Raspberry Pi Pico をセットアップしよう

Raspberry Pi Pico およびその他の RP2040 ベースのマイクロコントローラーボードを使用した C/C++ 開発

奥付

作成日: 2023-03-22
バージョン: 0880191-clean

SDK について

本文書中の「SDK」は、当社の Raspberry Pi Pico SDK を意味します。SDK の詳細については、「Raspberry Pi Pico C/C++ SDK」で参照することができます。本文書に記載されているソースコードは、 3 条項 BSD ライセンスに含まれています。Copyright © 2020-2022 Raspberry Pi Ltd (formerly Raspberry Pi (Trading) Ltd.)

法的免責事項

この「法的免責事項」の日本語訳は、情報提供のみを目的としていることにご注意ください。以下に記載されている「法的免責事項」の英語原文のみが法的な拘束力を持ちます。

随時変更される Raspberry Pi 製品の技術データおよび信頼性データ (データシートを含む) (以降は「資料」と表記) は、Raspberry Pi Ltd (以降は「RPL」と表記) によって「現状のまま」提供され、商品性および特定の用途への適合性の黙示的保証を含みますがこれに限定されず、いかなる明示的または黙示的な保証も否定します。いかなる場合も、適用される法律が許容する最大限の範囲において、原因の如何または法的責任の根拠にかかわらず、本資料の使用により何らかの形で生じる契約義務、厳格責任、(過失または故意の) 不法行為の直接的、間接的、付随的、特別、例示的、または結果的損害 (代替商品またはサービスの調達、用途、データ、または利益の喪失、または事業の中断を含みますがこれに限定されません) について、RPL は当該損害の発生の可能性を通知されていた場合を含め、責任を負わないものとします。

RPL は、本資料の内容または本資料に記載されている製品に対し、いつでも予告なしに拡張、改善、修正またはその他の変更を加える権利を留保するものとします。

本資料は、適切なレベルの設計知識を持つ熟練したユーザーを対象としています。ユーザーは、本資料の選択と使用、および本資料に記載されている製品の適用について、単独で責任を負うものとします。ユーザーは、本資料の使用により生じるすべての責任、費用、損害またはその他の損失について RPL を補償し、損害を与えないことに同意するものとします。

RPL は、Raspberry Pi 製品に関連してのみ本資料の使用をユーザーに許諾します。それ以外の本資料の使用は禁止されます。その他の RPL の知的財産権またはその他の第三者の知的財産権に対するライセンスは付与されません。

危険性の高い活動: Raspberry Pi 製品は、核施設、航空機の航行システムまたは通信システム、航空管制、兵器システム、セーフティクリティカルなアプリケーション (生命維持装置などの医療機器を含む) の運用など、製品の故障が死、人身障害、深刻な物理的または環境的損害に直接つながるような、フェイルセーフ性能を必要とする危険な環境での使用 (これ以降は「危険性の高い活動」と表記) を意図して設計または製造されていません。RPL は、危険性の高い活動に対する明示的または黙示的な適合性の保証を明確に否定し、危険性の高い活動における Raspberry Pi 製品の使用または適用について一切の責任を負いません。

Raspberry Pi 製品は、RPL の 標準規約に従って提供されます。RPL による本資料の提供は、RPL の 標準規約 (標準規約に記載されている免責事項および保証を含むがこれに限定されません) を拡張または変更するものではありません。

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME (“RESOURCES”) ARE PROVIDED BY RASPBERRY PI LTD (“RPL”) "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage (“High Risk Activities”). RPL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPL’s Standard Terms. RPL’s provision of the RESOURCES does not expand or otherwise modify RPL’s Standard Terms including but not limited to the disclaimers and warranties expressed in them.

目次

第章 1. Pico のクイックセットアップ

Raspberry Pi 4B または Raspberry Pi 400 で Raspberry Pi Pico の開発作業を行う場合は、セットアップスクリプトを実行すると、この入門ガイドに記載されているインストール手順のほとんどを省略することができます。

注記

このセットアップスクリプトを実行するには、SD カード上に約 2.5GB の空きディスク容量がなければなりません。十分な空き容量があることを確認してから、スクリプトを実行してください。`df -h` コマンドで、ディスクの空き容量を確認することができます。

ターミナルで以下のコマンドを実行すると、このセットアップスクリプトを取得することができます。

```
$ wget https://raw.githubusercontent.com/raspberrypi/pico-setup/master/pico_setup.sh ①
```

1. `wget` がインストールされていない場合は、最初に `sudo apt install wget` コマンドを実行する必要があります。

次に、以下のコマンドを実行して、スクリプトを起動可能な状態にします。

```
$ chmod +x pico_setup.sh
```

次に、以下のコマンドを実行してスクリプトを起動します。

```
$ ./pico_setup.sh
```

このスクリプトにより、以下の処理が実行されます。

- `pico` というディレクトリが作成されます。
- 必要な依存関係がインストールされます。
- `pico-sdk`、`pico-examples`、`pico-extras`、`pico-playground` というリポジトリがダウンロードされます。
- `~/.bashrc` ファイル内に、`PICO_SDK_PATH`、`PICO_EXAMPLES_PATH`、`PICO_EXTRAS_PATH`、`PICO_PLAYGROUND_PATH` が定義されます。
- サンプルの `blink` プログラムと `hello_world` プログラムが、`pico-examples/build/blink` フォルダー内と `pico-examples/build/hello_world` フォルダー内に作成されます。
- `picotool` (付録 B を参照) のダウンロードとビルドが実行され、`/usr/local/bin` フォルダーにコピーされます。
- `picoprobe` (付録 A を参照) のダウンロードとビルドが実行されます。
- OpenOCD のダウンロードとコンパイルが実行されます (デバッグ用)。
- 次のダウンロードとインストールが実行されます: Visual Studio Code
- Visual Studio Code の必須拡張機能がインストールされます (第章 7を参照)。
- Raspberry Pi Pico で使用できるように Raspberry Pi の UART が設定されます。

注記

`pico` ディレクトリは、`pico_setup.sh` スクリプトを実行したフォルダー内に作成されます。

このスクリプトを実行したら、以下のコマンドで Raspberry Pi を再起動する必要があります。

```
$ sudo reboot
```

これにより、UART の設定内容が有効になります。Raspberry Pi を再起動したら、「プログラミング」メニューで Visual Studio Code を開き、Section 7.2 に記載されている手順を実行します。

第章 2. SDK

重要

このマニュアルに記載されている説明は、Raspberry Pi Pico を使用することを前提としています。RP2040 ベースのボードを使用する場合は、細かい点がいくつか異なる場合があります。また、Raspberry Pi 4 上で稼働する Raspberry Pi OS (あるいは、別のプラットフォーム上で稼働する Debian ベースの同等の Linux ディストリビューション) を使用することを前提としています。このマニュアルには、Linux の代わりに Microsoft Windows (Section 9.2 を参照) または Apple macOS (Section 9.1 を参照) を使用する場合の説明も記載されています。

Raspberry Pi Pico は、Raspberry Pi 用に設計された RP2040 マイクロコントローラーをベースとして開発されています。ボード上での開発作業は、C/C++ SDK と公式 MicroPython ポートの両方で完全にサポートされています。本書では、SDK の使用方法と、SDK ツールチェーンの作成方法、インストール方法、操作方法について説明します。

ヒント

公式 MicroPython ポートの詳細については、「Raspberry Pi Pico Python SDK」と「Get started with MicroPython on Raspberry Pi Pico」(出版社: Raspberry Pi Press、著者: Gareth Halfacree、Ben Everard) を参照してください。

ヒント

C/C++ SDK の詳細については、API レベルのドキュメントと「Raspberry Pi Pico C/C++ SDK」を参照してください。

2.1. SDK とサンプルの入手

`pico-examples` リポジトリ (Pico Examples) には、`pico-sdk` リポジトリ (SDK) を使用して開発された一連のサンプルアプリケーションが用意されています。これらのリポジトリを複製する場合は、最初に pico ディレクトリを作成し、pico に関連するすべてのオブジェクトをこのディレクトリに保存してください。以下のコマンドを実行すると、pico ディレクトリが `/home/pi/pico` に作成されます。

```
$ cd ~/
$ mkdir pico
$ cd pico
```

次に、以下のコマンドを実行して、2 つの Git リポジトリ (`pico-sdk` と `pico-examples` を複製します。

```
$ git clone https://github.com/raspberrypi/pico-sdk.git --branch master
$ cd pico-sdk
$ git submodule update --init
$ cd ..
$ git clone https://github.com/raspberrypi/pico-examples.git --branch master
```

警告

上記の `git submodule update --init` コマンドを実行しなかった場合、`tinyusb` モジュールがリポジトリ内に保管されないため、USB 機能が SDK 内でコンパイルされることはありません。そのため、USB シリアルをはじめとする USB 機能やサンプルコードは機能しません。

注記

このほかにも、 Pico Extras リポジトリと Pico Playground リポジトリがあります。

2.2. ツールチェーンのインストール

`pico-examples` リポジトリ内のアプリケーションをビルドするには、いくつかのツールをインストールする必要があります。プロジェクトをビルドするには、ソフトウェアをビルドするための CMake というクロスプラットフォームツールと、 GNU Embedded Toolchain for Arm というツールが必要になります。いずれのツールも、コマンドラインで `apt` コマンドを実行してインストールすることができます。`apt` コマンドを実行しても、すでにインストールされているツールには影響しません。

```
$ sudo apt update
$ sudo apt install cmake gcc-arm-none-eabi libnewlib-arm-none-eabi build-essential ①
```

1. `pioasm` と `elf2uf2` をコンパイルするには、gcc と g++ が必要です。

注記

Ubuntu または Debian を使用している場合は、`libstdc++-arm-none-eabi-newlib` もインストールしなければならない可能性があります。

2.3. SDK の更新

新しいバージョンの SDK がリリースされた場合、SDK のコピーを更新する必要があります。これを行うには、SDK のコピーが保管されている `pico-sdk` ディレクトリに移動して、以下のコマンドを実行します。

```
$ cd pico-sdk
$ git pull
$ git submodule update
```

注記

`pico-sdk` リポジトリでカスタムウォッチを設定すると、新しいリリースの通知を受け取ることができます。これを行うには、 https://github.com/raspberrypi/pico-sdk にアクセスし、「Watch」→「Custom」→「Releases」の順に選択します。これにより、新しい SDK がリリースされるたびに、メールで通知が送信されるようになります。

第章 3. C 言語で LED を点滅させる

新しいプログラミング環境でハードウェア用のソフトウェアを開発する場合、通常は、LED の点灯、消灯、再点灯を行うプログラムを最初に作成します。LED の点滅をプログラムで制御する方法を理解することが、その後のプログラミングの基礎になります。ここでは、RP2040 のピン 25 に接続されている Raspberry Pi Pico ボードの LED を点滅させてみます。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/blink/blink.c Lines 9 - 23

```
 9 int main() {
10 #ifndef PICO_DEFAULT_LED_PIN
11 #warning blink example requires a board with a regular LED
12 #else
13     const uint LED_PIN = PICO_DEFAULT_LED_PIN;
14     gpio_init(LED_PIN);
15     gpio_set_dir(LED_PIN, GPIO_OUT);
16     while (true) {
17         gpio_put(LED_PIN, 1);
18         sleep_ms(250);
19         gpio_put(LED_PIN, 0);
20         sleep_ms(250);
21     }
22 #endif
23 }
```

3.1. 点滅のコーディング

先ほど作成した pico ディレクトリから `pico-examples` リポジトリに移動し (cd コマンド)、build ディレクトリを作成します。

```
$ cd pico-examples
$ mkdir build
$ cd build
```

次に、以下のコマンドを実行して `PICO_SDK_PATH` を設定します。ここでは、`pico-sdk` リポジトリと `pico-examples` リポジトリが同じディレクトリ内に複製されているという前提になっています。

```
$ export PICO_SDK_PATH=../../pico-sdk
```

ヒント

本書では、「`../../pico-sdk`」という相対パスを使用して、`PICO_SDK_PATH` に対して SDK をチェックアウトします。ただし、チェックアウトの場所によっては、この相対パスを「`/home/pi/pico/pico-sdk`」などの絶対パスに置き換えなければならない場合があります。

`cmake ..` コマンドを実行して、cmake build ディレクトリを準備します。

```
$ cmake ..
Using PICO_SDK_PATH from environment ('../../pico-sdk')
PICO_SDK_PATH is /home/pi/pico/pico-sdk
   .
   .
   .
-- Build files have been written to: /home/pi/pico/pico-examples/build
```

注記

コンパイラーの最適化機能が有効になっていて、デバッグ情報が削除されている状態で `cmake` コマンドを実行すると、デフォルトで `Release` ビルドが作成されます。デバッグ用のビルドを作成する場合は、`cmake -DCMAKE_BUILD_TYPE=Debug ..` コマンドを実行します。これについては、Section 6.1 で詳しく説明します。

CMake により、`pico-examples` ツリー用のビルド領域が作成されました。この領域で `make` と入力することにより、すべてのサンプルアプリケーションをビルドすることができます。ただし、ここでビルドするのは `blink` アプリケーションだけであるため、`blink` ディレクトリに移動してから `make` と入力します。

ヒント

`-j4` オプションを指定して `make` コマンドを実行すると、4 つの make ジョブが同時に実行されるため、処理速度が上がります。Raspberry Pi 4 には 4 つのコアがあるため、`-j4` オプションを指定することをお勧めします。

```
$ cd blink
$ make -j4
Scanning dependencies of target ELF2UF2Build
Scanning dependencies of target boot_stage2_original
[  0%] Creating directories for 'ELF2UF2Build'

   .
   .
   .
[100%] Linking CXX executable blink.elf
[100%] Built target blink
```

これにより、以下のターゲットファイルが作成されます。

- `blink.elf`: このファイルは、デバッガーによって使用されます。
- `blink.uf2`: このファイルは、RP2040 USB マスストレージデバイス上にドラッグすることができます。

これらのバイナリファイルにより、RP2040 の GPIO25 に接続されている Raspberry Pi Pico ボードの LED が点滅します。

> サンプルコードの詳細情報の入手先
>
> 本書では、ソフトウェアをビルドして Raspberry Pi Pico に読み込む方法について説明します。`blink` アプリケーションがバイナリプログラムに変換されるまでにさまざまな処理が実行されますが、詳しい仕組みについては「Raspberry Pi Pico C/C++ SDK」に記載されています。ここでは、まだ詳しい仕組みについて理解する必要はないため、このまま本書を読み進めてください。

3.2. blink アプリケーションの読み込みと実行

RP2040 ベースのボードに初めてソフトウェアを読み込む場合、USB マスストレージデバイスとしてマウントするのが最も簡単な方法です。この方法により、ファイルをボード上にドラッグしてフラッシュをプログラミングすることができます。`BOOTSEL` ボタン (図 1 を参照) を押しながら、Micro USB ケーブルを使用して、Raspberry Pi Pico を Raspberry Pi に接続します。これにより、強制的に USB マスストレージモードに切り替わります。

3.2.1. デスクトップ上での作業

Raspberry Pi Desktop が稼働している場合、Raspberry Pi Pico が自動的に USB マスストレージデバイスとしてマウントされます。マウントされたら、`blink.uf2` ファイルをマスストレージデバイスにドラッグアンドドロップします。

RP2040 が再起動し、マスストレージデバイスのマウントが解除され、フラッシュされたコードの実行が開始されます (図 1 を参照)。

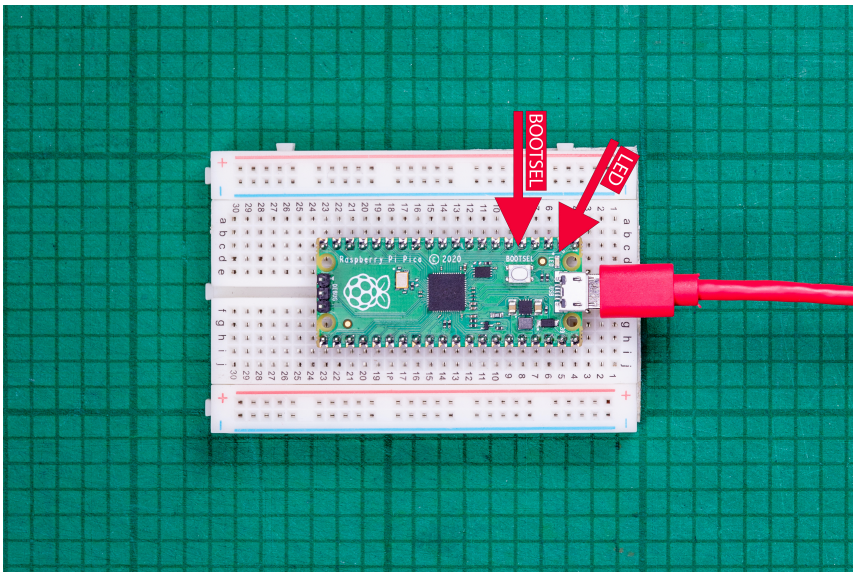


図 1. Raspberry Pi Pico ボードの LED が点滅します。図内の矢印は、ボード上の LED と `BOOTSEL` ボタンをそれぞれ示しています。

3.2.2. コマンドラインの使用

ヒント

`picotool` を使用して、UF2 バイナリプログラムを Raspberry Pi Pico に読み込むことができます (付録 B を参照)。

`ssh` などを使用してログインしている場合、マスストレージデバイスを手動でマウントしなければならないことがあります。

```
$ dmesg | tail
[  371.973555] sd 0:0:0:0: [sda] Attached SCSI removable disk
$ sudo mkdir -p /mnt/pico
$ sudo mount /dev/sda1 /mnt/pico
```

`/mnt/pico` フォルダーにファイルが表示されれば、USB マスストレージデバイスが正しくマウントされています。

```
$ ls /mnt/pico/
INDEX.HTM  INFO_UF2.TXT
```

`blink.uf2` ファイルを RP2040 にコピーします。

```
$ sudo cp blink.uf2 /mnt/pico
$ sudo sync
```

RP2040 は、USB マスストレージデバイスとしてはすでに切断されていますが、念のため、以下のコマンドで `/mnt/pico` のマウントを解除します。

```
$ sudo umount /mnt/pico
```

注記

ボードの電源を落としても、コードが削除されることはありません。ボードの電源を入れると、電源を落とす前に読み込んだコードがもう一度実行されます。新しいコードをボードにアップロードする場合は (ボード上のコードを上書きする場合)、BOOTSEL ボタンを押しながら電源を入れて、ボードをマスストレージモードに切り替えます。

3.2.3. 参考情報: その他のボード

Raspberry Pi Pico 以外のボードでは、BOOTSEL ボタン (図 1 を参照) がない場合があります。その場合は、そのボードの提供元が指定する方法でコードを読み込むことができます。

- ほとんどのボードに SWD インターフェイス (第章 5を参照) が付属しているため、ボタンを押すことなくボードのリセットやコードの読み込みを行うことができます。
- いくつかの方法で、フラッシュ CS ピンをプルダウンできる場合があります (これは、BOOTSEL ボタンを Raspberry Pi Pico で機能させる場合と同じです)。たとえば、1 組のジャンパーピンを短絡させる、などの方法です。
- リセットボタンはあるが BOOTSEL ボタンはないというボードもあります。こうしたボードの場合、コードをフラッシュに読み込んでリセットボタンの 2 度押しを検知し、ブートローダーを起動することができます。

いずれの場合であっても、使用するボードのドキュメントを参照し、そのボードにファームウェアを読み込むための最適な方法を実行する必要があります。

3.2.4. 参考情報: ハンズフリーフラッシュプログラミング

Raspberry Pi Pico で BOOTSEL モードに移行し、USB からコードを読み込むには、BOOTSEL ボタンを押しながら、何らかの方法でボードをリセットする必要があります。これを行うには、USB コネクターを取り外してからもう一度接続するか、RUN ピンを下げるための外部ボタンを追加します。

また、SWD ポート (第章 5を参照) を使用してボードをリセットし、コードを読み込んでプロセッサーの動作を設定することもできます。この方法は、コードがクラッシュした場合でも機能するため、手動でボードをリセットしたり、いずれかのボタンを押したりする必要はありません。プログラムをビルドするための準備がすべて整い、第章 4でサンプルの「Hello World」プログラムを実行したら、次のステップとして SWD の設定を行うことをお勧めします。

Raspberry Pi の場合、`pico-setup` スクリプトを実行し (第章 1を参照)、第章 5 のように Pi と Pico を 3 本のワイヤで接続すれば、SWD の設定を行うことができます。USB と SWD との間でデバッグプローブを使用することもできます。付録 A では、2 つの Pico を使用して、1 つ目の Pico の USB ポートを使用して 2 つ目の Pico の SWD ポートにアクセスする方法について説明しています。

第章 4. C 言語で「Hello World」というテキストを表示する

前の章では、LED を点滅させるプログラムを作成しました。この章では、シリアルポートを使用して「Hello World」というテキストを表示させるプログラムを作成します。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/hello_world/serial/hello_serial.c Lines 10 - 17

```
10 int main() {
11     stdio_init_all();
12     while (true) {
13         printf("Hello, world!\n");
14         sleep_ms(1000);
15     }
16     return 0;
17 }
```

4.1. Raspberry Pi Pico でのシリアル入力とシリアル出力

シリアル入力 (`stdin`) とシリアル出力 (`stdout`) は、シリアル UART に向けて行うことも、USB CDC (USB シリアル) に向けて行うこともできます。ただし、デフォルト設定の場合、`stdio` と `printf` の対象は、デフォルトの Raspberry Pi Pico UART0 になります。

デフォルトの UART0	物理ピン	GPIO ピン
GND	3	N/A
UART0_TX	1	GP0
UART0_RX	2	GP1

重要

デフォルトの Raspberry Pi Pico UART TX ピン (Raspberry Pi Pico から出力) はピン GP0 で、デフォルトの UART RX ピン (Raspberry Pi Pico に入力) はピン GP1 です。デフォルトの UART ピンは、ボード構成ファイルを使用してボードごとに設定します。Raspberry Pi Pico の構成は、 https://github.com/raspberrypi/pico-sdk/blob/master/src/boards/include/boards/pico.h で参照することができます。別のボードが指定されていない場合、SDK では Raspberry Pi Pico というボード名がデフォルトで使用されます。

SDK では、CMake を使用してビルドシステムが制御されます (第章 8 を参照)。また、`pico_stdlib` インターフェイスライブラリを使用して必要なソースファイルが集約され、各種の機能が提供されます。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/hello_world/serial/CMakeLists.txt

```
1 add_executable(hello_serial
2         hello_serial.c
3         )
4
5 # pull in common dependencies
6 target_link_libraries(hello_serial pico_stdlib)
7
8 # create map/bin/hex/uf2 file etc.
9 pico_add_extra_outputs(hello_serial)
```

```
10
11 # add url via pico_set_program_url
12 example_auto_set_url(hello_serial)
```

`stdout` の出力先は、CMake コマンドで変更することができます。出力先として、UART または USB CDC (あるいはその両方) を指定することができます。

```
pico_enable_stdio_usb(hello_world 1) ①
pico_enable_stdio_uart(hello_world 0) ②
```

1. USB CDC (USB シリアル) 経由での `printf` の出力を有効にします。
2. UART 経由での `printf` の出力を無効にします。

これにより、C 言語のソースコードを変更することなく、`stdio` の出力先を UART から USB に変更することができます。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/hello_world/usb/CMakeLists.txt

```
 1 if (TARGET tinyusb_device)
 2     add_executable(hello_usb
 3             hello_usb.c
 4             )
 5
 6     # pull in common dependencies
 7     target_link_libraries(hello_usb pico_stdlib)
 8
 9     # enable usb output, disable uart output
10     pico_enable_stdio_usb(hello_usb 1)
11     pico_enable_stdio_uart(hello_usb 0)
12
13     # create map/bin/hex/uf2 file etc.
14     pico_add_extra_outputs(hello_usb)
15
16     # add url via pico_set_program_url
17     example_auto_set_url(hello_usb)
18 elseif(PICO_ON_DEVICE)
19     message(WARNING "not building hello_usb because TinyUSB submodule is not initialized in the SDK")
20 endif()
```

4.2. 「Hello World」プログラムの作成

「Blink」プログラムの場合と同じように、`pico-examples/build` ツリー内の `hello_world` ディレクトリに移動して `make` コマンドを実行します。

```
$ cd hello_world
$ make -j4
Scanning dependencies of target ELF2UF2Build
[  0%] Creating directories for 'ELF2UF2Build'
   .
   .
[ 33%] Linking CXX executable hello_usb.elf
[ 33%] Built target hello_usb

   .
   .
[100%] Linking CXX executable hello_serial.elf
[100%] Built target hello_serial
```

これにより、hello_world/serial/ ディレクトリと hello_world/usb/ ディレクトリ内に 2 つのサンプルプログラムが作成されます。

また、以下のターゲットファイルが作成されます。

- serial/hello_serial.elf: このファイルは、デバッガーによって使用されます。
- serial/hello_serial.uf2: このファイルは、RP2040 USB マスストレージデバイス (UART シリアルバイナリ) 上にドラッグすることができます。
- usb/hello_usb.elf: このファイルは、デバッガーによって使用されます。
- usb/hello_usb.uf2: このファイルは、RP2040 USB マスストレージデバイス (USB シリアルバイナリ) 上にドラッグすることができます。

hello_serial の場合は、ピン GP0 とピン GP1 上の UART0 に stdio が送信され、hello_usb の場合は、USB CDC シリアルに stdio が送信されます。

警告

pico-sdk チェックアウト内の tinyusb サブモジュールが初期化されていない場合、SDK には USB 機能が含まれていないため、サンプルの USB CDC シリアルは機能しません。

4.3. 「Hello World」プログラムのフラッシュと実行

BOOTSEL ボタンを押しながら、Micro USB ケーブルを使用して、Raspberry Pi Pico を Raspberry Pi に接続します。これにより、強制的に USB マスストレージモードに切り替わります。接続されたら、BOOTSEL ボタンを離します。Raspberry Pi Desktop が稼働している場合、Raspberry Pi Pico が自動的に USB マスストレージデバイスとしてマウントされます。この状態で、hello_serial.uf2 ファイルまたは hello_usb.uf2 ファイルを、マスストレージデバイス上にドラッグすることができます。

RP2040 が再起動し、マスストレージデバイスのマウントが解除され、フラッシュされたコードの実行が開始されます。

「Hello World」プログラムは実行されていますが、この状態では、まだ「Hello World」というテキストは表示されません。テキストを表示するには、ホストコンピュータを Raspberry Pi Pico の適切な stdio インターフェイスに接続する必要があります。

4.4. USB で「Hello World」テキストを表示する

hello_usb.uf2 バイナリファイルを USB にドラッグアンドドロップすると、「Hello World」テキストが USB シリアルに送信されます。

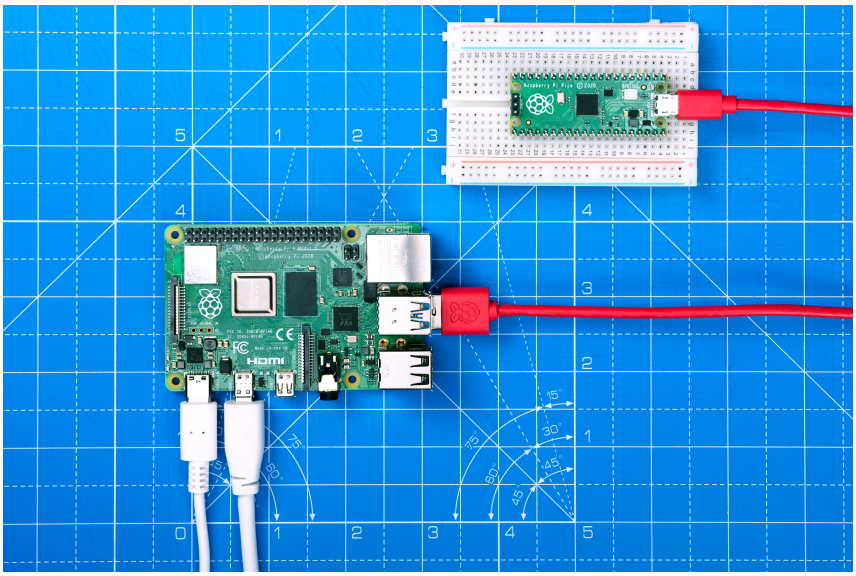


図 2. USB 経由で Raspberry Pi を Raspberry Pi Pico に接続する

USB を使用して Raspberry Pi Pico を Raspberry Pi に直接接続し (図 2 を参照)、次に以下のコマンドを実行して `minicom` をインストールします。

```
$ sudo apt install minicom
```

以下のコマンドを実行して、シリアルポートを開きます。

```
$ minicom -b 115200 -o -D /dev/ttyACM0
```

これにより、「`Hello, world!`」テキストがコンソールに表示されます。

ヒント

minicom を終了するには、`CTRL + A` キーを押してから `X` キーを押します。

注記

SWD を使用してデバッグを行う場合は (第章 6 を参照)、UART ベースのシリアル接続を使用する必要があります。これは、デバッグ中に RP2040 コアが停止すると USB スタックが一時停止し、その結果として USB デバイスが切断されてしまうためです。

4.5. UART で「Hello World」テキストを表示する

`hello_serial.uf2` バイナリファイルを USB にドラッグアンドドロップすると、「Hello World」テキストが、ピン GP0 とピン GP1 の UART0 に送信されます。このテキストを表示するには、最初に Raspberry Pi のホストで UART シリアル通信を有効にする必要があります。これを行うには、`raspi-config` コマンドを実行します。

```
$ sudo raspi-config
```

次に、「Interfacing Options」→「Serial」に移動し、「Would you like a login shell to be accessible over serial?」が表示された場合は「No」を選択し、「Would you like the serial port hardware to be enabled?」が表示された場合は「Yes」を選択します。これにより、図 3 のような画面が表示されます。

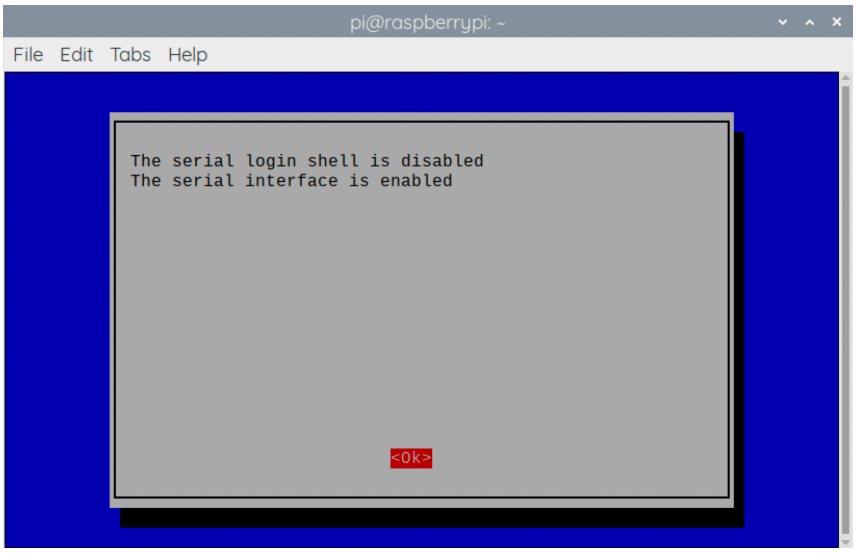


図 3. Raspberry Pi で raspi-config コマンドを実行して UART シリアル通信を有効にする

この画面で「OK」を選択して Raspberry Pi を再起動すると、シリアルポートが有効になります。``

次に、以下のマッピングに従い、Raspberry Pi と Raspberry Pi Pico を接続する必要があります。

Raspberry Pi	Raspberry Pi Pico
GND (ピン 14)	GND (ピン 3)
GPIO15 (UART_RX0、ピン 10)	GP0 (UART0_TX、ピン 1)
GPIO14 (UART_TX0、ピン 8)	GP1 (UART0_RX、ピン 2)

図 4 を参照してください。

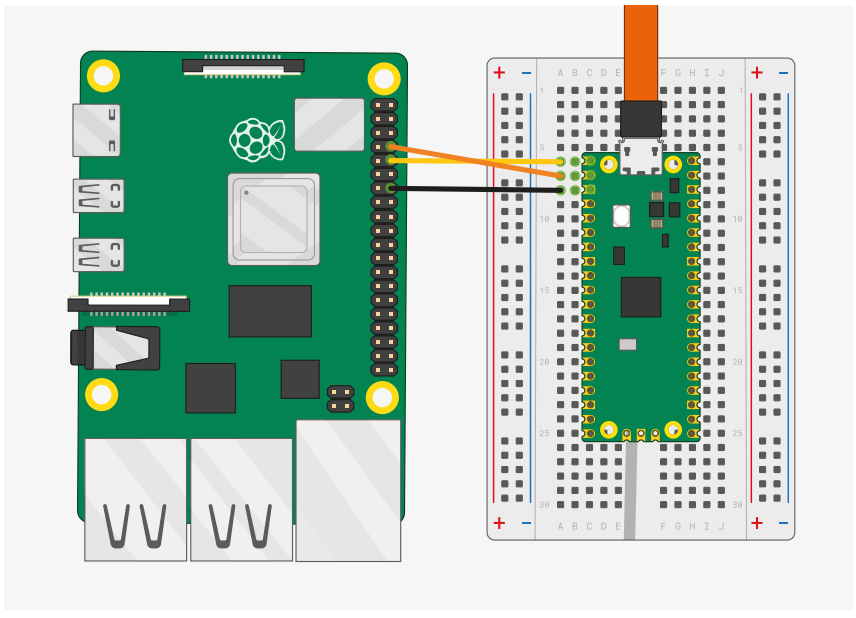


図 4. UART0 を使用して、Raspberry Pi 4 と Raspberry Pi Pico を接続する

接続が完了したら、以下のコマンドで `minicom` をインストールします (まだインストールされていない場合)。

```
$ sudo apt install minicom
```

以下のコマンドを実行して、シリアルポートを開きます。

```
$ minicom -b 115200 -o -D /dev/serial0
```

これにより、「`Hello, world!`」テキストがコンソールに表示されます。

ヒント

minicom を終了するには、`CTRL + A` キーを押してから `X` キーを押します。

4.6. ボードの電源を入れる

USB から Raspberry Pi Pico を取り外し、ダイオード経由で Raspberry Pi の 5V ピンを Raspberry Pi Pico の VSYS ピンに接続すると、ボードの電源がオンになります (図 5 を参照)。 ショットキーダイオードを使用することをお勧めします。

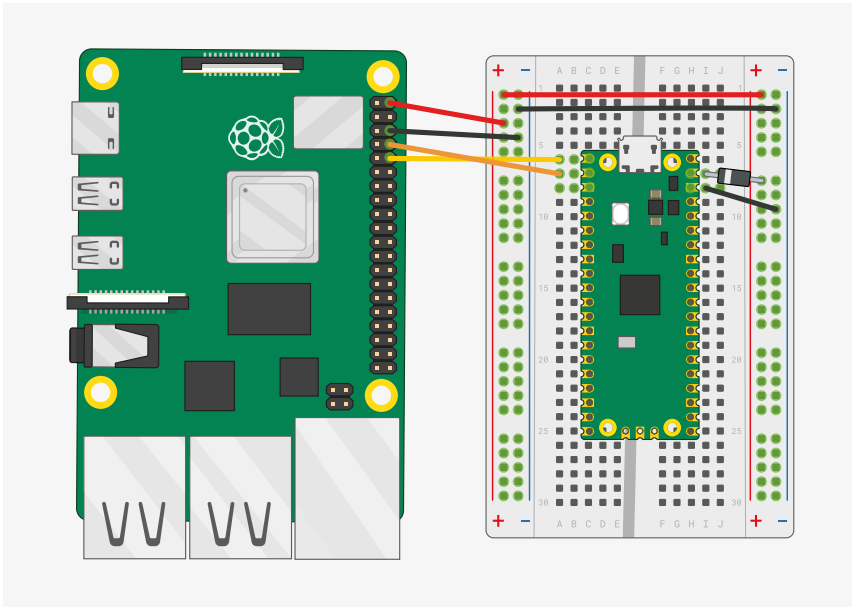


図 5. GPIO ピンだけを使用して、Raspberry Pi と Raspberry Pi Pico を接続する

Raspberry Pi の 5V ピンを Raspberry Pi Pico の VBUS ピンに接続することはできますが、これはお勧めしません。5V レールを短絡させると、Micro USB が使用できなくなります。ただし、Raspberry Pi Pico を USB ホストモードで使用する場合は例外です。この場合は、VBUS ピンに 5V ピンを接続する必要があります。

3.3V ピンは、Raspberry Pi Pico の出力ピンです。このピンを使用して Raspberry Pi Pico に電源を供給することはできないため、このピンを電源に接続しないでください。

Raspberry Pi Pico の電源について詳しくは、「RP2040 を使用したハードウエア設計」の「Power」セクションを参照してください。

第章 5. SWD を使用してフラッシュプログラミングを行う

シリアルワイヤーデバッグ (SWD) は、Cortex-M ベースのマイクロコントローラーの標準インターフェイスです。コード開発用のマシン (通常は「ホスト」といいます) を使用して、ボードのリセット、フラッシュへのコードの読み込み、コード実行の設定を行うことができます。Raspberry Pi Pico では、ボード下端の 3 つのピンを使用して RP2040 SWD インターフェイスが機能します。SWD ポート経由で、ホストから RP2040 の内部へいつでもアクセスできるため、手動でボードをリセットしたり、BOOTSEL ボタンを長押ししたりする必要はありません。

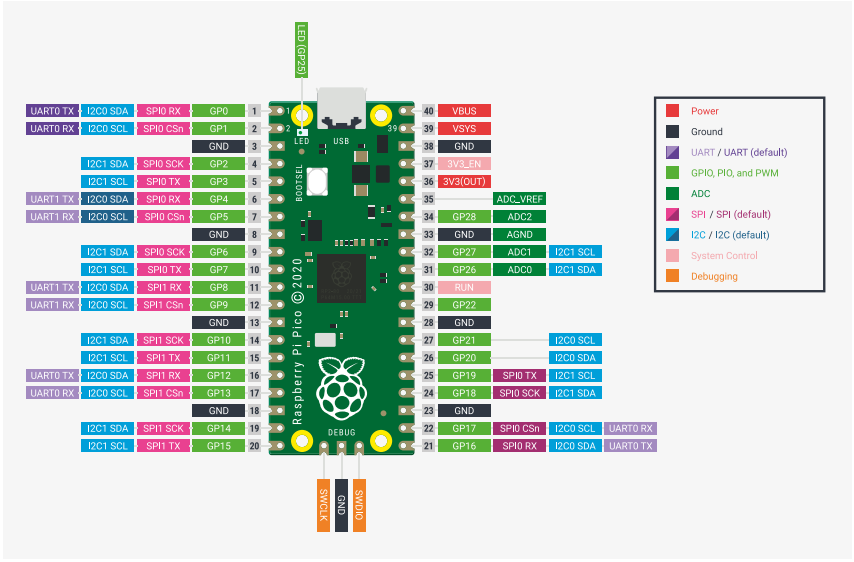


図 6. この図は、Pico のピンの配置を示しています。この図の最下部にあるのが SWD ポートです。ホストと Pico 間の信号の品質 (シグナルインテグリティ) を良好な状態に保つには、アース (GND) 接続を行う必要があります。SWDIO ピンにより、RP2040 とホストとの間で双方向のデバッグトラフィックが送信されます。SWCLK ピンにより、接続の同期が確保されます。これらのピンは、RP2040 の専用 SWD インターフェイスに接続されているため、SWD ポートを使用しても GPIO に影響することはありません。

Raspberry Pi では、Pico の SWD ポートに Pi の GPIO を直接接続し、そこからコードを読み込むことができます。その他のマシンでは、ホストマシン上の接続部 (USB ポートなど) と Pico 上の SWD ピンを連結するためのハードウェア (デバッグプローブ) が必要になります。これを行うための最も安価な方法の 1 つは、別の Pico をデバッグプローブとして使用するという方法です。これについては、付録 A で説明します。

この章では、マシンを Raspberry Pi Pico の SWD ポートに接続し、このマシンを使用してフラッシュにプログラムを書き込んで実行する方法について説明します。

ヒント

Visual Studio Code (第章 7) などの IDE を使用する場合、バックグラウンドで自動的に SWD を使用するように設定できるため、再生ボタンをクリックすると、自分のマシンでネイティブコードを実行する場合と同じようにコードが実行されます。

注記

また、SWD を使用して、ブレークポイントの設定、1 行ごとのコード実行、IO レジスターへのアクセスなど、インタラクティブなデバッグ操作を実行することができます。これらの操作は、マシンから直接実行することができます。RP2040 用のソフトウェアを作成する必要はありません。これについては、第章 6で説明します。

5.1. OpenOCD のインストール

マイクロコントローラーの SWD ポートにアクセスするには、ホストマシン上に「デバッグトランスレーター」というプログラムをインストールする必要があります。このプログラムにより、SWD プロトコルが認識され、マイクロコントローラー内のプロセッサー (RP2040 の場合は 2 つの Cortex-M0+) が制御されます。また、SWD ポートに接続されているデバッグプローブとの通信方法と、デバイス上のフラッシュのプログラミング方法も認識されます。

このセクションでは、OpenOCD というデバッグトランスレーターのインストール方法について説明します。

ヒント

Raspberry Pi 上で `pico-setup` スクリプトを実行すると (第章 1を参照)、OpenOCD がインストールされます。すでにインストールされている場合は、次のセクションに進んでください。

注記

このセクションでは、`/home/pi/pico/openocd` ディレクトリ内に OpenOCD をインストールするという前提になっています。

```
$ cd ~/pico
$ sudo apt install automake autoconf build-essential texinfo libtool libftdi-dev libusb-1.0-0-dev
$ git clone https://github.com/raspberrypi/openocd.git --branch rp2040 --recursive --depth=1
$ cd openocd
$ ./bootstrap
$ ./configure --enable-ftdi --enable-sysfsgpio --enable-bcm2835gpio
$ make -j4
$ sudo make install
```

ターミナルで `openocd` としてこのコマンドを実行すると、OpenOCD がインストールされます。

注記

macOS の場合、Homebrew を使用して `texinfo` の新しいバージョンをインストールしなければならない場合があります。

5.2. SWD ポートの配線

SWD 経由で RP2040 のコードをプログラミングして実行するには、SWD ポートにワイヤを接続する必要があります。

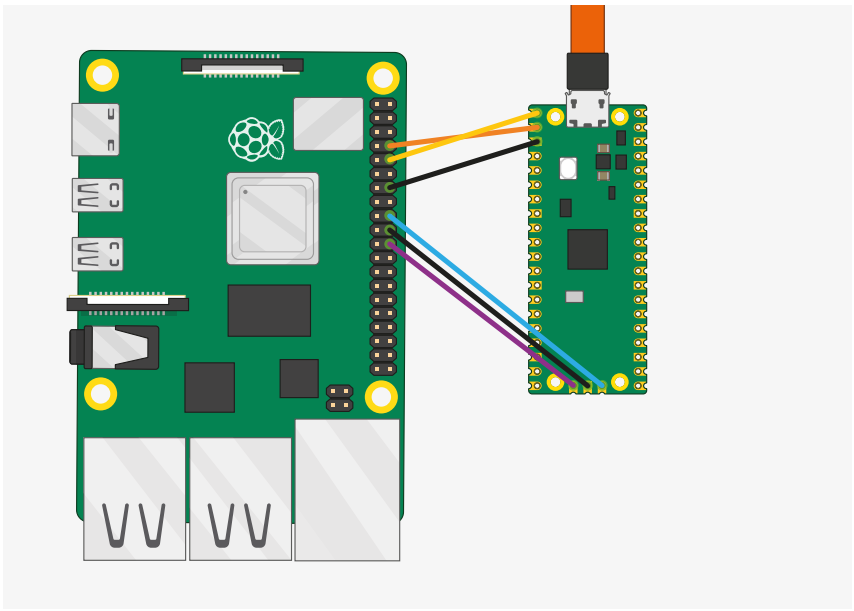


図 7. この図は、UART と SWD ポートが搭載された Raspberry Pi 4 と Raspberry Pi Pico をワイヤで接続した状態を示しています。ブレッドボードを使用することなく、Raspberry Pi 4 と Raspberry Pi Pico が直接接続されています。SWD へのアクセスで必要になるのは、下部の 3 本のワイヤだけです。オプションとして、上部の 3 本のワイヤで Pi の UART を接続し、Pico のシリアルポートに直接アクセスすることもできます。

デフォルト設定の場合、Pi の GPIO 24 に SWDIO、GPIO 25 に SWCLK が配置されます。以下のマッピングに従い、Raspberry Pi を Raspberry Pi Pico に配線することができます。

Raspberry Pi	Raspberry Pi Pico
GND (ピン 20)	SWD GND
GPIO24 (ピン 18)	SWDIO
GPIO25 (ピン 22)	SWCLK

図 7 を参照してください。

ヒント

Picoprobe (付録 A を参照) などのデバッグプローブを使用する場合、そのプローブの GND ピン、SWCLK ピン、SWDIO ピンを Raspberry Pi Pico (または、別の RP2040 ベースのボード) 上の対応するピンに接続する必要があります。

可能であれば、Raspberry Pi を SWD ポートに直接接続することをお勧めします。ブレッドボードなどの間接的な方法で SWD ポートを配線すると、シグナルインテグリティが大幅に低下し、コードの読み込み動作が不安定になったり、読み込みが失敗したりする可能性があります。その際、Raspberry Pi と SWD ポートをアースワイヤ (0v) で直接接続することが重要です。

デバッグを行う場合は、USB などを使用して Raspberry Pi Pico にも電源を供給する必要があります。マルチドロップ SWD を機能させるには、OpenOCD ブランチをビルドする必要があります。

5.3. プログラムの読み込み

OpenOCD を使用する前提条件として、プログラムのバイナリファイルが BOOTSEL モードで使用される `.uf2` ファイル形式ではなく、`.elf` ファイル形式 (リンク可能な実行可能ファイル形式) になっている必要があります。SDK では、両方のファイル形式がデフォルトでビルドされますが、これらのファイル形式を混在させないことが重要です。

第章 3の手順に従ってサンプルの `blink` プログラムがビルドされている場合、以下のコマンドにより、SWD 経由で `.elf`

ファイルをプログラミングして実行することができます。

```
$ openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg -c "program blink/blink.elf verify reset exit"
```

このコマンドには多くの引数が指定されているため、それぞれの引数について説明します。

`-f interface/raspberrypi-swd.cfg`
: これは、Raspberry Pi の GPIO ピンを使用して SWD ポートにアクセスするための引数です。Picoprobe (付録 A を参照) などの外部 USB→SWD プローブを使用する場合は、そのプローブをこの `interface` 引数で指定する必要があります。

`-f target/rp2040.cfg`
: これは、RP2040 ベースのボードに接続するための引数です。この `.cfg` ファイルには、プロセッサーの種類 (Cortex-M0+) やフラッシュメモリーへのアクセス方法など、OpenOCD に関する情報が保管されています。

`-c`
: これは、コマンドラインに入力した一連のコマンドを OpenOCD に直接渡すための引数です。OpenOCD には対話型のターミナルインターフェイスも付属しているため、コマンドラインではなくターミナルインターフェイスでコマンドを入力することもできます。使用するコマンドは、次のとおりです:

`program blink/blink.elf`
: この引数を指定すると、`.elf` ファイルがフラッシュに書き込まれ、必要に応じて最初にフラッシュのターゲット領域が消去されます。`.elf` ファイルには、OpenOCD の各種パーツを読み込む場所や、それらのパーツのサイズなどに関する情報が保管されています。

`verify`
: これは、プログラミング後にフラッシュからデータを読み取り、プログラミングが正常に実行されたかどうかを確認するための引数です。

`reset`
: この引数は、RP2040 を電源投入直後の状態に初期化して、コードの実行準備を整えるための引数です。

`exit`
: これは、RP2040 から接続を解除して処理を終了するための引数です。OpenOCD を切断すると、プログラミングされたコードが実行されます。

ヒント

OpenOCD が SWD インターフェイス上の RP2040 を認識できなかった場合は、「`Info: DAP init failed`」などのエラーが表示されます。このエラーの最も一般的な原因としては、USB ケーブルなどを使用してボードに電源が供給されていない、SWD の配線が正しくない (アースワイヤが接続されていない、SWDIO と SWCLK の位置が逆になっているなど)、ジャンパーワイヤの長さや緩みが原因でシグナルインテグリティに何らかの問題が発生している、などが考えられます。

修正後のプログラムが正しく読み込まれているかどうかを確認するには、`blink/blink.c` を以下のように修正して LED の点滅速度を上げ、その後プログラムの再ビルドを行い、上記の `openocd` コマンドをもう一度実行します。

```
int main() {
    const uint LED_PIN = 25;
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (true) {
        gpio_put(LED_PIN, 1);
        // Blink faster! (this is the only line that's modified)
        sleep_ms(25);
```

```
        gpio_put(LED_PIN, 0);
        sleep_ms(250);
    }
}
```

次に、以下のコマンドを実行します。

```
$ cd pico-examples/build
$ make blink
# (the application is rebuilt)
$ openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg -c "program blink/blink.elf verify reset exit"
```

第章 6. SWD を使用してデバッグを行う

RP2040 ベースのボード (Raspberry Pi Pico など) で SWD ポートを使用すると、ボードのリセット、コードの読み込み、コードの実行だけでなく、読み込んだプログラムを対話方式でデバッグすることもできます。以下のようなデバッグ操作を実行することができます。

- コード内にブレークポイントを設定する
- コードを 1 行ずつ実行していく
- プログラム内のさまざまな位置で変数の値を確認する

第章 5では、OpenOCD をインストールして、Raspberry Pi Pico の SWD ポートにアクセスする方法について説明しました。対話方式でコードをデバッグするには、汎用的な GNU デバッガーである GDB などのデバッガーが必要になります。

SDK では、詳細に最適化されたプログラムがデフォルトでビルドされますが、このプログラムは制御フローやデータフローの点において、最初に作成したプログラムとは大きく異なっている場合があります。そのため、対話形式でコードをステップごとに実行しようとすると、混乱する可能性があります。このような場合は、それほど最適化されていないデバッグ用のプログラムを作成し、実際のデバイス上の制御フローをソースコード上の制御フローに近づけることをお勧めします。

6.1. 「Hello World」プログラムのデバッグ版をビルドする

警告

SWD を使用してデバッグを行う場合は、UART ベースのシリアル接続 (第章 4を参照) を使用する必要があります。これは、デバッグ中に RP2040 コアが停止すると USB スタックが一時停止し、その結果として USB デバイスが切断されてしまうためです。デバッグ中は、USB CDC のシリアル接続を使用することはできません。

以下のように `CMAKE_BUILD_TYPE=Debug` を指定することにより、「Hello World」プログラムのデバッグ版をビルドすることができます。

```
$ cd ~/pico/pico-examples/
$ rm -rf build
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ cd hello_world/serial
$ make -j4
```

6.2. GDB のインストール

以下のコマンドを実行して `gdb-multiarch` をインストールします。

```
$ sudo apt install gdb-multiarch
```

6.3. GDB と OpenOCD を使用して Hello World プログラムをデバッグする

Raspberry Pi 4 と Raspberry Pi Pico が正しく配線されていれば、`raspberrypi-swd` インターフェイスを使用して、OpenOCD をチップに接続することができます。

```
$ openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg
```

上記のコマンドを実行すると、以下のような出力が表示されます。

```
...
Info : rp2040.core0: hardware has 4 breakpoints, 2 watchpoints
Info : rp2040.core1: hardware has 4 breakpoints, 2 watchpoints
Info : starting gdb server for rp2040.core0 on 3333
Info : Listening on port 3333 for gdb connections
```

警告

「`Info : DAP init failed`」などのエラーが表示された場合、Raspberry Pi Pico の電源が切れている、配線が正しくない、シグナルインテグリティに問題がある、などの原因が考えられます。このようなエラーが表示された場合は、別の GPIO ジャンパーケーブルを使用してください。

使用する OpenOCD ターミナルは、稼働状態のままにしておく必要があります。次に、別のターミナルを起動し、OpenOCD に gdb インスタンスを接続します。Hello World プログラムのサンプルコードに移動し、コマンドラインで `gdb` を起動します。

```
$ cd ~/pico/pico-examples/build/hello_world/serial
$ gdb-multiarch hello_serial.elf
```

GDB を OpenOCD に接続します。

```
(gdb) target remote localhost:3333
```

ヒント

`.gdbinit` ファイルを作成すれば、毎回「`target remote localhost:3333`」と入力する必要はありません。このファイルを作成するには、`echo "target remote localhost:3333" > ~/.gdbinit` コマンドを実行します。ただし、このファイルを作成すると、VSCode でのデバッグ (第章 7を参照) に影響します。

`hello_serial.elf` ファイルを Falsh に読み込みます。

```
(gdb) load
Loading section .boot2, size 0x100 lma 0x10000000
Loading section .text, size 0x22d0 lma 0x10000100
Loading section .rodata, size 0x4a0 lma 0x100023d0
Loading section .ARM.exidx, size 0x8 lma 0x10002870
Loading section .data, size 0xb94 lma 0x10002878
Start address 0x10000104, load size 13324
Transfer rate: 31 KB/sec, 2664 bytes/write.
```

読み込んだファイルを実行します。

```
(gdb) monitor reset init
(gdb) continue
```

重要

OpenOCD 経由で Raspberry Pi Pico にバイナリを読み込もうとしたときに、「`Error finishing flash operation`」や「`Error erasing flash with vFlashErase packet`」などのエラーが GDB で表示された場合は、Raspberry Pi と Raspberry Pi Pico との間のシグナルインテグリティが低下している可能性があります。Pi と Pico の SWD が直接接続されていない場合は、直接接続してください (図 7 を参照)。または、Pi と Pico が正しく接続されるまで、`raspberrypi-swd.cfg` 設定ファイル内の `adapter_khz` の値を半減してみてください。ボード間のビットバンギングにおいては、タイミングよりもシグナルインテグリティの方が重要です。シグナルインテグリティが低下すると、エラーが発生する可能性が高くなります。

実行可能ファイルを実行する前に、`main()` でブレークポイントを設定する場合は、以下のように入力します。

```
(gdb) monitor reset init
(gdb) b main
(gdb) continue

Thread 1 hit Breakpoint 1, main () at /home/pi/pico/pico-examples/hello_world/serial/hello_serial.c:11
11        stdio_init_all();
```

ブレークポイントに到達すると、プログラムが停止します。プログラムを続行するには、以下のように入力します。

```
(gdb) continue
```

`gdb` を終了するには、以下のように入力します。

```
(gdb) quit
```

`gdb` の詳しい使用方法については、 https://www.gnu.org/software/gdb/documentation/ を参照してください。

第章 7. Visual Studio Code を使用する

Visual Studio Code (VSCode) は、Microsoft が開発したオープンソースエディターで、多くのユーザーが使用しています。VSCode は、Raspberry Pi 4 の推奨統合開発環境 (IDE) です。グラフィカルなインターフェイス上でコードの編集やデバッグを行う場合は、VSCode を使用することをお勧めします。

7.1. Visual Studio Code をインストールする

重要

以下に記載されているインストール手順は、コマンドラインツールチェーンのダウンロードとインストールが完了していること (第章 3を参照)、OpenOCD を使用してボードが SWD に接続されていること (第章 5を参照)、コマンドラインでのデバッグ用に GDB がセットアップされていること (第章 6を参照) が前提条件になっています。

Raspberry Pi 用の Visual Studio Code の ARM バージョンは、 https://code.visualstudio.com/Download からダウンロードすることができます。32 ビット版の OS (Raspberry Pi のデフォルト OS など) を使用している場合は、`ARM .deb` ファイルをダウンロードし、64 ビット版の OS を使用している場合は、`ARM 64 .deb` ファイルをダウンロードしてください。ダウンロードが完了したら .deb パッケージをダブルクリックし、画面の指示に従ってインストールを実行してください。

コマンドラインを使用して、ダウンロードした `.deb` パッケージをインストールします。これを行うには、`cd` コマンドを実行して、.deb パッケージをダウンロードしたフォルダーに移動し、`dpkg -i <ダウンロードしたファイル名.deb>` と入力してインストールを実行します。

インストールが完了したら、Raspberry Pi Pico でのデバッグに必要な拡張機能をインストールします。

```
$ code --install-extension marus25.cortex-debug
$ code --install-extension ms-vscode.cmake-tools
$ code --install-extension ms-vscode.cpptools
```

最後に、ターミナルウィンドウから Visual Studio Code を起動します。

```
$ export PICO_SDK_PATH=/home/pi/pico/pico-sdk
$ code
```

Visual Studio Code で SDK を認識できるように、`PICO_SDK_PATH` 環境変数を設定する必要があります。

注記

`PICO_SDK_PATH` 環境変数がシェル環境内でデフォルト設定されていない場合は、新しいターミナルウィンドウを開くたびにこの変数を設定して、VSCode を起動する必要があります (または、メニューから VSCode を起動するたびに、この変数を設定する必要があります)。そのため、この環境変数を `.profile` ファイルまたは `.bashrc` ファイルに追加しておくことをお勧めします。

注記

「View」>「Command Palette」>「C/C++: Change Configuration Provider…」>「CMake Tools」でプロバイダーを切り替えることにより、CMake コマンド用の IntelliSense 機能を設定することができます。

7.2. プロジェクトを読み込む

エクスプローラーのツールバー (`Ctrl + Shift + E`) で「フォルダーを開く」を選択し、ファイルのポップアップ画面で `/home/pi/pico/pico-examples` に移動して、`pico-examples` フォルダーを開きます。次に「OK」をクリックして、このフォルダーを VSCode に読み込みます。

CMake Tools 拡張機能がインストールされている場合、1～2 秒後に、VSCode ウィンドウの右下にポップアップが表示されます。

「Yes」を押すと、プロジェクトが設定されます。その後、コンパイラーの選択画面が表示されます (図 8 を参照)。

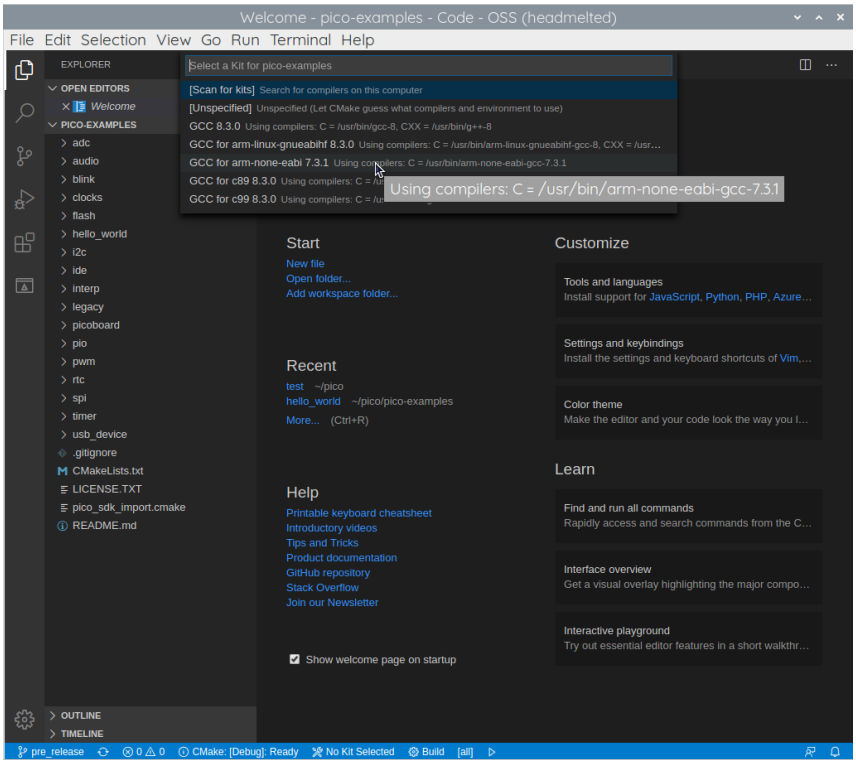


図 8. プロジェクトに適したコンパイラーの選択画面

ドロップダウンメニューで、「`GCC for arm-none-eabi`」を選択する必要があります。

ヒント

このポップアップは、数秒経過すると非表示になります。このポップアップを見逃した場合は、VSCode ウィンドウ下部の青いバーにある「No Kit Selected」をクリックすると、コンパイラーの選択画面が表示されます。

次に、ウィンドウ下部の青いバーにある「Build」ボタンをクリックして `pico-examples` フォルダー内のすべてのサンプルをビルドするか、青いバーの「[all]」の部分をクリックします。プロジェクトを選択するためのドロップダウンが表示されます。ここでは「hello_usb」と入力し、「Hello USB」という実行可能ファイルを選択します。この場合、VSCode によって「Hello USB」というサンプルだけがビルドされるため、コンパイル時間が短縮されます。

ヒント

ウィンドウ下部の青いバーで「CMake: [Debug]: Ready」をクリックすると、ビルドする実行ファイルとして「Debug」と「Release」を切り替えることができます。デフォルト設定の場合、SWD デバッグに対応した「Debug」実行ファイルがビルドされます。

ウィンドウ下部の青いバーにある「Build」ボタン (歯車が付いているボタン) をクリックします。これにより、ビルドディレクトリが作成され、ビルドが起動する前に、Section 3.1 で手動で実行した CMake コマンドが自動的に実行されます (図 9 を参照)。

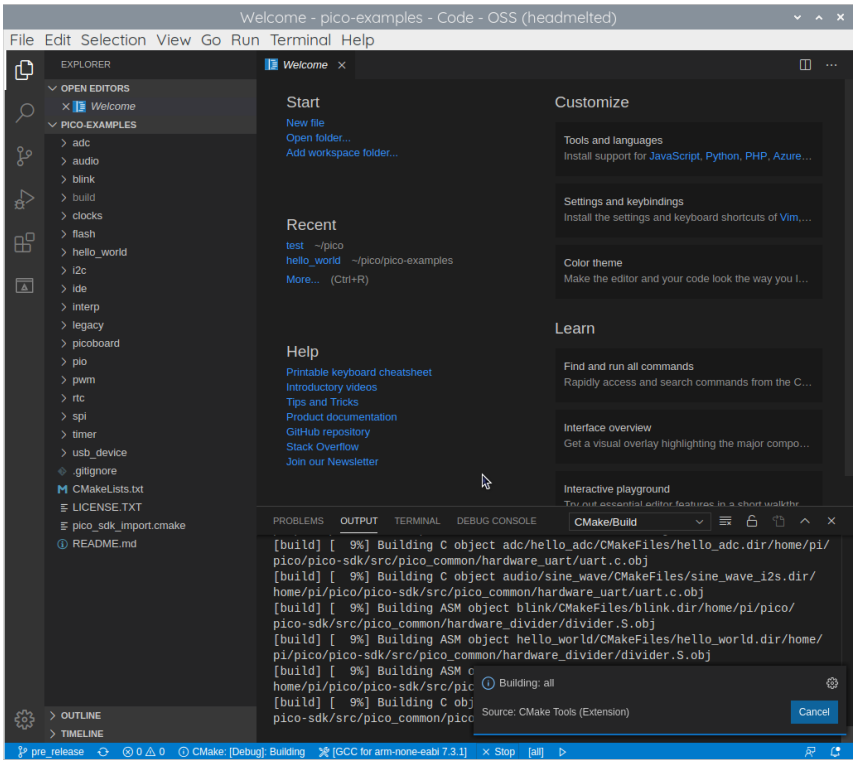


図 9. Visual Studio Code で `pico-examples` プロジェクトをビルドする

また、コマンドラインを使用した場合と同様に、以下のターゲットファイルが作成されます。

- `hello_usb.elf`: このファイルは、デバッガーによって使用されます。
- `hello_usb.uf2`: このファイルは、RP2040 の USB マスストレージデバイス上にドラッグすることができます。

7.3. プロジェクトのデバッグを行う

pico-examples リポジトリには、サンプルのデバッグ設定情報が保管されています。この設定情報により、OpenOCD が起動して GDB が接続され、CMake によってビルドされるアプリケーションが起動します。launch-raspberrypi-swd.json ファイルのコピーを launch.json というファイル名で作成し、pico-examples/.vscode ディレクトリに保存してください。また、settings.json ファイルもコピーすることをお勧めします。この settings.json ファイルにより、混乱する可能性のあるオプション (ホスト上で Pico バイナリを実行しようとする破損した「Debug」ボタンや「Run」ボタンなど) が CMake プラグインから削除されます。

```
$ cd ~/pico/pico-examples
$ mkdir .vscode
$ cp ide/vscode/launch-raspberrypi-swd.json .vscode/launch.json
$ cp ide/vscode/settings.json .vscode/settings.json
```

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/ide/vscode/launch-raspberrypi-swd.json

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Pico Debug",
            "cwd": "${workspaceRoot}",
            "executable": "${command:cmake.launchTargetPath}",
            "request": "launch",
            "type": "cortex-debug",
            "servertype": "openocd",
            // This may need to be arm-none-eabi-gdb depending on your system
            "gdbPath" : "gdb-multiarch",
            "device": "RP2040",
            "configFiles": [
                "interface/raspberrypi-swd.cfg",
                "target/rp2040.cfg"
            ],
            "svdFile": "${env:PICO_SDK_PATH}/src/rp2040/hardware_regs/rp2040.svd",
            "runToMain": true,
            // Work around for stopping at main on restart
            "postRestartCommands": [
                "break main",
                "continue"
            ]
        }
    ]
}
```

注記

gdb-multiarch の代わりに arm-none-eabi-gdb という gdb を使用する場合は、launch.json ファイル内の gdbPath を修正しなければならないことがあります。

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/ide/vscode/settings.json

```
{
    // These settings tweaks to the cmake plugin will ensure
    // that you debug using cortex-debug instead of trying to launch
    // a Pico binary on the host
    "cmake.statusbar.advanced": {
        "debug": {
            "visibility": "hidden"
        },
        "launch": {
            "visibility": "hidden"
        },
        "build": {
            "visibility": "hidden"
        },
        "buildTarget": {
```

```
16             "visibility": "hidden"
17         }
18     },
19     "cmake.buildBeforeRun": true,
20     "C_Cpp.default.configurationProvider": "ms-vscode.cmake-tools"
21 }
```

7.3.1. Raspberry Pi Pico で「Hello USB」プログラムを実行する

重要

サンプルの「Hello USB」コードがデバッグバイナリとしてビルドされていることを確認してください (`CMAKE_BUILD_TYPE=Debug`)。

デバッグツールバーにアクセスし (`Ctrl + Shift + D`)、左側のウィンドウペインの上部にある小さな緑の矢印 (再生ボタン) をクリックすると、Raspberry Pi Pico にコードが読み込まれ、デバッグが開始されます。

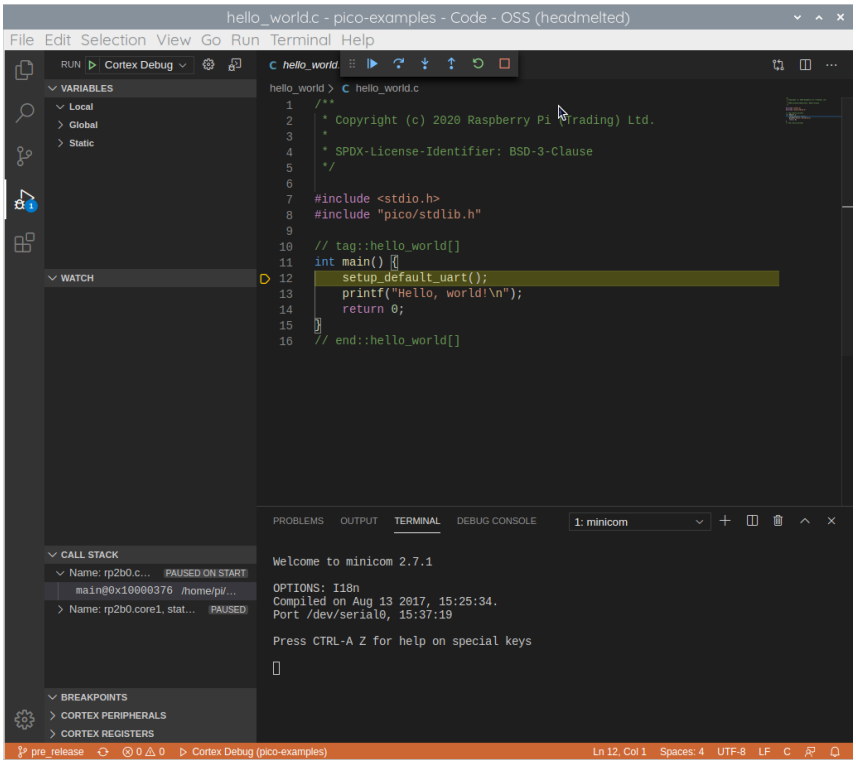


図 10. Visual Studio Code で「Hello USB」バイナリをデバッグする

コードが Raspberry Pi Pico に読み込まれると、ウィンドウ右上のメインペインに「Hello USB」のソースコードが表示されます。最初のブレークポイントに達するまで、コードが実行されます。ブレークポイントは、`launch.json` ファイル内の `runToMain` ディレクティブによって有効になります。ソースコードのメインウインドウ上部にある小さな青い矢印 (再生ボタン) をクリックすると、コードの実行が再開されます (`F5` キーを押してもかまいません)。

ヒント

`hello_usb.c` コードの下にある「Terminal」タブに切り替えると (右下のペイン)、VSCode 内で `minicom` を開いて以下のように入力することにより、「Hello USB」サンプルの UART 出力を表示することができます。

```
$ minicom -b 115200 -o -D /dev/ttyACM0
```

このコマンドは、これまでと同様にターミナルプロンプトで入力します (Section 4.4 を参照)。

第章 8. プロジェクトを作成する

pico-sdk ディレクトリとともに、テストプロジェクトを保管するディレクトリを作成します。

```
$ ls -la
total 16
drwxr-xr-x   7 aa  staff   224  6 Apr 10:41 ./
drwx------@ 27 aa  staff   864  6 Apr 10:41 ../
drwxr-xr-x  10 aa  staff   320  6 Apr 09:29 pico-examples/
drwxr-xr-x  13 aa  staff   416  6 Apr 09:22 pico-sdk/
$ mkdir test
$ cd test
```

次に、そのディレクトリ内に test.c ファイルを作成します。

```
 1 #include <stdio.h>
 2 #include "pico/stdlib.h"
 3 #include "hardware/gpio.h"
 4 #include "pico/binary_info.h"
 5
 6 const uint LED_PIN = 25;
 7
 8 int main() {
 9
10     bi_decl(bi_program_description("This is a test binary."));①
11     bi_decl(bi_1pin_with_name(LED_PIN, "On-board LED"));
12
13     stdio_init_all();
14
15     gpio_init(LED_PIN);
16     gpio_set_dir(LED_PIN, GPIO_OUT);
17     while (1) {
18         gpio_put(LED_PIN, 0);
19         sleep_ms(250);
20         gpio_put(LED_PIN, 1);
21         puts("Hello World\n");
22         sleep_ms(1000);
23     }
24 }
```

① これらの行により、picotool を使用して文字列がバイナリに追加されます (付録 B を参照)。

次に、CMakeLists.txt ファイルを作成します。

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(test
    test.c
)

pico_enable_stdio_usb(test 1)①
pico_enable_stdio_uart(test 1)②

pico_add_extra_outputs(test)

target_link_libraries(test pico_stdlib)
```

1. これにより、USB 経由でのシリアル出力が有効になります。

2. これにより、UART 経由でのシリアル出力が有効になります。

次に、`pico_sdk_import.cmake` ファイルを `pico-sdk` インストール環境の `external` フォルダーからテストプロジェクトフォルダーにコピーします。

```
$ cp ../pico-sdk/external/pico_sdk_import.cmake .
```

以下のような出力が表示されます。

```
$ ls -la
total 24
drwxr-xr-x  5 aa  staff   160  6 Apr 10:46 ./
drwxr-xr-x  7 aa  staff   224  6 Apr 10:41 ../
-rw-r--r--@ 1 aa  staff   394  6 Apr 10:37 CMakeLists.txt
-rw-r--r--  1 aa  staff  2744  6 Apr 10:40 pico_sdk_import.cmake
-rw-r--r--  1 aa  staff   383  6 Apr 10:37 test.c
```

サンプルコードの「Hello World」の場合と同様にビルドすることができます。

```
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake ..
$ make
```

make プロセスにより、さまざまなファイルが作成されます。以下の表に、重要なファイルを示します。

ファイル拡張子	説明
.bin	プログラムコードとデータの未加工バイナリダンプを保管するためのファイル
.elf	プログラムのすべての出力 (デバッグ情報など) を保管するためのファイル
.uf2	UF2 形式のプログラムコードとデータを保管するためのファイル (UF2 形式のプログラムコードとデータは、USB ドライブとしてマウントされた RP2040 ボードにドラッグアンドドロップすることができます)
.dis	コンパイルされたバイナリの逆アセンブルデータを保管するためのファイル
.hex	コンパイルされたバイナリの 16 進数ダンプを保管するためのファイル
.map	リンカーによるメモリー内でのセグメント配置が記述された .elf ファイルに付随するマップファイル

注記

UF2 (USB Flashing Format) は、Microsoft が開発した、USB 経由で RP2040 ボードをフラッシュするためのファイル形式です。詳細については、次のリンクを参照してください: Microsoft UF2 の仕様

注記

フラッシュメモリーではなく SRAM で動作するバイナリをビルドする場合は、`-DPICO_NO_FLASH=1` を使用して `cmake` ビルドを設定することも、`pico_set_binary_type(TARGET_NAME no_flash)` を追加して、`CMakeLists.txt` ファイルでバイナリごとに制御することもできます。 UF2 経由で、RAM バイナリを RP2040 にダウンロードすることができます。たとえば、ボード上にフラッシュチップがない場合、データが通過する場所のアドレスを指定する UF2 を使用して、チップ上の RAM で動作するバイナリをダウンロードすることができます。ダウンロードできるのは RAM と FLASH のいずれか一方だけで、両方をダウンロードすることはできません。

8.1. プロジェクトのデバッグ

コマンドラインを使用してプロジェクトをデバッグする方法は、Section 6.3 で説明した「Hello World」サンプルをデバッグする方法と同じです。図 11 のように、Raspberry Pi と Raspberry Pi Pico を接続します。

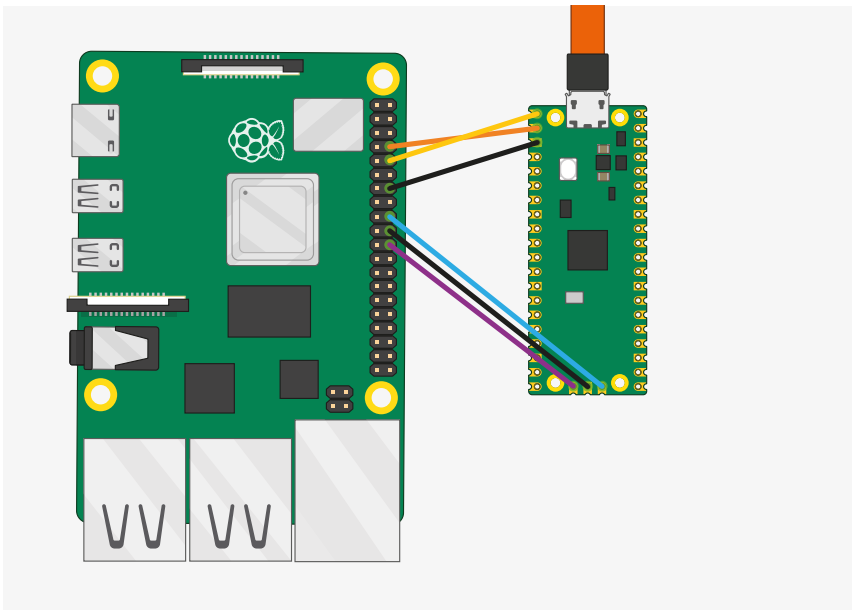


図 11. この図は、UART と SWD デバッグポートが搭載された Raspberry Pi 4 と Raspberry Pi Pico を接続した状態を示しています。ブレッドボードを使用することなく、Raspberry Pi 4 と Raspberry Pi Pico が直接接続されています。

次に、`CMAKE_BUILD_TYPE=Debug` を以下のように指定して、プロジェクトのデバッグ版をビルドします。

```
$ cd ~/pico/test
$ rmdir build
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make
```

次に、ターミナルウィンドウを開き、raspberrypi-swd インターフェイスを使用して OpenOCD を接続します。

```
$ openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg
```

使用する OpenOCD ターミナルは、稼働状態のままにしておく必要があります。別のターミナルウィンドウを開き、以下のように入力して `gdb-multiarch` を開始します。

```
$ cd ~/pico/test/build
$ gdb-multiarch test.elf
```

GDB を OpenOCD に接続し、`test.elf` バイナリをフラッシュに読み込みます。

```
(gdb) target remote localhost:3333
(gdb) load
```

読み込んだバイナリを実行します。

```
(gdb) monitor reset init
(gdb) continue
```

8.2. Visual Studio Code での操作

コマンドラインではなく、Visual Studio Code を使用して作業を行う場合の手順については、第章 7を参照してください。環境の設定方法と、新しいプロジェクトを開発環境に読み込んでコードをビルドする方法が記載されています。

Visual Studio Code を使用してデバッグを実行し、コードを Raspberry Pi Pico に読み込む場合は、プロジェクト用として `launch.json` ファイルを作成する必要があります。その際に、サンプルの `launch-raspberrypi-swd.json` ファイルを使用することができます (第章 7を参照)。このファイルのコピーを `.vscode/launch.json` というファイル名で作成し、プロジェクトディレクトリに保管する必要があります。

8.3. プロジェクト作成の自動化

Pico のプロジェクト生成機能により、プロジェクトの作成に必要なすべてのファイルとともに「スタブ」プロジェクトが自動的に作成されます。これを行うには、プロジェクト作成スクリプトを Git リポジトリから複製する必要があります。

```
$ git clone https://github.com/raspberrypi/pico-project-generator.git --branch master
```

このスクリプトは、グラフィカルモードで実行することができます。

```
$ cd pico-project-generator
$ ./pico_project.py --gui
```

グラフィカルモードの場合、プロジェクトを設定するための GUI インターフェイスが表示されます (図 12 を参照)。

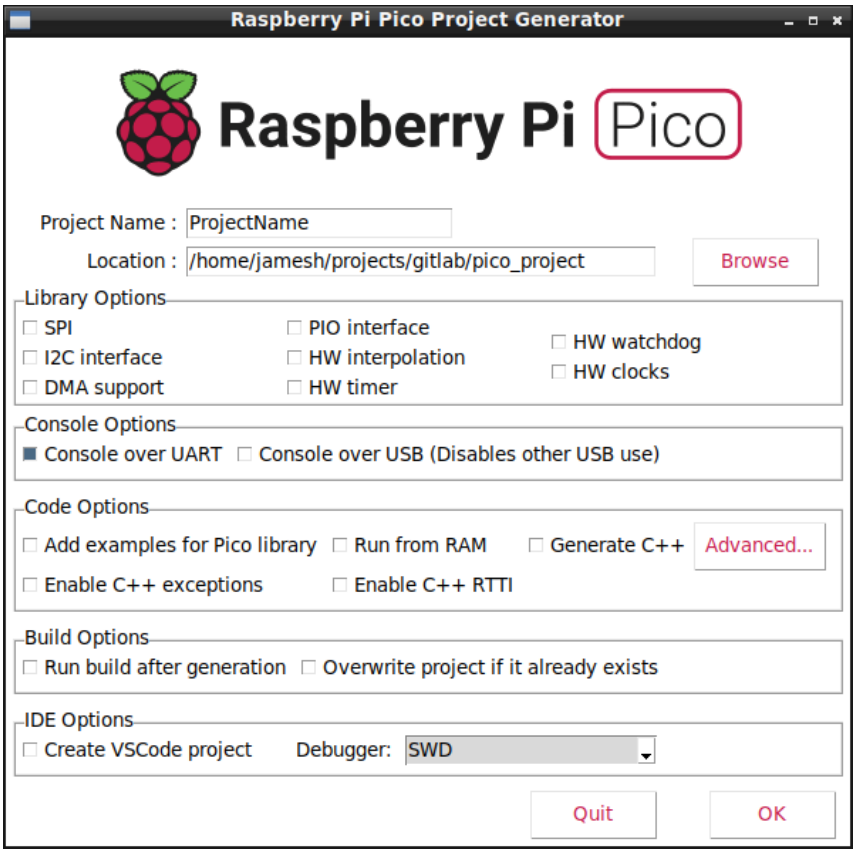


図 12. グラフィカルプロジェクト作成ツールを使用して RP2040 プロジェクトを作成する

GUI のチェックボックスで特定の機能を選択し、その機能をプロジェクトに追加することができます。機能をプロジェクトに追加すると、ビルドシステムによって適切なコードがビルドに追加され、その機能の使用方法を示す簡単なサンプルコードがプロジェクトに追加されます。

以下に示すオプションが用意されています。

コンソールオプション	説明
Console over UART	UART 経由でシリアルコンソールを使用することができます。これが、デフォルトのオプションです。
Console over USB	USB 経由でコンソールを使用することができます。デバイスが USB シリアルポートとして機能します。このオプションは、「Console over UART」オプションと組み合わせて使用することも、「Console over UART」オプションの代わりに使用することもできますが、このオプションを有効にすると、他の USB 機能が使用できなくなることに注意してください。

コードオプション	説明
Add examples for Pico library	一部の標準ライブラリ機能 (UART サポート機能や HW 分割機能など) に対してサンプルコードが生成されます。これらの機能は、デフォルトでビルドに取り込まれます。
Run from RAM	通常、フラッシュメモリーにインストールされるバイナリが作成されます。これにより、バイナリが RAM から直接実行されます。
Generate C++	C++ に対応したソースファイルが生成されます。

コードオプション	説明
Enable C++ exceptions	C++ の例外が有効になります。通常は、コードスペースを節約するため、このオプションを無効にします。
Enable C++ RTTI	C++ ランタイムタイプ情報が有効になります。通常は、コードスペースを節約するため、このオプションを無効にします。
Advanced	特定のボードビルドオプションを選択するためのテーブルが表示されます。このオプションを使用すると、特定の機能の動作が変更されるため、注意して使用する注意があります。

ビルドオプション	説明
Run Build	作成済みプロジェクトがビルドされます。その際、Raspberry Pi Pico にダウンロードするファイルが作成されます。
Overwrite Project	指定したフォルダー内にプロジェクトがすでに存在する場合、そのプロジェクトが新しいプロジェクトによって上書きされます。プロジェクトの変更内容もすべて上書きされます。

IDE オプション	説明
Create VSCode Project	CMake ファイルと同様に、適切な Visual Studio Code プロジェクトファイルが作成されます。
Debugger	VSCode デバッグシステムで使用される Pico デバッガーを選択することができます。デフォルトのデバッガーは、Serial Wire Debug です。

8.3.1. コマンドラインを使用してプロジェクトを生成する

以下のように入力することにより、コマンドラインからプロジェクトを作成することができます。

```
$ export PICO_SDK_PATH="/home/pi/pico/pico-sdk"
$ ./pico_project.py --feature spi --feature i2c --project vscode test
```

`--feature` オプションを使用して特定の機能を指定すると、適切なライブラリコードがビルドに追加され、その機能の基本的な使用方法を示すサンプルコードが追加されます。RP2040 のメモリー制限に達するまで、複数の機能を追加することができます。`--list` オプションを指定すると、使用可能なすべての機能が一覧表示されます。上記のサンプルスクリプトでは、I2C インターフェイスと SPI インターフェイスのサポート機能を追加しています。

`--project` オプションを指定すると、CMake プロジェクトファイルのほかに、`.vscode/launch.json` ファイル、`.vscode/c_cpp_properties.json` ファイル、`.vscode/settings.json` ファイルが作成されます。

プロジェクトの作成後は、コマンドラインから通常の方法でそのプロジェクトをビルドすることができます。

```
$ cd test/build
$ cmake ..
$ make
```

Visual Studio Code からビルドすることもできます。

`--help` オプションを使用すると、複数のコマンドライン引数を指定することができます。グラフィカルモードを使用する場合も、これらの引数が適用されます。

詳細情報が必要な場合

本書には、Raspberry Pi Pico の使用を開始するための情報が記載されていますが、一部のファイルやコードについてもっと詳しく知りたいと思うかもしれません。その場合は、「Raspberry Pi Pico C/C++ SDK」を参照してください。このドキュメントには、プロジェクトどのようにビルドされるのか、`CMakeLists.txt` ファイル内の行が SDK の構造にどのように関係しているのかなどについて、詳しい説明が記載されています。

第章 9. 別のプラットフォーム上でビルドを行う

RP2040 の開発で主に使用されるプラットフォームは Raspberry Pi ですが、Apple macOS や Microsoft Windows など、別のプラットフォームを使用することもできます。

9.1. Apple macOS 上でのビルド

macOS 上で RP2040 のコードをビルドする方法は、Linux の場合と非常によく似ています。

9.1.1. ツールチェーンのインストール

ツールチェーンをインストールするには、Homebrew が必要になります。 Homebrew がインストールされていない場合は、インストールしてください。

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

次に、ツールチェーンをインストールします。

```
$ brew install cmake
$ brew tap ArmMbed/homebrew-formulae
$ brew install arm-none-eabi-gcc
```

Raspberry Pi の手順に従い、RP2040 のコードをビルドすることができます。ツールチェーンのインストール後は、macOS と Linux での手順に違いはありません。Section 2.1 の手順に従って SDK を入手し、「Blink」サンプルをビルドしてください。

9.1.2. Visual Studio Code を使用する

Visual Studio Code (VSCode) は、macOS、Linux、Microsoft Windows に対応しています。 macOS 版をダウンロードして解凍し、Applications フォルダーにドラッグしてください。

Applications フォルダーに移動してアイコンをクリックすると、Visual Studio Code が起動します。

9.1.3. CMake Tools によるビルド

Visual Studio Code を起動したら、 CMake Tools という拡張機能をインストールする必要があります。左側のツールバーの拡張機能アイコンをクリックし (または、`Cmd + Shift + X` キーを押し)、「CMake Tools」を検索します。検索結果のリスト内で「CMake Tools」をクリックし、インストールボタンをクリックします。

ここで、`PICO_SDK_PATH` 環境変数を設定する必要があります。`pico-examples` ディレクトリに移動して `.vscode` ディレクト

リを作成し、`settings.json` というファイルを追加します。これにより、CMake Tools が SDK の場所を認識できるようになります。また、CMake Tools で Visual Studio を指定する必要があります。

```
{
    "cmake.environment": {
        "PICO_SDK_PATH":"../../pico-sdk"
    },
}
```

インターフェイス左側のナビゲーションバー下部にある歯車アイコンをクリックし、「Settings」を選択します。次に、「Settings」ペインで「Extensions」>「CMake Tools configuration」をクリックします。次に、「Cmake: Generator」までスクロールダウンし、ボックス内に「Unix Makefiles」と入力します。

注記

ローカル環境での設定内容によっては、CMake ジェネレーターを手動で「Unix Makefiles」に設定する必要がない場合がありますただし、この手動設定を行わなかった場合、Visual Studio は `make` ではなく `ninja` をデフォルトで使用し、GCC は `ninja` が理解できない形式で依存性情報を出力するため、ビルドが失敗する可能性があります。

この環境変数を手動で設定する場合は、`settings.json` ファイルに行を追加して、CMake Tools で Visual Studio を明示的に指定しなければならないことがあります。

```
{
    "cmake.environment": {
        "PICO_SDK_PATH": "../../pico-sdk"
    },
    "C_Cpp.default.configurationProvider": "ms-vscode.cmake-tools"
}
```

次に、「File」メニューにアクセスして「Add Folder to Workspace…」をクリックし、`pico-examples` リポジトリに移動して「Okay」をクリックします。プロジェクトが読み込まれ、図 13 のようなコンパイラー選択画面が表示されます (表示されない場合もあります)。ここでは、「GCC for arm-none-eabi」コンパイラーを選択します。

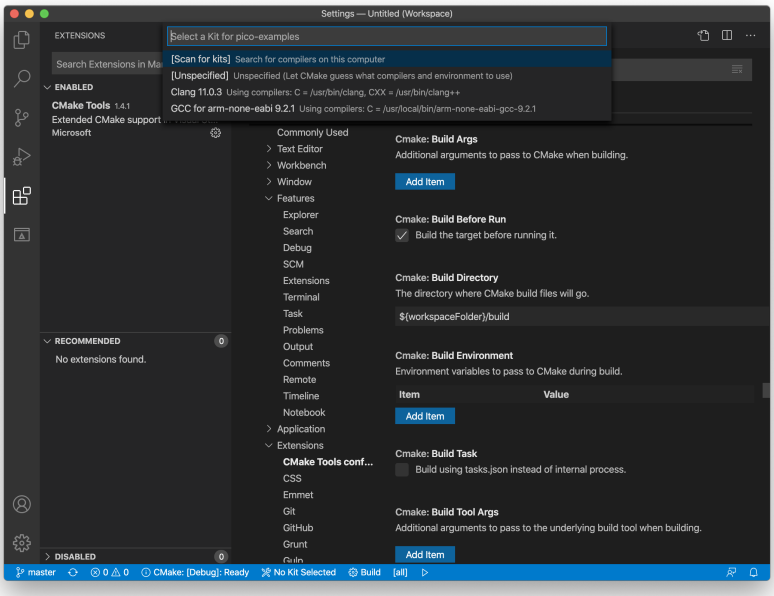


図 13. プロジェクトに適したコンパイラーの選択画面

ウィンドウ下部の青いバーにある「Build」ボタン (歯車が付いているボタン) をクリックします。これにより、ビルド

ディレクトリが作成され、ビルドが起動する前に、Section 3.1 で手動で実行した CMake コマンドが自動的に実行されます (図 9 を参照)。

ターゲットファイルとして、elf ファイル、`bin` ファイル、`uf2` ファイルが生成されます。これらのファイルは、新しく作成された `build` ディレクトリ内の `hello_world/serial` ディレクトリと `hello_world/usb` ディレクトリに保存されます。UF2 バイナリは、USB でコンピューターに接続された RP2040 ボードに直接ドラッグアンドドロップすることができます。

9.1.4. 「Hello World」プログラムの出力の表示

第章 4で説明したように、`stdio` をピン GP0 とピン GP1 の USB CDC (シリアル) または UART0 に送信して、「Hello World」サンプルプログラムをビルドすることができます。USB CDC をターゲット出力としてビルドする場合は、USB CDC がクラスコンプライアントデバイスになるため、ドライバーをインストールする必要はありません。 Serial などのターミナルプログラムを使用して USB シリアルポートに接続するだけでかまいません。

9.1.4.1. UART 出力

Raspberry Pi Pico の標準 UART に接続して出力を表示する場合は、USB - UART シリアルコンバーター (SparkFun FTDI Basic ボードなど) を使用して Raspberry Pi Pico を Mac に接続する必要があります (図 14 を参照)。

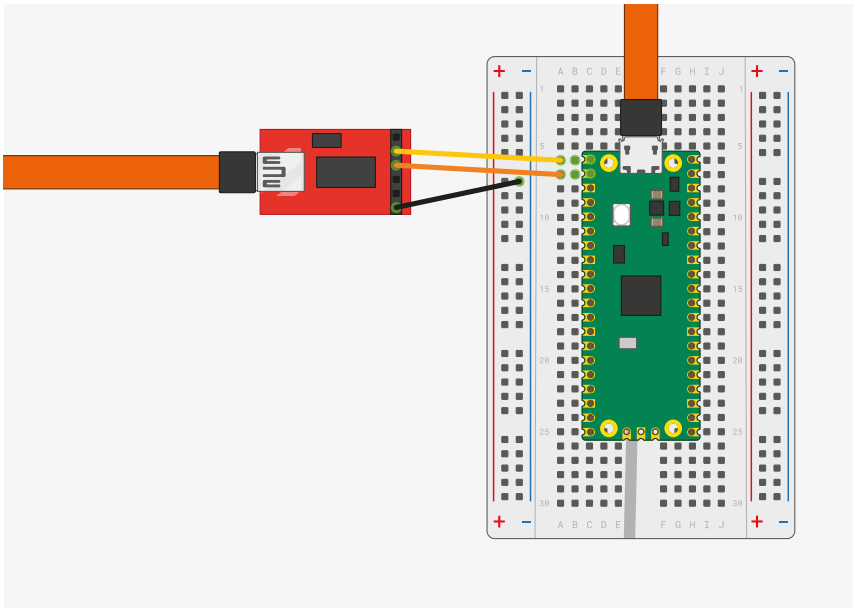


図 14. Sparkfun FTDI Basic アダプターを Raspberry Pi Pico に接続する

Catalina などの新しい macOS バージョンを使用している場合、ドライバーはすでに読み込まれています。ドライバーが読み込まれていない場合は、 FTDI Chip ドライバー の Web サイトを参照してください。

次に、 Serial などのターミナルプログラムを使用してシリアルポートに接続します。Serial には、 ドライバーサポート機能も組み込まれています。

9.2. MS Windows 上でのビルド

Microsoft Windows 10 へのツールチェーンのインストール方法は、他のプラットフォームとは多少異なっています。ただし、インストール後の RP2040 用コードのビルド方法は類似しています。

ヒント

Windows 10 が Raspberry Pi で直接サポートされていない場合は、Windows 10 用のサードパーティ製インストーラースクリプトを使用することができます。このスクリプトは、Raspberry Pi 用の `pico-setup.sh` スクリプトとほぼ同等の機能を持っています (第章 1を参照)。詳しくは、 https://github.com/ndabas/pico-setup-windows を参照してください。

警告

Windows 7 と Windows 8 の場合、Raspberry Pi Pico は正式にはサポートされていませんが、 動作させることはできます。

9.2.1. ツールチェーンのインストール

プログラムをビルドするには、以下のツールをインストールする必要があります。

- ARM GCC コンパイラー
- CMake
- Build Tools for Visual Studio 2019
- Python 3.9
- Git

上記のリンクから各ツールのインストーラー (実行可能ファイル) をダウンロードし、以下の各セクションの手順に従い、5 つのインストーラーすべてを Windows コンピューターにインストールします。

9.2.1.1. ARM GCC コンパイラーのインストール

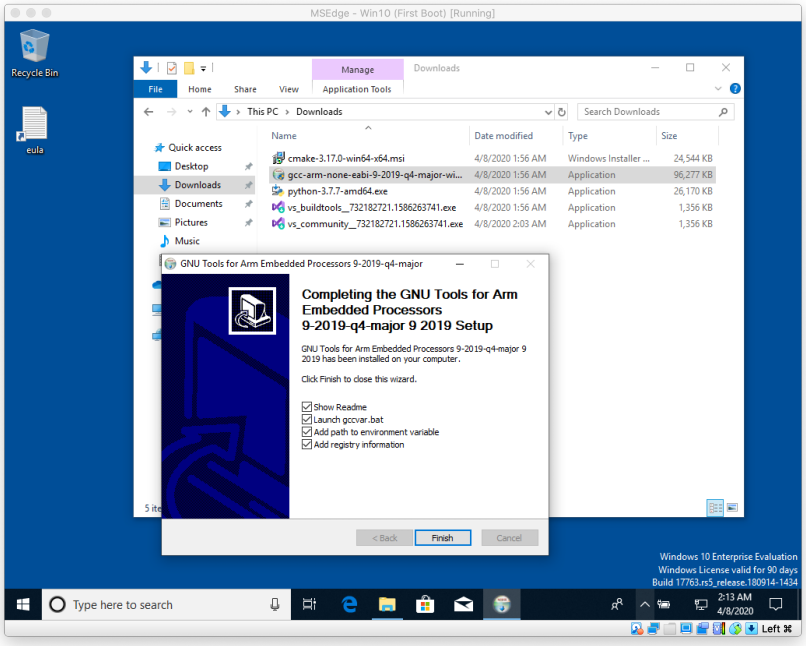


図 15. 必要なツールを Windows コンピューターにインストールします。コンパイラーのパスをコマンドラインで指定できるように、コンパイラーのパスを環境変数として登録する必要があります。

インストール時に、ARM コンパイラーのパスを Windows シェルの環境変数として登録するためのボックスが表示され

たら、そのボックスを選択します。

9.2.1.2. CMake のインストール

インストール時にプロンプト画面が表示されたら、すべてのユーザーについて、CMake を `PATH` システム環境変数に追加します。

9.2.1.3. Build Tools for Visual Studio 2019 のインストール

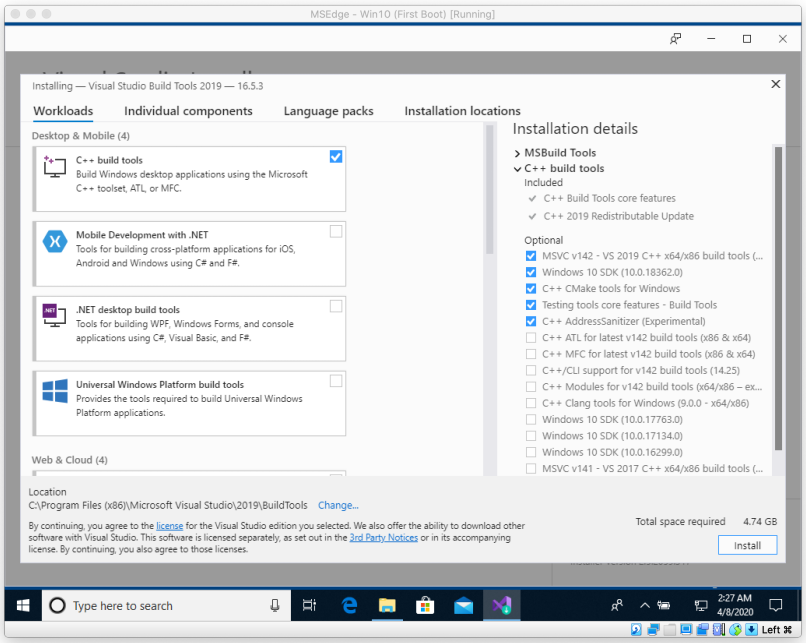


図 16. Build Tools for Visual Studio 2019 をインストールします。

インストール時にプロンプト画面が表示された場合は、C++ ビルドツールだけをインストールする必要があります。

注記

SDK では、`pioasm` ツールと `elf2uf2` ツールをローカルでビルドする必要があるため、Windows 10 SDK のフルパッケージをインストールする必要があります。インストールされている項目の一覧からこれを削除すると、Raspberry Pi Pico のバイナリをビルドできなくなります。

9.2.1.4. Python 3.9 のインストール

インストール時にプロンプト画面が表示されたら、すべてのユーザーに対して Python 3.9 をインストールし、Python 3.9 を `PATH` システム環境変数に追加します。インストールの最後にプロンプト画面が表示された場合は、`MAX_PATH` 変数の長さ制限を無効にする必要があります。

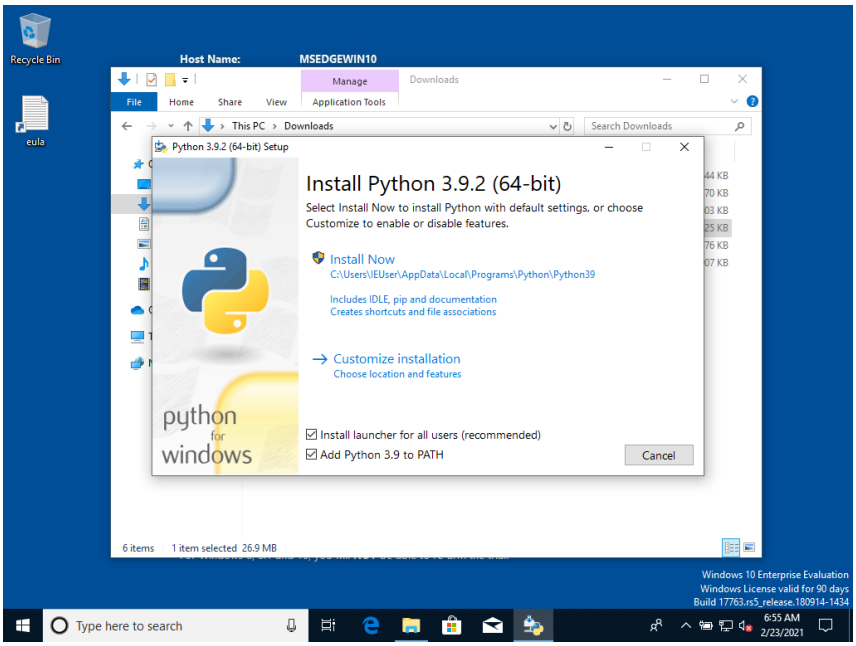


図 17. 「Add Python 3.9 to PATH」ボックスを選択して Python 3.9 をインストールします。

9.2.1.5. Git のインストール

Git をインストールする場合は、デフォルトのエディターを `vim` 以外のエディターに変更する必要があります (図 18 を参照)。

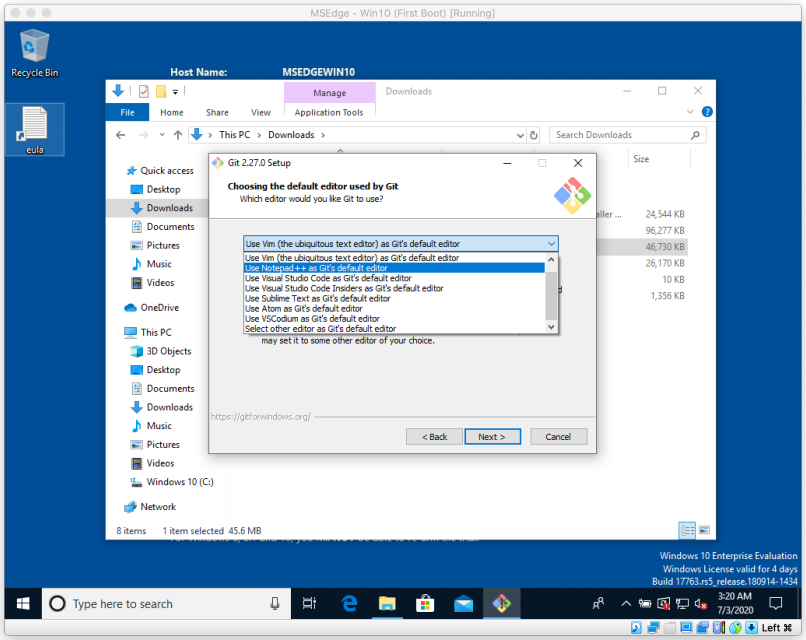


図 18. Git のインストール

サードパーティー製ツールで Git を使用するためのチェックボックスを選択します。また、Git のインストール時には、特別な理由がない限り、「Checkout as is, commit as-is」ボックスを選択し、「Use Windows' default console window」と「Enable experimental support for pseudo consoles」を選択する必要があります。

9.2.2. SDK とサンプルの入手

```
C:\Users\pico\Downloads> git clone https://github.com/raspberrypi/pico-sdk.git --branch master
C:\Users\pico\Downloads> cd pico-sdk
C:\Users\pico\Downloads\pico-sdk> git submodule update --init
C:\Users\pico\Downloads\pico-sdk> cd ..
C:\Users\pico\Downloads> git clone https://github.com/raspberrypi/pico-examples.git --branch master
```

9.2.3. コマンドラインを使用して「Hello World」プログラムをビルドする

メニューで `Windows > Visual Studio 2019 > Developer Command Prompt` を選択し、コマンドプロンプトウィンドウを開きます。

次に、SDK のパスを以下のように設定します。

```
C:\Users\pico\Downloads> setx PICO_SDK_PATH "..\..\pico-sdk"
```

*開いているコマンドプロンプトウィンドウを終了*し、新しいコマンドプロンプトウィンドウを開きます。これにより、PICO_SDK_PATH 環境変数が正しく設定されます。

`pico-examples` フォルダーに移動し、以下のように「Hello World」サンプルプログラムをビルドします。

```
C:\Users\pico\Downloads> cd pico-examples
C:\Users\pico\Downloads\pico-examples> mkdir build
C:\Users\pico\Downloads\pico-examples> cd build
C:\Users\pico\Downloads\pico-examples\build> cmake -G "NMake Makefiles" ..
C:\Users\pico\Downloads\pico-examples\build> nmake
```

これにより、ターゲットファイルとして、elf ファイル、`bin` ファイル、`uf2` ファイルが生成されます。これらのファイルは、`build` ディレクトリ内の `hello_world/serial` ディレクトリと `hello_world/usb` ディレクトリに保存されます。UF2 バイナリは、USB でコンピューターに接続された RP2040 ボードに直接ドラッグアンドドロップすることができます。

9.2.4. Visual Studio Code を使用して「Hello World」プログラムをビルドする

ツールチェーンのインストールが完了したら、Visual Studio Code をインストールします。Visual Studio Code を使用すると、現在の環境内でプロジェクトをビルドすることができます。コマンドラインを使用する必要はありません。

Windows 用の Visual Studio Code を ダウンロード してインストールします。インストールが完了したら、メニューで `Windows > Visual Studio 2019 > Developer Command Prompt` を選択し、コマンドプロンプトウィンドウを開きます。コマンドプロンプトで、以下のように入力します。

```
C:> code
```

これにより、すべての環境変数が正しく設定された状態で Visual Studio Code が起動し、ツールチェーンが正しく設定されます。

警告

デスクトップのアイコンをクリックして Visual Studio Code を起動した場合や、スタートメニューから直接 Visual Studio Code を起動した場合、ビルド環境は*正しく設定されません*。環境変数の設定は、後で「CMake Tools Settings」オプションを使用して手動で行うこともできますが、Visual Studio Code の環境を設定する最も簡単な方法は、コマンドプロンプトウィンドウで Visual Studio Code を開くという方法です。コマンドプロンプトウィンドウで Visual Studio Code を起動するだけで、すべての環境変数が正しく設定されます。

次に、CMake Tools という拡張機能をインストールする必要があります。左側のツールバーの拡張機能アイコンをクリックし (または、`Ctrl + Shift + X` キーを押し)、「CMake Tools」を検索します。検索結果のリスト内で「CMake Tools」をクリックし、インストールボタンをクリックします。

インターフェイス左側のナビゲーションバー下部にある歯車アイコンをクリックし、「Settings」を選択します。次に、「Settings」ペインで「Extensions」>「CMake Tools configuration」をクリックします。次に、「Cmake: Configure Environment」までスクロールダウンし、「Add Item」をクリックして、`PICO_SDK_PATH` 環境変数の値を「`..\..\pico-sdk`」に設定します (図 19 を参照)。

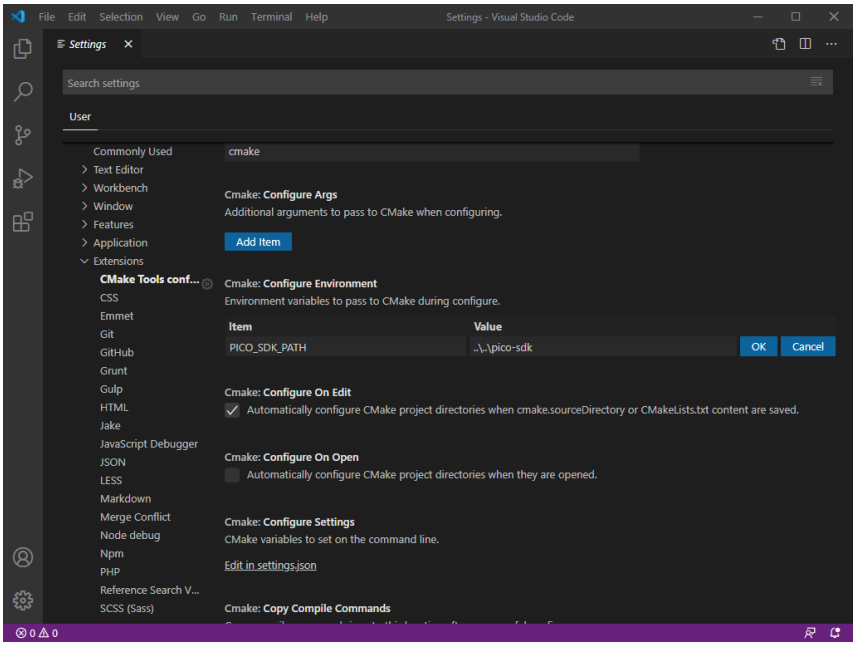


図 19. CMake 拡張機能で `PICO_SDK_PATH` 環境変数を設定する

「Cmake: Generator」までスクロールダウンし、ボックス内に「NMake Makefiles」と入力します。

重要

「Cmake: Generator」を変更しなかった場合、Visual Studio は `ninja` をデフォルトで使用し、GCC は `ninja` が理解できない形式で依存性情報を出力するため、ビルドが失敗する可能性があります。

「Settings」ページを終了して「File」メニューに移動し、「Open Folder」をクリックして `pico-examples` リポジトリに移動し、「Okay」をクリックします。これにより、プロジェクトの設定画面が表示されます (図 20 を参照)。ここでは、「GCC for arm-none-eabi」コンパイラーを選択します。

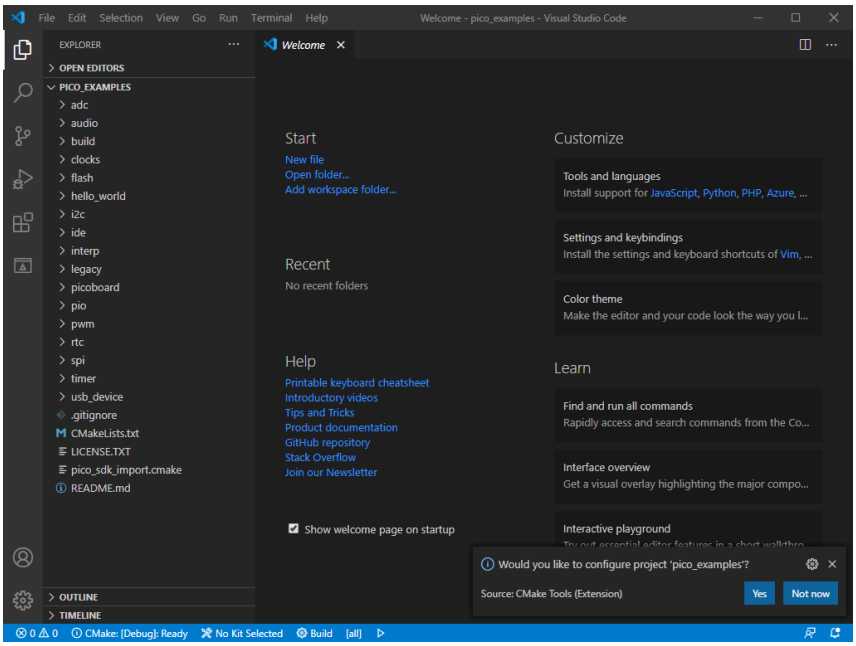


図 20. Visual Studio Code のプロジェクト設定画面

ウィンドウ下部の青いバーにある「Build」ボタン (歯車が付いているボタン) をクリックします。これにより、ビルドディレクトリが作成されて CMake が起動し、「Hello World」などのサンプルプロジェクトがビルドされます。

ターゲットファイルとして、elf ファイル、`bin` ファイル、`uf2` ファイルが生成されます。これらのファイルは、新しく作成された `build` ディレクトリ内の `hello_world/serial` ディレクトリと `hello_world/usb` ディレクトリに保存されます。UF2 バイナリは、USB でコンピューターに接続された RP2040 ボードに直接ドラッグアンドドロップすることができます。

9.2.5. 「Hello World」プログラムのフラッシュと実行

`BOOTSEL` ボタンを押しながら、Micro USB ケーブルを使用して、Raspberry Pi Pico を Raspberry Pi に接続します。これにより、強制的に USB マスストレージモードに切り替わります。 ボードは、自動的に外付けドライブとして表示されます。この状態で、UF2 バイナリを外付けドライブにドラッグアンドドロップすることができます。

Raspberry Pi Pico が再起動し、外付けドライブとしてのマウントが解除され、フラッシュされたコードの実行が開始されます。

第章 4で説明したように、`stdio` をピン GP0 とピン GP1 の USB CDC (シリアル) または UART0 に送信して、「Hello World」サンプルプログラムをビルドすることができます。USB CDC をターゲット出力としてビルドする場合は、USB CDC がクラスコンプライアントデバイスになるため、ドライバーをインストールする必要はありません。

9.2.5.1. UART 出力

Raspberry Pi Pico の標準 UART に接続して出力を表示する場合は、USB - UART シリアルコンバーター (SparkFun FTDI Basic ボードなど) を使用して Raspberry Pi Pico をコンピューターに接続する必要があります (図 21 を参照)。

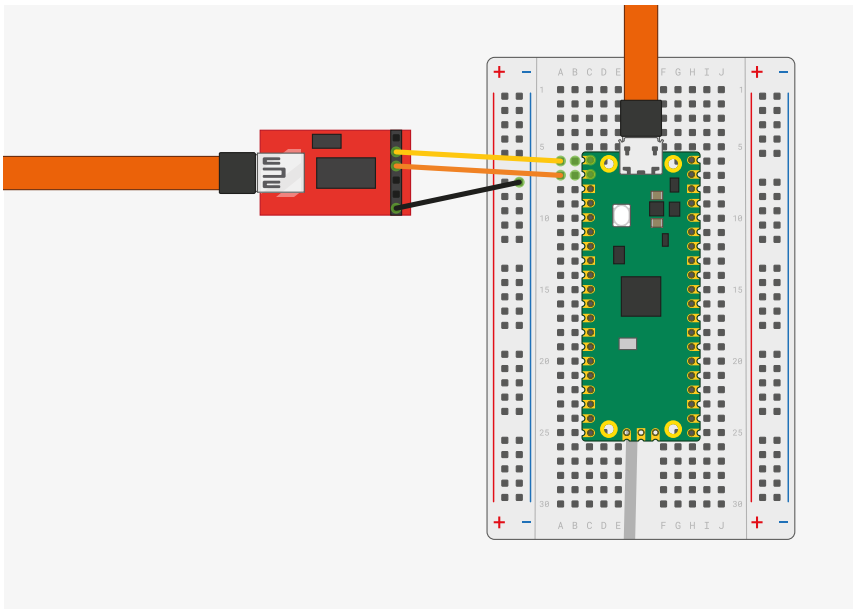


図 21. Sparkfun FTDI Basic アダプターを Raspberry Pi Pico に接続する

Windows 10 の新しいバージョンを使用している場合は、適切なドライバーがすでに読み込まれています。ドライバーが読み込まれていない場合は、 FTDI Chip ドライバー の Web サイトを参照してください。

ドライバーがインストールされていない場合は、 PuTTY をダウンロードしてインストールしてください。次に、このドライバーを実行して「Serial」を選択し、ボーレートとして「115,200」という値を「Speed」ボックスに入力し、UART コンバーターで使用されているシリアルポートを入力します。このシリアルポートがわからない場合は、`chgport` コマンドで確認することができます。

```
C:> chgport
COM4 = \Device\ProlificSerial10
COM5 = \Device\VCP0
```

これにより、アクティブなシリアルポートが一覧表示されます。ここでは、USB - UART シリアルコンバーターは `COM5` に配置されています。

注記

複数のシリアルデバイスのうち、どれが UART - USB シリアルコンバーターなのかわからない場合は、ケーブルを抜いてから `chgport` コマンドを実行して、どの COM ポートがコマンド出力に表示されなくなるかを確認してください。

速度とポートを入力してから「Open」ボタンをクリックすると、Raspberry Pi Pico からの UART 出力がターミナルウィンドウに表示されます。

第章 10. 別の統合開発環境を使用する

現在の推奨統合開発環境 (IDE) は Visual Studio Code です (第章 7を参照)。ただし、RP2040 と Raspberry Pi Pico で別の環境を使用することもできます。

10.1. Eclipse を使用する

Eclipse は、マルチプラットフォームの統合開発環境 (IDE) で、x86 Linux、Windows、Mac に対応しています。また、最新版の IDE は 64 ビットの ARM システムにも対応し、64 ビット OS が搭載された Raspberry Pi 4/400 シリーズ (4GB 以上) でも問題なく動作します。ここでは、Raspberry Pi Pico で使用できるように、Linux デバイスで Eclipse をセットアップする方法について説明します。Raspberry Pi Pico への接続方法は異なりますが、ほかのシステムについてもほぼ同じ手順になります。Linux 以外のプラットフォームについては、Section 9.2 とSection 9.1 を参照してください。

10.1.1. Linux マシンで Pico を Eclipse 用にセットアップする

前提条件

- 4GB 以上の RAM が搭載された最新バージョンの Linux が動作するデバイス
- 64 ビットのオペレーティングシステム
- バージョン 3.11 以降の CMake

注記

現在、64 ビット版の Raspberry Pi OS は、ベータ版としてテストを行っている段階です。最新のベータ版は、http://downloads.raspberrypi.org/raspios_arm64/images/ からダウンロードすることができます。別の 64 ビット版 Linux ディストリビューションも使用できますが、一部のディストリビューション (Raspberry Pi 用の Ubuntu など) については、当社はテストを行っていません。SD カードに OS イメージをインストールする場合は、通常の手順でインストールしてください。

Raspberry Pi を使用する場合は、config.txt ファイルに以下の行を追加して、標準の UART を有効にする必要があります。

```
enable_uart=1
```

また、OpenOCD と SWD デバッグシステムもインストールする必要があります。インストール方法については、第章 5 を参照してください。

10.1.1.1. Eclipse と Eclipse プラグインのインストール

通常の手順で、最新版の Eclipse Embedded CDT をインストールします。ARM プラットフォームを使用する場合は、Eclipse の AArch64 バージョン (64 ビットの ARM バージョン) をインストールする必要があります。Eclipse の Web サイト (https://projects.eclipse.org/projects/iot.embed-cdt/downloads) で、すべてのバージョンを確認することができます。

使用しているシステムに対応するファイルをダウンロードして解凍してください。解凍された eclipse という実行ファ

イルを実行すると、Eclipse が起動します。

```
$ ./eclipse
```

Eclipse の Embedded CDT バージョンには、C/C++ 開発キットと Embedded 開発キットが付属しています。これらのキットにより、Raspberry Pi Pico の開発で必要なすべての作業を行うことができます。

10.1.1.2. pico-examples フォルダーの使用

Pico 環境の標準的なビルドシステムは CMake です。ただし、Eclipse には専用のビルドシステムがあるため、CMake は使用しません。そのため、pico-examples フォルダー内の CMake ビルドを Eclipse プロジェクトに変換する必要があります。

- `pico-examples` フォルダーと同じ階層に、`pico-examples-eclipse` などの名前で新しいフォルダーを作成します。
- 新しいフォルダーに移動します。
- パスを PICO_SDK_PATH に設定します。

```
$ export PICO_SDK_PATH=<wherever>
```

コマンドラインで、以下のように入力します。

```
$ cmake -G"Eclipse CDT4 - Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug ../pico-examples
```

これにより、元の CMake ツリーのソースを使用して、pico-examples-eclipse フォルダー内に Eclipse プロジェクトファイルが作成されます。

これで、ファイルメニューの「`Open project From File System`」オプションを使用して、新しいプロジェクトファイルを Eclipse に読み込むことができるようになりました。

10.1.1.3. ビルド

プロジェクトエクスプローラーで目的のプロジェクトを右クリックし、「`Build`」を選択します。これにより、すべてのサンプルがビルドされます。

10.1.1.4. OpenOCD

ここでは、OpenOCD システムを使用して Raspberry Pi Pico と通信します。コードを実行する前に、ホストデバイスと Raspberry Pi Pico を 2 線式デバッグインターフェイスで接続する必要があります。Raspberry Pi の場合、GPIO 接続を使用してこれを行うことができますが、ノートパソコンやデスクトップパソコンでは、追加のハードウェアを使用する必要があります。1 つの方法として、Picoprobe が稼働する 2 台目の Raspberry Pi Pico を使用します (付録 A を参照)。デバッグ接続に関する詳しい説明については、第章 5を参照してください。

OpenOCD のインストールと接続が完了したら、プログラムの実行時に Eclipse と OpenOCD が通信できるように設定する必要があります。OpenOCD により、GDB インターフェイスが Eclipse に提供されますが、デバッグ時にはこのイ

ンターフェイスを使用します。

OpenOCD システムを設定するには、`Window` メニューの「`Preferences`」を選択します。

「`MCU`」の矢印をクリックしてオプションを展開し、「`Global OpenOCD path`」をクリックします。

「Executable」フィールドに「openocd」と入力します。「Folder」フィールドで、GitHub から Pico OpenOCD フォークを複製したファイルシステムの場所を選択します。

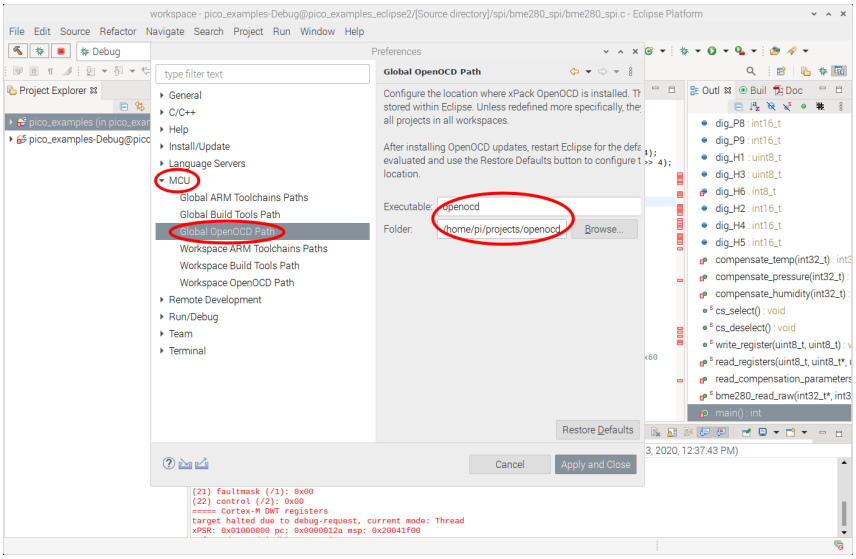


図 22. Eclipse で、OCD の実行ファイルの名前とパスを設定する

10.1.1.5. 実行設定情報の作成

Eclipse でコードを実行したりデバッグを行ったりするには、実行設定情報を作成する必要があります。この設定情報により、実行するコード、必要なパラメーター、デバッガー、ソースパス、SVD 情報を特定できるようになります。

Eclipse の「Run」メニューで「`Run Configurations`」を選択します。デバッグ設定情報を作成するには、「`GDB OpenOCD Debugging`」オプションを選択してから「`New Configuration`」ボタンを選択します。

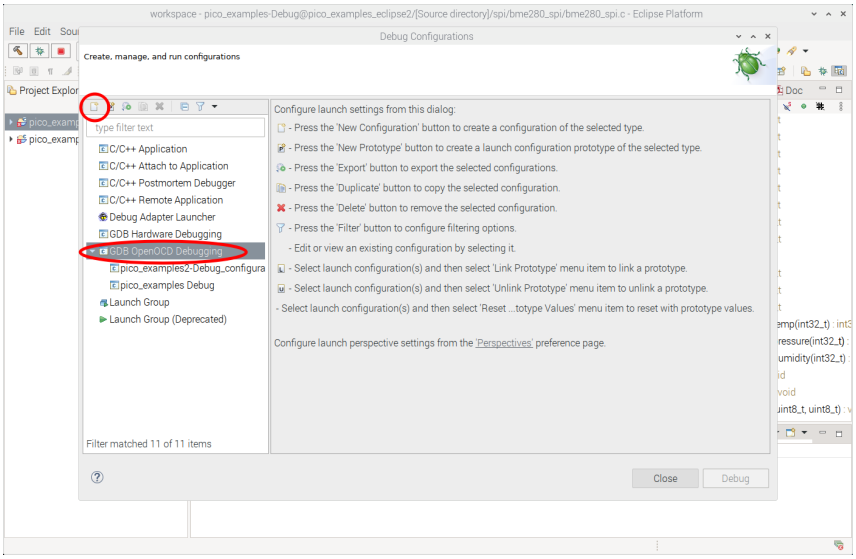


図 23. Eclipse で、新しい実行設定情報/デバッグ設定情報を作成する

10.1.1.5.1. 実行するアプリケーションの設定

pico-examples をビルドすると、さまざまなアプリケーションの実行ファイルが作成されるため、どのファイルを実行するか (またはデバッグするか) を選択する必要があります。

「Run configuration」ページの「`Main`」タブで「`Browse`」オプションを使用して、実行する C/C++ アプリケーションを選択します。

Eclipse をビルドすると、Eclipse プロジェクトフォルダーのサブフォルダー内に実行ファイルが作成されます。ここでは、以下のように指定します。

```
…/pico-examples-eclipse/<name of example folder>/<optional name of example subfolder>/executable.elf
```

たとえば、LED を点滅させるサンプルプログラムの場合、以下の実行ファイルが作成されます。

```
…/pico-examples-eclipse/blink/blink.elf
```

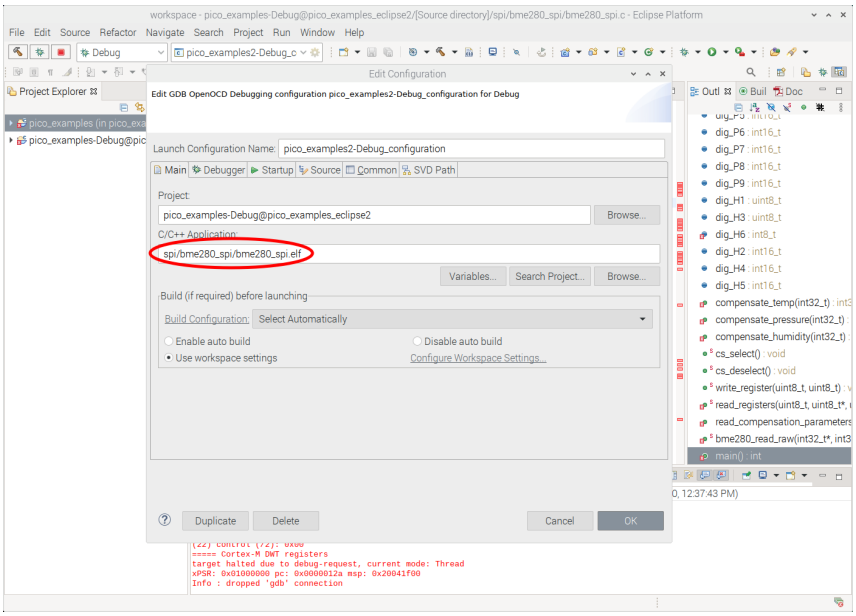


図 24. Eclipse で、実行ファイルのデバッグを設定する

10.1.1.5.2. デバッガーの設定

ここでは、OpenOCD を使用して Raspberry Pi Pico と通信するため、この動作を設定する必要があります。

「Executable path」フィールドと「Actual Executable」フィールドに「`openocd`」と入力します。また、Pico 固有の設定を OpenOCD で使用するように設定する必要があるため、「Config options」フィールドに以下のコードを入力します。Pico 版の OpenOCD がインストールされている場所を指すようにパスを変更する必要があることに注意してください。

```
-f interface/raspberrypi-swd.cfg -f target/rp2040.cfg
```

その他の OpenOCD 設定は、すべてデフォルト値に設定する必要があります。

実際に使用するデバッガーは GDB です。GDB は OpenOCD デバッガーと対話しながら Raspberry Pi Pico と通信しますが、GDB には、IDE に対する標準的なインターフェイスが付属しています。

使用する GDB のバージョンは `gdb-multiarch' であるため、「Executable name」フィールドと「Actual Executable」フィールドにはこのバージョン名を入力します。

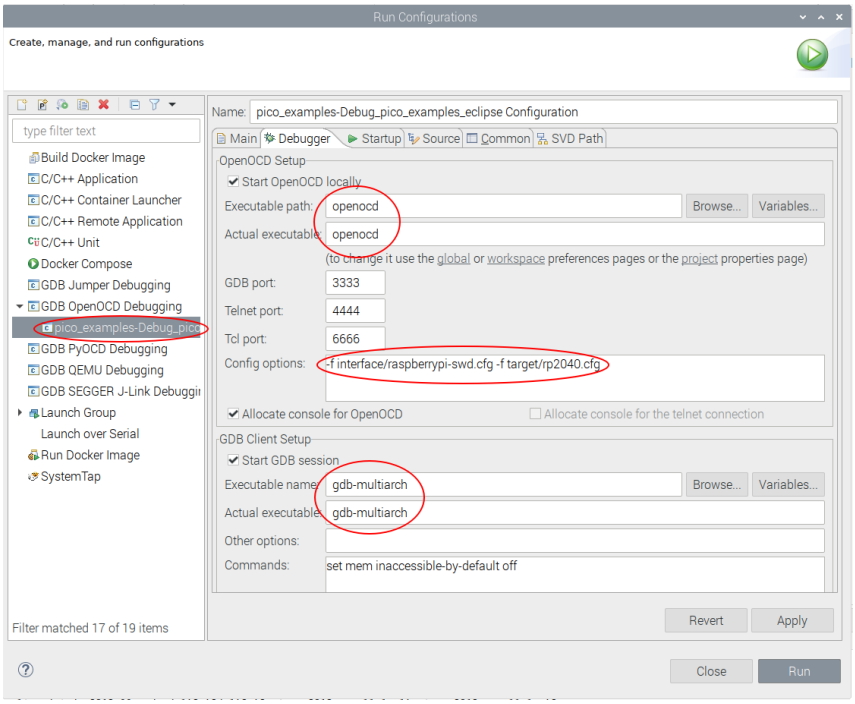


図 25. Eclipse でデバッガーと OpenOCD を設定する

10.1.1.5.3. SVD プラグインの設定

SVD には、Pico ボードの周辺レジスターの表示と設定を行うための機能が組み込まれています。SVD ファイルには、レジスタ-の場所と説明が記録されています。Eclipse 用 SVD プラグインにより、SVD の機能が Eclipse IDE に統合されます。SVD プラグインは、Embedded 開発プラグインに付属しています。

「Launch configuration」の「SVD path」タブを選択し、SVD ファイルが存在するファイルシステム上の場所を入力します。通常は、pico-sdk ソースツリー内にあります。

以下に例を示します。

```
…/pico-sdk/src/rp2040/hardware_regs/rp2040.svd
```

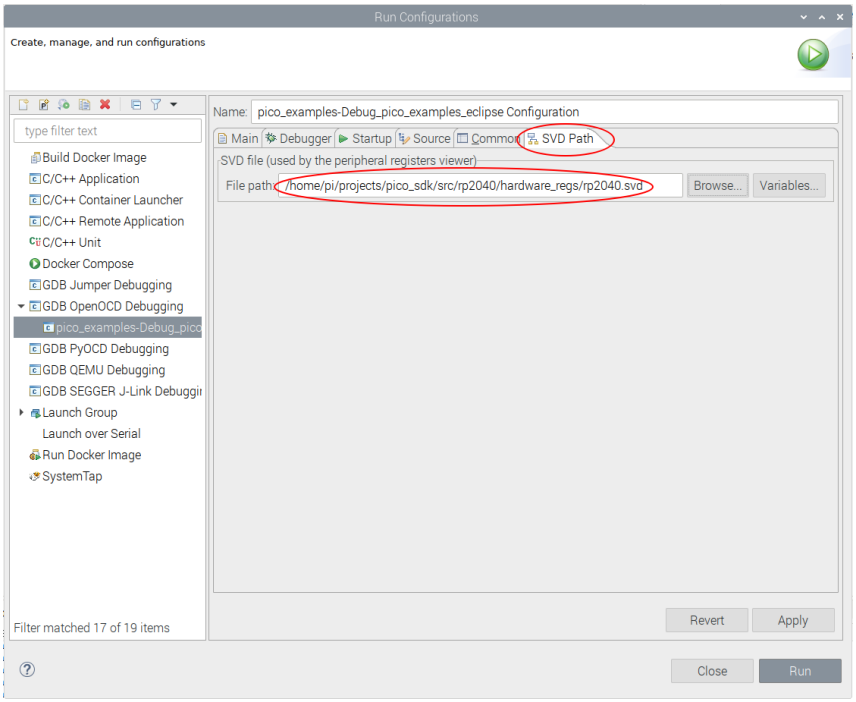


図 26. Eclipse で SVD パスを設定する

10.1.1.5.4. デバッガーの実行

実行設定情報を保存したら、ダイアログ右下の「`Run`」ボタンを使用して、すぐにデバッガーを起動することができます。または、「`Apply`」ボタンを使用して変更内容を適用し、「`Close`」ボタンを使用してダイアログを閉じることもできます。その後、「`Run`」メニューの「`Debug`」オプションを使用して、アプリケーションを実行することができます。

デバッガーを起動すると Eclipse がデバッグモードになり、複数のデバッグウィンドウとソースコードウィンドウが表示されます。また、「Peripherals」ビューも表示されます。これは、SVD データを使用して周辺レジスターにアクセスできる非常に便利なビューです。これで、Eclipse の標準的なデバッグセッションが開始されました。

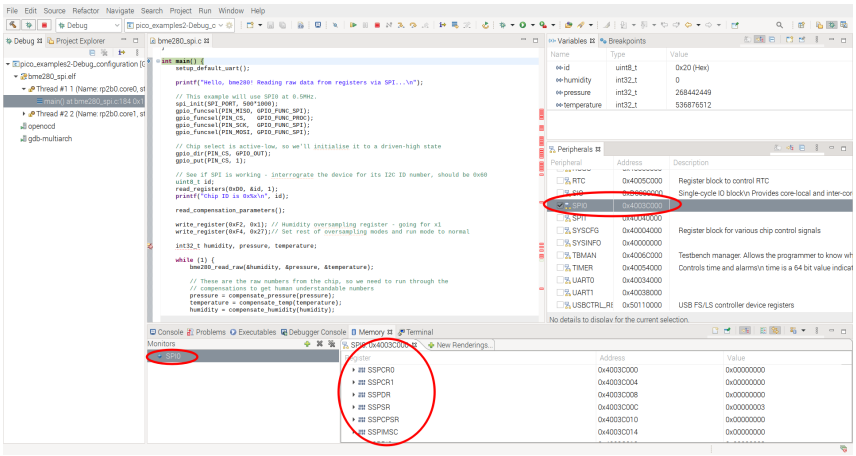


図 27. Eclipse デバッガーの起動時に表示されるデバッグウィンドウの例

10.2. CLion を使用する

CLion は、JetBrains 社が提供するマルチプラットフォームの統合開発環境 (IDE) で、Linux、Windows、Mac に対応しています。 CLion は、プロの開発者や JetBrains 社の IDE を好むユーザーがよく使用する商用 IDE ですが、 廉価版ライセンスや無料ライセンスもあります。CLion を Raspberry Pi で使用することはできますが、最適な性能で使用することはできません。 そのため、デスクトップパソコンやノートパソコンで CLion を使用することをお勧めします。

プロジェクトの設定、開発、ビルドは簡単に行うことができますが、デバッグの設定については、現時点ではまだ完成されていない部分があるため、注意が必要です。

10.2.1. CLion のセットアップ

CLion がインストールされていない場合は、 https://www.jetbrains.com/clion/ からダウンロードしてインストールすることができます。

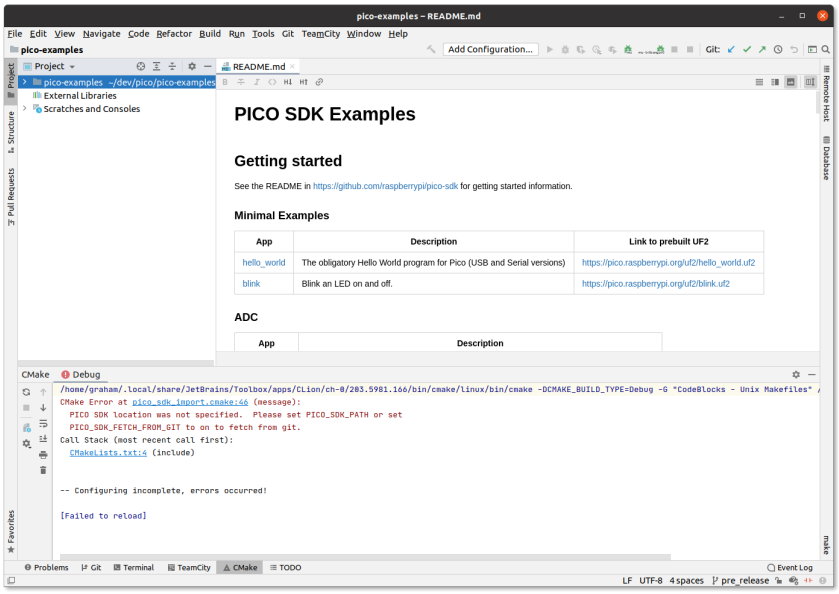
10.2.1.1. プロジェクトの設定

ここでは、サンプルプロジェクトとして pico-examples を使用します。

pico-examples プロジェクトを開くには、「`File`」メニューで「`Open…`」を選択し、チェックアウトした `pico-examples` ディレクトリを選択して「OK」を押します。

図 28 のような画面が表示されます。

図 28. CLion の pico-examples サンプルプロジェクト



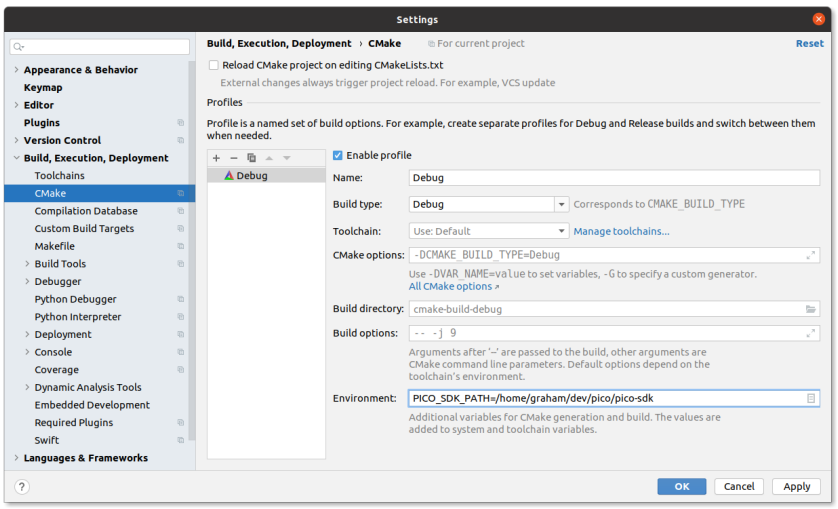
画面の下部にエラーメッセージが表示されていますが、これは「CLion が CMake プロジェクトを読み込もうとしたが、`PICO_SDK_PATH` が指定されていないためにエラーが発生した」ということを表しています。

10.2.1.1.1. CMake プロファイルの設定

「`File`」メニューで「`Settings…`」を選択し、「`Build, Execution, Deployment`」で「CMake」を選択します。

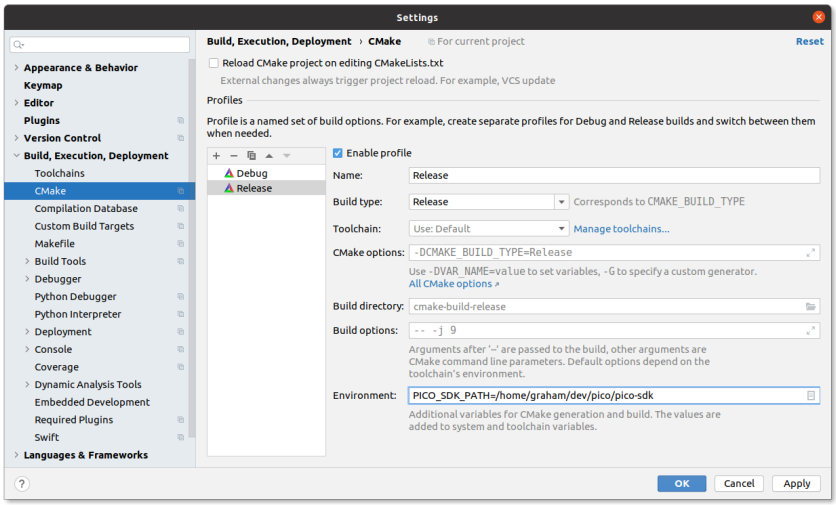
「`Environment:`」で `PICO_SDK_PATH` 環境変数を設定する (図 29 を参照) ことも、「`CMake options:`」で `-DPICO_SDK_PATH=xxx` を環境変数として設定することもできます。 これらの変数は、コマンドラインから `cmake` を呼び出す際に指定する環境変数やコマンドライン引数と同様に機能するため、`PICO_BOARD` や `PICO_TOOLCHAIN_PATH` などの CMake 設定はこれらの変数により指定されることになります。

図 29. CLion で CMake プロファイルを設定する



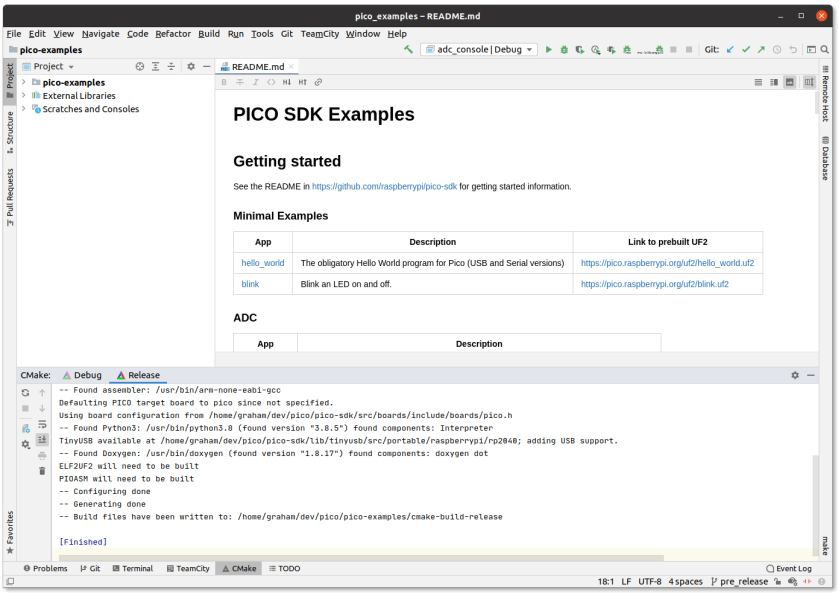
CMake プロファイルは、異なる設定で必要な数だけ作成することができます。`Release` プロファイルを追加する場合は、「+」ボタンを押して PICO_SDK_PATH 変数を再設定するか、「+」ボタンの 2 つ右にあるコピーボタンを押して、プロファイルの名前と設定を編集します (図 30 を参照)。

図 30. CLion で 2 つ目の CMake プロファイルを設定する



「OK」を押すと、図 31 のような画面が表示されます。この画面で、2 つのプロファイル (`Debug` と `Release`) 用として 2 つのタブが表示されていることに注意してください。 この画面では `Release` が選択されているため、CMake プロファイルが正常に設定されたことが確認できます。

図 31. CLion で 2 つ目の CMake プロファイルを設定する



10.2.1.1.2. ビルドの実行

ここでは、1 つ以上のターゲットをビルドします。たとえば、ツールバー中央のドロップダウンセレクターで「`hello_usb`」を選択または入力し、左側のツールアイコンを押すことにより、ターゲットをビルドすることができます (図 32 を参照)。または、「`Build`」メニューで、すべてのターゲットのフルビルドやその他のタイプのビルドを行うことができます。

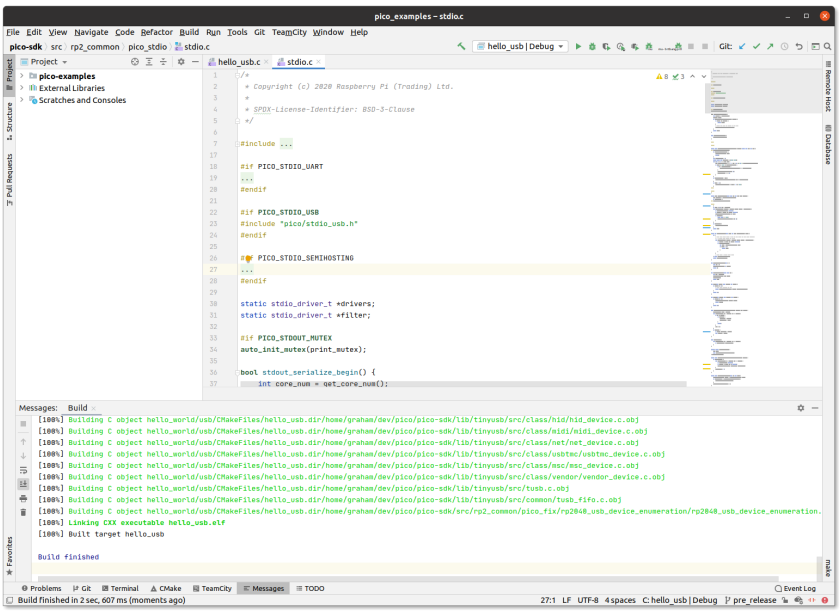


図 32. `hello_usb` が正常にビルドされたことを示す画面

ドロップダウンセレクターで、ビルドするターゲットと使用する CMake プロファイル (この場合は、`Debug` または `Release`) の両方を選択することができます。

ウィンドウ下部のステータスバーにも、`hello_usb` と `Debug` が表示されます (図 32 を参照)。 エディターの構文ハイライト機能などを制御するためのターゲットと CMake プロファイルが表示されます (以前は、`hello_usb` を選択すると、自動的に選択される仕組みになっていました)。 開いている `stdio.c` ファイルで、PICO_STDIO_USB が設定されていることが視覚的に確認できますが、 PICO_STDIO_UART は設定されていません (これらの変数は hello_usb の設定の一部です)。ライブラリのバイナリ設定別のビルド時間は、その大部分が SDK で費やされるため、視覚的に確認できる機能を使用すると便利です。

10.2.1.1.3. ビルドの成果物

ビルドの成果物は、プロジェクトのルートディレクトリ内の `cmake-build-<profile>` ディレクトリに保管されます (図 33 を参照)。この例では、`cmake-build-debug` ディレクトリに保管されます。

UF2 ファイルは、BOOTSEL モードの RP2040 デバイスにコピーすることができます。ELF ファイルをデバッグで使用することもできます。

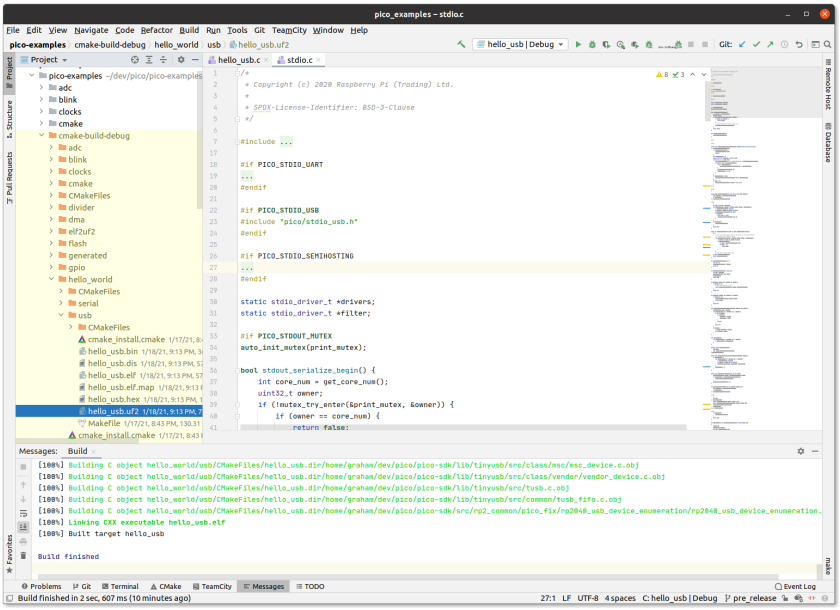


図 33. `hello_usb` ビルドの成果物

10.3. その他の環境

開発環境にはさまざまなものがあるため、そのすべてをここで紹介することはできませんが、多くの開発環境を SDK で使用することができます。Raspberry Pi Pico で IDE を使用するには、以下が必要になります。

- CMake 統合機能
- リモートオプションによる GDB サポート機能
- SVD (必須ではありませんが、周辺機器の状況を簡単に把握できるようになります)
- オプションの ARM 組み込み開発プラグイン (多くの場合、このようなプラグインにより、サポート作業が非常に容易になります)

10.3.1. openocd-svd を使用する

`openocd-svd` ツールは、Python ベースの GUI ユーティリティです。このツールを使用して OpenOCD と CMSIS-SVD を組み合わせることにより、ARM MCU の周辺レジスターにアクセスすることができます。

このツールをインストールする前に、以下のコマンドで依存関係をインストールする必要があります。

```
$ sudo apt install python3-pyqt5
$ pip3 install -U cmsis-svd
```

次に、以下のコマンドで `openocd-svd` という Git リポジトリーを複製します。

```
$ cd ~/pico
$ git clone https://github.com/esynr3z/openocd-svd.git
```

Raspberry Pi 4 と Raspberry Pi Pico が正しく配線されていれば、`swd` 設定と `rp2040` 設定を使用して、OpenOCD をチップに接続することができます。

```
$ openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg
```

警告

DORMANT モードのコードやシステムクロックを停止させるコードがフラッシュに含まれている場合、システムクロックが停止するため、デバッガーの接続が失敗します。この場合、ボードの動作が停止したように見えますが、ボタンを押せば問題なく BOOTSEL モードに戻ることができます。

使用する OpenOCD ターミナルは、稼働状態のままにしておく必要があります。次に、別のターミナルを起動し、OpenOCD に gdb インスタンスを接続します。

プロジェクトに移動して `gdb` を起動します。

```
$ cd ~/pico/test
$ gdb-multiarch test.elf
```

GDB を OpenOCD に接続します。

```
(gdb) target remote localhost:3333
```

OpenOCD をフラッシュに読み込んで起動します。

```
(gdb) load
(gdb) monitor reset init
(gdb) continue
```

`openocd` と `gdb` の両方が稼働している状態で 3 つ目のウィンドウを開き、SDK の SVD ファイルを指すように `openocd-svd` を起動します。

```
$ python3 openocd_svd.py /home/pi/pico/pico-sdk/src/rp2040/hardware_regs/rp2040.svd
```

これにより、`openocd-svd` ウィンドウが表示されます。次に、「File」メニューで「Connect OpenOCD」をクリックし、稼働中の `openocd` インスタンスに Telnet 経由で接続します。

これにより、Raspberry Pi Pico で実行されているコードのレジスターを確認できるようになります (図 34 を参照)。

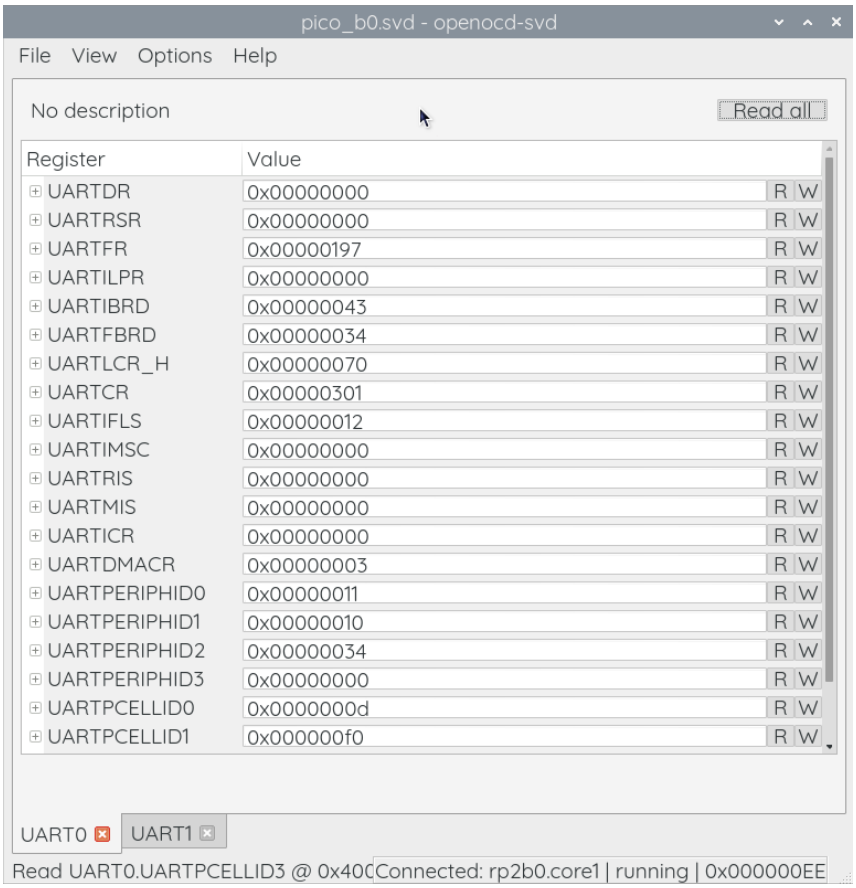


図 34. Raspberry Pi Pico に接続されている稼働中の OpenOCD SVD

付録 A: Picoprobe を使用する

Pico を、USB → SWD および USB - UART ブリッジに変換する `picoprobe` ファームウェアを使用すると、1 台の Raspberry Pi Pico を使用して、別の Pico の再プログラミングやデバッグを行うことができます。これにより、Raspberry Pi 以外のプラットフォームで (UART や SWD に直接接続するための GPIO はないが、USB ポートはあるプラットフォーム。例: Windows、Mac、Linux など)、Raspberry Pi Pico を簡単に使用することができます。

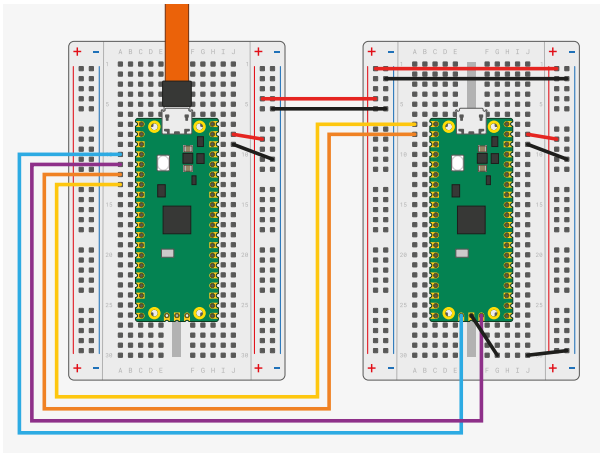


図 35. Pico A (左) と Pico B (右) の配線図。この場合、Pico A はデバッグプローブとして機能します。1台の Pico を使用して、別の Pico の再プログラミングやデバッグを行うには、少なくとも接地ワイヤと 2 本の SWD ワイヤを接続する必要があります。この図は、UART シリアルポートを接続し、テスト対象の Pico の UART シリアル出力を確認する方法と、両方のボードを 1 本の USB ケーブルで接続して電源を供給する方法を示しています。詳細については、「Picoprobe の配線」セクションを参照してください。

OpenOCD のビルド

Picoprobe を使用するには、Picoprobe ドライバーを有効にして OpenOCD をビルドする必要があります。

Linux

```
$ cd ~/pico
$ sudo apt install automake autoconf build-essential texinfo libtool libftdi-dev libusb-1.0-0-dev
$ git clone https://github.com/raspberrypi/openocd.git --branch rp2040 --depth=1
$ cd openocd
$ ./bootstrap
$ ./configure ①
$ make -j4
$ sudo make install
```

1. Raspberry Pi でビルドを行う場合は、`--enable-sysfsgpio --enable-bcm2835gpio` を指定すると、GPIO ピン経由で SWD をビットバンギングすることができます。

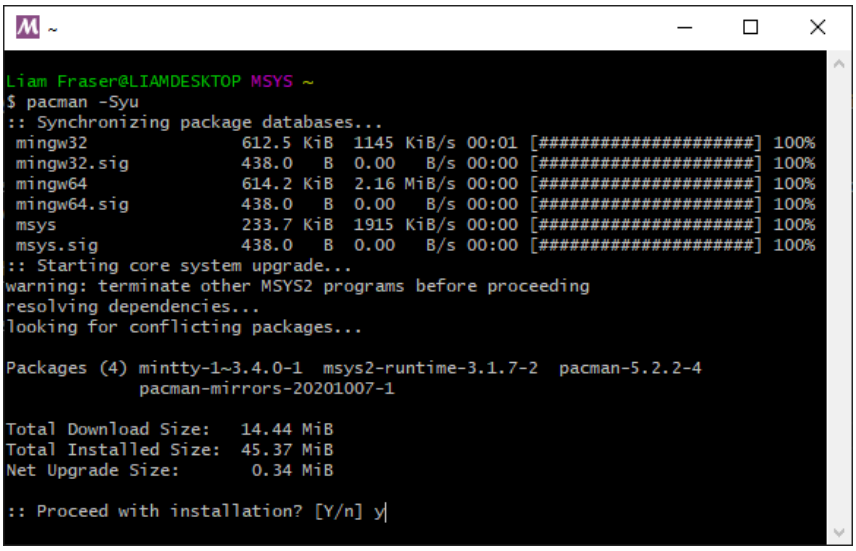
Windows

OpenOCD をできるだけ簡単にビルドするため、ここでは MSYS2 を使用します。MSYS2 の Web サイトには、次のような説明があります:「MSYS2 は、Windows ネイティブソフトウェアのビルド、インストール、起動を行うための使いやすい環境を実現する各種のツールとライブラリのコレクションです」

https://www.msys2.org/ からインストーラーをダウンロードして実行します。

最初に以下のコマンドを実行して、パッケージデータベースとコアシステムパッケージを更新します。

```
$ pacman -Syu
```



MSYS2 が終了した場合は、もう一度起動して (その際、64 ビット版が選択されていることを確認してください)、以下のコマンドを実行します。

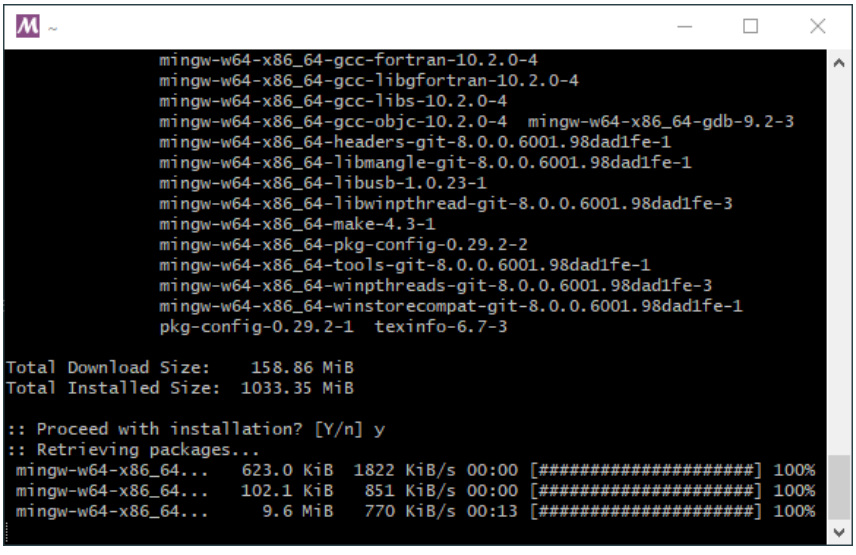
```
$ pacman -Su
```

これにより、更新処理が完了します。

以下のコマンドを実行して、必要な依存関係をインストールします。

```
$ pacman -S mingw-w64-x86_64-toolchain git make libtool pkg-config autoconf automake texinfo mingw-w64-x86_64-
libusb
```

mingw-w64-x86_64 ツールチェインをインストールする際に、Enter キーを押してすべてを選択します。



MSYS2 を終了して 64 ビット版をもう一度起動し、現在の環境で GCC が選択されることを確認します。

```
$ git clone https://github.com/raspberrypi/openocd.git --branch rp2040 --depth=1
$ cd openocd
$ ./bootstrap
$ ./configure --disable-werror ①
$ make -j4
```

1. Windows では、すべてがクリーンな状態でコンパイルされるわけではないため、`disable-werror` オプションを指定する必要があります。

最後に OpenOCD を実行し、正しくビルドされているかどうかを確認します。ここでは設定オプションが指定されていないため、以下のようなエラーが表示されます。

```
$ src/openocd.exe
Open On-Chip Debugger 0.10.0+dev-gc231502-dirty (2020-10-14-14:37)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
embedded:startup.tcl:56: Error: Can't find openocd.cfg
in procedure 'script'
at file "embedded:startup.tcl", line 56
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Error: Debug Adapter has to be specified, see "interface" command
embedded:startup.tcl:56: Error:
in procedure 'script'
at file "embedded:startup.tcl", line 56
```

Mac

必要に応じて、Homebrew をインストールします。

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

依存関係をインストールします。

```
$ brew install libtool automake libusb wget pkg-config gcc texinfo ①
```

1. OSX に付属している `texinfo` のバージョンは、OpenOCD ドキュメントのビルドに必要なバージョンよりも古いバージョンです。

```
$ cd ~/pico
$ git clone https://github.com/raspberrypi/openocd.git --branch rp2040 --depth=1
$ cd openocd
$ export PATH="/usr/local/opt/texinfo/bin:$PATH" ①
$ ./bootstrap
$ ./configure --disable-werror ②
$ make -j4
```

1. texinfo の新しいバージョンをパス上で指定します。
2. OSX では、すべてがクリーンな状態でコンパイルされるわけではないため、`disable-werror` オプションを指定する必要があります。

OpenOCD を実行して、正しくビルドされているかどうかを確認します。ここでは設定オプションが指定されていないため、以下のようなエラーが表示されます。

```
$ src/openocd
Open On-Chip Debugger 0.10.0+dev-gc231502-dirty (2020-10-15-07:48)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
embedded:startup.tcl:56: Error: Can't find openocd.cfg
in procedure 'script'
at file "embedded:startup.tcl", line 56
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Error: Debug Adapter has to be specified, see "interface" command
embedded:startup.tcl:56: Error:
in procedure 'script'
at file "embedded:startup.tcl", line 56
```

Picoprobe のビルドとフラッシュ

> Picoprobe UF2 のダウンロード
>
> Picoprobe の UF2 バイナリは、Raspberry Pi Pico Documentation ページの「Software Utilities」セクションからダウンロードすることができます。「Raspberry Pi Pico」セクションをクリックして「Software Utilities」までスクロールダウンし、「Debugging using another Raspberry Pi Pico」というタイトルの下に表示されている UF2 ファイルをダウンロードしてください。

ここに記載されているビルド手順は、SDK がインストールされている Linux 上で実行することを前提としています。

```
$ cd ~/pico
$ git clone https://github.com/raspberrypi/picoprobe.git
$ cd picoprobe
$ git submodule update --init
$ mkdir build
$ cd build
$ cmake ..
$ make -j4
```

`BOOTSEL` ボタンを押して、デバッガーとして使用する Raspberry Pi Pico を起動し、`picoprobe.uf2` ファイルにドラッグします。

Picoprobe の配線

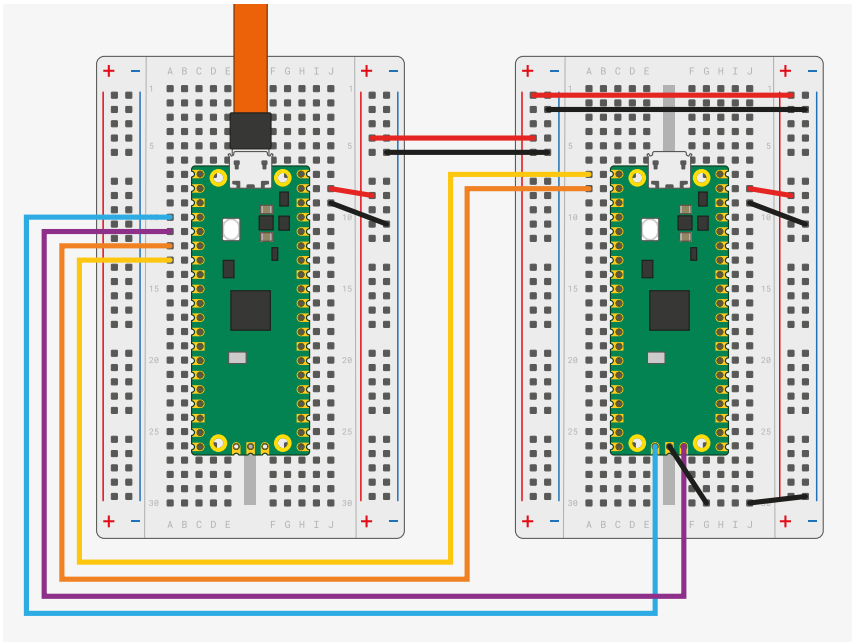


図 36. Pico A (左側) と Pico B (右側) の配線図 (Pico A をデバッガーとして使用)。Pico B を USB ホストとして使用する場合は、VSYS と VSYS を接続するのではなく、VBUS と VBUS を接続して 5V 電源を供給する必要があります。

図 36 は、2 枚の Pico ボードの配線を示しています。

```
Pico A GND -> Pico B GND
Pico A GP2 -> Pico B SWCLK
Pico A GP3 -> Pico B SWDIO
Pico A GP4/UART1 TX -> Pico B GP1/UART0 RX
Pico A GP5/UART1 RX -> Pico B GP0/UART0 TX
```

OpenOCD 経由でコードを読み込んで実行するための最小限の接続構成は、GND、SWCLK、SWDIO です。UART の配線を接続することにより、Pico A (左側) の USB 接続を経由して Pico B (右側) の UART シリアルポートと通信できるようになります。また、UART の配線だけを使用して、別の UART シリアルデバイス (Raspberry Pi 上のブートコンソールなど) と通信することもできます。

オプションとして、Pico B から Pico A に電源を供給するための配線も必要になります。

```
Pico A VSYS -> Pico B VSYS
```

重要

Pico B を USB ホストとして使用する場合、VSYS と VSYS を接続するのではなく、VBUS と VBUS を接続して、Pico B の USB コネクタで 5V 電源を供給する必要があります。**Pico B で USB をデバイスモードで使用する場合や、USB をまったく使用しない場合は、この処理は必要ありません.

Picoprobe ドライバーのインストール (Windows の場合のみ)

Picoprobe デバイスには、2 つの USB インターフェイスがあります。

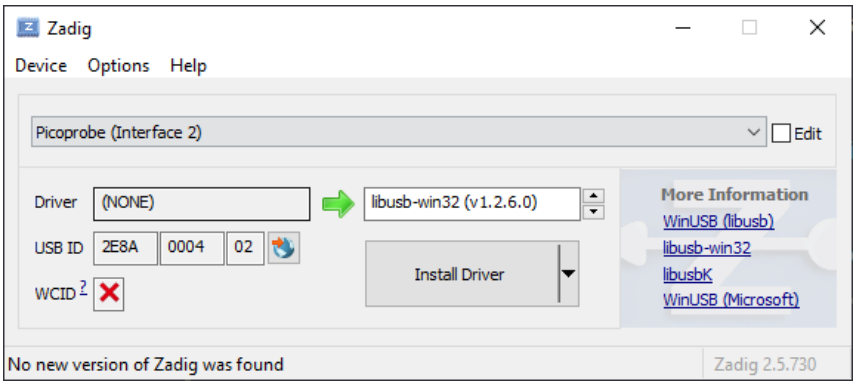
1. クラス準拠の CDC UART (シリアルポート): Windows ですぐに使用することができます。

2. SWD プローブデータ用のベンダー固有インターフェイス: ドライバーをインストールする必要があります。

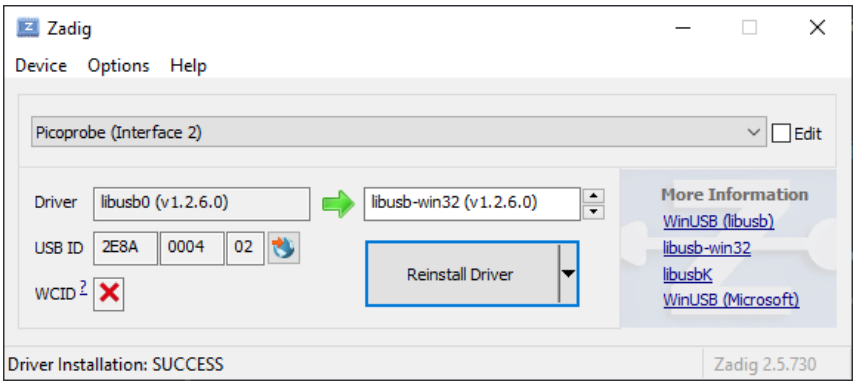
ここでは、Zadig (http://zadig.akeo.ie) を使用します。

以下の手順で、Zadig のダウンロードとインストールを行ってください。

ドロップダウンボックスで「Picoprobe (Interface 2)」を選択します。ドライバーとして「libusb-win32」を選択します。



「Install Driver」を選択します。



Picoprobe の UART を使用する

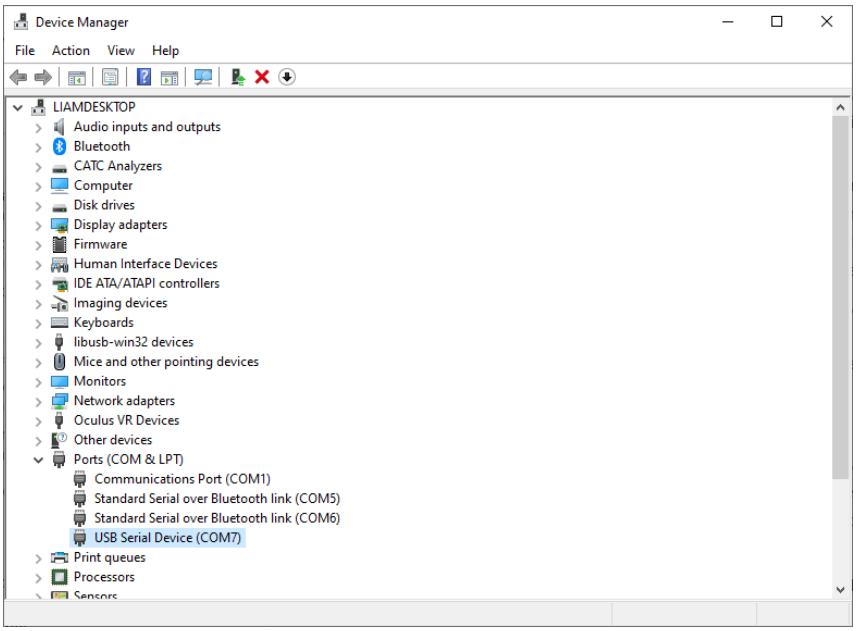
Linux

```
$ sudo minicom -D /dev/ttyACM0 -b 115200
```

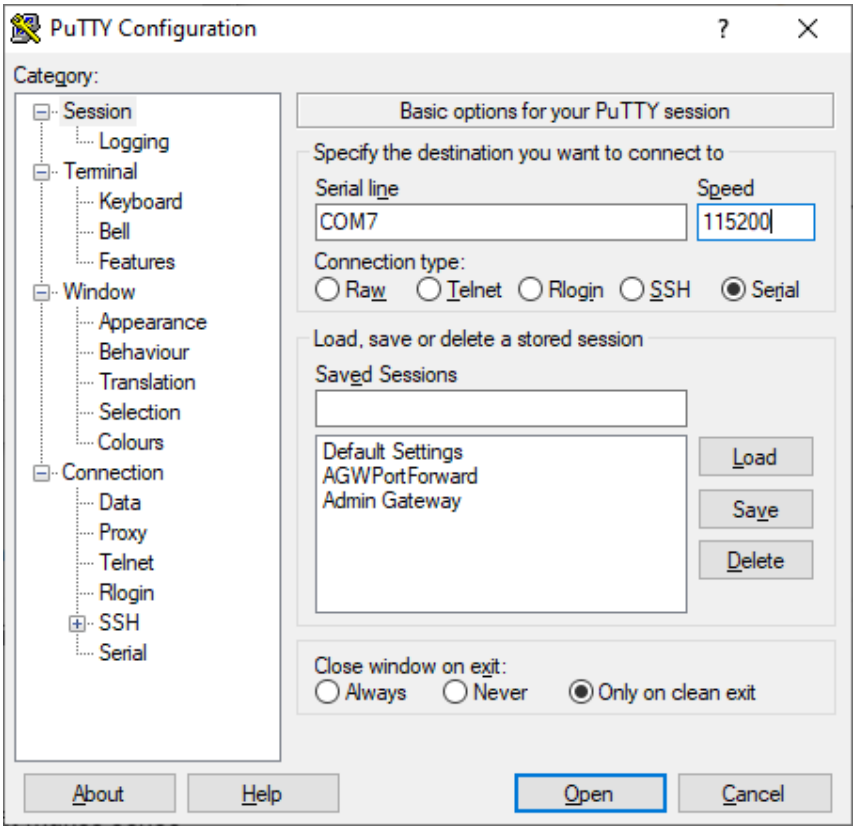
Windows

https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html から PuTTY をダウンロードしてインストールします。

Device Manager を開き、Picoprobe の COM ポート番号を確認します。この例では、`COM7` になっています。



PuTTY を起動し、「Connection type」で「`Serial`」を選択します。次に、COM ポートの名前 (COM7) と速度 (115200) を入力します。



「Open」を選択してシリアルコンソールを起動します。これで、アプリケーションを実行するための準備が整いました。

Mac

```
$ brew install minicom
$ minicom -D /dev/tty.usbmodem1234561 -b 115200
```

OpenOCD で Picoprobe を使用する (すべてのプラットフォームで共通)

以下のコマンドを実行します。

```
$ src/openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -s tcl
```

以下のコマンドを実行して、GDB を接続します。

```
target remote localhost:3333
```

付録 B: Picotool を使用する

Raspberry Pi Pico のバイナリに情報を組み込むことができます。`picotool` というコマンドラインユーティリティを使用して、この情報を取得することができます。

picotool の入手

`picotool` ユーティリティは、専用のリポジトリから入手することができます。pico-setup スクリプトを実行しなかった場合は、このユーティリティの複製とビルドを行う必要があります。

```
$ git clone https://github.com/raspberrypi/picotool.git --branch master
$ cd picotool
```

また、`libusb` をインストールする必要があります (まだインストールされていない場合)。

```
$ sudo apt install libusb-1.0-0-dev
```

注記

macOS で `picotool` をビルドする場合は、Homebrew を使用して `libusb` をインストールすることができます。

```
$ brew install libusb pkg-config
```

Microsoft Windows でビルドを行う場合は、`libusb` の Windows バイナリを libusb.info サイトから直接ダウンロードしてインストールすることができます。

picotool のビルド

`picotool` をビルドするには、以下のように入力します。

```
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=~/pico/pico-sdk
$ cmake ../
$ make
```

これにより、`picotool` コマンドラインバイナリが `build/picotool` ディレクトリ内に生成されます。

注記

Microsoft Windows でビルドを行う場合は、以下のように入力して CMake を起動します。

```
C:\Users\pico\picotool> mkdir build
C:\Users\pico\picotool> cd build
C:\Users\pico\picotool\build> cmake .. -G "NMake Makefiles"
C:\Users\pico\picotool\build> nmake
```

picotool の使用

`picotool` バイナリには、コマンドラインヘルプ機能が組み込まれています。

```
$ picotool help
PICOTOOL:
    Tool for interacting with a RP2040 device in BOOTSEL mode, or with a RP2040 binary

SYNOPSYS:
    picotool info [-b] [-p] [-d] [-l] [-a] [--bus <bus>] [--address <addr>] [-f] [-F]
    picotool info [-b] [-p] [-d] [-l] [-a] <filename> [-t <type>]
    picotool load [-n] [-N] [-v] [-x] <filename> [-t <type>] [-o <offset>] [--bus <bus>]
                [--address <addr>] [-f] [-F]
    picotool save [-p] [--bus <bus>] [--address <addr>] [-f] [-F] <filename> [-t <type>]
    picotool save -a [--bus <bus>] [--address <addr>] [-f] [-F] <filename> [-t <type>]
    picotool save -r <from> <to> [--bus <bus>] [--address <addr>] [-f] [-F] <filename> [-t
                <type>]
    picotool verify [--bus <bus>] [--address <addr>] [-f] [-F] <filename> [-t <type>] [-r
                <from> <to>] [-o <offset>]
    picotool reboot [-a] [-u] [--bus <bus>] [--address <addr>] [-f] [-F]
    picotool version [-s]
    picotool help [<cmd>]

COMMANDS:
    info      Display information from the target device(s) or file.
              Without any arguments, this will display basic information for all connected
              RP2040 devices in BOOTSEL mode
    load      Load the program / memory range stored in a file onto the device.
    save      Save the program / memory stored in flash on the device to a file.
    verify    Check that the device contents match those in the file.
    reboot    Reboot the device
    version   Display picotool version
    help      Show general help or help for a specific command

Use "picotool help <cmd>" for more info
```

注記

多くのコマンドは、BOOTSEL モードの RP2040 デバイスを接続して実行する必要があります。

重要

「No accessible RP2040 devices in BOOTSEL mode were found.」というエラーメッセージが「Device at bus 1, address 7 appears to be a RP2040 device in BOOTSEL mode, but picotool was unable to connect」のような通知とともに表示された場合は、Raspberry Pi Pico が接続されています。この場合は、sudo などのコマンドを使用して picotool を実行することができます。

```
$ sudo picotool info -a
```

Windows でこのメッセージが表示された場合は、Picoprobe 用のドライバーと同様のドライバーを追加することができます (「Picoprobe ドライバーのインストール (Windows の場合のみ)」を参照)。

バージョン 1.1 の picotool では、picotool の -f 引数を使用することにより、BOOTSEL モードになっていない、RP2040 の USB stdio サポート機能を使用する SDK デバイスを操作することができます。

情報の表示

SDK ではバイナリ情報がサポートされているため、picotool によって検索されたサイズの小さな情報を簡単に保存することができます (これ以降の「バイナリ情報」セクションを参照)。info コマンドを使用して、この情報を読み取ることができます。

この情報は、BOOTSEL モードで接続された 1 つ以上の RP2040 デバイスから読み取ることも、 ファイルから読み取ることもできます。このファイルは、ELF ファイル、UF2 ファイル、BIN ファイルのいずれかになります。

```
$ picotool help info
INFO:
    Display information from the target device(s) or file.
    Without any arguments, this will display basic information for all connected RP2040 devices
    in BOOTSEL mode

SYNOPSYS:
    picotool info [-b] [-p] [-d] [-l] [-a] [--bus <bus>] [--address <addr>] [-f] [-F]
    picotool info [-b] [-p] [-d] [-l] [-a] <filename> [-t <type>]

OPTIONS:
    Information to display
        -b, --basic
            Include basic information. This is the default
        -p, --pins
            Include pin information
        -d, --device
            Include device information
        -l, --build
            Include build attributes
        -a, --all
            Include all information

TARGET SELECTION:
    To target one or more connected RP2040 device(s) in BOOTSEL mode (the default)
        --bus <bus>
            Filter devices by USB bus number
        --address <addr>
            Filter devices by USB device address
        -f, --force
            Force a device not in BOOTSEL mode but running compatible code to reset so the
            command can be executed. After executing the command (unless the command itself is
            a 'reboot') the device will be rebooted back to application mode
        -F, --force-no-reboot
            Force a device not in BOOTSEL mode but running compatible code to reset so the
            command can be executed. After executing the command (unless the command itself is
            a 'reboot') the device will be left connected and accessible to picotool, but
            without the RPI-RP2 drive mounted
    To target a file
```

```
        <filename>
            The file name
        -t <type>
            Specify file type (uf2 | elf | bin) explicitly, ignoring file extension
```

たとえば Raspberry Pi Pico をマスストレージモードでコンピューターに接続する場合、BOOTSEL ボタンを長押ししてから USB に接続します。次に、ターミナルウィンドウを開いて以下のように入力します。

```
$ sudo picotool info
Program Information
 name:      hello_world
 features:  stdout to UART
```

または、以下のように入力します。

```
$ sudo picotool info -a
Program Information
 name:          hello_world
 features:      stdout to UART
 binary start:  0x10000000
 binary end:    0x1000606c

Fixed Pin Information
 20:  UART1 TX
 21:  UART1 RX

Build Information
 build date:        Dec 31 2020
 build attributes:  Debug build

Device Information
 flash size:   2048K
 ROM version:  2
```

このように入力すると、詳細な情報を確認することができます。または、以下のように入力すると、使用するピンの情報だけを確認することができます。

```
$ sudo picotool info -bp
Program Information
 name:      hello_world
 features:  stdout to UART

Fixed Pin Information
 20:  UART1 TX
 21:  UART1 RX
```

picotool は、ローカルファイルシステム上のバイナリに対して使用することもできます。

```
$ picotool info -a lcd_1602_i2c.uf2
File lcd_1602_i2c.uf2:

Program Information
 name:          lcd_1602_i2c
 web site:      https://github.com/raspberrypi/pico-examples/tree/HEAD/i2c/lcd_1602_i2c
 binary start:  0x10000000
 binary end:    0x10003c1c

Fixed Pin Information
 4:  I2C0 SDA
 5:  I2C0 SCL

Build Information
```

```
 build date:  Dec 31 2020
```

プログラムの保存

メモリーの範囲、特定のプログラム、デバイスのフラッシュ全体を BIN ファイルまたは UF2 ファイルに保存することができます。

```
$ picotool help save
SAVE:
    Save the program / memory stored in flash on the device to a file.

SYNOPSYS:
    picotool save [-p] [--bus <bus>] [--address <addr>] [-f] [-F] <filename> [-t <type>]
    picotool save -a [--bus <bus>] [--address <addr>] [-f] [-F] <filename> [-t <type>]
    picotool save -r <from> <to> [--bus <bus>] [--address <addr>] [-f] [-F] <filename> [-t
                <type>]

OPTIONS:
    Selection of data to save
        -p, --program
            Save the installed program only. This is the default
        -a, --all
            Save all of flash memory
        -r, --range
            Save a range of memory. Note that UF2s always store complete 256 byte-aligned
            blocks of 256 bytes, and the range is expanded accordingly
        <from>
            The lower address bound in hex
        <to>
            The upper address bound in hex
    Source device selection
        --bus <bus>
            Filter devices by USB bus number
        --address <addr>
            Filter devices by USB device address
        -f, --force
            Force a device not in BOOTSEL mode but running compatible code to reset so the
            command can be executed. After executing the command (unless the command itself is
            a 'reboot') the device will be rebooted back to application mode
        -F, --force-no-reboot
            Force a device not in BOOTSEL mode but running compatible code to reset so the
            command can be executed. After executing the command (unless the command itself is
            a 'reboot') the device will be left connected and accessible to picotool, but
            without the RPI-RP2 drive mounted
    File to save to
        <filename>
            The file name
        -t <type>
            Specify file type (uf2 | elf | bin) explicitly, ignoring file extension
```

以下に例を示します。

```
$ sudo picotool info
Program Information
name:      lcd_1602_i2c
web site:  https://github.com/raspberrypi/pico-examples/tree/HEAD/i2c/lcd_1602_i2c
$ picotool save spoon.uf2
Saving file: [============================]  100%
Wrote 51200 bytes to spoon.uf2
$ picotool info spoon.uf2
File spoon.uf2:
Program Information
name:      lcd_1602_i2c
web site:  https://github.com/raspberrypi/pico-examples/tree/HEAD/i2c/lcd_1602_i2c
```

バイナリ情報

バイナリ情報は、マシン上で検索することができます。通常、バイナリ情報はマシン上で使用されます。だれでも任意のバイナリ情報を追加して使用できますが、その情報を自己記述型の情報にするかどうかについては、自分自身で決める必要があります。

基本的な情報

Pico について確認する場合は、基本的な情報を確認することをお勧めします。

基本的な情報としては、以下のような情報があります。

- プログラムの名前
- プログラムの説明
- プログラムのバージョンを表す文字列
- プログラムのビルド日付
- プログラムの URL
- プログラムの終了アドレス
- SDK で表示可能なプログラム機能 (バイナリ内の個々の文字列から作成されたリスト。例: UART stdio 用のリストや USB stdio 用のリストなど)
- ビルド属性 (バイナリ自体に関する文字列のリスト。「Debug Build」など)

ピン

`hello_serial.elf` という実行ファイルをどの RP2040 ベースのボードに対してビルドしたのかを忘れた場合は、ピン情報を確認すると便利です。これは、各ボードによってピンの割り当てが異なる場合があるためです。

静的な (固定された) ピンの割り当て情報を、以下のような非常にシンプルな形式でバイナリに記録することができます。

```
$ picotool info --pins sprite_demo.elf
File sprite_demo.elf:

Fixed Pin Information
0-4:    Red 0-4
6-10:   Green 0-4
11-15:  Blue 0-4
16:     HSync
17:     VSync
18:     Display Enable
19:     Pixel Clock
20:     UART1 TX
21:     UART1 RX
```

バイナリ情報の挿入

バイナリ情報は、マクロを使用してプログラム内で宣言します。以下に例を示します。

```
$ picotool info --pins sprite_demo.elf
File sprite_demo.elf:

Fixed Pin Information
0-4:    Red 0-4
6-10:   Green 0-4
11-15:  Blue 0-4
16:     HSync
17:     VSync
18:     Display Enable
19:     Pixel Clock
20:     UART1 TX
21:     UART1 RX
```

`setup_default_uart` 関数には、以下のような行があります。

```
bi_decl_if_func_used(bi_2pins_with_func(PICO_DEFAULT_UART_RX_PIN, PICO_DEFAULT_UART_TX_PIN, GPIO_FUNC_UART));
```

これにより、2 つのピン番号と UART 関数が格納され、実際の関数名 (UART1 TX など) が picotool によってデコードされます。`bi_decl_if_func_used` は、指定された関数が呼び出された場合にのみバイナリ情報を挿入するためのコードです。

同様に、ビデオコードにも以下のような行が含まれています。

```
bi_decl_if_func_used(bi_pin_mask_with_name(0x1f << (PICO_SCANVIDEO_COLOR_PIN_BASE +
PICO_SCANVIDEO_DPI_PIXEL_RSHIFT), "Red 0-4"));
```

詳細

できるだけスペースを無駄にしないような設計になっていますが、`PICO_NO_BINARY_INFO=1` というプリプロセッサー変数を使用すると、すべてをオフにすることができます。また、専用のプリプロセッサー変数を使用して、バイナリ情報を挿入する SDK コードを個別に除外することができます。

バイナリ情報を挿入するには、以下のように入力します。

```
#include "pico/binary_info.h"
```

ヘッダーには、多くの `bi_` マクロが含まれています。

```
#define bi_binary_end(end)
#define bi_program_name(name)
#define bi_program_description(description)
#define bi_program_version_string(version_string)
#define bi_program_build_date_string(date_string)
#define bi_program_url(url)
#define bi_program_feature(feature)
#define bi_program_build_attribute(attr)
#define bi_1pin_with_func(p0, func)
#define bi_2pins_with_func(p0, p1, func)
#define bi_3pins_with_func(p0, p1, p2, func)
#define bi_4pins_with_func(p0, p1, p2, p3, func)
#define bi_5pins_with_func(p0, p1, p2, p3, p4, func)
#define bi_pin_range_with_func(plo, phi, func)
#define bi_pin_mask_with_name(pmask, label)
#define bi_pin_mask_with_names(pmask, label)
#define bi_1pin_with_name(p0, name)
```

```
#define bi_2pins_with_names(p0, name0, p1, name1)
#define bi_3pins_with_names(p0, name0, p1, name1, p2, name2)
#define bi_4pins_with_names(p0, name0, p1, name1, p2, name2, p3, name3)
```

以下のような基本的なマクロが使用されます。

```
#define bi_program_url(url) bi_string(BINARY_INFO_TAG_RASPBERRY_PI, BINARY_INFO_ID_RP_PROGRAM_URL, url)
```

バイナリ情報を無条件に挿入する場合は、`bi_decl(bi_blah(…))` を使用します。 エンクロージャー関数がリンカーによってバイナリに挿入されないときにバイナリ情報を削除する場合は (`--gc-sections` など)、`bi_decl_if_func_used(bi_blah(…))` を使用します。

以下に例を示します。

```
 1 #include <stdio.h>
 2 #include "pico/stdlib.h"
 3 #include "hardware/gpio.h"
 4 #include "pico/binary_info.h"
 5
 6 const uint LED_PIN = 25;
 7
 8 int main() {
 9
10     bi_decl(bi_program_description("This is a test binary."));
11     bi_decl(bi_1pin_with_name(LED_PIN, "On-board LED"));
12
13     setup_default_uart();
14     gpio_set_function(LED_PIN, GPIO_FUNC_PROC);
15     gpio_set_dir(LED_PIN, GPIO_OUT);
16     while (1) {
17         gpio_put(LED_PIN, 0);
18         sleep_ms(250);
19         gpio_put(LED_PIN, 1);
20         puts("Hello World\n");
21         sleep_ms(1000);
22     }
23 }
```

`picotool` を使用してクエリーを実行すると、以下のような出力が表示されます。

```
$ sudo picotool info -a test.uf2
File test.uf2:

Program Information
 name:          test
 description:   This is a test binary.
 features:      stdout to UART
 binary start:  0x10000000
 binary end:    0x100031f8

Fixed Pin Information
 0:   UART0 TX
 1:   UART0 RX
 25:  On-board LED

Build Information
 build date:  Jan  4 2021
```

このように、文字列が出力されます。

CMake による共通フィールドの設定

プロジェクトの CMake ファイルから直接フィールドを設定することもできます。

```
pico_set_program_name(foo "not foo") ①
pico_set_program_description(foo "this is a foo")
pico_set_program_version_string(foo "0.00001a")
pico_set_program_url(foo "www.plinth.com/foo")
```

1. この場合、「foo」がデフォルトのプログラム名になります。

注記

これらの共通フィールドは、すべてコマンドライン引数としてコンパイル処理に渡されます。引用符や改行記号などを定義する場合は、コード内で `bi_decl` を使用して定義することをお勧めします。

付録 C: 本文書のリリース履歴

表 1. 本文書のリリース履歴

リリース	日付	説明
1.0	2021 年 1 月 21 日	• 初回リリース
1.1	2021 年 1 月 26 日	• マイナーな修正 • ADC での DMA の使用に関する情報を追加 • M0+ レジスタと SIO CPUID レジスタに関する説明を追加 • タイマーについてより詳しい説明を追加 • Windows および macOS のビルド手順を更新 • 書籍名を変更、出力 PDF のサイズを最適化
1.2	2021 年 2 月 1 日	• マイナーな修正 • PIO ドキュメントのマイナーな改善 • 不足していた TIMER2 レジスタと TIMER3 レジスタを DMA に追加 • UART で MicroPython REPL を取得する方法を追加 • C SDK の V1.0.1 リリースに伴う情報を追加
1.3	2021 年 2 月 23 日	• マイナーな修正 • フォントを変更 • RP2040 のシンク/ソース制限に関するドキュメントを追加 • SWD ドキュメントのメジャーな改善 • MicroPython のビルド手順を更新 • MicroPython UART のサンプルコードを追加 • Thonny の手順を更新 • プロジェクト生成機能の手順を更新 • FAQ ドキュメントを追加 • 正誤表 E7、E8、E9 を追加
1.3.1	2021 年 3 月 5 日	• マイナーな修正 • C SDK の V1.1.0 リリースに伴う情報を追加 • MicroPython UART のサンプルを改善 • ピンの配置図を改善

リリース	日付	説明
1.4	2021 年 4 月 7 日	• マイナーな修正 • 正誤表 E10 を追加 • Github から C SDK を更新する方法に関する注意事項を追加 • C SDK の V1.1.2 リリースに伴う情報を追加
1.4.1	2021 年 4 月 13 日	• マイナーな修正 • ドキュメントに記載されているすべてのソースコードが 3 条項 BSD ライセンスに含まれているという説明を追加
1.5	2021 年 6 月 7 日	• マイナーな更新と修正 • FAQ を更新 • SDK のリリース履歴を追加 • C SDK の V1.2.0 リリースに伴う情報を追加
1.6	2021 年 6 月 23 日	• マイナーな更新と修正 • ADC 情報を更新 • 正誤表 E11 を追加
1.6.1	2021 年 9 月 30 日	• マイナーな更新と修正 • B2 リリースに関する情報を追加 • B2 リリースの正誤表を更新
1.7	2021 年 11 月 3 日	• マイナーな更新と修正 • 一部のレジスタのアクセスの種類と説明を修正 • コア 1 の起動シーケンスに関する情報を追加 • SDK の「パニック」処理に関する説明を追加 • picotool ドキュメントを更新 • *付録 A:アプリに関するメモ*付録 (「Raspberry Pi Pico C/C++ SDK」) に例を追加 • C SDK の V1.3.0 リリースに伴う情報を追加
1.7.1	2021 年 11 月 4 日	• マイナーな更新と修正 • USB ダブルバッファリングのドキュメントを改善 • Picoprobe ブランチを変更 • ドキュメントへのリンクを更新

最新リリースは、 https://datasheets.raspberrypi.com/pico/getting-started-with-pico-JP.pdf で参照することができます。

Raspberry Pi Ltd