Getting started with Raspberry Pi Pico-series

C/C++ development with Raspberry Pi Pico-series and other Raspberry Pi microcontroller-based boards

Colophon

Copyright © 2020-2024 Raspberry Pi Ltd (formerly Raspberry Pi (Trading) Ltd.)

The documentation of the RP2350 microcontroller is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND).

build-date: 2024-08-21
build-version: 522d2d4-clean

About the SDK

Throughout the text "the SDK" refers to our Raspberry Pi Pico SDK. More details about the SDK can be found in the **Raspberry Pi Pico-series C/C++ SDK** book. Source code included in the documentation is Copyright © 2023-2024 Raspberry Pi Ltd (formerly Raspberry Pi (Trading) Ltd.) and licensed under the 3-Clause BSD license.

Legal disclaimer notice

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.

Table of contents

Colophon . . . 1
Legal disclaimer notice . . . 1
1. Introduction . . . 4
2. Install Visual Studio Code . . . 5
3. Install the Raspberry Pi Pico VS Code Extension . . . 6
3.1. Install Dependencies . . . 6
3.1.1. Raspberry Pi OS and Windows . . . 6
3.1.2. Linux . . . 6
3.1.3. macOS . . . 6
3.2. Install the Extension . . . 7
4. Load and debug a project . . . 8
4.1. Compile and Run `blink` . . . 9
4.2. Make a Code Change and Re-run . . . 9
4.3. Debug . . . 10
5. Say "Hello World" in C . . . 13
5.1. Serial input and output on Pico-series devices . . . 13
5.2. Create a project . . . 13
5.3. Build your project . . . 13
5.4. See console output . . . 14
Appendix A: Debugprobe . . . 15
Building OpenOCD . . . 15
Install OpenOCD . . . 15
Debug Probe . . . 16
Debug Probe wiring . . . 16
Debug with a second Pico or Pico 2 . . . 17
Install `debugprobe` . . . 18
`debugprobe` wiring . . . 18
Debug Probe interfaces . . . 19
Use the UART . . . 19
Linux . . . 19
Windows . . . 20
macOS . . . 21
Debug with OpenOCD . . . 22
Debug with SWD . . . 22
Appendix B: Picotool . . . 24
Getting picotool . . . 24
Building picotool . . . 24
Using picotool . . . 25
Display information . . . 26
Save the program . . . 28
Binary Information . . . 29
Basic information . . . 29
Pins . . . 30
Full Information . . . 30
Appendix C: Manual toolchain setup . . . 31
Configure your environment via Script . . . 31
Manually Configure your Environment . . . 32
Get the SDK and examples . . . 32
Install the Toolchain . . . 32
Enable UART serial communications . . . 32
Update the SDK . . . 33
Use the CLI to Blink an LED in C . . . 33
Building "Blink" . . . 34
Load and run "Blink" . . . 35
Manually Create your own Project . . . 37

Debugging your project . . . 39
Appendix D: Use other Integrated Development Environments . . . 41
Use Eclipse . . . 41
Setting up Eclipse for Pico on a Linux machine . . . 41
Use CLion . . . 46
Setting up CLion . . . 46
Other Environments . . . 50
Appendix H: Documentation Release History . . . 51
August 9 2024 . . . 51
August 8 2024 . . . 51

Chapter 1. Introduction

To follow this guide you will need the following:

- Raspberry Pi Pico-series device
- a Micro USB cable

The following are required for some of the later steps:

- Raspberry Pi Debug Probe, or a second Raspberry Pi Pico-series device

The following instructions assume that you are using a Pico-series device; some details may differ if you use a different Raspberry Pi microcontroller-based board.

Pico-series devices are built around microcontrollers designed by Raspberry Pi. Development on the boards is fully supported with both a C/C++ SDK, and an official MicroPython port. This book talks about how to get started with the SDK, and walks you through how to build, install, and work with the SDK toolchain.

TIP

The main method covered in this book uses a VS Code extension to make your life easy. If you would like to set up your development environment manually, see Manually Configure your Environment.

For more information on the official MicroPython port, see Raspberry Pi Pico-series Python SDK and Get started with MicroPython on Raspberry Pi Pico. For more information on the C/C++ SDK, see Raspberry Pi Pico-series C/C++ SDK.

Chapter 2. Install Visual Studio Code

Visual Studio Code (VS Code) is a popular open source editor developed by Microsoft. The Raspberry Pi Pico VS Code Extension makes it easy to install dependencies and build software for Pico-series devices.

TIP

> If you don't want to use VS Code, you can either use VSCodium (the community-driven libre alternative) or configure your environment manually.

To install Visual Studio Code (VS Code) on **Raspberry Pi OS or Linux**, run the following commands:

```
$ sudo apt update
$ sudo apt install code
```

On **macOS and Windows**, you can install VS Code from https://code.visualstudio.com/Download.

On **macOS**, you can also install VS Code with `brew` using the following command:

```
$ brew install --cask visual-studio-code
```

Chapter 3. Install the Raspberry Pi Pico VS Code Extension

The Raspberry Pi Pico VS Code extension helps you create, develop, run, and debug projects in Visual Studio Code. It includes a project generator with many templating options, automatic toolchain management, one click project compilation, and offline documentation of the Pico SDK.

The VS Code extension supports all Raspberry Pi Pico-series devices.

3.1. Install Dependencies

3.1.1. Raspberry Pi OS and Windows

No dependencies needed.

3.1.2. Linux

Most Linux distributions come preconfigured with all of the dependencies needed to run the extension. However, but some distributions may require additional dependencies. The extension requires the following:

- Python 3.9 or later
- Git
- Tar
- a native C and C++ compiler (the extension supports GCC)

You can install these with:

```
$ sudo apt install python3 git tar build-essential
```

3.1.3. macOS

To install all requirements for the extension on macOS, run the following command:

```
$ xcode-select --install
```

This installs the following dependencies:

- Git
- Tar
- A native C and C++ compiler (the extension supports GCC and Clang)

3.2. Install the Extension

You can find the extension in the VS Code Extensions Marketplace. Search for the **Raspberry Pi Pico** extension, published by **Raspberry Pi**. Click the **Install** button to add it to VS Code.

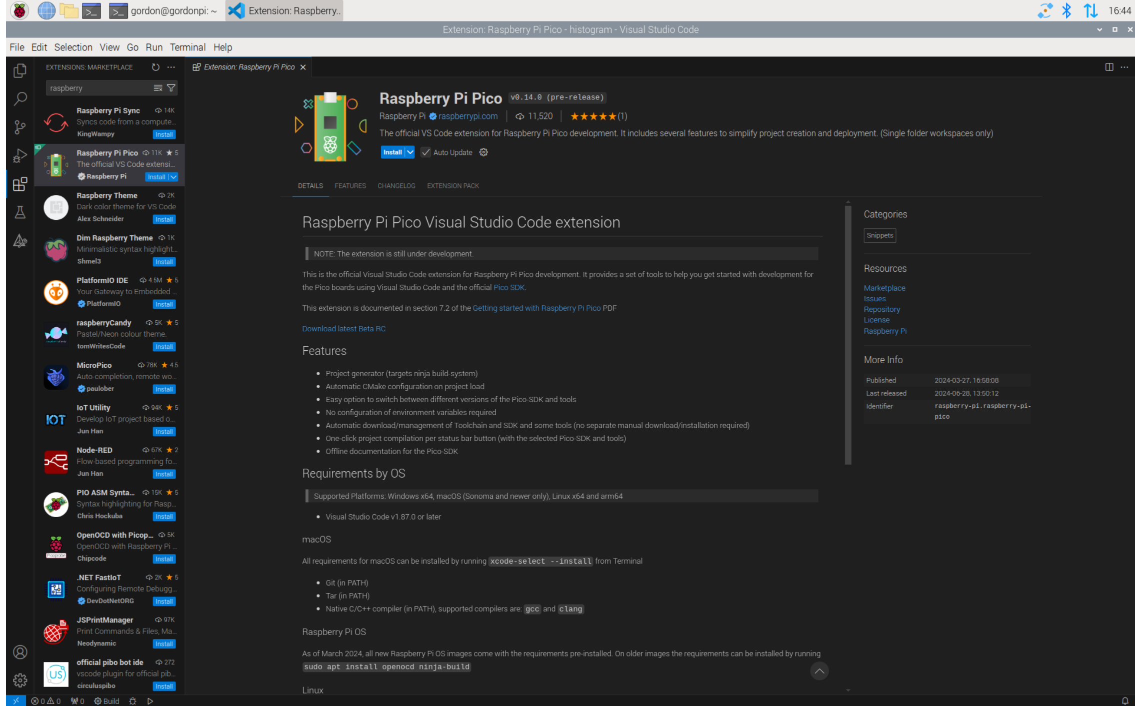


Figure 1. Debugging in VS Code.

You can find the store entry at https://marketplace.visualstudio.com/items?itemName=raspberry-pi.raspberry-pi-pico.

You can find the extension source code and release downloads at https://github.com/raspberrypi/pico-vscode.

When installation completes, check the Activity sidebar (by default, on the left side of VS Code). If installation was successful, a new sidebar section appears with a Raspberry Pi Pico icon, labelled "Raspberry Pi Pico Project".

Chapter 4. Load and debug a project

The VS Code extension can create projects based on the examples provided by Pico Examples. For an example, we'll walk you through how to create a project that blinks the LED on your Pico-series device:

1. In the VS Code left sidebar, select the Raspberry Pi Pico icon, labelled "Raspberry Pi Pico Project".
2. Select **New Project from Examples**.
3. In the **Name** field, select the **blink** example.
4. Choose the board type that matches your device.
5. Specify a folder where the extension can generate files. VS Code will create the new project in a sub-folder of the selected folder.
6. Click **Create** to create the project.

The extension will now download the SDK and the toolchain, install them locally, and generate the new project. The first project may take 5-10 minutes to install the toolchain. VS Code will ask you whether you trust the authors because we've automatically generated the `.vscode` directory for you. Select **yes**.

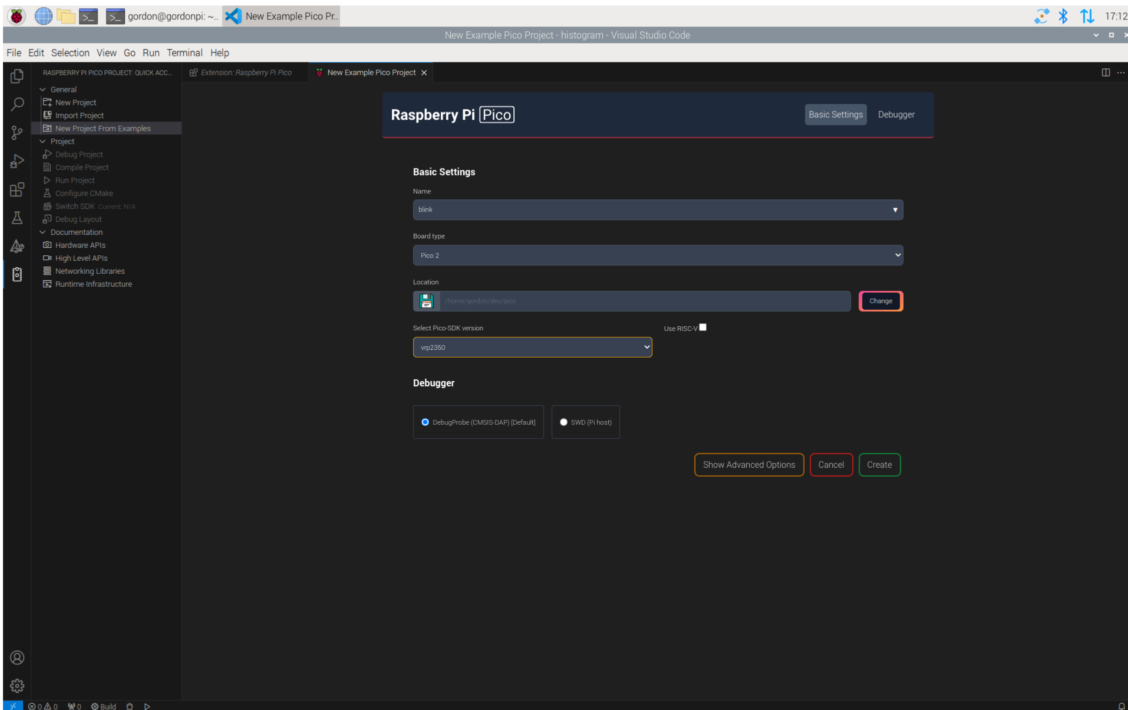


Figure 2. Creating a project in VS Code.

> **NOTE**
>
> The CMake Tools extension may display some notifications at this point. Ignore and close them.

On the left **Explorer** sidebar in VS Code, you should now see a list of files.

Open `blink.c` to view the blink example source code in the main window.

The Raspberry Pi Pico extension adds some capabilities to the status bar at the bottom right of the screen.

Compile

Compiles the sources and builds the target UF2 file. You can copy this binary onto your device to program it.

Run

Finds a connected device, flashes the code into it, and runs that code.

The extension sidebar also contains some quick access functions. Click on the Pico icon in the side menu and you'll see **Compile Project**.

Hit **Compile Project** and a terminal tab will open at the bottom of the screen displaying the compilation progress.

4.1. Compile and Run `blink`

To run the blink example:

1. Hold down the `BOOTSEL` button on your Pico-series device while plugging it into your development device using a micro USB cable to force it into USB Mass Storage Mode.
2. Press the **Run** button in the status bar or the **Run project** button in the sidebar.

You should see the terminal tab at the bottom of the window open. It will display information concerning the upload of the code. Once the code uploads, the device will reboot, and you should see the following output:

```
The device was rebooted to start the application.
```

Your blink code is now running. If you look at your device, the LED should blink twice every second.

4.2. Make a Code Change and Re-run

To check that everything is working correctly, click on the `blink.c` file in VS Code. Navigate to the definition of `LED_DELAY_MS` at the top of the code:

```
#ifndef LED_DELAY_MS
#define LED_DELAY_MS 250
#endif LED_DELAY_MS
```

1. Change the 250ms (a quarter of a second) to 100 (a tenth of a second):

   ```
   #ifndef LED_DELAY_MS
   #define LED_DELAY_MS 100
   #endif LED_DELAY_MS
   ```

2. Disconnect your device, then reconnect while holding the `BOOTSEL` button.
3. Press the **Run** button in the status bar or the **Run project** button in the sidebar.

You should see the terminal tab at the bottom of the window open. It will display information concerning the upload of the code. Once the code uploads, the device will reboot, and you should see the following output:

```
The device was rebooted to start the application.
```

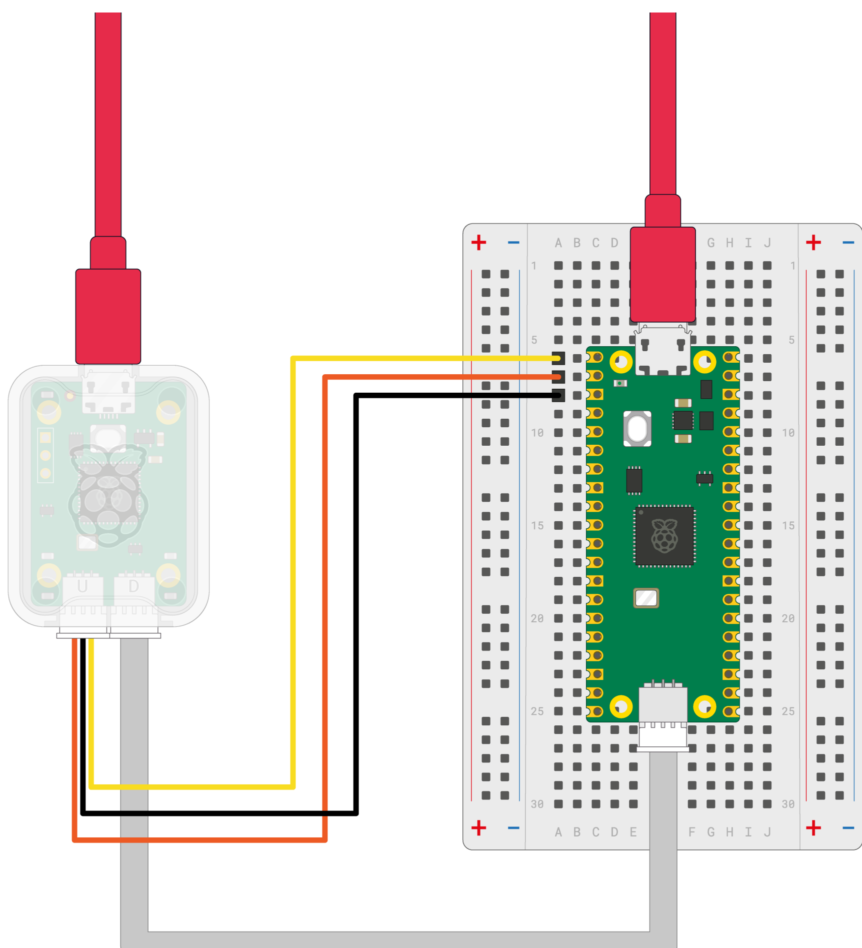
Your blink code is now running. If you look at your device, the LED should flash faster, ten times every second.

4.3. Debug

The Raspberry Pi Debug Probe is a debug solution for any Arm-based computer. You can use other debug hardware with Pico-series devices, but we recommend the Debug Probe to make configuration as simple as possible. If you'd like to use a Pico-series device as a Debug Probe, see Debug with a second Pico or Pico 2.

First, connect the Debug Probe to your Pico-series device through the debug connector on the board. Depending on which Pico device you have, different connectors will be required. For Pico, Pico W, and Pico 2, use a soldering iron to solder the Debug Probe connectors onto the board. For Pico H, Pico WH, and Pico with headers, the debug header is already added. Just connect the Debug Probe with the supplied cable.

Figure 3. Debug Probe wiring



For more information, see the Debug Probe documentation.

Now, plug the Debug Probe USB into your computer. The Debug Probe does not power the Pico device, it must be powered separately.

To start the debugger:

1. Open the extension sidebar by clicking on the Pico icon.
2. Select **Debug Project** or press **F5**.
3. If prompted to select a debugger, choose Pico Debug (Cortex-Debug)

The debugger will automatically download the code to the device, insert a breakpoint at the beginning of your main

function, and run until that breakpoint is hit.

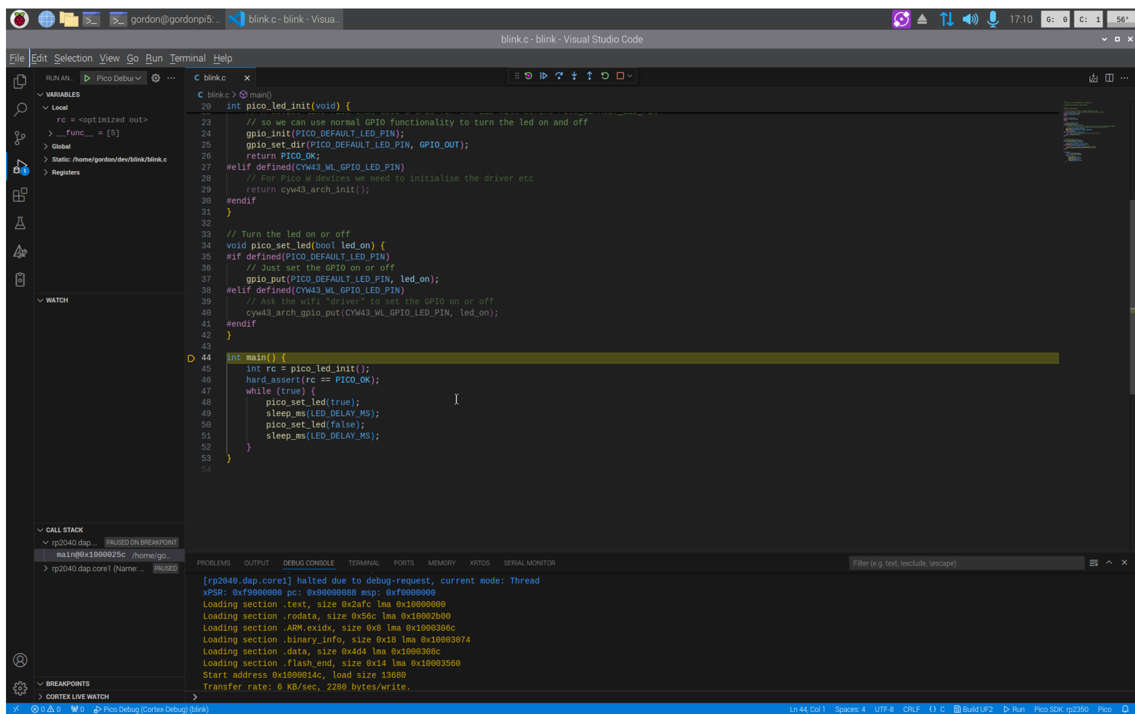


Figure 4. Debugging in VS Code.

Once in debugging mode, the sidebar has a number of windows displaying useful information about the current state of the device. At the top, a small control bar contains buttons that control code execution. Hover over the buttons to identify them. To continue code execution click **Continue** (**F5**).

Your blink code is now running. If you look at your device, the LED should be blinking as before. Now press Restart (**Ctrl+Shift+F5**) to go back to the beginning of `main`.

Press **Step-over** (**F10**) once. The highlighted line, which indicates the next line to be executed, will advance to the `pico_led_init` function call. To step into this function, press **Step-into** (**F11**). The source window will update to indicate execution is now at the beginning of the function. You can either continue to step over code until the function returns to main, or select **Step-out** (**Shift+F11**) to finish executing the function.

After returning to the main function, check the **Local Variables** window to see that the value of `rc` is `0` (`PICO_OK`).

Press Restart (**Ctrl+Shift+F5**) again to go back to the beginning of `main`. Then move the cursor down to the `pico_set_led` line and press **F9**. When you create the breakpoint, you'll see a red dot indicating the breakpoint location:

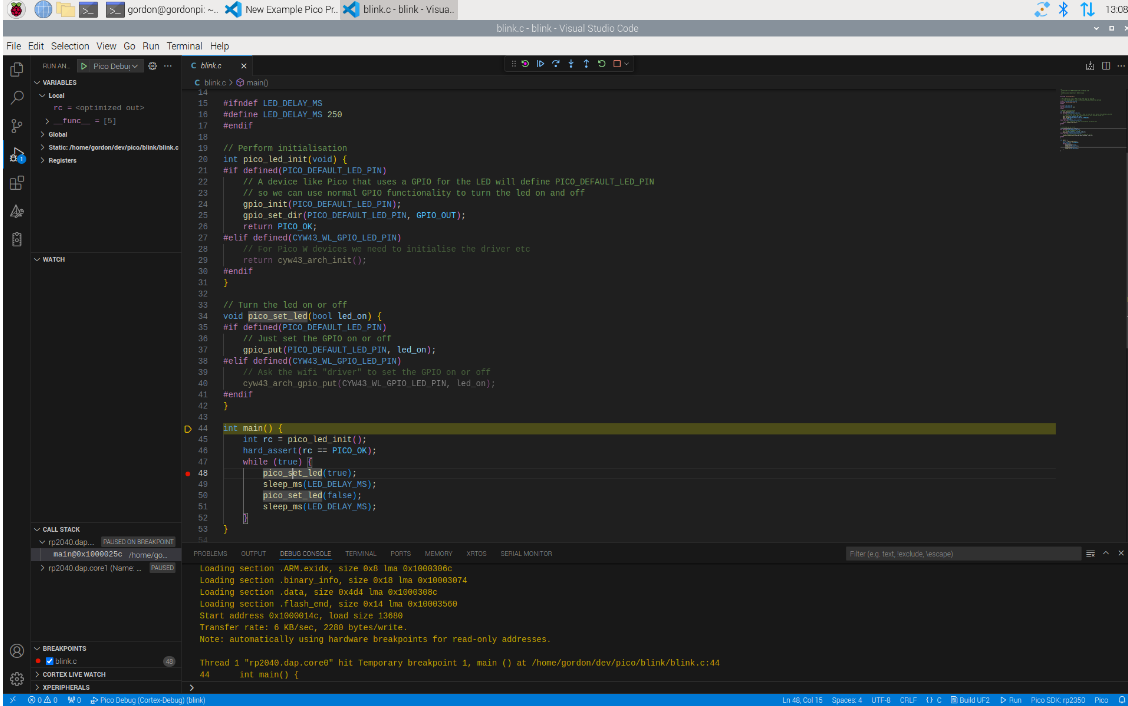


Figure 5. Debugging in VS Code.

You can add and remove a breakpoint by clicking on the red dot.

Press **Continue** (**F5**); execution should halt on the breakpoint. Next, press **Step-over** (**F10**) and you should see the LED light up.

Chapter 5. Say "Hello World" in C

After blinking an LED on and off, the next thing that most developers will want to do is create and use a serial port, and say "Hello World."

5.1. Serial input and output on Pico-series devices

Serial input (`stdin`) and output (`stdout`) can be directed to serial UART and/or to USB CDC (USB serial).

With a serial UART console, the input and output are sent over the UART pins on the device - by default it will use Pin 1 (`GP0`) for sending output (`UART0_TX`) and Pin 2 (`GP1`) for receiving input (`UART0_RX`). You will then need to connect the UART pins on the Pico-series device to a UART to USB converter, such as the Debug Probe, as shown in the Debug Probe Wiring diagram.

With a USB CDC console, the input and ouput are sent directly over the USB cable connected to your computer, so no additional wiring will be needed. However you may miss some of the printout when your code starts running, as your computer may take a second or two to connect to the Pico-series device after it restarts.

You can select either or both consoles when using the extension, depending on your preference.

5.2. Create a project

NOTE

> The SDK makes use of CMake to control its build system, see Manually Create your own Project if you don't want to use the VS Code extension

1. In the VS Code left sidebar, select the Raspberry Pi Pico icon, labelled "Raspberry Pi Pico Project".
2. Select **New Project**.
3. In the **Name** field, name your project. For example "hello_world".
4. Choose the board type that matches your device.
5. Specify a folder where the extension can generate files. VS Code will create the new project in a sub-folder of the selected folder.
6. Under "STDIO support", select which consoles you would like
7. Click **Create** to create the project.

The extension will now generate the new project. VS Code will ask you whether you trust the authors because we've automatically generated the `.vscode` directory for you. Select yes.

5.3. Build your project

To run the "Hello world" example:

1. Hold down the `BOOTSEL` button on your Pico-series device while plugging it into your development device using a micro USB cable to force it into USB Mass Storage Mode.
2. Press the **Run** button in the status bar or the **Run project** button in the sidebar.

You should see the terminal tab at the bottom of the window open. It will display information concerning the upload of the code. Once the code uploads, the device will reboot, and you should see the following output:

```
The device was rebooted to start the application.
```

Your "Hello world" code is now running.

Although the "Hello World" example is now running, we cannot yet see the text.

5.4. See console output

If using STDIO UART make sure you have wired it up first. STDIO USB does not need any wiring other than being connected to your computer.

In VS Code. Go to the view menu, and select "Terminal" to open the bottom pane. In this pane, you will find the "Serial Monitor" tab. Select the serial port. There may be more than one. The baud rate should be `115200`. Select "Start Monitoring" to see the output.

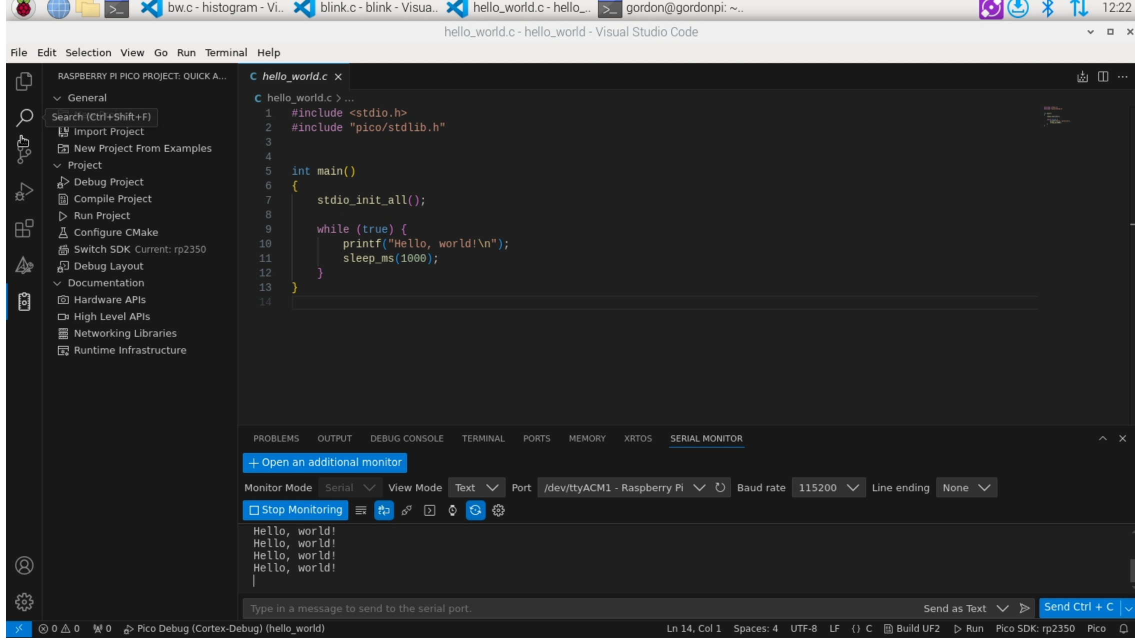


Figure 6. VS Code serial monitor

Appendix A: Debugprobe

Raspberry Pi provides two ways to debug Pico-series devices:

- the Raspberry Pi Debug Probe
- `debugprobe` firmware running on a second Pico or Pico 2

Both methods provide a way to debug Pico-series devices on platforms that lack GPIOs to connect directly to UART or SWD, such as Windows, macOS, and Linux. The debugging device connects to your usual computer using USB, and to the Pico using SWD and UART.

Building OpenOCD

Shortly after RP2350 launch you will likely need to build openocd from source if not using the VS Code extension. You can get a binary release from https://github.com/raspberrypi/pico-sdk-tools.

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

To start openocd from the build directory, you can use:

For RP2350:

```
sudo src/openocd -s tcl -f interface/cmsis-dap.cfg -f target/rp2350.cfg -c "adapter speed 5000"
```

For RP2040:

```
sudo src/openocd -s tcl -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed 5000"
```

Install OpenOCD

To get started, you'll need OpenOCD.

To install OpenOCD, run the following command in a terminal:

```
$ sudo apt install openocd
```

To install OpenOCD on macOS, run the following command:

```
$ brew install openocd
```

To run OpenOCD, use the `openocd` command in your terminal.

Debug Probe

The simplest way to debug a Pico-series device is the Raspberry Pi Debug Probe. The Raspberry Pi Debug Probe provides Serial Wire Debug (SWD), and a generic USB-to-Serial bridge.

NOTE

For more information about the Debug Probe, see the documentation site.

Debug Probe wiring

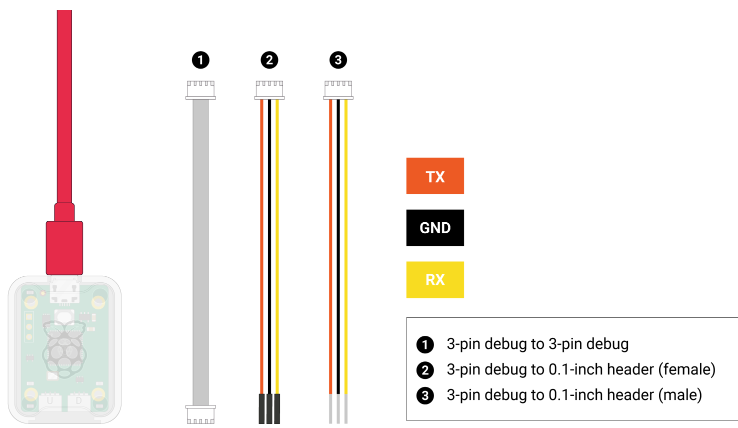


Figure 7. Wires included with the Debug Probe.

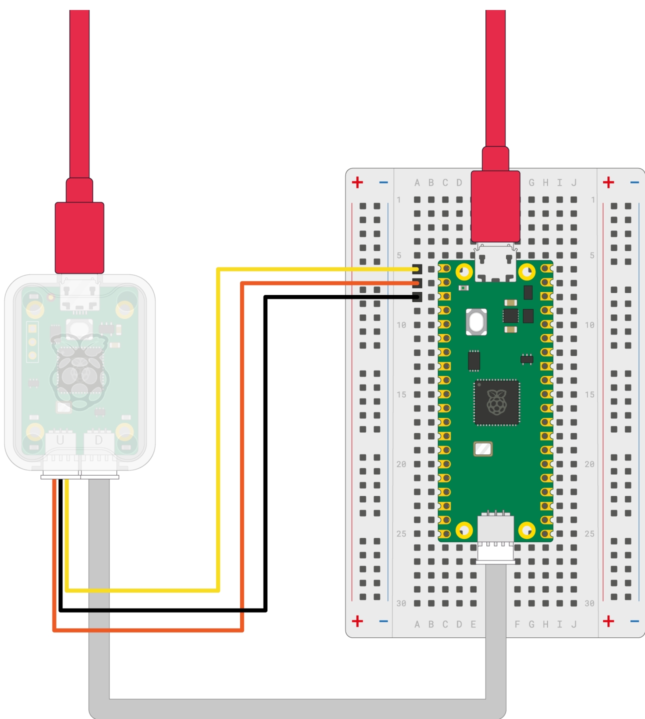


Figure 8. Wiring between the Debug Probe (left) and Pico (right).

To connect Debug Probe to Pico H, connect the following:

- Debug Probe "D" port to Pico H "DEBUG" SWD JST-SH connector
- Debug Probe "U" port, with the three-pin JST-SH connector to 0.1-inch header (male):
  - Debug Probe `RX` connected to Pico H `TX` pin
  - Debug Probe `TX` connected to Pico H `RX` pin
  - Debug Probe `GND` connected to Pico H `GND` pin

Then, connect two USB cables: one from your computer to the microUSB port on Debug Probe and another from your computer to the microUSB port on Pico.

NOTE

If you have a non-H Pico, Pico 2 or Pico W (without a JST-SH connector) you can still connect it to a Debug Probe. Solder a male connector to the `SWCLK`, `GND`, and `SWDIO` header pins on the board. Using the alternate 3-pin JST-SH connector to 0.1-inch header (female) cable included with the Debug Probe, connect to the Debug Probe "D" port. Connect `SWCLK`, `GND`, and `SWDIO` on the Pico or Pico W to the `SC`, `GND`, and `SD` pins on the Debug Probe, respectively.

The wiring loom between Pico and the Debug Probe is shown in Figure 7.

Debug with a second Pico or Pico 2

One Pico or Pico 2 can reprogram and debug another using the `debugprobe` firmware, which transforms the Pico or Pico 2 into a USB → SWD and UART bridge.

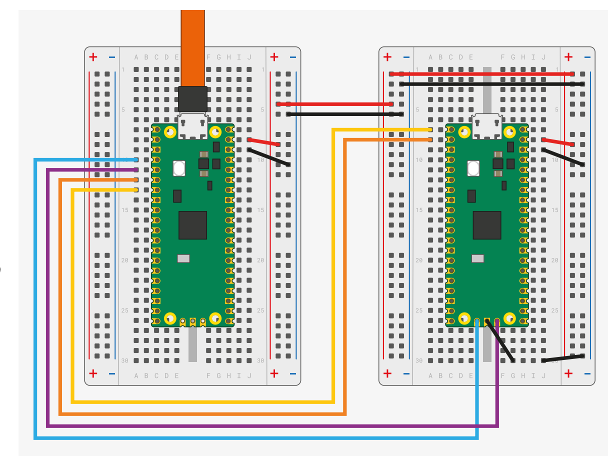


Figure 9. Wiring between Pico A (left) and Pico B (right) with Pico A acting as a debugger and Pico B as a system under test. You must connect at least the ground and the two SWD wires. Connect the UART serial port to provide access to the UART serial output of Pico B. You can also bridge the power supply to power both boards with one USB cable. For more information, see `debugprobe` wiring.

Install `debugprobe`

You can download a UF2 binary of `debugprobe` from the Pico-series documentation.

Boot the debugger Pico or Pico 2 with the `BOOTSEL` button pressed. Copy `debugprobe_on_pico.uf2` onto the device to begin debugging.

NOTE

> Use `debugprobe_on_pico.uf2` to use a Pico for debugging. Use `debugprobe.uf2` for the Debug Probe accessory hardware.

Build `debugprobe`

Alternatively, you can build `debugprobe` using the following instructions:

These build instructions assume you are running on Linux, and have installed the SDK.

NOTE

> These instructions are for Pico; replace the `-DPICO_BOARD=pico` with `-DPICO_BOARD=pico2` for Pico 2

```
$ cd ~/pico
$ git clone https://github.com/raspberrypi/debugprobe.git
$ cd debugprobe
$ git submodule update --init
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake -DDEBUG_ON_PICO=ON -DPICO_BOARD=pico ..
$ make -j4
```

Boot the debugger Pico or Pico 2 with the `BOOTSEL` button pressed. Copy `debugprobe.uf2` onto the device to begin debugging.

`debugprobe` wiring

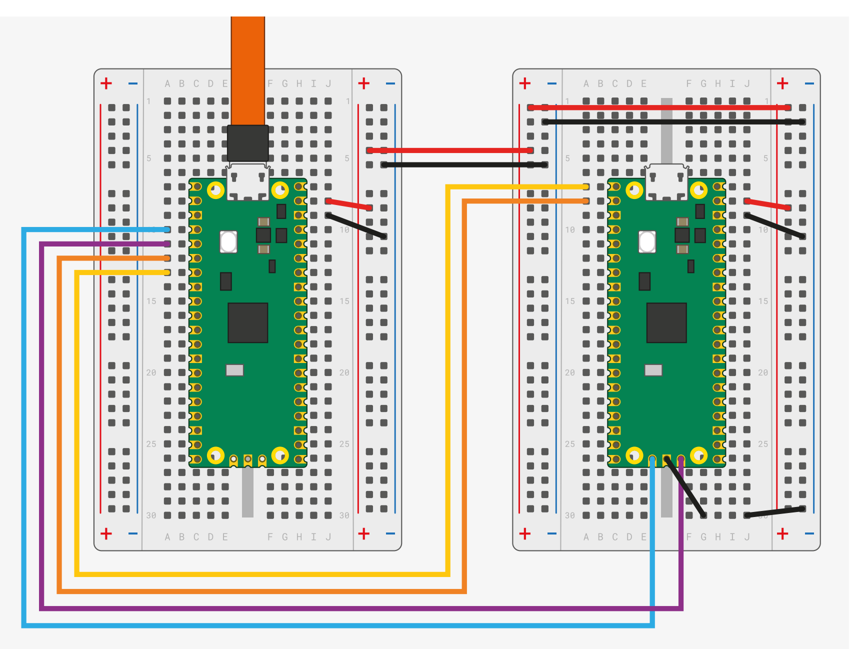


Figure 10. Wiring between Pico A (left) and Pico B (right), configuring Pico A as a debugger.

The wiring loom between the two Pico boards is shown in Figure 10.

```
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
```

The minimum set of connections required to load and run code via OpenOCD is `GND`, `SWCLK` and `SWDIO`. Connect the UART wires to communicate with Pico B's UART serial port through Pico A's USB connection. You can also use the UART wires to talk to any other UART serial device, such as the boot console on a Raspberry Pi.

To power Pico A with Pico B, connect the following pins:

- When using USB in device mode, or not at all, connect `VSYS` to `VSYS`
- When acting as a USB Host, connect `VBUS` to `VBUS` to provide 5V on the USB connector.

Debug Probe interfaces

Both the Debug Probe and any Pico-serires device running `debugprobe` are composite devices with two USB interfaces:

1. A class-compliant CDC UART (serial port), so it works on Windows out of the box.
2. A vendor-specific interface for SWD probe data conforming to CMSIS-DAP v2.

Use the UART

Linux

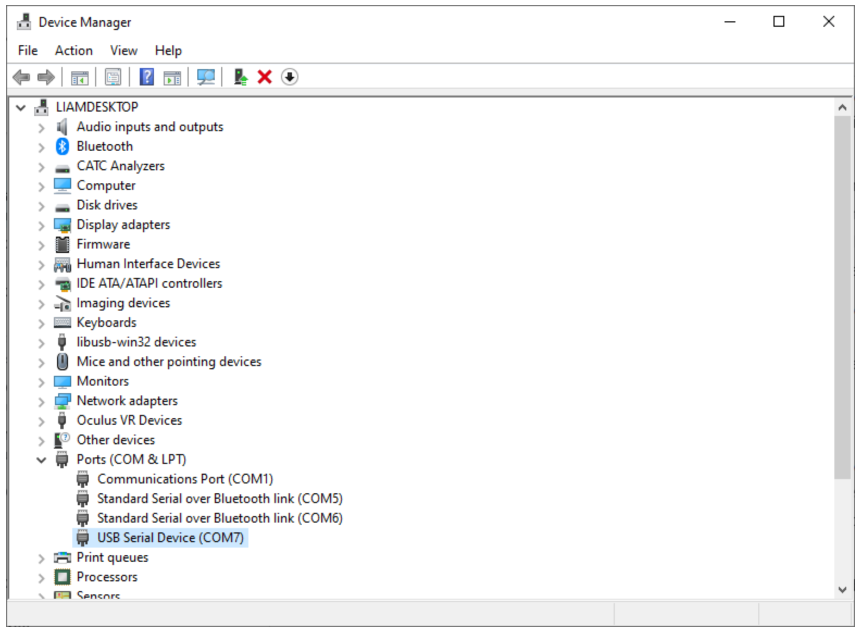
To use the UART connection on Linux, run the following command:

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

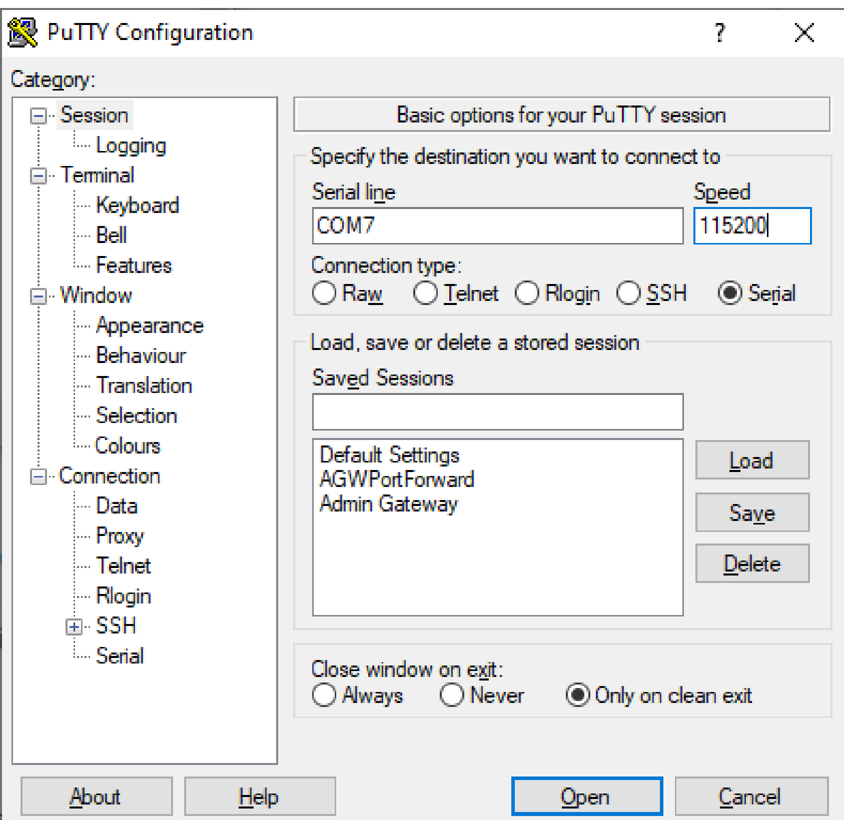
Windows

Download and install PuTTY.

Open Device Manager and locate the COM port number of the device running `debugprobe`. In this example it is `COM7`.



Open PuTTY. Select `Serial` under connection type. Then type the name of your COM port along with 115200 as the speed.



Select Open to start the serial console. You are now ready to run your application.

macOS

First, install `minicom` using Homebrew:

```
$ brew install minicom
```

Then, run the following command to use the UART connection:

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

Debug with OpenOCD

With Debug Probe, you can load binaries via the SWD port and OpenOCD.

First, build a binary. Then, run the following command to upload the binary to the Pico, replacing `blink.elf` with the name of the ELF file you just built:

```
$ sudo openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed 5000" -c
"program blink.elf verify reset exit"
```

If you are using a RP2350 based board, then use

```
$ sudo openocd -f interface/cmsis-dap.cfg -f target/rp2350.cfg -c "adapter speed 5000" -c
"program blink.elf verify reset exit"
```

Debug with SWD

You can also use `openocd` in server mode and connect a debugger that provides break points and more.

IMPORTANT

To allow debugging, build your binaries with the `Debug` build type using the `DCMAKE_BUILD_TYPE` option:

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

Note: you should use `-DPICO_BOARD=pico2` for a Raspberry Pi Pico 2.

The debug build provides more information when you run it under the debugger.

First, run an OpenOCD server:

```
$ sudo openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed 5000"
```

For a RP2350-based device use `-f target/rp2350.cfg` instead.

Then, open a second terminal window. Start your debugger, passing your binary as an argument:

```
$ gdb blink.elf
> target remote localhost:3333
> monitor reset init
> continue
```

GDB doesn't work on all platforms. Use one of the following alternatives instead of `gdb`, depending on your operating system and device:

- On Linux devices that are *not* Raspberry Pis, use `gdb-multiarch`.
- On Arm-based macOS devices, use `lldb`.

Appendix B: Picotool

It is possible to embed information into a Pico-series binary, which can be retrieved using a command line utility called `picotool`.

Getting picotool

The `picotool` utility is available in its own repository. You will need to clone and build it if you haven't ran the pico-setup script.

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

You will also need to install `libusb` if it is not already installed,

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

NOTE

If you are building `picotool` on macOS you can install `libusb` using Homebrew,

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

While if you are building on Microsoft Windows you can download and install a Windows binary of `libusb` directly from the libusb.info site.

Building picotool

Building `picotool` can be done as follows,

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

this will generate a `picotool` command-line binary in the `build/picotool` directory.

> **NOTE**
>
> If you are building on Microsoft Windows you should invoke CMake as follows,
>
> ```
> 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
> ```

Using picotool

The `picotool` binary includes a command-line help function,

```
$ picotool help
PICOTOOL:
    Tool for interacting with RP2040/RP2350 device(s) in BOOTSEL mode, or with an RP2040/RP2350
    binary

SYNOPSIS:
    picotool info [-b] [-p] [-d] [--debug] [-l] [-a] [device-selection]
    picotool info [-b] [-p] [-d] [--debug] [-l] [-a] <filename> [-t <type>]
    picotool config [-s <key> <value>] [-g <group>] [device-selection]
    picotool config [-s <key> <value>] [-g <group>] <filename> [-t <type>]
    picotool load [-p] [--family <family_id>] [-n] [-N] [-u] [-v] [-x] <filename> [-t <type>]
                [-o <offset>] [device-selection]
    picotool encrypt [--quiet] [--verbose] [--hash] [--sign] <infile> [-t <type>] [-o <offset>]
                <outfile> [-t <type>] <aes_key> [-t <type>] [<signing_key>] [-t <type>]
    picotool seal [--quiet] [--verbose] [--hash] [--sign] [--clear] <infile> [-t <type>] [-o
                <offset>] <outfile> [-t <type>] [<key>] [-t <type>] [<otp>] [-t <type>] [--major
                <major>] [--minor <minor>] [--rollback <rollback> [<rows>..]]
    picotool link [--quiet] [--verbose] <outfile> [-t <type>] <infile1> [-t <type>] <infile2>
                [-t <type>] [<infile3>] [-t <type>] [-p] <pad>
    picotool save [-p] [device-selection]
    picotool save -a [device-selection]
    picotool save -r <from> <to> [device-selection]
    picotool verify [device-selection]
    picotool reboot [-a] [-u] [-g <partition>] [-c <cpu>] [device-selection]
    picotool otp list|get|set|load|dump|permissions|white-label
    picotool partition info|create
    picotool uf2 info|convert
    picotool version [-s] [<version>]
    picotool coprodis [--quiet] [--verbose] <infile> [-t <type>] <outfile> [-t <type>]
    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
    config      Display or change program configuration settings from the target device(s) or
                file.
    load        Load the program / memory range stored in a file onto the device.
    encrypt     Encrypt the program.
    seal        Add final metadata to a binary, optionally including a hash and/or signature.
    link        Link multiple binaries into one block loop.
    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
    otp         Commands related to the RP2350 OTP (One-Time-Programmable) Memory
```

```
    partition   Commands related to RP2350 Partition Tables
    uf2         Commands related to UF2 creation and status
    version     Display picotool version
    coprodis    Post-process coprocessor instructions in dissassembly files.
    help        Show general help or help for a specific command

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

NOTE

The majority of commands require a Raspberry Pi microcontroller device in BOOTSEL mode to be connected.

IMPORTANT

If you get an error message `No accessible RP2040/RP2350 devices in BOOTSEL mode were found.` accompanied with a note similar to `Device at bus 1, address 7 appears to be a RP2040 device in BOOTSEL mode, but picotool was unable to connect` indicating that there was a Pico-series device connected then you can run picotool using `sudo`, e.g.

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

If you get this message on Windows you will need to install a driver.

Download and run Zadig, select Picotool from the dropdown box and select `libusb-win32` as the driver, and click on the "Install Driver" button.

As of version 1.1 of `picotool` it is also possible to interact with Raspberry Pi microcontrollers that are not in BOOTSEL mode, but are using USB stdio support from the SDK by using the `-f` argument of `picotool`.

Display information

So there is now *Binary Information* support in the SDK which allows for easily storing compact information that `picotool` can find (See Binary Information below). The `info` command is for reading this information.

The information can be either read from one or more connected Raspberry Pi microcontrollers in BOOTSEL mode, or from a file. This file can be an ELF, a UF2 or a BIN file.

```
$ 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

SYNOPSIS:
    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
```

For example, connect your Pico-series device to your computer as mass storage mode, by pressing and holding the BOOTSEL button before plugging it into the USB. Then open up a Terminal window and type,

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

or,

```
$ 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
```

for more information. Alternatively you can just get information on the pins used as follows,

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

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

The tool can also be used on binaries still on your local filesystem,

```
$ 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
```

Save the program

Save allows you to save a range of memory or a program or the whole of flash from the device to a BIN file or a UF2 file.

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

SYNOPSIS:
    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
```

For example,

```
$ 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
```

Binary Information

Binary information is machine-locatable and machine-readable information that is embedded in the binary at build time.

Basic information

This information is really handy when you pick up a Pico-series device and don't know what is on it!

Basic information includes

- program name
- program description
- program version string
- program build date
- program url
- program end address
- program features, this is a list built from individual strings in the binary, that can be displayed (e.g. we will have one for UART stdio and one for USB stdio) in the SDK

- build attributes, this is a similar list of strings, for things pertaining to the binary itself (e.g. Debug Build)

Pins

This is certainly handy when you have an executable called `hello_serial.elf` but you forgot what Raspberry Pi microcontroller-based board it was built for, as different boards may have different pins broken out.

Static (fixed) pin assignments can be recorded in the binary in very compact form:

```
$ 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
```

Full Information

Full information is available with the `-a` option:

```
$ picotool info -a i2c_bus_scan.elf
File i2c_bus_scan.elf:

Program Information
 name:          i2c_bus_scan
 web site:      https://github.com/raspberrypi/pico-examples/tree/HEAD/i2c/bus_scan
 features:      UART stdin / stdout
 binary start:  0x10000000
 binary end:    0x10004c74

Fixed Pin Information
 0:  UART0 TX
 1:  UART0 RX
 4:  I2C0 SDA
 5:  I2C0 SCL

Build Information
 sdk version:       2.0.0-develop
 pico_board:        pico
 build date:        Aug  1 2024
 build attributes:  Debug
```

Appendix C: Manual toolchain setup

Configure your environment via Script

If you are developing for a Pico-series device on the Raspberry Pi 5, the Raspberry Pi 4B, or the Raspberry Pi 400, most of the installation steps in this Getting Started guide can be skipped by running the `pico_setup.sh` script.

The script automates the following setup:

- Creates a directory called `pico` in the folder where you *run* the `pico_setup.sh` script
- Installs required dependencies
- Downloads the `pico-sdk`, `pico-examples`, `pico-extras`, and `pico-playground` repositories
- Defines `PICO_SDK_PATH`, `PICO_EXAMPLES_PATH`, `PICO_EXTRAS_PATH`, and `PICO_PLAYGROUND_PATH` in your `~/.bashrc`
- Builds the `blink` and `hello_world` examples in `pico-examples/build/blink` and `pico-examples/build/hello_world`
- Downloads and builds `picotool` (see Appendix B), and copy it to `/usr/local/bin`.
- Downloads and builds `debugprobe` (see Appendix A).
- Downloads and compiles OpenOCD (for debug support)
- Configures your development Raspberry Pi UART for use with Pico-series devices

TIP

This setup script requires approximately 2.5GB of disk space on your SD card, so make sure you have enough free space before running it. You can check how much free disk space you have with the `df -h` command.

First, run the following command to install `wget`:

```
$ sudo apt install wget
```

You can get this script by running the following command in a terminal:

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

Then mark the script as executable with `chmod`:

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

Run the script with the following command:

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

Finally, reboot your Raspberry Pi to load your UART configuration changes:

```
$ sudo reboot
```

Manually Configure your Environment

Get the SDK and examples

The Pico Examples repository provides a set of example applications written using the SDK. To clone these repositories, create a pico directory where you can store pico-related files. The following commands create a subdirectory named `pico` in your home directory:

```
$ mkdir ~/pico
```

Then, clone the `pico-sdk` and `pico-examples` git repositories:

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

Install the Toolchain

To build the applications in `pico-examples`, you'll need to install some extra tools. To build projects you'll need CMake, a cross-platform tool used to build the software, gcc, and the GNU Embedded Toolchain for Arm. Run the following command to install these dependencies:

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

Ubuntu and Debian users might additionally need to do:

```
 apt install g++ libstdc++-arm-none-eabi-newlib
```

Enable UART serial communications

To enable UART serial communications on your development device. To do so on a Raspberry Pi running Raspberry Pi OS, run `raspi-config`:

```
$ sudo raspi-config
```

1. Navigate to `Interfacing Options` > `Serial`.
2. When asked "Would you like a login shell to be accessible over serial?", answer "No".
3. When asked "Would you like the serial port hardware to be enabled?", answer "Yes".

You should see something like Figure 11:

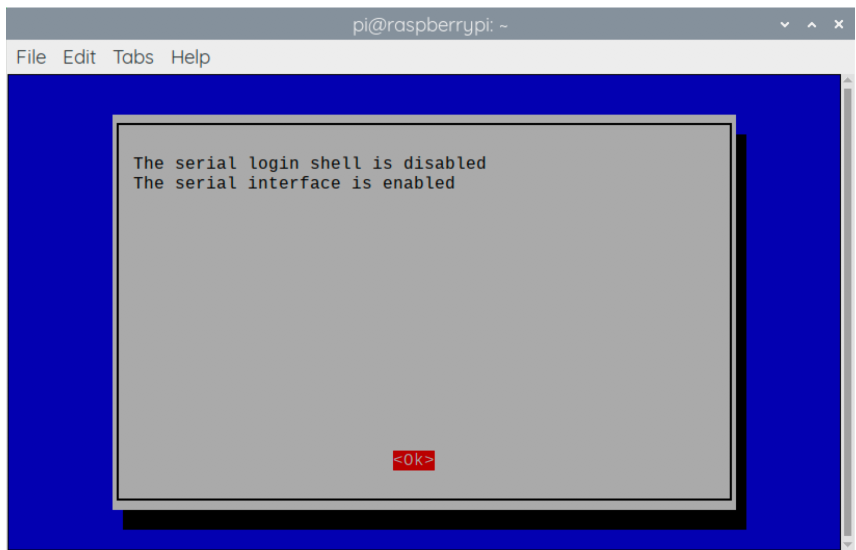


Figure 11. Enabling a serial UART using `raspi-config` on the Raspberry Pi.

Exit `raspi-config` with **Esc**. Choose "Yes" and reboot your Raspberry Pi to enable the serial port.

IMPORTANT

Raspberry Pi 5 makes the UART on the 3-pin debug header the default for `serial0`. To use use GPIO pins 15 and 14 instead, append `dtparam=uart0_console` to `/boot/firmware/config.txt`.

Update the SDK

When a new version of the SDK is released, you must update your copy of the SDK. To update, navigate into `pico-sdk` and run the following command:

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

NOTE

To be informed of new releases, set up a custom watch on the `pico-sdk` GitHub repository. Navigate to https://github.com/raspberrypi/pico-sdk and select Watch → Custom → Releases. You will receive an email notification when a new SDK release occurs.

Use the CLI to Blink an LED in C

When you're writing software for hardware, turning an LED on, off, and then on again, is typically the first program that gets run in a new programming environment. Learning how to blink an LED gets you half way to anywhere.

So let's blink the LED on a Pico-series device.

The following example blinks the LED connected to pin 25 of a Raspberry Pi Pico or Pico 2:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/blink_simple/blink_simple.c Lines 31 - 39

```
31 int main() {
32     pico_led_init();
33     while (true) {
34         pico_set_led(true);
35         sleep_ms(LED_DELAY_MS);
36         pico_set_led(false);
37         sleep_ms(LED_DELAY_MS);
38     }
39 }
```

The actual code for the "blink" example is slightly complicated as it also supports blinking the LED connected to the Infineon 43439 wireless chip on the Pico W. The full code can be found here sdkexampleref::blink/blink.c

Building "Blink"

From the `pico` directory we created earlier, navigate into `pico-examples` and create a build directory:

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

Then, set the `PICO_SDK_PATH`, assuming you cloned the `pico-sdk` and `pico-examples` repositories into the same directory:

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

TIP

Throughout this book we use the relative path `../../pico-sdk` to the SDK repository for `PICO_SDK_PATH`. Depending on the location of your repository, you could replace this with an absolute path.

Build "Blink"

Prepare your `cmake` build directory by running the following command:

```
$ 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
```

> **IMPORTANT**
>
> The SDK builds binaries for the Raspberry Pi Pico 2 by default. To build a binary for a different board, pass the `-DPICO_BOARD=<board>` option to CMake, replacing the `<board>` placeholder with the name of the board you'd like to target. To build a binary for Pico 2, pass `-DPICO_BOARD=pico2`. To build a binary for Pico W, pass `-DPICO_BOARD=pico_w`. You can specify a Wi-Fi network and password that your Pico W examples should connect to, by passing `-DWIFI_SSID="Your Network" -DWIFI_PASSWORD="Your Password"` too.

You can now type `make` to build all example applications. However, for this example we only need to build `blink`. To build a specific subtree of examples, navigate into the corresponding subtree before running `make`. In this case, we can build only the `blink` task by first navigating into the `blink` directory, then running `make`:

```
$ 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
```

> **TIP**
>
> Invoking `make` with `-j4` speeds the build up by running four jobs in parallel. A Raspberry Pi 5 has four cores, so four jobs spreads the build evenly across the entire SoC.

Amongst other targets, this builds:

`blink.elf`
: used by the debugger

`blink.uf2`
: the file we'll copy onto the USB Mass Storage Device that represents your Raspberry Pi microcontroller

Load and run "Blink"

To load software onto a Raspberry Pi microcontroller-based board, mount it as a USB Mass Storage Device and copy a `uf2` file onto the board to program the flash.

Hold down the `BOOTSEL` button (Figure 12) while plugging in your device using a micro-USB cable to force it into USB Mass Storage Mode.

The device will reboot, unmount itself as a Mass Storage Device, and run the flashed code, see Figure 12.

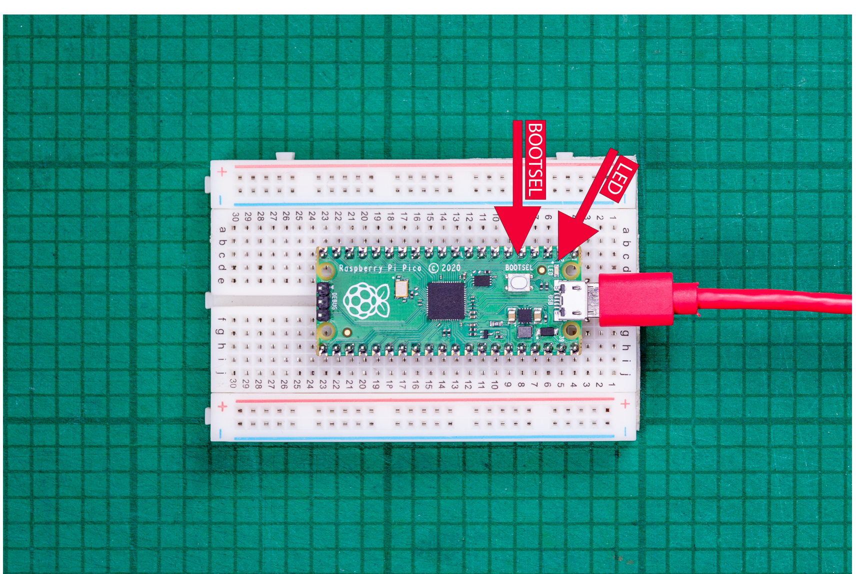


Figure 12. Blinking the on-board LED on the Raspberry Pi Pico 2. Arrows point to the on-board LED, and the `BOOTSEL` button.

Using the command line

TIP

You can use `picotool` to load a UF2 binary onto your Pico-series device, see Appendix B.

Depending on the platform you use to compile binaries, you may have to mount the mass storage device manually:

```
$ 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
```

If you can see files in `/mnt/pico`, the USB Mass Storage Device has mounted correctly:

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

Copy your `blink.uf2` onto the device:

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

The microcontroller automatically disconnects as a USB Mass Storage Device and runs your code, but just to be safe, you should unmount manually as well:

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

> **NOTE**
>
> Removing power from the board does not remove the code. When you restore power to the board, the flashed code will run again.

Aside: Other Boards

If you are not following these instructions on a Raspberry Pi Pico-series device, you may not have a `BOOTSEL` button (as labelled in Figure 12). Your board may have some other way of loading code, which the board supplier should have documented:

- Most boards expose the SWD interface ([debug_probe_section]) which can reset the board and load code without any button presses
- There may be some other way of pulling down the flash CS pin (which is how the `BOOTSEL` button works on Pico-series devices), such as shorting together a pair of jumper pins
- Some boards have a reset button, but no `BOOTSEL`; they might detect a double-press of the reset button to enter the bootloader

In all cases you should consult the documentation for the specific board you are using, which should describe the best way to load firmware onto that board.

Manually Create your own Project

Go ahead and create a directory to house your test project sitting alongside the `pico-sdk` directory,

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

and then create a `test.c` file in the directory,

```
 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;①
 7
 8 int main() {
 9
10     bi_decl(bi_program_description("This is a test binary."));②
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 The onboard LED is connected to GP25 on Pico and Pico 2, if you're building for Pico W the LED is connected to `CYW43_WL_GPIO_LED_PIN`. For more information see the Pico W blink example in the Pico Examples Github repository.

② These lines will add strings to the binary visible using `picotool`, see Appendix B.

along with a `CMakeLists.txt` file,

```
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. Enables serial output via USB.
2. Enables serial output via UART.

Then copy the `pico_sdk_import.cmake` file from the `external` folder in your `pico-sdk` installation to your test project folder.

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

You should now have something that looks like this,

```
$ 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
```

and can build it as we did before with our "Hello World" example.

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

IMPORTANT

The SDK builds binaries for the Raspberry Pi Pico by default. To build a binary for a different board, pass the `-DPICO_BOARD=<board>` option to CMake, replacing the `<board>` placeholder with the name of the board you'd like to target. To build a binary for Pico 2, pass `-DPICO_BOARD=pico2`. To build a binary for Pico W, pass `-DPICO_BOARD=pico_w`. To specify a Wi-Fi network and password that your Pico W should connect to, pass `-DWIFI_SSID="Your Network" -DWIFI_PASSWORD="Your Password"`.

The make process will produce a number of different files. The important ones are shown in the following table.

File extension	Description
.bin	Raw binary dump of the program code and data
.elf	The full program output, possibly including debug information
.uf2	The program code and data in a UF2 form that you can drag-and-drop on to the device when it is mounted as a USB drive
.dis	A disassembly of the compiled binary
.hex	Hexdump of the compiled binary
.map	A map file to accompany the .elf file describing where the linker has arranged segments in memory

NOTE

UF2 (USB Flashing Format) is a Microsoft-developed file format used for flashing Raspberry Pi microcontrollers over USB. For more information, see the Microsoft UF2 Specification Repo.

NOTE

To build a binary to run in SRAM, rather than Flash memory you can either setup your `cmake` build with `-DPICO_NO_FLASH=1` or you can add `pico_set_binary_type(TARGET_NAME no_flash)` to control it on a per binary basis in your `CMakeLists.txt` file. You can download the RAM binary to Raspberry Pi microcontrollers via UF2. For example, if there is no flash chip on your board, you can download a binary that runs on the on-chip RAM using UF2 as it simply specifies the addresses of where data goes. Note you can only download in to RAM or FLASH, not both.

Debugging your project

Debugging your own project from the command line follows the same processes as we used for the "Hello World" example back in Debug with SWD.

Need more detail?

There should be enough here to show you *how* to get started, but you may find yourself wondering *why* some of these files and incantations are needed. The **Raspberry Pi Pico-series C/C++ SDK** book dives

deeper into how your project is actually built, and how the lines in our `CMakeLists.txt` files here relate to the structure of the SDK, if you find yourself wanting to know more at some future point.

Appendix D: Use other Integrated Development Environments

The recommended Integrated Development Environment (IDE) is Visual Studio Code. However other environments can be used with Raspberry Pi microcontrollers and Raspberry Pi Pico-series.

Use Eclipse

Eclipse is a multiplatform Integrated Development environment (IDE) available for Linux, macOS, and Windows. The latest version works well on the Raspberry Pi 4, 400, and 5 (4GB and up) running a 64-bit OS. The following instructions describe how to set up Eclipse on a Linux device for to develop on Pico-series devices. Instructions for other systems will be broadly similar, although the details of connecting to Pico-series devices vary.

Setting up Eclipse for Pico on a Linux machine

Prerequisites:

- Device running a recent version of Linux with at least 4GB of RAM
- 64-bit operating system.
- CMake 3.11 or newer

If using a Raspberry Pi, you should enable the standard UART by adding the following to config.txt

`enable_uart=1`

You should also install OpenOCD and the SWD debug system. See [debug_probe_section] for instructions on how to do this.

Installing Eclipse and Eclipse plugins

Install the latest version of Eclipse IDE for Embedded C/C++ Developers using the standard instructions. If you are running on an ARM platform, you will need to install an AArch64 (64-bit ARM) version of Eclipse. All versions can be found on the Eclipse website. https://www.eclipse.org/downloads/packages

Download the correct file for your system, and extract it. You can then run it by going to the place where it was extracted and running the 'eclipse' executable.

```
$ ./eclipse
```

The Embedded CDT version of Eclipse includes the C/C++ development kit and the Embedded development kit, so has everything you need to develop for Pico-series devices.

Using pico-examples

The standard build system for the Pico environment is CMake. However Eclipse does not use CMake as it has its own build system, so we need to convert the pico-examples CMake build to an Eclipse project.

1. At the same level as the `pico-examples` folder, create a new folder, for example `pico-examples-eclipse`
2. Change directory to that folder
3. Set the path to the `PICO_SDK_PATH`

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

4. Run the following command:

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

> **IMPORTANT**
>
> The SDK builds binaries for the Raspberry Pi Pico by default. To build a binary for a different board, pass the `-DPICO_BOARD=<board>` option to CMake, replacing the `<board>` placeholder with the name of the board you'd like to target. To build a binary for Pico 2, pass `-DPICO_BOARD=pico2`. To build a binary for Pico W, pass `-DPICO_BOARD=pico_w`. To specify a Wi-Fi network and password that your Pico W should connect to, pass `-DWIFI_SSID="Your Network" -DWIFI_PASSWORD="Your Password"`.

This will create the Eclipse project files in our `pico-examples-eclipse` folder, using the source from the original CMake tree.

You can now load your new project files into Eclipse using the `Open project From File System` option in the File menu.

Building

Right click on the project in the project explorer, and select `Build`. This will build all the examples.

OpenOCD

This example uses the OpenOCD system to communicate with a Raspberry Pi microcontroller. You will need to have provided the 2-wire debug connections from the host device to the microcontroller prior to running the code. On a Raspberry Pi, this can be done via GPIO connections, but on a laptop or desktop device, you need to use extra hardware for this connection. One way is to use the Debug Probe.

Once OpenOCD is installed and the correct connection made, Eclipse needs to be set up to talk to OpenOCD when programs are run. OpenOCD provides a GDB interface to Eclipse, and it is that interface that is used when debugging.

To set up the OpenOCD system, select `Preferences` from the `Window` menu.

Click on `MCU` arrow to expand the options and click on `Global OpenOCD path`.

For the executable, type in "openocd". For the folder, select the location in the file system where you have cloned the Pico OpenOCD fork from github.

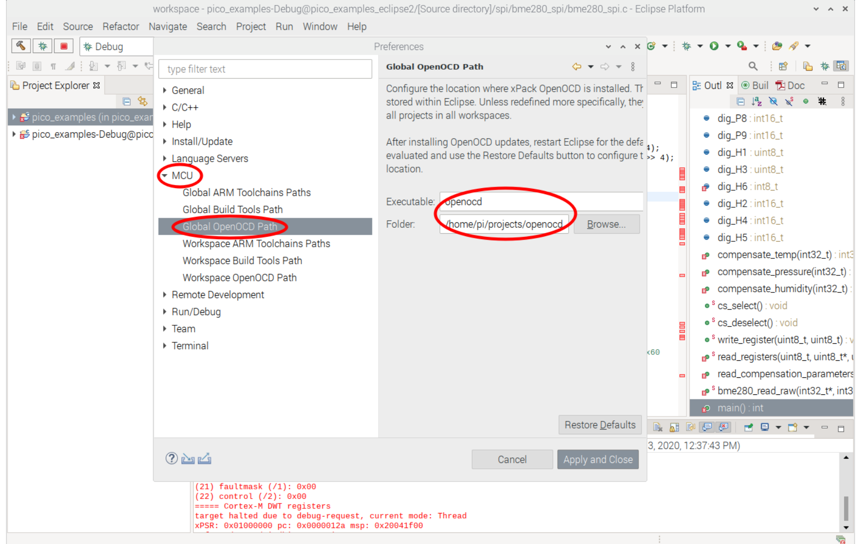


Figure 13. Setting the OCD executable name and path in Eclipse.

Creating a Run configuration

In order to run or debug code in Eclipse you need to set up a Run Configuration. This sets up all the information needed to identify the code to run, any parameters, the debugger, source paths and SVD information.

From the Eclipse Run menu, select `Run Configurations`. To create a debugger configuration, select `GDB OpenOCD Debugging` option, then select the `New Configuration` button.

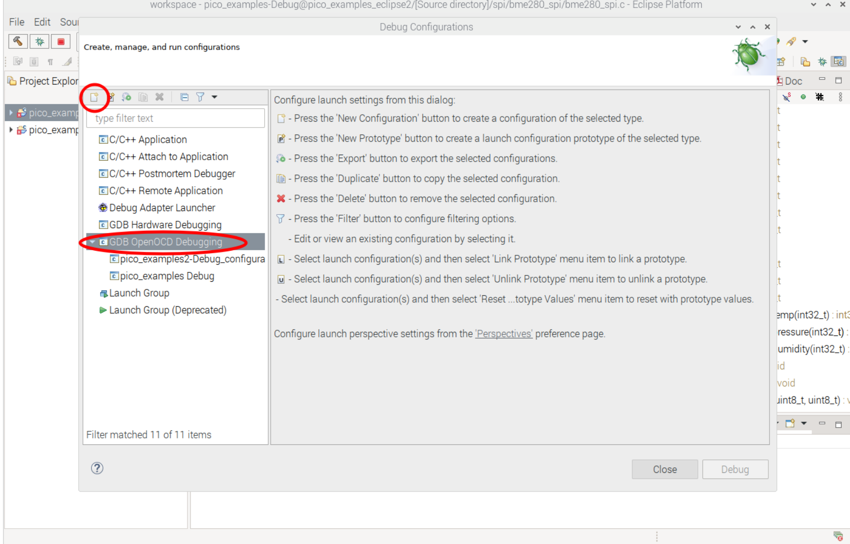


Figure 14. Creating a new Run/Debug configuration in Eclipse.

Setting up the application to run

Because the pico-examples build creates lots of different application executables, you need to select which specific one is to be run or debugged.

On the `Main` tab of the Run configuration page, use the `Browse` option to select the C/C++ applications you wish to run.

The Eclipse build will have created the executables in sub folders of the Eclipse project folder. In our example case this is

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

So for example, if we running the LED blink example, this can be found at:

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

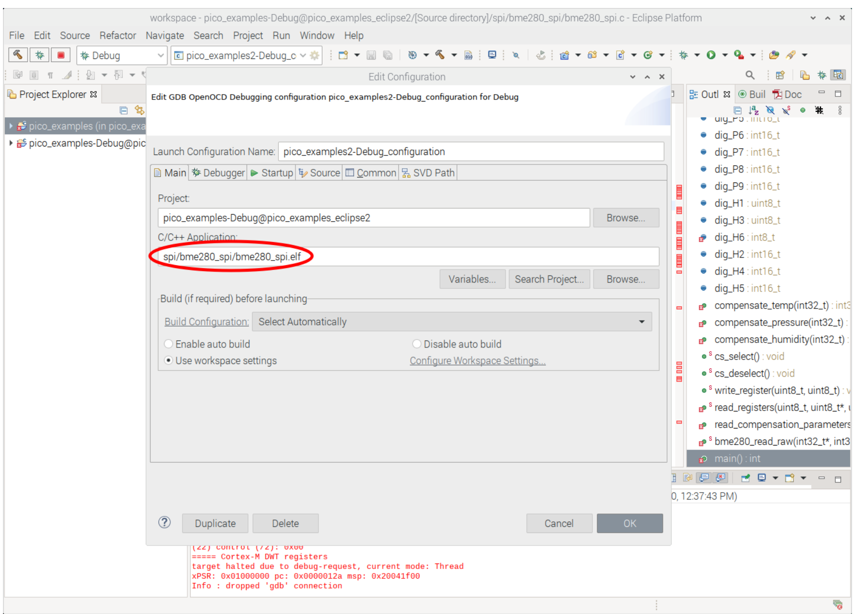


Figure 15. Setting the executable to debug in Eclipse.

Setting up the debugger

Let’s set up OpenOCD to talk to the Raspberry Pi microcontroller.

Set `openocd` in the boxes labelled **Executable** and **Actual Executable**. We also need to set up OpenOCD to use the Pico specific configuration, so in the Config options sections add the following. You will need to change the path to point to the location where the Pico version of OpenOCD is installed.

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

Replace `rp2040.cfg` with `rp2350.cfg` if you are using a RP2350-based device.

All other OpenOCD settings should be set to the default values.

The actual debugger used is GDB. This talks to the OpenOCD debugger for the actual communications with the Raspberry Pi microcontroller, but provides a standard interface to the IDE.

The particular version of GDB used is `gdb-multiarch`, so enter this in the fields labelled **Executable Name** and **Actual Executable**.

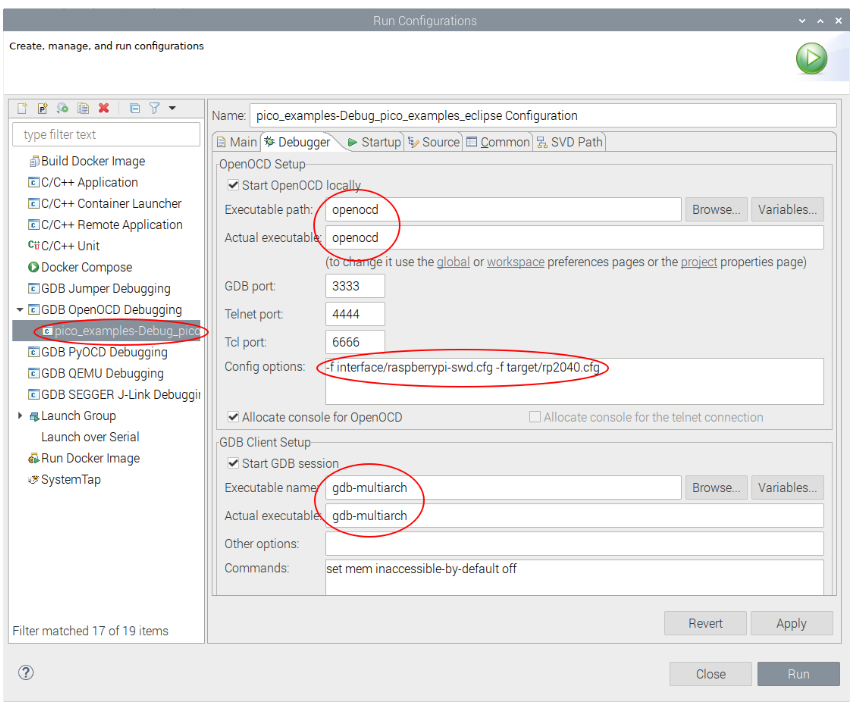


Figure 16. Setting up the Debugger and OpenOCD in Eclipse.

Setting up the SVD plugin

SVD provides a mechanism to view and set peripheral registers on the Pico board. An SVD file provides register locations and descriptions, and the SVD plugin for Eclipse integrates that functionality in to the Eclipse IDE. The SVD plugin comes as part of the Embedded development plugins.

Select the SVD path tab on the Launch configuration, and enter the location on the file system where the SVD file is located. This is usually found in the pico-sdk source tree.

E.g.

`…/pico-sdk/src/rp2040/hardware_regs/RP2040.svd` or `…/pico-sdk/src/rp2350/hardware_regs/RP2350.svd`

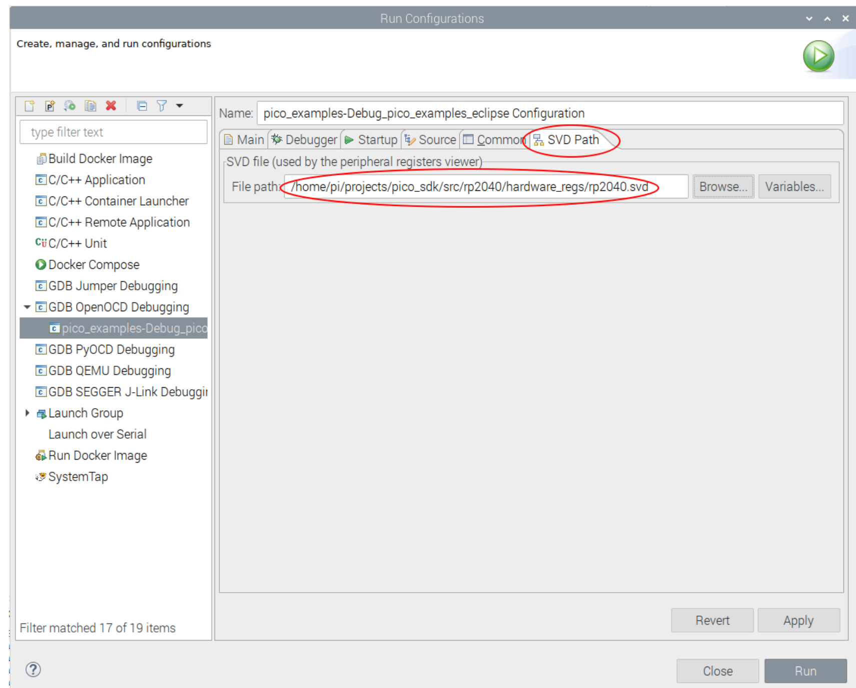


Figure 17. Setting the SVD path in Eclipse.

Running the Debugger

Once the Run configuration is complete and saved, you can launch immediately using the `Run` button at the bottom right of the dialog, or simply `Apply` the changes and `Close` the dialog. You can then run the application using the `Run` Menu `Debug` option.

This will set Eclipse in to debug perspective, which will display a multitude of different debug and source code windows, along with the very useful Peripherals view which uses the SVD data to provide access to peripheral registers. From this point on this is a standard Eclipse debugging session.

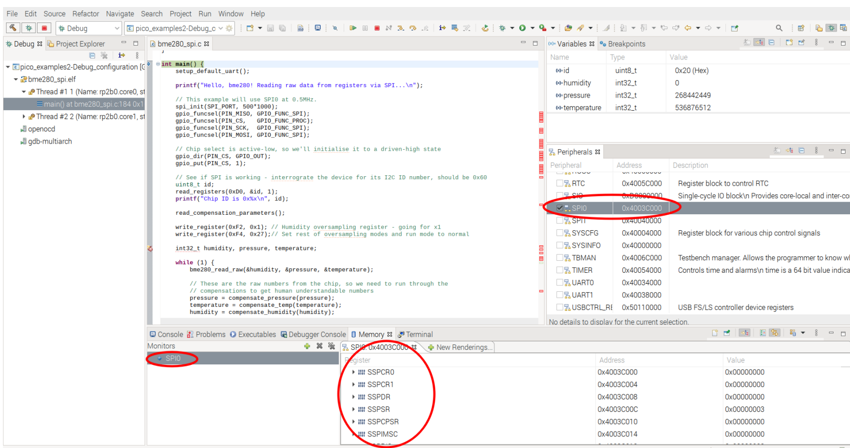


Figure 18. The Eclipse debugger running, showing some of the debugging window available.

Use CLion

CLion is a multiplatform Integrated Development environment (IDE) from JetBrains, available for Linux, Windows and Mac. This is a commercial IDE often the choice of professional developers (or those who love JetBrains IDEs) although there are free or reduce price licenses available. It *will* run on a Raspberry Pi, however the performance is not ideal, so it is expected you would be using CLion on your desktop or laptop.

Whilst setting up projects, development and building are a breeze, setting up debug is still not very mainstream at the moment, so be warned.

Setting up CLion

If you are planning to use CLion we assume you either have it installed or can install it from https://www.jetbrains.com/clion/

Setting up a project

Here we are using pico-examples as the example project.

To open the pico-examples project, select `Open…` from the `File` menu, and then navigate to and select the `pico-examples` directory you checked out, and press OK.

Once open you'll see something like Figure 19.

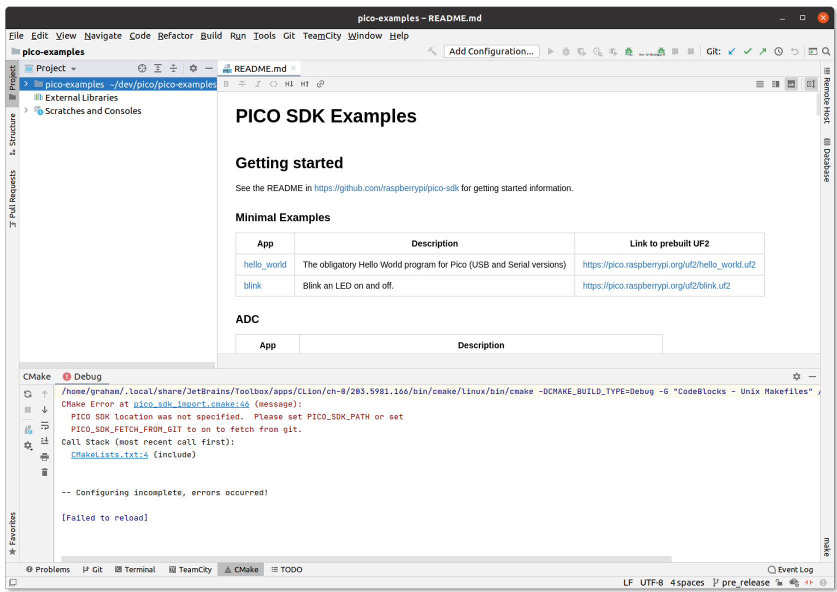


Figure 19. A newly opened CLion pico-examples project.

Notice at the bottom that CLion attempted to load the CMake project, but there was an error; namely that we hadn't specified `PICO_SDK_PATH`

Configuring CMake Profiles

Select `Settings…` from the `File` menu, and then navigate to and select 'CMake' under `Build, Execution, Deployment`.

You can set the environment variable `PICO_SDK_PATH` under `Environment:` as in Figure 20, or you can set it as `-DPICO_SDK_PATH=xxx` under `CMake options:`. These are just like the environment variables or command line args when calling `cmake` from the command line, so this is where you'd specify CMake settings such as `PICO_BOARD`, `PICO_TOOLCHAIN_PATH` etc.

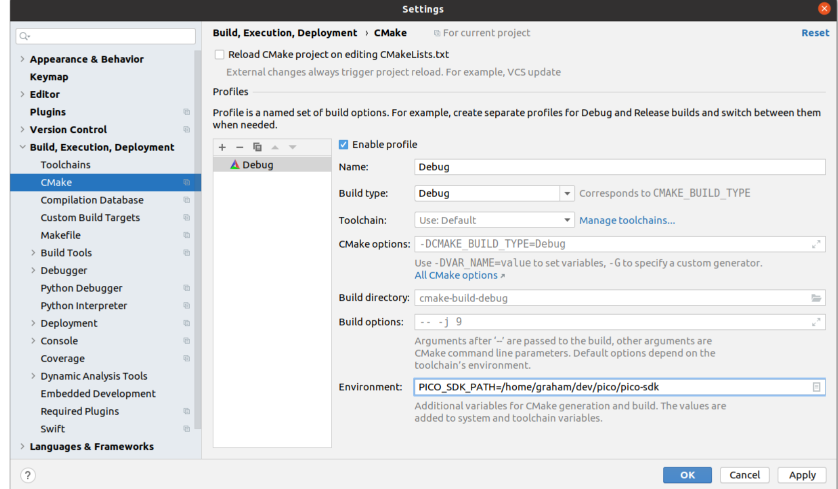


Figure 20. Configuring a CMake profile in CLion.

> **IMPORTANT**
>
> The SDK builds binaries for the Raspberry Pi Pico by default. To build a binary for a different board, pass the `-DPICO_BOARD=<board>` option to CMake, replacing the `<board>` placeholder with the name of the board you'd like to target. To build a binary for Pico 2, pass `-DPICO_BOARD=pico2`. To build a binary for Pico W, pass `-DPICO_BOARD=pico_w`. To specify a Wi-Fi network and password that your Pico W should connect to, pass `-DWIFI_SSID="Your Network" -DWIFI_PASSWORD="Your Password"`.

You can have as many CMake profiles as you like with different settings. You probably want to add a `Release` build by hitting the `+` button, and then filling in the PICO_SDK_PATH again, or by hitting the copy button two to the right, and fixing the name and settings (see Figure 21)

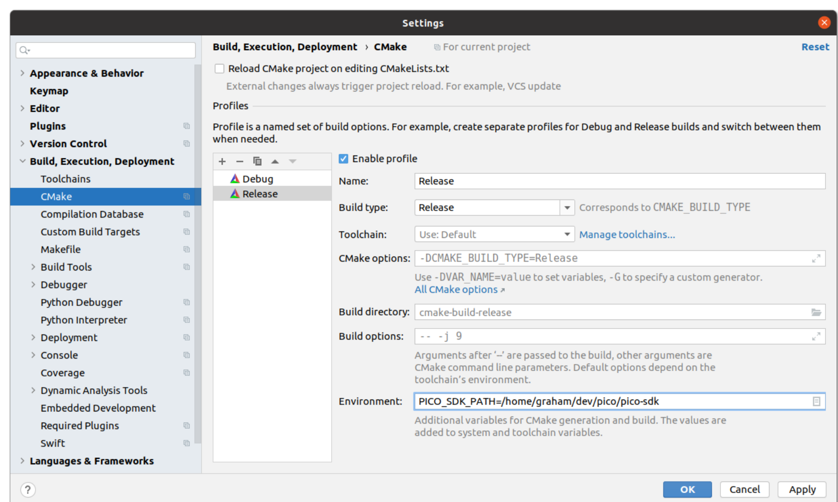


Figure 21. Configuring a second CMake Profile in CLion.

After pressing OK, you'll see something like Figure 22. Note that there are two tabs for the two profiles (`Debug` and `Release`) at the bottom of the window. In this case `Release` is selected, and you can see that the CMake setup was successful.

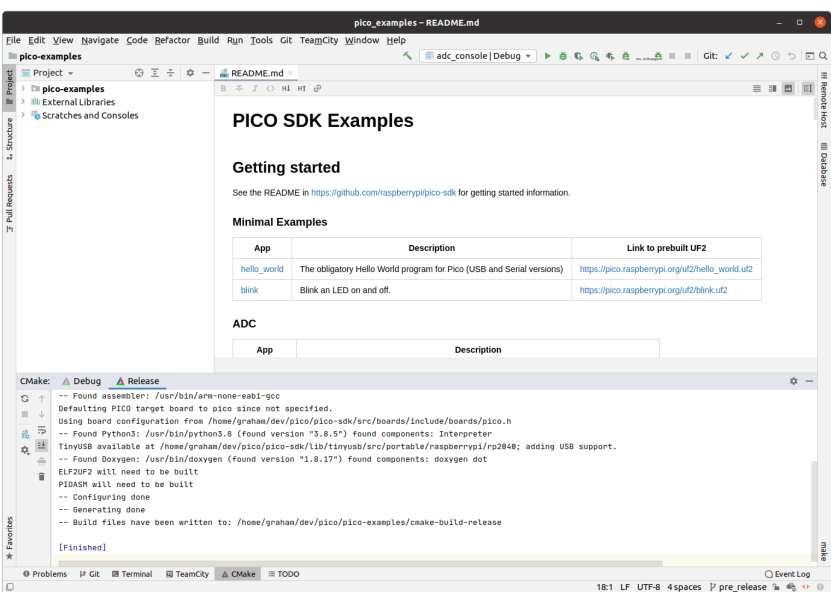


Figure 22. Configuring a second CMake profile in CLion.

Running a build

Now we can choose to build one or more targets. For example you can navigate to the drop down selector in the middle of the toolbar, and select or starting typing `hello_usb`; then press the tool icon to its left to build (see Figure 23).

Alternatively you can do a full build of all targets or other types of build from the `Build` menu.

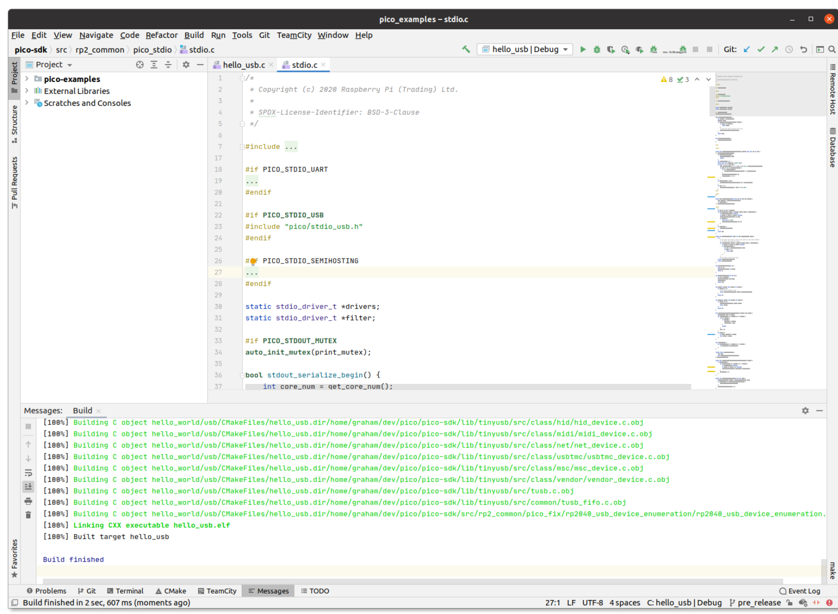


Figure 23. `hello_usb` successfully built.

Note that the drop down selector lets you choose both the target you want to build and a CMake profile to use (in this case one of `Debug` or `Release`)

Another thing you'll notice Figure 23 shows is that in the bottom status bar, you can see `hello_usb` and `Debug` again. These are showing you the target and CMake profile being used to control syntax highlighting etc. in the editor (This was auto selected when you chose `hello_usb` before). You can visually see in the `stdio.c` file that has been opened by the user, that `PICO_STDIO_USB` is set, but `PICO_STDIO_UART` is not (which are part of the configuration of hello_usb). Build time per binary configuration of libraries is heavily used within the SDK, so this is a very nice feature.

Build Artifacts

The build artifacts are located under `cmake-build-<profile>` under the project root (see Figure 24). In this case this is the `cmake-build-debug` directory.

The UF2 file can be copied onto a Raspberry Pi microcontroller in BOOTSEL mode, or the ELF can be used for debugging.

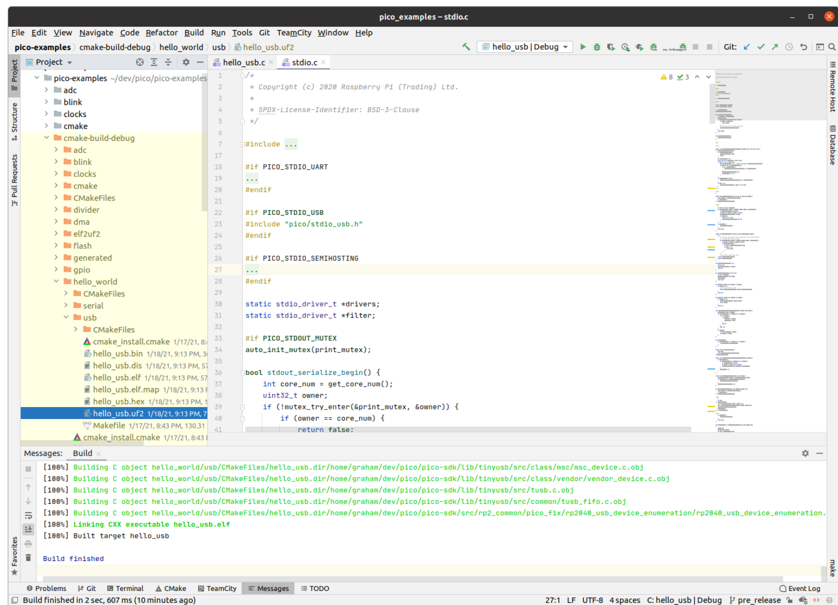


Figure 24. Locating the `hello_usb` build artifacts

Other Environments

There are too development environments available to describe all of them here. You can use many of them with the SDK. In general, IDEs require the following features to support Pico-series devices:

- CMake integration
- GDB support with remote options
- SVD. Not essential but makes reading peripheral status much easier
- Optional Arm embedded development plugin. These types of plugin often make support much easier.

Appendix H: Documentation Release History

August 9 2024

- Rename "Load a project" chapter to "Load and debug a project"
- Fix incorrect image label
- Reword *VSCode* as *VS Code*

August 8 2024

Initial release.

Raspberry Pi

Raspberry Pi is a trademark of Raspberry Pi Ltd