Getting Started with Raspberry Pi Pico

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

Colophon

build-date: 2020-11-09
build-version: githash: 5ddd120-clean (pico-sdk: f430778-clean pico-examples: e835d44-clean)

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 (TRADING) LTD ("RPTL) "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 RPTL 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.

RPTL 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 RPTL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPTL 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 RPTL 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 pf 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"). RPTL 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 RPTL's Standard Terms. RPTL's provision of the RESOURCES does not expand or otherwise modify RPTL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Table of Contents

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. You can get this script by doing the following:

TO DO: We can just `wget` after launch to avoid needing to git clone it. Remove warning then.

```
$ git clone git@github.com:raspberrypi/pico-setup.git ①
```

1. `sudo apt install git` if you don't have it installed

WARNING

Until launch all the repositories cloned by the script are **private**. That means that before you run the script you need to upload a SSH key to Github as the clones will fail. Before proceeding generate a SSH key on the Raspberry Pi as follows,

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/pi/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/pi/.ssh/id_rsa.
Your public key has been saved in /home/pi/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:yITLA+9tqti7EoI+dP60QGz1uzoTMddBocPbBJ67fw4 pi@pi400
The key's randomart image is:
+---[RSA 2048]----+
|       ..o.      |
|     .o +.       |
|  . ...*...      |
|  .+.*.o*.       |
|.  += *+S.       |
|+.oo +  o        |
|+.o...+o E       |
|.= .o=. o ..     |
|..*+o++. .o.     |
+----[SHA256]-----+
$
```

and then upload this file to your Github account. You should now be able to clone, pull and push to git repositories hosted at Github from the command line.

Then run,

```
$ pico-setup/pico_setup.sh
```

The script will:

- Create a directory called `pico`

- Install required dependencies
- Download the `pico-sdk` and `pico-examples` repositories
- Define `PICO_SDK_PATH` in your `~/.bashrc`
- Build the `blink` and `hello_world` examples in `pico-examples/build/blink` and `pico-examples/build/hello_world`
- Download and compile OpenOCD (for debug support)
- Download and install Visual Studio Code
- Install the required Visual Studio Code extensions (see Chapter 6 for more details)
- Configure the Raspberry Pi UART for use with Raspberry Pi Pico

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

```
$ sudo reboot
```

for the UART reconfiguration to take effect. Then hook up your Pico as shown in Figure 5.

Once you have done that you can open Visual Studio Code in the "Programming" menu and follow the instructions from Section 6.2.

Chapter 2. The Pico 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 8.2) or Apple macOS (see Section 8.1) 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 **Pico Python SDK** book which documents the port, and "**Get Started with MicroPython on Raspberry Pi Pico**" by Gareth Halfacree published by Pi Press.

TIP

For more information on the C/C++ SDK, along with API-level documentation, see the **Pico C/C++ SDK** book.

2.1. Get the Pico SDK and examples

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 -b pre_release git@github.com:raspberrypi/pico-sdk.git
$ cd pico-sdk
$ git submodule update --init --recursive
$ cd ..
$ git clone -b pre_release git@github.com:raspberrypi/pico-examples.git
```

TO DO: Talk about **which versions of Linux** this works on, including Raspberry Pi

NOTE

The `pico-examples` repository (https://github.com/raspberrypi/pico-examples) provides a set of example applications that are written using the `pico-sdk` (https://github.com/raspberrypi/pico-sdk).

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 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 gcc g++ ①
```

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

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. We're going to go ahead and blink the on-board LED on the Raspberry Pi Pico which is connected to pin 25 of the RP2040.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/blink/blink.c Lines 10 - 20

```
10 int main() {
11     const uint LED_PIN = 25;
12     gpio_init(LED_PIN);
13     gpio_dir(LED_PIN, GPIO_OUT);
14     while (true) {
15         gpio_put(LED_PIN, 1);
16         sleep_ms(250);
17         gpio_put(LED_PIN, 0);
18         sleep_ms(250);
19     }
20 }
```

3.1. Building "Blink"

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

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

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

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

Prepare your cmake build directory by running `cmake ..`

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

NOTE

`cmake` will default to a `Release` build with compiler optimisations enabled and debugging information removed. To build a debug version, run `cmake -DCMAKE_BUILD_TYPE=Debug ..`. We will explore this later in Section 5.1.

CMake has now prepared a build area for the `pico-examples` tree. From here, it is possible to type `make` to build all example applications. However, as we are building `blink` we will only build that application for now by changing directory into the `blink` directory before typing `make`.

TIP

Invoking `make` with `-jx` will run make jobs in parallel to speed it up. A Raspberry Pi 4 has 4 cores so `-j4` is a reasonable number.

```
$ 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
```

Amongst other targets, we have now built:

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

This binary will blink the on-board LED of the Raspberry Pi Pico which is connected to GPIO25 of RP2040.

3.2. Load and run "Blink"

The simplest method to load software onto a RP2040-based board is by mounting it as a USB Mass Storage Device. Doing this allows you to drag a file onto the board to program the flash. Go ahead and connect the Raspberry Pi Pico to your Raspberry Pi using a micro-USB cable, making sure that you hold down the `BOOTSEL` button (Figure 1) to force it into USB Mass Storage Mode.

NOTE

Loading code via the USB Mass Storage method is great if you know your program is going to work first time, but if you are developing anything new it is likely you will want to debug it. So you can also load your software onto RP2040 using the Serial Wire Debug interface, see Chapter 5. As well as loading software this allows you to; set breakpoints, inspect variables, and inspect memory contents.

NOTE

If you are not following these instructions on a Raspberry Pi Pico, you may not have a `BOOTSEL` button, see Figure 1. If this is the case, you should check if there is some other way grounding the flash `CS` pin, such as a jumper, to tell RP2040 to enter the USB boot mode on boot. If there is no such method, you can load code using the Serial Wire Debug interface.

3.2.1. From the desktop

If you are running the Raspberry Pi Desktop the Raspberry Pi Pico should automatically mount as a USB Mass Storage Device. From here, you can Drag-and-drop `blink.uf2` onto the Mass Storage Device.

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

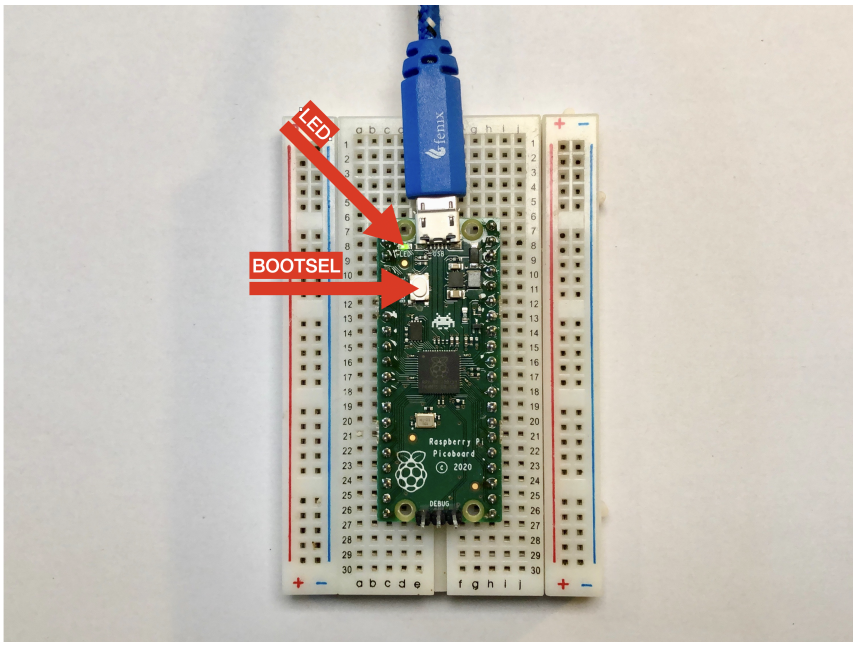


Figure 1. Blinking the on-board LED on the Raspberry Pi Pico. Arrows point to the on-board LED, and the `BOOTSEL` *button.*

3.2.2. Using the command line

If you are logged in via `ssh` for example, 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` then the USB Mass Storage Device has been 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 has already disconnected as a USB Mass Storage Device and is running your code, but for tidiness unmount `/mnt/pico`

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

NOTE

Removing power from the board does not remove the code. When the board is reattached to power the code you have just loaded will begin running again. If you want to remove the code from the board, and upload new code, press and hold the BOOTSEL switch when applying power to put the board into Mass Storage mode.

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 UART serial port, and say "Hello World."

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/hello_world/hello_world.c Lines 11 - 25

```
11 int main() {
12     setup_default_uart();
13     for(int32_t i = -1000; i<1000; i++) {
14         int32_t r = ((int16_t)i * 0xaaaa) >> 11;
15         int32_t exp = (int32_t)floor(i/12.0);
16         if (r != exp) {
17             printf("%d %d %d %d\n", i, r, exp, i/12);
18         }
19     }
20     while (true) {
21         printf("Hello, world!\n");
22         sleep_ms(1000);
23     }
24     return 0;
25 }
```

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/tree/pre_release/src/boards/include/boards/pico.h. The Pico SDK defaults to a boar dname of Raspberry Pi Pico if no other board is specified.

4.1. Build "Hello World"

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

```
$ cd hello_world
$ make -j4
[  0%] Built target boot_stage2_original
[  0%] Creating directories for 'ELF2UF2Build'
   .
   .
   .
[100%] Linking C executable hello_world.elf
[100%] Built target hello_world
```

Amongst other targets, we have now built:

- `hello_world.elf`, which is used by the debugger

- `hello_world.uf2`, which can dragged onto the RP2040 USB Mass Storage Device

4.2. 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 Raspbrerry Pi Desktop it should automatically mount as a USB Mass Storage Device. From here, you can Drag-and-drop `hello_world.uf2` onto the Mass Storage Device.

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

4.3. See "Hello World" UART output

Although the "Hello World" example is now running, we cannot yet see the text. We need to connect our host computer to the standard UART on the Raspberry Pi Pico to see the output. The first thing you'll need to do 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 2.

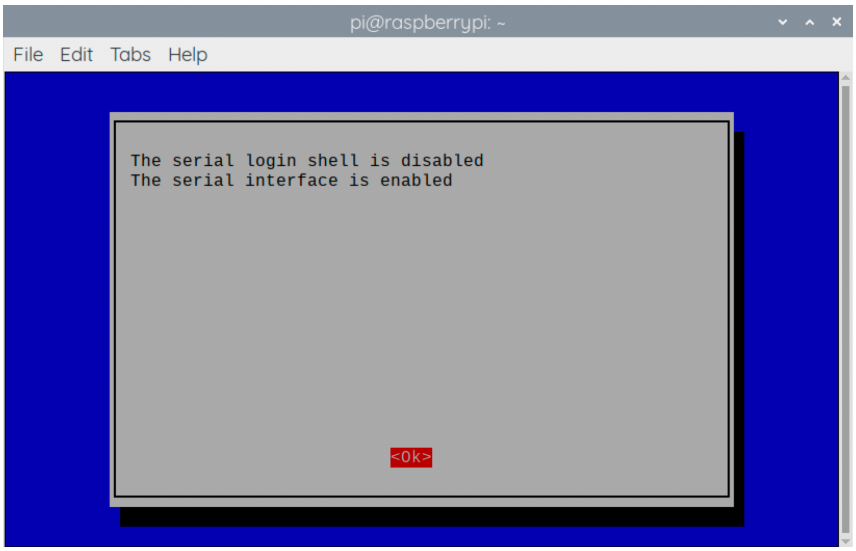


Figure 2. Enabling a serial UART using `raspi-config` on the Raspberry Pi.

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

You should then wire the 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 3.

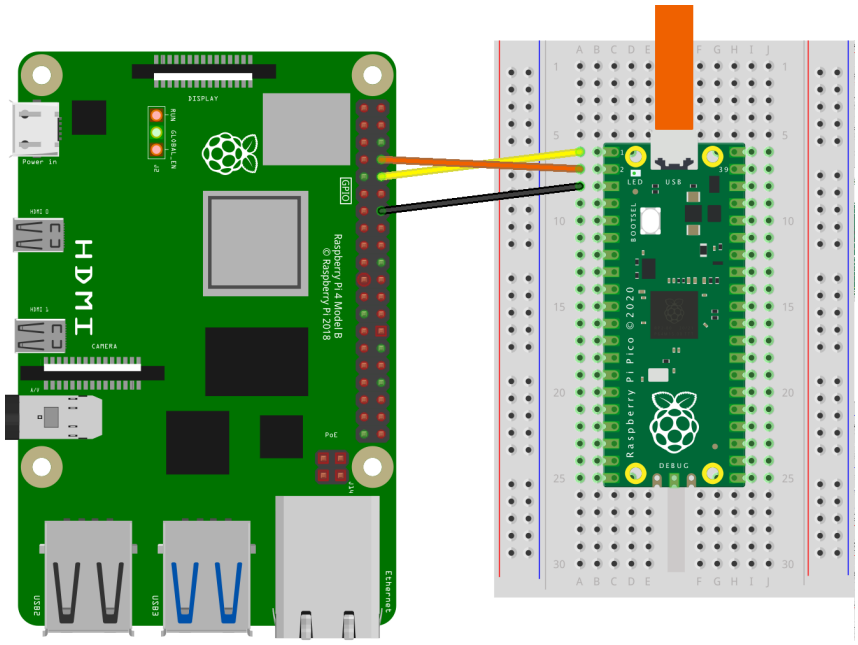


Figure 3. A Raspberry Pi 4 and the Raspberry Pi Pico with UART0 connected together.

Once the two boards are wired together you should install `minicom`:

```
$ sudo apt install minicom
```

and open the serial port:

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

Toggling the power to Raspberry Pi Pico you should see `Hello, world!` printed to the console when the Raspberry Pi Pico is first powered on.

TIP

In minicom, `CTRL-A` followed by `U` will add carriage returns to the serial output so that each print will end with a newline.

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

NOTE

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 4, where in the ideal case the diode would be a Schottky diode.

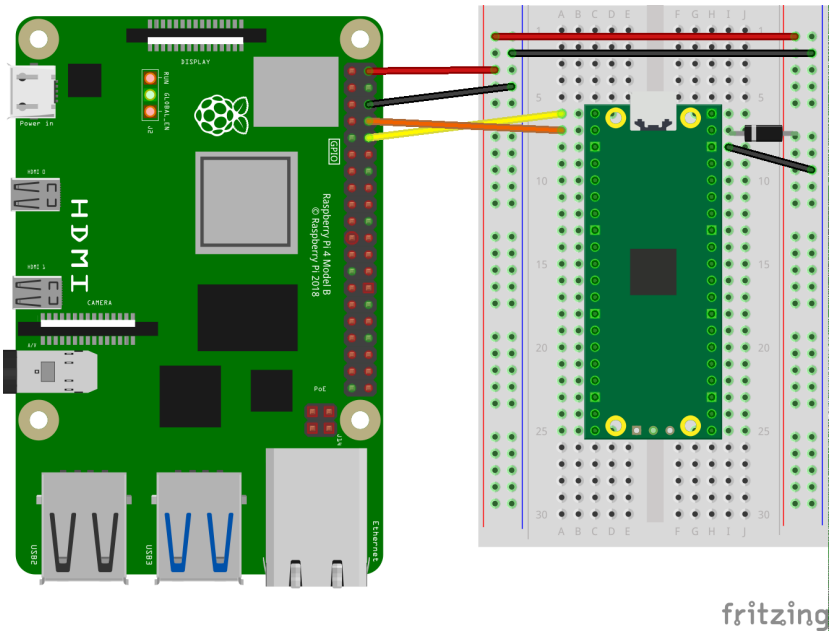


Figure 4. Raspberry Pi and Raspberry Pi Pico connected only using the GPIO pins.

TODO : This para is unclear

It is possible to connect the Raspberry Pi's 5V pin to the Raspberry Pi Pico VBUS pin, but this is not recommended as shorting the 5V rails together will mean the Micro USB cannot be used. Unless you want to use Raspberry Pi Pico in USB host mode, then you have to connect 5V 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.

Chapter 5. Debugging with SWD

IMPORTANT

These instructions assume that you are using a Raspberry Pi Pico, details may differ if you are using an alternative RP2040-based board.

The Raspberry Pi Pico provides a SWD (Single Wire Debug) port which can be used to interactively debug a binary running on RP2040. However to use it you will first need build a special debug version of your binary and install some additional tools.

5.1. Build "Hello World" debug version

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
$ make -j4
```

5.2. Installing OpenOCD

OpenOCD is a debug translator: it allows a host system to load, run and debug software on RP2040, and to interactively poke and explore hardware registers. OpenOCD can attach to RP2040's SWD port via a number of hardware interfaces, including direct bitbanging from Raspberry Pi GPIOs.

The default configuration is to have SWDIO on Pi GPIO 24, and SWCLK on GPIO 25 — this can be wired to a Raspberry Pi Pico as seen in Figure 5.

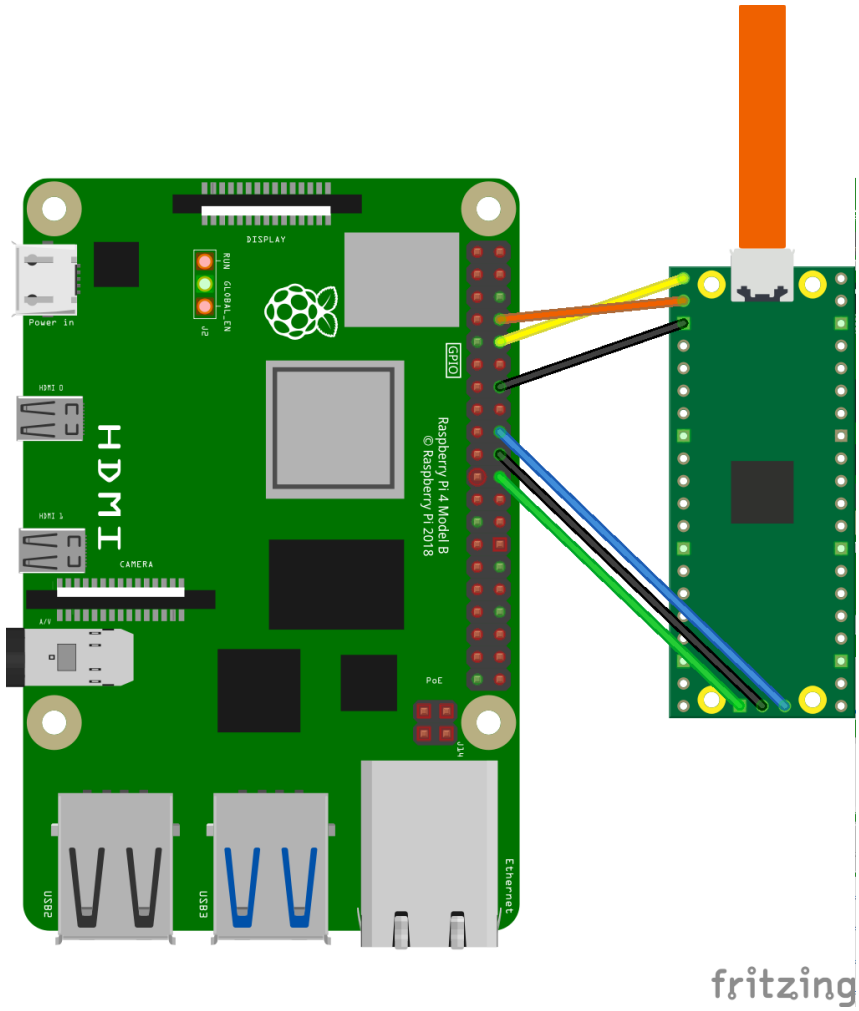


Figure 5. A Raspberry Pi 4 and the Raspberry Pi Pico with UART and SWD port connected together. Both are jumpered directly back to the Raspberry Pi 4 without using a breadboard.

If possible you should wire the SWD port directly to the Raspberry Pi as latency 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.

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:

TO DO: Build our fork as a debian package so we can apt install it

NOTE

These instructions assume you want to build openocd in `/home/pi/pico/openocd`

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

TO DO: All of our openocd config files need to be moved to the proper place and called pico*

5.3. Installing GDB

Install `gdb-multiarch`,

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

5.4. 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
```

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
$ gdb-multiarch hello_world.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 (Section Chapter 6).

and load `hello_world.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 5, 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/hello_world.c:11
11      setup_standard_uart();
```

before continuing after you have hit the breakpoint,

```
(gdb) continue
```

To quit from `gdb` type,

```
(gdb) quit
```

Chapter 6. Using Visual Studio Code

Visual Studio Code (VSCode) is a popular open source editor developed by Microsoft. It is the recommended Integrated Development Environment (IDE) on the Raspberry Pi 4 if you want a graphical interface to edit and debug your code.

6.1. Installing the Environment

IMPORTANT

These installation instructions rely on you already having downloaded and installed the command line toolchain, see Chapter 3, as well as downloading and building both OpenOCD and GDB and configuring them for command line debugging, see Chapter 5.

ARM versions of Visual Studio Code for the Raspberry Pi can be downloaded from https://code.visualstudio.com/Download. If you are using a 32-bit operating system (e.g. the default Raspberry Pi OS) then download the `ARM .deb` file; if you are using a 64-bit OS, then download the `ARM 64 .deb` file. Once downloaded, double click on the .deb package and follow the instructions to install it.

NOTE

You can install the downloaded `.deb`. package fromm the command line. `cd` to the folder where the file was downloaded, then use `dpkg -i <downloaded file name.deb>` to install.

Once the install has completed, go ahead and start Visual Studio Code from a Terminal window as follows,

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

Ensure you set the `PICO_SDK_PATH` so the Visual Studio Code can find the Pico SDK.

NOTE

If `PICO_SDK_PATH` is not set by default in your shell's environment you will have to set it each time you open a new Terminal window before starting vscode, or start vscode from the menus. You may therefore want to add it to your `.profile` or `.bashrc` file.

6.1.1. Install Cortex-Debug

After starting Visual Studio Code you then need to install the Cortex-Debug extension by Marcel Ball. Click on the Extensions icon in the left-hand toolbar (or type `Ctrl + Shift + X`), and search for `cortex-debug` and click on the entry in the list. The click the install button (see Figure 6).

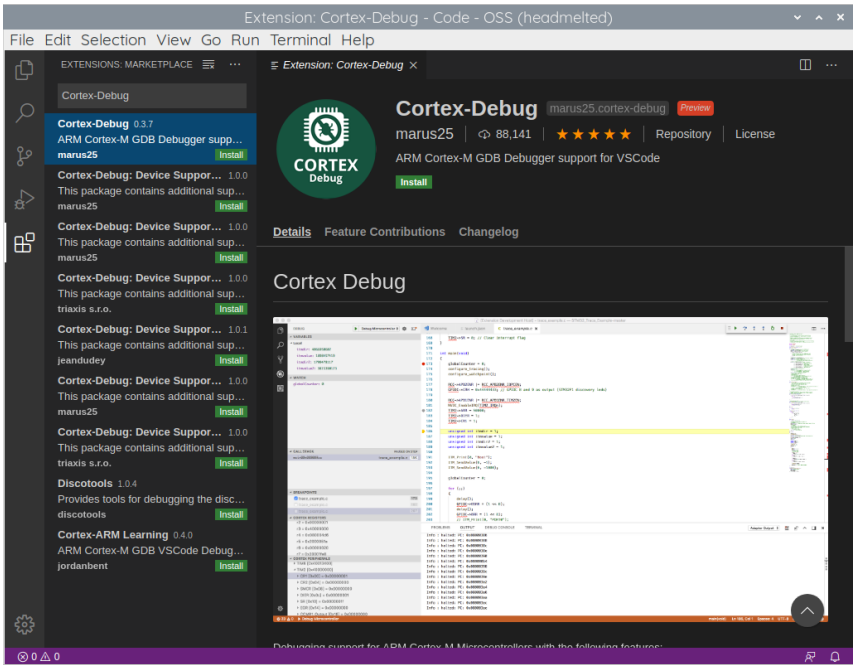


Figure 6. Installing the Cortex-Debug extension into Visual Studio Code.

Right now the Cortex-Debug extension expects `arm-none-eabi-gdb` to exist but we are using `gdb-multiarch` for debugging. To fix this, create a symbolic link to `gdb-multiarch` called `arm-none-eabi-gdb`,

```
$ sudo ln -s /usr/bin/gdb-multiarch /usr/local/bin/arm-none-eabi-gdb
```

TO DO: FIXME: This is a hack, needs to be resolved before launch.

6.1.2. Install CMake Tools

Similarly search for and install the CMake Tools extension, see Figure 7

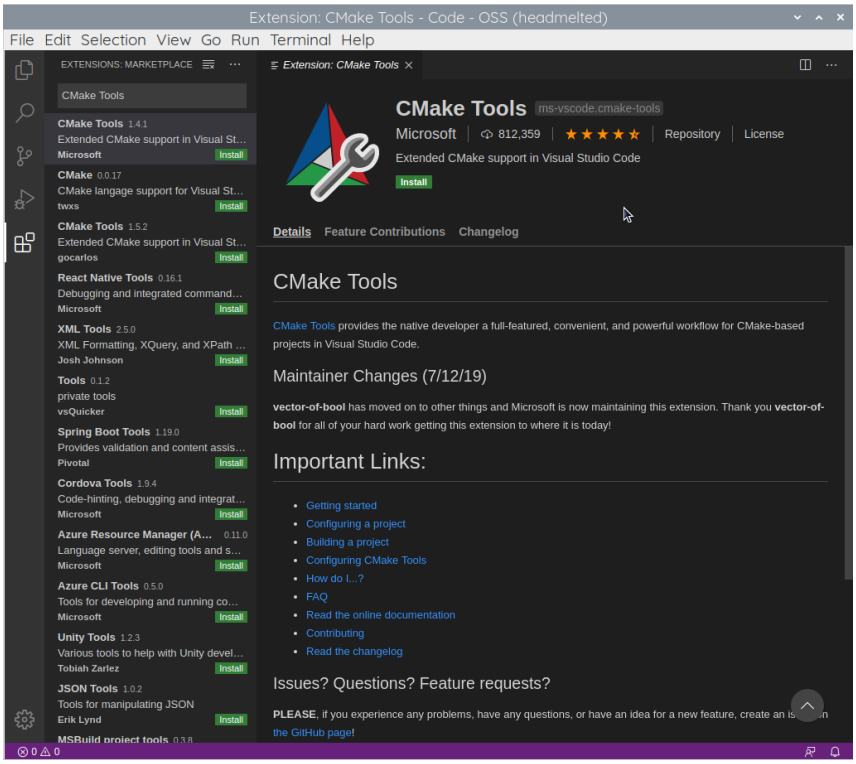


Figure 7. Installing the Cmake Tools extension into Visual Studio Code.

6.2. Loading a Project

Go ahead and open the `pico-examples` folder by going to the Explorer toolbar (`Ctrl + Shift + E`), selecting "Open Folder," and nagivating to, `/home/pi/pico/pico-examples` in the file popup. Then click "OK" to load the Folder into VSCode.

As long as the CMake Tools extension is installed, after a second or so you should see a popup in the lower right-hand corner of the vscode window.

Hit "Yes" to configure the project. You will then be prompted to choose a compiler, see Figure 8,

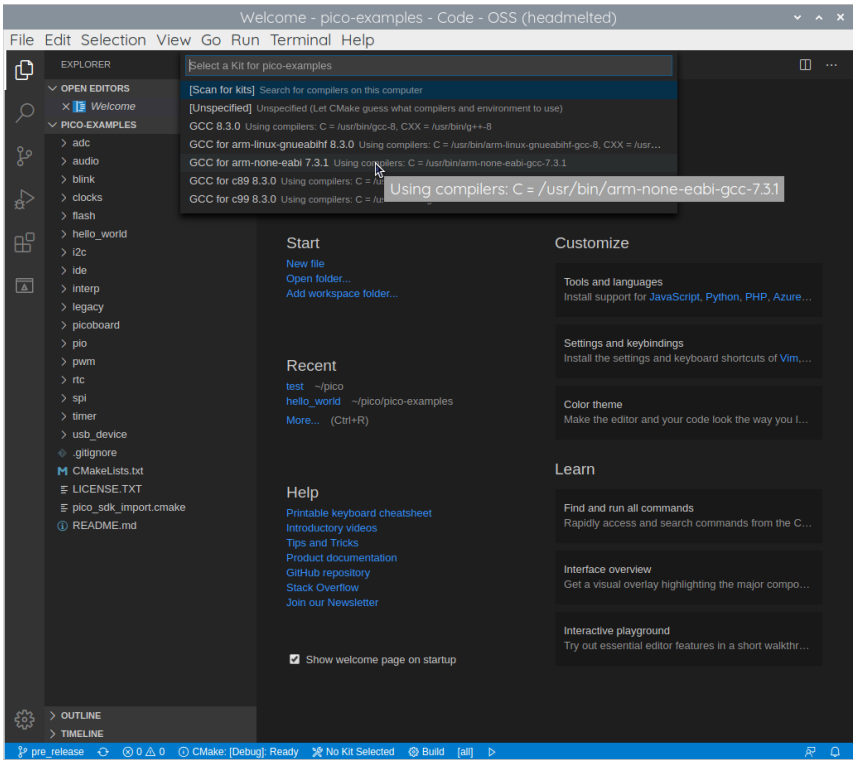


Figure 8. Prompt to choose the correct compiler for the project.

and you should select `GCC for arm-none-eabi` from the drop down menu.

TIP

If you miss the popups, which will close again after a few seconds, you can configure the compiler by clicking on "No Kit Selected" in the blue bottom bar of the VSCode window.

You can then either click on the "Build" button in the blue bottom bar to build all of the examples in `pico-example` folder, or click on where it says "[all]" in the blue bottom bar. This will present you with a drop down where you can select a project. For now type in "hello_world" and select the "Hello World" executable. This means that VSCode will only build the "Hello World" example saving compile time.

TIP

You can toggle between building "Debug" and "Release" executables by clicking on where it says "CMake: [Debug]: Ready" in the blue bottom bar. The default is to build a "Debug" enabled executable ready for SWD debugging.

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 as we did by hand in Section 3.1, before starting the build itself, see Figure 9.

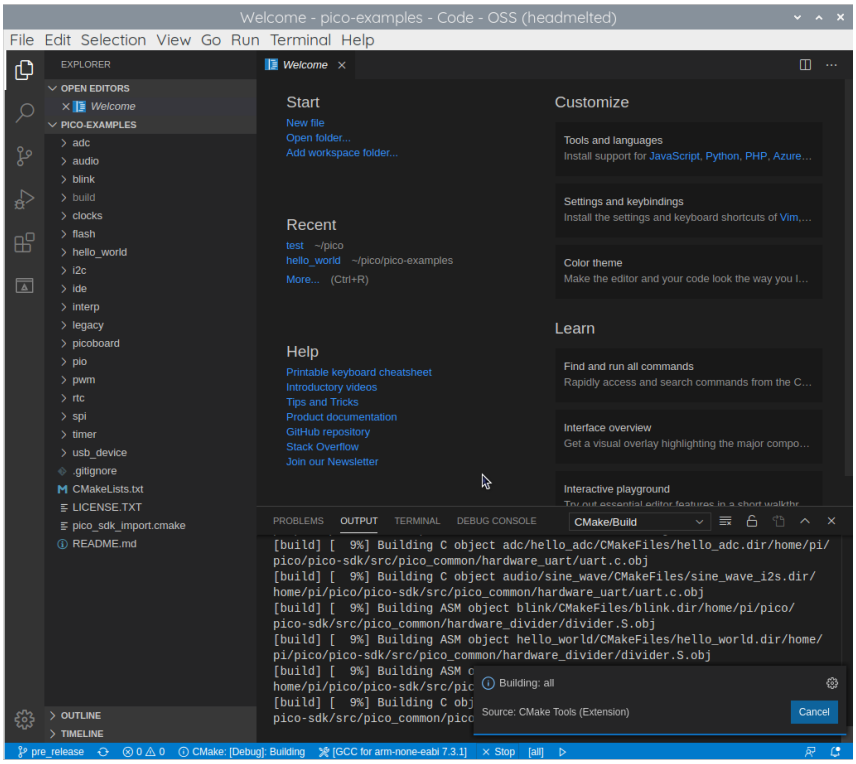


Figure 9. Building the `pico-examples` project in Visual Studio Code

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

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

6.3. Debugging a Project

The `pico-examples` repo contains an example debug configuration that will start OpenOCD, attach GDB, and finally launch the application CMake is configured to build. Go ahead and copy this file (`launch-pi-bitbang.json`) into the `pico-examples/.vscode` directory as `launch.json`.

```
$ cd ~/pico/pico-examples
$ mkdir .vscode
$ cp ide/vscode/launch-pi-bitbang.json .vscode/launch.json
```

NOTE

If the file isn't renamed Visual Studio Code will not be able to find it.

Pico Examples: https://github.com/raspberrypi/pico-examples/tree/pre_release/ide/vscode/launch-pi-bitbang.json Lines 1 - 23

```
1 {
2     // Use IntelliSense to learn about possible attributes.
3     // Hover to view descriptions of existing attributes.
4     // For more information, visit:
  https://go.microsoft.com/fwlink/?linkid=830387
5     "version": "0.2.0",
6     "configurations": [
```

```
        {
            "name": "Pico Debug",
            "cwd": "${workspaceRoot}",
            "executable": "${command:cmake.launchTargetPath}",
            "request": "launch",
            "type": "cortex-debug",
            "servertype": "openocd",
            "device": "RP2040"
            "configFiles": [
                "interface/raspberrypi-swd.cfg",
                "target/rp2040.cfg"
            ],
            "svdFile": "/home/pi/pico/pico-
sdk/src/rp2040/hardware_regs/rp2040.svd",
            "runToMain": true,
        }
    ]
}
```

6.3.1. Running "Hello World" on the Raspberry Pi Pico

IMPORTANT

Ensure that the example "Hello World" code has been as a Debug binary (`CMAKE_BUILD_TYPE=Debug`).

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

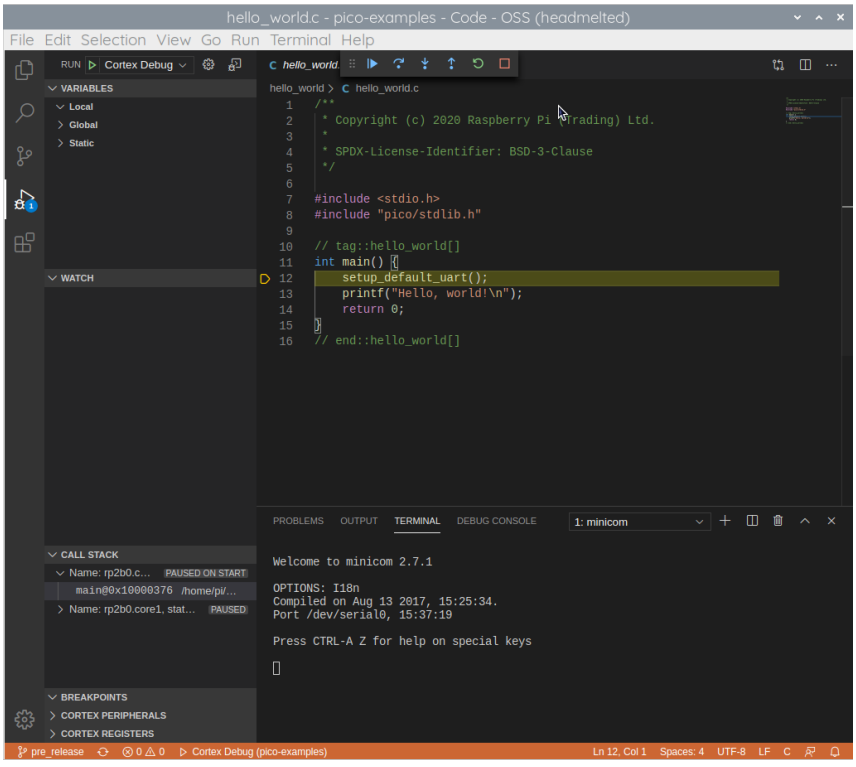


Figure 10. Debugging the "Hello World" binary inside Visual Studio Code

The code should now be loaded and on to the Raspberry Pi Pico, and you should see the source code for "Hello World" in the main right-hand (upper) pane of the window. The code will start to run and it will proceed to the first breakpoint —

enabled by the `runToMain` directive in the `launch.json` file. Click on the small blue arrow (play button) at the top of this main source code window to Continue (`F5`) and start the code running.

TIP

If you switch to the "Terminal" tab in the bottom right-hand pane, below the `hello_world.c` code, you can use this to open `minicom` inside VSCode to see the UART output from the "Hello World" example by typing,

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

at the terminal prompt as we did before, see Section 4.3.

Chapter 7. 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/stdlib.h"
 3 #include "hardware/gpio.h"
 4
 5 const uint LED_PIN = 25;
 6
 7 int main() {
 8     setup_default_uart();
 9     gpio_funcsel(LED_PIN, GPIO_FUNC_PROC);
10     gpio_dir(LED_PIN, GPIO_OUT);
11     while (1) {
12         gpio_put(LED_PIN, 0);
13         sleep_ms(250);
14         gpio_put(LED_PIN, 1);
15         puts("Hello World\n");
16         sleep_ms(1000);
17     }
18 }
```

along with a `CMakeLists.txt` file,

```
cmake_minimum_required(VERSION 3.12)

include(pico_sdk_import.cmake)

project(test_project)

add_pico_sdk_targets()

add_executable(test
    test.c
)

pico_add_extra_outputs(test)
```

```
target_link_libraries(test pico_stdlib)
```

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

```
$ 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
```

This will create a `test.uf2` binary — it can be found in the `build` directory — which you can now drag-and-drop directly onto the RP2040 board which can be mounted as a USB drive.

TO DO: Explain how to create your own project, so need to explain how to create (what should be in) the CMakeLists.txt file in pico-examples/ and pico-examples/hello_world. Talk about CMake in general. Talk about the output prodiucts ELF and .bin, what these are and how they relate to what needs to go on the chip. Talk about UF2 and what that is.

TO DO: you can download RAM binary to Mu via UF2. if there isn't any flash chips attached you can download a binary that runs on the on-chip RAM using UF2. UF2 specifies the addresses of where data goes, so we can load into RAM.. (note the full RAM UF2 is coincidentally an entirely sensible constraint for demo coding competitions) there are some restrictions in what we support which should also be documented (i.e. you basically get to download into either RAM or FLASH not both)... which is of course largely a moot point with toolds like MS's UF2 generator since it only deals with a single flat binary

7.1. 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 5.4. Connect your Raspberry Pi and the Raspberry Pi Pico as in Figure 11.

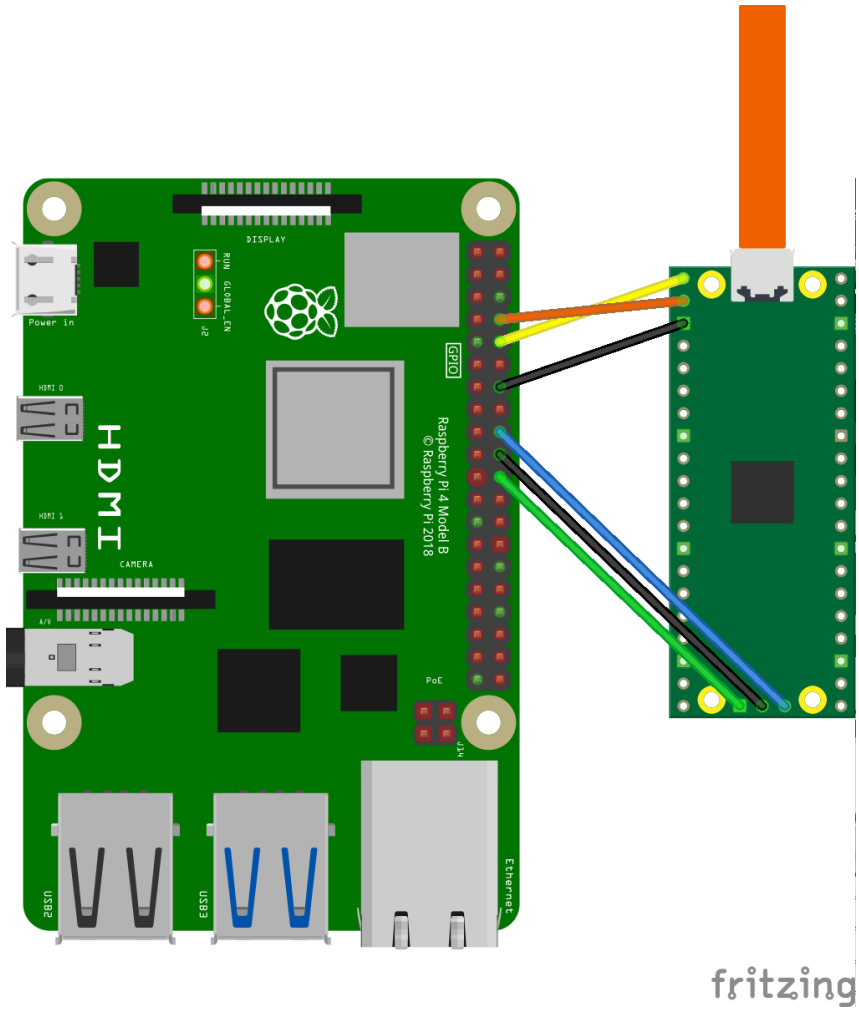


Figure 11. A Raspberry Pi 4 and the Raspberry Pi Pico with UART and SWD debug port connected together. Both are jumpered directly back to the Raspberry Pi 4 without using a breadboard.

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
```

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
```

7.2. 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 6 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. You can do this by simply modifying the example `launch-pi-bitbang.json` file found in the `pico-examples` repository in the `ide/vscode` directory to point to the ELF file in the test project,

TODO: UPDATE BIGBANG PATH. Might as well wait until we've packaged openocd

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Cortex Debug",
            "cwd": "${workspaceRoot}",
            "executable": "./build/test.elf",
            "request": "launch",
            "type": "cortex-debug",
            "servertype": "openocd",
            "device": "Pico2040"
            "configFiles": [
                "/home/pi/pico/openocd/mu-bitbang.cfg"
            ],
            "svdFile": "/home/pi/pico/pico-
sdk/src/rp2040/hardware_regs/rp2040.svd",
            "runToMain": true,
        }
    ]
}
```

and then copying this file into a directory called `.vscode` in the root directory of your project.

7.3. Automating project creation

Some automation has been created which will automatically create 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 git@git.pitowers.org:jamesh/pico_project.git
```

It can then be run in graphical mode,

```
$ cd pico_project
$ python3 pico_project.py -g
```

which will bring up a GUI interface allowing you to configure your project, see Figure 12.

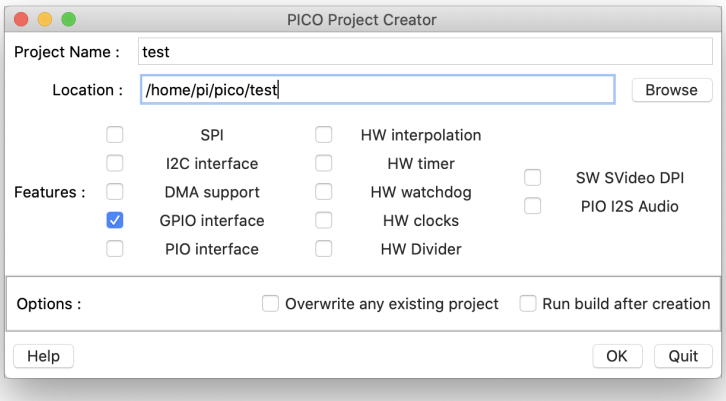


Figure 12. Creating a RP2040 project using the graphical project creation tool.

Alternatively you can create a project from the command line, e.g.

```
$ export PICO_SDK_PATH="/home/pi/pico/pico-sdk"
$ python3 pico_project.py -f gpio -p vscode test
```

Here passing the `-p` option will mean that at `.vcode/launch.json` file is created along with the project.

```
{
  "version": "0.2.0",
  "configurations": [
  {
    "name": "Cortex Debug",
    "cwd": "${workspaceRoot}",
    "executable": "./build/test.elf",
    "request": "launch",
    "type": "cortex-debug",
    "servertype": "openocd",
    "device": "Pico2040"
    "configFiles": [
       "/home/pi/pico/openocd/mu-bitbang.cfg"
       ],
    "svdFile": "/Users/aa/Repos/pico-sdk/src/rp2040/hardware_regs/rp2040.svd",
    "runToMain": true,
```

```
}
```

TO DO: Hopefully the mu-bitbang config will be locatable somehow

This `launch.json` file will have the correct `PICO_SDK_PATH`, but (currently) the path to `mu-bitbang.cfg` used by `openocd` is hardcoded and you will (probably) need to change it depending on your own set up.

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

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

or from Visual Studio code.

Chapter 8. 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.

8.1. Building on Apple macOS

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

8.1.1. Installing the Toolchain

Installation depends on Homebrew, if you don't have Homebrew installed you should go ahead and 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 arm-none-eabi-gcc
```

However after that you can follow the Raspberry Pi instructions to build code for the RP2040. Once the toolchain is installed there are no differences between macOS and Linux to, so see Section 2.1 and follow the instructions from there to fetch the Pico SDK and build the "Blink" example.

8.1.2. Using Visual Studio Code

The Visual Studio Code (VSCode) is a cross platform environment and runs on macOS, as well as Linux, and Microsoft Windows. Go ahead and download the macOS version, unzip it, and drag it to your Applications Folder.

Navigate to Applications and click on the icon to start Visual Studio Code.

8.1.3. Building with CMake Tools

After starting Visual Studio Code you then need to 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" and click on the entry in the list, and then click on 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 Pico SDK.

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

```
        },
    }
```

Now go to the File menu and click on "Add Folder to Workspace..." and navigate to `pico-examples` repo and hit "Okay". The project will load and you'll (probably) be prompted to choose a compiler, see Figure 13. Select "GCC for arm-none-eabi" for your compiler.

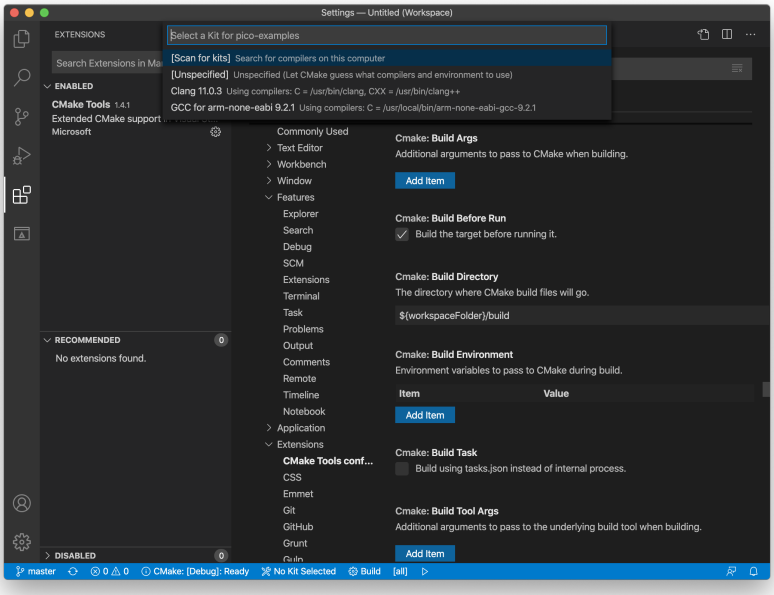


Figure 13. Prompt to choose the correct compiler for the project.

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 as we did by hand in Section 3.1, before starting the build itself, see Figure 9.

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

8.1.4. See "Hello World" UART output

After building the "Hello World" example, see Section 4.1, you need to connect to the the Raspberry Pi Pico standard UART to see the output.

To enable this 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, see Figure 14.

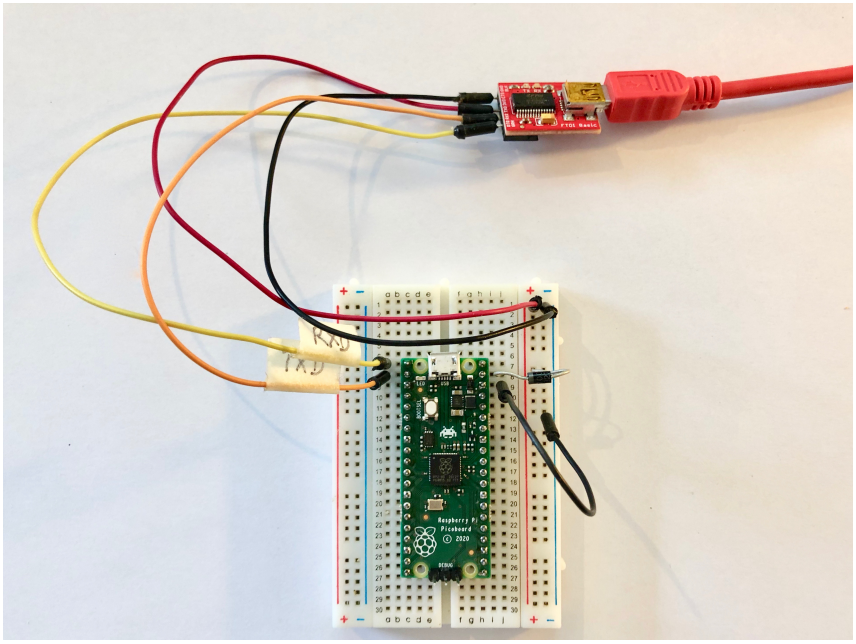


Figure 14. Sparkfun FTDI Basic adaptor connected to the Raspberry Pi Pico

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

Then you should use a Terminal program, e.g. Serial or similar to connect to the serial port. Serial also includes driver support.

8.2. Building on MS Windows

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

8.2.1. Installing the Toolchain

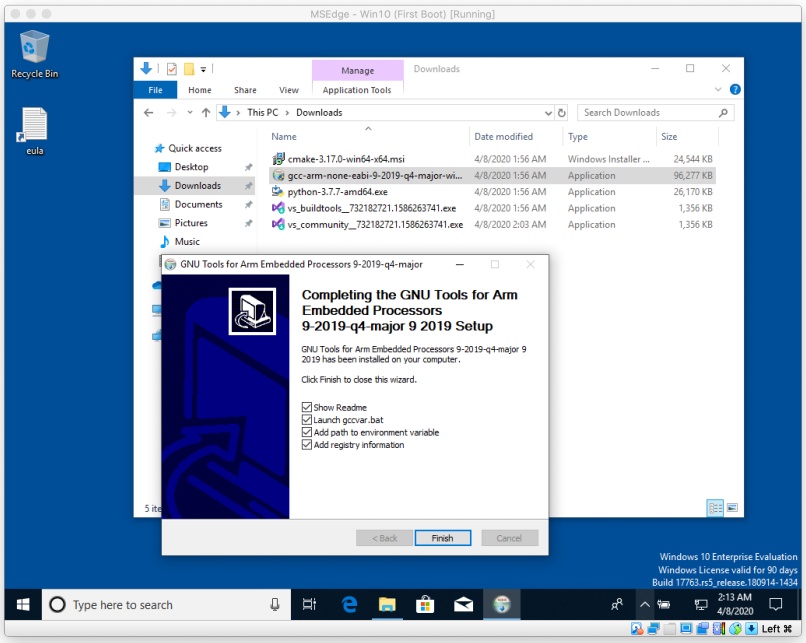
To build you will need to install some extra tools.

- ARM GCC compiler
- CMake
- Build Tools for Visual Studio 2019
- Python 3
- Git (OPTIONAL)

Download the executable installer for each of these, and then go ahead and install all five packages on to your Windows machine.

8.2.1.1. Installing ARM GCC Compiler

Figure 15. Installing the needed tools to your Windows machine. Ensure that you register the path to the compiler as an environment variable so that it accessible from the command line.



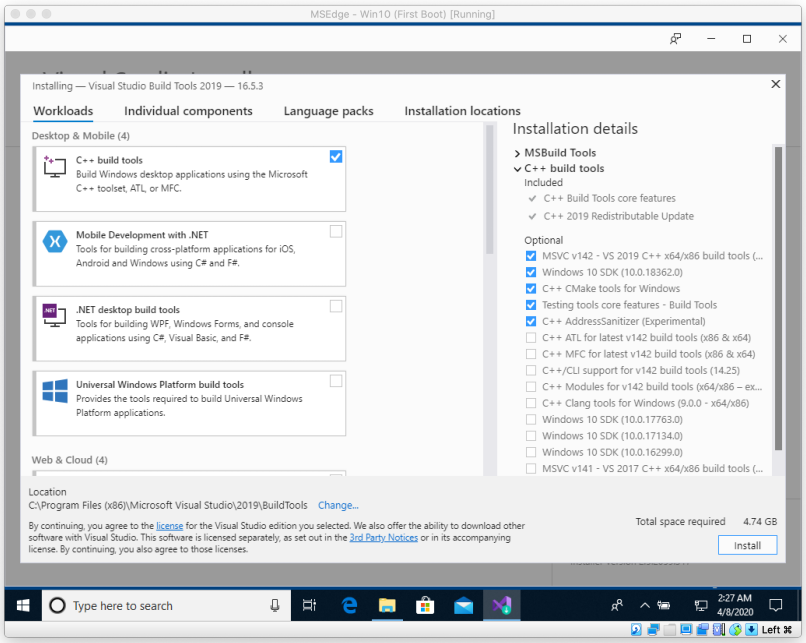
During installation you should tick the box to register the path to the ARM compiler as an environment variable in the Windows shell when prompted to do so.

8.2.1.2. Installing CMake

During the installation add CMake to the system `PATH` for all users when prompted by the installer.

8.2.1.3. Installing Visual Studio Code

Figure 16. Installing the Build Tools for Visual Studio 2019.



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

8.2.1.4. Installing Python 3

During the installation add Python 3.7 to the system `PATH` for all users when prompted by their installers. You should additionally disable the `MAX_PATH` length when prompted at the end of the Python installation.

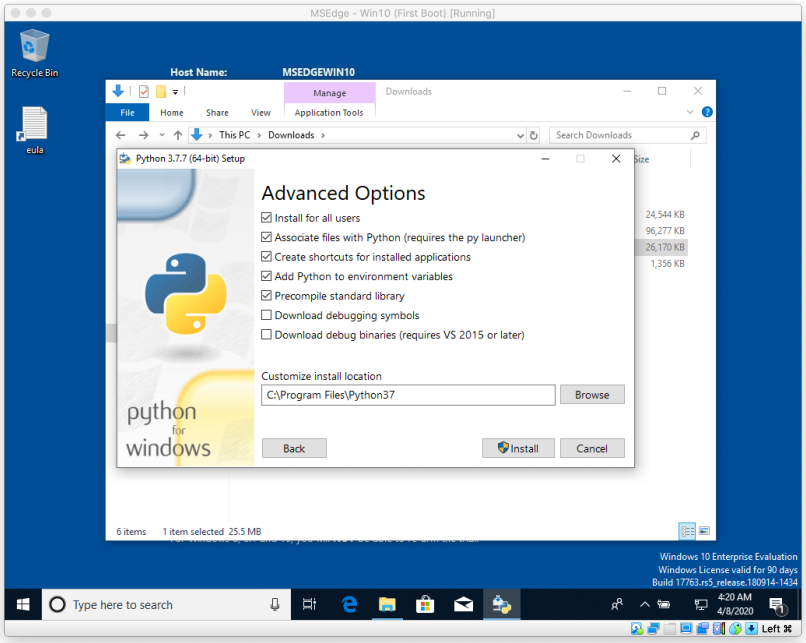


Figure 17. Installing Python 3.7 tick the "Install for all users" box under Advanced Options.

Additionally, when installing Python chose 'Customize installation,' click through 'Optional Features' and then under 'Advanced Features' choose to 'Install for all users'.

NOTE

You may have to make a symbolic link so that the Makefile can find Python 3. To do so type `cmd` in the Run Window next to the Windows Menu to open a Developer Command Prompt Window but select "Run as administrator" in the right hand pane to open the window with administrative privileges. Then navigate to `C:\Program Files\Python37` and make a symlink.

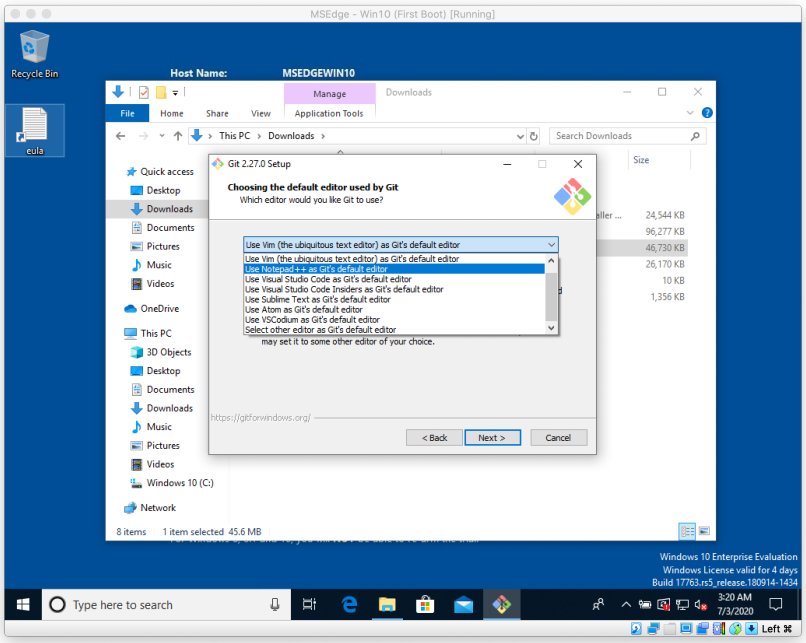
```
C:\Program Files\Python37> mklink python3.exe python.exe
```

This **should** no longer be necessary. However if your build fails because make can't find your Python installation you should add the symlink to the executable. That may resolve things.

8.2.1.5. Installing Git (OPTIONAL)

If you are optionally installing Git you should ensure that you change the the default editor away from `vim`, see Figure 18.

Figure 18. Installing Git



Ensure you tick the checkbox to allow Git to be used from third-party tools 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.

After installing Git you will need to generate an SSH key and upload it to Github.

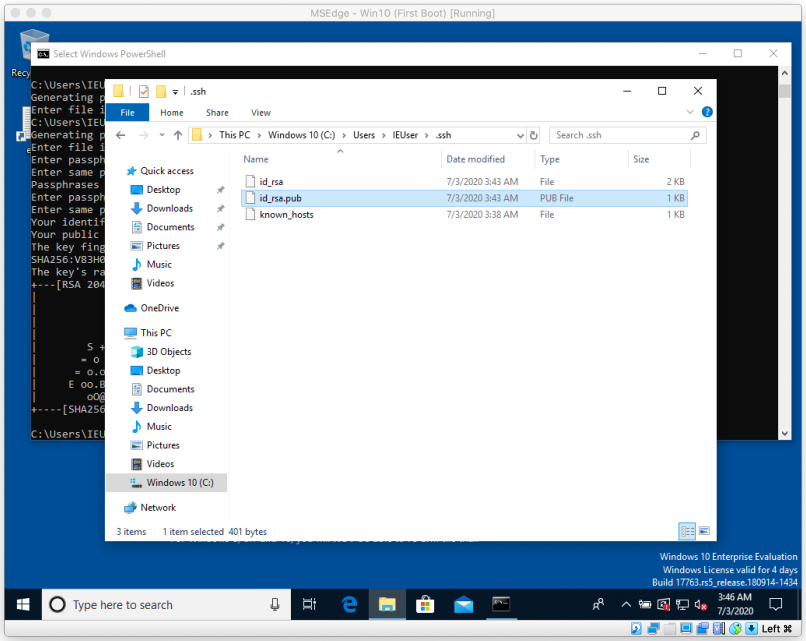
Open a Developer Command Prompt Window from the Windows Menu, by selecting `Windows > Visual Studio 2019 > Developer Command Prompt` from the menu, and generate a key as follows,

```
C:\Users\pico\Downloads> ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\pico/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in C:\Users\pico/.ssh/id_rsa.
Your public key has been saved in C:\Users\pico/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:V83H0HEWXNphLOq+4Jyvbdpo5xYxXtJrObjsjsJ0OKY ieuser@MSEDGEWIN10
The key's randomart image is:
+---[RSA 2048]----+
|              +*B|
|             +.O+|
|            + = +|
|           * o . |
|        S + * o  |
|       = o = =   |
|      = o.o + .  |
|     E oo.BB     |
|        oO@O.    |
+----[SHA256]-----+

C:\Users\pico\Downloads>
```

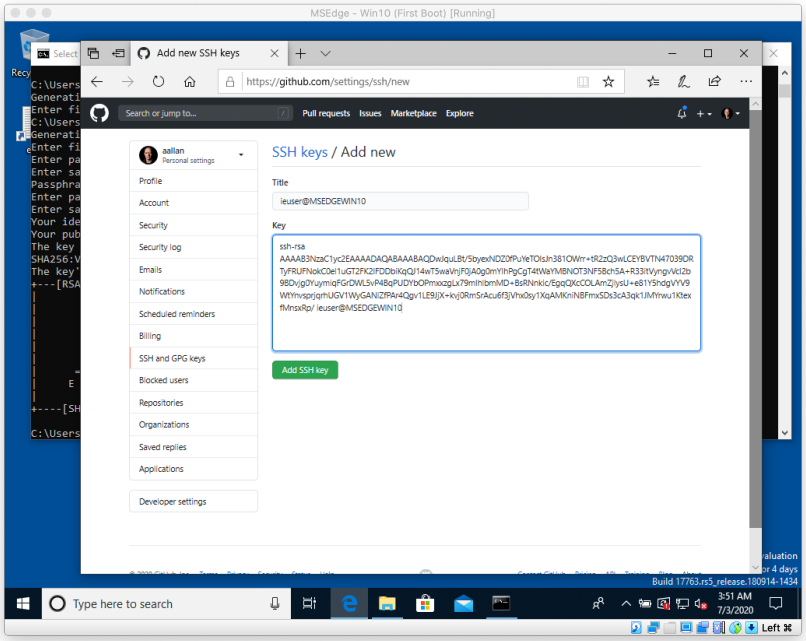
Then navigate to `C:\Users\pico\.ssh` and find your public key file, `id_rsa.pub`,

Figure 19. Finding your `id_rsa.pub` public key file.



and upload this file to your Github account, see Figure 20.

Figure 20. Uploading your `id_rsa.pub` public key file to Github



You should now be able to clone, pull and push to git repositories hosted at Github from the command line.

8.2.2. Getting the Pico SDK and examples

You should go to both the `pico-sdk` and `pico-examples` repositories on Github and download both packages as zip files.

NOTE

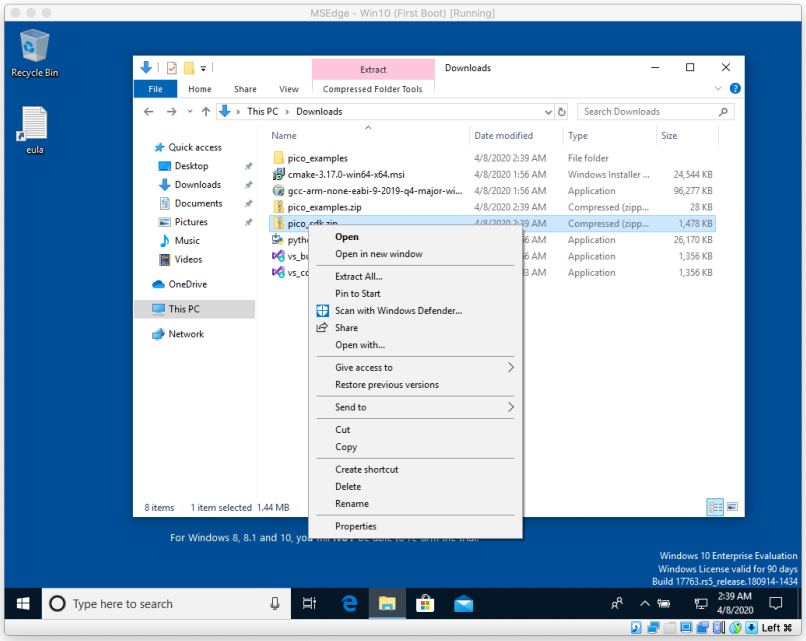
Alternatively if you have Git installed and have uploaded your SSH key to Github, you can go ahead and grab the Pico SDK and examples directly from their git repositories.

```
C:\Users\pico\Downloads> git clone -b pre_release git@github.com:raspberrypi/pico-sdk.git
C:\Users\pico\Downloads> git clone -b pre_release git@github.com:raspberrypi/pico-examples.git
```

8.2.3. Building "Hello World" from the Command Line

Now the necessary infrastructure is installed go ahead and grab both the `pico-sdk`, and `pico-examples` packages from Github and unzip both into your Downloads folder.

Figure 21. Unzipping the Pico SDK



Go ahead and open a Developer Command Prompt Window from the Windows Menu, by selecting `Windows > Visual Studio 2019 > Developer Command Prompt` from the menu.

Then set the path to the Pico SDK as follows,

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

You now need **close your current Command Prompt Window** and open a second 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
```

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

8.2.4. 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 2019 > Developer Command Prompt` 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 the "CMake Tools configuration". Then scroll down to "Cmake: Configure Environment". Click on "Add Item" and add set the `PICO_SDK_PATH` to be `..\..\pico-sdk` as in Figure 22.

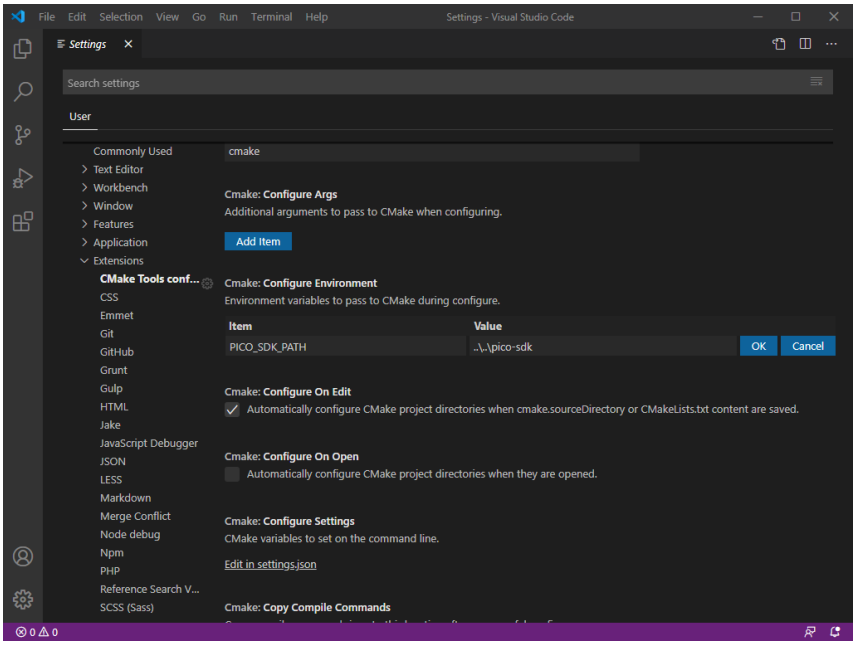


Figure 22. Setting `PICO_SDK_PATH` Environment Variable in the CMake Extension

Now go to the File menu and click on "Open Folder" and navigate to `pico-examples` repo and hit "Okay". You'll be prompted to configure the project, see Figure 23. Select "GCC for arm-none-eabi" for your compiler.

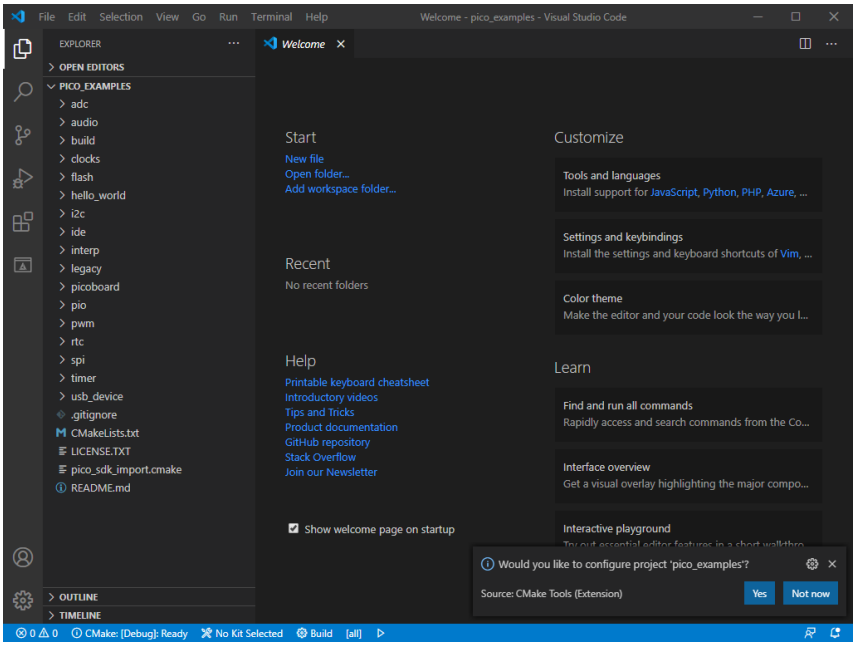


Figure 23. Prompt to configure your project in Visual Studio Code.

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` directory inside the newly created `build` directory. The UF2 binary can be dragged-and-dropped directly onto a RP2040 board attached to your computer using USB.

8.2.5. Flashing and Running "Hello World"

IMPORTANT

The following instructions assume that you are using a Raspberry Pi Pico, some details will differ if you are using a different RP2040-based board.

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 `hello_world.uf2` onto the external drive.

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

Connecting to the board is the same as for a Mac. Use a USB to UART Serial converter, for example a SparkFun FTDI Basic, see Figure 24.

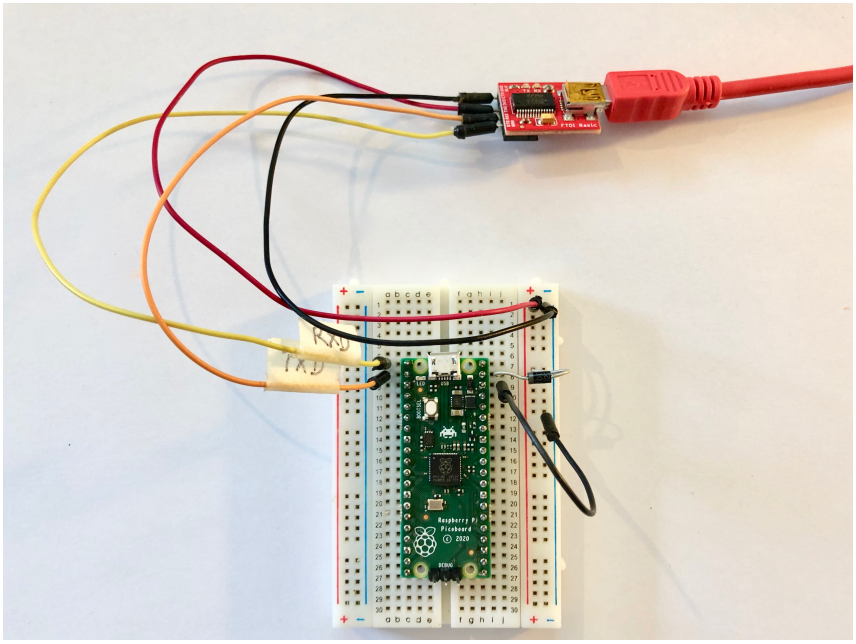


Figure 24. Sparkfun FTDI Basic adaptor connected to the Raspberry Pi Pico

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, hit the "Open" button and you should see the UART output from the Raspberry Pi Pico in your Terminal window.

Chapter 9. Using other Integrated Development Environments

TO DO: Figure out what IDE's are we supporting? We want to let developers use the tools they already do, and putting effort into making sure the experience is good in common IDEs

Currently the recommended Integrated Development Environment (IDE) is Visual Studio Code, see Chapter 6. However other environments can be used with RP2040 and the Raspberry Pi Pico.

9.1. 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 64bit ARM systems, and works well on the Raspberry Pi 4 range (4GB and up) running a 64bit OS. The following instructions describe how to set up Eclipse on a Raspberry Pi 4 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 8.2 and Section 8.1 for more details on non-Linux platforms.

9.1.1. Setting up Eclipse for Pico on Raspberry Pi

Prerequisites:

- Raspberry Pi4 4GB or 8GB
- Fully up to date Raspberry Pi OS
- 64 bit kernel - userland 32 or 64bit

For the 64bit kernel, add the following to the config.txt file

```
arm_64bit=1
```

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.

9.1.1.1. Installing Eclipse

Install an AARCH (64bit ARM) version of Eclipse. The latest drops of this can be found here (https://download.eclipse.org/eclipse/downloads/). These are prerelease versions using the very latest source trees so should be used with caution.

You will need to install the Eclipse CDT (C/C++ development tooling) plugins in to Eclipse.

From the Eclipse `Help` menu, select `Install New Software`.

Select `Add..` to add a new plugins site.

Give the site a name and enter `http://download.eclipse.org/tools/cdt/releases/9.11` for the location

Now select the following items from the list of options.

- CDT Main features - All
- CDT Optional features that are needed
  - C/C++ Debug Adapter GDB Hardware Debugger Integration
  - C/C++ GCC Cross Compiler Support

- C/C++ Remote Launch
- C/C++ GDB Hardware Debugging
- C/C++ Debug Adapter GDB Debugger Integration

Now we need to install the Embedded development plugins.

TO DO: What specific needed here?

9.1.1.2. 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" -D CMAKE_BUILD_TYPE=Debug ../pico-examples
```

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.

9.1.1.3. Building

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

9.1.1.4. 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 Raspberry Pi to the Raspberry Pi Pico prior to running the code. Instructions can be found Chapter 5.

Once OpenOCD is installed and the correct connection made, Eclipse needs to be set up to talk to the 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.

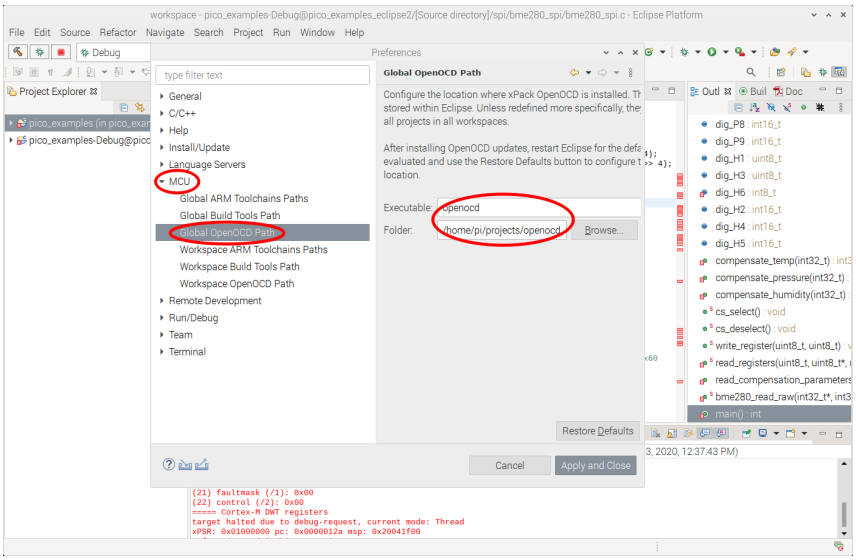


Figure 25. Setting the OCD executable name and path in Eclipse.

9.1.1.5. 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.

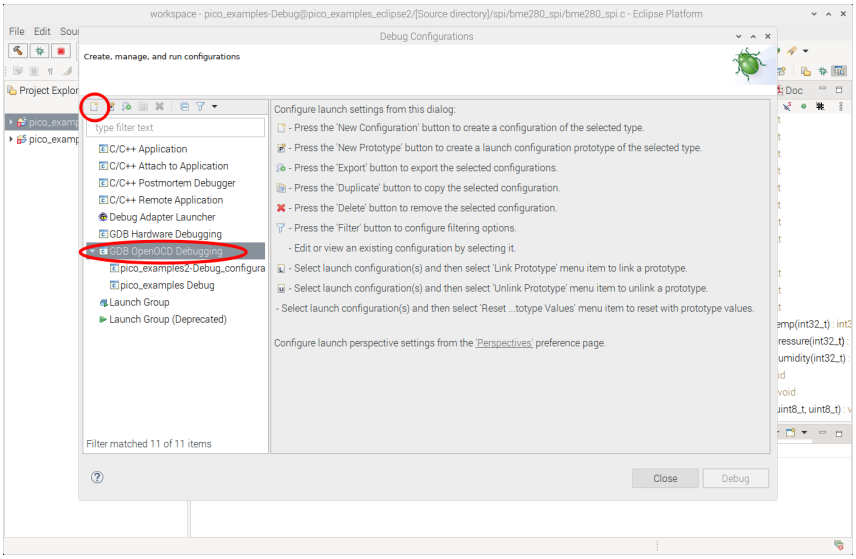


Figure 26. Creating a new Run/Debug configuration in Eclipse.

9.1.1.5.1. 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`

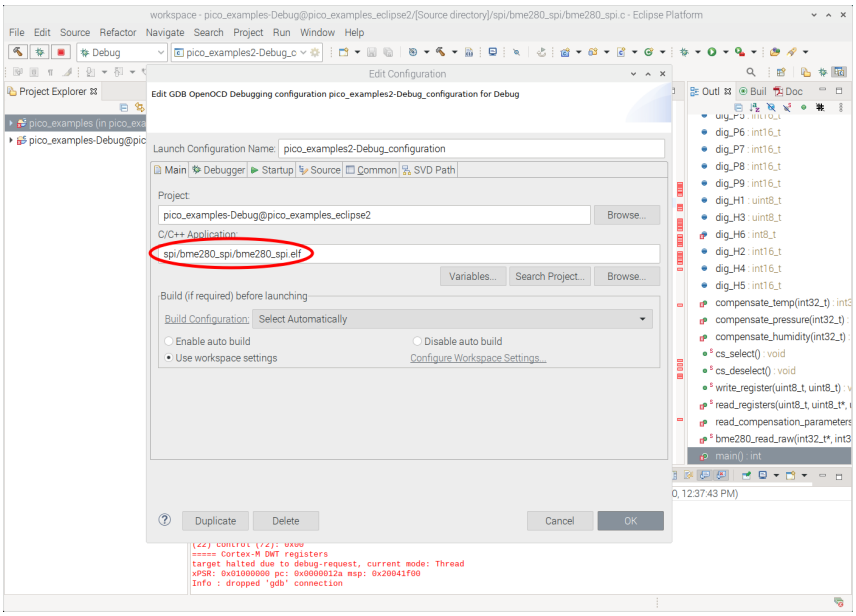


Figure 27. Setting the executable to debug in Eclipse.

9.1.1.5.2. 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 Executable box and Actual Executable box. We also need to set up OpenOCD to use the Pico specific configuration, so in the Config options sections add the following. Note you will need to change the path to point to the location where the Pico version of OpenOCD is installed.

TODO: Fix me including picture `-f ⋯/openocd/mu-bitbang.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 Executable name and Actual Executable fields.

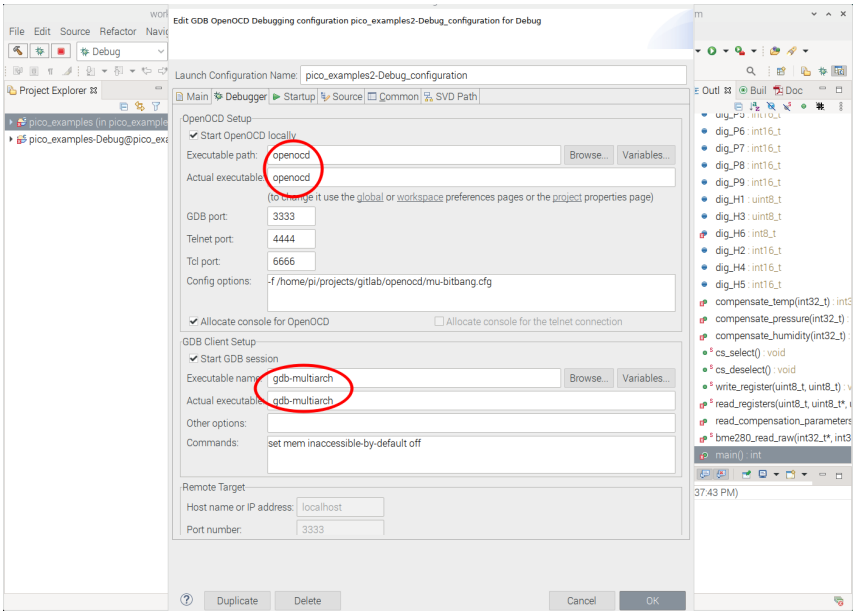


Figure 28. Setting up the Debugger and OpenOCD in Eclipse.

9.1.1.5.3. 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 in to the Eclipse IDE. The SVD plugin comes as part of the Embedded development plugins.

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

E.g.

TODO FIX PICTURE `···/pico-sdk/src/rp2040/hardware_regs/rp2040.svd`

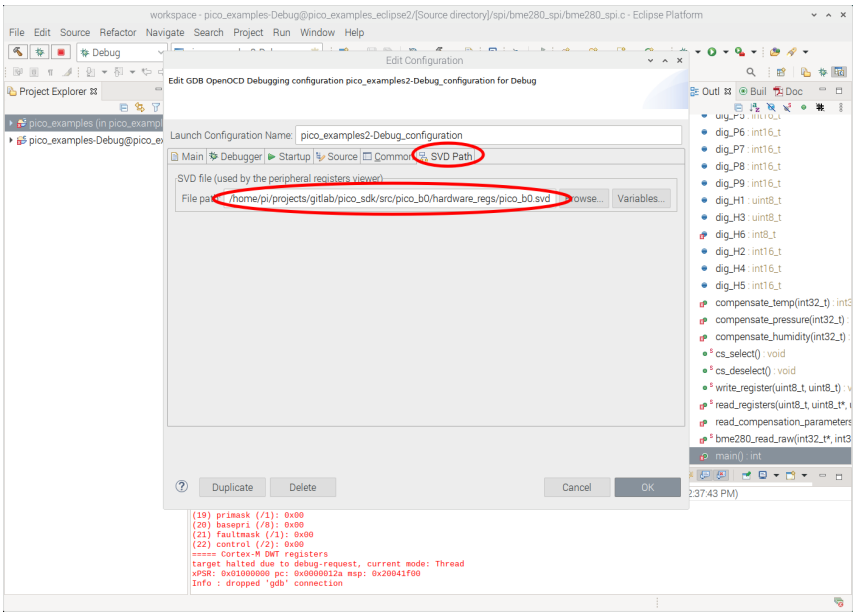


Figure 29. Setting the SVD path in Eclipse.

9.1.1.5.4. 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.

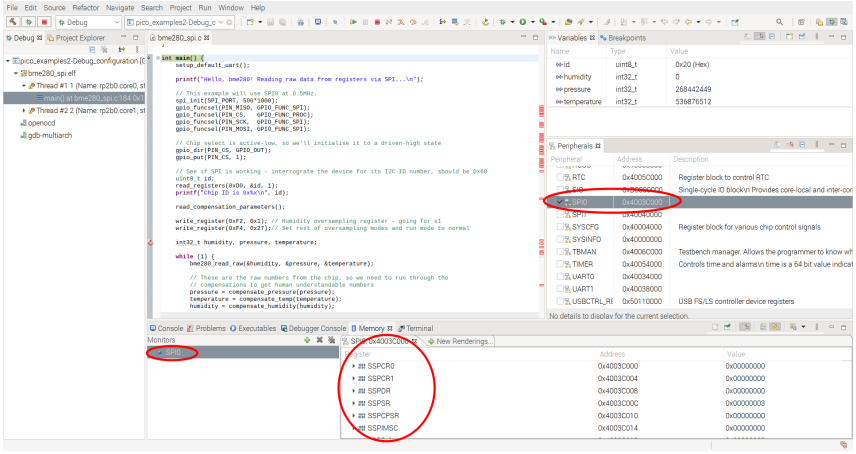


Figure 30. The Eclipse debugger running, showing some of the debugging window available.

9.2. Using CLion

TO DO: This is not free, decide whether we should we mention it? Graham uses this, so we should be fairly easily be able to write some documentation for it

9.3. Using XCode

TO DO: Contact the Tasmanians maybe?

9.4. Other Environments

TO DO: Talk about general support, how you'd go about integrating things into your environment. So talk about CMake project support and SVD support. Perhaps take Code Blocks as our example here, as we've done work to integrate there?

9.4.1. Using openocd-svd

The `openocd-svd` tool is a Python-based GUI utility that gives you access 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 cmsus-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 `bitbang` config.

```
$ openocd -f ~/pico/openocd/mu-bitbang.cfg
```

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 Pico 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 31.

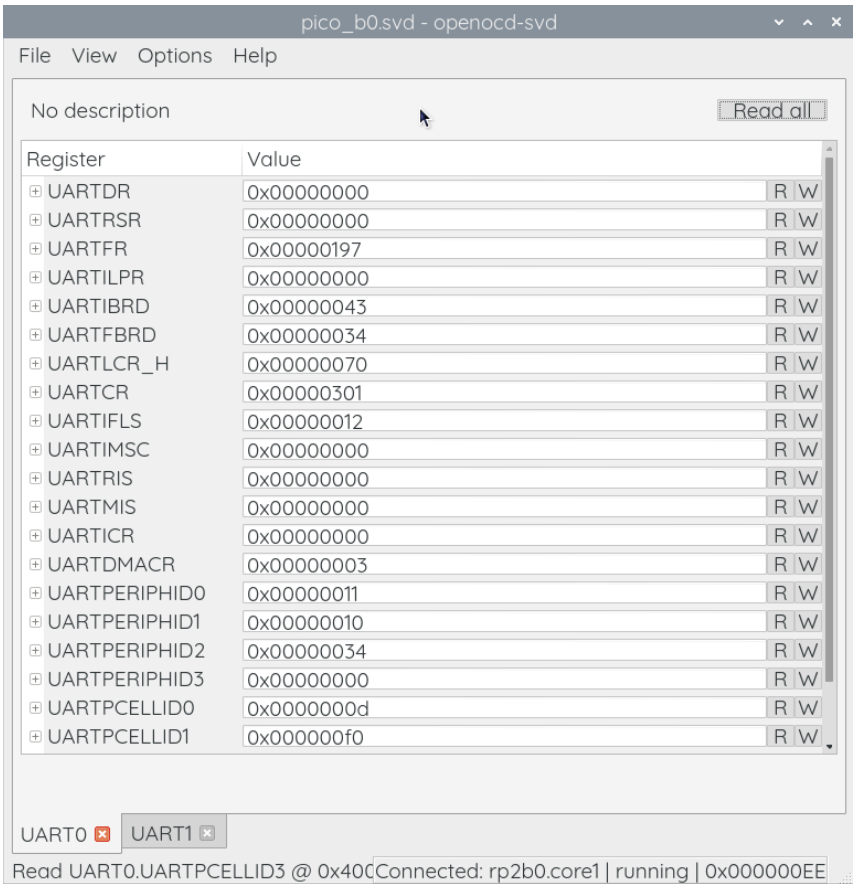


Figure 31. OpenOCD SVD running and connected to the Raspberry Pi Pico.

Appendix A: Using Picoprobe

It is possible to use one Raspberry Pi Pico to debug another Raspberry Pi Pico. This is possible via `picoprobe`, an application that allows a Raspberry Pi Pico to act as a USB -> SWD and UART converter. This makes it easy to use a Raspberry Pi Pico on non Raspberry Pi platforms such as Windows, Mac, and Linux computers where you don't have GPIOs to connect to.

A.1. Build OpenOCD

For picoprobe to work, you need to build openocd with the picoprobe driver enabled.

A.1.1. Linux

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

1. If you are building on a Raspberry Pi you can also pass `--enable-sysfsgpio --enable-bcm2835gpio` to allow bitbanging SWD via the GPIO pins.

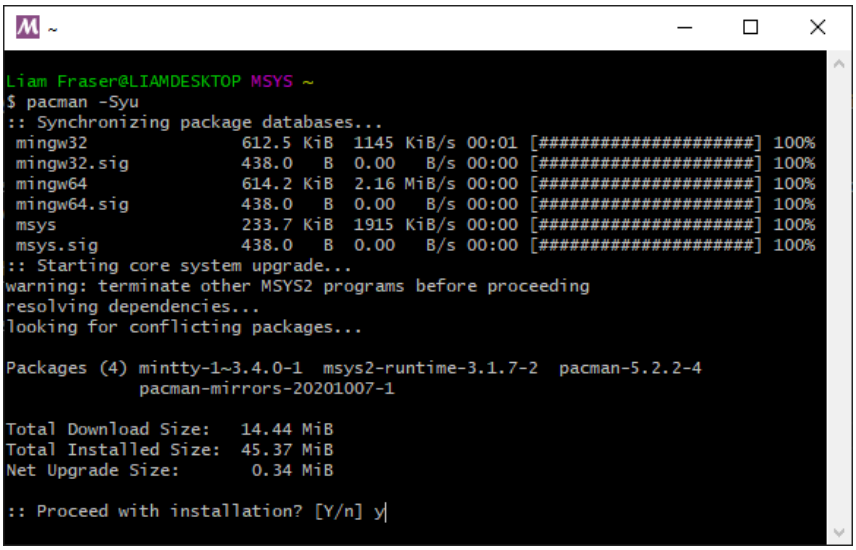
A.1.2. Windows

To make building OpenOCD as easy as possible, we will use MSYS2. To quote their website: "MSYS2 is a collection of tools and libraries providing you with an easy-to-use environment for building, installing and running native Windows software."

Download and run the installer from https://www.msys2.org/.

Start by updating the package database and core system packages with:

```
pacman -Syu
```



If MSYS2 closes, start it again (making sure you select the 64 bit version) and run

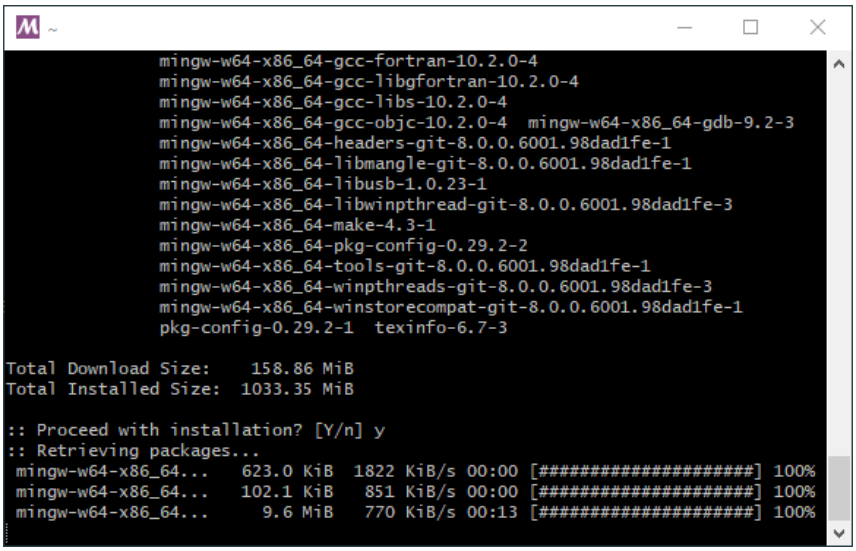
```
pacman -Su
```

to finish the update.

Install required dependencies:

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

Pick all when installing the mingw-w64-x86_64 toolchain by pressing enter.



Close MSYS2 and reopen the 64 bit version to make sure the environment picks up GCC.

```
$ cd ~/pico
$ git clone git@github.com:raspberrypi/openocd.git --branch picoprobe --depth=1
$ cd openocd
```

```
$ ./bootstrap
$ ./configure --enable-picoprobe --disable-werror ①
$ make -j4
```

1. Unfortunately `disable-werror` is needed because not everything compiles cleanly on Windows

Finally run OpenOCD to check it has built correctly. Expect it to error out because no configuration options have been passed.

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

A.1.3. Mac

Install brew if needed

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

Install dependencies

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

1. The version of `texinfo` shipped with OSX is below the version required to build OpenOCD docs

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

1. Put newer version of texinfo on the path

2. Unfortunately `disable-werror` is needed because not everything compiles cleanly on OSX

Check OpenOCD runs. Expect it to error out because no configuration options have been passed.

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

A.2. Build and flash picoprobe

TODO Linux only instructions: Just provide a UF2 here?

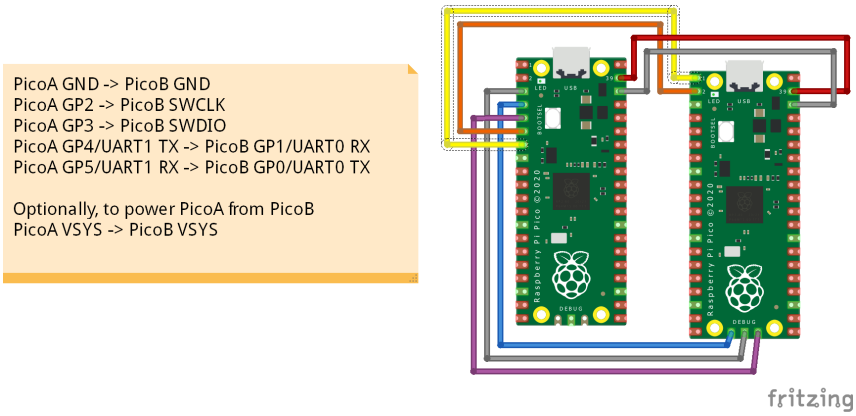
```
cd ~/pico
git clone git@asic-git.pitowers.org:projectmu/picoprobe.git
cd picoprobe
mkdir build
cd build
cmake ..
make -j4
```

Boot the Raspberry Pi Pico you would like to act as a debugger with the `BOOTSEL` button pressed and drag on `picoprobe.uf2`.

A.3. Picoprobe Wiring

```
PicoA GND -> PicoB GND
PicoA GP2 -> PicoB SWCLK
PicoA GP3 -> PicoB SWDIO
PicoA GP4/UART1 TX -> PicoB GP1/UART0 RX
PicoA GP5/UART1 RX -> PicoB GP0/UART0 TX

Optionally, to power PicoA from PicoB
PicoA VSYS -> PicoB VSYS
```



A.4. Install Picoprobe driver (only needed on Windows)

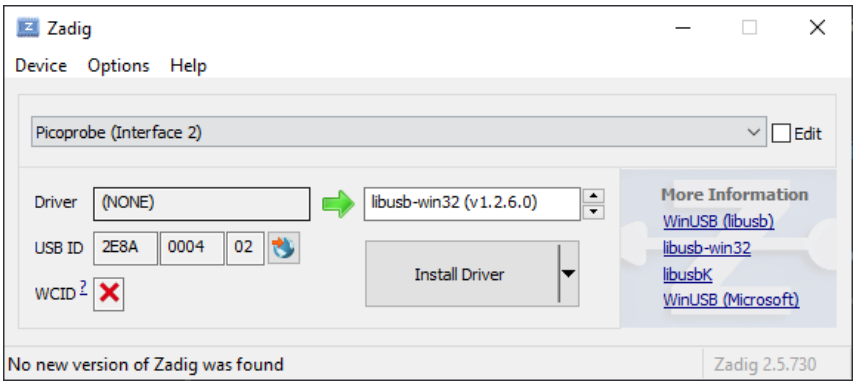
The Picoprobe device has two usb interfaces:

1. A class compliant CDC UART (serial port), which means it works on Windows out of the box
2. A vendor specific interface for SWD probe data. This means we need to install a driver to make it work.

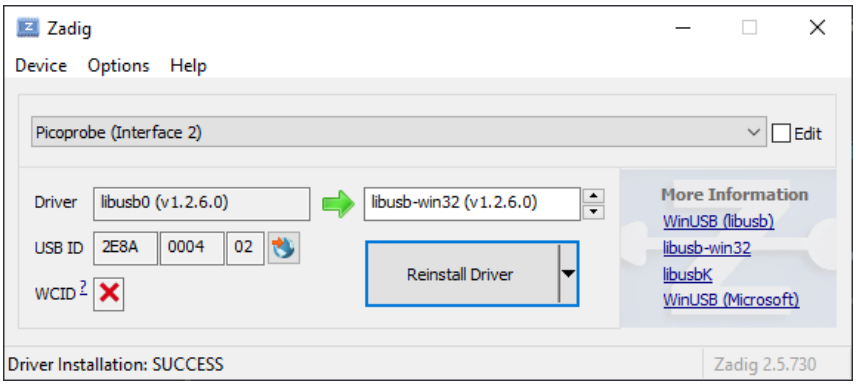
We will use Zadig (http://zadig.akeo.ie) for this.

Download and run Zadig.

Select Picoprobe (Interface 2) from the dropdown box. Select libusb-win32 as the driver.



Then select install driver.



A.5. Using Picoprobe's UART

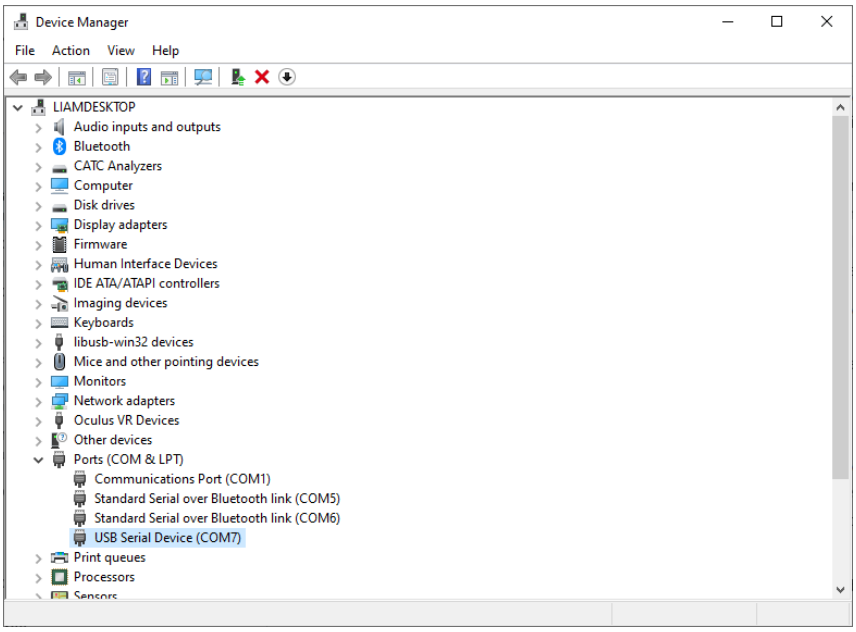
A.5.1. Linux

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

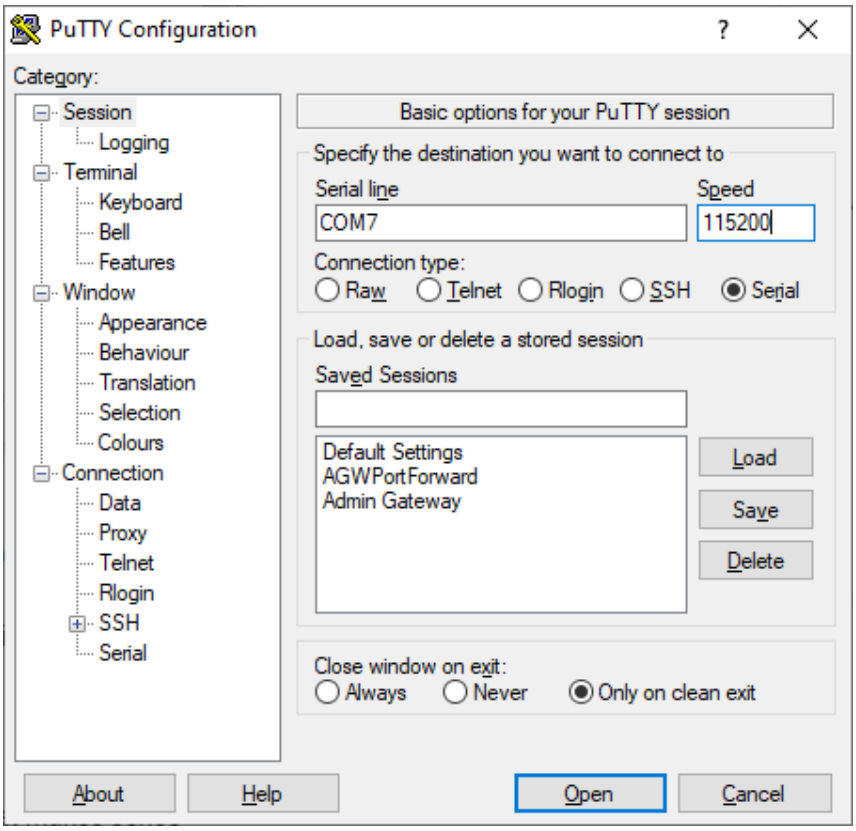
A.5.2. Windows

Download and install PuTTY https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

Open Device Manager and locate Picoprobe's COM port number. 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!

A.5.3. Mac

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

A.6. Using Picoprobe with OpenOCD

Same for all platforms

```
src/openocd -f mu-b0-picoprobe-defaultports.cfg -s tcl
```

Connect GDB as you usually would with

```
target remote localhost:3333
```

Raspberry Pi Trading Ltd