

Getting started with Raspberry Pi Pico

C/C++ development with
Raspberry Pi Pico and
other RP2040-based
microcontroller boards

Colophon

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

The documentation of the RP2040 microcontroller is licensed under a Creative Commons [Attribution-NoDerivatives 4.0 International](#) (CC BY-ND).

build-date: 2024-05-02

build-version: 576cee3-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 C/C++ SDK](#) book. Source code included in the documentation is

Copyright © 2020-2023 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. Quick Pico Setup	4
2. The SDK	6
2.1. Get the SDK and examples	6
2.2. Install the Toolchain	7
2.3. Updating the SDK	7
3. Blinking an LED in C	8
3.1. Building "Blink"	8
3.1.1. Build "Blink" for the Pico	9
3.1.2. Build "Blink" for the Pico W	10
3.2. Load and run "Blink"	10
3.2.1. Using the command line	11
3.2.2. Aside: Other Boards	12
4. Saying "Hello World" in C	13
4.1. Serial input and output on Raspberry Pi Pico	13
4.2. Build "Hello World"	14
4.3. Flash and Run "Hello World"	15
4.4. See "Hello World" USB output	15
4.5. See "Hello World" UART output	16
4.6. Powering the board	17
4.6.1. Aside: Hands-free Flash Programming	18
5. Flash Programming with SWD	19
5.1. Installing OpenOCD	19
5.1.1. Installing OpenOCD on macOS	20
5.2. SWD Port Wiring	20
5.3. Loading a Program	21
6. Debugging with SWD	24
6.1. Build "Hello World" debug version	24
6.2. Installing GDB	24
6.3. Use GDB and OpenOCD to debug Hello World	25
7. Using Visual Studio Code	27
7.1. Installing Visual Studio Code	27
7.2. Raspberry Pi Pico VS Code Extension	27
7.2.1. Install the VS Code extension	27
7.2.2. Create a project	29
7.2.3. Build	30
7.2.4. Run	31
7.2.5. Debug	31
7.3. Configure Manually	32
7.4. Loading a Project	32
7.5. Debugging a Project	34
7.5.1. Running "Hello USB" on the Pico	35
8. Creating your own Project	37
8.1. Debugging your project	39
8.2. Working in Visual Studio Code	40
8.3. Automating project creation	40
8.3.1. Project generation from the command line	42
9. Building on other platforms	44
9.1. Building on Apple macOS	44
9.1.1. Installing the Toolchain	44
9.1.2. Using Visual Studio Code	44
9.1.3. Building with CMake Tools	44
9.1.4. Saying "Hello World"	46
9.2. Building on MS Windows	46

9.2.1. Installing the Toolchain	47
9.2.2. Alternative manual installation	48
10. Using other Integrated Development Environments	56
10.1. Using Eclipse	56
10.1.1. Setting up Eclipse for Pico on a Linux machine	56
10.2. Using CLion	61
10.2.1. Setting up CLion	61
10.3. Other Environments	65
10.3.1. Using openocd-svd	65
Appendix A: Using debugprobe	67
Install OpenOCD	67
Debug Probe	67
Debug Probe wiring	67
Debug with a second Pico	69
Install <code>debugprobe</code>	69
<code>debugprobe</code> wiring	70
Debug Probe interfaces	70
Use the UART	71
Linux	71
Windows	71
macOS	72
Debug with OpenOCD	73
Debug with SWD	73
Appendix B: Using Picotool	74
Getting picotool	74
Building picotool	74
Using picotool	75
Display information	76
Save the program	78
Binary Information	79
Basic information	79
Pins	79
Including Binary information	80
Details	80
Setting common fields from CMake	82
Appendix C: Documentation release history	83

Chapter 1. Quick Pico Setup

If you are developing for Raspberry Pi Pico on 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 setup script.

NOTE

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.

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
```

1. You should first `sudo apt install wget` if you don't have `wget` already installed.

Then make the script executable with,

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

and run it with,

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

The script will:

- Create a directory called `pico`
- Install required dependencies
- Download the `pico-sdk`, `pico-examples`, `pico-extras`, and `pico-playground` repositories
- Define `PICO_SDK_PATH`, `PICO_EXAMPLES_PATH`, `PICO_EXTRAS_PATH`, and `PICO_PLAYGROUND_PATH` in your `~/.bashrc`
- Build the `blink` and `hello_world` examples in `pico-examples/build/blink` and `pico-examples/build/hello_world`
- Download and build `picotool` (see [Appendix B](#)), and copy it to `/usr/local/bin`.
- Download and build `debugprobe` (see [Appendix A](#)).
- Download and compile OpenOCD (for debug support)
- Download and install [Visual Studio Code](#)
- Install the required Visual Studio Code extensions (see [Chapter 7](#) for more details)
- Configure the Raspberry Pi UART for use with Raspberry Pi Pico

 **NOTE**

The `pico` directory will be created in the folder where you *run* the `pico_setup.sh` script.

Once it has run, you will need to reboot your Raspberry Pi,

```
$ sudo reboot
```

for the UART reconfiguration to take effect. Once your Raspberry Pi has rebooted you can open Visual Studio Code in the "Programming" menu and follow the instructions from [Section 7.4](#).

Chapter 2. The SDK

! IMPORTANT

The following instructions assume that you are using a Raspberry Pi Pico and some details may differ if you are using a different RP2040-based board. They also assume you are using Raspberry Pi OS running on a Raspberry Pi 4, or an equivalent Debian-based Linux distribution running on another platform. Alternative instructions for those using Microsoft Windows (see [Section 9.2](#)) or Apple macOS (see [Section 9.1](#)) are also provided.

The Raspberry Pi Pico is built around the RP2040 microcontroller designed by Raspberry Pi. Development on the board 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

For more information on the official MicroPython port see the [Raspberry Pi Pico Python SDK](#) book which documents the port, and [Get started with MicroPython on Raspberry Pi Pico](#) by Gareth Halfacree and Ben Everard, published by Raspberry Pi Press.

💡 TIP

For more information on the C/C++ SDK, along with API-level documentation, see the [Raspberry Pi Pico C/C++ SDK](#) book.

2.1. Get the SDK and examples

The [Pico Examples](#) repository provides a set of example applications that are written using the [SDK](#). To clone these repositories start by creating a pico directory to keep all pico related checkouts in. These instructions create a pico directory at `/home/pi/pico`.

```
$ cd  
$ mkdir pico  
$ cd pico
```

Then clone the `pico-sdk` and `pico-examples` git repositories.

```
$ 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
```

⚠️ WARNING

Failure to run the `git submodule update --init` command above will mean that the `tinyusb` module will not be included, and as a result USB functionality will not be compiled into the SDK. This means that USB serial, other USB functions, and example code will not work.

ℹ️ NOTE

There are additional repositories: [Pico Extras](#), and [Pico Playground](#) that you may also be interested in.

2.2. 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, and the [GNU Embedded Toolchain for Arm](#). You can install both of these via `apt` from the command line. Anything you already have installed will be ignored by `apt`.

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

1. Native gcc and g++ are needed to compile `pioasm` and `elf2uf2`.

ℹ️ NOTE

Ubuntu and Debian users might additionally need to also install `libstdc++-arm-none-eabi-newlib`.

2.3. Updating the SDK

When a new version of the SDK is released you will need to update your copy of the SDK. To do this go into the `pico-sdk` directory which contains your copy of the SDK, and do the following:

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

ℹ️ NOTE

If you wish to be informed of new releases you can get notified by setting up a custom watch on the `pico-sdk` repository. Navigate to <https://github.com/raspberrypi/pico-sdk> and then select Watch → Custom → Releases. You will receive an email notification every time there is a new SDK release.

Chapter 3. Blinking 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 go ahead blink the LED on the Raspberry Pi Pico.

This example uses the following code for the Raspberry Pi Pico, blinking the LED connected to pin 25 of the RP2040:

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

```

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

And the following alternate code for the Raspberry Pi Pico W, which blinks the LED connected to the Infineon 43439 wireless chip with `WL_GPIO0`:

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pico_w/wifi/blink/picow_blink.c Lines 10 - 22

```

10 int main() {
11     stdio_init_all();
12     if (cyw43_arch_init()) {
13         printf("Wi-Fi init failed");
14         return -1;
15     }
16     while (true) {
17         cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, 1);
18         sleep_ms(250);
19         cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, 0);
20         sleep_ms(250);
21     }
22 }
```

3.1. 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.

3.1.1. Build "Blink" for the Pico

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
```

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 applications, 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 RP2040 USB Mass Storage Device

3.1.2. Build "Blink" for the Pico W

Prepare your `cmake` build directory by running the following command, which specifies that CMake should target the Pico W board:

```
$ cmake .. -DPICO_BOARD=pico_w
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 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 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 now type `make` to build all example applications. However, for this example we only need to build `blink`. To build a specific subtree of applications, navigate into the corresponding subtree before running `make`. In this case, we can build only the `blink` task by first navigating to `pico_w/wifi/blink`, then running `make`:

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

.

.

[100%] Linking CXX executable picow_blink.elf
[100%] Built target picow_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:

`picow_blink.elf`
used by the debugger
`picow_blink.uf2`
the file we'll copy onto RP2040 USB Mass Storage Device

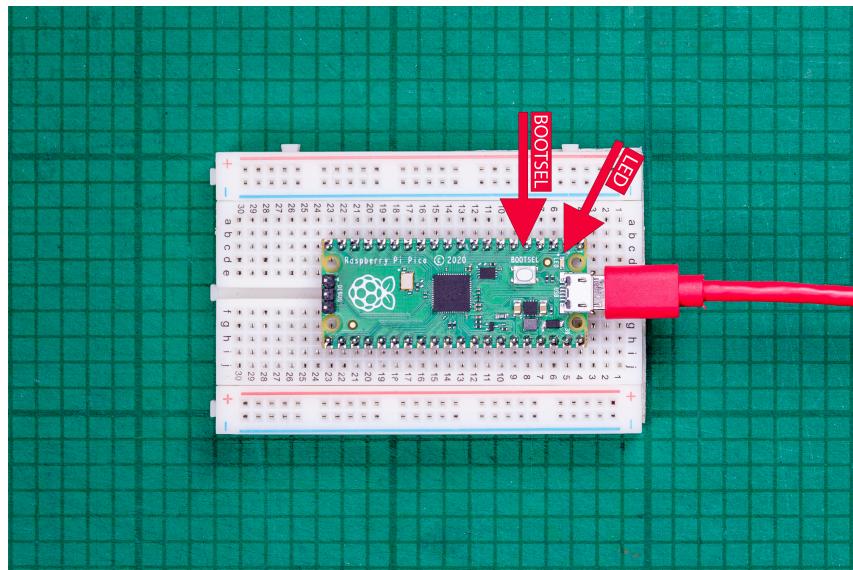
3.2. Load and run "Blink"

To load software onto a RP2040-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 1) while plugging in your device using a micro-USB cable to force it into USB Mass Storage Mode.

RP2040 will reboot, unmount itself as a Mass Storage Device, and run the flashed code, see Figure 1.

*Figure 1. Blinking the on-board LED on the Raspberry Pi Pico. Arrows point to the on-board LED, and the **BOOTSEL** button.*



3.2.1. Using the command line

💡 TIP

You can use **picotool** to load a UF2 binary onto your Raspberry Pi Pico, see [Appendix B](#).

Depending on the computer you use to compile binaries for your Pico, 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 RP2040:

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

RP2040 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.

3.2.2. Aside: Other Boards

If you are not following these instructions on a Raspberry Pi Pico, you may not have a **BOOTSEL** button (as labelled in [Figure 1](#)). Your board may have some other way of loading code, which the board supplier should have documented:

- Most boards expose the SWD interface ([Chapter 5](#)) 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 Raspberry Pi Pico), 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.

Chapter 4. Saying "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."

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/hello_world/serial/hello_serial.c Lines 10 - 16

```

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

4.1. Serial input and output on Raspberry Pi Pico

Serial input (`stdin`) and output (`stdout`) can be directed to either serial UART or to USB CDC (USB serial). However by default `stdio` and `printf` will target the default Raspberry Pi Pico UART0.

Default UART0	Physical Pin	GPIO Pin
GND	3	N/A
UART0_TX	1	GP0
UART0_RX	2	GP1

! IMPORTANT

The default Raspberry Pi Pico UART TX pin (out from Raspberry Pi Pico) is pin GP0, and the UART RX pin (in to Raspberry Pi Pico) is pin GP1. The default UART pins are configured on a per-board basis using board configuration files. The Raspberry Pi Pico configuration can be found in <https://github.com/raspberrypi/pico-sdk/blob/master/src/boards/include/boards/pico.h>. The SDK defaults to a board name of Raspberry Pi Pico if no other board is specified.

The SDK makes use of CMake to control its build system, see [Chapter 8](#), making use of the `pico_stdlib` interface library to aggregate necessary source files to provide capabilities.

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

```

1 add_executable(hello_serial
2     hello_serial.c
3     )
4
5 # pull in common dependencies
6 target_link_libraries(hello_serial pico_stdlib)
7
8 # create map/bin/hex/uf2 file etc.
9 pico_add_extra_outputs(hello_serial)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(hello_serial)
```

The destination for `stdout` can be changed using CMake directives, with output directed to UART or USB CDC, or to both,

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

1. Enable `printf` output via USB CDC (USB serial)
2. Disable `printf` output via UART

This means that **without changing the C source code**, you can change the destination for `stdio` from UART to USB.

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

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

4.2. Build "Hello World"

As we did for the previous "Blink" example, change directory into the `hello_world` directory inside the `pico-examples/build` tree, and run `make`.

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

[ 33%] Linking CXX executable hello_usb.elf
[ 33%] Built target hello_usb

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

This will build two separate examples programs in the `hello_world/serial/` and `hello_world/usb/` directories.

Amongst other targets, we have now built:

- `serial/hello_serial.elf`, which is used by the debugger
- `serial/hello_serial.uf2`, which can be dragged onto the RP2040 USB Mass Storage Device (UART serial binary)
- `usb/hello_usb.elf`, which is used by the debugger
- `usb/hello_usb.uf2`, which can be dragged onto the RP2040 USB Mass Storage Device (USB serial binary)

Where `hello_serial` directs `stdio` to UART0 on pins GP0 and GP1, and `hello_usb` directs `stdio` to USB CDC serial.

⚠️ WARNING

If you have not initialised the `tinyusb` submodule in your `pico-sdk` checkout then the USB CDC serial example will not work as the SDK will contain no USB functionality.

4.3. Flash and Run "Hello World"

Connect the Raspberry Pi Pico to your Raspberry Pi using a micro-USB cable, making sure that you hold down the `BOOTSEL` button to force it into USB Mass Storage Mode. Once it is connected release the `BOOTSEL` button and if you are running the Raspberry Pi Desktop it should automatically mount as a USB Mass Storage Device. From here, you can Drag-and-drop either the `hello_serial.uf2` or `hello_usb.uf2` onto the Mass Storage Device.

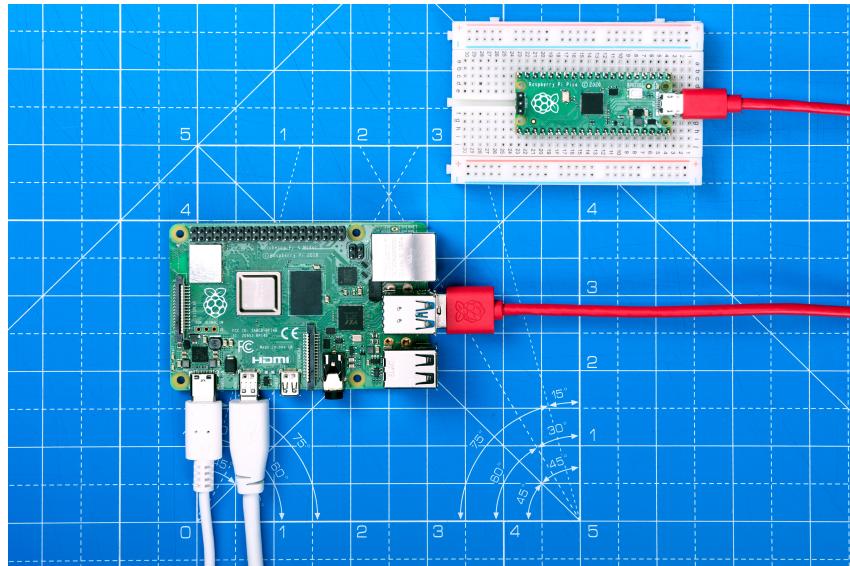
RP2040 will reboot, unmounting itself as a Mass Storage Device, and start to run the flashed code.

However, although the "Hello World" example is now running, we cannot yet see the text. We need to connect our host computer to the appropriate `stdio` interface on the Raspberry Pi Pico to see the output.

4.4. See "Hello World" USB output

If you have dragged and dropped the `hello_usb.uf2` binary, then the "Hello World" text will be directed to USB serial.

Figure 2. Connecting the Raspberry Pi to Raspberry Pi Pico via USB.



With your Raspberry Pi Pico connected directly to your Raspberry Pi via USB, see [Figure 2](#), you can see the text by installing `minicom`:

```
$ sudo apt install minicom
```

and open the serial port:

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

You should see `Hello, world!` printed to the console.

💡 TIP

To exit minicom, use `CTRL-A` followed by `X`.

ℹ NOTE

If you are intending to use SWD for debugging (see [Chapter 6](#)) you need to use a UART based serial connection as the USB stack will be paused when the RP2040 cores are stopped during debugging, which will cause any attached USB devices to disconnect.

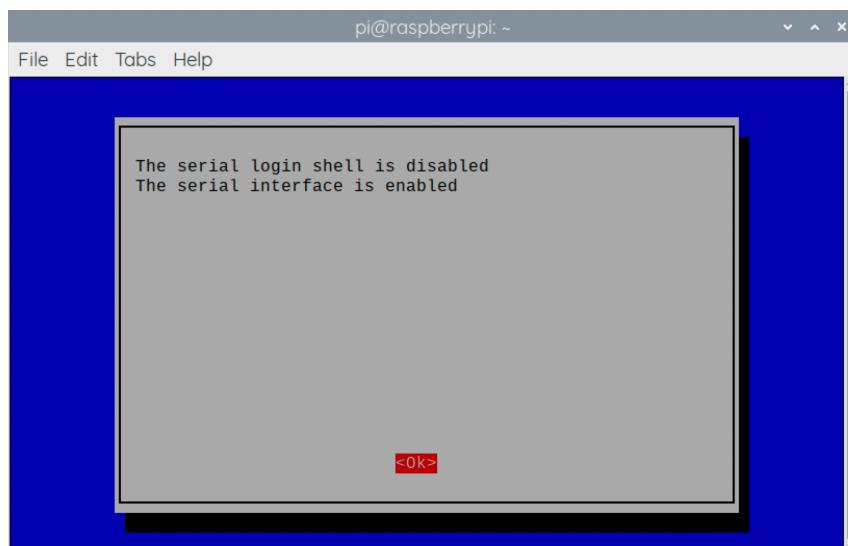
4.5. See "Hello World" UART output

Alternatively if you dragged and dropped the `hello_serial.uf2` binary, then the "Hello World" text will be directed to UART0 on pins GP0 and GP1. The first thing you'll need to do to see the text is enable UART serial communications on the Raspberry Pi host. To do so, run `raspi-config`,

```
$ sudo raspi-config
```

and go to [Interfacing Options](#) → [Serial](#) and select "No" when asked "Would you like a login shell to be accessible over serial?" and "Yes" when asked "Would you like the serial port hardware to be enabled?" You should see something like [Figure 3](#).

Figure 3. Enabling a serial UART using `raspi-config` on the Raspberry Pi.



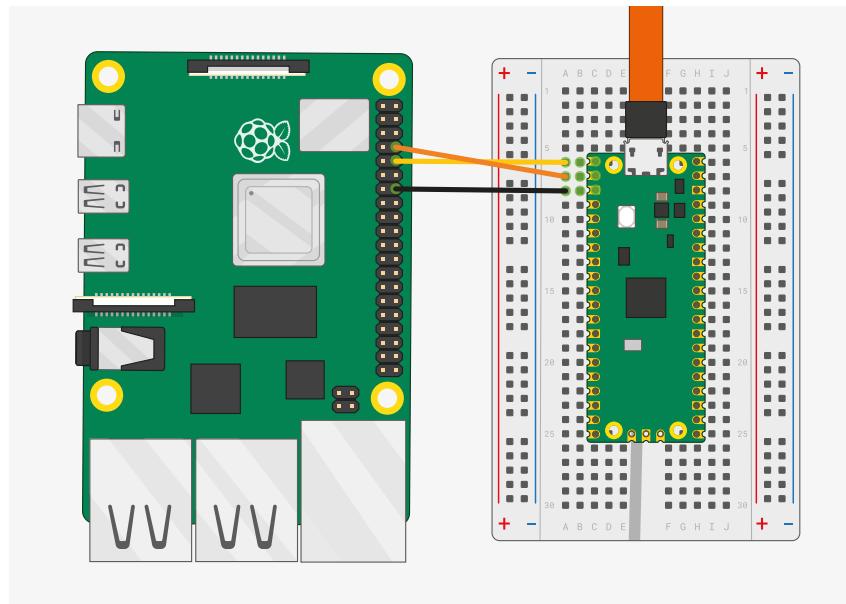
Leaving `raspi-config` you should choose "Yes" and reboot your Raspberry Pi to enable the serial port.

You should then wire the Raspberry Pi and the Raspberry Pi Pico together with the following mapping:

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

See [Figure 4](#).

Figure 4. A Raspberry Pi 4 and the Raspberry Pi Pico with UART0 connected together.



Once the two boards are wired together if you have not already done so you should install [minicom](#):

```
$ sudo apt install minicom
```

and open the serial port:

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

You should see `Hello, world!` printed to the console.

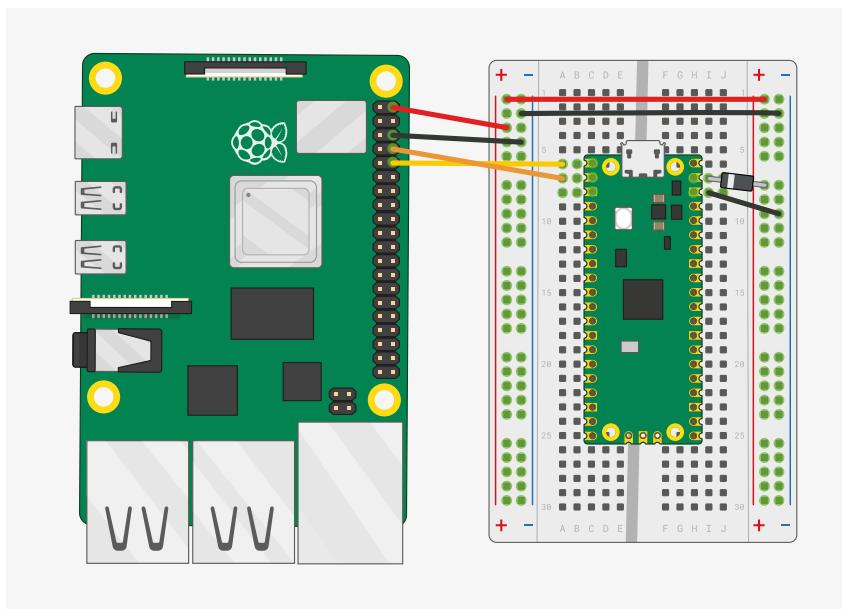
 **TIP**

To exit minicom, use `CTRL-A` followed by `X`.

4.6. Powering the board

You can unplug the Raspberry Pi Pico from USB, and power the board by additionally connecting the Raspberry Pi's 5V pin to the Raspberry Pi Pico VSYS pin via a diode, see [Figure 5](#), where in the ideal case the diode would be a [Schottky diode](#).

Figure 5. Raspberry Pi and Raspberry Pi Pico connected only using the GPIO pins.



Whilst it is possible to connect the Raspberry Pi's 5V pin to the Raspberry Pi Pico VBUS pin, this is not recommended. Shorting the 5V rails together will mean that the Micro USB cannot be used. An exception is when using the Raspberry Pi Pico in USB host mode, in this case 5V must be connected to the VBUS pin.

The 3.3V pin is an OUTPUT pin on the Raspberry Pi Pico, you cannot power the Raspberry Pi Pico via this pin, and it should NOT be connected to a power source.

See the [Power](#) section in [Hardware design with RP2040](#) for more information about powering the Raspberry Pi Pico.

4.6.1. Aside: Hands-free Flash Programming

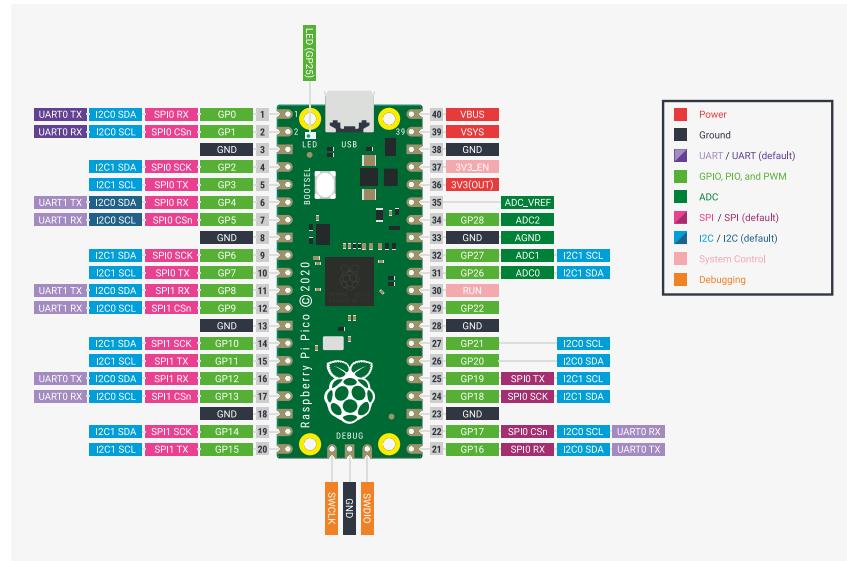
You can use the SWD port ([Chapter 5](#)) to reset the board and run code. This works even if your code has crashed, removing the need to press buttons and unplug. Setting up SWD is a good next step after Hello World.

If you code Pico programs on a Raspberry Pi, you can set up SWD by running the `pico-setup` script ([Chapter 1](#)) and connecting 3 wires from your Pi to the Pico as shown in [Chapter 5](#). You can also use the [Debug Probe](#) to access debug information via the SWD port.

Chapter 5. Flash Programming with SWD

Serial Wire Debug (SWD) is a standard interface on Cortex-M-based microcontrollers, which the machine you are using to develop your code (commonly called the *host*) can use to reset the board, load code into flash, and set the code running. Raspberry Pi Pico exposes the RP2040 SWD interface on three pins at the bottom edge of the board. The host can use the SWD port to access RP2040 internals at any time, so there is no need to manually reset the board or hold the BOOTSEL button.

Figure 6. The SWD port is labelled at the bottom of this Pico pinout diagram. The ground (GND) connection is required to maintain good signal integrity between the host and the Pico. The SWDIO pin carries debug traffic in both directions, between RP2040 and the host. The SWCLK pin keeps the connection well-synchronised. These pins connect to a dedicated SWD interface on RP2040, so you don't need to sacrifice any GPIOs to use the SWD port.



On a Raspberry Pi, you can connect the Pi GPIOs directly to Pico's SWD port, and load code from there. On other machines you will need an extra piece of hardware – a *debug probe* – to bridge a connection on your host machine (like a USB port) to the SWD pins on the Pico. One of the cheapest ways to do this is to use another Pico as the debug probe, and this is covered in [Appendix A](#).

This chapter covers how you can connect your machine to Pico's SWD port, and use this to write programs into flash and run them.

💡 TIP

If you use an IDE like Visual Studio Code ([Chapter 7](#)), this can be configured to use SWD automatically behind the scenes, so you click the play button and the code runs, as though you were running native code on your own machine.

ℹ️ NOTE

You can also use SWD for interactive debugging techniques like setting breakpoints, stepping through code execution line-by-line, or even peeking and poking IO registers directly from your machine without writing any RP2040 software. This is covered in [Chapter 6](#).

5.1. Installing OpenOCD

 **TIP**

If you have run the `pico-setup` script on your Raspberry Pi (Chapter 1), OpenOCD is already installed and you can skip to the next section.

To access the SWD port on a microcontroller, you need a program on your host machine called a *debug translator*. Debug translators understand the SWD protocol, and know how to control the processor (two Cortex-M0+s in the case of RP2040) inside the microcontroller. The debug translator also knows how to talk to the specific debug probe that you have connected to the SWD port, and how to program the flash on your device.

We recommend installing a debug translator called OpenOCD. To install OpenOCD, run the following command:

```
$ sudo apt install openocd
```

You can run OpenOCD using the `openocd` command in your terminal.

5.1.1. Installing OpenOCD on macOS

To install OpenOCD on macOS, run the following commands:

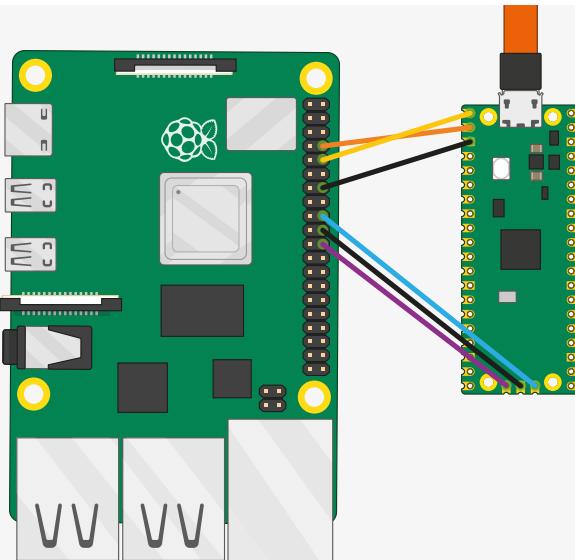
```
$ brew install openocd
```

You can run OpenOCD using the `openocd` command in your terminal.

5.2. SWD Port Wiring

You need to connect wires to the SWD port in order to program and run code on RP2040 via SWD.

Figure 7. A Raspberry Pi 4 and the Pico with UART and SWD port connected together. Both are jumpered directly back to the Raspberry Pi 4 without using a breadboard. Only the lower three wires in this diagram are needed for SWD access; optionally you can also connect the Pi UART, as shown by the upper 3 wires, to directly access the Pico's serial port.



The default configuration is to have SWDIO on Pi GPIO 24, and SWCLK on GPIO 25 – and can be wired to a Pico with the following mapping,

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

as seen in [Figure 7](#).

 **TIP**

When using [Pico as a debug probe](#), connect the [GND](#), [SWCLK](#) and [SWDIO](#) pins on the debugging board to the matching pins on the board under test.

If possible you should wire the SWD port directly to the Raspberry Pi as signal integrity is important; wiring the SWD port via a breadboard or other indirect methods may reduce the signal integrity sufficiently so that loading code over the connection is erratic or fails completely. It is important to also wire the ground wire (0V) between the two directly and not rely on another ground path.

Note the Raspberry Pi Pico must also be powered (e.g. via USB) in order to debug it! You must build our OpenOCD branch to get working multidrop SWD support.

5.3. Loading a Program

OpenOCD expects program binaries to be in the form of [.elf](#) (executable linkable format) files, not the [.uf2](#) files used by BOOTSEL mode. The SDK builds both types of file by default, but it's important not to mix them up.

Assuming you have already built the [blink](#) example, using the instructions in [Chapter 3](#), you can run the following command to program the resulting [.elf](#) file over SWD, and run it:

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

These arguments modify the [openocd](#) command in the following ways:

-f interface/raspberrypi-swd.cfg Tells OpenOCD to use Raspberry Pi's GPIO pins to access the SWD port. To use an external USB→SWD probe such as the [Debug Probe](#), specify a different interface.

-f target/rp2040.cfg Tells OpenOCD we are connecting to a RP2040-based board. This [.cfg](#) file contains information for OpenOCD like the type of processor (Cortex-M0+) and how it should access the flash memory.

-c Passes a series of *commands* to OpenOCD directly from the command line. We passed the following commands:

program blink/blink.elf Tells OpenOCD to write our [.elf](#) file into flash, erasing the target region of flash first if necessary. The [.elf](#) file contains all the information telling OpenOCD where different parts of it must be loaded, and how big those parts are.

verify Tells OpenOCD to read back from the flash after programming, to check that the programming was successful.

reset Puts the RP2040 into a clean initial state, as though it just powered up, so that it is ready to run our code.

`exit`

Disconnects from the RP2040 and exits. The new program will launch when OpenOCD disconnects.

NOTE

To run this command on a Raspberry Pi 5, create a file named `raspberrypi5-swd.cfg`. Copy and paste the following instructions into the file:

```
adapter driver linuxgpiod
adapter gpio swclk 25 -chip 4
adapter gpio swdio 24 -chip 4
```

In the above `openocd` command, replace `interface/raspberrypi-swd.cfg` with `raspberrypi5-swd.cfg` to use the Pi 5-specific configuration file.

TIP

The error `Info: DAP init failed` indicates that OpenOCD could not see an RP2040 on the SWD interface used. This can happen for the following reasons:

- the board is not powered
- the SWD wiring is not correct (e.g. the ground wire is disconnected, or SWDIO and SWCLK have been swapped)
- long or loose jumper wires have damaged signal integrity

To check that you really have loaded a new program, you can modify `blink/blink.c` to flash the LED more quickly using the following code:

```
int main() {
    const uint LED_PIN = 25;
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (true) {
        gpio_put(LED_PIN, 1);
        // Blink faster! (this is the only line that's modified)
        sleep_ms(25);
        gpio_put(LED_PIN, 0);
        sleep_ms(250);
    }
}
```

Then rebuild and rerun the above `openocd` command to load the new program:

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

If the LED blinks faster than before, you have successfully loaded a new program.

 **TIP**

You can reset Pico from the command line with openocd with the following command:

```
openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg -c "init; reset; exit"
```

Chapter 6. Debugging with SWD

As well as resetting the board, loading and running code, the SWD port on RP2040-based boards like Raspberry Pi Pico can be used to interactively debug a program you have loaded. This includes things like:

- Setting breakpoints in your code
- Stepping through execution line by line
- Inspecting the values of variables at different points in the program

[Chapter 5](#) showed how to install OpenOCD to access the SWD port on your Raspberry Pi Pico. To debug code interactively, we also need a *debugger*, such as the ubiquitous GNU Debugger, GDB.

Note that by default the SDK builds highly optimised program binaries, which can look very different in terms of control flow and dataflow from the original program you wrote. This can be confusing when you try and step through the code interactively, so it's often helpful to create a *debug build* of your program which is less aggressively optimised, so that the real on-device control flow is a closer match to your source code.

6.1. Build "Hello World" debug version

⚠️ WARNING

When using SWD for debugging you need to use a UART based serial connection (see [Chapter 4](#)) as the USB stack will be paused when the RP2040 cores are stopped during debugging, which will cause any attached USB devices to disconnect. You cannot use a USB CDC serial connection during debugging.

You can build a debug version of the "Hello World" with `CMAKE_BUILD_TYPE=Debug` as shown below,

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

6.2. Installing GDB

Install `gdb-multiarch`,

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

NOTE

On Intel Macs, you can install `gdb` using Homebrew:

```
$ brew install gdb
```

Despite the warning after installation, you do not have to codesign the binary to use it for remote debugging.

6.3. Use GDB and OpenOCD to debug Hello World

Ensuring your Raspberry Pi 4 and Raspberry Pi Pico are correctly wired together, we can attach OpenOCD to the chip, via the `raspberrypi-swd` interface.

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

Your output should look like this:

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

WARNING

If you see an error like `Info : DAP init failed` then your Raspberry Pi Pico is either powered off, wired incorrectly, or has signal integrity issues. Try different GPIO jumper cables.

This OpenOCD terminal needs to be left open. So go ahead and open another terminal, in this one we'll attach a `gdb` instance to OpenOCD. Navigate to the "Hello World" example code, and start `gdb` from the command line.

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

Connect GDB to OpenOCD,

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

TIP

You can create a `.gdbinit` file so you don't have to type `target remote localhost:3333` every time. Do this with `echo "target remote localhost:3333" > ~/.gdbinit`. However, this interferes with debugging in VSCode (Chapter 7).

and load `hello_serial.elf` into flash,

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

and then start it running.

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

❗️ IMPORTANT

If you see errors similar to [Error finishing flash operation](#) or [Error erasing flash with vFlashErase packet](#) in GDB when attempting to load the binary onto the Raspberry Pi Pico via OpenOCD then there is likely poor signal integrity between the Raspberry Pi and the Raspberry Pi Pico. If you are not directly connecting the SWD connection between the two boards, see [Figure 7](#), you should try to do that. Alternatively you can try reducing the value of `adapter_khz` in the `raspberrypi-swd.cfg` configuration file, trying halving it until you see a successful connection between the boards. As we're bitbanging between the boards timing is marginal, so poor signal integrity may cause errors.

Or if you want to set a breakpoint at `main()` before running the executable,

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

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

before continuing after you have hit the breakpoint,

```
(gdb) continue
```

To quit from `gdb` type,

```
(gdb) quit
```

More information about how to use `gdb` can be found at <https://www.gnu.org/software/gdb/documentation/>.

Chapter 7. Using Visual Studio Code

Visual Studio Code (VS Code) is a popular open source editor developed by Microsoft. We recommend an Integrated Development Environment (IDE) for users who want a graphical interface for code editing and debugging.

7.1. Installing Visual Studio Code

Before you use VS Code to build software for your Pico, you must complete the following steps:

- [download and install the command line toolchain](#)
- [connect SWD to your board via OpenOCD](#)
- [set up GDB for command-line debugging](#)

To install Visual Studio Code (VS Code) on Raspberry Pi OS, 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
```

7.2. Raspberry Pi Pico VS Code Extension

7.2.1. Install the 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 Pico and Pico W.

7.2.1.1. Prerequisites

To use the Raspberry Pi Pico VS Code extension, you need:

- Visual Studio Code v1.87.0 or later

Additional requirements depend upon your operating system.

7.2.1.1.1. 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)

7.2.1.1.2. Raspberry Pi OS

To install all requirements for the extension on Raspberry Pi OS, run the following command:

```
$ sudo apt install openocd ninja-build
```

7.2.1.1.3. 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 and Clang)
- Ninja (only for the arm64 architecture)

7.2.1.2. Install

Use the following command to install the extension:

```
$ code --install-extension raspberry-pi.raspberry-pi-pico
```

You can also 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.

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>.

To install from a GitHub release, download the **vsix** file and run the following command, replacing the **<version>** placeholder with the version number in your **vsix** file:

```
$ code --install-extension raspberry-pi-pico-<version>.vsix
```

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".

7.2.1.3. Configure

7.2.1.3.1. Prevent GitHub requests from timing out

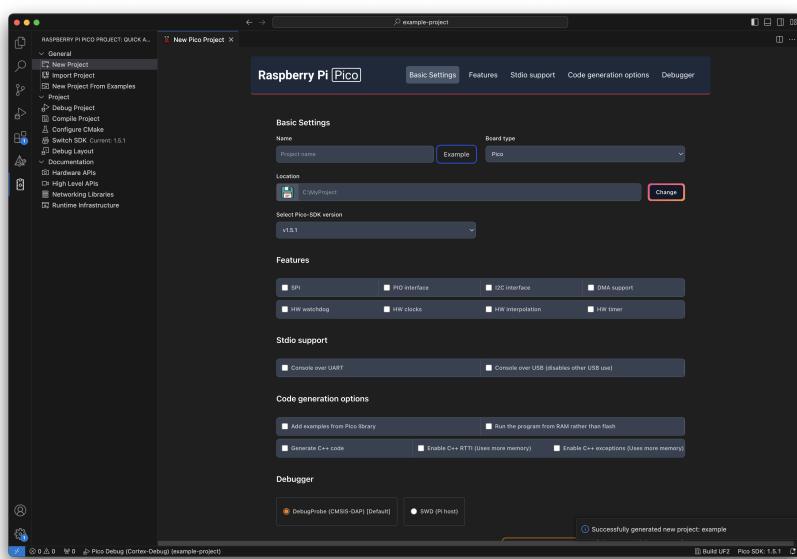
The extension uses the public GitHub API to retrieve available Pico SDK versions and other data. This API has IP address rate limits that can be triggered by large amounts of human activity. You'll know that you've triggered rate limiting if you see an error bearing the message "Error while retrieving SDK and toolchain versions". If you trigger GitHub's API rate limiting, consider configuring a personal GitHub token for the extension. The extension only uses the token to query the GitHub API for Pico SDK versions. The token allows significantly more activity before triggering rate limiting.

To configure an access token, sign into GitHub and access the **Settings** menu for your profile at <https://github.com/settings/tokens>. Click the **Generate new token** button and select **Generate new token (classic)** from the menu to create a new personal access token. Enter a note, select an expiration period, and ensure that you grant the token **public_repo** access. No other permissions are required. Click **Generate token** to create the token. Leave the web page open while you navigate to the settings in VS Code.

In the main menu of VS Code, navigate to **Settings**. In the **Extensions** list, navigate to **Raspberry Pi Pico Extension**. Copy your newly-generated token from the GitHub web page into the **Github token** field. Close the settings tab to automatically save the configuration.

7.2.2. Create a project

Figure 8. Configuring a new project using the VS Code extension.



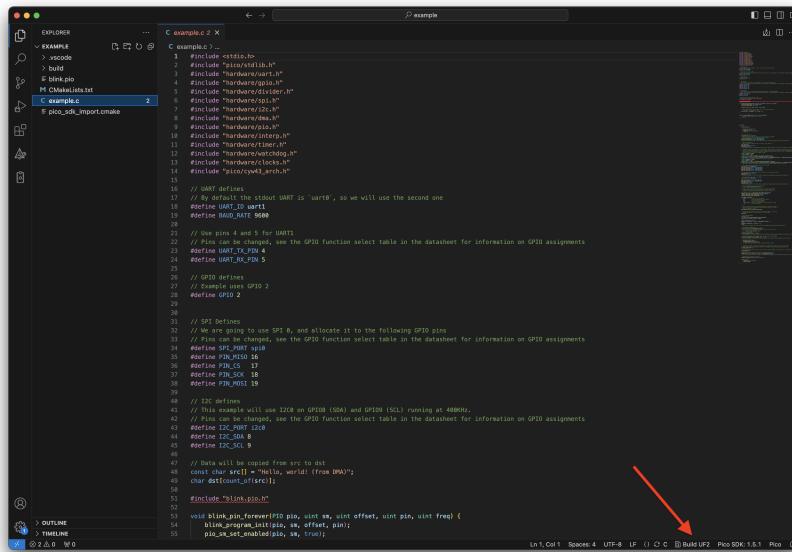
To create a project, complete the following steps:

1. In the Activity sidebar (by default, on the left side of VS Code), click the **Raspberry Pi Pico icon**, labelled "Raspberry Pi Pico Project".
2. Select the **New project** option.
3. Specify a name.
4. Specify a board type (**Pico** or **Pico W**)
5. Specify a folder where the extension can generate files. You should probably create a new folder for your project, since the extension uses the selected location as the project root.
6. Select a Pico SDK version from the menu. In most cases, you should use the latest version.

7. Specify the features you would like automatically generate code for in your generated project.
8. If you would like to enable `stdin` and `stdout` over UART or USB, enable those.
9. If you would like to enable networking support over wireless over `lwIP`, enable your preferred option.
10. Choose which code generation options you would like to enable, including example code snippets, C sample code generation, C exceptions, C++ Runtime Type Information (RTTI), and in-memory (instead of flash) execution.
11. Choose your **preferred debugger option**: Debug Probe hardware or an additional Pico running `debugprobe` software.
12. To specify the versions of Ninja, CMake, Python, and the ARM Embedded Toolchain, click the **Show Advanced options** button. Those options will appear next to the Pico SDK version selector.
13. Click the **Create** button to generate the project using your specified options. A new VS Code window will appear displaying the root of the newly-generated project.

7.2.3. Build

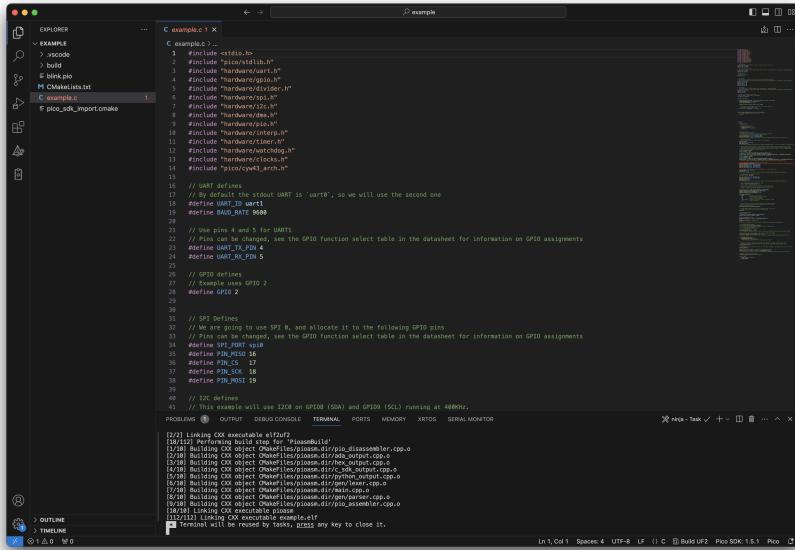
Figure 9. To build a project, use the VS Code extension 'Build UF2' button in the status bar.



With the VS Code extension, you can build an executable you can run on your Pico or Pico W.

Click the **Build UF2** option on the bottom right of the status bar (by default, located at the bottom of your window) to compile your program into a UF2 binary that you can load onto your Pico.

Figure 10. A successful project build using the VS Code extension.



A successful build generates a UF2 file named `<project name>.uf2` in the `build` directory and prints output similar to the following:

```
* Executing task: /Users/example/.pico-sdk/ninja/v1.11.1/ninja -C /Users/example/Documents/test/myproject/example/build

ninja: Entering directory '/Users/example/Documents/test/myproject/example/build'
[1/8] Performing build step for 'ELF2UF2Build'
ninja: no work to do.
[2/8] Performing build step for 'PioasmBuild'
ninja: no work to do.
[8/8] Linking CXX executable example.elf
* Terminal will be reused by tasks, press any key to close it.
```

7.2.4. Run

To run a program on your Pico:

1. Generate a UF2 file using the build instructions above.
2. Plug in your Pico while holding down the **BOOTSEL** button to enter bootloader mode. This makes the Pico available for flashing as a drive with the name "RPI-RP2".
3. Copy and paste (or drag and drop) the UF2 file from your computer to the "RPI-RP2" drive. You can find the file at `/build/<project name>.uf2` in the project you generated with the VS Code extension.
4. Your Pico will reboot and begin executing the program.

7.2.5. Debug

Debugging with the VS Code extension works the same as usual. Be sure to select the correct debugger device (Debug Probe hardware or an additional Pico running `debugprobe` software) when you generate your project.

For more information about debugging, see [Appendix A](#).

7.3. Configure Manually

Run the following commands to install the extensions needed to debug a Pico or Pico W:

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

Configure the `PICO_SDK_PATH` environment variable so that VS Code can find the SDK:

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

If `PICO_SDK_PATH` is not set by default in your shell environment, you must set it each time you start VS Code. To set `PICO_SDK_PATH` automatically, add the following lines to your `.profile`, `.bashrc`, or `.zprofile`:

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

Run the following command to start Visual Studio Code from a Terminal window:

```
$ code
```

 **TIP**

You can configure Intellisense for CMake by changing the provider by toggling: **View** → **Command Palette** → **C/C++: Change Configuration Provider...** → **CMake Tools**.

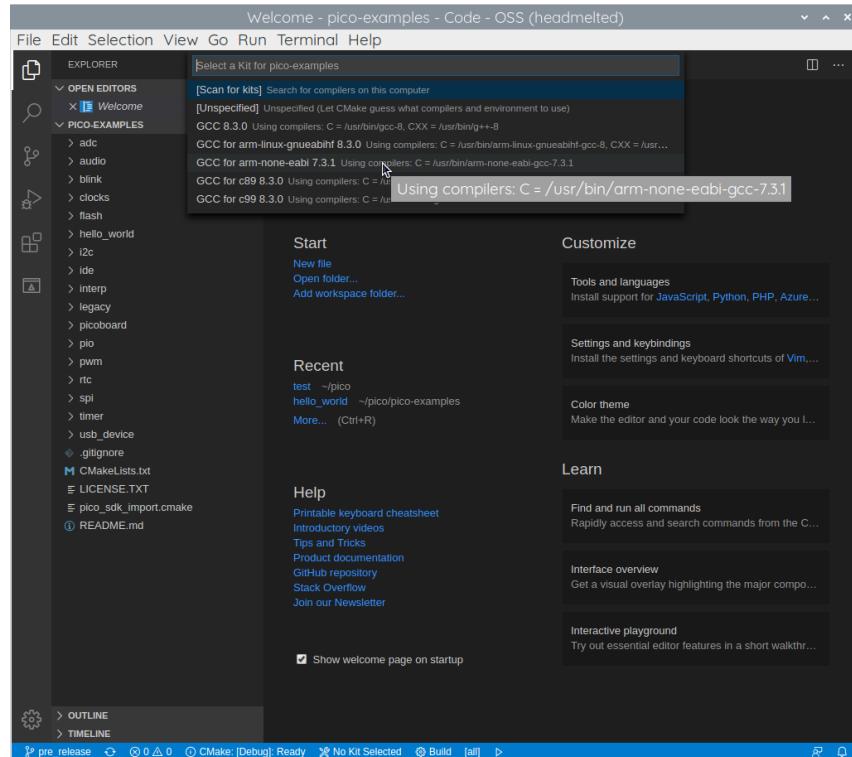
7.4. Loading a Project

Open the `pico-examples` folder by selecting the **Explorer** toolbar (**Ctrl + Shift + E**), selecting **Open Folder**, and navigating to `/home/pi/pico/pico-examples` in the file browser dialog. Click **OK** to open the folder in VS Code.

As long as the **CMake Tools** extension is installed, after a second or so you should see an alert in the lower right-hand corner of the VS Code window.

Hit **Yes** to configure the project. You will then be prompted to choose a compiler, see [Figure 11](#):

Figure 11. Prompt to choose the correct compiler for the project.



Select **GCC for arm-none-eabi** from the menu.

💡 TIP

The alert closes after a few seconds. If you miss it, configure the compiler by selecting **No Kit Selected** in the blue bottom bar of the VS Code window.

To build, use one of the following methods:

- Click the **Build** button in the blue bottom bar to build all of the examples in **pico-examples**.
- Click **[all]** in the blue bottom bar to select and build a single project.

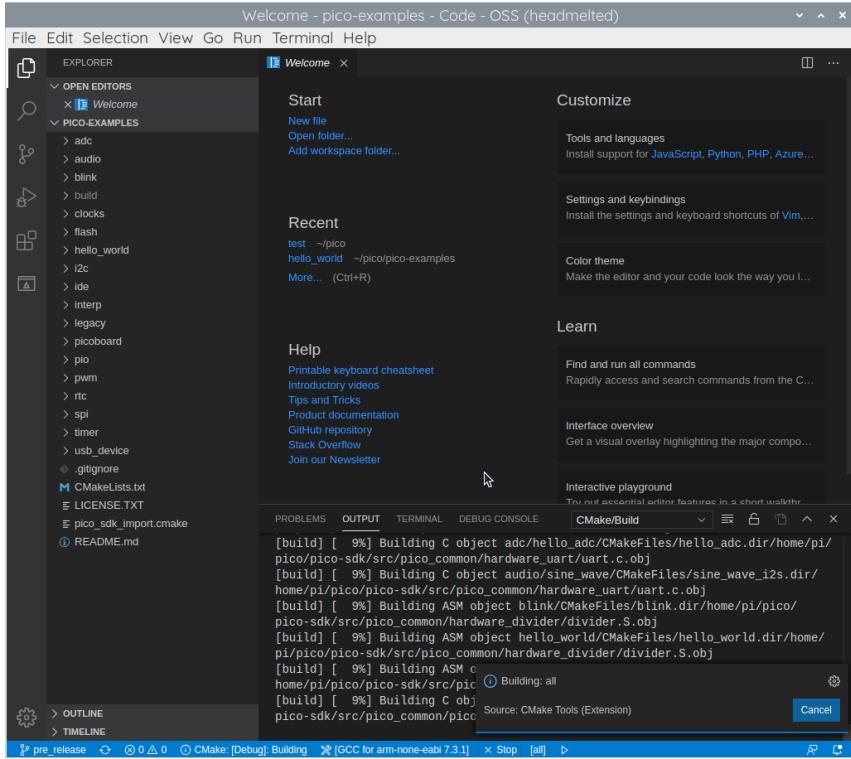
For now, click **[all]** in the blue bottom bar, type in "hello_usb", and select the **Hello USB** executable. This will build only the "Hello USB" example, saving compile time.

💡 TIP

You can toggle between "Debug" and "Release" builds by clicking **CMake: [Debug]: Ready** in the blue bottom bar. By default, this configuration builds a debug-enabled executable ready for SWD debugging.

Click the **Build** button (the cog wheel) in the blue bottom bar. This creates the build directory and runs CMake as we did by hand in [Section 3.1](#). Before starting the build itself, see [Figure 12](#):

Figure 12. Building the `pico-examples` project in Visual Studio Code



As we did from the command line previously, amongst other targets, we have now built:

- `hello_usb.elf`, which is used by the debugger
- `hello_usb.uf2`, which can be dragged onto the RP2040 USB Mass Storage Device

7.5. Debugging a Project

The `pico-examples` repo contains an example debug configuration that starts OpenOCD, attaches GDB, and launches an application. Copy `launch-raspberrypi-swd.json` into the `pico-examples/.vscode` directory and rename it `launch.json`. You should also copy the provided `settings.json` file, which removes some potentially confusing options from the CMake plugin, including broken **Debug** and **Run** buttons that attempt to run a Pico binary on the host.

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

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

```
1 {
2     "version": "0.2.0",
3     "configurations": [
4         {
5             "name": "Pico Debug",
6             "cwd": "${workspaceRoot}",
7             "executable": "${command:cmake.launchTargetPath}",
8             "request": "launch",
9             "type": "cortex-debug",
10            "serverType": "openocd",
11            // This may need to be "arm-none-eabi-gdb" for some previous builds
```

```

12         "gdbPath" : "gdb-multiarch",
13         "device": "RP2040",
14         "configFiles": [
15             "interface/raspberrypi-swd.cfg",
16             "target/rp2040.cfg"
17         ],
18         "svdFile": "${env:PICO_SDK_PATH}/src/rp2040/hardware_regs/rp2040.svd",
19         "runToEntryPoint": "main",
20         // Work around for stopping at main on restart
21         "postRestartCommands": [
22             "break main",
23             "continue"
24         ]
25     }
26 ]
27 }
```

ⓘ NOTE

If your operating system uses a distribution of GDB called `arm-none-eabi-gdb` (instead of `gdb-multiarch`), replace the GDB name in `launch.json`'s `'gdbPath'` setting.

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

```

1 {
2     // These settings tweaks to the cmake plugin will ensure
3     // that you debug using cortex-debug instead of trying to launch
4     // a Pico binary on the host
5     "cmake.statusbar.advanced": {
6         "debug": {
7             "visibility": "hidden"
8         },
9         "launch": {
10            "visibility": "hidden"
11        },
12        "build": {
13            "visibility": "hidden"
14        },
15        "buildTarget": {
16            "visibility": "hidden"
17        }
18    },
19    "cmake.buildBeforeRun": true,
20    "C_Cpp.default.configurationProvider": "ms-vscode.cmake-tools"
21 }
```

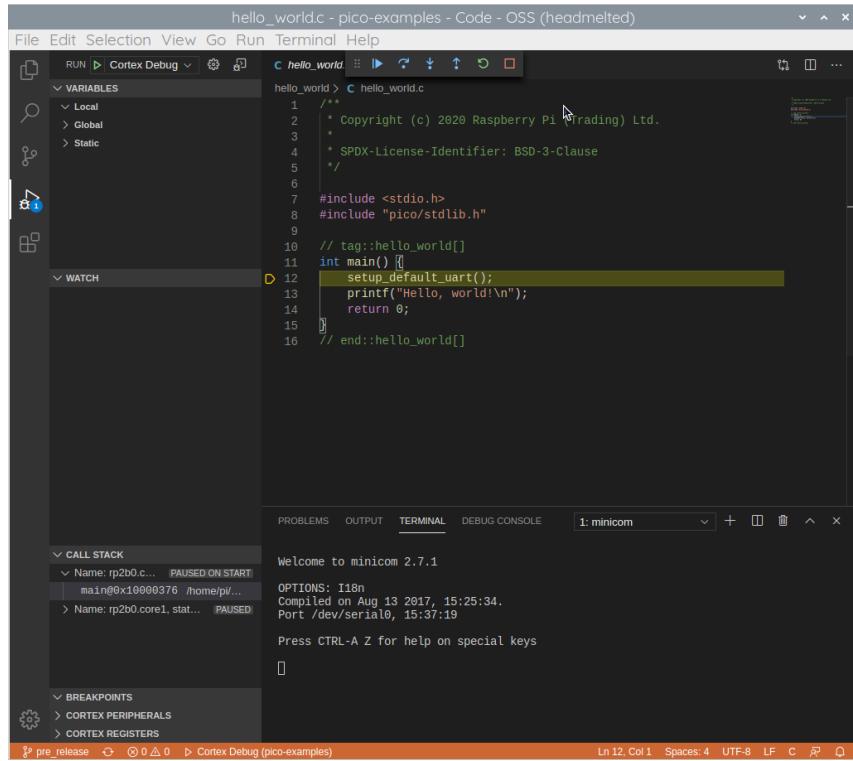
7.5.1. Running "Hello USB" on the Pico

❗ IMPORTANT

Ensure that you build the "Hello USB" example as a Debug binary (`CMAKE_BUILD_TYPE=Debug`).

To load your code on the Pico and start debugging, click the **Debug** toolbar (`Ctrl + Shift + D`) and click the small green arrow (play button) at the top of the left-hand window pane.

Figure 13. Debugging the "Hello USB" binary inside Visual Studio Code



This will load your code onto your Pico. You should see the source code for "Hello USB" in the main right-hand (upper) pane of the window. Because of the `runToMain` directive in the `launch.json` file, the code will start to run, proceeding to the first breakpoint. Click on the small blue arrow (play button) at the top of this main source code window to Continue (`F5`) and resume running the code.

TIP

If you switch to the "Terminal" tab in the bottom right pane (below the `hello_usb.c` code) you can use the following line at the terminal prompt (see [Section 4.4](#)) to open `minicom` inside VSCode:

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

This displays the UART output from the "Hello USB" example.

Chapter 8. Creating your own Project

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

```
$ 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/stl.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, 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_stdlb)

```

1. This will enable serial output via USB.
2. This will enable 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

```

➊ 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 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

File extension	Description
.uf2	The program code and data in a UF2 form that you can drag-and-drop on to the RP2040 board 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 file format, developed by Microsoft, that is used for flashing the RP2040 board over USB. More details can be found on the [Microsoft UF2 Specification Repo](#)

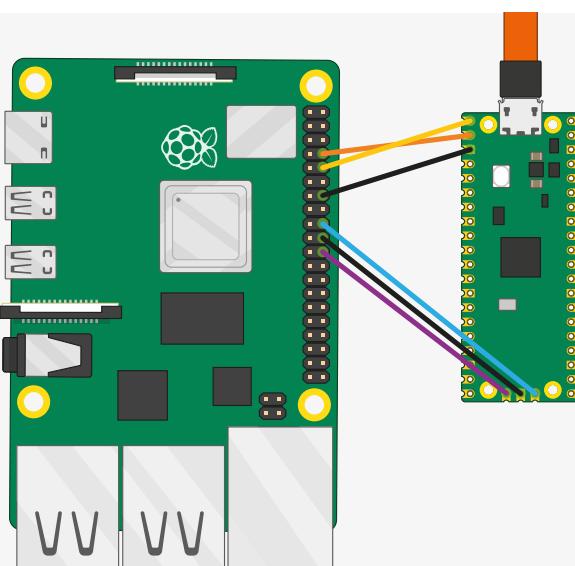
NOTE

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 RP2040 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.

8.1. 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 [Section 6.3](#). Connect your Raspberry Pi and the Raspberry Pi Pico as in [Figure 14](#).

Figure 14. A Raspberry Pi 4 and the Raspberry Pi Pico with UART and SWD debug port connected together. Both are jumpered directly back to the Raspberry Pi 4 without using a breadboard.



Then go ahead and build a debug version of your project using `CMAKE_BUILD_TYPE=Debug` as below,

```
$ cd ~/pico/test
$ rmdir build
$ mkdir build
$ cd build
```

```
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ 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 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"`.

Then open up a terminal window and attach OpenOCD using the `raspberrypi-swd` interface.

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

This OpenOCD terminal needs to be left open. So go ahead and open another terminal window and start `gdb-multiarch` using

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

Connect GDB to OpenOCD, and load the `test.elf` binary into flash,

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

and then start it running,

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

8.2. Working in Visual Studio Code

If you want to work in Visual Studio Code rather than from the command line you can do that, see [Chapter 7](#) for instructions on how to configure the environment and load your new project into the development environment to let you write and build code.

If you want to also use Visual Studio Code to debug and load your code onto the Raspberry Pi Pico you'll need to create a `launch.json` file for your project. The example `launch-raspberrypi-swd.json` in [Chapter 7](#) should work. You need to copy it into your project directory as `.vscode/launch.json`.

8.3. Automating project creation

The pico project generator, automatically creates a "stub" project with all the necessary files to allow it to build. If you want to make use of this you'll need to go ahead and clone the project creation script from its Git repository,

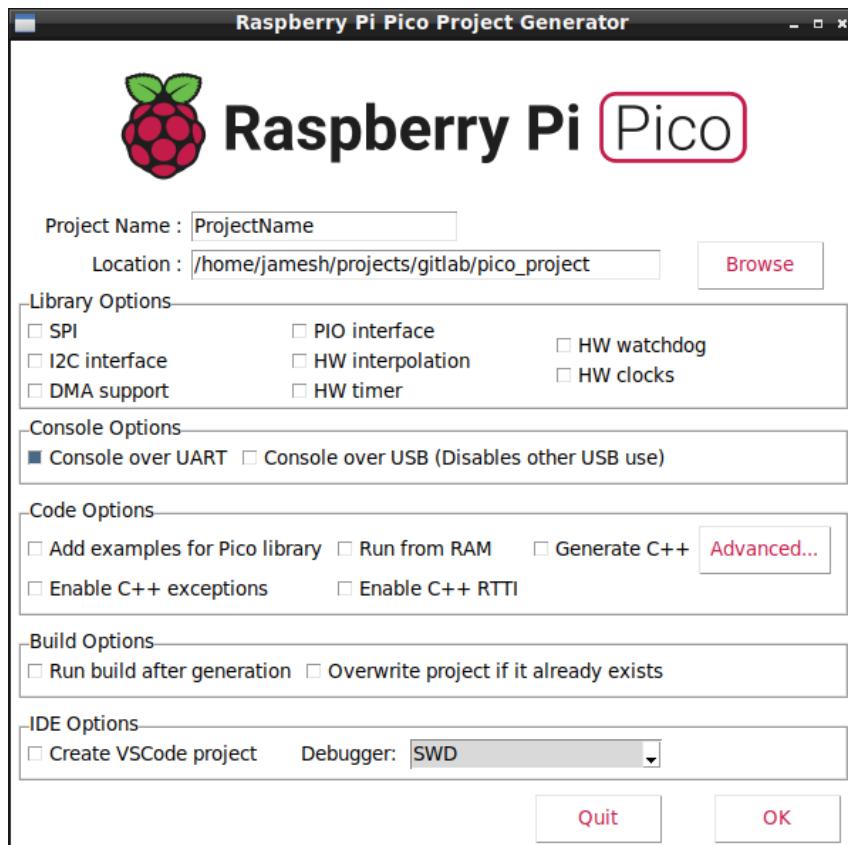
```
$ git clone https://github.com/raspberrypi/pico-project-generator.git --branch master
```

It can then be run in graphical mode,

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

which will bring up a GUI interface allowing you to configure your project, see [Figure 15](#).

Figure 15. Creating a RP2040 project using the graphical project creation tool.



You can add specific features to your project by selecting them from the check boxes on the GUI. This will ensure the build system adds the appropriate code to the build, and also adds simple example code to the project showing how to use the feature.

There are a number of options available, which provide the following functionality.

Console Options	Description
Console over UART	Enable a serial console over the UART. This is the default.
Console over USB	Enable a console over the USB. The device will act as a USB serial port. This can be used in addition to or instead of the UART option, but note that when enabled other USB functionality is not possible.

Code Options	Description
Add examples for Pico library	Example code will be generated for some of the standard library features that by default are in the build, for example, UART support and HW dividers.

Code Options	Description
Run from RAM	Usually, the build creates a binary that will be installed to the flash memory. This forces the binary to work directly from RAM
Generate C++	The source files generated will be C++ compatible.
Enable C++ exceptions	Enable C++ exceptions. Normally disabled to save code space.
Enable C++ RTTI	Enable C++ Run Time Type Information. Normally disabled to save code space.
Advanced	Brings up a table allowing selection of specific board build options. These options alter the way the features work, and should be used with caution.

Build Options	Description
Run Build	Once the project has been created, build it. This will produce files ready for download to the Raspberry Pi Pico.
Overwrite Project	If a project already exists in the specified folder, overwrite it with the new project. This will overwrite any changes you may have made.

IDE Options	Description
Create VSCode Project	As well as the CMake files, also create the appropriate Visual Studio Code project files.
Debugger	Select which Pico Debugger the VSCode debugging system will use. Defaults to Serial Wire Debug.

8.3.1. Project generation from the command line

The script also provides the ability to create a project from the command line, e.g.

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

The `--feature` options add the appropriate library code to the build, and also example code to show basic usage of the feature. You can add multiple features, up to the memory limitation of the RP2040. You can use the `--list` option of the script to list all the available features. The example above adds support for the I2C and SPI interfaces.

Here passing the `--project vscode` option will mean that `.vscode/launch.json`, `.vscode/c_cpp_properties.json` and `.vscode/settings.json` files are also created, in addition to the CMake project files.

Once created you can build the project in the normal way from the command line,

```
$ cd test/build
$ 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 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"`.

or from Visual Studio Code.

You can use the `--help` option to give a list of command line arguments, these will also be applied when using the graphical mode.

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 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.

Chapter 9. Building on other platforms

While the main supported platform for developing for the RP2040 is the Raspberry Pi, support for other platforms, such as Apple macOS and Microsoft Windows, is available.

9.1. Building on Apple macOS

Using macOS to build code for RP2040 is very similar to Linux.

9.1.1. Installing the Toolchain

Installation depends on [Homebrew](#). If you don't have it installed, install it:

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

Then, install the toolchain:

```
$ brew install cmake
$ brew tap ArmMbed/homebrew-formulae
$ brew install gcc-arm-embedded
```

Now, you can follow the instructions to build code for the RP2040. Once you've installed the toolchain, developing on macOS and Linux behave the same.

See [Section 2.1](#) and follow the instructions from there – skipping the section where you install the toolchain – to fetch the SDK and build the "Blink" example.

9.1.2. Using Visual Studio Code

Visual Studio Code (VSCode) is a cross platform environment that runs on macOS, Linux, and Microsoft Windows. [Download the macOS version](#) and install it into to your Applications folder.

To start Visual Studio Code, click the VSCode icon in your Applications folder.

9.1.3. Building with CMake Tools

After starting Visual Studio Code, install the [CMake Tools](#) extension. Click on the Extensions icon in the left-hand toolbar (or type **Cmd + Shift + X**) and search for "CMake Tools". Click the "CMake Tools" entry in the list, then click the install button.

We now need to set the `PICO_SDK_PATH` environment variable. Navigate to the `pico-examples` directory and create a `.vscode` directory and add a file called `settings.json` to tell CMake Tools to location of the SDK. Additionally point Visual Studio at the CMake Tools extension.

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

Now click on the Cog Wheel at the bottom of the navigation bar on the left-hand side of the interface and select **Settings**. Then, in the Settings pane, click on **Extensions > CMake Tools**. Then scroll down to **Cmake: Generator** and enter "Unix Makefiles" into the box.

NOTE

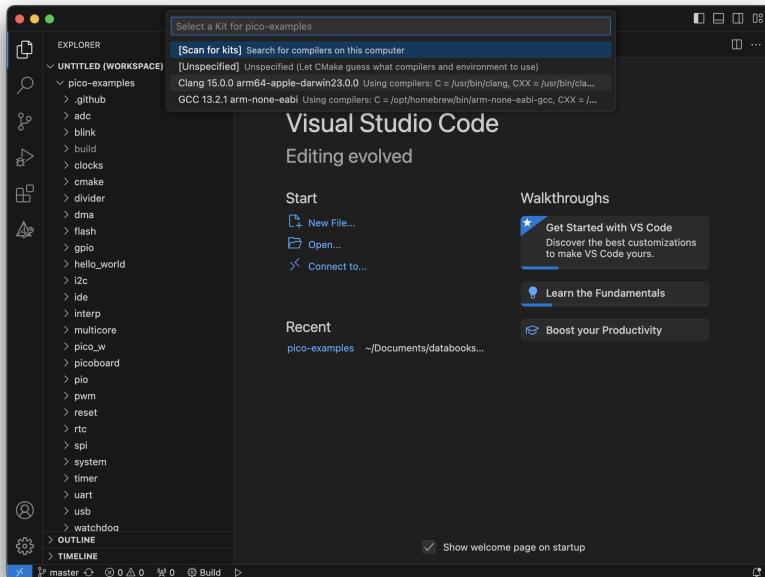
Depending on your local setup you may not need to set the CMake generator manually to "Unix Makefiles". However if you do not do so in some cases Visual Studio will default to `ninja` rather than `make` and the build might fail as `GCC` outputs dependency-information in a slightly-incorrect format that `ninja` can't understand.

If you do find yourself having to configure this variable manually, you may need to point Visual Studio at the CMake Tools extension explicitly by adding the following additional `configurationProvider` line to your `settings.json` file:

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

Then, go to the **File** menu and click on **Add Folder to Workspace**.... Navigate to and select the `pico-examples` repo. The project will load and you'll (probably) be prompted to choose a compiler, see [Figure 16](#). Select **Clang...** as your compiler.

Figure 16. Prompt to choose a compiler for your project.



Finally, click on the **Build** button at the bottom of the window. This will create the build directory and run CMake as we did by hand in [Section 3.1](#), before starting the build itself, see [Figure 12](#).

This will produce `elf`, `bin`, and `uf2` targets, you can find these in the `hello_world/serial` and `hello_world/usb` directories inside the newly created `build` directory. The UF2 binaries can be dragged and dropped directly onto a RP2040 board attached to your computer using USB.

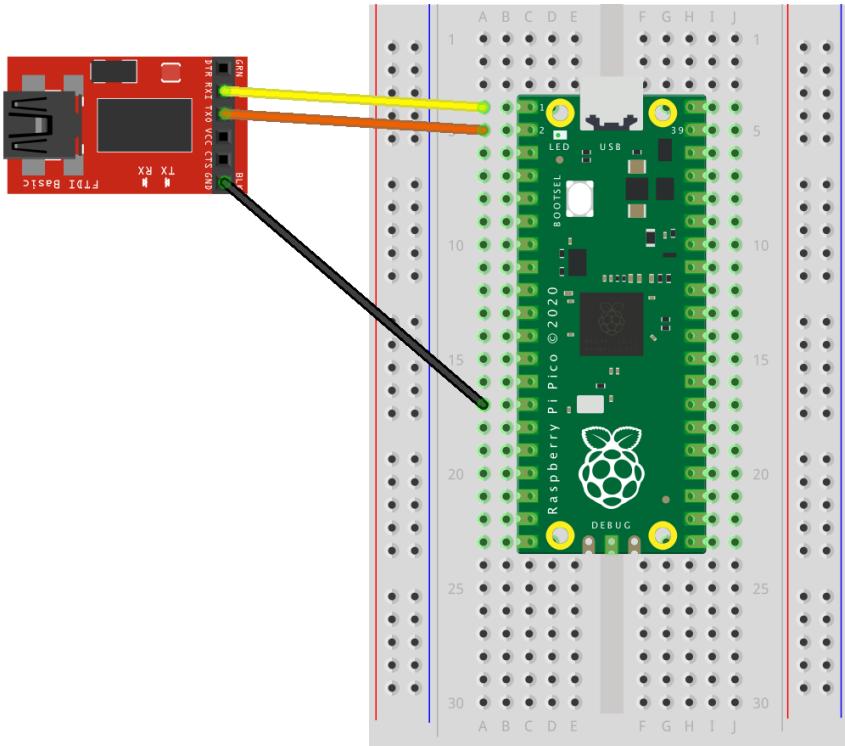
9.1.4. Saying "Hello World"

As we did previously in [Chapter 4](#), you can build the Hello World example with `stdio` routed either to USB CDC (Serial) or to UART0 on pins GP0 and GP1. No driver installation is necessary if you're building with USB CDC as the target output, as it's a class-compliant device. Use a Terminal program, such as [Serial](#), to connect to the USB serial port.

9.1.4.1. UART output

Alternatively, if you want to connect to the Raspberry Pi Pico standard UART to see the output you will need to connect your Raspberry Pi Pico to your Mac using a USB to UART Serial converter, for example a [SparkFun FTDI Basic](#) board. For more information, see [Figure 17](#).

Figure 17. Sparkfun FTDI Basic adaptor connected to the Raspberry Pi Pico



So long as you're using a recent version of macOS, the drivers should already be loaded. Otherwise see the manufacturers' website for [FTDI Chip Drivers](#).

Then, use a Terminal program, such as [Serial](#), to connect to the USB serial port.

9.2. Building on MS Windows

Installing the toolchain on Microsoft Windows 10 or Windows 11 is somewhat different to other platforms. However once installed, building code for the RP2040 is somewhat similar.

WARNING

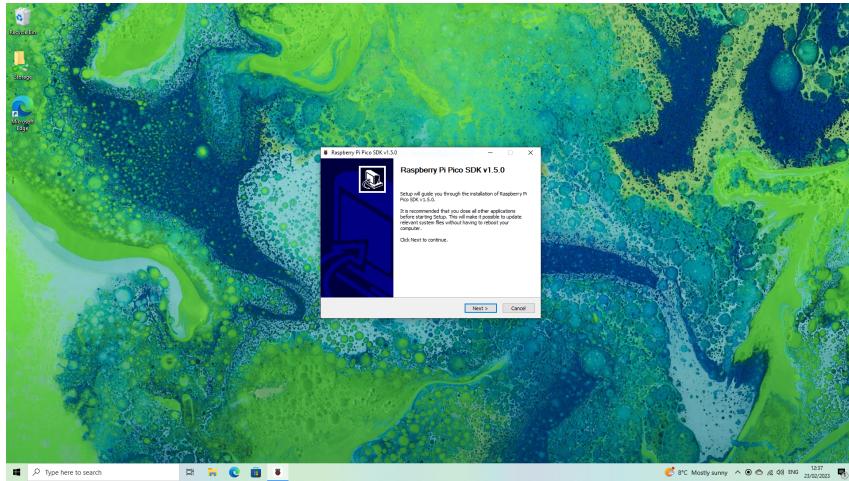
Using Raspberry Pi Pico with Windows 7 or 8 is not officially supported but can be [made to work](#).

9.2.1. Installing the Toolchain

If you are developing for Raspberry Pi Pico on Microsoft Windows, installation of the toolchain can be done using the [Windows Pico Installer](#).

Go ahead and [download the latest release](#) of the installer from Github, <https://github.com/raspberrypi/pico-setup-windows>, and run it. The installation wizard, see [Figure 18](#), will guide you through installing the toolchain.

Figure 18. Installing the Raspberry Pi Pico SDK using the Windows Pico Installer.



After the wizard has completed installation it will prompt you whether it should display a tutorial, and optionally clone and build the [pico-examples](#) repository from Github.

Included software:

- [Arm GNU Toolchain](#)
- [CMake](#)
- [Ninja](#)
- [Python 3.9](#)
- [Git for Windows](#)
- [Visual Studio Code](#)
- [OpenOCD](#)

9.2.1.1. Starting Visual Studio Code

In your Start Menu, look for the *Pico - Visual Studio Code* shortcut, in the *Raspberry Pi Pico SDK <version>* folder. The shortcut sets up the needed environment variables and then launches Visual Studio Code.

NOTE

If you have previously had Visual Studio Code installed we often see problems due to random settings inserted by other extensions, or by the user, in an existing installation. If this is the case for you, please go to the [Pico Installer Wiki](#) for a checklist of known issues and solutions.

9.2.1.2. Opening the examples

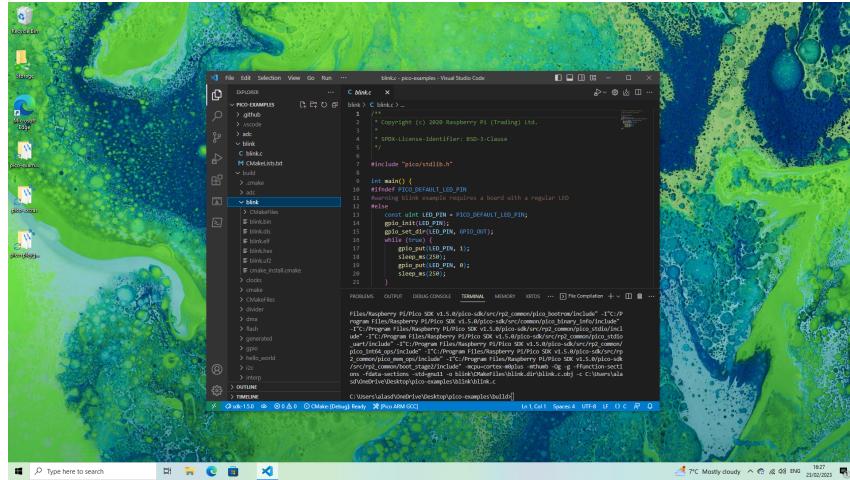
The first time you launch Visual Studio Code using the Start Menu shortcut, it will open the [pico-examples](#) repository.

To re-open the examples repository later, you can open the copy installed at `C:\ProgramData\Raspberry Pi\Pico SDK <version>\pico-examples`.

9.2.1.3. Building the examples

Visual Studio Code will ask if you want to configure the pico-examples project when it is first opened; click *Yes* on that prompt to proceed. (If you miss the prompt look for the 'bell' icon in the bottom right.) You will then be prompted to select a kit – select the *Pico ARM GCC - Pico SDK Toolchain with GCC arm-none-eabi* entry. If the *Pico ARM GCC* entry is not present, select *Unspecified* to have the SDK auto-detect the compiler.

Figure 19. Visual Studio code and the pico-examples repository.



To build one of the examples, click the "CMake" button on the sidebar. You should be presented with a tree view of the example projects; expand the project you'd like to build, and click the small build icon to the right of the target name to build that specific project.

To build everything instead, click the "Build All Projects" button at the top of the CMake Project Outline view.

NOTE

You can find more information about how to use Visual Studio Code in the [Pico Setup for Windows tutorial](#).

IMPORTANT

If you are having problems installing or using the toolchain under Windows there is a [troubleshooting guide](#) available on the Pico Setup for Windows GitHub repository.

9.2.2. Alternative manual installation

WARNING

Manual installation of the toolchain under MS Windows is complicated and not recommended.

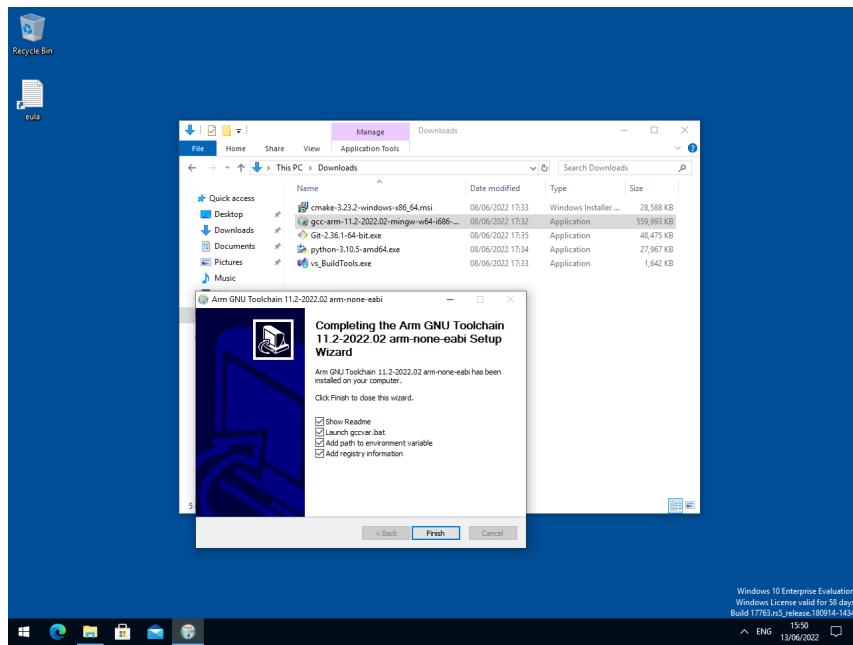
To build you will need to install some extra tools.

- [Arm GNU Toolchain](#) (you need the filename ending with `-arm-none-eabi.exe`)
- [CMake](#)
- [Build Tools for Visual Studio 2022](#)
- [Python 3.10](#)
- [Git](#)

Download the executable installer for each of these from the links above, and then carefully follow the instructions in the following sections to install all five packages on to your Windows computer.

9.2.2.1. Installing Arm GNU Toolchain

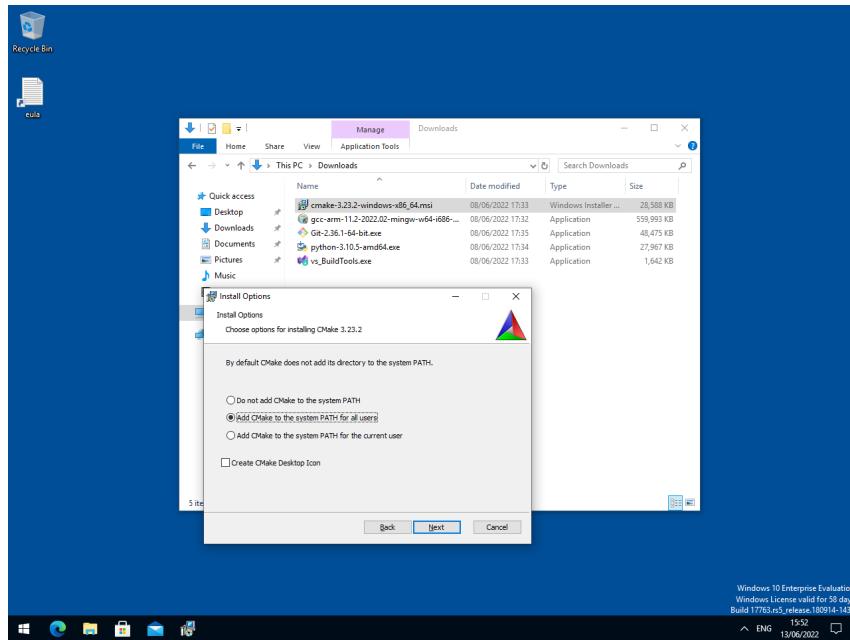
Figure 20. Installing the Arm GNU Toolchain. Ensure that you register the path to the compiler as an environment variable so that it is accessible from the command line.



During installation you should tick the box to register the path to the Arm compiler as an environment variable in the Windows shell when prompted to do so.

9.2.2.2. Installing CMake

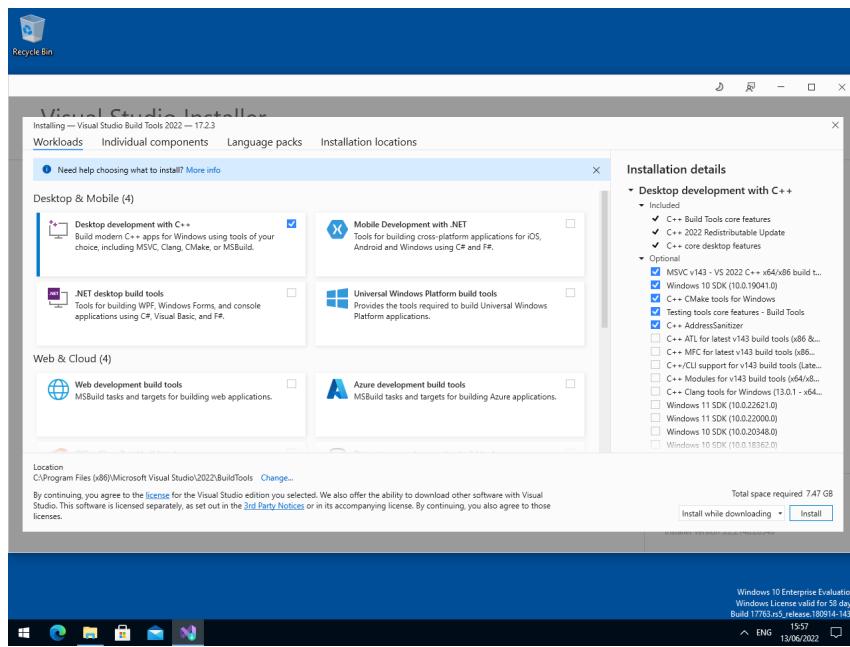
Figure 21. Installing CMake.



During the installation add CMake to the system [PATH](#) for all users when prompted by the installer.

9.2.2.3. Installing Build Tools for Visual Studio 2022

Figure 22. Installing the Build Tools for Visual Studio 2022.



When prompted by the Build Tools for Visual Studio installer you need to install the C++ build tools only.

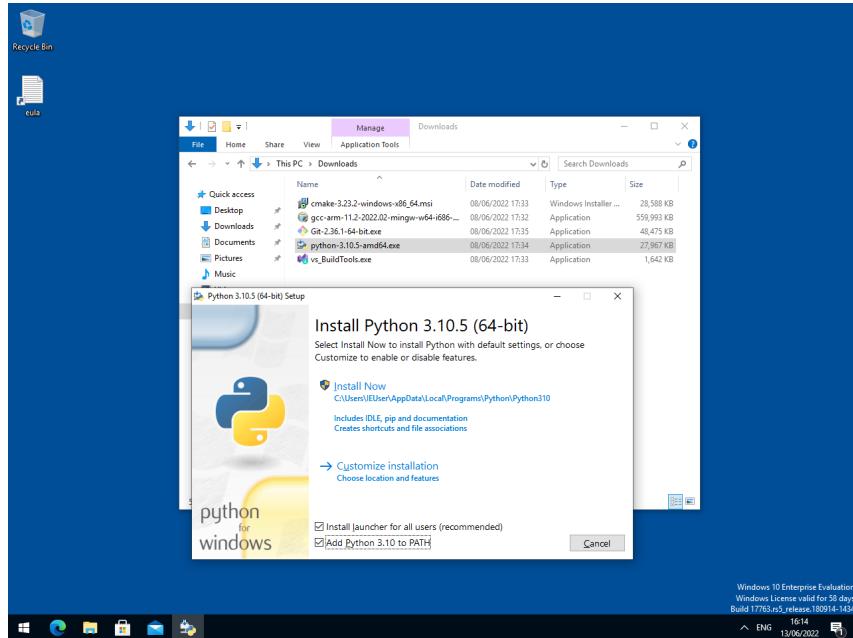
NOTE

You must install the full "Windows 10 SDK" package as the SDK will need to build the `pioasm` and `elf2uf2` tools locally. Removing it from the list of installed items will mean that you will be unable to build Raspberry Pi Pico binaries.

9.2.2.4. Installing Python 3.10

During the installation, ensure that it's installed 'for all users' and add Python 3.10 to the system `PATH` when prompted by the installer. You should additionally disable the `MAX_PATH` length limit when prompted at the end of the Python installation.

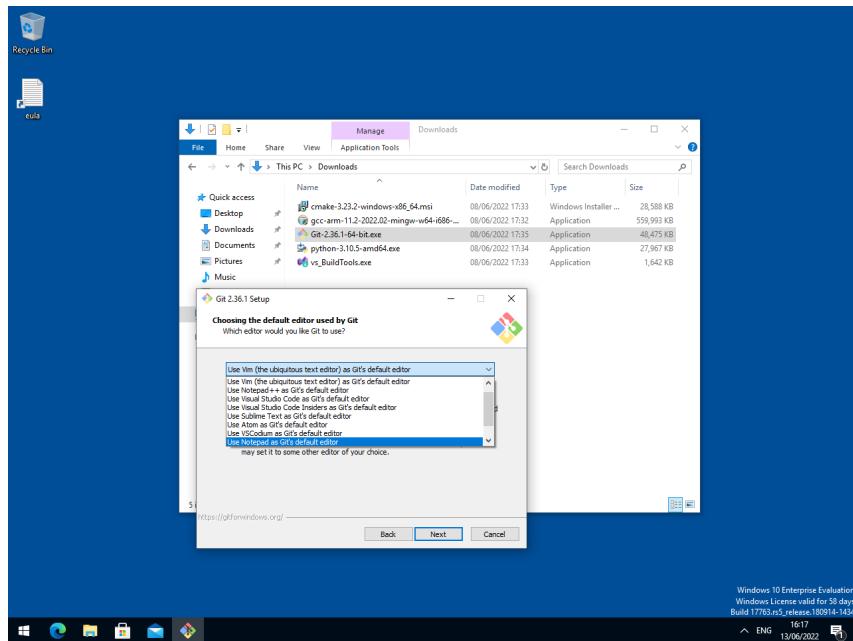
Figure 23. Installing Python 3.10 tick the "Add Python 3.10 to PATH" box.



9.2.2.5. Installing Git

When installing Git you should ensure that you change the default editor away from `vim`, see Figure 24.

Figure 24. Installing Git



Ensure you tick the checkbox to allow Git to be used from 3rd-party software and, unless you have a strong reason otherwise, when installing Git you should also check the box "Checkout as is, commit as-is", select "Use Windows' default console window", and "Enable experimental support for pseudo consoles" during the installation process.

9.2.2.6. Getting the SDK and examples

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

9.2.2.7. Building "Hello World" from the Command Line

Go ahead and open a Developer Command Prompt window from the Windows Menu, by selecting [Windows > Visual Studio 2022 > Developer Command Prompt for VS 2022](#) from the menu.

Then set the path to the SDK as follows,

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

You now need to **close your current Command Prompt window** and open a second Developer Command Prompt window where this environment variable will now be set correctly before proceeding.

Navigate into the [pico-examples](#) folder, and build the 'Hello World' example as follows,

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

! 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 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"`.

to build the target. This will produce `elf`, `bin`, and `uf2` targets, you can find these in the `hello_world/serial` and `hello_world/usb` directories inside your `build` directory. The UF2 binaries can be dragged-and-dropped directly onto a RP2040 board attached to your computer using USB.

9.2.2.8. Building "Hello World" from Visual Studio Code

Now you've installed the toolchain you can install [Visual Studio Code](#) and build your projects inside the that environment rather than from the command line.

Go ahead and [download](#) and install Visual Studio Code for Windows. After installation open a Developer Command

Prompt window from the Windows Menu, by selecting [Windows > Visual Studio 2022 > Developer Command Prompt for VS2022](#) from the menu. Then type,

```
C:> code
```

at the prompt. This will open Visual Studio Code with all the correct environment variables set so that the toolchain is correctly configured.

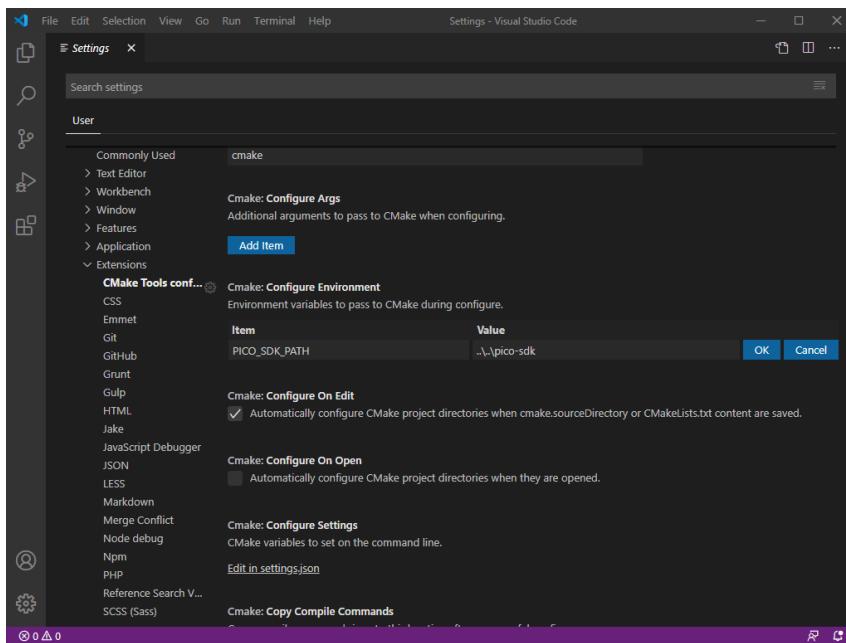
⚠️ WARNING

If you start Visual Studio Code by clicking on its desktop icon, or directly from the Start Menu then the build environment will **not** be correctly configured. Although this can be done manually later in the CMake Tools settings, the easiest way to configure the Visual Studio Code environment is just to open it from a Developer Command Prompt window where these environmental variables are already set.

We'll now need to install the [CMake Tools](#) extension. Click on the Extensions icon in the left-hand toolbar (or type [Ctrl + Shift + X](#)), and search for "CMake Tools" and click on the entry in the list, and then click on the install button.

Then click on the Cog Wheel at the bottom of the navigation bar on the left-hand side of the interface and select "Settings". Then in the Settings pane click on "Extensions" and then "CMake Tools". Then scroll down to "Cmake: Configure Environment". Click on "Add Item" and set the `PICO_SDK_PATH` to be `..\..\pico-sdk` as in [Figure 25](#).

Figure 25. Setting `PICO_SDK_PATH` Environment Variable in the CMake Extension



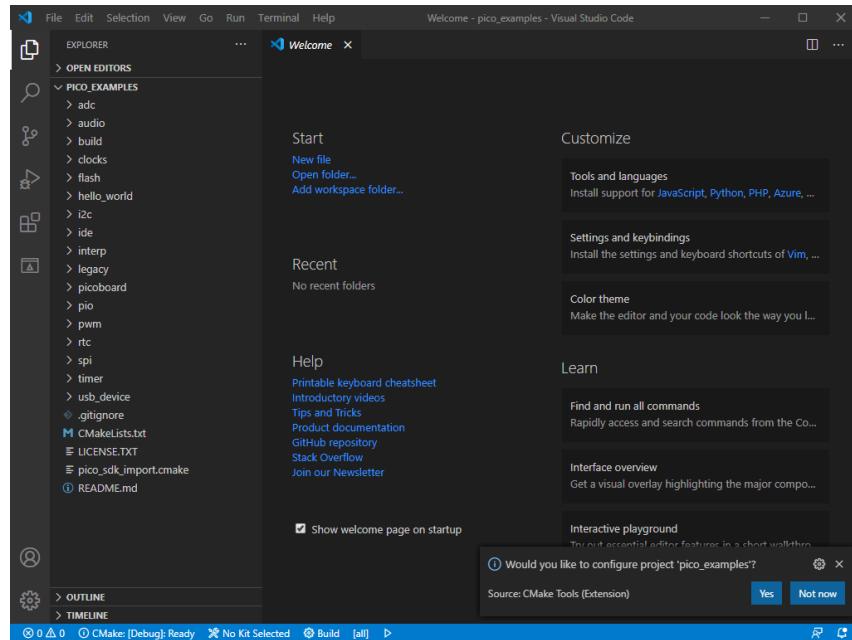
Additionally you will need to scroll down to "Cmake: Generator" and enter "NMake Makefiles" into the box.

❗️ IMPORTANT

If you do not change the "Cmake: Generator" Visual Studio will default to `ninja` and the build might fail as GCC outputs dependency-information in a slightly-incorrect format that `ninja` can't understand.

Now close the Settings page and go to the File menu and click on "Open Folder" and navigate to [pico-examples](#) repo and click "Select Folder". You'll be prompted to configure the project, see [Figure 26](#). Select "GCC for arm-none-eabi" for your compiler.

Figure 26. Prompt to configure your project in Visual Studio Code.



Go ahead and click on the "Build" button (with a cog wheel) in the blue bottom bar of the window. This will create the build directory and run CMake and build the examples project, including "Hello World".

This will produce `elf`, `bin`, and `uf2` targets, you can find these in the `hello_world/serial` and `hello_world/usb` directories inside the newly created `build` directory. The UF2 binaries can be dragged-and-dropped directly onto a RP2040 board attached to your computer using USB.

9.2.2.9. Flashing and Running "Hello World"

Connect the Raspberry Pi Pico to your Raspberry Pi using a micro-USB cable, making sure that you hold down the `BOOTSEL` button to force it into USB Mass Storage Mode. The board should automatically appear as a external drive. You can now drag-and-drop the UF2 binary onto the external drive.

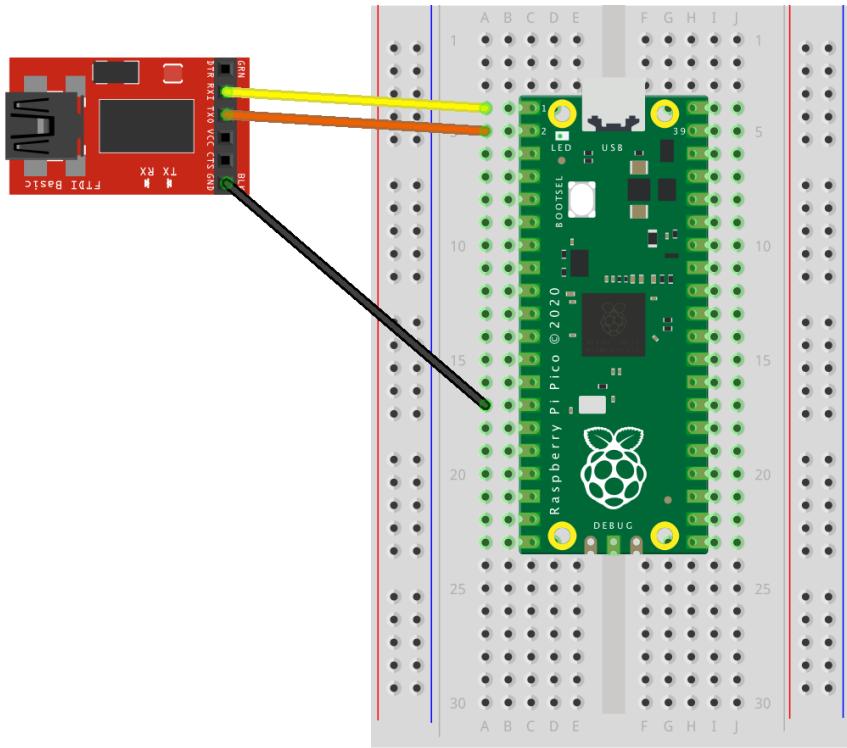
The Raspberry Pi Pico will reboot, and unmount itself as an external drive, and start running the flashed code.

As we did in [Chapter 4](#) you can build the Hello World example with `stdio` routed either to USB CDC (Serial) or to UART0 on pins GP0 and GP1. No driver installation is necessary if you're building with USB CDC as the target output, as it's a class-compliant device.

9.2.2.9.1. UART output

Alternatively if you want to connect to the Raspberry Pi Pico standard UART to see the output you will need to connect your Raspberry Pi Pico to your computer using a USB to UART Serial converter, for example a [SparkFun FTDI Basic](#) board, see [Figure 27](#).

Figure 27. Sparkfun FTDI Basic adaptor connected to the Raspberry Pi Pico



So long as you're using a recent version of Windows 10, the appropriate drivers should already be loaded. Otherwise see the manufacturers' website for [FTDI Chip Drivers](#).

Then if you don't already have it, download and install [PuTTY](#). Run it, and select "Serial", enter 115,200 as the baud rate in the "Speed" box, and the serial port that your UART converter is using. If you don't know this you can find out using the `chgport` command,

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

this will give you a list of active serial ports. Here the USB to UART Serial converter is on `COM5`.

ⓘ NOTE

If you have multiple serial devices and can't figure out which one is your UART to USB serial converter, try unplugging your cable, and running `chgport` again to see which COM port disappears.

After entering the speed and port, click the "Open" button and you should see the UART output from the Raspberry Pi Pico in your Terminal window.

Chapter 10. Using other Integrated Development Environments

Currently the recommended Integrated Development Environment (IDE) is Visual Studio Code, see [Chapter 7](#). However other environments can be used with RP2040 and the Raspberry Pi Pico.

10.1. Using Eclipse

Eclipse is a multiplatform Integrated Development environment (IDE), available for x86 Linux, Windows and Mac. In addition, the latest version is now available for 64-bit ARM systems, and 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 use with the Raspberry Pi Pico. Instructions for other systems will be broadly similar, although connections to the Raspberry Pi Pico will vary. See [Section 9.2](#) and [Section 9.1](#) for more details on non-Linux platforms.

10.1.1. 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 [Chapter 5](#) for instructions on how to do this.

10.1.1.1. Installing Eclipse and Eclipse plugins

Install the latest version of Eclipse with Embedded CDT 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://projects.eclipse.org/projects/iot.embed-cdt/downloads>

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 the Raspberry Pi Pico.

10.1.1.2. 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.

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

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

- On the command line enter:

```
$ 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 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.

10.1.1.3. Building

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

10.1.1.4. OpenOCD

This example uses the OpenOCD system to communicate with the Raspberry Pi Pico. You will need to have provided the 2-wire debug connections from the host device to the Raspberry Pi Pico 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](#). More instructions on the debug connections can be found in [Chapter 5](#).

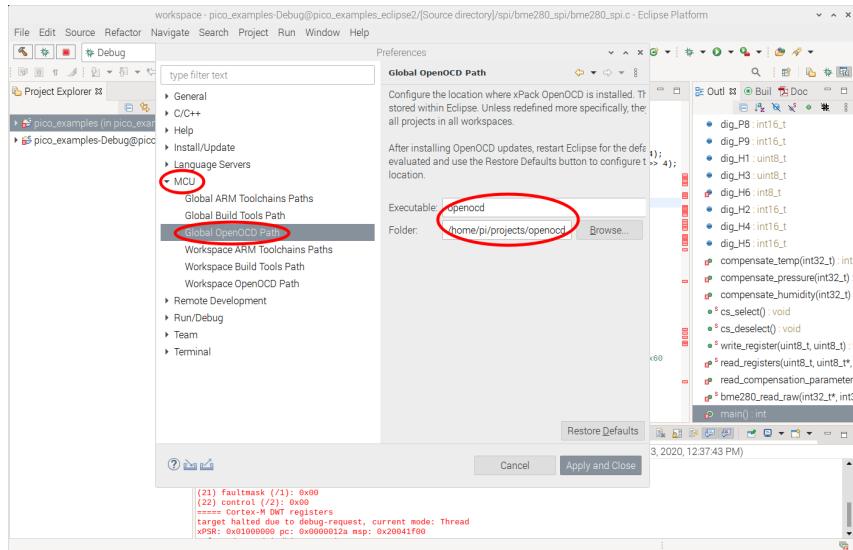
Once OpenOCD is installed and the correct connection made, Eclipse needs to be set up to talk to OpenOCD when programs are run. OpenOCD provides a GDB interface to Eclipse, and it is that interface that is used when debugging.

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 28. Setting the OCD executable name and path in Eclipse.

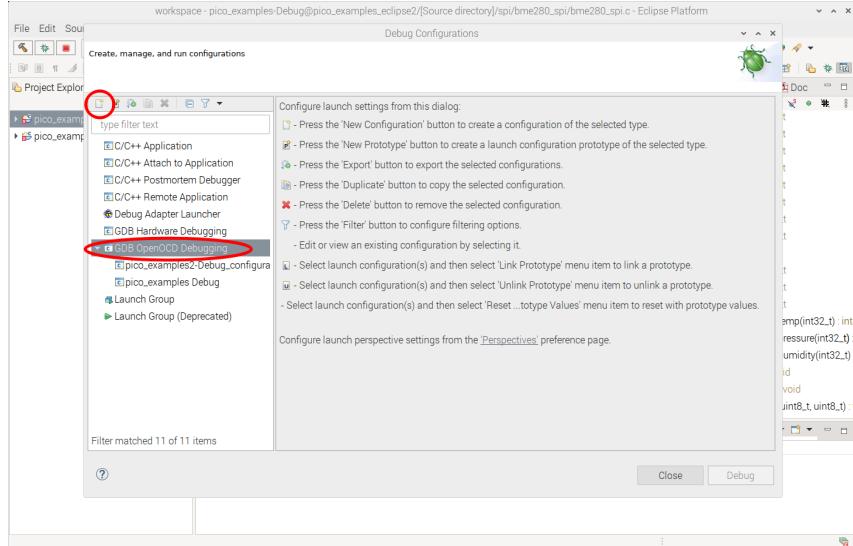


10.1.1.5. Creating a Run configuration

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 29. Creating a new Run/Debug configuration in Eclipse.



10.1.1.5.1. 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.

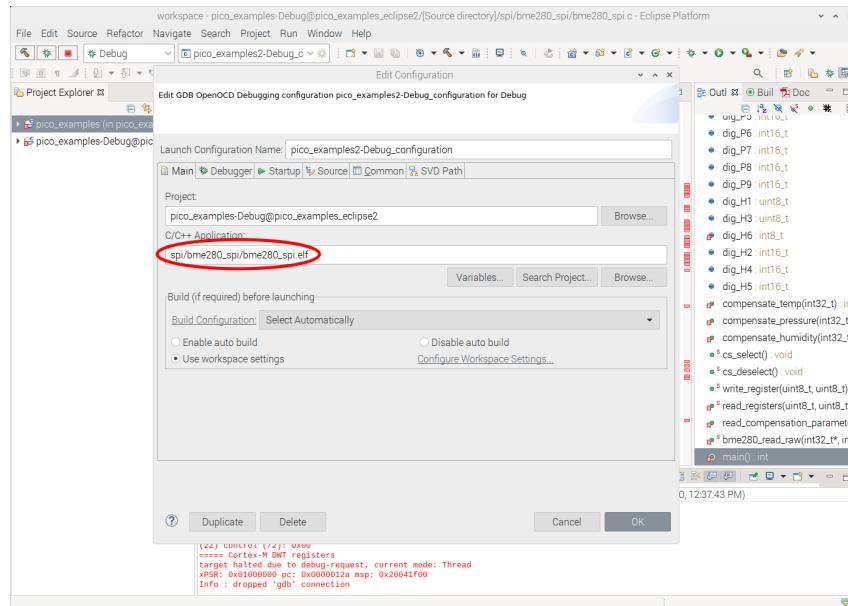
The Eclipse build will have created the executables in sub folders of the Eclipse project folder. In our example case this is

`.../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 30. Setting the executable to debug in Eclipse.



10.1.1.5.2. Setting up the debugger

We are using OpenOCD to talk to the Raspberry Pi Pico, so we need to set this up.

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.

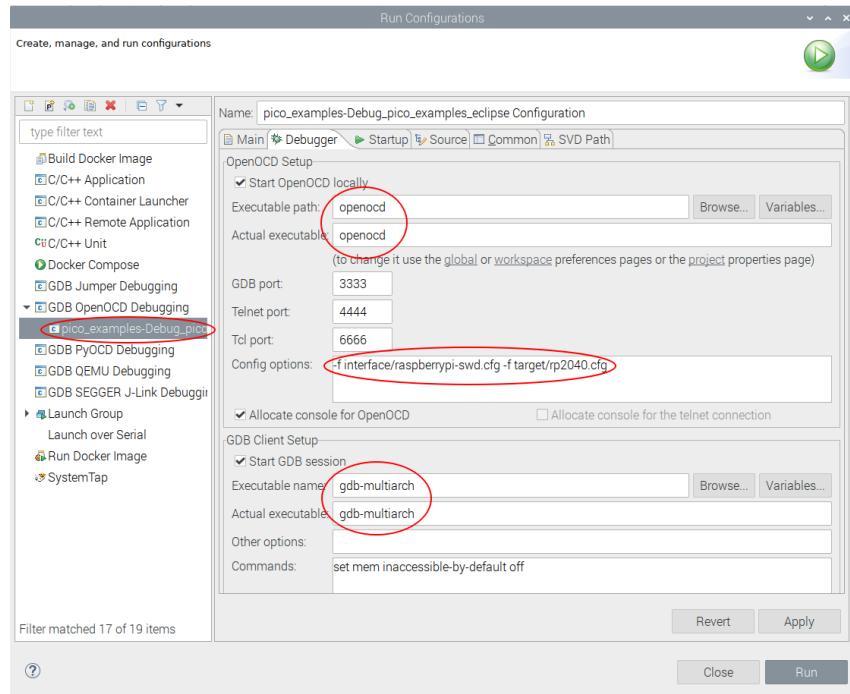
```
-f interface/raspberrypi-swd.cfg -f target/rp2040.cfg
```

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 Pico, 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 31. Setting up the Debugger and OpenOCD in Eclipse.



10.1.1.5.3. Setting up the SVD plugin

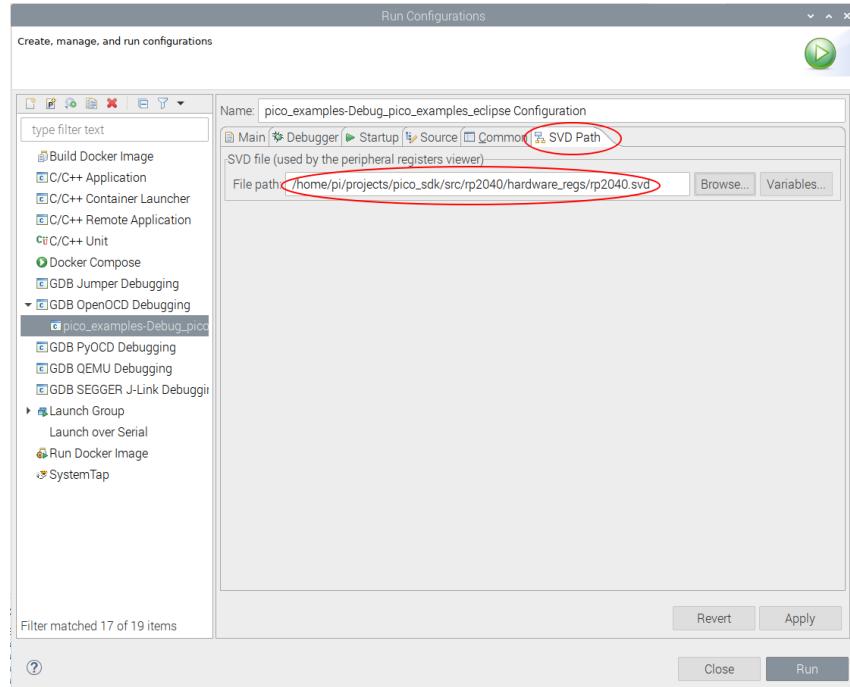
SVD provides a mechanism to view and set peripheral registers on the Pico board. An SVD file provides register locations and descriptions, and the SVD plugin for Eclipse integrates that functionality into the Eclipse IDE. The SVD plugin comes as part of the Embedded development plugins.

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

Figure 32. Setting the SVD path in Eclipse.

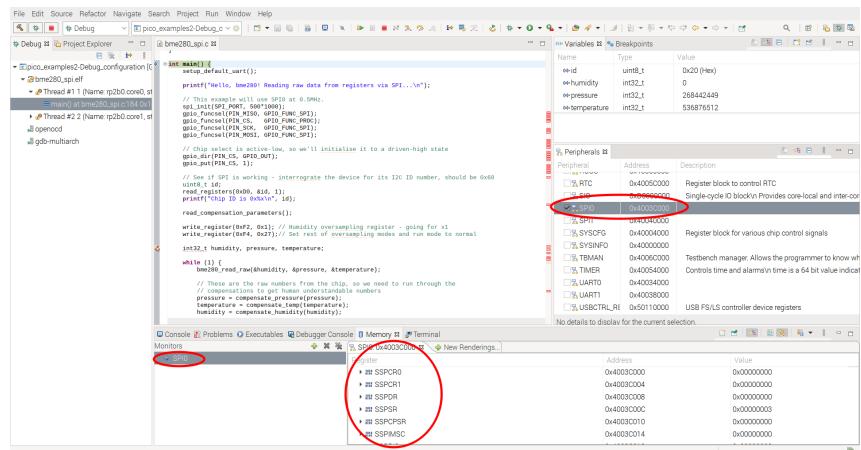


10.1.1.5.4. Running the Debugger

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



10.2. Using 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.

10.2.1. 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/>

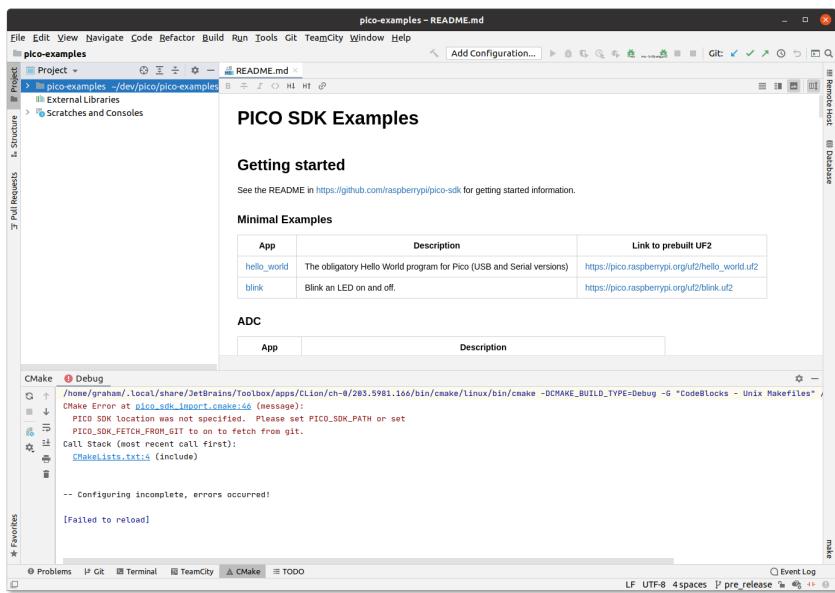
10.2.1.1. Setting up a project

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 34](#).

Figure 34. A newly opened CLion pico-examples project.



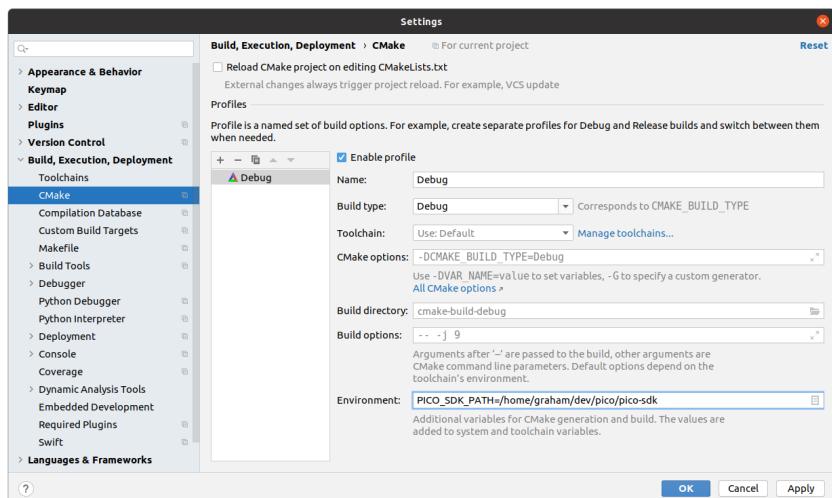
Notice at the bottom that CLion attempted to load the CMake project, but there was an error; namely that we hadn't specified `PICO_SDK_PATH`

10.2.1.1.1. 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 35, 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 35. Configuring a CMake profile in CLion.

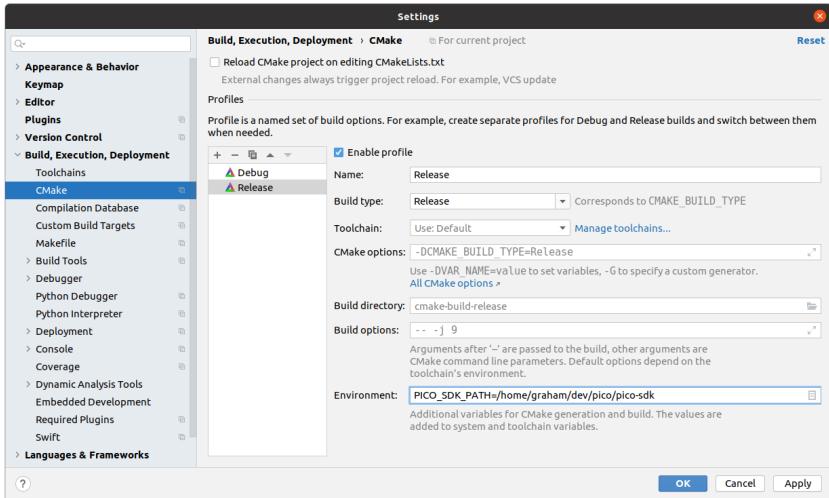


! IMPORTANT

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 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"`.

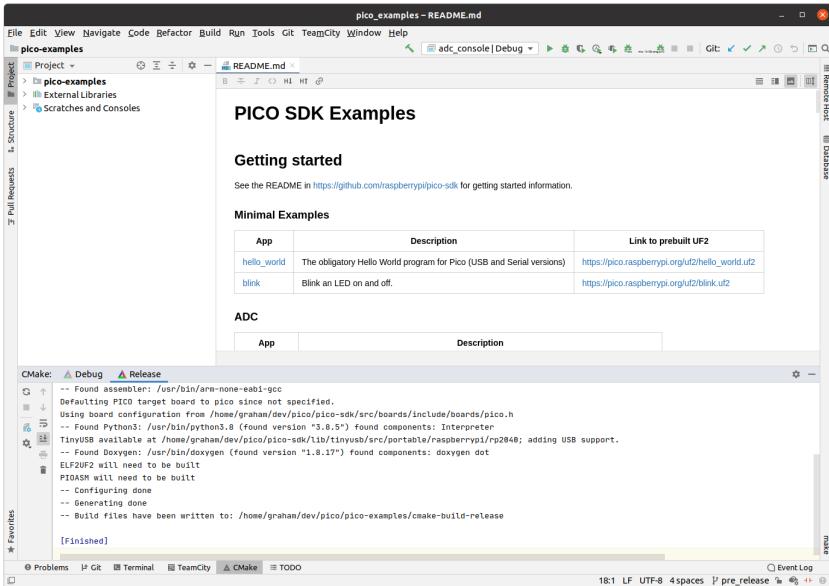
You can have as many CMake profiles as you like with different settings. You probably want to add a `Release` build by hitting the `+` button, and then filling in the `PICO_SDK_PATH` again, or by hitting the copy button two to the right, and fixing the name and settings (see [Figure 36](#))

[Figure 36. Configuring a second CMake profile in CLion.](#)



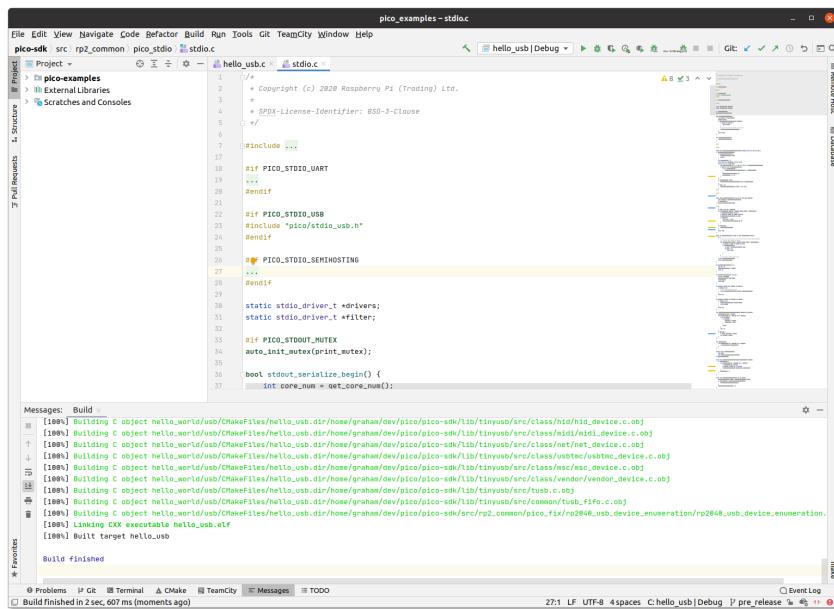
After pressing `OK`, you'll see something like [Figure 37](#). Note that there are two tabs for the two profiles (`Debug` and `Release`) at the bottom of the window. In this case `Release` is selected, and you can see that the CMake setup was successful.

[Figure 37. Configuring a second CMake profile in CLion.](#)



10.2.1.1.2. 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 start typing `hello_usb`; then press the tool icon to its left to build (see [Figure 38](#)). Alternatively you can do a full build of all targets or other types of build from the `Build` menu.

Figure 38. `hello_usb` successfully built.

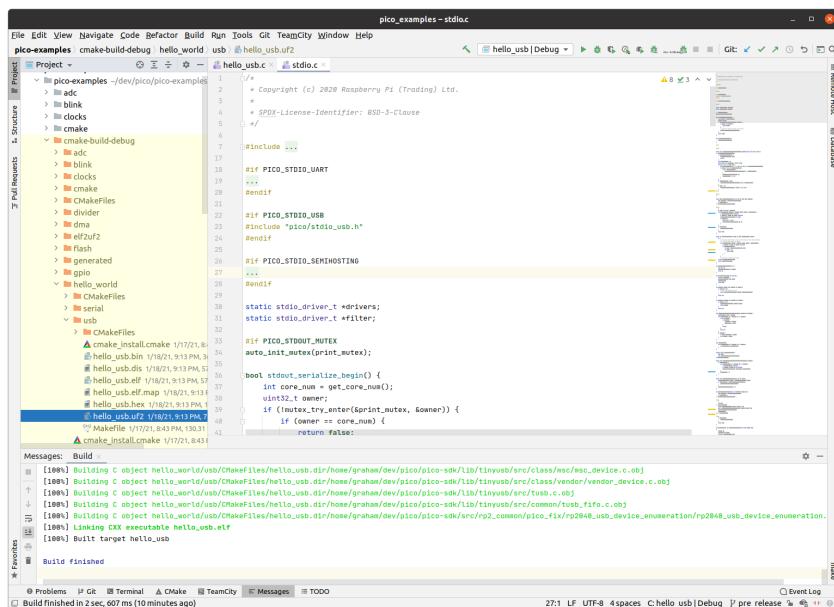
Note that the drop down selector lets you choose both the target you want to build and a CMake profile to use (in this case one of `Debug` or `Release`)

Another thing you'll notice Figure 38 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.

10.2.1.1.3. Build Artifacts

The build artifacts are located under `cmake-build-<profile>` under the project root (see Figure 39). In this case this is the `cmake-build-debug` directory.

The UF2 file can be copied onto an RP2040 device in BOOTSEL mode, or the ELF can be used for debugging.

Figure 39. Locating the `hello_usb` build artifacts

10.3. Other Environments

There are many development environments available, and we cannot describe all of them here, but you will be able to use many of them with the SDK. There are a number of things needed by your IDE that will make Raspberry Pi Pico support possible:

- 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.

10.3.1. Using openocd-svd

The `openocd-svd` tool is a Python-based GUI utility that gives you access to peripheral registers of ARM MCUs via combination of OpenOCD and CMSIS-SVD.

To install it you should first install the dependencies,

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

before cloning the `openocd-svd` git repository.

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

Ensuring your Raspberry Pi 4 and Raspberry Pi Pico are correctly wired together, we can attach OpenOCD to the chip, via the `swd` and `rp2040` configs.

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

⚠️ WARNING

If your flash has DORMANT mode code in it, or any code that stops the system clock, the debugger will fail to attach because the system clock is stopped. While this may present as a "bricked" board you can return to BOOTSEL mode using the button without problems.

This OpenOCD terminal needs to be left open. So go ahead and open another terminal, in this one we'll attach a `gdb` instance to OpenOCD.

Navigate to your project, and start `gdb`,

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

Connect GDB to OpenOCD,

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

and load it into flash, and start it running.

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

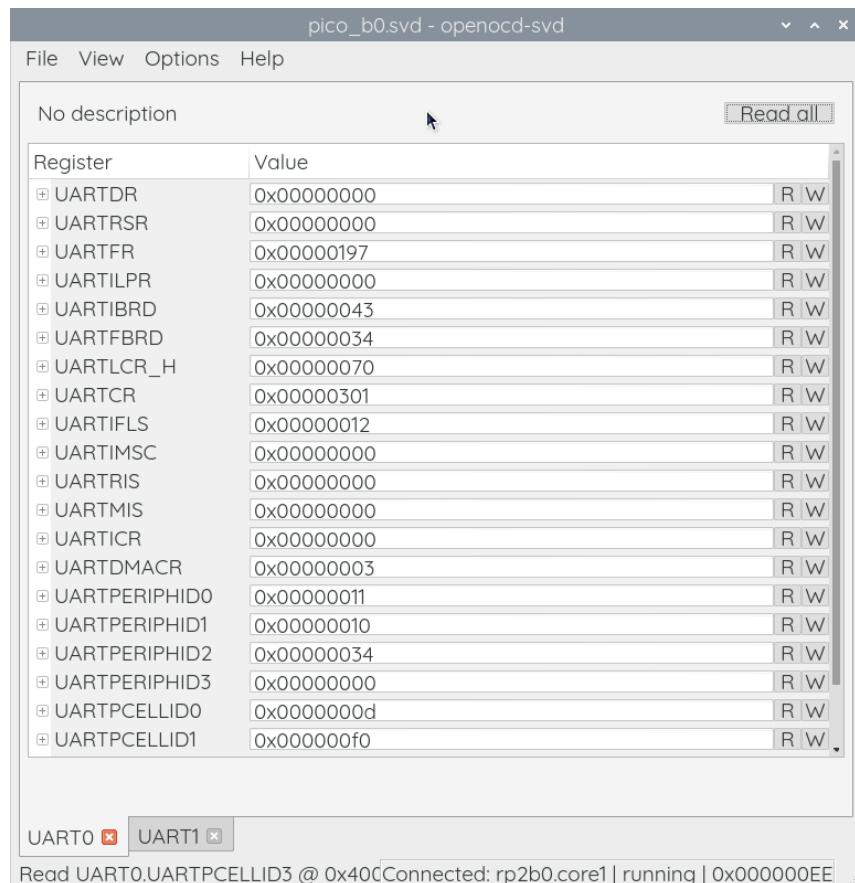
With both `openocd` and `gdb` running, open a third window and start `openocd-svd` pointing it to the SVD file in the SDK.

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

This will open the `openocd-svd` window. Now go to the File menu and click on "Connect OpenOCD" to connect via telnet to the running `openocd` instance.

This will allow you to inspect the registers of the code running on your Raspberry Pi Pico, see [Figure 40](#).

Figure 40. OpenOCD SVD running and connected to the Raspberry Pi Pico.



Appendix A: Using debugprobe

Raspberry Pi provides two ways to debug Raspberry Pi Pico:

- the [Raspberry Pi Debug Probe](#)
- `debugprobe` firmware running on a second Pico

Both methods provide a way to debug Pico 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.

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 Pico 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 41. Wires included with the Debug Probe.

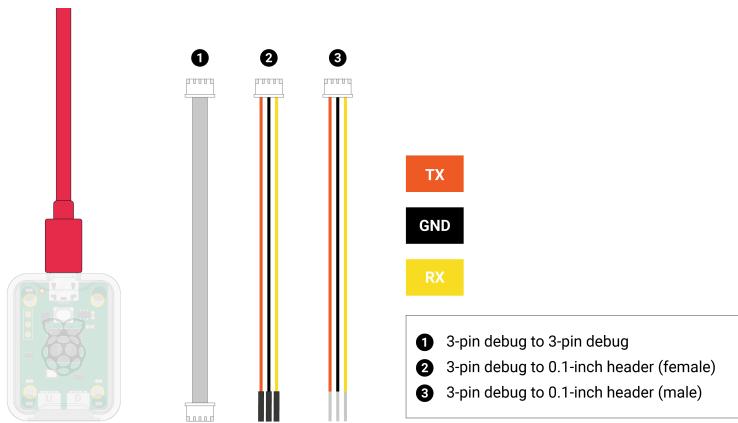
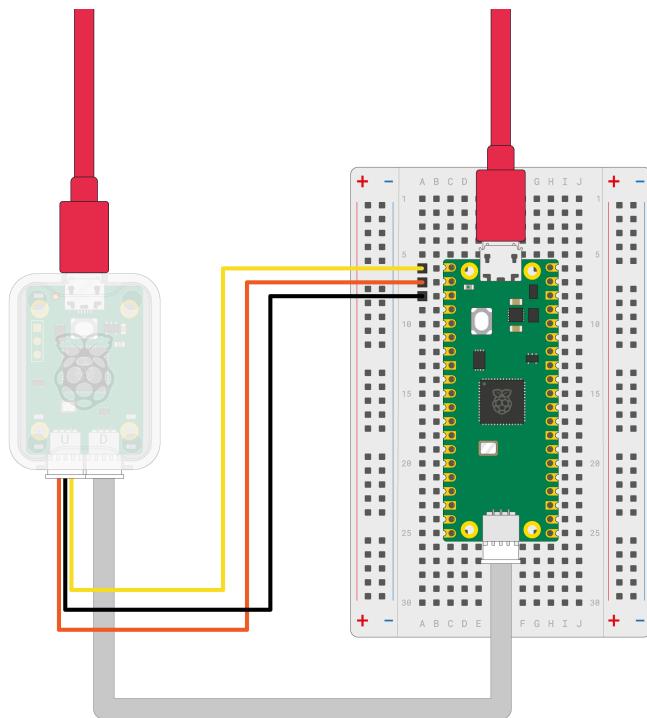


Figure 42. Wiring between the Debug Probe (left) and Pico (right).



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

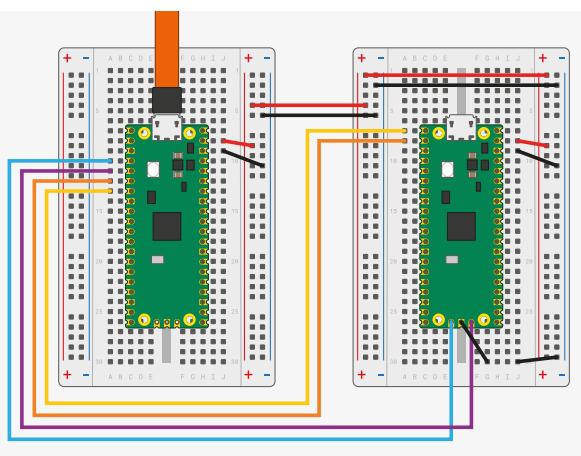
If you have a non-H Pico or Pico W (without a JST-SH connector) you can still connect it to a Debug Probe. Solder a male connector to the `SWCLK`, `GND`, and `SWDIO` header pins on the board. Using the alternate 3-pin JST-SH connector to 0.1-inch header (female) cable included with the Debug Probe, connect 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 41](#).

Debug with a second Pico

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

Figure 43. 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 documentation.

Boot the debugger Pico 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.

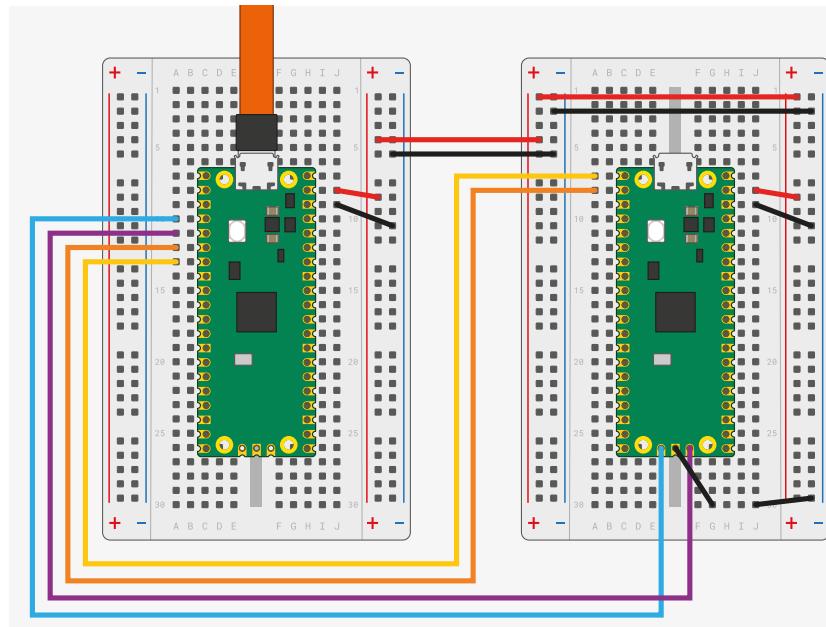
```
$ cd ~/pico
$ git clone https://github.com/raspberrypi/debugprobe.git
$ cd debugprobe
$ git submodule update --init
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake -DDEBUG_ON_PICO=ON ..
```

```
$ make -j4
```

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

debugprobe wiring

Figure 44. 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 44](#).

```

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 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

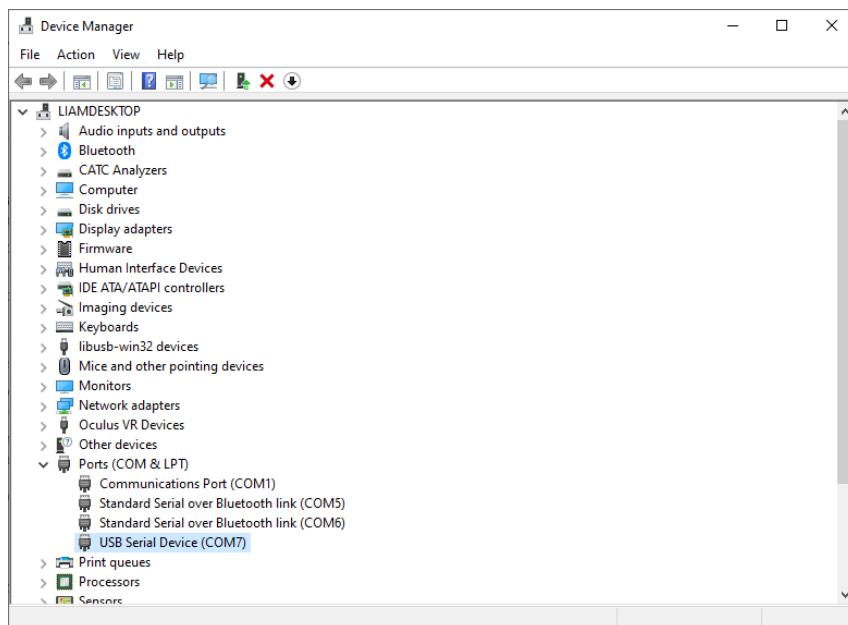
To use the UART connection on Linux, run the following command:

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

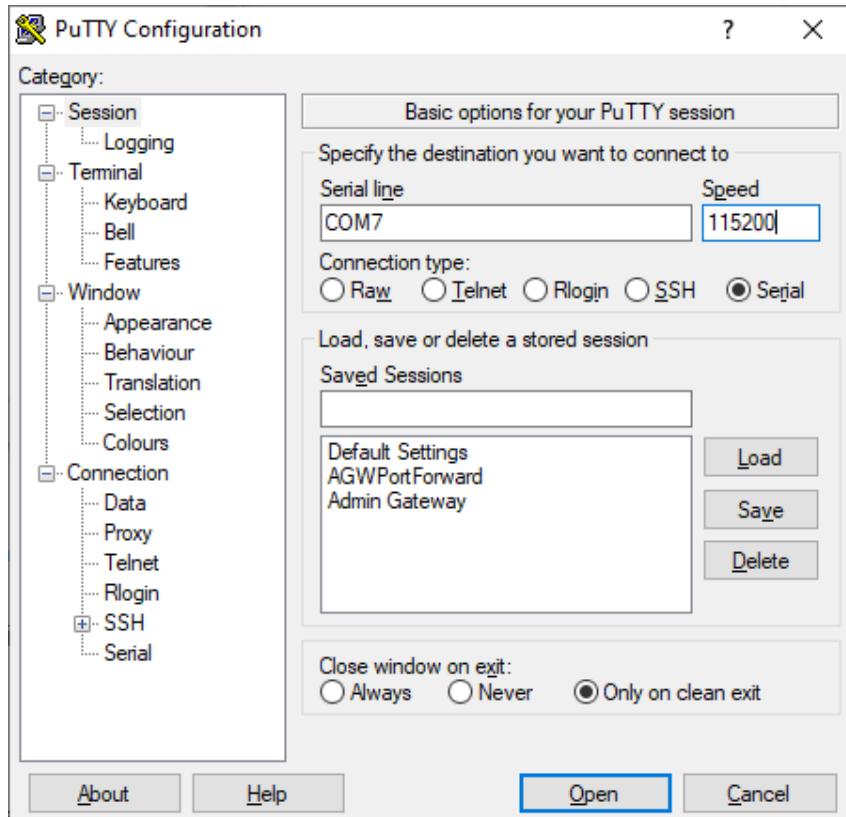
Windows

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"
```

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 ..
$ cd blink
$ make -j4
```

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"
```

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: Using Picotool

It is possible to embed information into a Raspberry Pi Pico 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 --branch master
$ 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
```

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

ℹ 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 a RP2040 device in BOOTSEL mode, or with a RP2040 binary

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

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

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

NOTE

The majority of commands require an RP2040 device in BOOTSEL mode to be connected.

IMPORTANT

If you get an error message `No accessible RP2040 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 Raspberry Pi Pico 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 RP2040 devices 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 RP2040 devices 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 Raspberry Pi Pico 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 generally machine-consumable. I say generally because anyone can include any information, and we can tell it from ours, but it is up to them whether they make their data self-describing.

Basic information

This information is really handy when you pick up a Pico 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 RP2040-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
```

Including Binary information

Binary information is declared in the program by macros; for the previous example:

```
$ 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
```

There is one line in the `setup_default_uart` function:

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

The two pin numbers, and the function UART are stored, then decoded to their actual function names (UART1 TX etc) by picotool. The `bi_decl_if_func_used` makes sure the binary information is only included if the containing function is called.

Equally, the video code contains a few lines like this:

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

Details

Things are designed to waste as little space as possible, but you can turn everything off with preprocessor var `PICO_NO_BINARY_INFO=1`. Additionally any SDK code that inserts binary info can be separately excluded by its own preprocessor var.

You need,

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

There are a bunch of `bi_` macros in the headers

```
#define bi_binary_end(end)
#define bi_program_name(name)
#define bi_program_description(description)
#define bi_program_version_string(version_string)
#define bi_program_build_date_string(date_string)
#define bi_program_url(url)
#define bi_program_feature(feature)
#define bi_program_build_attribute(attr)
#define bi_1pin_with_func(p0, func)
#define bi_2pins_with_func(p0, p1, func)
#define bi_3pins_with_func(p0, p1, p2, func)
#define bi_4pins_with_func(p0, p1, p2, p3, func)
#define bi_5pins_with_func(p0, p1, p2, p3, p4, func)
#define bi_pin_range_with_func(plo, phi, func)
#define bi_pin_mask_with_name(pmask, label)
#define bi_pin_mask_with_names(pmask, label)
#define bi_1pin_with_name(p0, name)
#define bi_2pins_with_names(p0, name0, p1, name1)
#define bi_3pins_with_names(p0, name0, p1, name1, p2, name2)
#define bi_4pins_with_names(p0, name0, p1, name1, p2, name2, p3, name3)
```

which make use of underlying macros, e.g.

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

You then either use `bi_decl(bi_bla(...))` for unconditional inclusion of the binary info bla, or `bi_decl_if_func_used(bi_bla(...))` for binary information that may be stripped if the enclosing function is not included in the binary by the linker (think `--gc-sections`).

For example,

```
1 #include <stdio.h>
2 #include "pico/stl.h"
3 #include "hardware/gpio.h"
4 #include "pico/binary_info.h"
5
6 const uint LED_PIN = 25;
7
8 int main() {
9
10     bi_decl(bi_program_description("This is a test binary."));
11     bi_decl(bi_1pin_with_name(LED_PIN, "On-board LED"));
12
13     setup_default_uart();
14     gpio_set_function(LED_PIN, GPIO_FUNC_PROC);
15     gpio_set_dir(LED_PIN, GPIO_OUT);
16     while (1) {
17         gpio_put(LED_PIN, 0);
```

```

18     sleep_ms(250);
19     gpio_put(LED_PIN, 1);
20     puts("Hello World\n");
21     sleep_ms(1000);
22 }
23 }
```

when queried with `picotool`,

```

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

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

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

Build Information
build date:  Jan  4 2021
```

shows our information strings in the output.

Setting common fields from CMake

You can also set fields directly from your project's CMake file, e.g.,

```

pico_set_program_name(foo "not foo") ①
pico_set_program_description(foo "this is a foo")
pico_set_program_version_string(foo "0.0001a")
pico_set_program_url(foo "www.plinth.com/foo")
```

1. The name "foo" would be the default.

ⓘ NOTE

All of these are passed as command line arguments to the compilation, so if you plan to use quotes, newlines etc. you may have better luck defining it using `bi_decl` in the code.

Appendix C: Documentation release history

Table 1.
Documentation release history

Release	Date	Description
1.0	21 Jan 2021	<ul style="list-style-type: none"> Initial release
1.1	26 Jan 2021	<ul style="list-style-type: none"> Minor corrections Extra information about using DMA with ADC Clarified M0+ and SIO CPUID registers Added more discussion of Timers Update Windows and macOS build instructions Renamed books and optimised size of output PDFs
1.2	01 Feb 2021	<ul style="list-style-type: none"> Minor corrections Small improvements to PIO documentation Added missing TIMER2 and TIMER3 registers to DMA Explained how to get MicroPython REPL on UART To accompany the V1.0.1 release of the C SDK
1.3	23 Feb 2021	<ul style="list-style-type: none"> Minor corrections Changed font Additional documentation on sink/source limits for RP2040 Major improvements to SWD documentation Updated MicroPython build instructions MicroPython UART example code Updated Thonny instructions Updated Project Generator instructions Added a FAQ document Added errata E7, E8 and E9
1.3.1	05 Mar 2021	<ul style="list-style-type: none"> Minor corrections To accompany the V1.1.0 release of the C SDK Improved MicroPython UART example Improved Pinout diagram
1.4	07 Apr 2021	<ul style="list-style-type: none"> Minor corrections Added errata E10 Note about how to update the C SDK from Github To accompany the V1.1.2 release of the C SDK

Release	Date	Description
1.4.1	13 Apr 2021	<ul style="list-style-type: none"> Minor corrections Clarified that all source code in the documentation is under the 3-Clause BSD license.
1.5	07 Jun 2021	<ul style="list-style-type: none"> Minor updates and corrections Updated FAQ Added SDK release history To accompany the V1.2.0 release of the C SDK
1.6	23 Jun 2021	<ul style="list-style-type: none"> Minor updates and corrections ADC information updated Added errata E11
1.6.1	30 Sep 2021	<ul style="list-style-type: none"> Minor updates and corrections Information about B2 release Updated errata for B2 release
1.7	03 Nov 2021	<ul style="list-style-type: none"> Minor updates and corrections Fixed some register access types and descriptions Added core 1 launch sequence info Described SDK "panic" handling Updated picotool documentation Additional examples added to Appendix A: App Notes appendix in the Raspberry Pi Pico C/C++ SDK book To accompany the V1.3.0 release of the C SDK
1.7.1	04 Nov 2021	<ul style="list-style-type: none"> Minor updates and corrections Better documentation of USB double buffering Picoprobe branch changes Updated links to documentation
1.8	17 Jun 2022	<ul style="list-style-type: none"> Minor updates and corrections Updated setup instructions for Windows in Getting started with Raspberry Pi Pico Additional explanation of SDK configuration RP2040 now qualified to -40°C, minimum operating temperature changed from -20°C to -40°C Increased PLL min VCO from 400MHz to 750MHz for improved stability across operating conditions Added reflow-soldering temperature profile Added errata E12, E13 and E14 To accompany the V1.3.1 release of the C SDK

Release	Date	Description
1.9	30 Jun 2022	<ul style="list-style-type: none"> Minor updates and corrections Update to VGA board hardware description for launch of Raspberry Pi Pico W To accompany the V1.4.0 release of the C SDK
Pico and Pico W databooks combined into a unified release history		
2.0	01 Dec 2022	<ul style="list-style-type: none"> Minor updates and corrections Added RP2040 availability information Added RP2040 storage conditions and thermal characteristics Replace SDK library documentation with links to the online version Updated Picoprobe build and usage instructions
2.1	03 Mar 2023	<ul style="list-style-type: none"> A large number of minor updates and corrections SMT footprint of Pico W corrected Updated for the 1.5.0 release of the Raspberry Pi Pico C SDK Added errata E15 Added documentation around the new Pico Windows Installer Added documentation around the Pico-W-Go extension for Python development Added a wireless networking example to the Python documentation Added package marking specifications Added RP2040 baseline power consumption figures Added antenna keep out diagram to Pico W datasheet
2.2	14 Jun 2023	<ul style="list-style-type: none"> Minor updates and corrections Updated for the 1.5.1 release of the Raspberry Pi Pico C SDK Documentation around Bluetooth support for Pico W
2.3	02 Feb 2024	<ul style="list-style-type: none"> Numerous minor updates and corrections Update ROSC register information Updated getting started documentation for MS Windows and Apple macOS Updates arising from the release of Raspberry Pi 5 Reintroduced updated SDK library documentation (was withdrawn in 2.0 due to XML conflicts) Updated to include the new recommended part number for crystals used with RP2040 Added new paste stencil information for Pico and Pico W Other updates to supporting documentation

Release	Date	Description
2.4	02 May 2024	<ul style="list-style-type: none">• Numerous minor updates and corrections• Formatting fixes for Pico C SDK API level documentation• Renamed <code>picoprobe</code> firmware to <code>debugprobe</code>• Clarified that CMake build configuration uses cache variables, not configuration variables• Fixed incorrect parameter names used in the <code>@asm_pio</code> decorator and <code>StateMachine</code> initialiser examples• Expanded MicroPython <code>rshell</code> 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.

The latest release can be found at <https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf>.



Raspberry Pi is a trademark of Raspberry Pi Ltd