てのユーザに対して有効化されます。

例えば、信号処理プリミティブを提供するライブラリがあるとします。それぞれが、大きな定数テーブルをコンパイルまたは宣言するのに余分な時間がかかるとすると、その時間を回避したいと思うでしょう。 `Cargo.toml` 内で各コンポーネントのフィーチャを宣言することができます。

```
[features]
FIR = []
IIR = []
```

それから、コード内で、何をインクルードするか制御するために `#[cfg(feature="FIR")]`を使います。

```
/// トップレベルのlib.rs内

#[cfg(feature="FIR")]
pub mod fir;

#[cfg(feature="IIR")]
pub mod iir;
```

同様に、フィーチャが有効になって いない 場合にだけコードブロックをインクルードすることができます。また、フィーチャの組み合わせや、フィーチャが有効か無効かに関わらず、コードブロックをインクルードすることもできます。

さらに、Rust は、自動的に設定される数々の条件を提供します。例えば、アーキテクチャに基づいて異なるコードを選択する `target_arch` です。条件コンパイルがサポートしている全ての詳細については、Rust リファレンスの条件コンパイルの章を参照して下さい。

条件コンパイルは、次のステートメントまたはブロックにのみ適用されます。 現在のスコープ内でブロックが使えない場合、`cfg` アトリビュートは複数回必要になります。ほとんどの場合、単純に全てのコードをインクルードして、コンパイラが最適化時にデッドコードを削除できるようにするほうが良いことに、注意すべきです。 これは、あなたにも、あなたのユーザに

とってもより簡単です。そして、通常、コンパイラは使用されていないコードをうまく削除します。

#### 10.0.1.2. コンパイル時サイズとコンパイル時計算

Rust は `const fn` を提供しています。この関数はコンパイル時に評価されることが保証されているため、配列のサイズなど、定数が求められる場所で使用できます。 const fn は、上述したフィーチャと同時に使う事ができます。例を示します。

```
const fn array_size() -> usize {
    #[cfg(feature="use_more_ram")]
    { 1024 }
    #[cfg(not(feature="use_more_ram"))]
    { 128 }
}

static BUF: [u32; array_size()] = [0u32; array_size()];
```

これらは、Rust 1.31 以降の新しい機能であるため、ドキュメントはまだわずかしかありません。執筆時点では、`const fn` で利用可能な機能は、非常に限られています。将来の Rust では、`const fn` 内で許可されることが拡張されていくでしょう。

#### 10.0.1.3. マクロ

Rust は、極めて強力なマクロシステムを提供しています。C プリプロセッサがソースコードのテキストにほぼ直接作用するのに対して、Rust のマクロシステムはより上位レベルで作用します。 Rust のマクロは 2 種類あります。例によるマクロ と 手続きマクロ です。前者はより単純で最も一般的なものです。関数呼び出しのように見えて、完全な式やステートメント、アイテム、パターンに展開できます。手続きマクロは、より複雑ですが、Rust 言語に非常に強力な拡張を許可します。任意の Rust 構文を、新しい Rust 構文に変換することができます。

通常、C プリプロセッサマクロを使っていた場所に、例によるマクロで同じことができるかどうか、確認したいと思います。マクロは、クレート内に定義でき、自身のクレート内で簡単に使ったり、他のユーザにエクスポートしたりできます。マクロは、完全な式や、ステートメント、アイテム、パターンに展開されなければならないため、C プリプロセッサマクロのいくつかのユースケースではうまく機能しません。例えば、変数名や、リスト内の不完全なアイテムの一部に展開するようなマクロです。

Cargo フィーチャと同様、本当にマクロが必要かどうか、は検討する価値があります。多くの場合、通常の関数は理解しやすく、マクロと同様にインライン化されます。`#[inline]`および`#[inline(always)]`のアトリビュートを使用すると、このプロセスをさらに細かく制御できます。ここでも注意が必要です。コンパイラは、適切な場合、関数を自動的にインライン化します。そのため、不適切なインライン化を強制すると、パフォーマンスが低下する可能性があります。

Rust のマクロシステムの全体を説明することは、このヒントページのスコープ範囲外です。詳細については、Rust のドキュメントの参照をお勧めします。

### 10.0.2. ビルドシステム

(必須ではありませんが) ほとんどの Rust のクレートは、Cargo を使ってビルドされます。Cargo は、従来のビルドシステムに関する多くの難しい問題の面倒を見ています。しかし、ビルドプロセスをカスタマイズしたいと思うかもしれません。このため、Cargo は `build.rs` スクリプト

を提供しています。 build.rs スクリプトは Rust で書かれたスクリプトで、必要に応じて Cargo のビルドシステムとやり取りします。

ビルドスクリプトの一般的なユースケースを示します。

- ビルド時の情報を提供します。例えば、実行ファイルにビルド日時や Git のコミットハッシュを静的に埋め込みます。
- 選択されたフィーチャやその他のロジックに応じて、リンカスクリプトをビルド時に生成します。
- Cargo のビルド設定を変更します。
- リンクする静的ライブラリを追加します。

現状、ビルド後に実行するスクリプトは提供されていません。 そのようなスクリプトは、従来では、ビルドしたオブジェクトからバイナリを自動的に生成したり、ビルド情報を表示したりするタスクに使われています。

#### 10.0.2.1. クロスコンパイル

Cargo をビルドシステムに使用するとクロスコンパイルも簡単になります。多くの場合、Cargo に `--target thumbv6m-none-eabi` を伝えるだけで十分です。 そうすると、適切な実行ファイルが `target/thumbv6m-none-eabi/debug/myapp` に見つかります。

Rust が本来サポートしていないプラットフォームの場合、ターゲットの `libcore` を自分自身でビルドする必要があります。そのようなプラットフォームでは、Xargo を Cargo の代わりに使うことができ、自動的に `libcore` をビルドしてくれます。

### 10.0.3. イテレータ vs 配列アクセス

C では、おそらくインデックスによって直接配列にアクセスしているでしょう。

```
int16_t arr[16];
int i;
for(i=0; i<sizeof(arr)/sizeof(arr[0]); i++) {
    process(arr[i]);
}
```

Rust では、これはアンチパターンです。インデックスによるアクセスは、低速（境界チェックが必要なため）で様々なコンパイラの最適化を妨げます。これは重要な違いであり、繰り返す価値があります。Rust は、メモリ安全性を保証するために、手動で配列のインデックスを指定する際、境界を越えたアクセスをチェックします。一方、C では配列外のインデックスにアクセスできてしまいます。

代わりに、イテレータを使います。

```
let arr = [0u16; 16];
for element in arr.iter() {
    process(*element);
}
```

イテレータは、chaining、zipping、enumerating、最小値や最大値の検索、合計の算出など、C では手動で実装する必要がある強力な配列の機能を提供します。 イテレータのメソッドは、連鎖することができ、非常に読みやすいデータ処理のコードになります。

詳細は the Book のイテレータとイテレータのドキュメントを参照して下さい。

### 10.0.4. 参照 vs ポインタ

Rust でも、ポインタ（[生ポインタ と呼びます]）は存在しますが、限られた状況でしか使いません。ポインタの参照外しは、常に `unsafe` と考えられるからです。Rust は、ポインタの背後にあるかもしれないものについて、通常の保証を提供できません。

代わりに、ほとんどの場合、`&` のシンボルで表現される 参照 もしくは `&mut` で表現されるミュータブルな参照 を使います。 参照は、ポインタと似た働きをします。つまり、裏にある値にアクセスするために参照外しができます。しかし、参照は、Rust の所有権システムの重要な一部です。Rust は、どんな時でも同じ値に対して、唯一のミュータブル参照を持つか、あるいは、複数のイミュータブル参照を持つか、を厳密に強制します。

実際のところ、データへのミュータブルアクセスが必要かどうか、をより慎重に検討する必要があることを意味します。C ではデフォルトがミュータブルであり、明示的に `const` をつける必要があります。Rust ではその反対です。

生ポインタを使う可能性のある状況の 1 つは、直接ハードウェアとやり取りする時です（DMA ペリフェラルのレジスタにバッファのポインタを書き込むなど）。また、生ポインタは、メモリマップドレジスタの読み書きを可能にするために、ペリフェラルアクセスクレートの内部で使われています。

### 10.0.5. Volatile アクセス

C では、個別の変数に `volatile` を付けることができます。 これは、変数の値がアクセスごとに変わるかもしれない、ということをコンパイラに伝えます。 組込みでは、Volatile 変数はメモリマップドレジスタに広く使用されています。

Rust では、変数に `volatile` を付けるのではなく、volatile アクセスをするための特定のメソッドを使います。`core::ptr::read_volatile` と `core::ptr::write_volatile` です。これらのメソッドは、`*const T` か`*mut T`（上述の通り 生ポインタ です）を受け取り、volatile な読み書きを行います。

例えば、C では次のように書きます。

```
volatile bool signalled = false;

void ISR() {
    // 割り込みが発生したというシグナル
    signalled = true;
}

void driver() {
    while(true) {
        // シグナルがあるまでスリープします
        while(!signalled) { WFI(); }
        // シグナルをリセットします
        signalled = false;
        // 割り込みを待っていた何らかのタスクを実行します
        run_task();
    }
}
```

Rust で同じことをするには、各アクセスに volatile メソッドを使用します。

```
static mut SIGNALLED: bool = false;

#[interrupt]
```

```
fn ISR() {
    // 割り込みが発生したというシグナル
    //  （実際のコードでは、アトミック型のような、より上位レベルのプリミティブを検討して下さい）
    unsafe { core::ptr::write_volatile(&mut SIGNALLED, true) };
}

fn driver() {
    loop {
        // シグナルがあるまでスリープします
        while unsafe { !core::ptr::read_volatile(&SIGNALLED) } {}
        // シグナルをリセットします
        unsafe { core::ptr::write_volatile(&mut SIGNALLED, false) };
        // 割り込みを待っていた何らかのタスクを実行します
        run_task();
    }
}
```

このコードサンプルには、いくつかの注目すべき点があります。

- `*mut T` を要求する関数に、`&mut SIGNALLED` を渡すことができます。

これは、`&mut T` が`*mut T` に自動的に変換されるためです（`*const T` についても同じです）。

- `read_volatile`/`write_volatile` メソッドに `unsafe` ブロックが必要です。

これらの関数は `unsafe` だからです。安全な使用を保証することはプログラマの責任です。 詳細は、メソッドのドキュメントを参照して下さい。

これらの関数をコードに直接書くことは稀です。通常、より上位レベルのライブラリで面倒を見てくれます。 メモリマップドペリフェラルについては、ペリフェラルアクセスクレートが volatile アクセスを自動的に実装します。 並行性プリミティブの場合、より優れた抽象化が利用できます（並行性の章を参照して下さい）。

#### 10.0.6. パック型と整列型

組込み C では、通常、特定のハードウェアやプロトコルの要件を満たすために、変数に特定のアライメントが必要なことや、 構造体が整列されているだけでなくパックされている必要があることを、コンパイラに指示することが一般的です。

Rust では、これは構造体または共用体の `repr` アトリビュートによって制御されます。デフォルトでは、レイアウトは保証されないため、ハードウェアや C とやり取りするコードでは使うべきではありません。コンパイラは、構造体のメンバを並べ替えたり、パディングを挿入したりする可能性があります。この動作は将来のバージョンの Rust で変更になる可能性があります。

```
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
}

// 0x7ffecb3511d0 0x7ffecb3511d4 0x7ffecb3511d2
// データの詰め方を改善するために、x, y, zの順序が入れ替わっていることに注目して下さい。
```

C と相互にやり取りできるレイアウトを保証するためには、`repr(C)`を使います。

```
#[repr(C)]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
}

// 0x7fffd0d84c60 0x7fffd0d84c62 0x7fffd0d84c64
// 順序は維持され、レイアウトは時間が経っても変化しません。
// `z`は2バイトで整列されており、`y`と`z`の間には、1バイトのパディングが存在します。
```

パックされた表現を保証する場合、`repr(packed)`を使います。

```
#[repr(packed)]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    // パックされた構造体のフィールドを借用するには、アンセーフが必要です。
    let px = std::ptr::addr_of!(v.x);
    let py = std::ptr::addr_of!(v.y);
    let pz = std::ptr::addr_of!(v.z);
    println!("{:p} {:p} {:p}", px, py, pz);
}

// 0x7ffd33598490 0x7ffd33598492 0x7ffd33598493
// `y`と`z`の間にパディングは挿入されていません。そのため、`z`は整列されていません。
```

`repr(packed)`を使うと、型のアライメントも 1 に設定されることに注意して下さい。

最後に、特定のアライメントを指定するために、`repr(align(n))`を使います。 ここで `n` は、整列するバイト数です（2 の累乗である必要があります）。

```
#[repr(C)]
#[repr(align(4096))]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    let u = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
    println!("{:p} {:p} {:p}", &u.x, &u.y, &u.z);
}

// 0x7ffec909a000 0x7ffec909a002 0x7ffec909a004
```

```
// 0x7ffec909b000 0x7ffec909b002 0x7ffec909b004
// 2つのインスタンス`u`と`v`は4096バイトのアライメントで配置されます。
// インスタンスのアドレスの最後は`000`になっています。
```

整列されていて C と互換性のあるレイアウトを取得するため、`repr(C)`と `repr(align(n))`とを組み合わせることができます。 `repr(align(n))`と `repr(packed)`とを組み合わせることはできません。`repr(packed)`はアライメントを 1 に設定するからです。 `repr(packed)`の型を`repr(align(n))`の型に含めることもできません。

型レイアウトに関するさらなる詳細は、Rust リファレンスの型レイアウトの章を参照して下さい。

#### 10.0.7. その他のリソース

- 本書内
  - Rust と少しの C
  - C と少しの Rust
- 組込み Rust よくある質問
- C プログラマのための Rust のポインタ
- 昔はポインタを使っていました。今は？

# 11. 相互運用性

翻訳が古くなっています Rust と C との相互運用性は、常に 2 つの言語間のデータ変換に依存しています。 そこで、2 つの専用モジュールが `stdlib` 内にあります。 `std::ffi` と呼ばれるものです。

未翻訳

未翻訳

| Rust の型 | 中間表現 | C の型 |
|---|---|---|
| `String` | `CString` | `char *` |
| `&str` | `CStr` | `const char *` |
| `()` | `c_void` | `void` |
| u32 or u64 | | |

, `c_uint`, `unsigned int`, tr(( en: [etc], de: [usw.], ja: [他], )), [...], [...], ) )

未翻訳

```
fn foo(num: u32) {
    let c_num: c_uint = num;
    let r_num: u32 = c_num;
}
```

### 11.0.1. 他のビルドシステムとの相互運用性

組込みプロジェクトに Rust を組み込むための共通の要件は、Cargo と make や cmake のような既存のビルドシステムとを組み合わせることです。

issue #61 でこれに関する事例とユースケースを集めています。

### 11.0.2. RTOS との相互運用性

Rust を FreeRTOS や ChibiOS といった RTOS に統合することは、まだ作業を進めている状態です。 特に、RTOS の関数を Rust から呼び出すことはトリッキーです。

未翻訳

issue #62 でこれに関する事例とユースケースを集めています。

## 11.1. Rust と少しの C

C または C++を Rust プロジェクトの内部で使うには、主に 2 つの対応をします。

- 公開されている C の API を、Rust で使えるようにラッピングする
- C または C++のコードを、Rust のコードと一緒にビルドする

C++は、Rust コンパイラがターゲットにできる安定した ABI を持っていないため、C または C++と Rust とを組み合わせるときは、`C` の ABI を使うのがお勧めです。

### 11.1.1. インタフェースの定義

Rust から C または C++のコードを使う前に、リンクされるコードにどのようなデータ型や関数シグネチャが存在するかを、(Rust に) 定義する必要があります。C または C++では、このデータを定義したヘッダファイル（`.h` または `.hpp`）をインクルードします。 Rust では、これらの定義を手動で変換するか、定義を自動生成するツールを使うか、のいずれかが必要です。

まずは、C/C++から Rust に、定義を手動で変換する方法を説明します。

#### 11.1.1.1. C の関数とデータ型のラッピング

通常、C または C++で書かれたライブラリは、公開インタフェースで使用する全ての型と関数を定義するヘッダファイルを提供します。 ヘッダファイルの例は次の通りです。

```
/* File: cool.h */
typedef struct CoolStruct {
    int x;
    int y;
} CoolStruct;

void cool_function(int i, char c, CoolStruct* cs);
```

Rust に変換すると、このインタフェースは次のようになります。

```
/* File: cool_bindings.rs */
#[repr(C)]
pub struct CoolStruct {
    pub x: cty::c_int,
    pub y: cty::c_int,
}

extern "C" {
    pub fn cool_function(
        i: cty::c_int,
        c: cty::c_char,
        cs: *mut CoolStruct
    );
}
```

各部分の説明をするため、この定義を 1 つずつ見ていきましょう。

```
#[repr(C)]
pub struct CoolStruct { ... }
```

デフォルトでは、Rust は `struct` に含まれるデータの順序やパディング、サイズを保証しません。C のコードとの互換性を保証するために、`#[repr(C)]`アトリビュートを使います。このアトリビュートにより、Rust コンパイラは、構造体のデータを C と同じルールで構成します。

```
pub x: cty::c_int,
pub y: cty::c_int,
```

C または C++が `int` や `char` を定義する方法は柔軟であるため、`cty` で定義されているプリミティブデータ型の使用をお勧めします。 cty は、C の型を Rust の型にマップします。

```
extern "C" { pub fn cool_function( ... ); }
```

このステートメントは、`cool_function` という名前の、C の ABI を使った関数シグネチャを定義します。関数本体の定義なしにシグネチャを定義することで、この関数の定義は別の場所で与えられるか、静的ライブラリから最終的なライブラリまたはバイナリにリンクされる必要があります。

```
    i: cty::c_int,
    c: cty::c_char,
    cs: *mut CoolStruct
```

上記のデータ型と同様に、C と互換の定義を使って、関数の引数のデータ型を定義します。わかりやすくするために、引数の名前は同じままにしてあります。

*mut CoolStruct という新しい型があります。C は、&mut CoolStruct のような Rust の参照という概念を持っていません。代わりに、生ポインタを使います。このポインタの参照外しは、unsafe です。また、このポインタは実際に null ポインタになる可能性があります。C または C++のコードとやり取りする時には、Rust が通常行う保証を確実にするように気をつける必要があります。

#### 11.1.1.2. インタフェースの自動生成

面倒であり間違いの元である手動のインタフェース生成ではなく、bindgen と呼ばれるインタフェース変換を自動で行ってくれるツールがあります。bindgen の使用手順は、bindgen ユーザマニュアルを参照して下さい。bindgen を使用するための一般的なプロセスは、次の通りです。

1. Rust で使いたいインタフェースやデータ型を定義している全ての C または C++ヘッダを集めます。
2. ステップ 1 で集めたヘッダファイルを #include "..."する bindings.h ファイルを書きます。
3. bindgen にコンパイルフラグとコードと共に、bindings.h を与えます。ヒント：#![no_std] 互換なコードを生成するために、Builder.ctypes_prefix("cty")と--ctypes-prefix=cty を使って下さい。
4. bindgen は、端末のウィンドウに生成した Rust コードを出力します。これは、bindings.rs のようなプロジェクトのファイルにパイプすることができます。外部ライブラリとしてコンパイル、リンクされた C/C++コードとやり取りするために、このファイルを Rust プロジェクトで使用できます。 翻訳が古くなっています

### 11.1.2. C/C++コードのビルド

Rust コンパイラは、C または C++のコード（または、C のインタフェースを提供する他の言語のコード）をコンパイルする方法を直接は知らないため、Rust でないコードは事前にコンパイルする必要があります。

組込みプロジェクトでは、C/C++のコードを（cool-library.a のような）静的なアーカイブにコンパイルすることを意味します。このアーカイブは、最終リンク時に、Rust のコードに組み込むことができます。

使おうとしているライブラリが、既に静的なアーカイブとして配布されている場合、そのコードを再度ビルドする必要はありません。提供されているインタフェースのヘッダファイルを、上述の方法で、単に変換するだけです。 そして、その静的なアーカイブをコンパイル/リンク時に組み込みます。

もしコードがソースファイルとして存在するのであれば、C/C++のコードを静的ライブラリとしてコンパイルする必要があります。（make や CMake のような）ビルドシステムを使うか、必要なコンパイルステップを cc クレートを使って移植するか、いずれかの方法を取れます。どちらの方法でも、build.rs スクリプトを使う必要があります。

#### 11.1.2.1. Rust の build.rs ビルドスクリプト

build.rs スクリプトは、Rust の構文で書かれたファイルです。 このスクリプトは、プロジェクトの依存をビルドした後、プロジェクトをビルドする前に、コンパイルを行っているコンピュータ上で実行されます。

完全なリファレンスはここにあります。 build.rs スクリプトは、（bindgen のようなツールを用いて）コードを生成したり、 Make のような外部ビルドシステムを呼び出したり、cc クレートを使って C/C++を直接ビルドしたりするのに便利です。

#### 11.1.2.2. 外部ビルドシステムの使用

複雑な外部プロジェクトや外部ビルドシステムを使うプロジェクトに関しては、`std::process::Command` を使って、他のビルドシステムに対して「シェルを呼び出す」のが最も簡単でしょう。これにより、相対パスを渡り歩いたり、(`make library` のような) 固定のコマンドを呼び出したり、その後、`target` ビルドディレクトリにある静的ライブラリをコピーしたりすることができます。

作っているクレートが `no_std` な組込みプラットフォームをターゲットにしていたとしても、`build.rs` はクレートをコンパイルしているコンピュータ上でのみ動作します。そのため、コンパイルしているホスト上で動作する、あらゆる Rust のクレートを使うことができます。

#### 11.1.2.3. C/C++コードを cc クレートでビルド

依存や複雑さが少ないプロジェクトや、(最終バイナリや実行ファイルではなく) 静的ライブラリを作成するためにビルドシステムを修正するのが難しいプロジェクトであれば、代わりに `cc` クレートを使う方が簡単でしょう。cc クレートは、ホスト上のコンパイラへの、扱いやすい Rust インタフェースを提供します。

単一の C ファイルを静的ライブラリにコンパイルする最も単純な例では、`cc` クレートを使った `build.rs` スクリプトは次のようになります。

```
fn main() {
    cc::Build::new()
        .file("src/foo.c")
        .compile("foo");
}
```

未翻訳

## 11.2. C と少しの Rust

C または C++のプロジェクト内で Rust のコードを使うためには、主に次の 2 つの対応をします。

- C が扱いやすい API を Rust に作ります
- 外部ビルドシステムに Rust プロジェクトを組み込みます

`cargo` と `meson` は別として、ほとんどのビルドシステムは Rust をサポートしていません。そのため、自分のクレートとその依存関係をコンパイルするには、`cargo` を使うのが一番です。

### 11.2.1. プロジェクトの準備

いつもどおり、新しい `cargo` プロジェクトを作成します。

通常の Rust のターゲットではなく、システムライブラリを出力するように、`cargo` に指示するフラグがあります。クレートの残り部分と異なる名前を付けたい場合、ライブラリに対して、別の名前を設定することもできます。

```
[lib]
name = "your_crate"
crate-type = ["cdylib"]      # 動的ライブラリを作ります
# crate-type = ["staticlib"] # 静的ライブラリを作ります
```

### 11.2.2. C API の作成

C++は、Rust コンパイラがターゲットにできる安定した ABI を持っていないため、別言語との相互運用には、C の ABI を使用します。C と C++のコード内で Rust を使うとき、このことに例外はありません。

#### 11.2.2.1. `#[no_mangle]`

Rust コンパイラは、シンボル名をネイティブコードリンカが期待するものとは異なるものにマングルします。 そのため、Rust の外にエクスポートする Rust の関数は、マングルしないようにコンパイラに指示する必要があります。

#### 11.2.2.2. `extern "C"`

デフォルトでは、Rust に書いたいずれの関数も Rust の ABI（これも安定化されていません）を使います。 代わりに、外に公開する FFI API はシステム ABI を使うように、コンパイラに指示する必要があります。

プラットフォームによっては、特定の ABI バージョンをターゲットにしたい場合があります。ABI については、ここにドキュメントがあります。

---

これらの部品をまとめると、おおよそ次のような関数になります。

```
#[no_mangle]
pub extern "C" fn rust_function() {

}
```

Rust プロジェクトで C コードを使った時と同様に、異なる言語間で理解できるデータ型に変換する必要があります。

### 11.2.3. リンクとより大きなプロジェクトの状況

ここまでで、問題の半分は解決しました。 これをどうやって使うのでしょうか？

**それは、プロジェクトやビルドシステムに強く依存します。**

`cargo` は、プラットフォームと設定に依存して、`my_lib.so`、`my_lib.dll` または `my_lib.a` ファイルを作成します。このライブラリは、そのプラットフォームのビルドシステムによって簡単にリンクすることができます。

しかし、C から Rust を呼ぶためには、関数シグネチャを宣言するためのヘッダファイルが必要です。

Rust-ffi API の関数全てが、対応する関数ヘッダを持つ必要があります。

```
#[no_mangle]
pub extern "C" fn rust_function() {}
```

上記は、次のようになるでしょう。

```
void rust_function();
```

などなど。

このプロセスを自動化する cbindgen というツールがあります。 このツールは、Rust のコードを解析して、C と C++プロジェクトのためのヘッダを生成します。

この時点で、C から Rust の関数を使うには、単にヘッダをインクルードして、それを呼び出すだけです！

```
#include "my-rust-project.h"
rust_function();
```

# 12. 未分類のトピック

## 12.1. 最適化：速度とサイズのトレードオフ

誰もがプログラムを超高速で超小さくしたいと願いますが、両方の特性を最大化することはできません。このセクションは、`rustc` が提供する異なる最適化レベルについて、どのようにプログラムの実行時間とバイナリサイズに影響するかを説明します。

### 12.1.1. 最適化なし

これはデフォルトです。`cargo build` を実行する場合、開発（別名 `dev`）プロファイルを使います。このプロファイルはデバッグに最適化されています。そのため、デバッグ情報が有効化されており、最適化は一切有効化されていません。 つまり、`-C opt-level = 0` を使用します。

少なくともベアメタルの開発では、デバッグ情報はある意味ゼロコストです。 デバッグ情報は、Flash/ROM の容量を使いません。そのため、リリースプロファイルで、デフォルトで無効化されているデバッグ情報を有効化することをお勧めします。これにより、リリースビルドをデバッグする時、ブレイクポイントを使うことができます。

```
[profile.release]
# シンボルは素晴らしく、Flashのサイズを増やしません
debug = true
```

最適化しないことはデバッグでは重要です。コードをステップ実行する時、プログラムをステートメントごとに実行しているように感じられるからです。さらに、スタックの変数や関数の引数を GDB で `print` することができます。 コードが最適化されると、変数を表示しようとしても、`$0 = <value optimized out>`と表示されます。

`dev` プロファイルの最大の欠点は、バイナリが大きく、遅いことです。 バイナリサイズは通常、より問題となります。最適化されていないバイナリは数十 KiB も Flash を専有するからです。 ターゲットデバイスは、数十 KiB もの Flash を持っていないかもしれず、最適化されていないバイナリは、デバイス内に納まりません。

デバッガで扱いやすい、小さなバイナリを作ることができるのでしょうか？できます。良いやり方があります。

#### 12.1.1.1. 依存関係の最適化

nightly では、`profile-overrides` と呼ばれる Cargo フィーチャがあります。これは、依存関係の最適化レベルをオーバーライドします。このフィーチャを使って、全ての依存クレートのサイズを最適化しながら、トップクレートを最適化しないでデバッガで扱いやすくすることができます。。

未翻訳

例を示します。

```
# Cargo.toml
[package]
name = "app"
# ..

[profile.dev.package."*"] # +
opt-level = "z" # +
```

オーバーライドなしでは、次の通りです。

```
$ cargo size --bin app -- -A
app  :
section               size        addr
.vector_table         1024   0x8000000
.text                 9060   0x8000400
.rodata               1708   0x8002780
.data                    0  0x20000000
.bss                     4  0x20000000
```

オーバーライドをすると、以下のようになります。

```
$ cargo size --bin app -- -A
app  :
section               size        addr
.vector_table         1024   0x8000000
.text                 3490   0x8000400
.rodata               1100   0x80011c0
.data                    0  0x20000000
.bss                     4  0x20000000
```

トップクレートのデバッグ性を失うことなしに、Flash 使用量を 6KiB 減らしています。依存クレートの中に足を踏み入れると、`<value optimized out>`のメッセージを目にするでしょう。しかし、依存クレートではなく、トップクレートをデバッグしたい場合がほとんどでしょう。依存クレートをデバッグする必要がある場合、特定の依存クレートを最適化から除外するために、`profile-overrides` フィーチャを使えます。 例えば、以下のようになります。

```
# ..

# `cortex-m-rt`クレートは最適化しません
[profile.dev.package.cortex-m-rt] # +
opt-level = 0 # +

# しかし、他の依存クレートは最適化します
[profile.dev.package."*"]
codegen-units = 1 # 未翻訳
opt-level = "z"
```

これで、トップクレートと `cortex-m-rt` クレートはデバッガで扱いやすくなります！

#### 12.1.2. 速度の最適化

2018-09-18 の `rustc` は、3 つの「速度最適化」を提供しています。`opt-level = 1`、`2` と `3` です。`cargo build --release` を実行した時、デフォルトでは `opt-level = 3` のリリースプロファイルを使います。

`opt-level = 2` と `3` は、バイナリサイズを犠牲にして、速度を最適化します。レベル 3 はレベル 2 より、ベクトル化とインライン化を行います。 特に、`opt-level` が 2 以上の場合、LLVM がループを展開するのがわかるでしょう。ループ展開は、Flash/ROM の観点からはよりコストが高いです（例えば、配列のループをゼロにする場合、26 バイトから 194 バイトまで増加します）。 しかし、適切な条件では、実行時間が半分になります（例えば、イテレーションの回数が十分大きい場合）。

現在、`opt-level = 2` と `3` でループ展開を無効化する方法はありません。 ループ展開のコストを払うことができない場合、プログラムサイズの最適化をするべきです。

### 12.1.3. サイズの最適化

2018-09-18 の rustc は、2 つのサイズ最適化を提供しています。opt-level = "s"と"z"です。 これらの名前は、clang / LLVM から受け継いでおり、意味がわかりにくいです。 "z"は、"s"より小さなバイナリを作る意図を意味します。

リリースバイナリのサイズを最適化したい場合、Cargo.toml の profile.release.opt-level 設定を下記の通り変更します。

```
[profile.release]
# または"z"
opt-level = "s"
```

これらの 2 つの最適化レベルは、LLVM のインラインしきい値を大幅に減らします。インラインしきい値は、関数をインライン化するか否かを決めるために使われる基準値です。 Rust の原則の 1 つは、ゼロコスト抽象化です。これらの抽象化は、不変条件を保持するため、多くの新しい型と小さな関数を使う傾向にあります（例えば、deref や as_ref のような内部の値を借用するための関数）。そのため、インラインしきい値を低くすると、LLVM が最適化の機会を失います（例えば、不要な分岐を削除したり、クロージャをインライン呼び出しにする、など）。

サイズの最適化を行っている時、バイナリサイズに影響があるかどうかを見るために、インラインしきい値を増やしたいかもしれません。 インラインしきい値を変更するお勧めの方法は、 .cargo/config 内の rustflags に-C inline-threshold フラグを追加することです。

```
# .cargo/config.toml
# cortex-m-quickstartテンプレートを使っていることを想定しています
[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = [
  # ..
  "-C", "inline-threshold=123", # +
]
```

この値は何に使われるのでしょうか？1.29.0 では、下記の値は、異なる最適化レベルで使われるインラインしきい値です

- opt-level = 3 は 275 を使います
- opt-level = 2 は 225 を使います
- opt-level = "s"は 75 を使います
- opt-level = "z"は 25 を使います

サイズの最適化をするときは、225 と 275 を試してみるべきです。

## 12.2. 未翻訳

未翻訳

```
//! 未翻訳

fn main() {
    let float: f32 = 4.82832;
    let floored_float = float.floor();

    let sqrt_of_four = floored_float.sqrt();

    let sinus_of_four = floored_float.sin();

    let exponential_of_four = floored_float.exp();
    println!("Floored test float {} to {}", float, floored_float);
```

```
    println!("The square root of {} is {}", floored_float, sqrt_of_four);
    println!("The sinus of four is {}", sinus_of_four);
    println!(
        "The exponential of four to the base e is {}",
        exponential_of_four
    )
}
```

未翻訳

```
#![no_main]
#![no_std]

use panic_halt as _;

use cortex_m_rt::entry;
use cortex_m_semihosting::{debug, hprintln};
use libm::{exp, floorf, sin, sqrtf};

#[entry]
fn main() -> ! {
    let float = 4.82832;
    let floored_float = floorf(float);

    let sqrt_of_four = sqrtf(floored_float);

    let sinus_of_four = sin(floored_float.into());

    let exponential_of_four = exp(floored_float.into());
    hprintln!("Floored test float {} to {}", float, floored_float).unwrap();
    hprintln!("The square root of {} is {}", floored_float, sqrt_of_four).unwrap();
    hprintln!("The sinus of four is {}", sinus_of_four).unwrap();
    hprintln!(
        "The exponential of four to the base e is {}",
        exponential_of_four
    )
    .unwrap();
    // 未翻訳
    // debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

未翻訳

- CMSIS DSP library binding
- constgebra
- micromath
- microfft
- nalgebra

## A. 未翻訳

未翻訳

**BSP**

未翻訳

**FPU**

未翻訳

**HAL**

未翻訳

**I2C**

未翻訳

**PAC**

未翻訳

**SPI**

未翻訳

**SVD**

未翻訳

**UART**

未翻訳

**USART**

未翻訳