Part 07
-
C/C++ under Windows

Version: 2021-02-17

Windows C/C++ programming

Programming in C or C++ brings you so much closer to the bare metal than an interpreted language like Python can. With the SDK, you're not totally there: compiling a simple Hello, World program, for example, generates a very small binary but it's still a little larger than you might expect. There is a lot of aadditonal code added by the SDK in there too. That's inevitable: this way you don't have to worry about the complexities of installing your code into the correct part of the target chip's memory map and can instead focus on your application code.
For those of us who aren't experienced embedded developers, the SDK is set up to generate .uf2 files so that you can just mount the Pico's internal storage on your computer and drag the binary across, just as you would with MicroPython source. When you do, the Pico reboots, installs the compiled code and runs it.
This approach is convenient for embedded newbies, but it's a hasle having to re-mount your Pico every time you update your code — not to mention the USB connector. The Pico supports a much better alternative: Serial Wire Debug (SWD), accessed through the three pins marked `DEBUG` on the far edge from the USB connector.

These SWD pins can be used for transferring code to the Pico without all that tedious plugging and unplugging of USB cables. More to the point, it is used to enable on-chip debugging so that you can see how your code is operating on the machine and do useful things like set breakpoints so you can pause the program to check the value of variables and such. This is much better way of debugging development code, but I'll cover this is the second part.

Installing the Toolchain

If not installed already, you will need to install the listed below.
Note: I always choose for the 64-bit portable (or .zip) version if availalble except for the Microsoft tools where I choose the installers.
As an example, I'll use the directory `D:\Pico` as the base installation location and the project name will be `PicoTest`, but you can use whatever paths and names you prefer.

- ***git*** from https://git-scm.com/download/win
  Download the Windows 64bit Portable version. It is a selfextracting .exe file. So run the exe file and let it extract in D:\Pico\Git.
  Next extend the system environment variable PATH with the "D:\Pico\Git\bin".
  Note: If you do not know how to do this, see appendix

- ***Cmake*** from https://cmake.org/download/
  Download the Windows 64bit ZIP version. Unzip in D:\Pico\CMake. It will unzip it eg. D:\Pico\CMake\cmake-3.19.4-win64-x64. So drag and drop all files and dirs out of it into D:\Pico\CMake and get rid of the (empty) directory cmake-3.19.4-win64-x64.
  Next extend the system environment variable PATH with the "D:\Pico\CMake\bin".
  Note: If you do not know how to do this, see appendix.

- ***ARM gcc*** from https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads
  Download the Windows 32bit ZIP version. Unzip in D:\Pico\GCC. It will unzip it eg. D:\Pico\GCC\gcc-arm-none-eabi-10-2020-q4-major. So drag and drop all files and dirs out of it into D:\Pico\GCC and get rid of the (empty) directory gcc-arm-none-eabi-10-2020-q4-major.
  Next extend the system environment variable PATH with the "D:\Pico\GCC\bin".
  Note: If you do not know how to do this, see appendix.

- ***Python3*** from https://www.python.org/downloads/
  This is an installer package only. So download and install. When installing Python chose 'Customize installation,' click through 'Optional Features' and then 'Advanced Options' ticked: Install for all users, Associate fileswith Python, Create shortcuts, Add Python to environment variables, Precompile standard libraries.
  Note: if available, you should additionally disable the MAX_PATH length when prompted at the end of the Python installation.
  Once installed, open a command prompt with administrative rights. To do so type cmd in the Run Window but select "Run as administrator" in the right hand pane to open the window with administrative privileges. Then go to the directory where you installed Python (by default eg. C:\Program Files\Python39).

  ```
  cd /D "C:\Program Files\Python39"
  ```

  Now make a symlink so Makefile will be able to find Python3

  ```
  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. That may resolve things.

- ***Microsoft Visual Studio Code*** from https://code.visualstudio.com/download#
  Download the System Installer 64bit and run the installer with default options

- ***Build tools for Visual Studio 2019*** from https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2019
  Download and run the installer. When ask what to install, choose only "C++ build tools" and let it install.

It's a good practice to reboot your computer now.

Install the Pico C/C++ SDK
Open a command prompt and go to Pico directory

```
cd /D D:\Pico
```

Once there, install the SDK this way

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

When done, extend the system environment variables as follows

```
setx PICO_SDK_PATH "D:\Pico\pico-sdk"
```

Close the command prompt and re-open a new one. This is needed to get the added environment variable active in the new command prompt.
Again goto the Pico directory

```
cd /D D:\Pico
```

Now install also the examples

```
git clone -b master https://github.com/raspberrypi/pico-examples.git
```

Test your installation
To make sure all is installed correctly and working, (re)build the examples as follows

```
cd pico-examples
mkdir build
cd build
cmake -G "NMake Makefiles" ..
nmake
```

It is a lengthly process, so take the time to sit back and watch as it rolls over your screen.

When all is completed without problems, go the the next step.

Configure and using Visual Studio Code
Now that we have installed the toolchain and Visual Studio Code we can build our projects inside the that environment rather than from the command line.
Open a Developer Command Prompt Window from the Windows Menu, by selecting `Windows > Visual Studio 2019 > Developer Command Prompt` from the menu. Once it has completed its startup, type

```
code
```

at the prompt. This will open Visual Studio Code with all the correct environment variables set so that the toolchain is correctly configured.
Note: 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`". In the `Settings` pane click on "`Extensions`" and the "`CMake Tools configuration`".
Scroll down to "`Cmake: Configure Environment`". Click on "`Add Item`" and add the `key` "`PICO_SDK_PATH`" and the `value` to "`D:\Pico\pico-sdk`".
Next, tick the option "`Cmake: Configure On Open`" to make it active
Now, scroll down and find "`Cmake: Generator`". Add "`NMake Makefiles`" in the textblock

Important: Before loading the pico-examples into Visual Studio Code, you need to delete the `build` folder from it. So with the Windows file browser, navigate tot `D:\Pico\pico-examples` and delete the `build` folder. Once done, go back to Visual Studio Code

Now go to the `File` menu and click on "`Open Folder`" and navigate to pico-examples directory (`D:\Pico\pico-examples`) and hit "`Select Folder`". You'll be prompted to configure the project. Select "`GCC for arm-none-eabi`" for your compiler.

Go ahead and click on the "`Build`" button (the cog wheel) in the blue bottom bar of the window. This will create the `build` directory, if not already present, 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 `build` directory. The `uf2` binary can be dragged-and-dropped directly onto a RP2040 board attached to your computer using USB.

Now all has been compiling and linking fine, you are ready to start creating your own applications for the Pico.

Set up a project
To setup a new project, you have to do some upfront stuff.
So open a command prompt and navigate to the directory where you want to have the project.
In my case `D:\Pico\Projects`

```
cd /D D:\Pico
mkdir Projects
cd Projects
```

Next, create a project directory

```
mkdir PicoTest
cd PicoTest
```

Then copy the pico_sdk_import.cmake into the directory

```
copy D:\pico\pico-sdk\external\pico_sdk_import.cmake .
```

Now create the CMake file CMakeLists.txt (use any editor you like)

```
notepad CMakeLists.txt

# What CMake to start at
cmake_minimum_required(VERSION 3.12)

# Include the subsidiary .cmake file to get the SDK
include(pico_sdk_import.cmake)

# Set the name and version of the project
project(PicoTest VERSION 1.0.0)

# Link the Project to a source file (step 4.6)
add_executable(PicoTest PicoTest.c)

# Link the Project to an extra library (pico_stdlib)
target_link_libraries(PicoTest pico_stdlib)

# Initalise the SDK
pico_sdk_init()

# Enable USB, UART output
pico_enable_stdio_usb(PicoTest 1)
pico_enable_stdio_uart(PicoTest 1)

# Enable extra outputs (SWD)
pico_add_extra_outputs(PicoTest)
```

Add also the source file for your project

```
notepad PicoTest.c

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


int main()
{
    return 0;
}
```

Open a Developer Command Prompt Window from the Windows Menu and type

```
code
```

From within Visual Studio Code, open the folder `PicoTest`. When asked to select a kit, select `GCC for arm-none-eabi x.y.z.`

With the project in view in Visual Studio Code, you can write some code in your .c file and then you just click `Build` in the status bar at the bottom to compile it. Assuming that completes without error, you'll have a `build` directory under `D:\Pico\Projects\PicoTest\` and inside that a `PicoTest.uf2` that you can drag to the Pico's mounted storage. On a raw board, this will mount automatically. If you have already copied a file over, you'll need to hold down the `BOOTSEL` button while you connect the Pico to a USB port, and then release it.

A basic Pi Pico C project file set
When you go beyond the basic `Hello, World` examples in the manual, you'll start using other Pico libraries. Make sure you add them to the `CMakeLists.txt` file's `target_link_libraries()` call. You can tell when you have forgotten to do this: compiling will fail with an error at one of your `#include` lines.
There's not a 1:1 correspondence between the library name and the relevant `#include`. For example, to use I²C, you add `#include "hardware/i2c.h"` to your source, but the name you add to `target_link_libraries()` is `hardware_i2c`.
The library names are listed in the Pico C/C++ SDK documentation (Chapter 4)

Project Script
Once the toolchain is in place, setting up a project is straightforward. To ease the setup of a project, I added in the appendix a "`makeproject.cmd`" batch file that will setup all the upfront stuff for you. Just run it in the command prompt adding a projectname

```
makeproject PicoTest
```

It will create the folder, add stub source file .c and configure a `CMakeLists.txt` for you.

A more elaborated project generator script (in Python), including a GUI, has been made by Raspberry Pi foundation. BUT … is it targetted for development on Linux. To get it (anyway), open a command prompt and go to `D:\Pico`

```
cd /D D:\Pico
git clone https://github.com/raspberrypi/pico-project-generator.git
```

Now read README.md file and use it with the options you want/need. To run it with the gui, type

```
python3 pico-project.py –gui
```

Examples
The Foundation has a wealth of Pico C programming examples over at GitHub. They cover all of the Pico's hardware features, including its Programmable IO (PIO) system for assembly-style DIY bus programming, so there's plenty there to get you started, whatever your project.

Debugging a Raspberry Pi Pico with a PC, SWD and... another Pico

When you've used Serial Wire Debug (SWD) to help you correct the C or C++ code running on your Raspberry Pi Pico, you'll never want to go back to USB and the UF2 file system again.

The Raspberry Pi Pico is ready for Serial Wire Debugging

SWD uses three pins: one for data, another for a clock signal and a third for ground. It's an ARM-developed technology supported by many MCU designs based on ARM's core architecture. Essentially, it's used to program MCUs and to do on-chip debugging (OCD): to run the code under the control of a remote bug-hunting tool, allowing you to do really useful diagnostic stuff like pause the program mid-flow to check the state of your application.
The Pico breaks out its RP2040 chip's SWD pins as **DEBUG**; just solder on some header pins.

But how do you make use of it? And how do you do so a PC?

We have the Raspberry Pi Foundation to thank for that. It has produced *picoprobe*, a program that runs on the Pico itself and turns it into a pocket SWD-to USB adaptor. It's a two-Pico setup: you have one Pico on which you'll run your code and a second one that operates as a bridge between debugger software running on your Windows computer and the target Pico, accessed through its SWD pins.
Yes, that means you have to sacrifice a Pico to the development process, but since it only cost you a few Euro's, so what? And it's not actually lost for good. *picoprobe* is installed in the usual way: by mounting the host device's storage and dragging a .UF2 file across. And if you can do it once, you can do it again with a different application when you've finished with *picoprobe*.
For me, there's no Earthly reason not to devote a Pico to this role given the benefits: fully interactive on-chip debugging and no faffing around with USB cables.

Build Your toolset for debugging

There's some extra software required, so you'll need to install that, plus a further *Visual Studio Code* extension so that you can run it all from this editor as well as configuring your project to allow debugging. Here are the steps.
As before, I'll assume you're storing everything in a directory whose path is D:\Pico and your project is called PicoTest, but you can change either or both of those values.

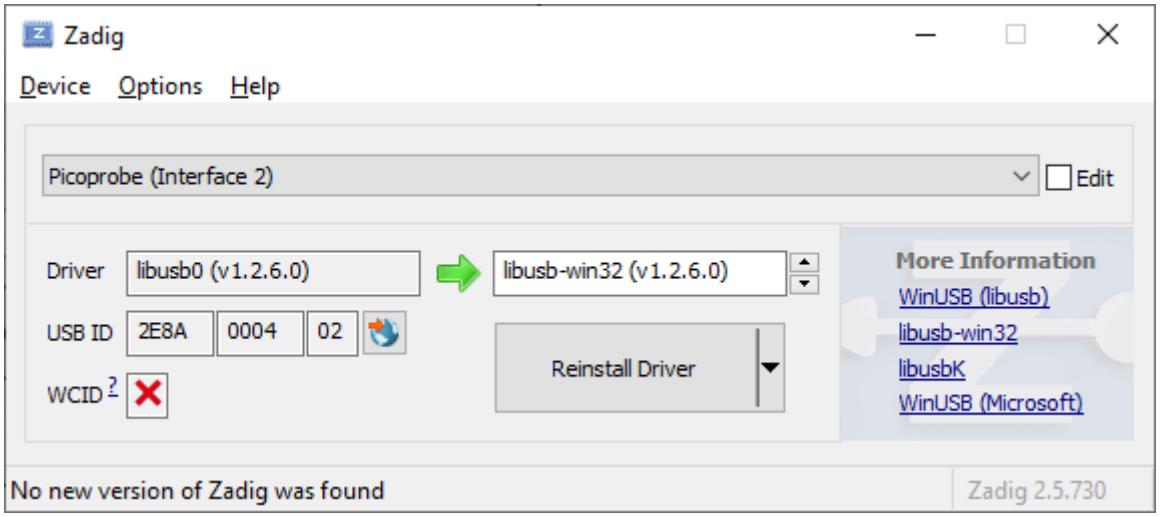
Build and install picoprobe

Open a command prompt and go to `D:\Pico`

```
cd /D D:\Pico
git clone https://github.com/raspberrypi/picoprobe.git
```

Now, open Visual Studio Code using the Developper Command prompt and typing code in it.
In VSC, click on File -> Open Folder and select the picoprobe folder. Select "`GCC for arm-none-eabi`" for your compiler. Once VSC has finished the setup, build the project.
When the building has finished, you find the picoprobe.uf2 file in the build folder.
Mount a Pico by holding down `BOOTSEL` and connect to USB. Release `BOOTSEL`. Drag the file `picoprobe.uf2` to the mounted Pico storage. It will reboot and it is ready to be used.
Keep it connect to your computer via the USB cable as we need immediately

Tweaking USB drivers
As explained in the official guide, you need to make sure that the right USB drivers are being used. Download Zadig from http://zadig.akeo.ie and run it. First select `Options -> List all devices` from the menu. Then you can choose `Picoprobe (Interface 2)` and make sure it's using the `libusb-win32` driver. Install it by hitting '`Replace Driver`'.
Note: `Interface 2` might be eg. `Interface 0` or `Interface 1` on your computer. If you see more than one Picoprobe, do it for all interfaces
When succesfully installed/replaced, you will see '`Reinstall Driver`' instead
Note: you will only see `Picoprobe (Interface 2)` if your Pico with the `picoprobe.uf2` code loaded and running, is attached to your computer with a USB cable.



Install OpenOCD
OpenOCD is the host-side tool that enables on-chip debugging: it manages the communication between MCU and debugger in co-operation with *picoprobe* on the Pico.
On Windows, building OpenOCD is not straightforward. Following the guidelines to build `openocd` in the Foundation documentation won't help you, at least not me.
The official guide describes pulling down a Raspberry Pi provided `openocd` repo and building `openocd` from source with the necessary options to use it with the Pico's RP2040 microcontroller and the Picoprobe. There are a number of confusing aspects to the guide. They tell you to pull down a single branch, but different branches are used depending on whether you are using a Raspberry Pi's GPIO as a debugger or a picoprobe. You also need to set up `msys2` in order to get the tools needed to compile `openocd`. I spent quite a while on this and failed to get it working. Then I spotted a post on Element14 that led me to a pre-built and working `openocd` which you will find here
https://drive.google.com/file/d/1SgZJepJWYQqCeC7m8se-yJLK9sv_AH8n/view.
It's provided by Liam Fraser and his name is all over the commits in the official Pi repo, so no need to worry about where it came from. Just download this and unzip it to an `openocd` folder alongside your Pico SDK. Eg. `D:\Pico\OpenOCD`

GDB
`GDB` is also an essential part of the debugging process, but should have been installed alongside the ARM GCC compiler when you got the main build running.

Testing these from the command line
From a command prompt in the openocd folder, run

```
cd /D D:\Pico\OpenOCD
openocd -f interface/picoprobe.cfg -f target/rp2040.cfg -s tcl
```

and you should see a display similar to the one below.

```
Open On-Chip Debugger 0.10.0+dev-g14c0d0d-dirty (2021-01-27-15:43)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
Warn : Transport "swd" was already selected
adapter speed: 5000 kHz

Info : Hardware thread awareness created
Info : Hardware thread awareness created
Info : RP2040 Flash Bank Command
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
...
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
```

If you get this, then openocd is OK. Leave this running and open a new command prompt. Open a folder where a known good elf file is. Eg: `D:\Pico\Projects\PicoTest\build`. You should find PicoTest.elf in here and run

```
cd /D  D:\Pico\Projects\PicoTest\build
arm-none-eabi-gdb PicoTest.elf
```

to start `gdb`. You should get something similar as output

```
GNU gdb (GNU Arm Embedded Toolchain 10-2020-q4-major) 10.1.90.20201028-git
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-none-eabi".
...
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test.elf...
(gdb)
```

You can proceed with the following commands at the (gdb) prompt to load and run your code

```
target remote localhost:3333
load
monitor reset init
continue
```

Configure Visual Studio Code
Go back to *Visual Studio Code*. Click on the **Extensions** icon in the left side toolbar.
Search for cortex-debug (by marus25) and install it. You only need install *Cortex-Debug. Y*ou don't need the device support extensions.
Note: installing extensions can also be done from the VSC Developper command prompt as follows:

```
code --install-extension marus25.cortex-debug
```

Already installed but just for your information

```
code --install-extension ms-vscode.cmake-tools
code --install-extension ms-vscode.cpptools
```

Cortex-Debug configuration
Within VS Code, select `File / Preferences / Settings...` (or just `Ctrl-,`). Within the settings page look for `Extensions / Cortex Debug Configuration`. Trying to change anything will take you to a json file - probably `C:/Users/(you)/AppData/Roaming/Code/user/settings.json`.
You will need to add the entries for `cortex-debug.gdbPath, cortex-debug.openocdPath,` along the other entries already in that file

```
"cortex-debug.openocdPath": "D:\\Pico\\OpenOCD\\openocd.exe",
"cortex-debug.gdbPath": "D:/Pico/GCC/bin/arm-none-eabi-gdb.exe",
```

Note: As you can see, you can either use / or \\ to separate your directories in a path.

Configure Your Project
In the command prompt, go to `D:\Pico\PicoTest`. Add the directory `.vscode` and create the file `launch.json`.
Note: If you made use of my cmd-script `makeproject.cmd` (see appendix) with the extra command line option `-debug`, this directory with files is created and already present.

```
cd /D D:\Pico\PicoTest
mkdir .vscode
cd .vscode
notepad launch.json
```

```
{
    "version": "0.2.0",
    "configurations": [
        {   "name": "Pico Debug",
            "cwd": "${workspaceRoot}",
            "executable": "${command:cmake.launchTargetPath}",
            "request": "launch",
            "type": "cortex-debug",
            "servertype": "openocd",
            "device": "RP2040",
            "configFiles": [
                "/interface/picoprobe.cfg",
                "/target/rp2040.cfg"
            ],
            "searchDir": ["H:/Electronica/MicroControllers/Pico/OpenOCD/tcl"],
            "svdFile": "${env:PICO_SDK_PATH}/src/rp2040/hardware_regs/rp2040.svd",
            "runToMain": true,
            "postRestartCommands": [
                "break main",
                "continue"
            ]
        }
    ]
}
```

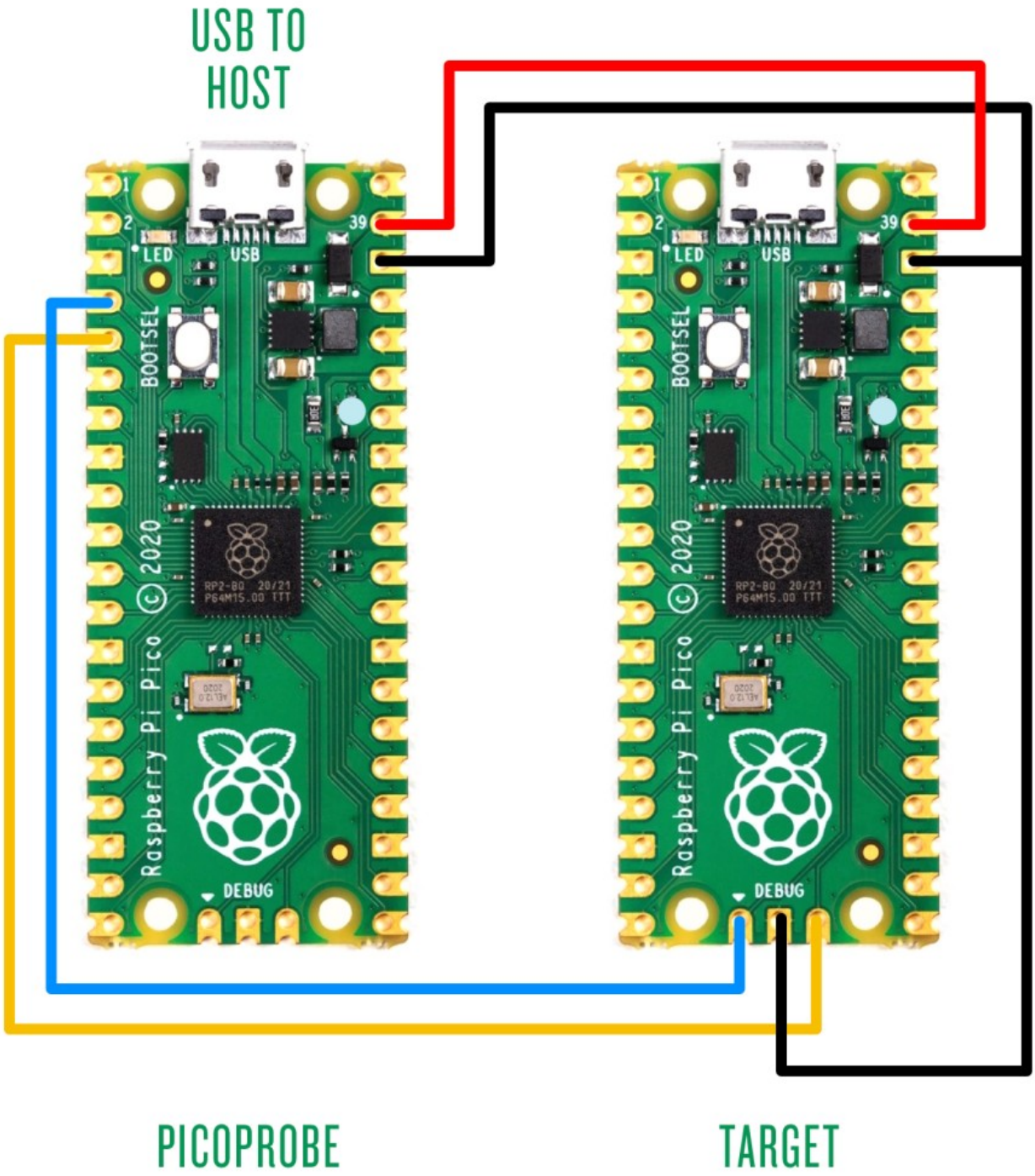
Save this file into `.vscode`. Now add a file `settings.json` as well

```
notepad settings.json

{
    "cmake.buildBeforeRun": true,
    "cmake.configureOnOpen": false,
    "cmake.statusbar.advanced": {
        "debug": {
            "visibility": "hidden"
        },
        "launch": {
            "visibility": "hidden"
        },
    },
    "C_Cpp.default.configurationProvider": "ms-vscode.cmake-tools"
}
```

Save this file into `.vscode`.

Wire up the Hardware
Now for the hardware, just connect the two Picos as shown below. The one with the USB cable, which you hook up to your Windows computer, is the one with picoprobe on it. The other one, the traget, will execute your code. You can power the target Pico from the *picoprobe* unit.

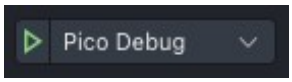


Debug Your Project

Important: If you've already built your code once, delete the existing `build` folder. This will save some toolchain confusion that might impede your progress when you run the debugger. You'll probably have to choose your compiler ('kit') `arm-none-eabi-gcc x.y.z.` again

In Visual Studio Code, open your project folder and let it rebuild the `build` folder.
Hit `Crtl-Shift-D` or click Visual Studio Code's `Run` icon in the left side toolbar.

The debugger will open. If you are asked to specify your target, choose the `.elf` option that matches your project name. Now click the green arrow next to `Pico Debug`.

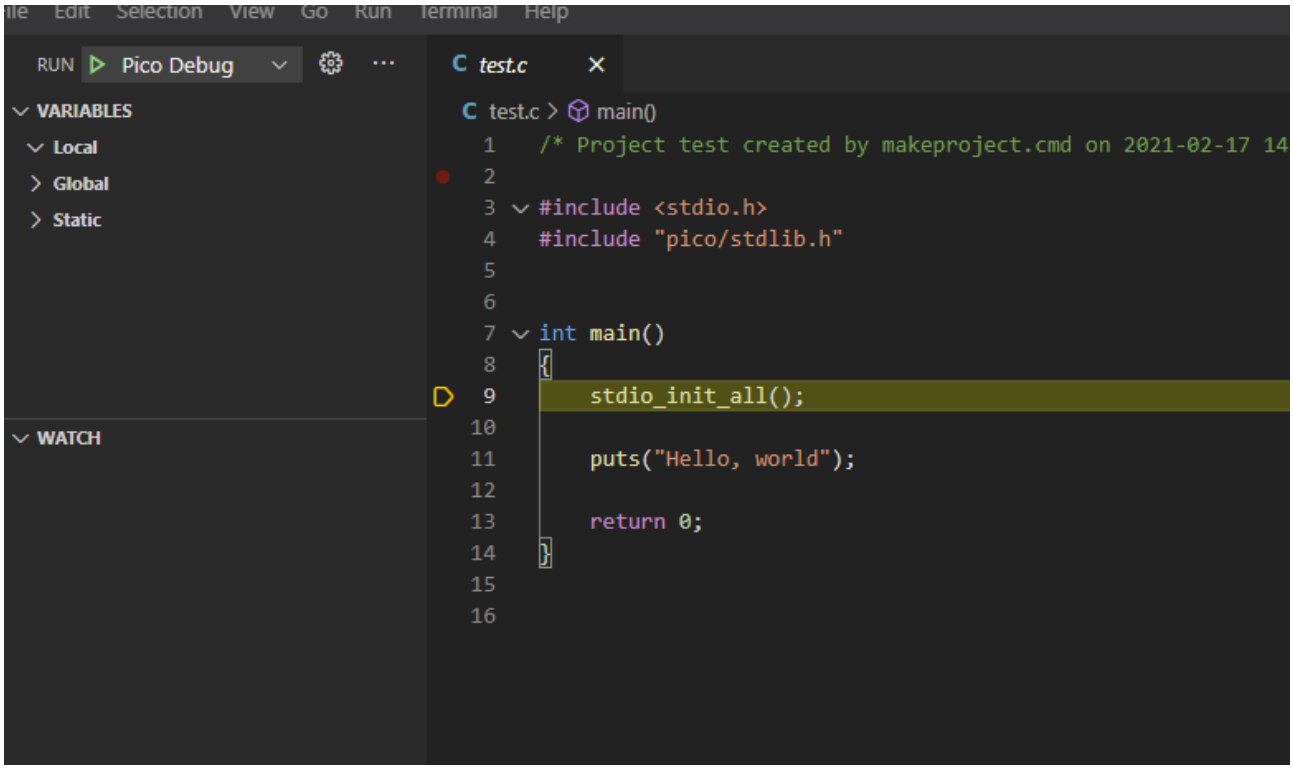


Your code will (re)build, be transferred to the target and run

Click to start debugging...

The debugger will then halt at the start of your program's `main()` function. You control program flow using the play/pause, step over, step in, step out, restart and play buttons right at the top of Visual Studio Code

You can set breakpoints by clicking to the left of the line number identifying a line of code

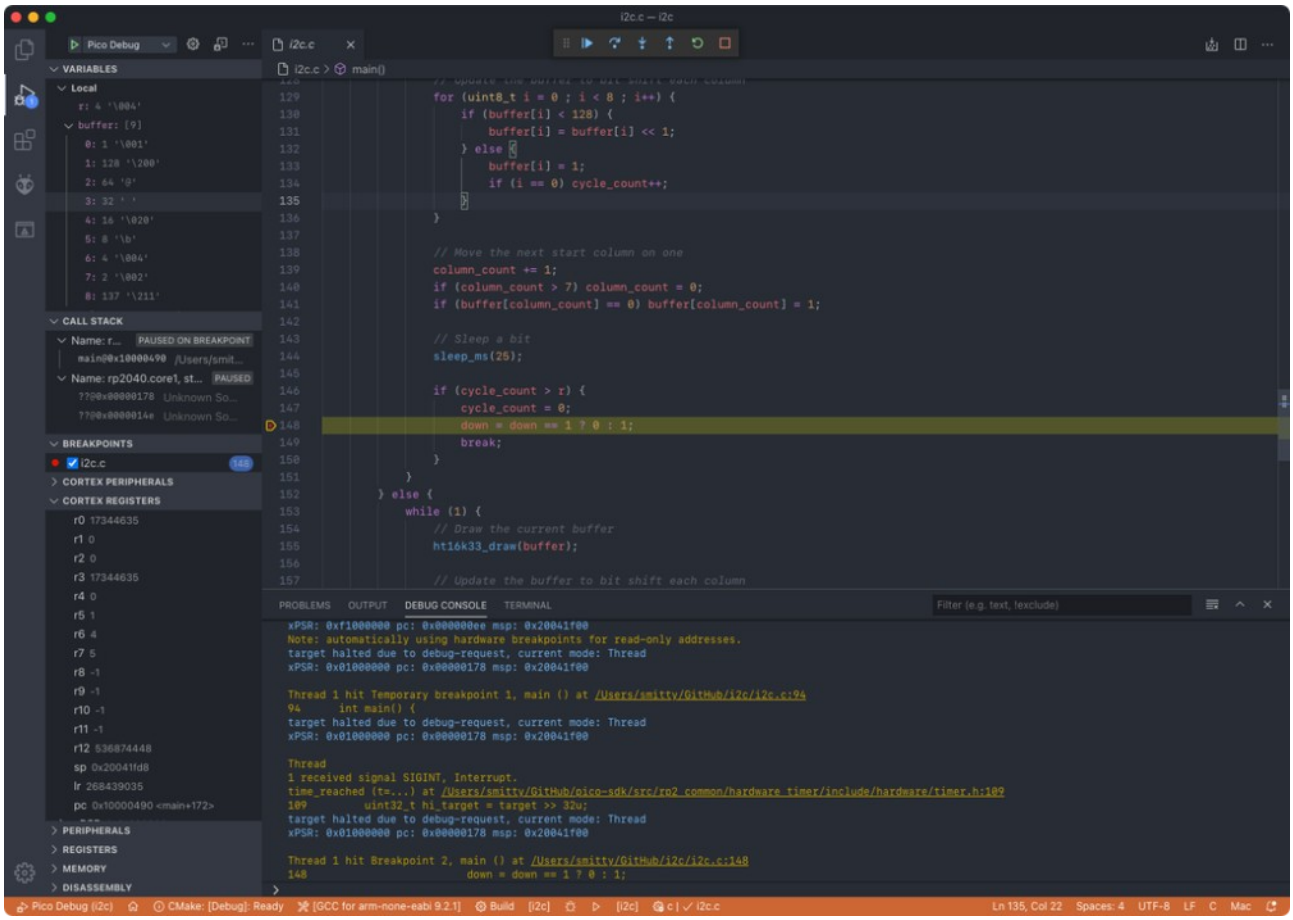


To ellaborate the debugging functions, you'll need a more sofisticate program than this simple "Hello World".
So, if you have a program with variables, functions etc, you can do a lot more with the debugger.

Set breakpoints to pause your program and inspect its state. The debugger will pause execution when it hits a breakpoint and display variables and their values in the left-hand column

The debugger will show you your variables values

Step through the code line by line to see how these values change. At a function call, you can click the `Step In` button to jump into that function's code, or click `Step Over` to run the function and stop again at the next line.



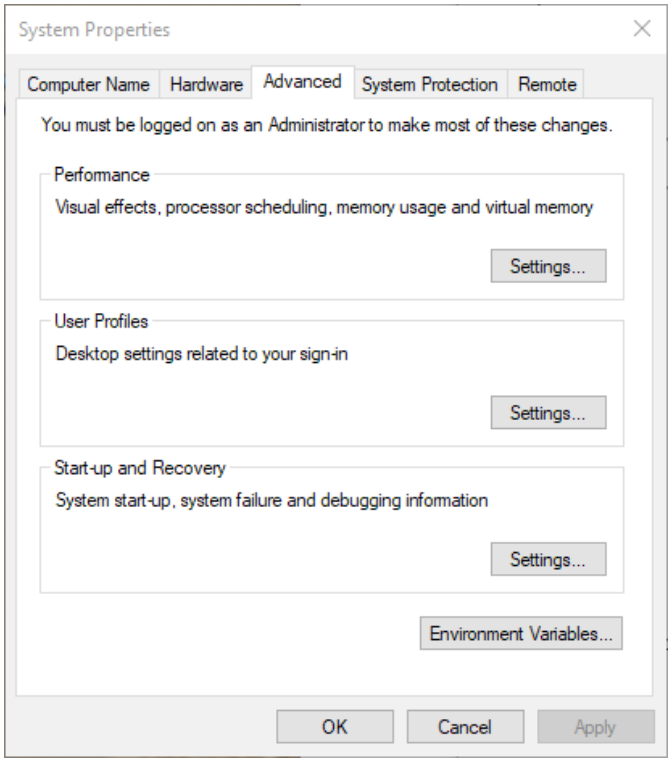
If you make some changes to your code, click the `Stop` button to stop debugging, and then click the green `Pico Debug` arrow again to build and load the updated code, and to begin debugging again.

Troubleshooting
If you see debugging errors, check your wiring. I started out getting nothing back from OpenOCD but DAP Init Failed errors. I traced it down to an incorrectly grounded SWD GND pin. You may need to try different GND pins on the *picoprobe* Pico if this persists.

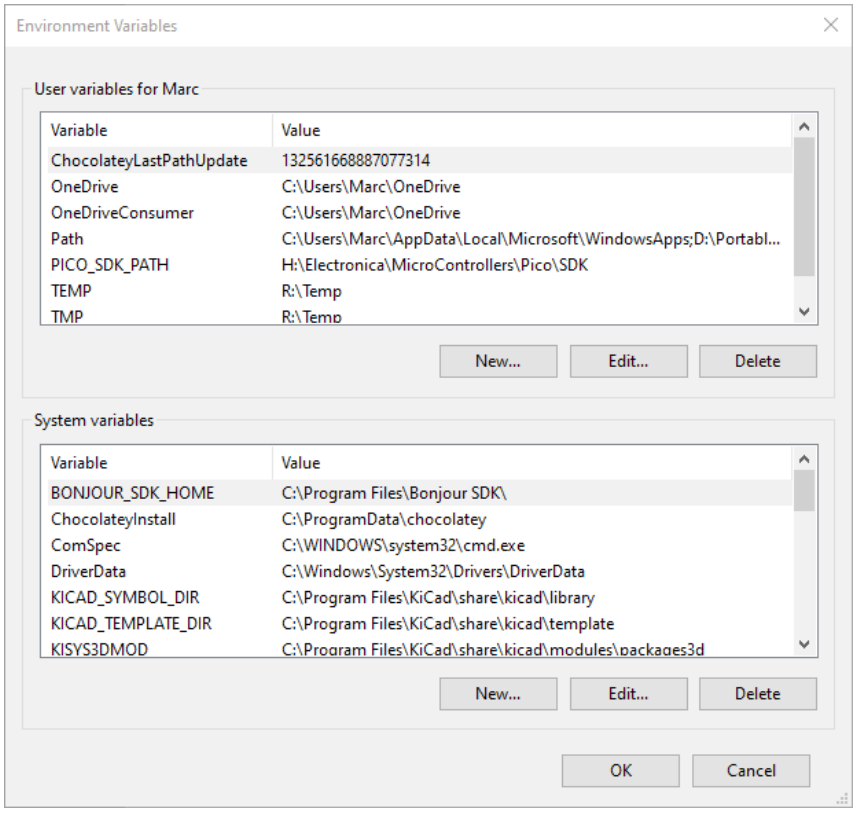
Appendix

Extending the Windows PATH environment variable

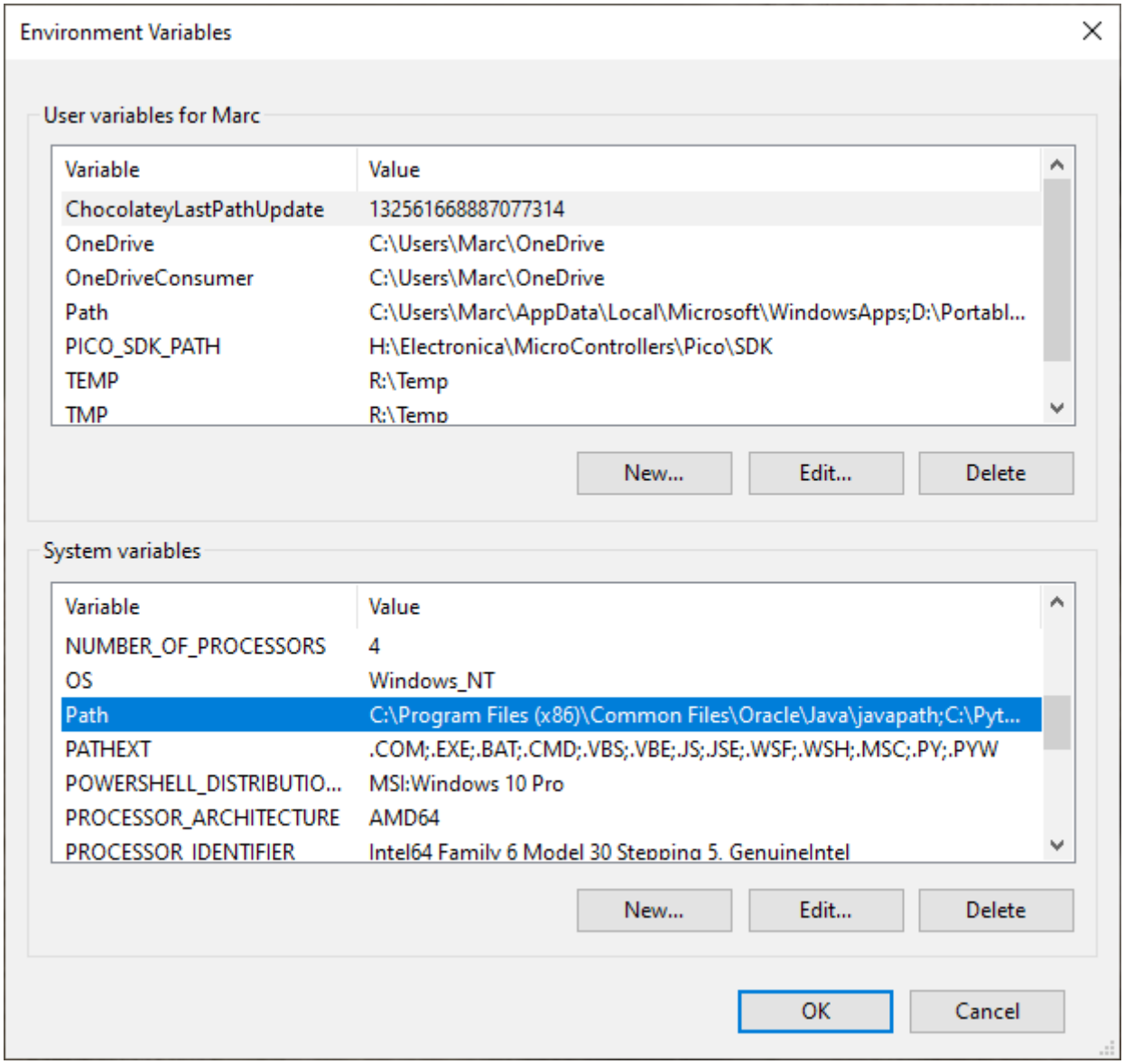
Open Windows Start menu and type '`Control Panel`'. Hit `Enter`. The Control Panel will open and select here '`System`'. Select '`Advanced system settings`' from the right-hand side panel. You should end up seeing this



Now click on the button '`Environment Variables...`'

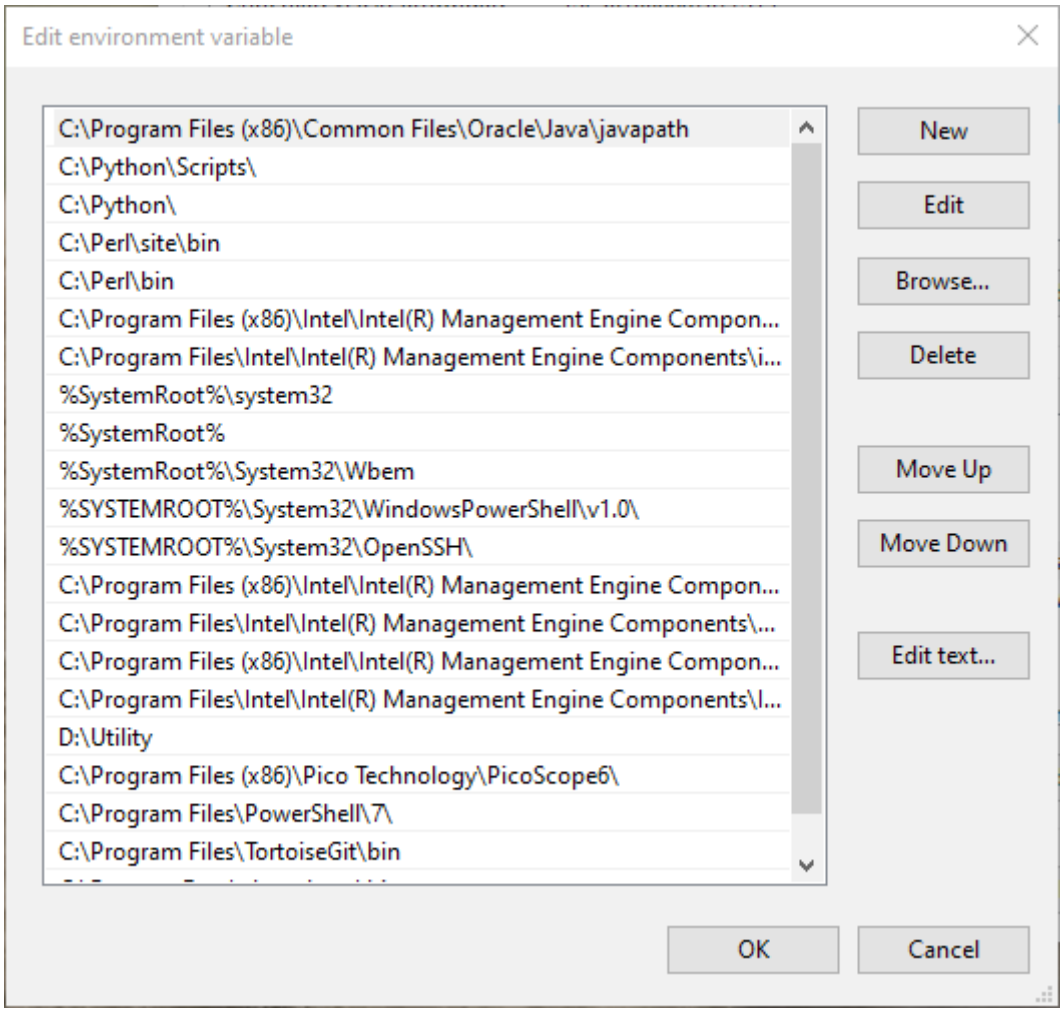


Scroll down in System Variables till you find '`Path`'.



Note: system environment variables are for all users and system wide, while user environment variables are for a specific user only in their own workspace. Windows will load user environment variables first and after them, system variables. So user enivronment variables have priority on system variables.

Click on `Edit`



Now, click on the one of the button on the right to do whatever you want to do (adding, deleting, moving, …).

When done, hit `OK` till you exit out of the whole.

Note: of course, using above, you can change, add, delete any system or user environment variable.

makeproject.cmd

```
@echo off
if "%1" EQU ""     goto :error_missing_project
if "%1" EQU "-gui" goto :gui
set PROJECT=%1

if /I "%2" EQU "-debug" (
  set DEBUG=true
) else (
  set DEBUG=false
)

:start
cls
setlocal enabledelayedexpansion
Title Creating Pico Project %PROJECT%
call :datetime
echo.
if "%DEBUG%" EQU "true" (
  echo %DT% - Creating Project %PROJECT% with debug options
) else (
  echo %DT% - Creating Project %PROJECT% without debug options
)

::Check the path is valid
:checkpath
call :datetime
if exist %PROJECT% (
  echo %DT% - ERROR: could not create path for %PROJECT%
  if exist %PROJECT%\nul (
    echo %DT% -   ^>^>^> %PROJECT% has already a directory. Please check and delete ^<^<^<
    goto :eof
  ) else (
    echo %DT% -   ^>^>^> %PROJECT% is a already file. Please check and delete ^<^<^<
    goto :eof
  )
) else (
  echo %DT% - Creating project directory %PROJECT%
  mkdir %PROJECT%
  if "%DEBUG%" EQU "true" mkdir %PROJECT%\.vscode
)

:make_source_file
call :datetime
echo %DT% - Creating project file %PROJECT%.c ...
echo /* Project %PROJECT% created by %0 on %DT% */ >> %PROJECT%\%PROJECT%.c
echo. >> %PROJECT%\%PROJECT%.c
echo #include ^<stdio.h^>  >> %PROJECT%\%PROJECT%.c
echo #include "pico/stdlib.h" >> %PROJECT%\%PROJECT%.c
echo. >> %PROJECT%\%PROJECT%.c
echo. >> %PROJECT%\%PROJECT%.c
echo int main() >> %PROJECT%\%PROJECT%.c
echo { >> %PROJECT%\%PROJECT%.c
echo     stdio_init_all(); >> %PROJECT%\%PROJECT%.c
echo. >> %PROJECT%\%PROJECT%.c
echo     puts("Hello, world!"); >> %PROJECT%\%PROJECT%.c
echo. >> %PROJECT%\%PROJECT%.c
echo     return 0; >> %PROJECT%\%PROJECT%.c
echo } >> %PROJECT%\%PROJECT%.c
echo. >> %PROJECT%\%PROJECT%.c

::Copy over the .make file from the SDK
set FILE=pico_sdk_import.cmake
call :datetime
echo %DT% - Creating project file %FILE% ...
call :datetime
if exist %PICO_SDK_PATH%\external\%FILE% (
  copy %PICO_SDK_PATH%\external\%FILE% %PROJECT%\%FILE% 1>nul
) else (
  echo %DT% - ERROR: could not copy %PICO_SDK_PATH%\external\%FILE% to %PROJECT%
  goto :eof
)

::Make the CMakeLists.txt file for this project
:make_cmake_file
call :datetime
echo %DT% - Creating CMakeLists.txt...
echo # What CMake to start at >> %PROJECT%\CMakeLists.txt
```

```
echo cmake_minimum_required(VERSION 3.13) >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo set(CMAKE_C_STANDARD 11) >> %PROJECT%\CMakeLists.txt
echo set(CMAKE_CXX_STANDARD 17) >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo # set path to Pico SDK >> %PROJECT%\CMakeLists.txt
echo set(PICO_SDK_PATH "%PICO_SDK_PATH:\=\\%") >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo # Include the subsidiary .cmake file >> %PROJECT%\CMakeLists.txt
echo include(pico_sdk_import.cmake) >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo # Give project details >> %PROJECT%\CMakeLists.txt
echo project(%PROJECT% C CXX ASM) >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo # Link the Project to a source file >> %PROJECT%\CMakeLists.txt
echo add_executable(%PROJECT% %PROJECT%.c) >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo # Link the Project to extra libraries >> %PROJECT%\CMakeLists.txt
echo target_link_libraries(%PROJECT% pico_stdlib) >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo # Initalise the Pico SDK >> %PROJECT%\CMakeLists.txt
echo pico_sdk_init() >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo # Enable/disable stdout and stdin (=stdio) via USB and/or UART output >> %PROJECT%\
CMakeLists.txt
echo # Enable = 1 / disable = 0 >> %PROJECT%\CMakeLists.txt
echo pico_enable_stdio_usb(%PROJECT% 1) >> %PROJECT%\CMakeLists.txt
echo pico_enable_stdio_uart(%PROJECT% 0) >> %PROJECT%\CMakeLists.txt
echo. >> %PROJECT%\CMakeLists.txt
echo # Enable extra outputs (SWD) >> %PROJECT%\CMakeLists.txt
echo pico_add_extra_outputs(%PROJECT%) >> %PROJECT%\CMakeLists.txt


if "%DEBUG%" EQU "false" goto :end

::Make the launch.json file
:make_launch_file
call :datetime
echo %DT% - Creating launch.json...
echo { >> %PROJECT%\.vscode\launch.json
echo     "version": "0.2.0", >> %PROJECT%\.vscode\launch.json
echo     "configurations": [ >> %PROJECT%\.vscode\launch.json
echo         {   "name": "Pico Debug", >> %PROJECT%\.vscode\launch.json
echo             "cwd": "${workspaceRoot}", >> %PROJECT%\.vscode\launch.json
echo             "executable": "${command:cmake.launchTargetPath}", >> %PROJECT%\.vscode\launch.json
echo             "request": "launch", >> %PROJECT%\.vscode\launch.json
echo             "type": "cortex-debug", >> %PROJECT%\.vscode\launch.json
echo             "servertype": "openocd", >> %PROJECT%\.vscode\launch.json
echo             "device": "RP2040", >> %PROJECT%\.vscode\launch.json
echo             "configFiles": [ >> %PROJECT%\.vscode\launch.json
echo                 "/interface/picoprobe.cfg", >> %PROJECT%\.vscode\launch.json
echo                 "/target/rp2040.cfg" >> %PROJECT%\.vscode\launch.json
echo             ], >> %PROJECT%\.vscode\launch.json
echo             "searchDir": ["H:/Electronica/MicroControllers/Pico/OpenOCD/tcl"], >> %PROJECT
%\.vscode\launch.json
echo             "svdFile": "${env:PICO_SDK_PATH}/src/rp2040/hardware_regs/rp2040.svd", >> %PROJECT
%\.vscode\launch.json
echo             "runToMain": true, >> %PROJECT%\.vscode\launch.json
echo             "postRestartCommands": [ >> %PROJECT%\.vscode\launch.json
echo                 "break main", >> %PROJECT%\.vscode\launch.json
echo                 "continue" >> %PROJECT%\.vscode\launch.json
echo             ] >> %PROJECT%\.vscode\launch.json
echo         } >> %PROJECT%\.vscode\launch.json
echo     ] >> %PROJECT%\.vscode\launch.json
echo } >> %PROJECT%\.vscode\launch.json

::Make the settings.json file
:make_settings_file
call :datetime
echo %DT% - Creating settings.json...
echo { >> %PROJECT%\.vscode\settings.json
echo     "cmake.buildBeforeRun": true, >> %PROJECT%\.vscode\settings.json
echo     "cmake.configureOnOpen": false, >> %PROJECT%\.vscode\settings.json
echo     "cmake.statusbar.advanced": { >> %PROJECT%\.vscode\settings.json
echo         "debug": { >> %PROJECT%\.vscode\settings.json
echo             "visibility": "hidden" >> %PROJECT%\.vscode\settings.json
echo         }, >> %PROJECT%\.vscode\settings.json
echo         "launch": { >> %PROJECT%\.vscode\settings.json
echo             "visibility": "hidden" >> %PROJECT%\.vscode\settings.json
echo         }, >> %PROJECT%\.vscode\settings.json
```

```
echo     }, >> %PROJECT%\.vscode\settings.json
echo     "C_Cpp.default.configurationProvider": "ms-vscode.cmake-tools" >> %PROJECT%\.vscode\
settings.json
echo } >> %PROJECT%\.vscode\settings.json

:: all done
:end
call :datetime
if "%DEBUG%" EQU "true" (
  echo %DT% - Created  Project %PROJECT% with debug options
) else (
  echo %DT% - Created  Project %PROJECT% without debug options
)
goto :eof

::------------------------------------------------------------------------------------------------
:error_missing_project
call :datetime
echo.
echo %DT% - ERROR: Missing project name
echo.
goto :eof
::------------------------------------------------------------------------------------------------

::------------------------------------------------------------------------------------------------
:gui
call :datetime
if exist pico_project.py (
  echo.
  echo %DT% - Starting picoproject using gui
  python3 pico_project.py --gui
  echo %DT% - Ending   picoproject using gui
) else (
  echo.
  echo %DT% - ERROR: pico_project.py not present
)
echo.
goto :eof
::------------------------------------------------------------------------------------------------

::------------------------------------------------------------------------------------------------
:datetime
for /f %%x in ('wmic path win32_localtime get /format:list ^| findstr "="') do set %%x
set MONTH=00%MONTH%
set MONTH=%MONTH:~-2%
set DAY=00%DAY%
set DAY=%DAY:~-2%
set HOUR=00%HOUR%
set HOUR=%HOUR:~-2%
set MINUTE=00%MINUTE%
set MINUTE=%MINUTE:~-2%
set SECOND=00%SECOND%
set SECOND=%SECOND:~-2%
set DT=%YEAR%-%MONTH%-%DAY% %HOUR%:%MINUTE%:%SECOND%
exit /b
::------------------------------------------------------------------------------------------------
```