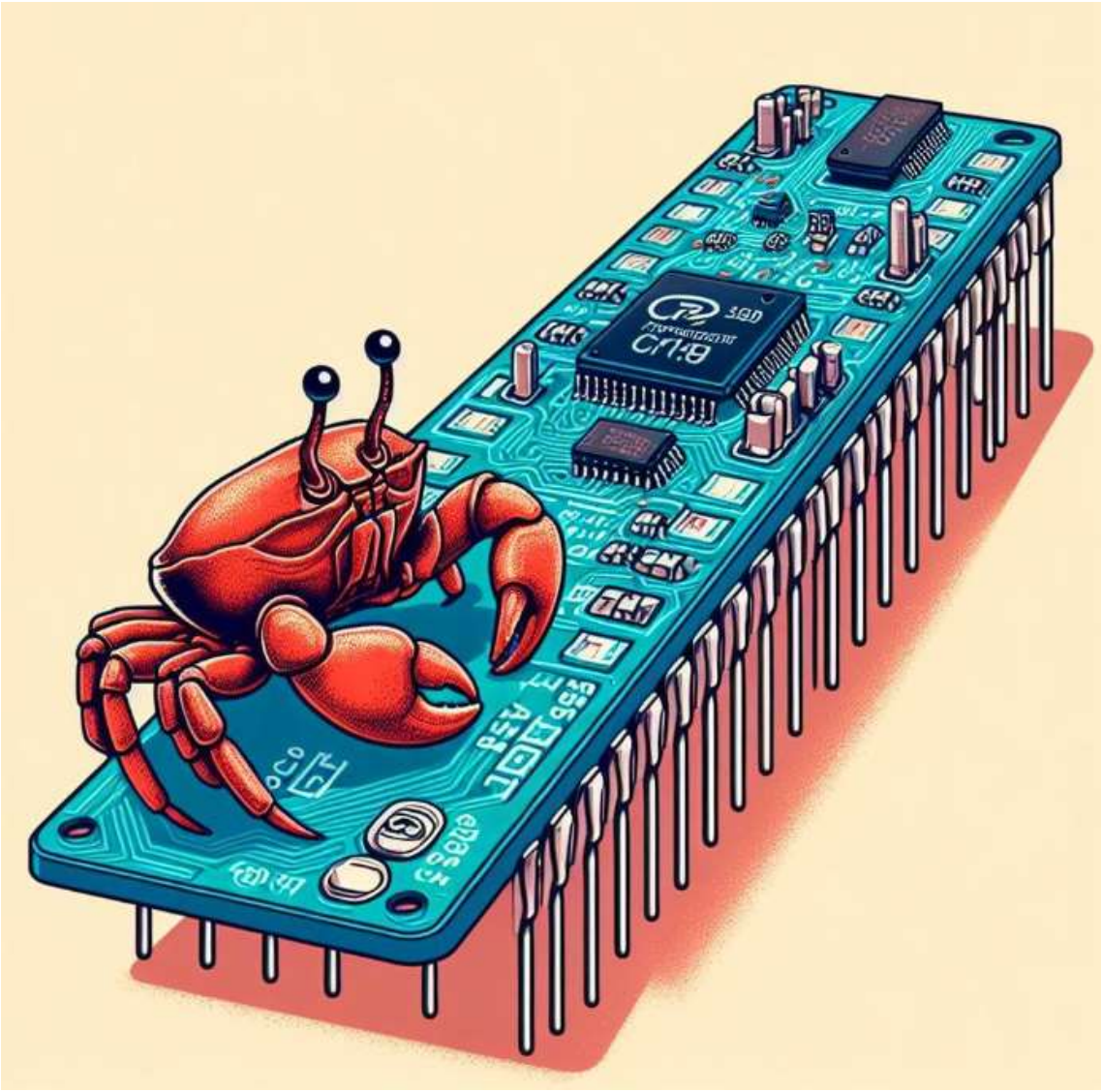
Raspberry Pi Pico で Rust の開発環境を構築する



moons3925 著

2024 年 2 月

まえがき

組み込みソフトウェアの言語は主に C/C++が使われてきましたが最近 Rust という言語が注目されています。Rust はコンパイル型の言語であり C/C++に近い実行速度を出すことや C/C++に近いコード効率を出すことができる他、モダンな言語でかつメモリーを安全に操作できる等の特徴があります。

ここ数年で組み込み業界にも Rust を推す動きが活発になってきているので私も開発環境を構築して Rust を学んでいくことにしました。

Rust は学習コストが高い言語と言われていますが、オープンソースであることや開発ツールを上手に扱いながらコードを書いていくことで次第にコードに慣れてきます。

本書は Visual Studio Code を拡張した開発環境の構築方法について説明しています。そのデバッグ機能により関数（メソッド）の内部まで入っていくことができるため、組み込み Rust を効率的に学習できるでしょう。

UART を使ったサンプルプログラムをひとつ掲載し、それについて解説しています。解説と言っても主に私の感想を書いています。

本書が皆さんの組み込み Rust の学習の一助になれば幸いです。

引用

本書の制作及び Rust を学ぶにあたり私が参考にさせて頂いている書籍です。

上の書籍は組み込み Rust 初心者にはお薦めの一冊です。
ただ Rust の細かい文法までは網羅できないため、私は下の書籍も併用しています。

中林智之・井田健太　著
「基礎から学ぶ組込み Rust」（株式会社シーアンドアール研究所、2021 年初版）

Steve Klabnik・Carol Nichols　著　尾崎亮太　翻訳
「プログラミング言語 Rust 公式ガイド」（株式会社ドワンゴ、2019 年初版）

目次

１．Raspberry Pi Pico とは

Raspberry Pi Pico は Raspberry Pi 財団が独自に開発した Raspberry Pi シリーズでは初めての「マイコン」と呼ばれる部類のデバイスを実装したボードです。最高 133MHz のクロックで動作する ARM Cortex M0+デュアルコアの RP2040 というマイクロコントローラーを搭載しています。

元々は C/C++及び MicroPython での開発が可能であるというところからの出発でしたが、今では Rust による開発も可能となっています。RP2040 はデュアルコアで RTOS を使うことが可能で USB の機能もマイコン内部に持っています。それから PIO（プログラマブル IO）という I/O の操作に特化したコアも持っています。もちろん、一般的なペリフェラルである UART, SPI, I2C, PWM, RTC, ADC, GPIO などの機能も備わっています。

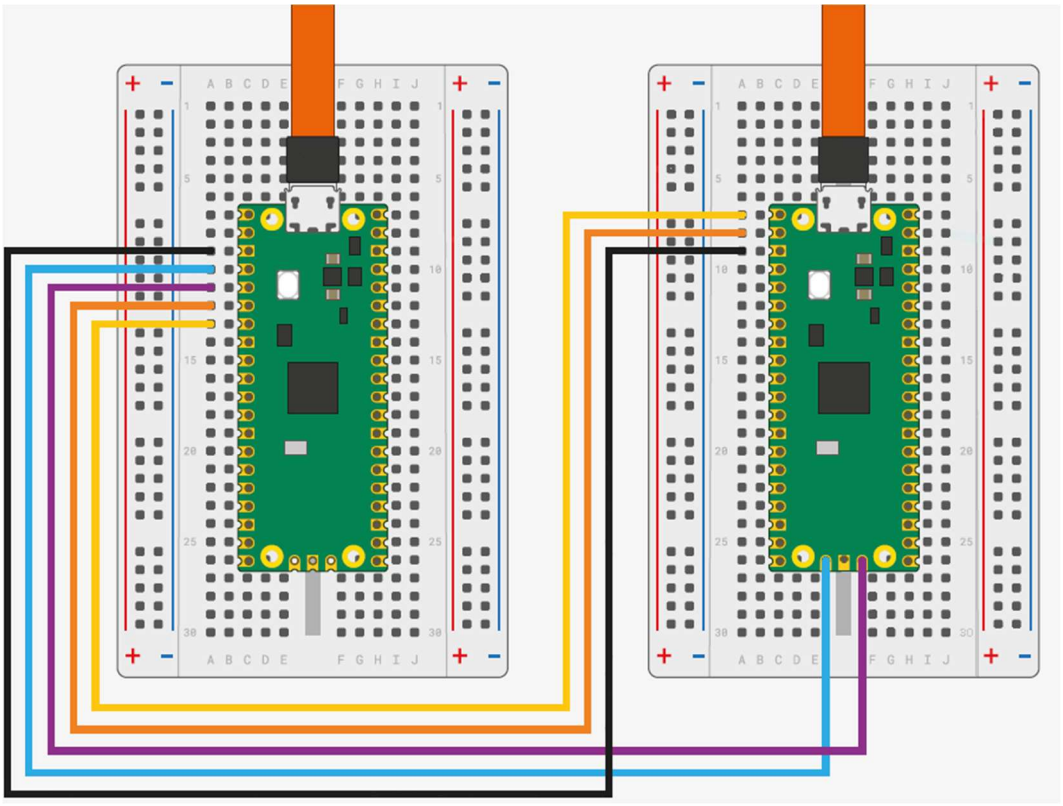
メモリーは SRAM（264KB）を内蔵しています。フラッシュ ROM は RP2040 から見ると外付けになり、Raspberry Pi Pico では 2MByte の部品が実装されています。それから Raspberry Pi Pico は Linux が動くボードではないので、その点はご注意ください。

2．必要なハードウェア

環境の構築には以下のハードウェアが必要です。

(1)開発用コンピューター： Windows10 OS のパソコン（1 台）
(2)USB ケーブル： USB microB – A のケーブル（2 本）
(3)開発用・デバッグ用のボード Raspberry Pi Pico（2 枚）
(4)その他（ブレッドボード・ジャンパーワイヤー等の線材）
(5)インターネットの接続環境

以下に配線のイメージ図を掲載しておきます。
オレンジ色のケーブルが (2)
緑色のプリント基板が(3)
それ以外が(4)



（注）パソコンは以降 PC と呼びます。

３．ソフトウェアのインストール

インターネットが接続できる環境が必要になります。

３．１．リンカー

Rust ではツール内にリンカーが含まれていませんので別途用意する必要があります。
リンカーをお持ちでない方は以下から Visual Studio Community をダウンロードして使うなどの方法があります。

https://visualstudio.microsoft.com/ja/free-developer-offers/

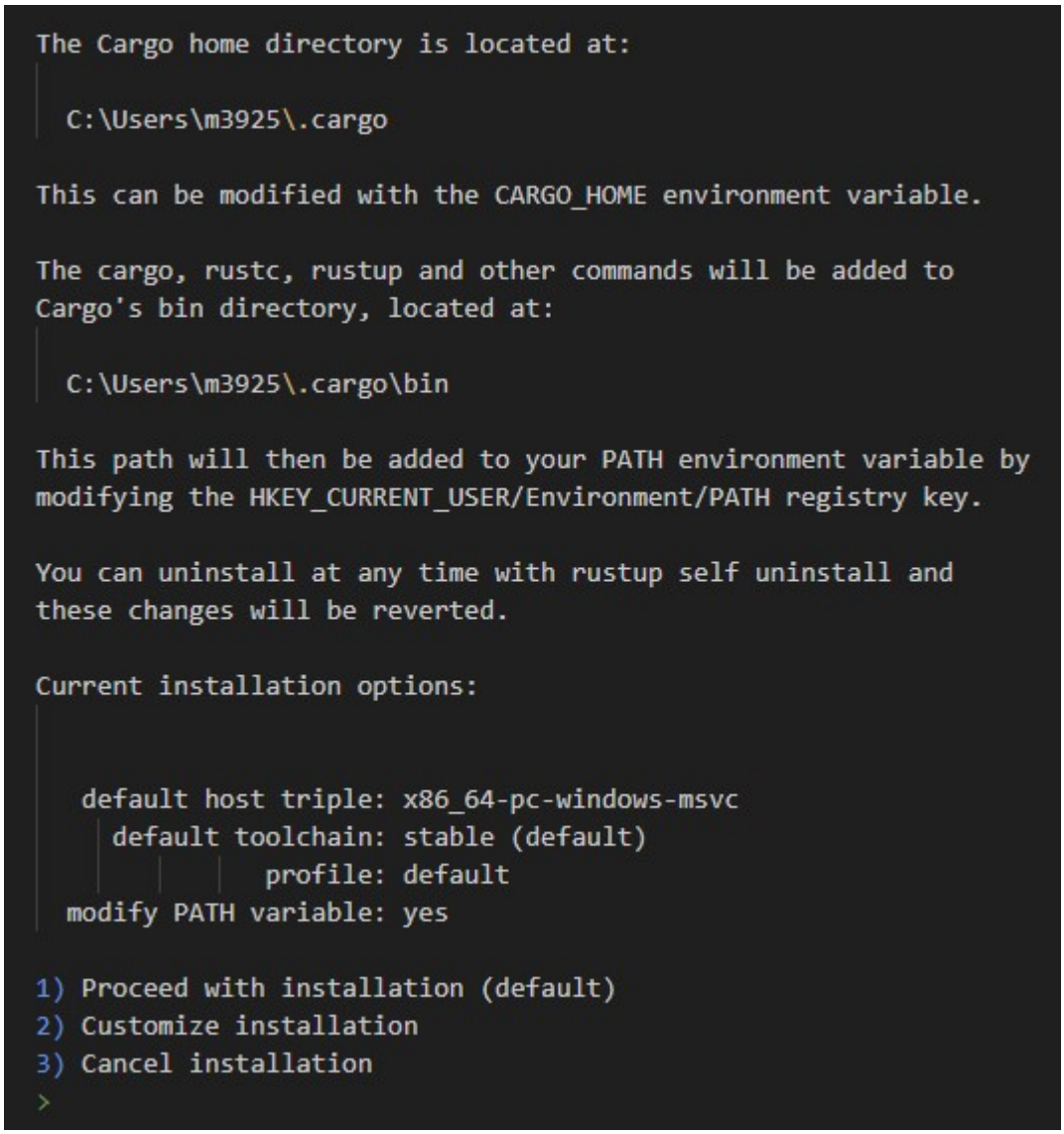
３．２．Rust ツールチェイン

rustup.rs からインストーラーをダウンロードします。
rustup-init.exe をクリックしてダウンロードした後、exe ファイルを実行します。

https://rustup.rs/

exe ファイルを実行すると次ページにあるコマンドプロンプトが起動するので、>に続いて 1（半角数字）を入力し Enter キーを押します。

C:¥Users¥に続く m3925 はユーザー名です。皆さんのものに置きかえてください。



>1 に続いてインストールの進捗が表示されます。(次ページ)

```
>1

info: profile set to 'default'
info: default host triple is x86_64-pc-windows-msvc
info: syncing channel updates for 'stable-x86_64-pc-windows-msvc'
info: latest update on 2023-11-16, rust version 1.74.0 (79e9716c9 2023-11-13)
info: downloading component 'cargo'
  5.8 MiB /   5.8 MiB (100 %)   5.8 MiB/s in  1s ETA:  0s
info: downloading component 'clippy'
info: downloading component 'rust-docs'
 14.4 MiB /  14.4 MiB (100 %)   6.0 MiB/s in  2s ETA:  0s
info: downloading component 'rust-std'
 19.7 MiB /  19.7 MiB (100 %)   6.1 MiB/s in  3s ETA:  0s
info: downloading component 'rustc'
 58.4 MiB /  58.4 MiB (100 %)   5.9 MiB/s in 10s ETA:  0s
info: downloading component 'rustfmt'
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
 14.4 MiB /  14.4 MiB (100 %)   1.0 MiB/s in 16s ETA:  0s
info: installing component 'rust-std'
 19.7 MiB /  19.7 MiB (100 %)  10.6 MiB/s in  1s ETA:  0s
info: installing component 'rustc'
 58.4 MiB /  58.4 MiB (100 %)  11.6 MiB/s in  5s ETA:  0s
info: installing component 'rustfmt'
info: default toolchain set to 'stable-x86_64-pc-windows-msvc'

  stable-x86_64-pc-windows-msvc installed - rustc 1.74.0 (79e9716c9 2023-11-13)


Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload its PATH environment variable to include
Cargo's bin directory (%USERPROFILE%\.cargo\bin).

Press the Enter key to continue.
```

インストールが終わると Press the Enter key to continu. と表示されるので Enter キーを押します。

Enter キーを押すとウィンドウが閉じます。
PC を再起動した後に、コマンドプロンプトを起動します。

３．２．１．ツールのバージョンを確認する

以下のコマンド３つを入力しバージョンを確認します。

```
>rustup --version
```

（version の前のハイフンは 2 つ必要です）

```
>rustc --version
```

（version の前のハイフンは 2 つ必要です）

```
>cargo --version
```

（version の前のハイフンは 2 つ必要です）

をそれぞれ実行します。

実行結果を以下に掲載しておきます。

```
C:\Users\m3925>rustup --version
rustup 1.26.0 (5af9b9484 2023-04-05)
info: This is the version for the rustup toolchain manager, not the rustc compiler.
info: The currently active `rustc` version is `rustc 1.72.0 (5680fa18f 2023-08-23)`

C:\Users\m3925>rustc --version
rustc 1.72.0 (5680fa18f 2023-08-23)

C:\Users\m3925>cargo --version
cargo 1.72.0 (103a7ff2e 2023-08-15)

C:\Users\m3925>
```

3．2．2．ツールの更新

以下のコマンドで rustup を更新することができます。

```
>rustup update
```

3．2．3．アンインストール

以下のコマンドで rustup をアンインストールすることができます。

```
>rustup self uninstall
```

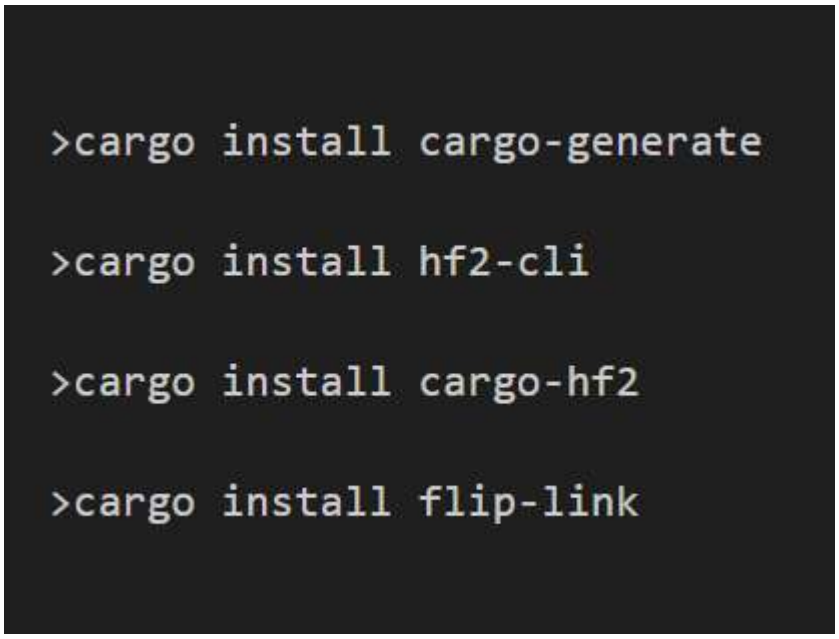
3．2．4．クロスビルドチェインのインストール

以下のコマンドを実行しターゲットとして Raspberry Pi Pico に搭載されているマイコンもビルドできるようにします。

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

3．2．5．Cargo サブコマンドのインストール

以下のコマンド 4 つを実行して Cargo サブコマンドをインストールします。
（使わないツールがあるかも知れませんが念のために全てインストールしておきます）



3．3．Visual Studio Code のインストール

以下のサイトから Visual Studio Code (以降 VSCode と称す)をインストールします。
VSCode はエディタや IDE などの機能が無償で使える優れたツールです。

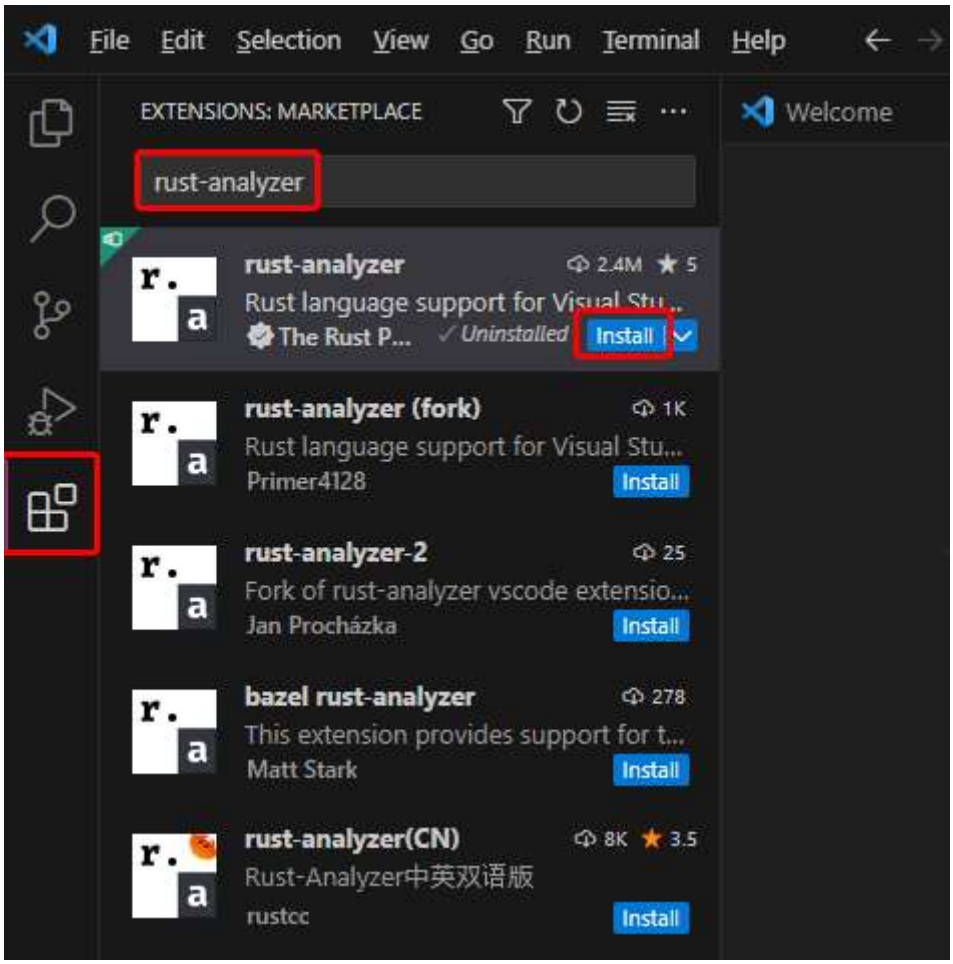
https://code.visualstudio.com/Download

３．３．１．rust-analyzer のインストール

rust-analyzer は入力補完や型の表示など便利な機能を提供してくれます。

VS Code の拡張機能から rust-analyzer をインストールします。

VS Code を起動し、左の赤枠のアイコンをクリックして拡張機能マーケットプレイスを表示させて、枠内に rust-analyzer と入力し青い install ボタンを押します。

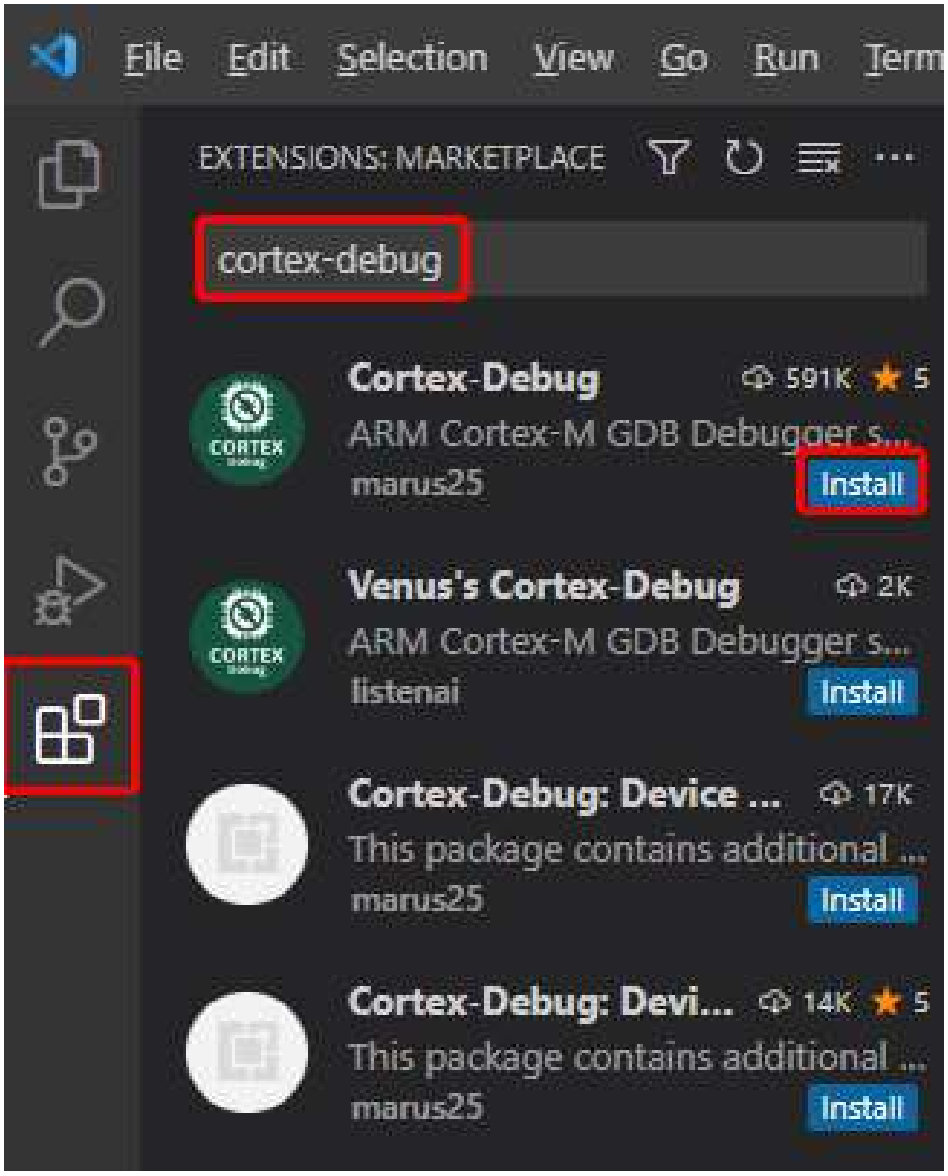


３．３．２．Cortex-Debug のインストール

Cortex-Debug はデバッグに必要なツールです。

VS Code の拡張機能から Cortex-Debug をインストールします。

VS Code を起動し、左の赤枠のアイコンをクリックして拡張機能マーケットプレイスを表示させて、枠内に cortex-debug と入力し青い install ボタンを押します。



３．４．Tera Term のインストール

Tera Term は UART の動作確認用に使えるターミナルソフトです。
本家のダウンロードサイトが調子悪いようなので、以下等からダウンロードして使います。

https://github.com/TeraTermProject/osdn-download/releases/tag/teraterm-5.0

３．５．OpenOCD のインストール

OpenOCD はデバッグ用のツールです。
以下から openocd-win.zip をダウンロードしてインストールします。
インストール先のフォルダをメモしておきます。
（後で設定ファイルにインストール先のパスを指定する必要があります）

C:¥から ... openocd.exe までのフルパスを指定します。

https://github.com/ciniml/debug-tools-builder/releases

３．６．GDB のインストール

GDB はデバッグ用のツールです。
以下から gcc-arm-none-eabi-10.3-2021.10-win32.zip をダウンロードしてインストールします。
インストール先のフォルダをメモしておきます。
（後で設定ファイルにインストール先のパスを指定する必要があります）

C:¥から ... arm-none-eabi-gdb.exe までのフルパスを指定します。

https://developer.arm.com/downloads/-/gnu-rm

３．７．git for windows のインストール

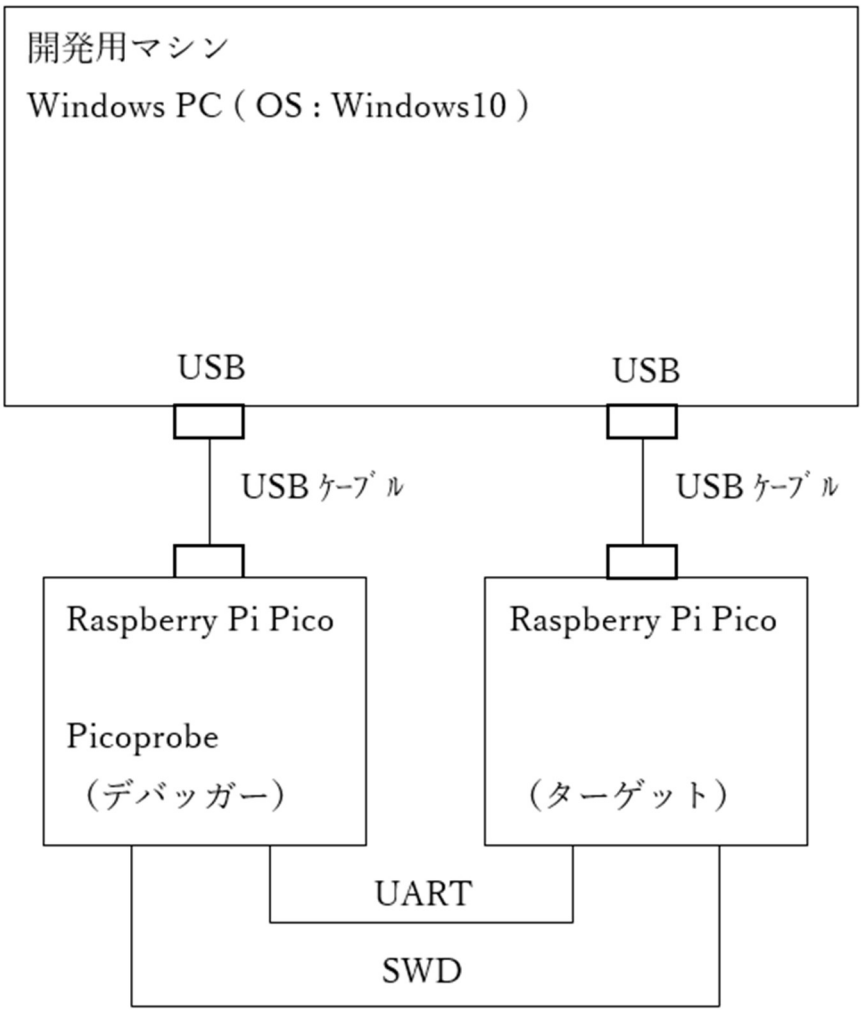
バージョン管理用のツールです。
以下からダウンロードします。

https://gitforwindows.org/

４．接続

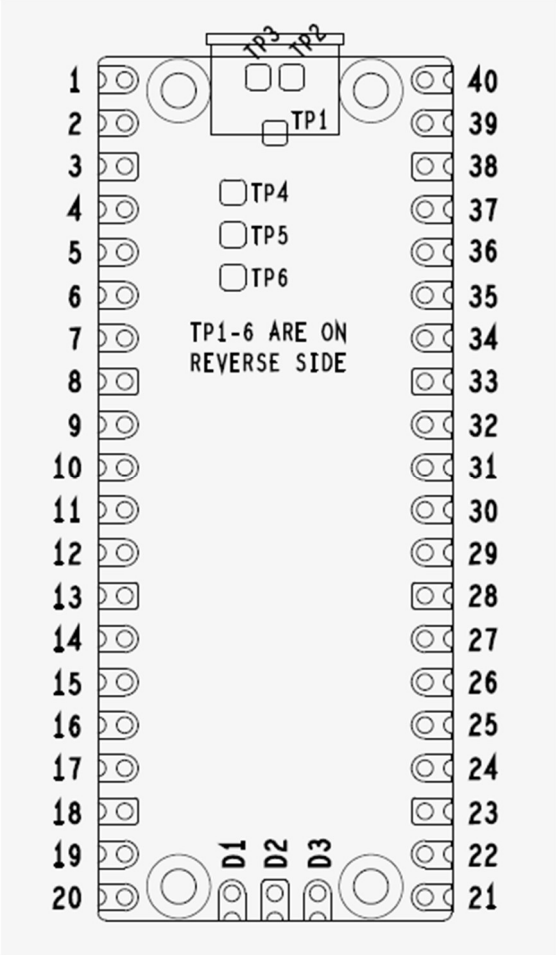
４．１．系統図

以下に Raspberry Pi Pico を使ったデバッグシステムの系統図を示します。
デバッグ用の信号は SWD です。GND を除けば信号線は２本なので接続が簡単です。
Raspberry Pi Pico２枚のうちの１枚は専用のファームウェアを書き込むことでデバッガーとして使うことができます。このデバッガーを PicoProbe と呼んでいます。
UART の信号はデバッグ用のインターフェースとしては不要ですが、UART の通信を Picoprobe を介して PC で確認することができるので図のように接続しておきます。
Picoprobe が USB-シリアル通信機能の役割を担ってくれます。



４．２．ボードのピン配置

ボードのピン配置は以下を参考にしてください。



４．３．接続表

信号の接続は以下の通りです。

色：次の姿図の配線の色を示しています。(次ページを参照)

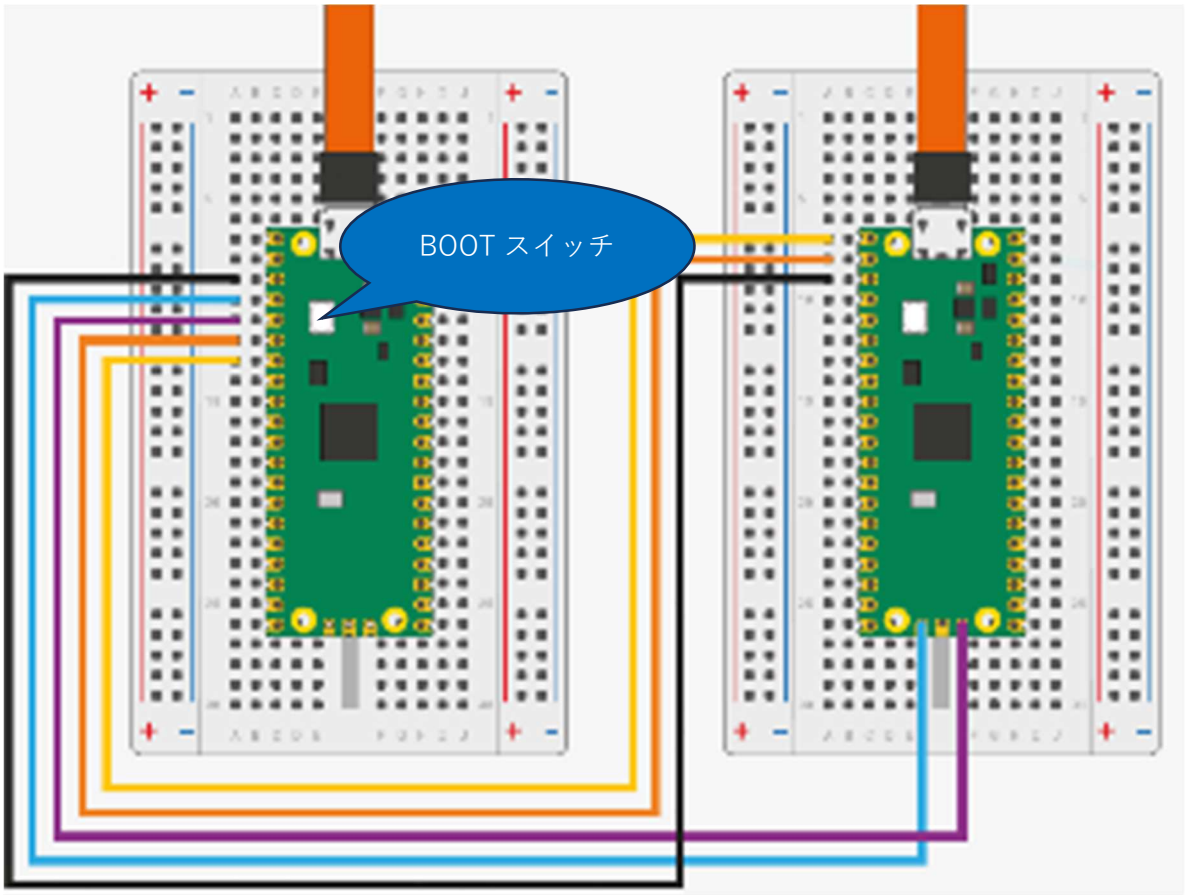
信号	Picoprobe	Raspberry Pi Pico	色
GND	3	3	黒色
SWCLK	4	D1	水色
SWDIO	5	D3	紫色
UART0 RX	6	2	橙色
UART0 TX	7	1	黄色

４．４．接続の姿図

接続した様子を以下に示しますので接続の際の参考にしてください。

左がデバッガーの Picoprobe、右がターゲットの Raspberry Pi Pico です。

２つの USB ケーブルは、まだ PC と接続しないでください。



４．５．Picoprobe にファームウェアを書き込む

以下のサイトで picoprobe.uf2 をクリックしてダウンロードしておきます。

https://github.com/raspberrypi/picoprobe/releases

次に上図の BOOT スイッチを押しながら左側のＵＳＢケーブルをＰＣにつないで給電します。ケーブルをつないで少し待ち、スイッチから手を離します。（３秒待てば充分）

４．６．Picoprobe.uf2 をドラッグ＆ドロップする

PC 上のエクスプローラーで見ると以下「RPI-RP2(E:)」のようにボードがドライブのように見えるので、先ほどダウンロードした Picoprobe.uf2 をドラッグし、以下のドライブにドロップしてコピーします。

これで Picoprobe.uf2 の書き込みは完了で、Picoprobe をデバッガーとして使うことができるようになります。

一度ＵＳＢケーブルをＰＣから抜き、３秒程度待ちます。
その後、両方のＵＳＢケーブルをＰＣにつなぎます。

（注）接続に問題がないことを充分確認しておいてください。

５．プロジェクトの構築

ゼロからプロジェクトを構築するのは大変ですから、こちらでサンプルプロジェクトを用意しました。GitHub に置きましたので、複製のプロジェクトを構築してください。

以下に Rust のクレート置き場があります。どうしてもゼロからプロジェクトを構築してみたいという方は以下にある Raspberry Pi Pico のクレートに書かれている情報を頼りにチャレンジしてみてください。ただし非常に手間がかかります。

Raspberry Pi Pico のクレート
https://crates.io/crates/rp-pico

サンプルプロジェクトの置き場所(GitHub)
https://github.com/moons3925/pico

５．１．プロジェクトをつくる

５．１．１．Rust 用のルートをつくる

以下のコマンドを入力しユーザー名の下に rp_pico というディレクトリをつくります。（ m3925 は皆さんのユーザー名に置きかえてください ）

```
>C:\Users\m3925\md rp_pico
```

ディレクトリを rp_pico に移動します。

```
>C:\Users\m3925\cd rp_pico
```

5．1．2．プロジェクトの複製をつくる

こちらで用意したサンプルプロジェクトを git clone して複製をつくります。

コマンドは「 git clone https://github.com/moons3925/pico.git 」です。

（ m3925 は皆さんのユーザー名に置きかえてください ）

```
>C:\Users\m3925\rp_pico\git clone https://github.com/moons3925/pico.git
```

5．2．VSCode を起動する

cd pico でディレクトリを移動し、

code . で VSCode を起動します。

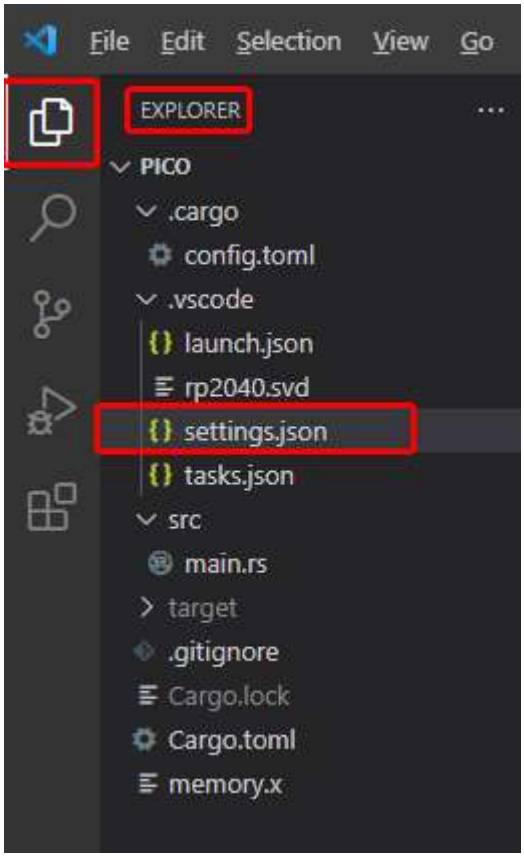
```
>C:\Users\m3925\rp_pico\cd pico

>C:\Users\m3925\rp_pico\pico>code .
```

５．２．１．setting.json を編集する

インストールした openocd と GDB のパスを setting.json で指定します。

左の赤枠のアイコンをクリックして EXPLORER ツリーを表示させて setting.json を選択しファイルの内容を表示します。



上の cortex-debug.openocdPath にメモしておいた openocd のパスを指定し、

下の cortex-debug.gdbPath にメモしておいた GDB のパスを指定し ctrl＋s で保存します。

```
{
    "cortex-debug.openocdPath": "C:/DevTools/openocd-win/bin/openocd.exe",
    "cortex-debug.gdbPath": "C:/Program Files (x86)/GNU Arm Embedded Toolchain/10 2021.10/bin/arm-none-eabi-gdb.exe",
    "rust-analyzer.checkOnSave.allTargets": false,
    "rust-analyzer.checkOnSave.extraArgs": [
        "--target",
        "thumbv7em-none-eabihf"
    ]
}
```

C:¥から … openocd.exe までのフルパスを指定します。

C:¥から … arm-none-eabi-gdb.exe　までのフルパスを指定します。

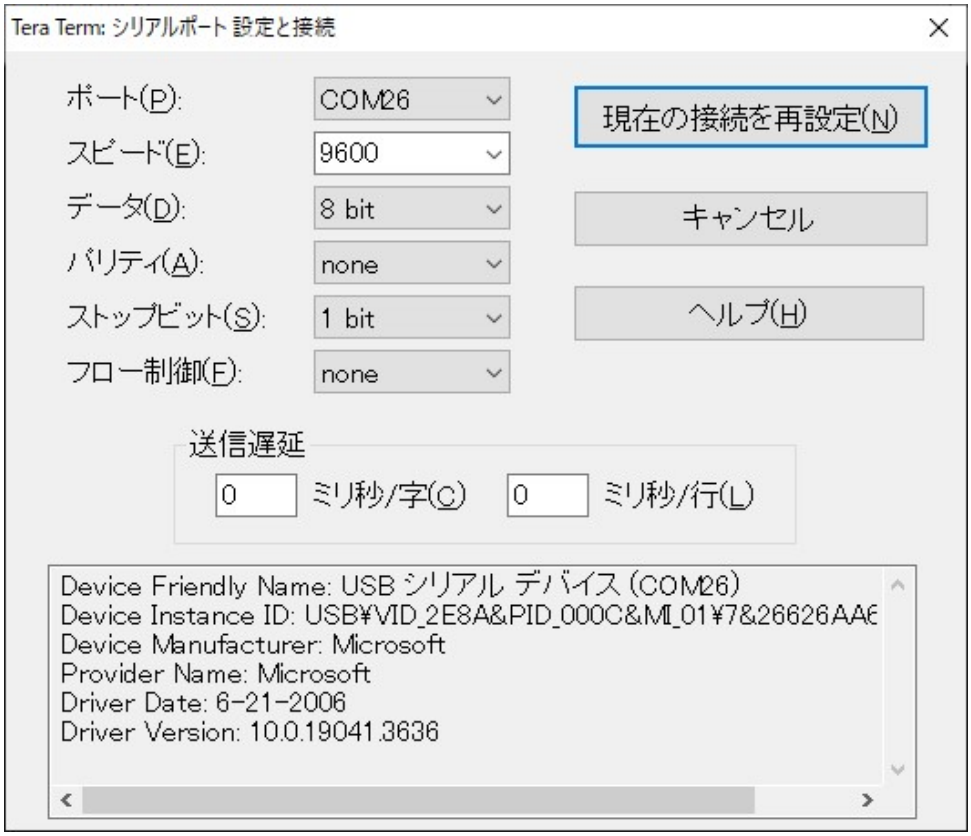
6．デバッグ

6．1．PC 側の準備

インストールしておいた Tera Term を起動します。
以下の設定画面が出るのでラジオボタンでシリアルを選択し、ポートで USB シリアルデバイスの COM ポートを選択します。

もしポートのコンボボックス内に複数の COM ポートがある場合には、デバイスマネージャーのポート(COM と LPT)を見ながら、Picoprobe の USB ケーブルを抜き差ししてみます。その時に見え隠れする COM ポート番号を指定します。

次に 設定 ― シリアルポートを選択し、ポート番号と通信メーターの設定を行います。
ポートは先ほどの COM 番号を指定します。
それ以外にはスピードを 9600 に指定し、現在の接続を再設定ボタンを押します。



６．２．デバッグを開始する

メニューから Run – Start Debugging　F5 を選択します。
起動に少し時間がかかりますが、しばらくすると#[entry]のところで停止します。もしうまく動かないようであれば接続を含むこれまでの環境構築を見直してみてください。
（長くても１分以内には起動します）

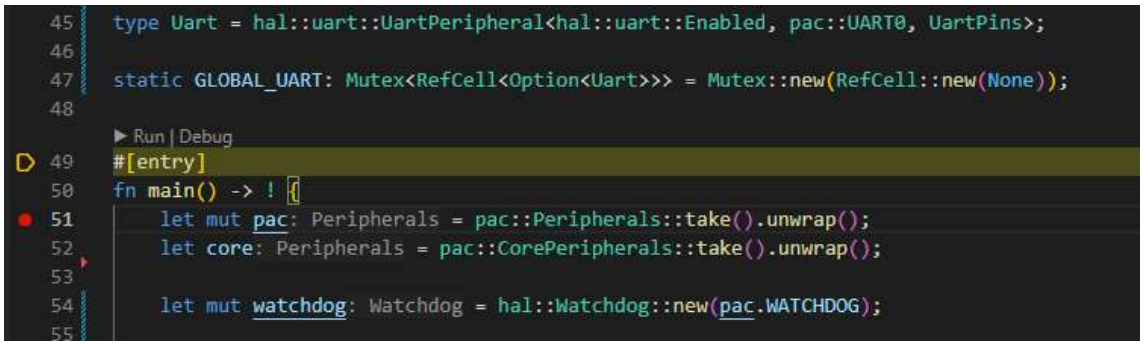
起動すると以下のように #[entry]　のところで停止します。

```
40  type UartPins = (
41      hal::gpio::Pin<Gpio0, hal::gpio::FunctionUart, hal::gpio::PullNone>,
42      hal::gpio::Pin<Gpio1, hal::gpio::FunctionUart, hal::gpio::PullNone>,
43  );
44
45  type Uart = hal::uart::UartPeripheral<hal::uart::Enabled, pac::UART0, UartPins>;
46
47  static GLOBAL_UART: Mutex<RefCell<Option<Uart>>> = Mutex::new(RefCell::new(None));
48
    ▶ Run | Debug
49  #[entry]
50  fn main() -> ! {
51      let mut pac: Peripherals = pac::Peripherals::take().unwrap();
52      let core: Peripherals = pac::CorePeripherals::take().unwrap();
53
54      let mut watchdog: Watchdog = hal::Watchdog::new(pac.WATCHDOG);
55
56      let clocks: ClocksManager = hal::clocks::init_clocks_and_plls(
57          rp_pico::XOSC_CRYSTAL_FREQ,
58          xosc_dev: pac.XOSC,
59          clocks_dev: pac.CLOCKS,
60          pll_sys_dev: pac.PLL_SYS,
61          pll_usb_dev: pac.PLL_USB,
62          &mut pac.RESETS,
63          &mut watchdog,
64      ) Result<ClocksManager, InitError>
65      .ok() Option<ClocksManager>
66      .unwrap();
```

（次ページへ続く）

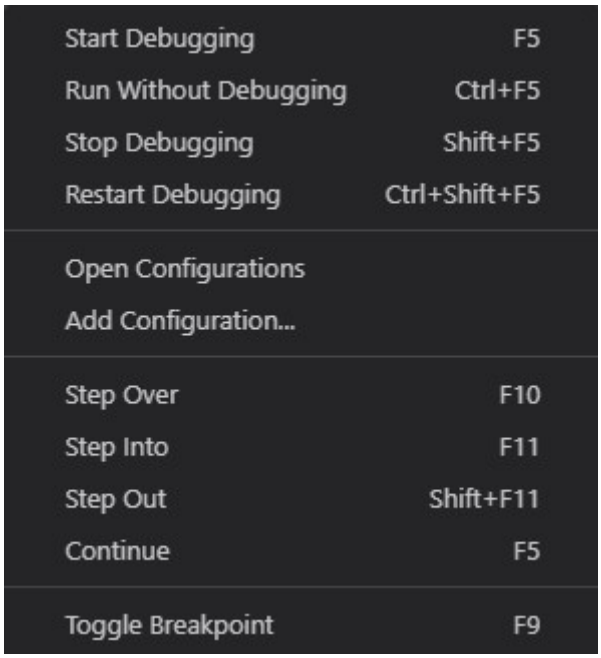
原因がわからないのですが起動時のステップ実行がうまく動いてくれません。ですからまず一度、main()の１行下の行にカーソルをあてて、F9 キーを押してブレークポイントを貼ります。

そうすると行番号の左に赤い〇がつきます。これがブレークポイントです。

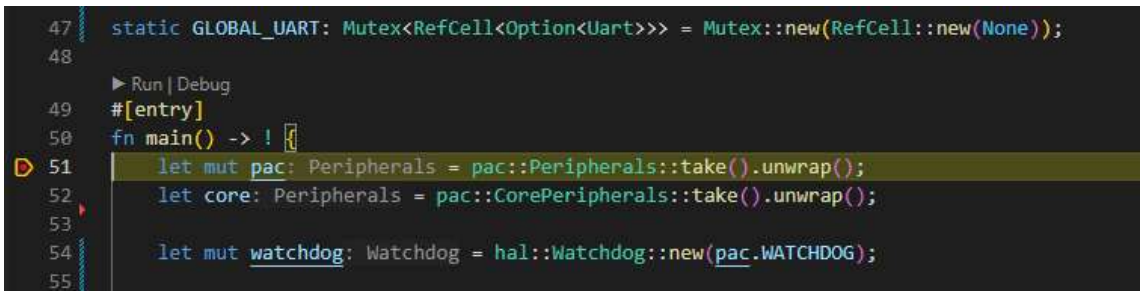


６．３．デバッグメニュー

Run メニューを選択すると以下のメニューが出てきます。
デバッグはマウス操作よりもキーを使う方が便利です。

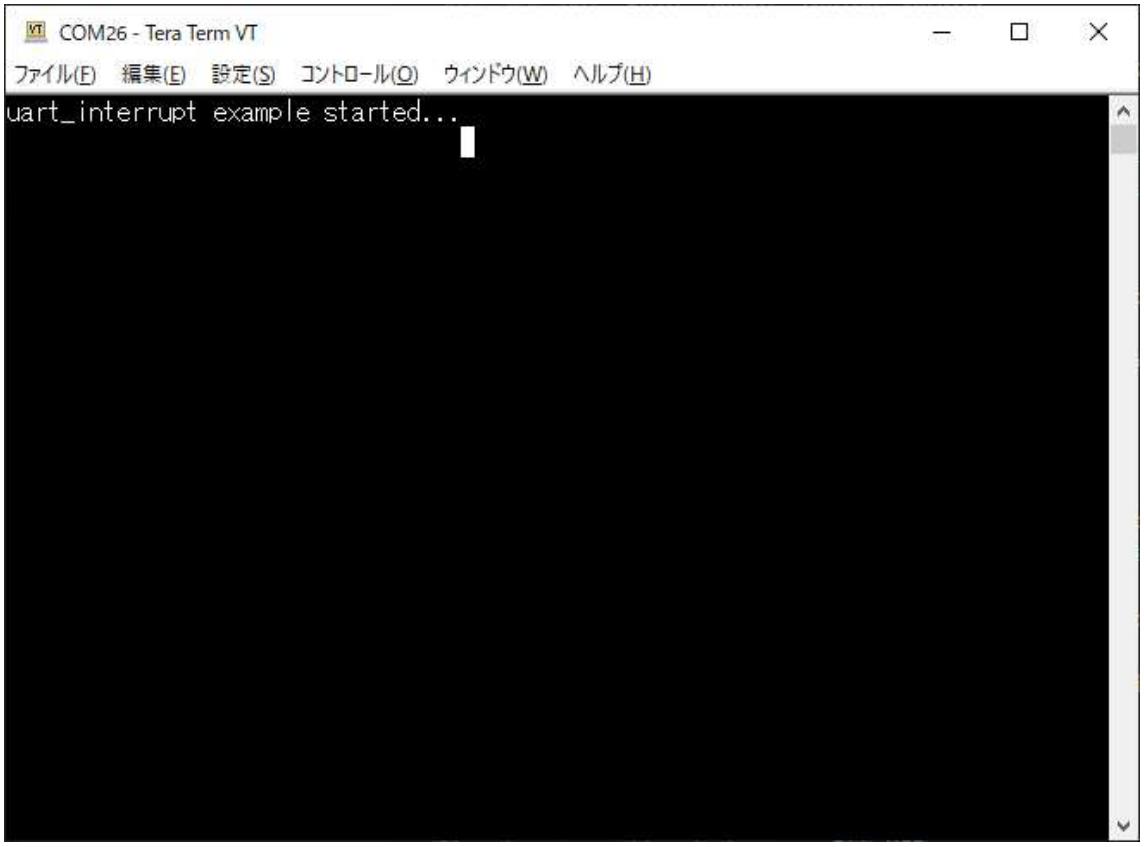


私は主に以下のキーを使っています。
F5:　デバッグ開始・実行
F10:　ステップオーバー（関数の中に入らずにステップ実行）
F11:　ステップイントゥ（関数の中に入ってステップ実行）
F9:　ブレークポイントの設定
手始めに F5 キーを押してプログラムを動かしてみます。
ブレークポイントを貼った行で停止すれば成功です。

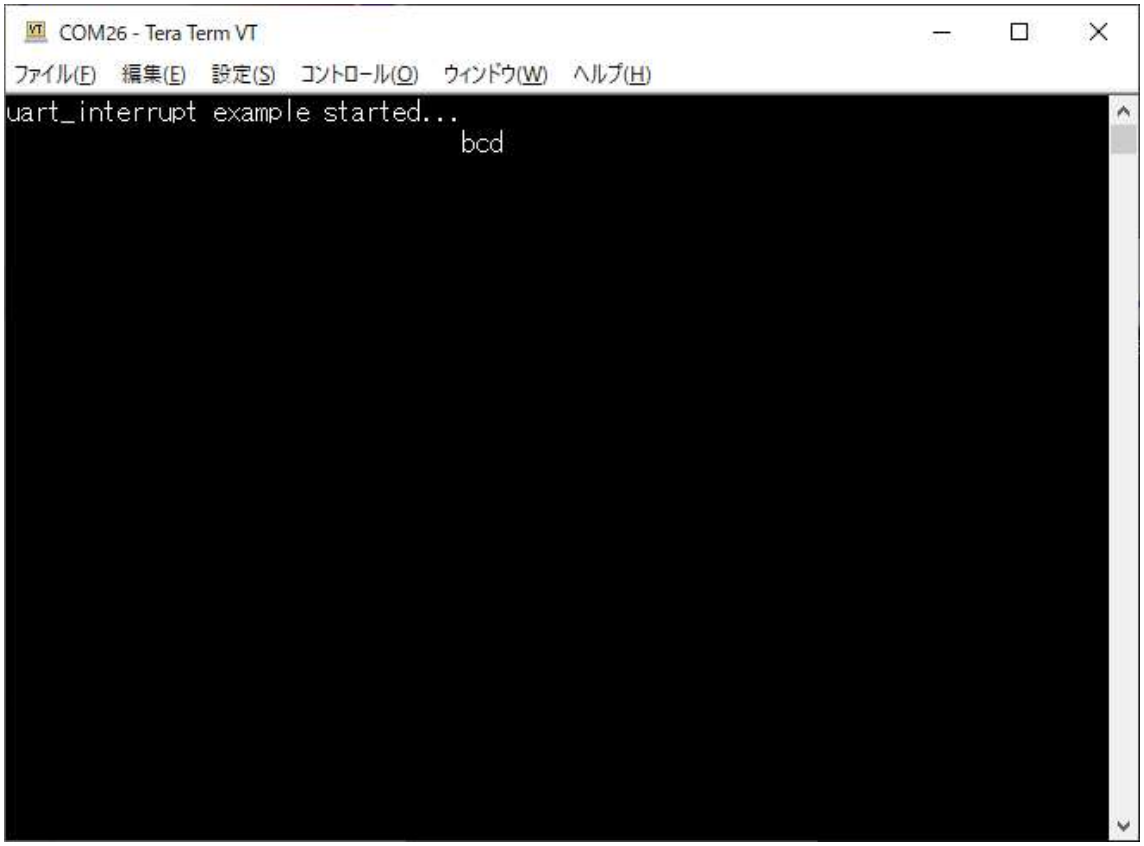


もう一度 F5 キーを押してプログラムを実行します。

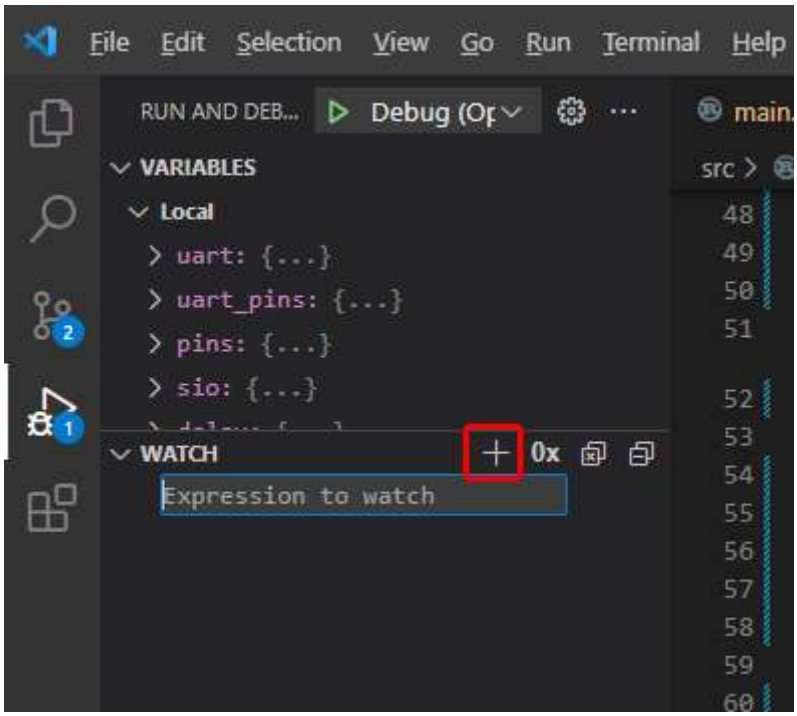
そして Tera Term に「 uart_interrupt example started... 」と表示されれば成功です。



次に Tera Term のウィンドウをクリックしてフォーカスをあてて、abc の文字を押してみます。
以下のように bcd の文字が表示されれば成功です。
一文字入力した値は UART で Raspberry Pi Pico に送られます。
Raspberry Pi Pico のマイコンは割り込みで受信した値に+1 したものを返信します。
ですから入力した値ではないものが戻り値として表示されているわけです。

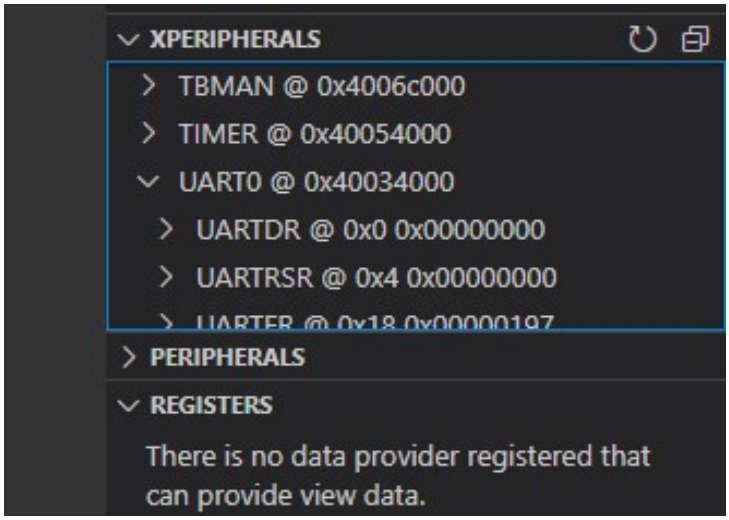


6．4．WATCH



WATCH の右側にある ＋ を選択すると Expression to watch という枠が出ますので、値を確認したい変数名を入力することでウォッチできるようになります。

6．5．XPERIPHERALS



XPERIPHERALS の左をクリックすると V に展開されて各レジスタの値を確認することができます。

6．6．VARIABLES

Local , Global , Static , Registers を表示します。

6．7．CALL STACK

コールスタック（関数が呼ばれた経路）を表示します。

７．各ファイルの解説

この章ではソースファイル（拡張子が rs のもの）以外のファイルについて説明します。

７．１．Cargo.toml

このファイルはコードと言うよりは設定用のファイルです。

[dependencies]の下に、プロジェクトで使う外部クレート（ライブラリのようなもの）を列挙します。
そうすると必要に応じて crate.io からクレートをダウンロードして来てくれます。
ローカルのダウンロード先は
C:¥xxxx¥.cargo¥registry¥src¥index.crates.io-xxxxxxxxxxxxxxxx です。

xxxx は皆さんのユーザー名
xxxxxxxxxxxxxxxx はランダム？な英数字の羅列です。

このサンプルの Cargo.toml では以下のクレートを[dependencies]の下に記述しています。

```
cortex-m = "0.7.7"
cortex-m-rt = "0.7.3"
embedded-hal = "0.2.7"
panic-halt = "0.2.0"
rp-pico = "0.8.0"
rp2040-hal = "0.9"
fugit = "0.3.7"
critical-section = "1.1.2"
```

例えば cortex-m = "0.7.7" の場合は上記の index.crates.io-xxxxxxxxxxxxxxxx の下に cortex-m-0.7.7 というフォルダをつくり、その中にクレートをコピーします。

７．２．config.toml

.cargo というフォルダの下に config.toml というファイルがあります。このファイルについて、わかる部分だけ簡単に説明します。

[build]
target = “thumbv6m-none-eabi”

これはラズピコのマイコンである RP2040 の ARM cortex-M0+ アーキテクチャをビルド対象に指定しています。

rustflags = [] の中（コンパイルオプションの指定）
“-C”, “link-arg=-Tlink.x”,
ここではリンカスクリプトに link.x というファイルを指定しています。
link.x はこちらで用意する必要はなく cortex-m-rt クレートによって自動的につくられます。

それからもうひとつ

“-C”, “linker=flip-link”,

これはスタックのプロテクションツールです。こちらでは取り上げませんが実行時のスタックのオーバーフローを防止するための機能です。

７．３．rp2040.svd

svd ファイルはデバイスのレジスタを定義したものです。
組込み Rust では svd2rust により PAC のソースコードを自動生成してくれます。
PAC は Peripheral Access Crate（ペリフェラル アクセス クレート）の略です。
このファイルは私がネットで探して来てパッケージ内に配置しました。
すでにビルドシステム内で用意されているかも知れませんが、念のために配置しました。

７．４．launch.json

VSCode のデバッグ時の構成や設定を記述したファイルです。
複製をつくって、実行ファイル名が変わる場合には "executable" の部分を編集する必要があります。

７．５．sesttings.json

上の２行は皆さまの構成に合わせてパスを設定する必要があります。
openocd.exe と arm-none-eabi-gdb.exe それぞれについて設定します。

下の２行は、Windows 環境で VSCode 上の表示に警告が出ることがあったので付加してみました。
特に編集する必要はありません。

７．６．tasks.json

ビルド時などのタスクを記述しています。
特に変更する必要はありません。

８．ソースコード解説

src フォルダの下にある main.rs が Rust のソースコードです。本章では main.rs に書かれている内容について詳しく見ていきます。
以下に出てくる(8-n-n)の部分は main.rs に // (8-n-n) のように表示した部分を示しています。先頭の 8 はこの章（8 章）を表わしています。ソースコードと照らし合わせて内容を確認するとわかりやすいと思います。

８．１．属性

#![...] と #[...] について　（ # の次に ！ があるかないかの違いです ）

これらはアトリビュート（属性）と言います。
...の部分がアトリビュートの名前です。

#![...]
こちらは内部属性と言いスコープ（範囲）を持ちます。
クレート（ファイル）単位で全体的に影響を受けます。

#[...]
対してこちらは外部属性と言い、そのすぐ下の要素だけが影響を受けます。

以下のマクロは内部属性でスコープを持つため main.rs 全体が no_std だと言っています。

```
#![no_std] // (8-1-1)
```

no_std は std ではない(スタンダードではない)の意味です。

Rust の標準ライブラリは core , alloc , std の 3 レベル構造になっています。
no_std アトリビュートを指定すると Rust コンパイラは std クレートではなく core クレートを使ってプログラムをビルドします。

OS が存在しないベアメタルな開発では no_std を使うようです。
std クレートを使わない代わりに core クレートを使うことになります。

alloc クレートはヒープメモリを利用する型を提供します。ヒープメモリを利用する Box や Vec 等の構造体が含まれています。
alloc クレートを利用するにはメモリアロケータの実装が必要で alloc-cortex-m クレート等が利用できます。
つまり、no_std だとヒープメモリを利用する Box や Vec は使えないけれど、alloc-cortex-m を使えば利用できるということになるようです。

（今回のプロジェクトでは alloc-cortex-m は使用していません）

```
#![no_main] // (8-1-2)
```

このマクロも内部属性でスコープを持つため main.rs 全体が no_main だと言っています。

std クレートに実装したコードが main()関数を呼び出すための前処理を行っています。
この前処理は OS の機能が必要ですが組込み Rust では同様の前処理を行わなえないため、このアトリビュートで処理不要であることを指示します。

(8-1-1)と(8-1-2)は併せて使うことになるでしょう。

```
#[entry] // (8-1-3)
fn main() -> ! { // (8-5-1)
```

このマクロは外部属性なので fn main() が entry だと言っています。

no_main アトリビュートにより main()関数が呼ばれなくなります。
そこで cortex-m-rt クレートの entry アトリビュートを使います。
cortex-m-rt クレートはマイコンをリセットし、スタックポインタの初期化や必要最小限のハードウェアの初期化を行った後に entry アトリビュートのついた関数を呼び出します。

```
#[interrupt] // (8-1-4)
fn UART0_IRQ() { // (8-6-20)
```

このマクロは外部属性なので fn UART0_IRQ() が interrupt だと言っています。
interrupt 属性は割り込みハンドラを定義するための属性です。
UART0_IRQ は UART0 の割り込みハンドラです。

ここまで８．１．項では属性について説明して来ましたが、no_std な Rust ではこれらの属性はそのまま定石として使えば良いと思います。
　もし割り込みを使わないシステムであれば#[interrupt]は当然不要になります。

８．２．use キーワード
　Rust の use は構造体などの型やクレート、関数他を定義しているパスを簡略化するために使われます。

```
use core::cell::RefCell; // (8-2-1)
```

これは RefCell 構造体を use している例です。
VSCode で main.rs を開き、RefCell の部分をクリックした後、右クリックしたメニューの一番上に出てくる 「 Go to Definition　　F12 」 を選択します。
そうすると RefCell を定義しているソースファイルを開いて､その部分を見せてくれます。
これは rust-analyzer の機能で非常に便利ですからぜひ活用してください。

開いたファイルの上部に RefCell までのパスが表示されています。(以下)

C: > Users > m3925 > .rustup > toolchains > stable-x86_64-pc-windows-msvc > lib > rustlib > src > rust > library > core > src > cell.rs > RefCell

これはたまたま RefCell が存在する core クレートが標準ライブラリなので、そのパスが表示されています。
「Go to Definition」は出てくる構造体型などの中を確認したい場合に役立ちます。
(m3925 は皆さんのユーザー名に置きかえてください)
この use がなぜ便利なのか説明しておきます。
(8-4-1)は以下のコードが書かれています。use を記述しているため、このように簡略化することができます。

```
static GLOBAL_UART: Mutex<RefCell<Option<Uart>>> =
Mutex::new(RefCell::new(None)); // (8-4-1)
```

use がない場合には以下のように RefCell までのパスを都度記述しなければなりません。

```
static GLOBAL_UART: Mutex<core::cell::RefCell<Option<Uart>>> =
    Mutex::new(core::cell::RefCell::new(None)); // (8-4-1)
```

キーの入力数が減るわけですから使わない手はありません。
ソースコードの見やすさも違います。

```
use rp2040_hal::Clock; // (8-2-2)
```

もうひとつ例を挙げておきます。
先と同様に Clock の部分を選択して右クリックから「Go to Definition」してみると、この Clock はトレイトであることがわかります。

開いたファイルの上部に Clock までのパスが表示されています。(以下)

C: > Users > m3925 > .cargo > registry > src > index.crates.io-xxxxxxxxxxxxxxxx >
rp2040-hal-0.9.1 > src > clocks > mod.rs > Clock

７．１項で紹介した外部クレートの例です。外部クレートは
Users¥ユーザー名¥.cargo¥registry¥src¥index.crates.io-xxxxxxxxxxxxxxxx¥クレート名
以下に保存されることを覚えておくと良いでしょう。どんなクレートが取り込まれているのか等確認することができます。

xxxxxxxxxxxxxxxx の部分はランダム？な英数字です。

```
use rp2040_hal as hal; // (8-2-11)
```

次は as を使った use の例です。
rp2040_hal では長いので hal に略して記述することができるようになります。

最後にもうひとつ use の例を挙げておきます。

```
use panic_halt as _; // (8-2-12)
```

これはパニックが発生した時にデフォルトのパニックハンドラーを使うことを指示しています。

８．３．type キーワード

Rust の type は型に別名をつけるために使われます。

```
type UartPins = (
    // (8-3-1)
    hal::gpio::Pin<Gpio0, hal::gpio::FunctionUart, hal::gpio::PullNone>,
    hal::gpio::Pin<Gpio1, hal::gpio::FunctionUart, hal::gpio::PullNone>,
);
```

この例はタプルを UartPins に略して記述しています。
これだけ長いと type せざるを得ません。

８．４．　static キーワード

Rust においてグローバル変数は static 宣言して使います。static 宣言では必ず型注釈しなければなりません。

```
static GLOBAL_UART: Mutex<RefCell<Option<Uart>>> =
Mutex::new(RefCell::new(None)); // (8-4-1)
```

Rust のグローバル変数は大文字を使うことになっています。

８．５．　！キーワード

！は Rust において "発散する" ことを表す特殊な型です。この型は「発散型（diverging type)」とも呼ばれます。絶対に終了しない関数である場合や、パニックなどでプログラムが異常終了することを示します。

```
fn main() -> ! { // (8-5-1)
```

８．６．　具体的なコード

いよいよ具体的なコードの説明に入ります。

```
    let mut pac = pac::Peripherals::take().unwrap(); // (8-6-1)
```

Raspberry Pi Pico のマイコン RP2040 は ARM Cortex-M 系のマイコンであり周辺機能は大きく分けて２つあります。
RP2040 は ARM core の機能をベースにしている他、RP2040 独自の周辺機能を持っています。
RP2040 独自の周辺機能は rp2040_pac クレートに、ARM core の周辺機能は cortex-m クレートに記述されています。

(8-6-1)のコードは前者 rp2040_pac クレートの Peripherals 構造体を取得します。
take()メソッドを右クリックから「Go to Definition」すると確認できますが、take()は Option<>型を返します。戻りは Some()か None のどちらかになります。
これはいろいろな場所から呼ばれないように制限をかけるもので、一度しか Some()が返らないようになっています。これをシングルトンと呼び、インスタンスが唯一であることを保証するための手段です。

８．６．１．パニックハンドラ

プログラムがパニックするパターンはいくつかありますが、Option 型の None を unwrap()で呼び出した場合にもパニックするので確認してみます。

まず(8-6-1)のコードをコピー＆ペーストして２行実装します。１行目の pac が使われないという警告が出ますがビルドは通りますのでそのまま進めます。

```
    let mut pac = pac::Peripherals::take().unwrap(); // (8-6-1)
    let mut pac = pac::Peripherals::take().unwrap(); // (8-6-1)
```

こうすることで2行目を実行すると、take()で Option の None が返り、それを unwrap()するのでパニックするはずです。
次に（8-2-12）の use panic_halt as _; の panic_halt を選択、右クリックして「Go to Definition」します。

関数 panic() の atomic::compiler_fence() のある行にブレークポイントを貼ります。

```
fn panic(_info: &PanicInfo) -> ! {
    loop {
        atomic::compiler_fence(Ordering::SeqCst);
    }
}
```

そしてプログラムを実行し、２回目の take() を実行すると panic()内で停止します。
２回目の take()で None が返ることが証明できました。

それでは（8-6-1）のコードを元通りの１行に戻しておきます。

let mut pac のように“mut”がついているのは、rp2040_pac クレートの Peripherals 構造体を可変で使う必要があるからです。

８．６．１．１．rust-analyzer と型注釈

```
let mut pac = pac::Peripherals::take().unwrap(); // (8-6-1)
```

このコードはエディタ上では以下のように見えているはずです。

```
let mut pac: Peripherals = pac::Peripherals::take().unwrap(); // (8-6-1)
```

（ mut pac の後に ：Peripherals がグレーの文字で追加されている ）

これを型注釈と言い、通常は Rust が型推論してくれるために私達は変数の型を省略することができます。
その省略したものがどんな型なのか rust-analyzer が翻訳して補足していくれています。
　あると型が明確にわかるので便利ですが、じゃまな場合があるかも知れません。

８．６．２．CorePeripherals

```
    let core = pac::CorePeripherals::take().unwrap(); // (8-6-2)
```

先ほど RP2040 マイコンには周辺機能が 2 つあることを述べました。
RP2040 系の周辺機能と ARM core 系の周辺機能です。

(8-6-2)のコードは後者 cortex-m クレートの Peripherals 構造体を取得します。
構造体の名前が全く同じなのでクレートを含んだパスで区別することになります。

先ほどと同じで、こちらの take()メソッドも Option 型が返ります。
やはり 2 回目以降の取得では None が返ります。
こちらの Peripherals もシングルトンであり、インスタンスが唯一であることを保証しています。

変数 core に mut がついていないのは可変で扱う必要がないためです。

８．６．３．ウォッチドッグタイマー

```
    let mut watchdog = hal::Watchdog::new(pac.WATCHDOG); // (8-6-3)
```

これはウォッチドッグタイマーを new()しています。
ウォッチドッグタイマーを使っているわけではないのですが、下の init_clocks_and_plls()関数の引数で必要になるために、ここで new()しています。

８．６．３．１．ペリフェラル・アクセス・クレート（PAC）

PAC はマイコンのペリフェラルにアクセスするための手段です。このプロジェクトでは.vscode 下にある rp2040.svd から Rust が自動的に生成します。ペリフェラルのレジスタにアクセスするための手段であり、HAL の内部などで使われています。

new()の中を覗いてみましょう。
new を選択し、右クリックしてから Go to Definition するとそのコードがある場所に飛んでいきます。

```rust
pub struct Watchdog {
    watchdog: WATCHDOG,
    load_value: u32,
}

impl Watchdog {
    pub fn new(watchdog: WATCHDOG) -> Self {
        Self {
            watchdog,
            load_value: 0,
        }
    }
}
```

Watchdog という構造体が定義されていて、そのメソッド new() の引数に pac.WATCHDOG が渡されています。
それはそのまま、Watchdog 構造体の watchdog フィールドに代入されます。
pac.WATCHDOG は参照ではなく実体ですから、new()によって move します。
move しますが、Watchdog 構造体の中に PAC の WATCHDOG は存在することになります。

8．6．3．2．ハードウェア・アブストラクト・レイヤー（HAL）

このジャンプ先は rp2040-hal-0.9.1 > src > watchdog.rs であり、Watchdog 構造体は HAL の一部であることがわかります。

例えば以下は Watchdog 構造体のメソッドの一部です。self の次の watchdog が PAC であり HAL は PAC を使って組み立てられていることがわかります。

```rust
    fn load_counter(&self, counter: u32) {
        self.watchdog.load.write(|w| unsafe { w.bits(counter) });
    }
```

８．６．４．クロックの初期化

```
    let clocks = hal::clocks::init_clocks_and_plls(
        // (8-6-4)
        rp_pico::XOSC_CRYSTAL_FREQ,
        pac.XOSC,
        pac.CLOCKS,
        pac.PLL_SYS,
        pac.PLL_USB,
        &mut pac.RESETS,
        &mut watchdog,
    )
    .ok()
    .unwrap();
```

このコードはクロックを扱うために必要な関数です。
１番目の引数は定数です。 値は 12000000（12MHz）で使っている水晶の周波数です。
２～５番目の引数は PAC の XOSC, CLOCKS, PLL_SYS, PLL_USB で実体を渡しています。

関数の戻り値が Result<ClocksManager, InitError> 型になっています。
渡した実体で必要なものは ClocksManager の中で存在することになります。
６番目の引数 pac.RESETS と７番目の引数 watchdog は可変で扱いたいので &mut をつけて渡しています。当然 init_clocks_and_plls()のシグニチャもそう受け取るようになっています。

init_clocks_and_plls() の後ろに .ok() がついています。
これも右クリックして Go to Definition してみると Option<T>が返ることがわかります。
ok()は Result 型のメソッドです。
Result 型の Ok を Some に変換するか、 Err を None に変換しています。
更に .unwrap() がついていますので、T が返るか None が返ることになります。

T の型は ClocksManager で、 None が返るとパニックします。

８．６．５．Delay::new()関数

```
    let mut delay = cortex_m::delay::Delay::new(core.SYST,
clocks.system_clock.freq().to_Hz()); // (8-6-5)
```

Delay は時間待ちするための構造体で、cortex_m クレートで定義されています。
第 1 引数に core.SYST の実体を渡します。
第 2 引数にシステムクロック（単位 Hz）を渡します。

delay_ms() , delay_us() のメソッドがあります。
指定した msec , μsec 分の時間待ちを行います。

８．６．６．SIO

```
    let sio = hal::Sio::new(pac.SIO); // (8-6-6)
```

pac.SIO を引数として渡し、move して HAL の Sio 構造体を new()しています。
SIO は GPIO を制御するためのハードウェアで私たちが直接的に扱うペリフェラルではありません。次の Pins::new() の引数で渡すためにここで new()しています。
この SIO は Raspberry Pi Pico 特有の機能と言っていいでしょう。

8．6．7．Pins

```
    let pins = rp_pico::Pins::new(
        // (8-6-7)
        pac.IO_BANK0,
        pac.PADS_BANK0,
        sio.gpio_bank0,
        &mut pac.RESETS,
    );
```

new()を右クリックして Go to Definition してもメソッドにたどり着けませんでした。
そこで実際にプログラムを動かしてこの中にステップ実行して入ってみました。

マクロになっていると簡単に定義にたどり着けないようです。
ステップ実行の先は rp2040-hal-0.9.1 > src > gpio > mod.rs です。

その周辺を見ると Pins は構造体であることがわかり、new()はそのメソッドであることがわかりました。
第１～３引数には IO_ , PADS_, gpio_ それぞれの BANK0 を指定していますが、GPIO0-29 番は全て BANK0 であり、Pins::new()する場合にはこれらを使うことが決まっているようです。
第４引数には pac.RESETS を &mut で渡します。

この後 pins のフィールドを使って UART のピンを設定します。

（コメント）
ここに出てくる rp_pico クレートですがファイルを開くと以下のように書かれています。

```
//! A Hardware Abstraction Layer for the Raspberry Pi Pico.
```

でも実際にはピンの定義などを行っているので、位置づけとしては BSP になるのだと思います。

8．6．8．UART のピンを組み合わせる

```
    let uart_pins = (
        // (8-6-8)
        pins.gpio0.reconfigure(),
        pins.gpio1.reconfigure(),
    );
```

Pins 構造体のフィールド gpio0 , gpio1 はそれぞれ GPIO0, GPIO1 のピンに相当します。
GPIO0 には UART0 TX の機能が割り当てられています。
また GPIO1 には UART0 RX の機能が割り当てられています。
これらのピンを UART の機能として Picoprobe と接続しています。
そこでこれらをタプルにして uart_pins にまとめて、次の UartPeripheral::new()の引数として渡しています。
タプルの要素 0（左側）には tx , 要素 1（右側）には rx に相当するピンを指定する必要があります。仮に gpio0 と gpio1 をひっくり返してビルドするとエラーを検出してくれます。

８．６．９．UartPeripheral 構造体

```
    let mut uart = hal::uart::UartPeripheral::new(pac.UART0, uart_pins,
&mut pac.RESETS) // (8-6-9)
        .enable(
            UartConfig::new(9600.Hz(), DataBits::Eight, None,
StopBits::One),
            clocks.peripheral_clock.freq(),
        )
        .unwrap();
```

上で定義したタプルなどを使って UartPeripheral 構造体を new()します。
new()した UartPeripheral を enable() , unwrap() します。
enable()は Result<UartPeripheral, Error> が返るので、unwrap()して UartPeripheral を取り出しています。
（エラーの場合にはパニックします）

８．６．１０．cortex-m::peripheral::NVIC::unmask()関数

```
    unsafe {
        // (8-6-10)
        pac::NVIC::unmask(hal::pac::Interrupt::UART0_IRQ);
    }
```

割り込みベクターである UART0_IRQ に対して pac::NVIC::unmask()を unsafe に実行します。
この処理を行わないと割り込みハンドラが使えません。

use を使っているので一部省略されていますがフルパスで表現すると unmask()は以下のようにして呼ばれます。

cortex-m::peripheral::NVIC::unmask()

おそらく cortex-M 系のマイコンの場合、割り込みのアンマスクでは常にこの関数が呼ばれているのだと思います。
割り込みを使うためには、これ以外にもペリフェラルに対する割り込み要因を許可する必要があります。

８．６．１１．UART 受信割り込みのイネーブル

```
    uart.enable_rx_interrupt(); // (8-6-11)
```

UART0_IRQ を許可しました。
この uart は new()の第 1 引数に pac.UART0 を指定しているので、ここでは UART0 の受信割り込みを許可することになります。

これは HAL の UartPeripheral 構造体のメソッドです。

８．６．１２．UART 送信

```
    uart.write_full_blocking(b"uart_interrupt example started...¥n"); //
(8-6-12)
```

「ブロッキングして全てを書く」という意味ですから、全ての文字を送信するまでこのメソッドから戻ってきません。

８．６．１３．クリティカルセクション(1)

```
    critical_section::with(|cs| {
        // (8-6-13)
        GLOBAL_UART.borrow(cs).replace(Some(uart));
    });
```

GLOBAL_UART はグローバル変数です。
以下のように宣言されています。

```
static GLOBAL_UART: Mutex<RefCell<Option<Uart>>> =
Mutex::new(RefCell::new(None)); // (8-4-1)
```

グローバル変数はできるだけ mut で扱うべきではないので、とりあえず不変な値で初期化しておきます。
初期化する際に型が決まっていないので Option<Uart>型を使ってとりあえず None で初期化しておきます。
後に可変でアクセスするために RefCell でラップし、更に排他制御するために Mutex でラップします。
グローバル変数の宣言には型注釈が必要になります。
型注釈の Uart の部分は実際には以下のように長ったらしい型になっています。

```
type Uart = hal::uart::UartPeripheral<hal::uart::Enabled, pac::UART0,
UartPins>; // (8-3-2)
```

critical_section::with()の内部で None を Some(uart) に置きかえています。
クリティカルセクションは GLOBAL_UART を排他制御するためのしくみです。割り込みを含む別のスレッドと同時にアクセスすることができないようになっています。

以下のコードの説明です。

```
        GLOBAL_UART.borrow(cs).replace(Some(uart));
```

borrow()に cs が紐づけされているので Mutex の借用も cs のライフタイムと同じになります。

borrow()は Mutex 構造体のメソッドで以下のように実装されています。

```
    pub fn borrow<'cs>(&'cs self, _cs: CriticalSection<'cs>) -> &'cs T {
        unsafe { &*self.inner.get() }
    }
```

inner は UnsafeCell です。UnsafeCell の get メソッドは以下のシグネチャです。

pub const fn get(&self) -> *mut T

*mut T は RefCell へのポインター ですから *self.inner.get()は RefCell の値であり &*self.inner.get() は RefCell の参照を返します。

生ポインターを読むことから UnsafeCell と名付けられているのでしょう。

ですから borrow() でここを読む時に unsafe {} でくくっています。
&'cs T なので戻り値（RefCell の参照）のライフタイムはクリティカルセクションと同じになります。これは制約をかけているのだと思います。

続いて replase()は RefCell のメソッドで値を T に置きかえて T を返します。

pub fn replace(&self, t: T) -> T　この場合、T は Some(uart)

こうして GLOBAL_UART の中身は None から Some(uart) に置きかえられて使えるようになりました。

８．６．１４．LED

```
    let mut led_pin = pins.led.into_push_pull_output(); // (8-6-14)
```

into_push_pull_output()は Pin 構造体のメソッドです。

プッシュプルの出力ピンに設定することで LED を ON/OFF します。

８．６．１５．loop{}

継続して LED を ON/OFF するためにループします。

８．６．１６．cortex_m::asm::wfe()関数

```
        cortex_m::asm::wfe(); // (8-6-16)
```

wfe は wait for event の略です。

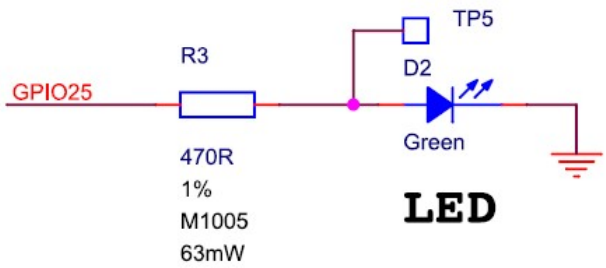
イベントが発生するまで待機します。今回のコードでは UART の受信割り込み処理が終わるとイベントが発生します。

８．６．１７．set_high()メソッド

```
        led_pin.set_high().unwrap(); // (8-6-17)
```

LED を制御するピン（GPIO25）を H レベルにします。

以下の回路になっているので、set_high()で LED が点灯します。



８．６．１８．delay_ms()メソッド

```
        delay.delay_ms(100); // (8-6-18)
```

引数に設定した値の分だけミリ秒待ちします。この場合は 100msec 待ちになります。

８．６．１９．set_low()メソッド

```
        led_pin.set_low().unwrap(); // (8-6-19)
```

LED を制御するピン（GPIO25）を L レベルにし、LED を消灯します。

８．６．２０．UART0_IRQ
UART0 の割り込みハンドラ関数です。
今回の初期設定のコードでは UART0 の受信により、この関数に飛んでくることになります。

８．６．２１．クリティカルセクション(2)

クリティカルセクションを実行します。

８．６．２２．GLOBAL_UART を使う

```
        if let Some(ref mut uart) =
GLOBAL_UART.borrow(cs).borrow_mut().deref_mut() {
            // (8-6-22)
```

8.6.13.で None から Some()に置きかえたのでこの if()の中は実行されることになります。

８．６．２３．read()メソッド

```
            while let Ok(byte) = uart.read() {
                // (8-6-23)
```

読めるだけ読みます。
read()を選択して右クリックし、メニューから「Go to Definition」します。

```
impl<D: UartDevice, P: ValidUartPinout<D>> Read<u8> for
UartPeripheral<Enabled, D, P> {
    type Error = ReadErrorType;

    fn read(&mut self) -> nb::Result<u8, Self::Error> {
        let byte: &mut [u8] = &mut [0; 1];

        match self.read_raw(byte) {
            Ok(_) => Ok(byte[0]),
            Err(e) => match e {
                Other(inner) => Err(Other(inner.err_type)),
                WouldBlock => Err(WouldBlock),
            },
        }
    }
}
```

UartPeripheral は HAL の構造体です。Read がトレイトの名前で、read()はトレイトのメソッドです。
この部分は embedded-hal の Read トレイトの read()メソッドの実装になります。
Read<u8>の部分を選択して右クリックし、メニューから「Go to Definition」します。
そうすると embedded-hal-0.2.7 > src > serial.rs というファイルが開きます。

Read トレイトの定義は以下のようになっています。
HAL の構造体にこのシグネチャに合わせて read()を実装してあげると、抽象化することができます。
これはアプリケーション層などの上位から下位レイヤーを意識することなく UART のリードを行うしくみです。
　いろいろなマイコンが存在しそれぞれの HAL が定義されますが、この Read トレイトの read()のシグネチャに合わせて HAL の構造体に read()を実装してあげれば上位層から下位の中を意識することなく扱えるようになります。
　トレイトはこのようにインターフェース的な役割を果たすことができます。

```
pub trait Read<Word> {
    /// Read error
    type Error;

    /// Reads a single word from the serial interface
    fn read(&mut self) -> nb::Result<Word, Self::Error>;
}
```

８．６．２４．write()メソッド

```
                let _ = uart.write(byte + 1); // (8-6-24)
```

ループバックしたことがわかるように読んだ値に 1 を加えたものを write()しています。
write()を選択して右クリックし、メニューから「Go to Definition」します。

```
impl<D: UartDevice, P: ValidUartPinout<D>> Write<u8> for
UartPeripheral<Enabled, D, P> {
    type Error = Infallible;

    fn write(&mut self, word: u8) -> nb::Result<(), Self::Error> {
        if self.write_raw(&[word]).is_err() {
            Err(WouldBlock)
        } else {
            Ok(())
        }
    }
}
```

UartPeripheral は HAL の構造体です。Write がトレイトの名前で、write()はトレイトのメソッドです。
この部分は embedded-hal の Write トレイトの write()メソッドの実装になります。
Write<u8>の部分を選択して右クリックし、メニューから「Go to Definition」します。
そうすると embedded-hal-0.2.7 > src > serial.rs というファイルが開きます。
read()の時と全く同じで抽象化して使うことができるようになっています。

８．６．２５．cortex_m::asm::sev()関数

```
    cortex_m::asm::sev(); // (8-6-25)
```

sev は set event の略です。
イベントを発生させます。
8.6.16 の wfe()と組みで使われていて、イベントを発生されることで wfe()の待ちが解除されます。

９．関連サイトのご紹介

今回は Raspberry Pi Pico を使った組み込み Rust の開発環境についてお話しました。

Rust 以外にも Raspberry Pi Pico の SDK を使ったシステム構築、ゼロから作る OS 等の記事をブログにしています。

以下にサイトマップを掲載させて頂きますので、よろしければご覧になってください。

https://moons.link/pico/sitemap/

１０．おわりに

ここまでお付き合いくださり、ありがとうございました。
この本が少しでも組み込み Rust の発展に役立てば幸いです。
どうぞよろしくお願いいたします。