Getting Started with Raspberry Pi Pico-series

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

Colophon

Release	24
Build date	03/07/2026
Build version	d4e0f1799616

Legal disclaimer notice

Table of Contents

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 . . . 6
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 . . . 9
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
    A.1. Debug Probe . . . 15
        A.1.1. Debug Probe wiring . . . 15
    A.2. Debug with a second Pico or Pico 2 . . . 16
        A.2.1. Install `debugprobe` . . . 17
    A.3. `debugprobe` wiring . . . 17
    A.4. Debug Probe interfaces . . . 18
    A.5. Use the UART . . . 18
        A.5.1. Linux . . . 18
        A.5.2. Windows . . . 18
        A.5.3. macOS . . . 19
    A.6. Debug with OpenOCD . . . 20
        A.6.1. Getting OpenOCD . . . 20
        A.6.2. Debug with SWD . . . 20
        A.6.3. Rescue Debug Port . . . 21
Appendix B. Picotool . . . 22
    B.1. Getting picotool . . . 22
    B.2. Using picotool . . . 22
        B.2.1. Display information . . . 23
        B.2.2. Save the program . . . 25
        B.2.3. Binary Information . . . 26
        B.2.4. Basic information . . . 26
        B.2.5. Pins . . . 27
        B.2.6. Full Information . . . 27
Appendix C. Command line toolchain setup . . . 28
    C.1. Configure your environment via script . . . 28
        C.1.1. Manually Configure your Environment . . . 28
        C.1.2. Update the Pico SDK . . . 28
    C.2. Manually Create your own Project . . . 29
        C.2.1. Debugging your project . . . 31
Appendix D. Use other Integrated Development Environments . . . 32
    D.1. Use CLion . . . 32
        D.1.1. Setting up CLion . . . 32
    D.2. Other Environments . . . 35
Appendix E. Documentation Release History . . . 36
    E.1. 03 July 2026 . . . 36
    E.2. 15 October 2024 . . . 36

E.3. 06 September 2024 . . . 36
E.4. 09 August 2024 . . . 36
E.5. 08 August 2024 . . . 36
E.6. 02 May 2024 . . . 36
E.7. 02 February 2024 . . . 37
E.8. 14 June 2023 . . . 37
E.9. 03 March 2023 . . . 37
E.10. 01 December 2022 . . . 37
E.11. 30 June 2022 . . . 37
E.12. 17 June 2022 . . . 37
E.13. 04 November 2021 . . . 37
E.14. 03 November 2021 . . . 37
E.15. 30 September 2021 . . . 37
E.16. 23 June 2021 . . . 38
E.17. 07 June 2021 . . . 38
E.18. 13 April 2021 . . . 38
E.19. 07 April 2021 . . . 38
E.20. 05 March 2021 . . . 38
E.21. 23 February 2021 . . . 38
E.22. 01 February 2021 . . . 38
E.23. 26 January 2021 . . . 38
E.24. 21 January 2021 . . . 38

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 wish to develop on the command line instead (for example on Raspberry Pi OS Lite), see Appendix C. Command line toolchain setup.

For more information on the official MicroPython port, see:

- the documentation for the Raspberry Pi Pico-series Python SDK
- the Raspberry Pi Press book Get Started with MicroPython on Raspberry Pi Pico - 2nd Edition

For more information on the C/C++ SDK, see the documentation for the Raspberry Pi Pico-series C/C++ SDK.

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:

Code
```
$ 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:

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

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, some distributions may require additional dependencies. The extension requires the following:

- Python 3.9 or later
- Git
- Tar
- [Optional] gdb-multiarch for debugging (x86_64 only)
- [Optional] udev rules installed to use OpenOCD without `sudo`, for debugging
- [Optional] udev rules installed to use picotool without `sudo`, for loading
- For Ubuntu 22.04, install `libftdi1-2` and `libhidapi-hidraw0` packages to use OpenOCD

You can install these with:

Code
```
$ sudo apt install python3 git tar
```

3.1.3. macOS

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

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

This installs the following dependencies:

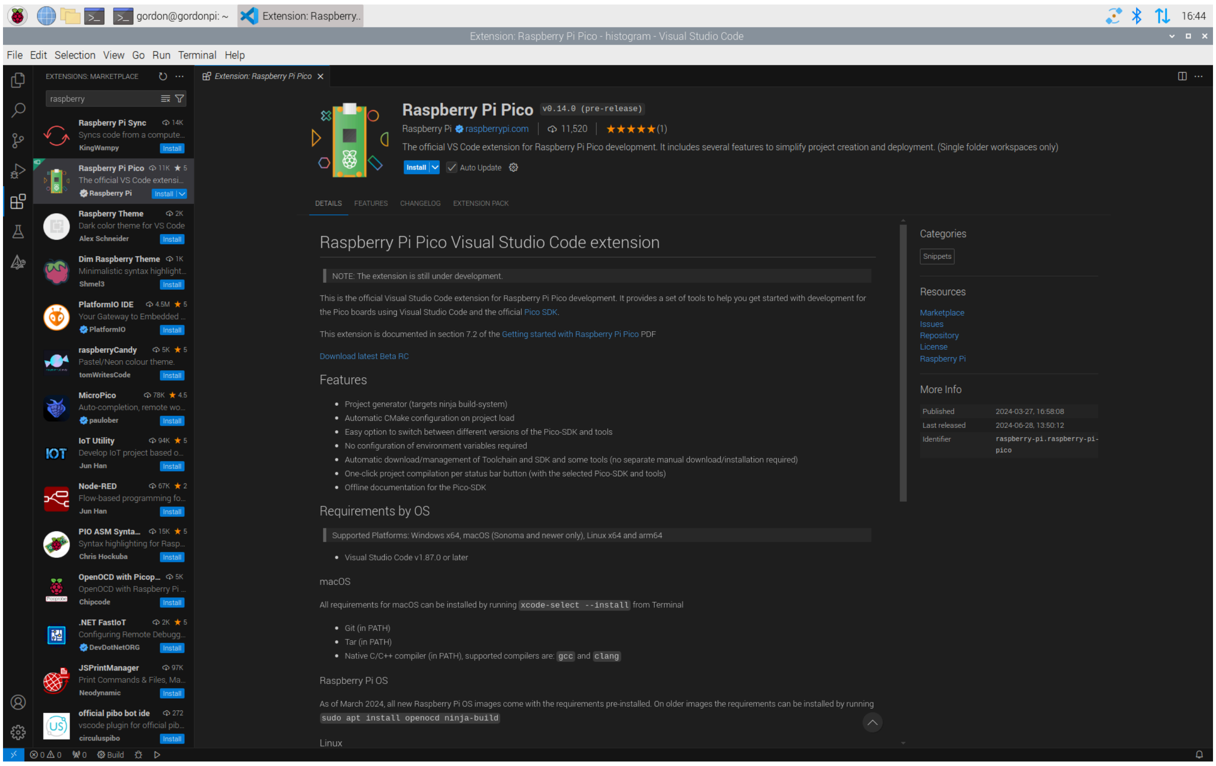
- Git
- Tar

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”.

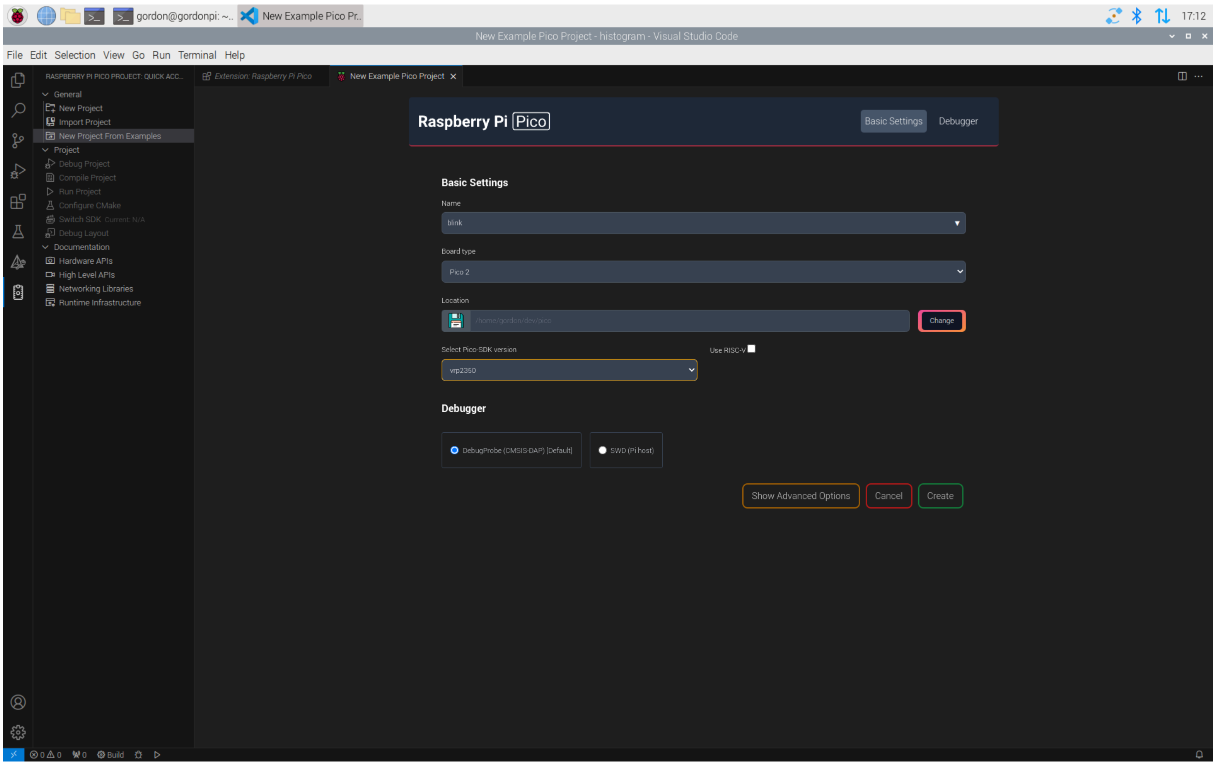
4. Load and debug a project

The VS Code extension can create projects based on the examples provided by https://github.com/raspberrypi/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:

Code
```
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:

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):

   Code
   ```
   #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:

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

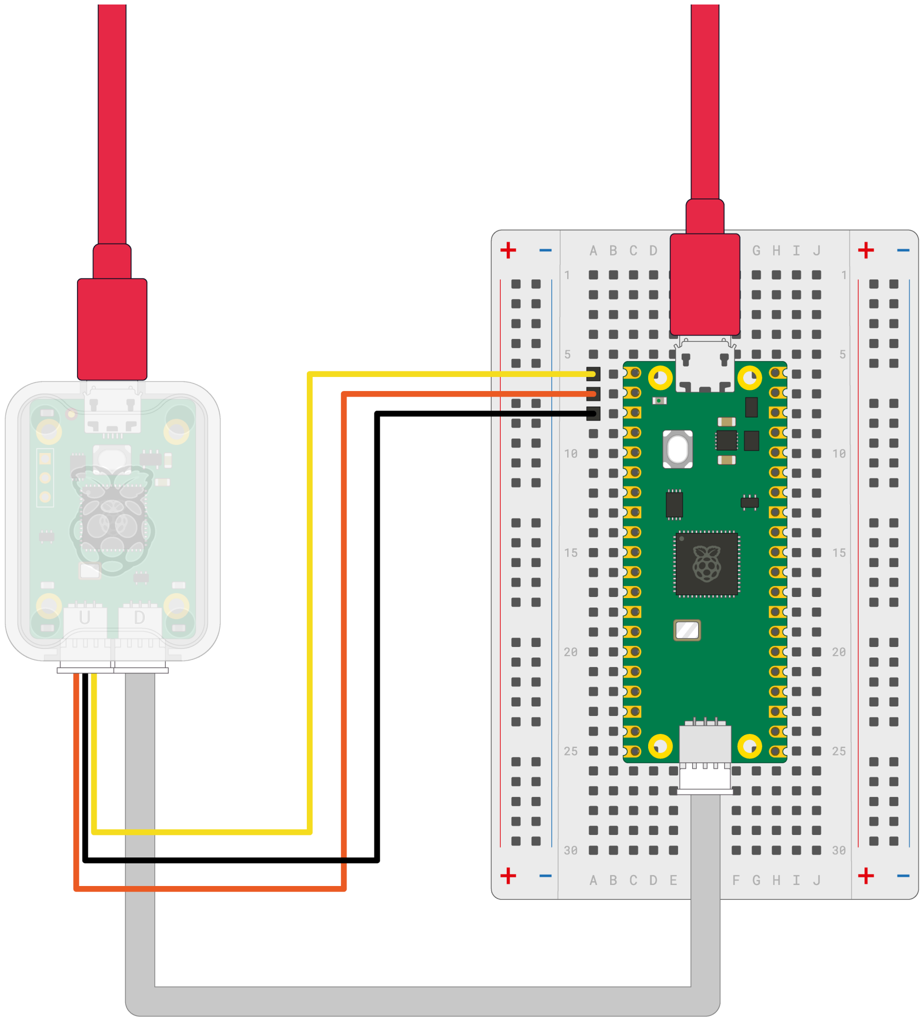
Your blink code is now running. If you look at your device, the LED should flash faster, five times every second.

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 Appendix A.2. 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.

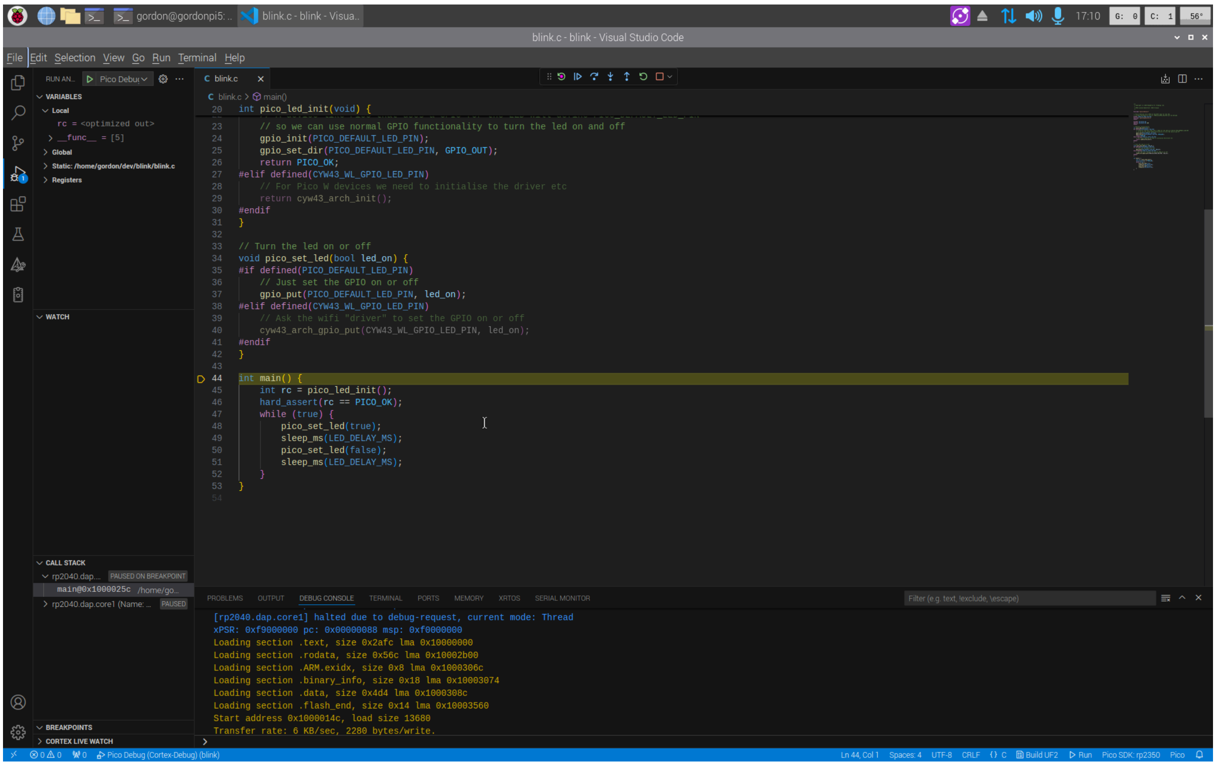
To start the debugger:

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`.

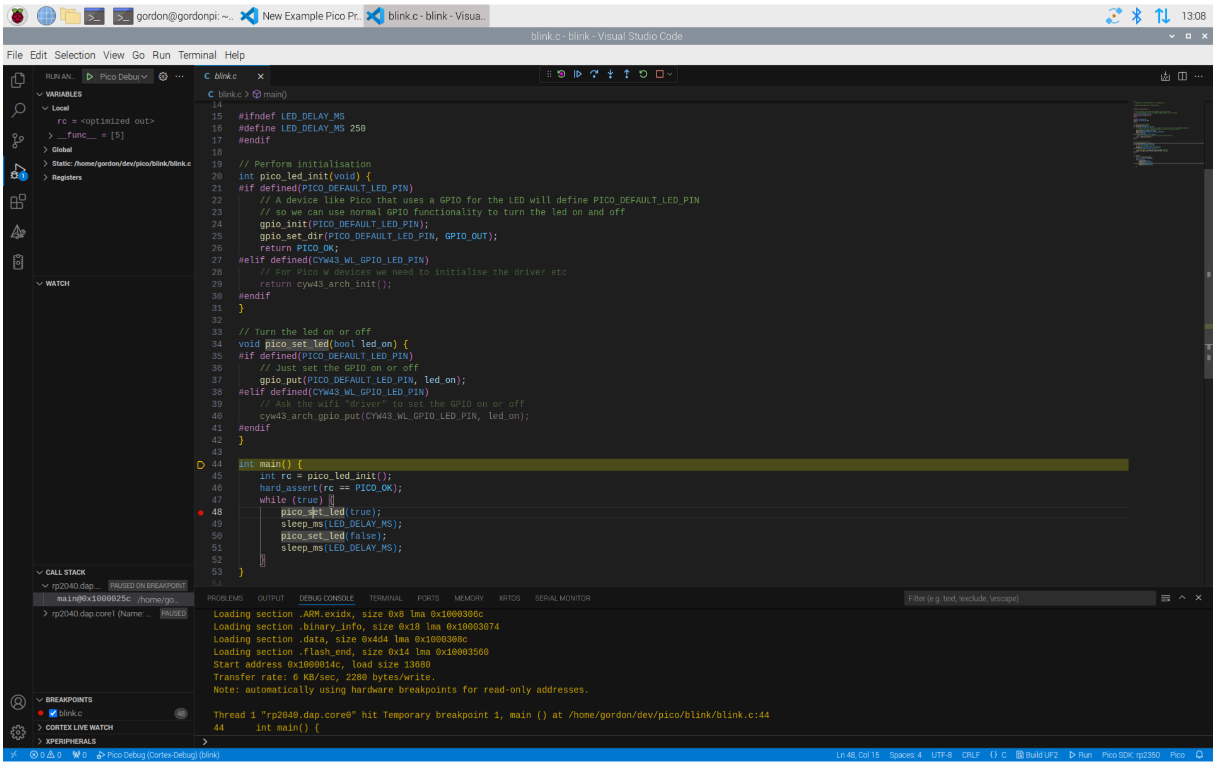
Press **Step-over** (**F10**) once. The highlighted line, which indicates the next line to be executed, will advance to the `pico_led_init` function call. To step into this function, press **Step-into** (**F11**). The source window will update to indicate execution is now at the beginning of the function. 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.

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 Pico SDK makes use of CMake to control its build system, see Appendix C.2. 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:

Code

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

Your “Hello world” code is now running.

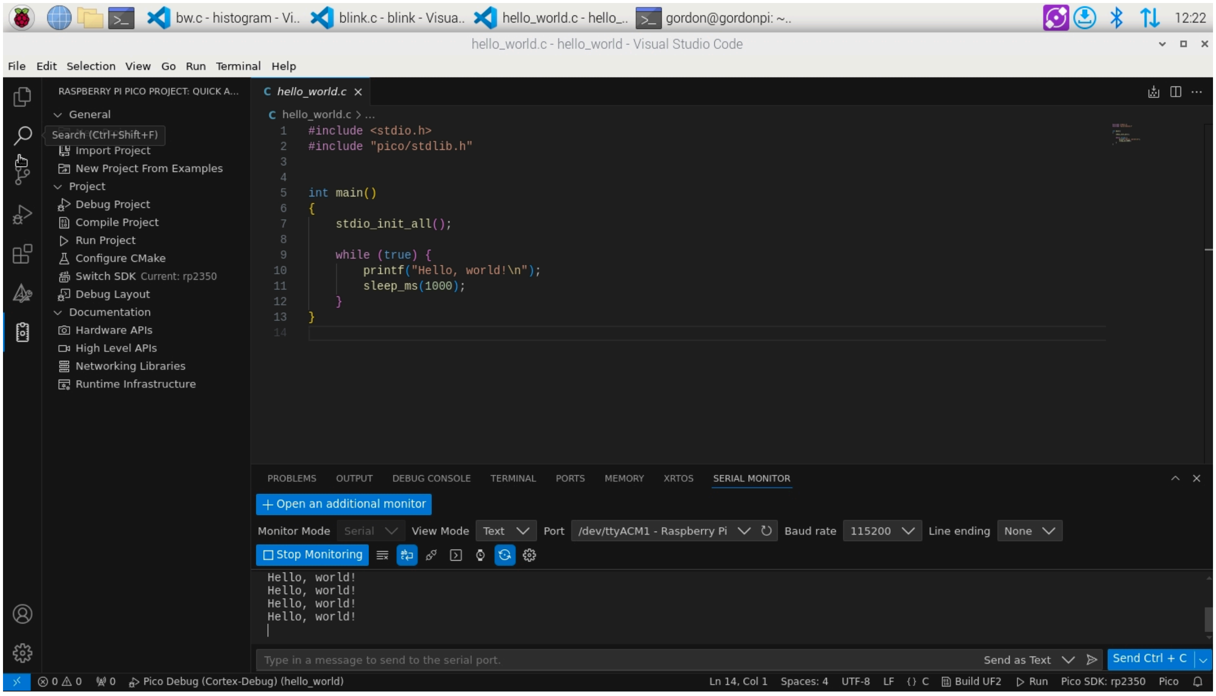
Although the “Hello World” example is now running, we cannot yet see the text.

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 including Windows, macOS, and Linux. The debugging device connects to your usual computer using USB, and to the Pico using SWD and UART.

A.1. 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 documemtation site .

A.1.1. Debug Probe wiring

Figure 7.
Wires included with the Debug Probe.

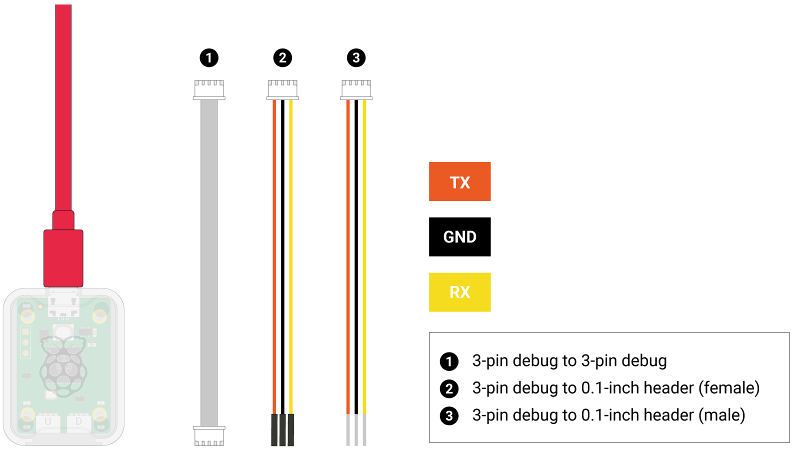
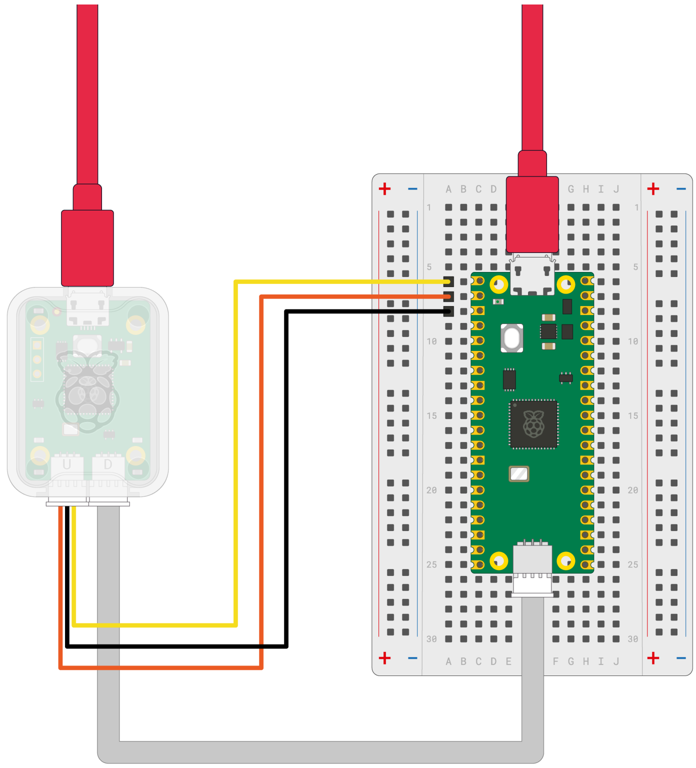


Figure 8.
Wiring between the Debug Probe (left) and Pico (right).



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.

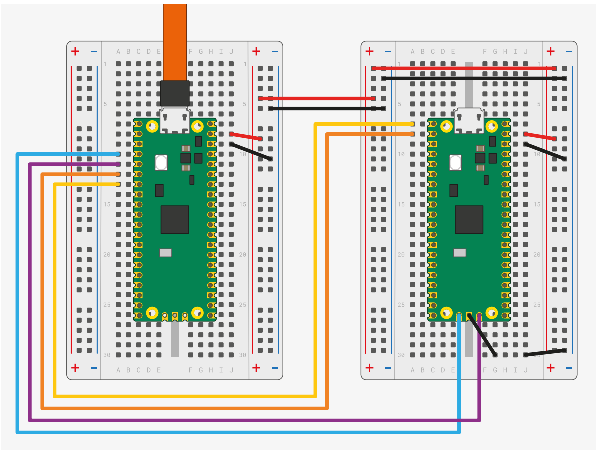
> **Note**
>
> If you have a non-H Pico, Pico 2 or Pico W (without a JST-SH connector) you can still connect it to a Debug Probe. Solder a male connector to the `SWCLK`, `GND`, and `SWDIO` header pins on the board. Using the alternate 3-pin JST-SH connector to 0.1-inch header (female) cable included 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 8.

A.2. 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 Appendix A.3. `debugprobe` wiring.



A.2.1. Install `debugprobe`

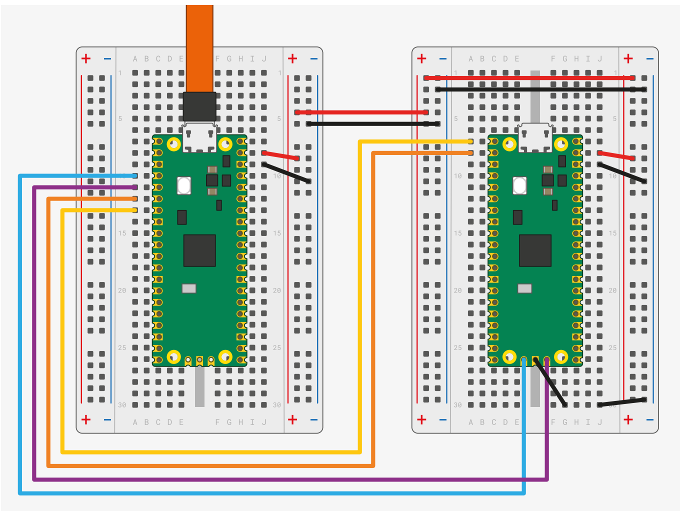
You can download a UF2 binary of `debugprobe` from the latest debugprobe release on GitHub.

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, or `debugprobe_on_pico2.uf2` for a Pico 2. Use `debugprobe.uf2` for the Debug Probe accessory hardware.

A.3. `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.

Code
```
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 5 V on the USB connector.

A.4. 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.

A.5. Use the UART

A.5.1. Linux

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

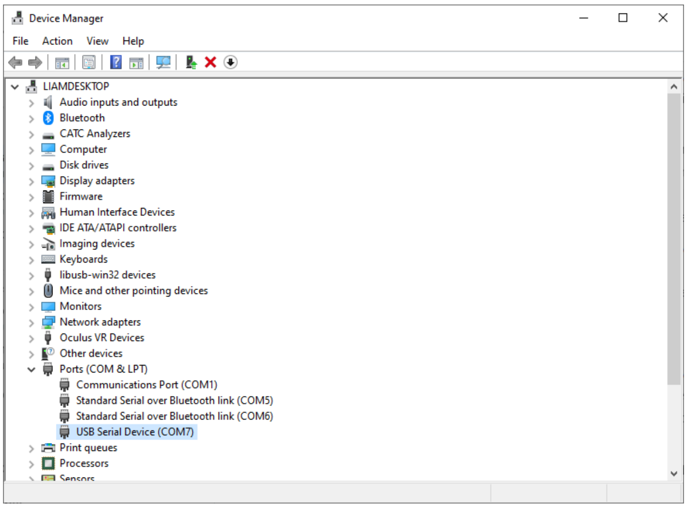
Code
```
$ sudo minicom -D /dev/ttyACM0 -b 115200
```

A.5.2. Windows

Download and install PuTTY.

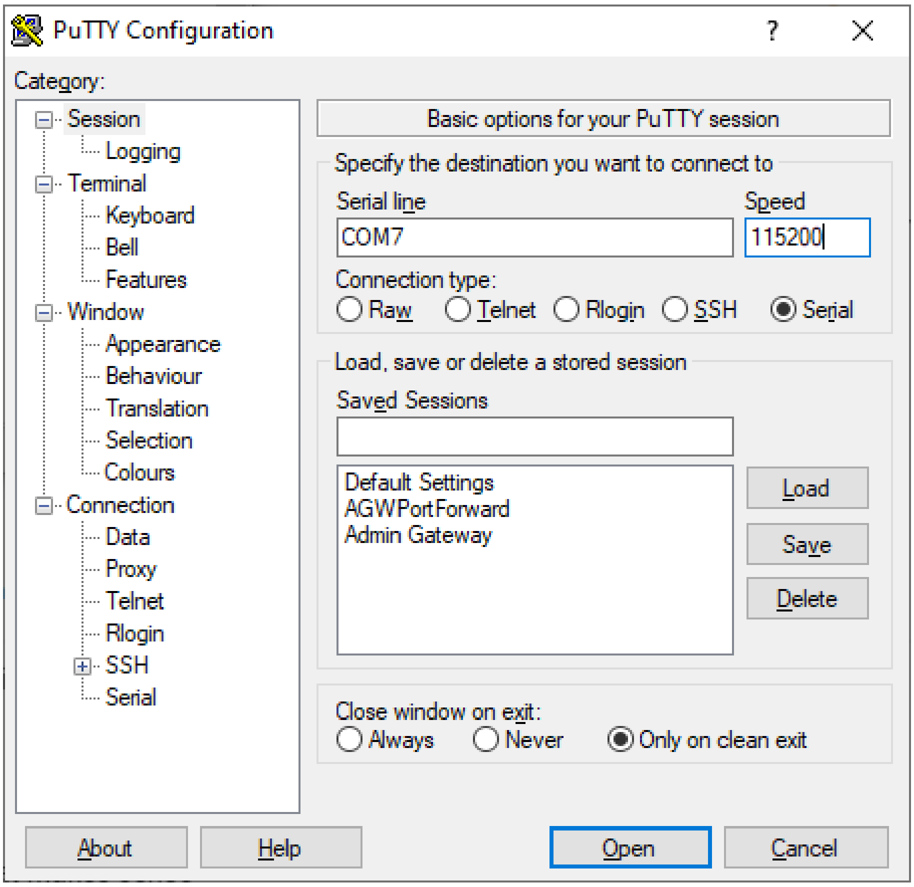
Open Device Manager and locate the COM port number of the device running `debugprobe`. In this example it is `COM7`.

Figure 11.



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

Figure 12.



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

Figure 13.

A.5.3. macOS

First, install `minicom` using Homebrew:

Code
```
$ brew install minicom
```

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

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

A.6. Debug with OpenOCD

A.6.1. Getting OpenOCD

Binary releases of OpenOCD which support all Pico-series devices are available from the Pico SDK tools repo on GitHub. You can also build and install it yourself with these commands:

Code
```
$ git clone https://github.com/raspberrypi/openocd.git --branch sdk-2.2.0
$ cd openocd
$ ./bootstrap
$ ./configure --disable-werror
$ make -j4
$ sudo make install
```

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:

Code
```
$ 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

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

A.6.2. Debug with SWD

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

Important

To make debugging easier, you can build your binaries with the `Debug` build type using the `CMAKE_BUILD_TYPE` option:

Code
```
$ cd ~/pico/pico-examples/
$ cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug -DPICO_BOARD=pico
$ cmake --build build
```

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

The debug build performs less optimisation, which makes it easier to see what's going on when you run it under the debugger.

First, run an OpenOCD server:

Code
```
$ 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:

Code

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

A.6.3. Rescue Debug Port

The rescue debug port (DP) on RP2040 or RP2350 can be used to reset the chip into a known state if the user has programmed some bad code into the flash. For example, code that turned off the system clock would stop the processor debug ports being accessed, but the rescue DP would still work because it is clocked from the `SWCLK` of the SWD interface. A successful Rescue DP operation will reset the chip and halt the bootrom early in the boot flow.

A.6.3.1. Activating the Rescue DP from OpenOCD

Two targets are provided that access the rescue DP:

- `rp2040-rescue.cfg`
- `rp2350-rescue.cfg`

Start OpenOCD with the configuration file that corresponds to your target device.

Code

```
$ openocd -f interface/cmsis-dap.cfg -f target/rp2040-rescue.cfg
...
Warn : gdb services need one or more targets defined
Now attach a debugger to your RP2040 and load some code
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections

Ctrl + C
```

Now start OpenOCD with the normal configuration.

Code

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

Note

On RP2350, the effects of the `CRIT1.DEBUG_DISABLE` and `CRIT1.SECURE_DEBUG_DISABLE` OTP flags (or setting a debug key index) cannot be overridden by the Rescue DP reset.

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`.

B.1. Getting picotool

The `picotool` utility is available in its own repository. If you're not using the VS Code extension, or you haven't run the pico-setup script, then you will need to either download the pre-built binaries or build it yourself - details for doing both can be found in the picotool README.md.

B.2. Using picotool

The `picotool` binary includes a command-line help function, which lists all the available commands:

Code
```
$ picotool help
PICOTOOL:
    Tool for interacting with RP-series device(s) in BOOTSEL mode, or with an RP-series binary

SYNOPSIS:
    picotool info [-b] [-m] [-p] [-d] [--debug] [-l] [-a] [device-selection]
    picotool info [-b] [-m] [-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 [--ignore-partitions] [--family <family_id>] [-p <partition>] [-n] [-N] [-u]
                [-v] [-x] <filename> [-t <type>] [-o <offset>] [device-selection]
    picotool encrypt [--quiet] [--verbose] [--embed] [--fast-rosc] [--use-mbedtls]
                [--otp-key-page <page>] [--hash] [--sign] <infile> [-t <type>] [-o <offset>]
                <outfile> [-t <type>] <aes_key> <iv_salt> <signing_key> <otp>
    picotool seal [--quiet] [--verbose] [--hash] [--sign] [--clear] <infile> [-t <type>] [-o
                <offset>] <outfile> [-t <type>] <key> <otp> [--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] [-v] [--family <family_id>] <filename> [-t <type>] [device-selection]
    picotool save -a [-v] [--family <family_id>] <filename> [-t <type>] [device-selection]
    picotool save -r <from> <to> [-v] [--family <family_id>] <filename> [-t <type>]
                [device-selection]
    picotool erase [-a] [device-selection]
    picotool erase -p <partition> [device-selection]
    picotool erase -r <from> <to> [device-selection]
    picotool verify <filename> [-t <type>] [device-selection] [-r <from> <to>] [-o <offset>]
                [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> <outfile>
    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
                RP-series 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.
    erase       Erase the program / memory stored in flash on the device.
    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 disassembly 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.

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

If you get this message with RP2040 on Windows you will need to install a driver. This is not necessary for RP2350.

Download and run Zadig, select `RP2 Boot (Interface 1)` from the dropdown box and select `WinUSB` as the driver, and click on the “Install Driver” button. Wait for the installation to complete - this may take a few minutes.

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 Pico SDK by using the `-f` argument of `picotool`.

B.2.1. Display information

So there is now *Binary Information* support in the SDK which allows for easily storing compact information that `picotool` can find (See Appendix B.2.3. 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.

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

SYNOPSIS:
    picotool info [-b] [-m] [-p] [-d] [--debug] [-l] [-a] [device-selection]
    picotool info [-b] [-m] [-p] [-d] [--debug] [-l] [-a] <filename> [-t <type>]

OPTIONS:
    Information to display
        -b, --basic
            Include basic information. This is the default
        -m, --metadata
            Include all metadata blocks
        -p, --pins
            Include pin information
```

```
        -d, --device
            Include device information
        --debug
            Include device debug information
        -l, --build
            Include build attributes
        -a, --all
            Include all information

TARGET SELECTION:
    To target one or more connected RP-series device(s) in BOOTSEL mode (the default)
        --bus <bus>
            Filter devices by USB bus number
        --address <addr>
            Filter devices by USB device address
        --vid <vid>
            Filter by vendor id
        --pid <pid>
            Filter by product id
        --ser <ser>
            Filter by serial number
        -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 USB 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,

Code

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

or,

Code

```
$ 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,

Code
```
$ 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,

Code
```
$ 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
```

B.2.2. 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.

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

SYNOPSIS:
    picotool save [-p] [-v] [--family <family_id>] <filename> [-t <type>] [device-selection]
    picotool save -a [-v] [--family <family_id>] <filename> [-t <type>] [device-selection]
    picotool save -r <from> <to> [-v] [--family <family_id>] <filename> [-t <type>]
                [device-selection]

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
    Other
        -v, --verify
            Verify the data was saved correctly
```

```
        --family
            Specify the family ID to save the file as
        <family_id>
            family ID to save file as
    File to save to
        <filename>
            The file name
        -t <type>
            Specify file type (uf2 | elf | bin) explicitly, ignoring file extension
    Source device selection
        --bus <bus>
            Filter devices by USB bus number
        --address <addr>
            Filter devices by USB device address
        --vid <vid>
            Filter by vendor id
        --pid <pid>
            Filter by product id
        --ser <ser>
            Filter by serial number
        -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 USB drive mounted
```

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
```

B.2.3. Binary Information

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

B.2.4. 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)

B.2.5. 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:

Code
```
$ 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
```

B.2.6. Full Information

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

Code
```
$ 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. Command line toolchain setup

C.1. Configure your environment via script

If you are developing for a Pico-series device on a Raspberry Pi running Raspberry Pi OS, then you can use the `pico_setup.sh` script to set up a command line environment.

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`
- Downloads, builds & installs `picotool` (see Appendix B. Picotool).
- Downloads and builds `debugprobe` (see Appendix A. Debugprobe).
- Builds the `blink` and `hello_world` example projects
- Downloads and compiles OpenOCD (for debug support)
- Configures your development Raspberry Pi UART for use with Pico-series devices

The `pico_setup.sh` README.md also contains instructions about how to build, load, and debug programs on a Pico-series device from the command line.

> **Tip**
>
> This setup script requires approximately 4 GB 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`:

Code
```
$ sudo apt install wget
```

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

Then mark the script as executable with `chmod`:

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

Run the script with the following command:

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

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

Code
```
$ sudo reboot
```

C.1.1. Manually Configure your Environment

See the `pico_setup.sh` README.md for details on manually configuring your environment.

C.1.2. Update the Pico SDK

When a new version of the Pico SDK is released, you must update your copy of the SDK, and it's recommended to delete any local build directories. To update, navigate into `pico-sdk` and run the following command (you should also do the same for `pico-examples`):

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

You will also need to update your copy of picotool and rebuild and reinstall it - to do this, navigate into `picotool` and run the following commands:

```
$ cd picotool
$ git pull
$ cmake --build build
$ sudo cmake --install build
```

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

C.2. 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,

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "pico/binary_info.h"

const uint LED_PIN = 25;<1>

int main() {

    bi_decl(bi_program_description("This is a test binary."));<2>
    bi_decl(bi_1pin_with_name(LED_PIN, "On-board LED"));

    stdio_init_all();

    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (1) {
        gpio_put(LED_PIN, 0);
        sleep_ms(250);
        gpio_put(LED_PIN, 1);
        puts("Hello World\n");
        sleep_ms(1000);
    }
}
```

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

along with a `CMakeLists.txt` file,

Code
```
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
)

target_link_libraries(test pico_stdlib)

pico_enable_stdio_usb(test 1)<1>
pico_enable_stdio_uart(test 1)<2>

pico_add_extra_outputs(test)
```

Enables serial output via USB. This creates a USB CDC interface to add support for stdin/stdout over USB serial (CDC). Enables serial output via UART. This adds support for stdin/stdout over the default UART pins on the board (GPIOs 0 & 1 for Pico-series devices).

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

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

You should now have something that looks like this,

Code
```
$ 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 the same way as `pico-examples`.

Code
```
$ cmake -S . -B build
$ cmake --build build
```

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 build 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

File extension	Description
.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.

C.2.1. Debugging your project

Debugging your own project from the command line follows the same processes in Appendix A.6.2. 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.

D.1. 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.

D.1.1. 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/

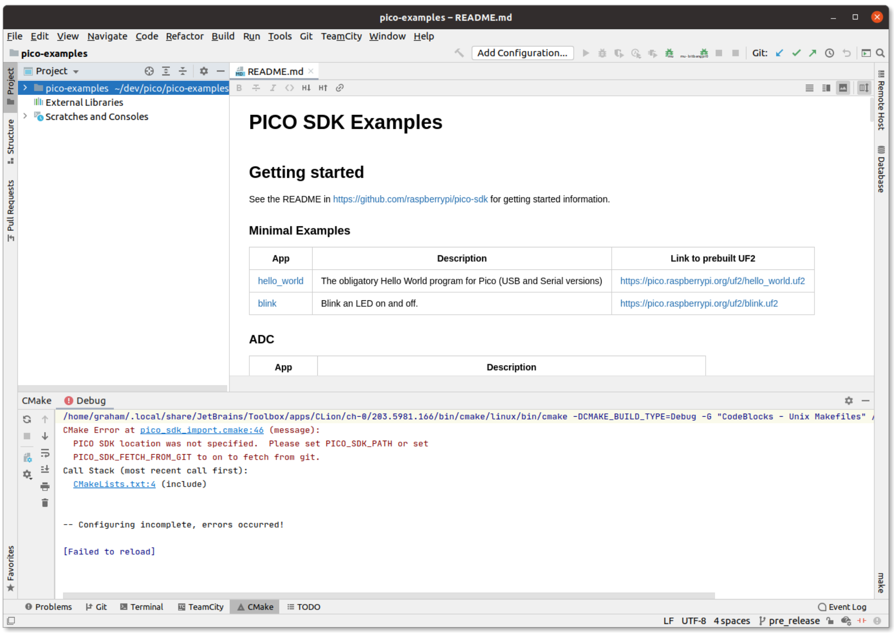
D.1.1.1. Setting up a project

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 14.

Figure 14.
A newly opened CLion pico-examples project.



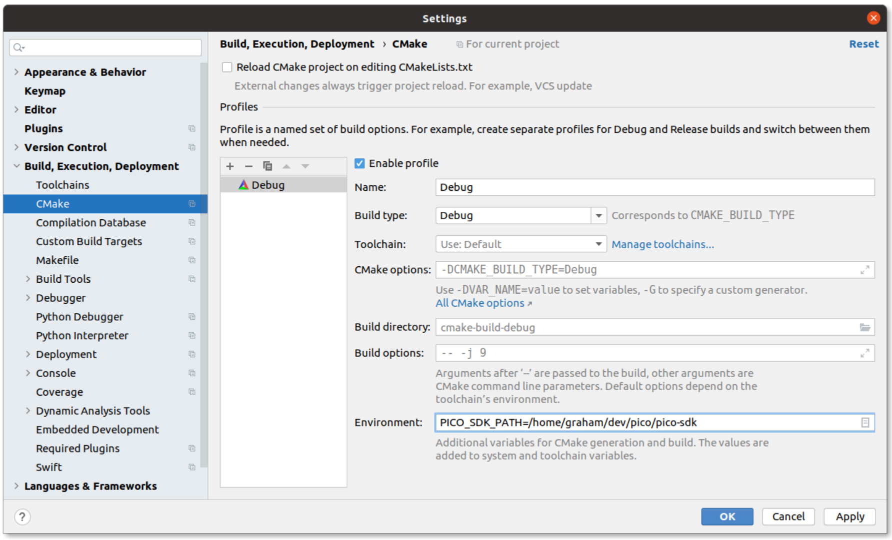
Notice at the bottom that CLion attempted to load the CMake project, but there was an error; namely that we hadn't specified `PICO_SDK_PATH`

D.1.1.1.1. 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 15, 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.

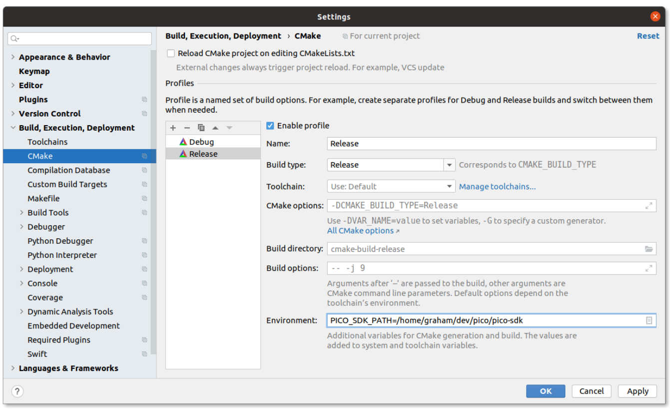
Figure 15.
Configuring a CMake profile in CLion.



> **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"`.

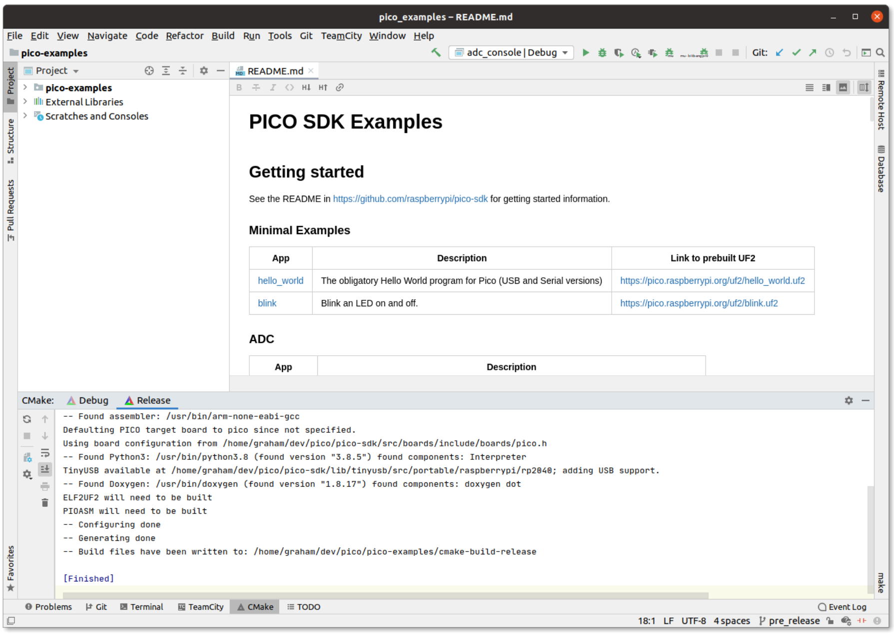
You can have as many CMake profiles as you like with different settings. You probably want to add a `Release` build by hitting the `+` button, and then filling in the PICO_SDK_PATH again, or by hitting the copy button two to the right, and fixing the name and settings (see Figure 16)

Figure 16.
Configuring a second CMake Profile in CLion.



After pressing OK, you'll see something like Figure 17. 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.

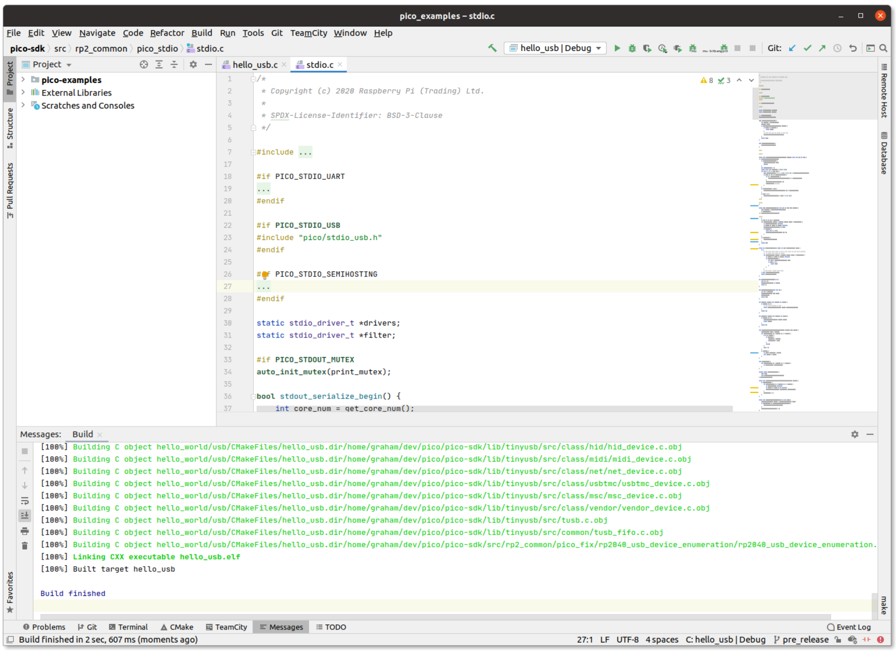
Figure 17.
Configuring a second CMake profile in CLion.



D.1.1.1.2. 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 18). Alternatively you can do a full build of all targets or other types of build from the `Build` menu.

Figure 18.
`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`)

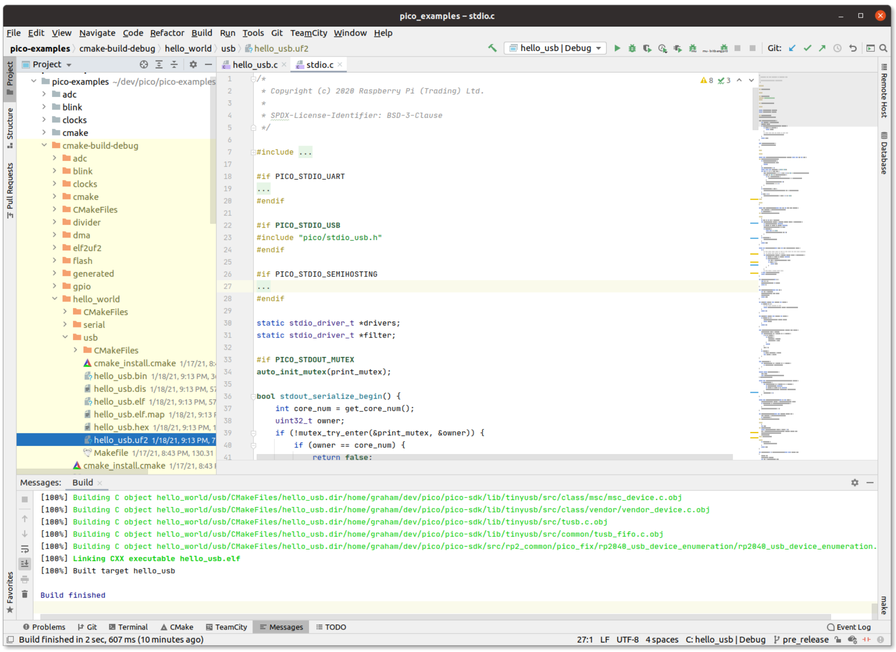
Another thing you'll notice Figure 18 shows is that in the bottom status bar, you can see `hello_usb` and `Debug` again. These are showing you the target and CMake profile being used to control syntax highlighting etc. in the editor (This was auto selected when you chose `hello_usb` before). You can 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 Pico SDK, so this is a very nice feature.

D.1.1.1.3. Build Artifacts

The build artifacts are located under `cmake-build-<profile>` under the project root (see Figure 19). 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 19.
Locating the `hello_usb` build artifacts



D.2. Other Environments

There are too many development environments available to describe all of them here. You can use many of them with the Pico 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 E. Documentation Release History

E.1. 03 July 2026

- Converted to new build toolchain.
- Refocus this guide on using VS Code:
  - Removed Eclipse IDE instructions.
  - Removed some command-line instructions. For information about command-line set up, see the `pico-setup.sh` README.
- Improved debugprobe section.

E.2. 15 October 2024

- Corrected minor typos and formatting issues.
- Switched back to separate release histories per PDF.

E.3. 06 September 2024

- Corrected minor typos and formatting issues.
- Fixed some minor problems with environment setup.

E.4. 09 August 2024

- Renamed “Load a project” chapter to “Load and debug a project”
- Fixed incorrect image label.
- Reword *VSCode* as *VS Code*.

E.5. 08 August 2024

- Added support for Pico 2.

E.6. 02 May 2024

- Corrected minor typos and formatting issues.
- Renamed `picoprobe` firmware to `debugprobe`.
- Clarified that CMake build configuration uses cache variables, not configuration variables.
- Fixed incorrect parameter names used in the `@asm_pio` decorator and `StateMachine` initialiser examples.
- Expanded MicroPython `rshell` examples to include a complete guide on loading and running programs on your device.
- Added an example that demonstrates how to reset a Pico from the command line using OpenOCD.
- Enhanced VS Code MicroPico plugin documentation to reflect the plugin’s new name, removal of the built-in FTP server, and some additional usage instruction.
- Added documentation on the official Raspberry Pi Pico VS Code extension.

E.7. 02 February 2024

- Corrected minor typos and formatting issues.
- Updated Getting Started documentation for MS Windows and Apple macOS.
- Updated documentation to include information about Raspberry Pi 5.

E.8. 14 June 2023

- Corrected minor typos and formatting issues.
- Updated instructions for the 1.5.1 release of the Raspberry Pi Pico C SDK.

E.9. 03 March 2023

- Corrected minor typos and formatting issues.
- Updated instructions for the 1.5.0 release of the Raspberry Pi Pico C SDK.
- Added documentation around the new Pico Windows Installer.

E.10. 01 December 2022

- Corrected minor typos and formatting issues.
- Updated `picoprobe` build and usage instructions.

E.11. 30 June 2022

- Corrected minor typos and formatting issues.

E.12. 17 June 2022

- Corrected minor typos and formatting issues.
- Updated setup instructions for Windows.
- Added additional explanation of SDK configuration.

E.13. 04 November 2021

- Corrected minor typos and formatting issues.
- Updated links to documentation.

E.14. 03 November 2021

- Corrected minor typos and formatting issues.
- Updated `picotool` documentation.

E.15. 30 September 2021

- Corrected minor typos and formatting issues.

E.16. 23 June 2021

- Corrected minor typos and formatting issues.

E.17. 07 June 2021

- Corrected minor typos and formatting issues.

E.18. 13 April 2021

- Corrected minor typos and formatting issues.
- Clarified that all source code in the documentation is under the 3-Clause BSD license.

E.19. 07 April 2021

- Corrected minor typos and formatting issues.

E.20. 05 March 2021

- Corrected minor typos and formatting issues.

E.21. 23 February 2021

- Corrected minor typos and formatting issues.
- Changed font.
- Made major improvements to SWD documentation.
- Updated MicroPython build instructions.
- Added MicroPython UART example code.
- Updated Thonny instructions.
- Updated Project Generator instructions.

E.22. 01 February 2021

- Corrected minor typos and formatting issues.
- Explained how to access a MicroPython REPL over UART.

E.23. 26 January 2021

- Corrected minor typos and formatting issues.
- Updated Windows and macOS build instructions.
- Renamed books and optimised size of output PDFs.

E.24. 21 January 2021

- Initial release.

Raspberry Pi