**Blinky with Rust embedded (v1.1), Rust Rover edition**

Oct 2025 ENR325

If you encountered hundreds of problems in VS Code/Rust Analyzer, it’s not your problem. The quickest solution is to use a different IDE. I recommend to use Rust Rover. https://www.jetbrains.com/rust/

(The same developer made PyCharm.)

All the codes have been tested in the Rust Rover without any issue. At the moment it’s free to use Rust Rover with a student/teacher’s edu email account.

I recommend disable/uninstall their AI agent.

1. Prologue

When learning embedded dev 10~20 years ago, you need individual IDE provided by individual chip vendors. You will code with C, with vendor’s complier, debugger hardware, and mostly using Microsoft Windows.

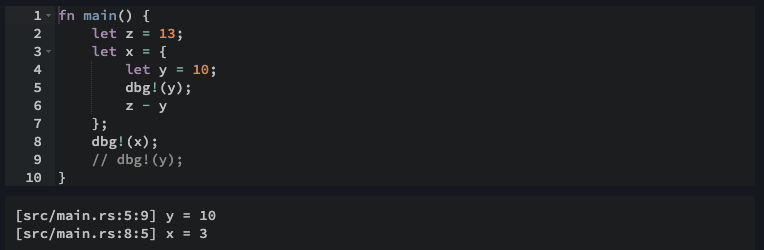
In recent years with LLVM (<https://llvm.org/>) and the philosophy of open source, it is possible to set up embedded dev with anything on anywhere.

Most embedded work will still use C and C++. We are using Rust, a relatively new language (10 years old vs 50 years old C). It has some good community support and who knows? *It might go places.*

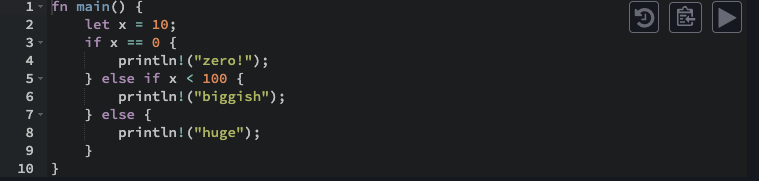
Look up “Tiny Glade” on steam, it was coded with Rust by a two-person dev team.

***Some features that I appreciate about Rust***:

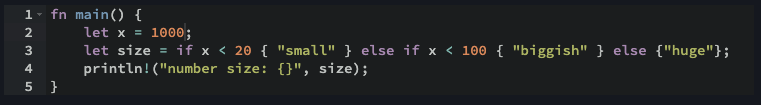
e.g. 1: Rust has built-in macro tools such as dbg!, for debugging. If you run the following code, dbg! will output the state of the variables.



e.g. 2: In Rust the “if” statement is just like “if” in other languages:



But it can also be used as an expression:



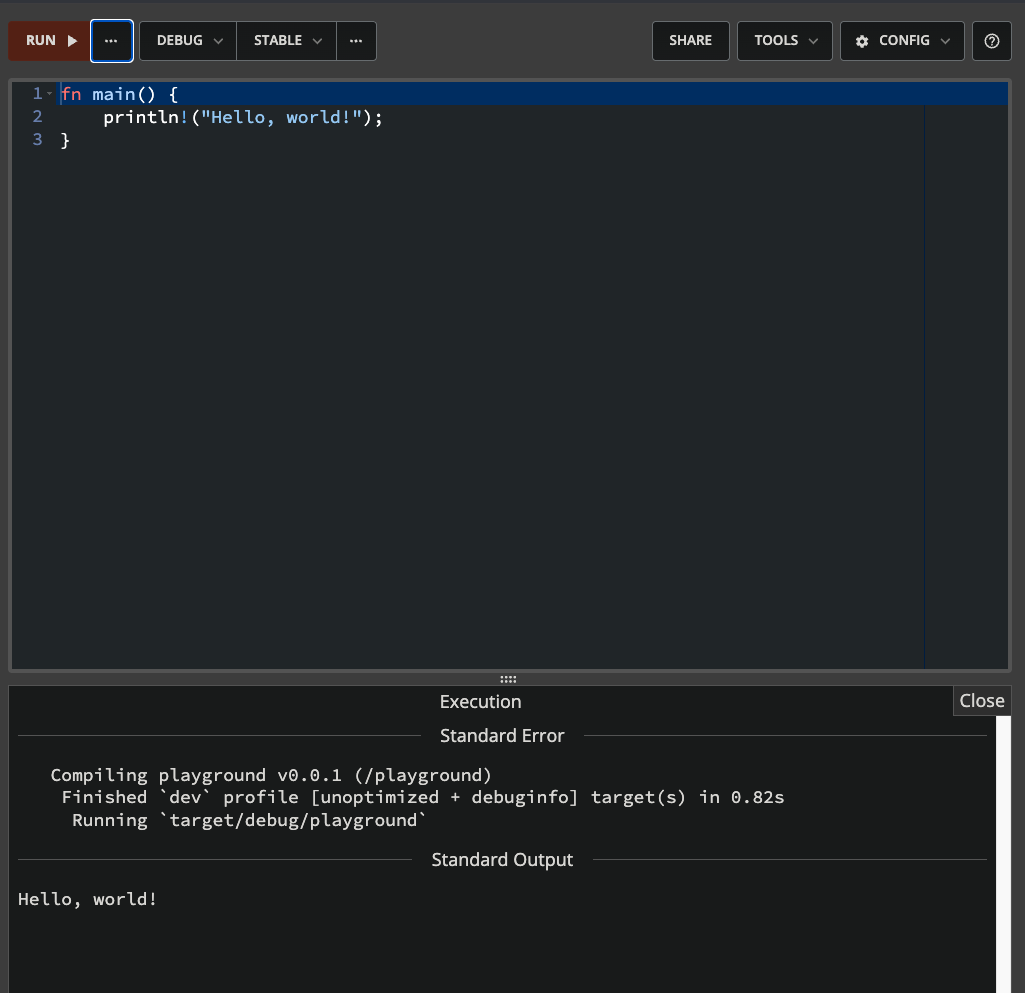
Neet, yeah? Today, we are going to do more “Hello World” in embedded: blink LED lights.

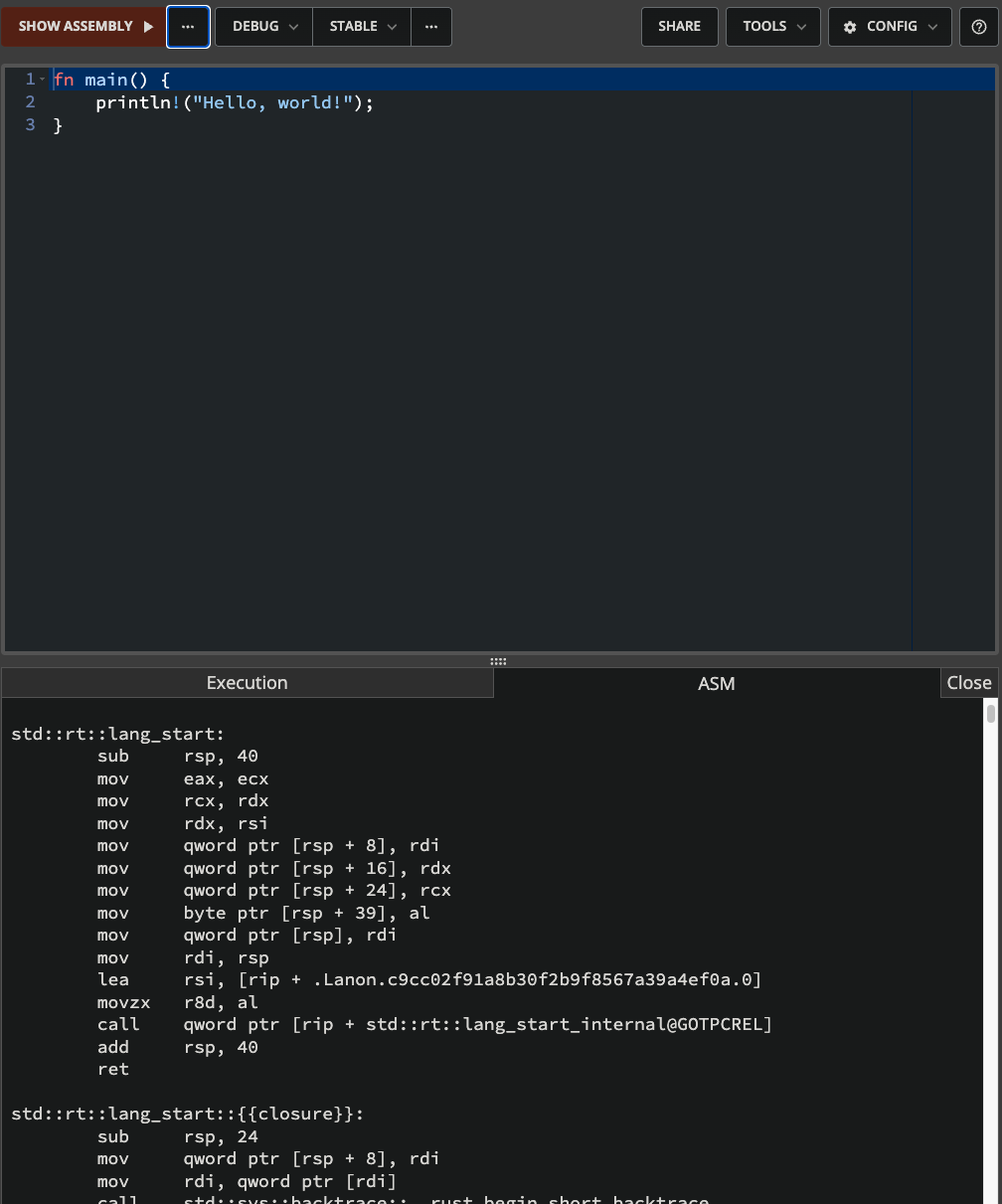
1. **Blink LED, bare metal style**

--- It’s called bare metal because it’s the lowest level of software control on an MCU.

Notes: You can go assembly if you want to go lower.

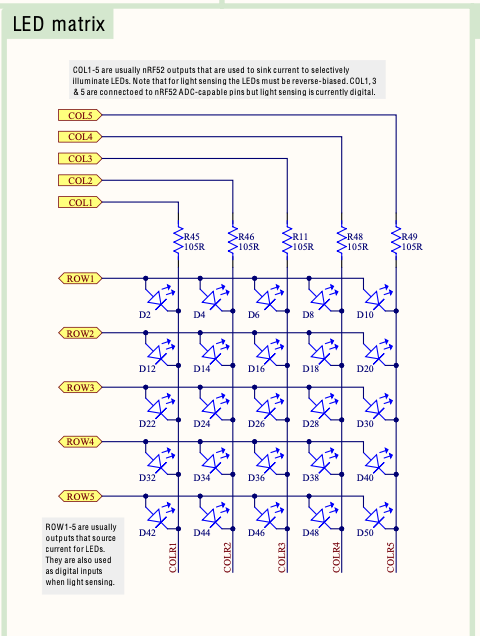
If you want to have a try: go to rust playground: <https://play.rust-lang.org/?version=stable&mode=debug&edition=2024>

Instead of RUN: 

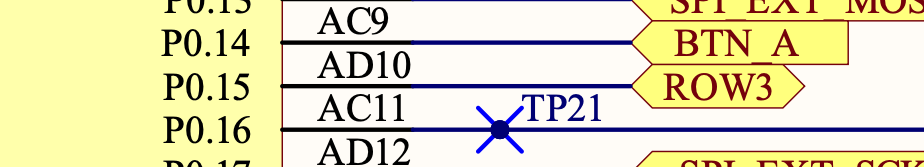
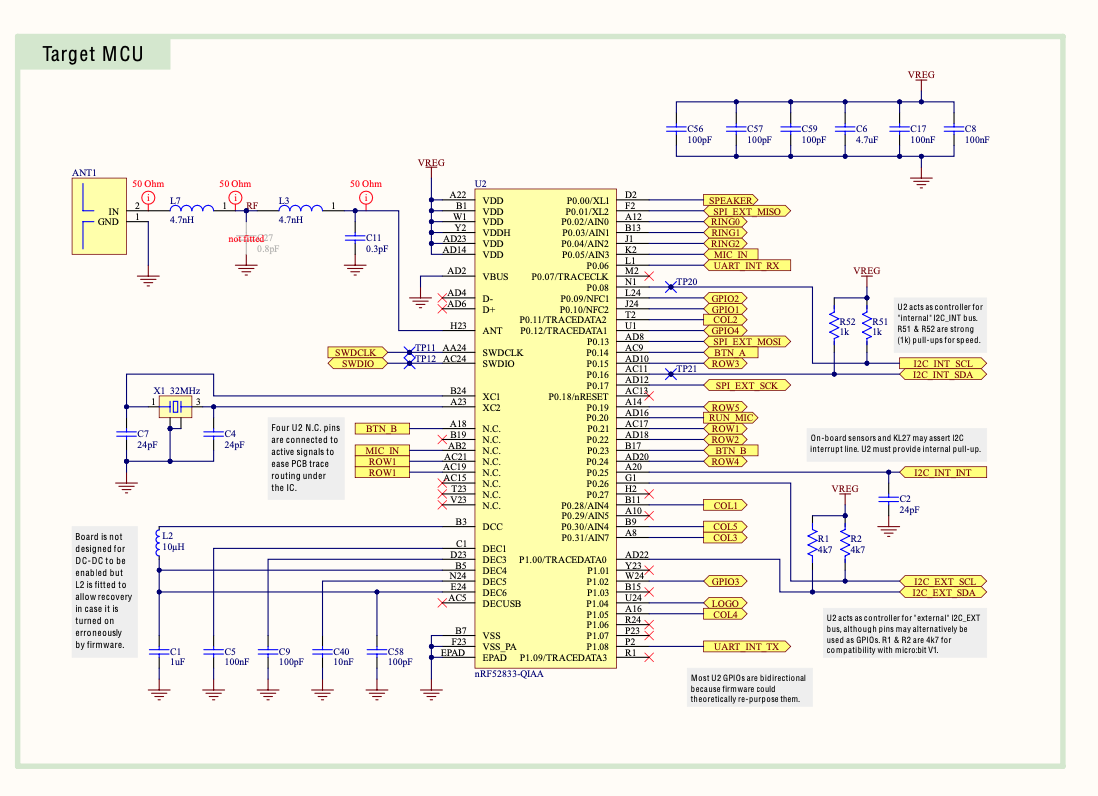
Try ASM (SHOW ASSEBLY): 

First, let’s try to find the Pins we need to drive an LED:

* 1. Google “microbit v2 schematics”.
  2. Go to the github page hosting the hardware info.
  3. On their github page, find the schematic pdf, download and save the file on your PC, then open it.
  4. Find the page for LED matrix, pick an LED you would like to blink, find the ROW# and COL#. Write it down.



If you click on the ROW & COL, you will be redirected to the target MCU page, where you can find the Pin number for the ROW & COL. For example, ROW3 was connected to P0.15. Write down the Pin number for your LED somewhere.



For example, ROW3 was connected to P0.15.

So now, how do we code and drive 3.3V through two pins? We are going to use a *magic* spell called MMIO.

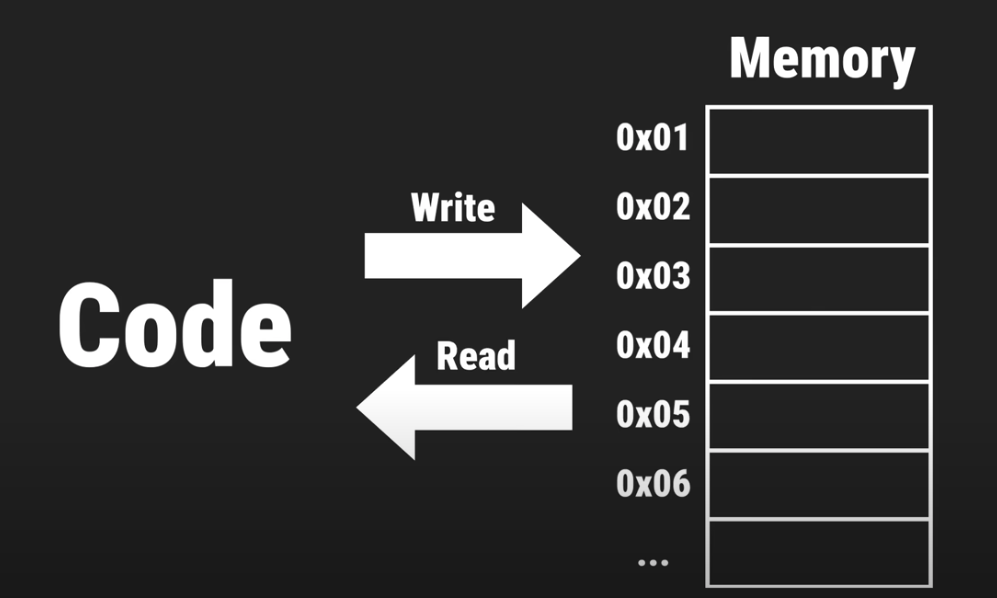
* 1. Memory-Mapped Input/Output (MMIO)

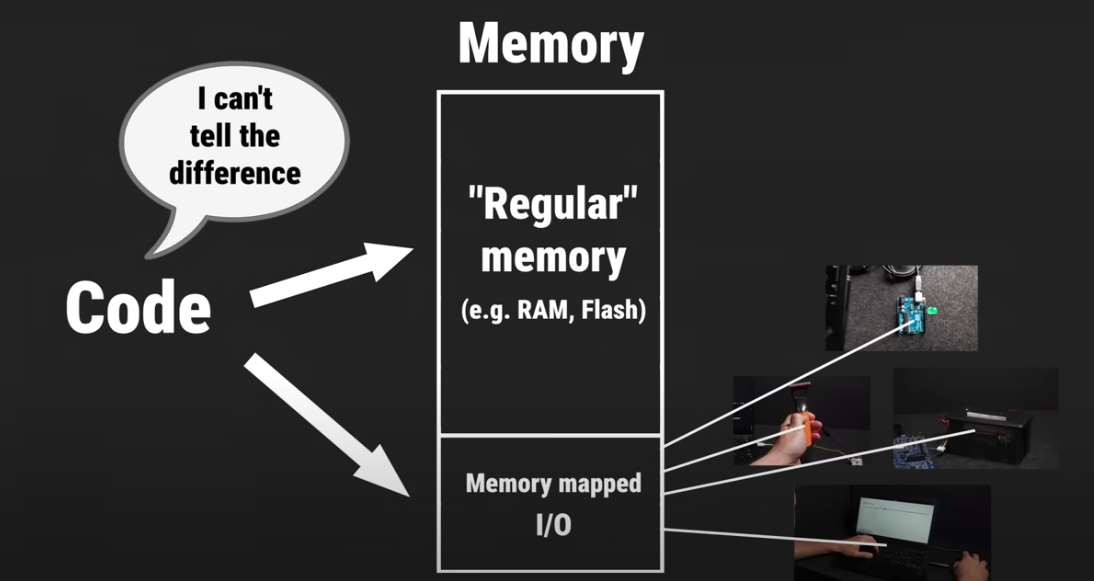
For background knowledge, here’s a good blog post:

<https://ctf.re/kernel/pcie/tutorial/dma/mmio/tlp/2024/03/26/pcie-part-2/#:~:text=Memory%20Mapped%20Input/Output%20(abbrev,to%20and%20from%20the%20device>.

Or an even better YouTube video: <https://www.youtube.com/watch?v=sp3mMwo3PO0> by Arful Bytes.

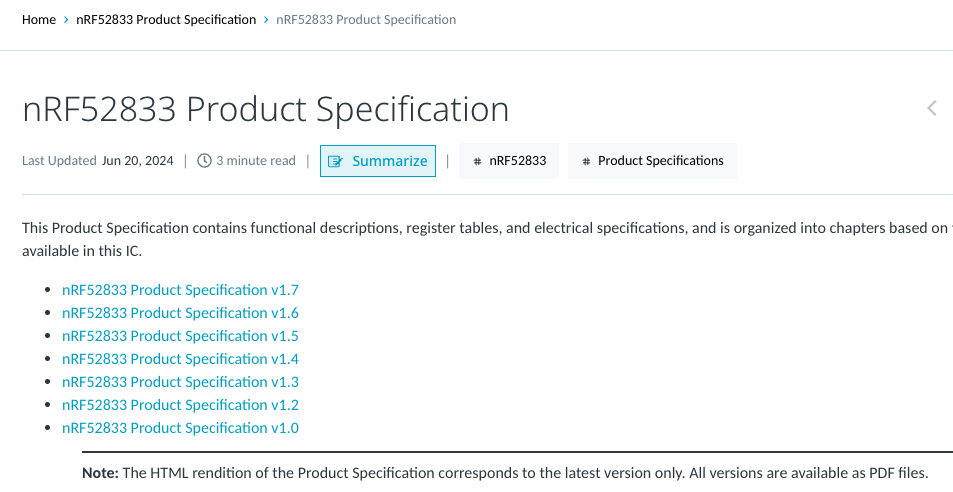
In brief summary, your code is just binary codes (expressed in base 16, cuz base 2 would be too long) *Read* and *Write* into *Memory* (like pointers in C):



MMIO is just a *Memory* dedicated to a hardware

Everything is just a memory operation. You can config, write or read your hardware by write values into a specific address. Now let’s find out what’s the address and what is the value for blinky.

* 1. To find that MMIO address of your chosen LED, search the spec of that MCU. Google “nRF52833 product spec”. Find page that’s provided by its chip designer NORDIC semiconductor. 

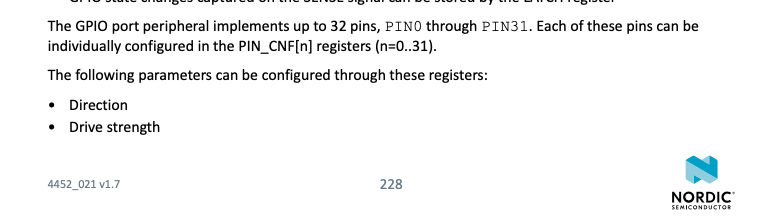


Download the PDF file and open it.

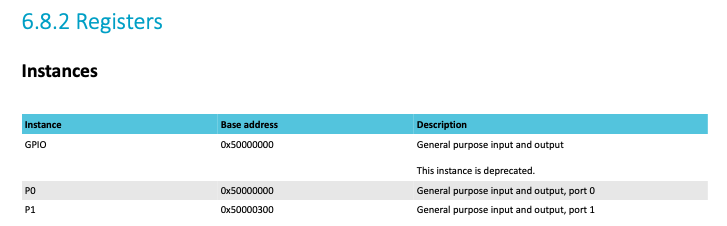


In that PDF file, search for “GPIO”. Click on the first page and that’s the info we need:

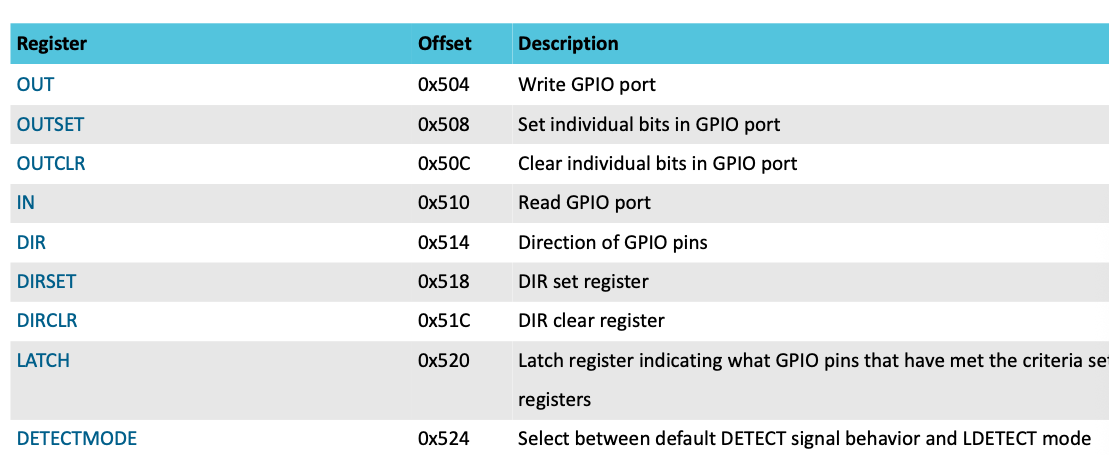


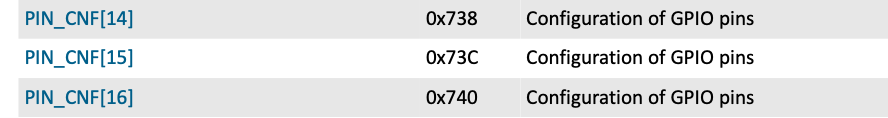


Registers ehh? Go down the file till you see Registers.



We need to look for the address for the Pin you need. Go down the page until you see the Register overview list.



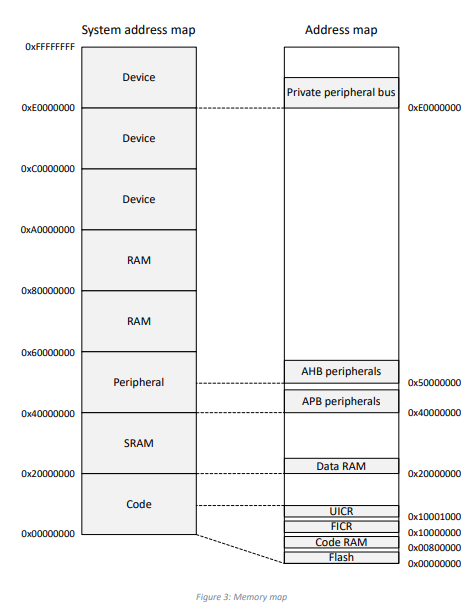


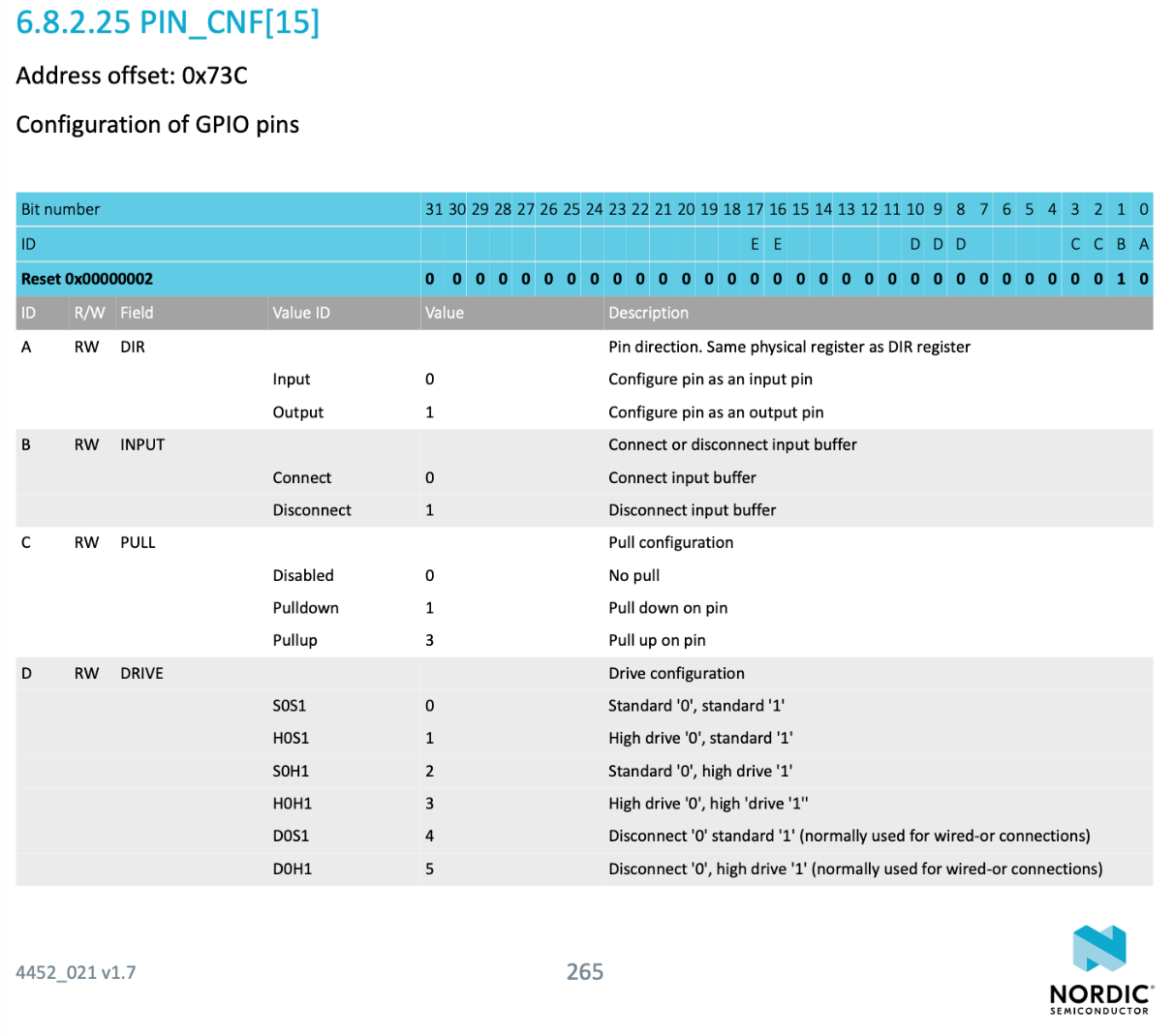
For example, to config GPIO pin 14, we need to go to 0x50000000 + 0x738 = 0x50000738.

Write down the PIN\_CNF Offset for your pins.

Before we move on, here’s the brief explanation of what is going on: that’s simply how MMIO is organized in the MCU. The base address 0x50000000 is the starting point of memory location for all GPIOs. “0x” means it’s hexadecimal (16 base). If you have played with Arduino before, you will know GPIO is the general-purpose input/output pins for hardware such as sensors and motors (aka peripherals).

Those memory, like all memory, are organized in a stack, so you have to offset from the starting point to go to exact locations so your specific pins will perform a specific task (registers). From the memory map of the datasheet, looks like all the peripherals are located at 0x40000000 – 0x60000000 location.



Click on the PIN\_CNF[your number here], will bring you to the bit map so you can check *what value* you have send to the register address for *Pin configuration* (PIN\_CNF).

To do a blinky config, we only need to care about two fields:

1. A defines the direction of the pin. 0 for input and 1 for output.
2. D the drive config, S0S1 should be powerful enough for LED (3.3V). (To drive a motor (12V) will be a different story.)

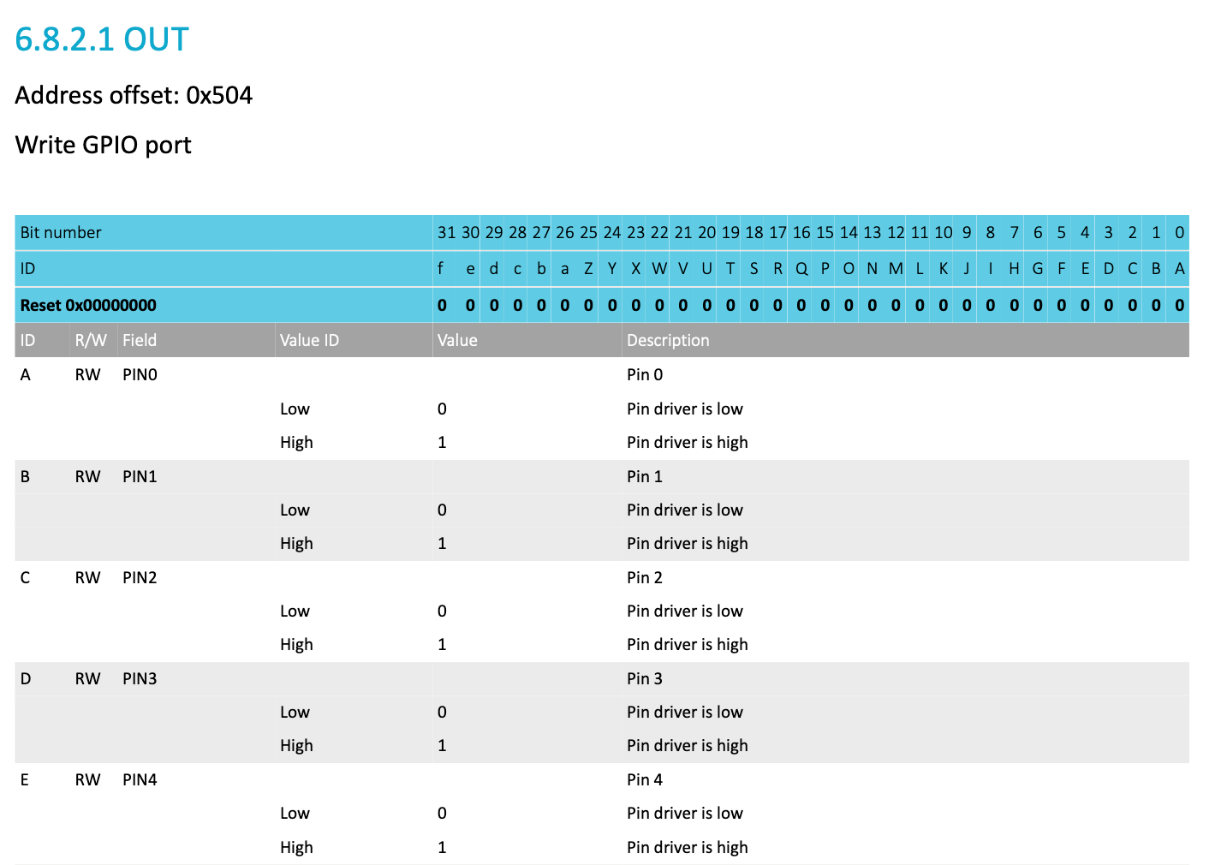
Note: The “value” in the table here is expressed in base 10. But you have to convert it to base 2 for programming. E.g., for field C (internal pull resistor):

|  |  |  |
| --- | --- | --- |
| Value (base 10) | To base 2 | What will happen |
| 0 | 00 | No pull |
| 1 | 01 | Pull down |
| 3 | 11 | Pull up |

And that’s why field C requires 2-bits space.

Write down the bit number for A and D.

Once the config is complete, the control of the output is set by, you guess it right, an OUT register. Look for the OUT address:



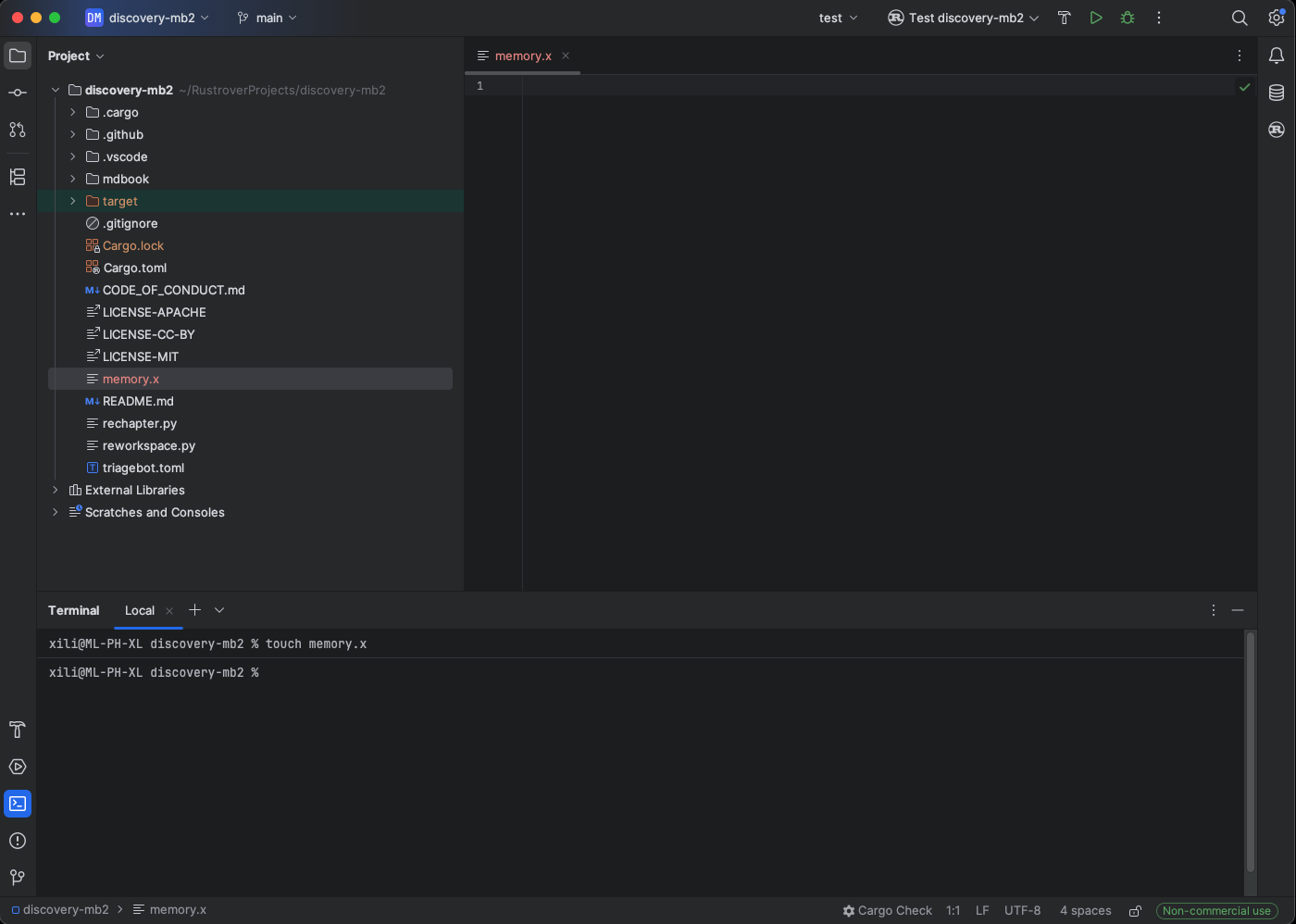
The address is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

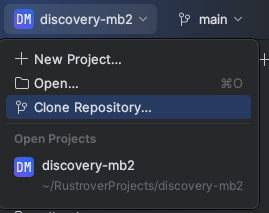
Now we need to look for the value sent to this address, e.g., for PIN4 it’s E, Bit number 4, and 1 will drive it high (3.3V).

Write down the Bit number for your two PINs.

Now we know (to config and to control,) the specific address we need to send to, and the specific value we need to send. What’s next?

* 1. Rust Rover warm-ups for windows:

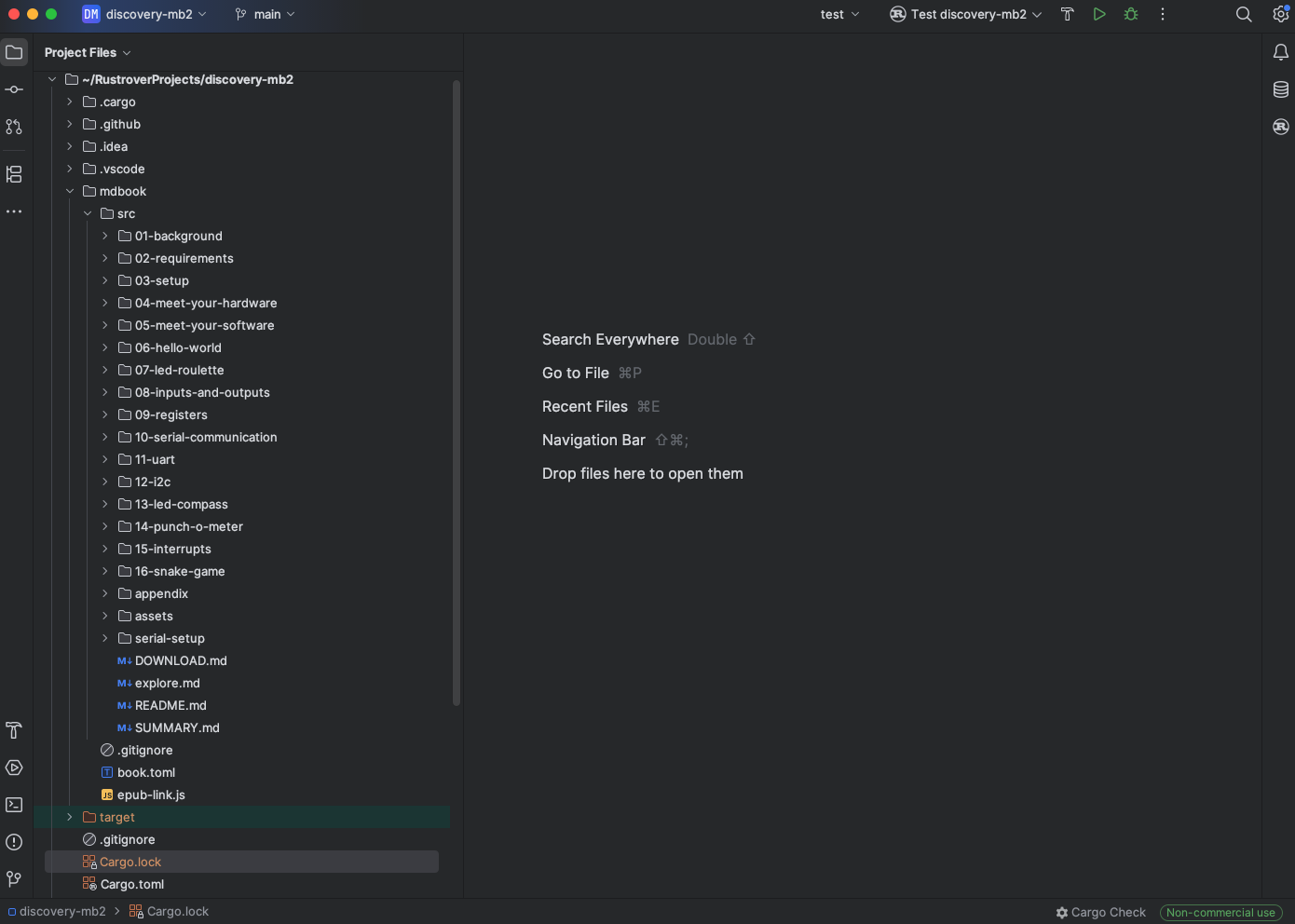


Time to start your Rust embedded environment in Rust Rover. Let’s start from the beginning and:

The address for the repo is：

<https://github.com/rust-embedded/discovery-mb2.git>

Rust Rover will ask where you would like to put the folder in your PC. Pick a spot and start cloning. If everything works well, you would see the DISCOVERY-MB2 folder shown up as follows:

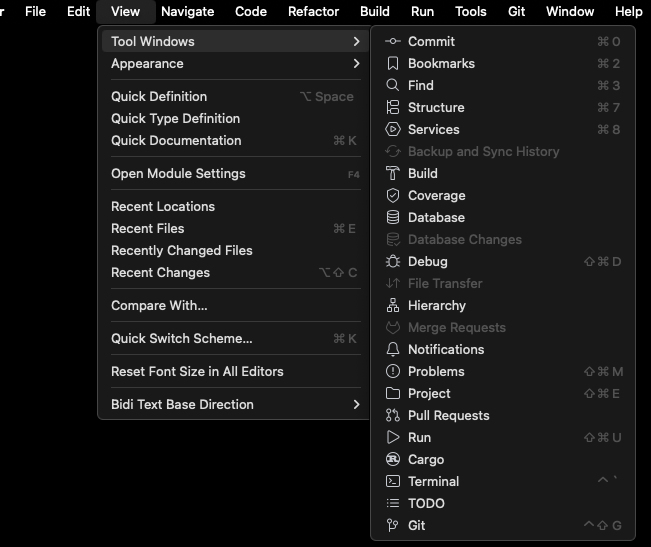


You can’t have Rust Analyzer messing up your code if you don’t use Rust Analyzer.

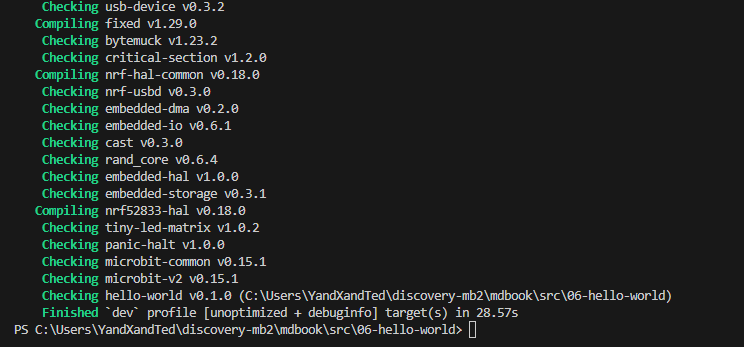
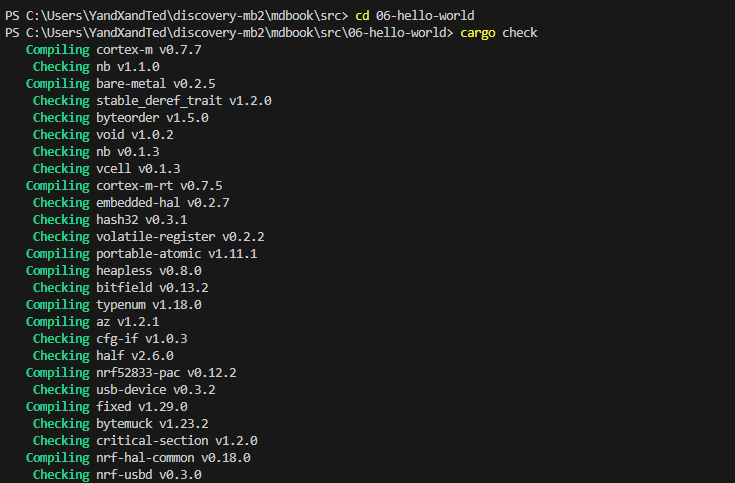
No need for global setting .vscode/settings.json anymore.

Now open the terminal: Open: View | Tool Windows | Terminal

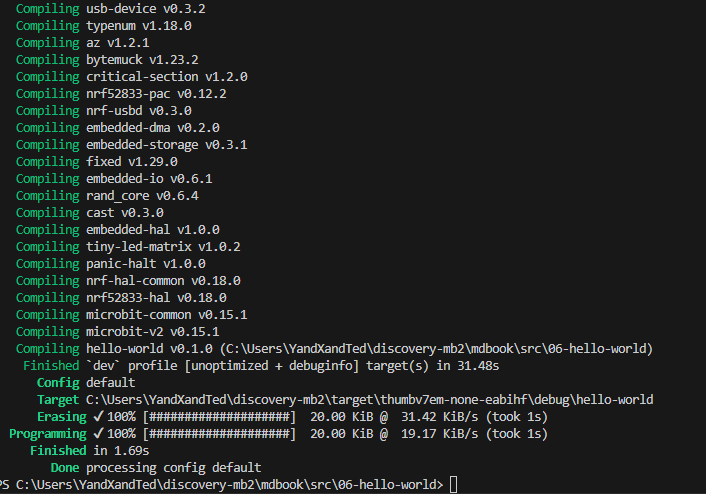
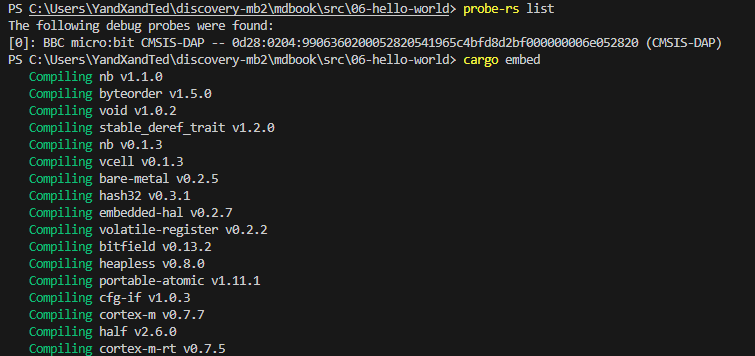
The shortcut in Windows is Alt+F12.



Try go to one of the folders (like 06-hello-world), see if you can flash and run it on Microbit without issue:

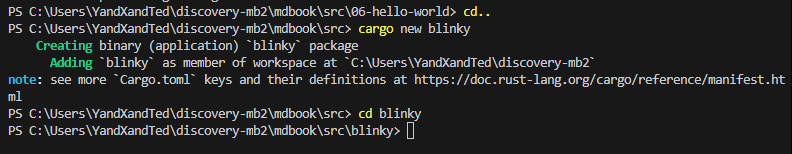


If you type “Cargo embed”, it should work on your microbit:

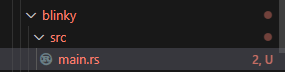


Note: don’t forget to plug in your microbit.

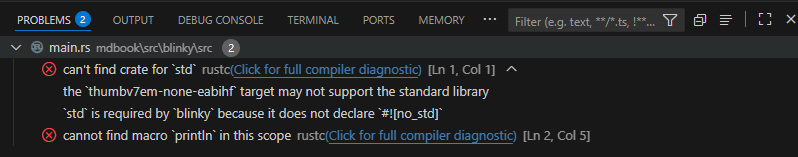
* + 1. Open the DISCOVERY-MB2 folder, and let’s make a new embedded program from scratch.
    2. Go to src folder with all the working code by the command “cd <your directory here>” in the terminal.
    3. Generate a new project folder, use the command “cargo new <your new folder name here>” in the terminal.
    4. You can also change the name of the new folder by the command “mv <old name> <new name>”.



The new project folder “blinky” is red, that’s ok.



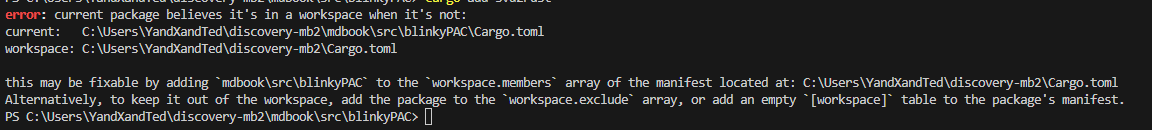
If you click on the problem, you can see there’s something wrong with the main and std library. We don’t need them anyway.

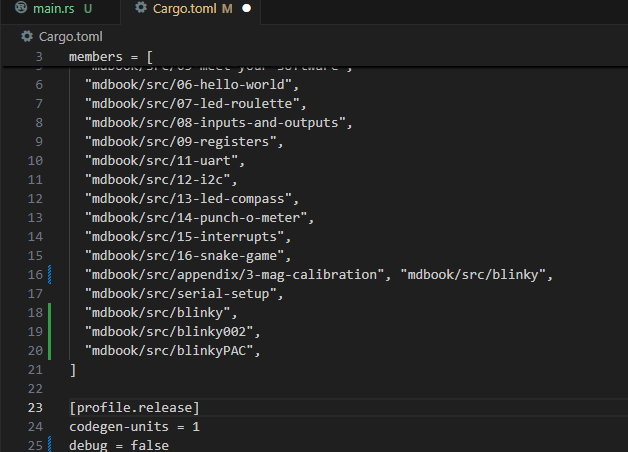


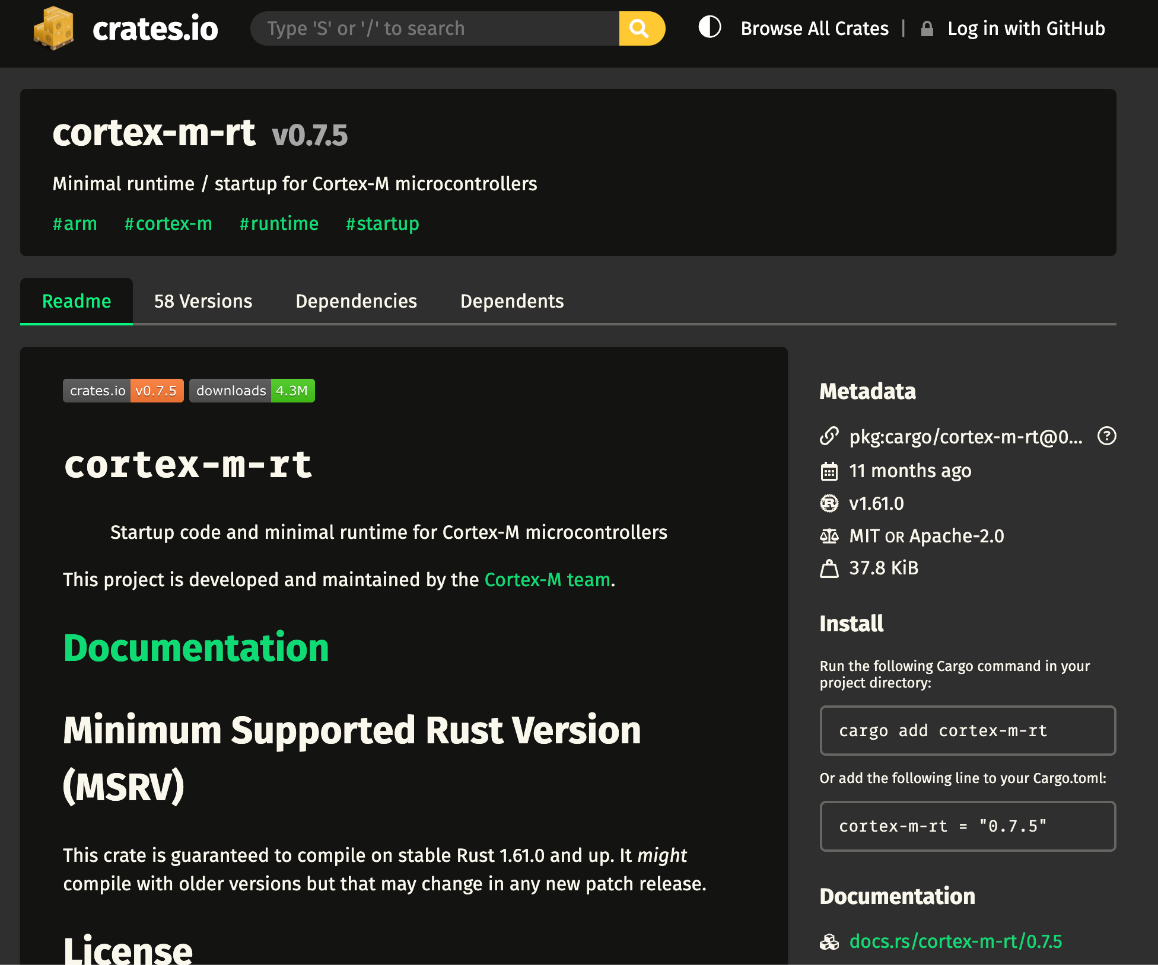
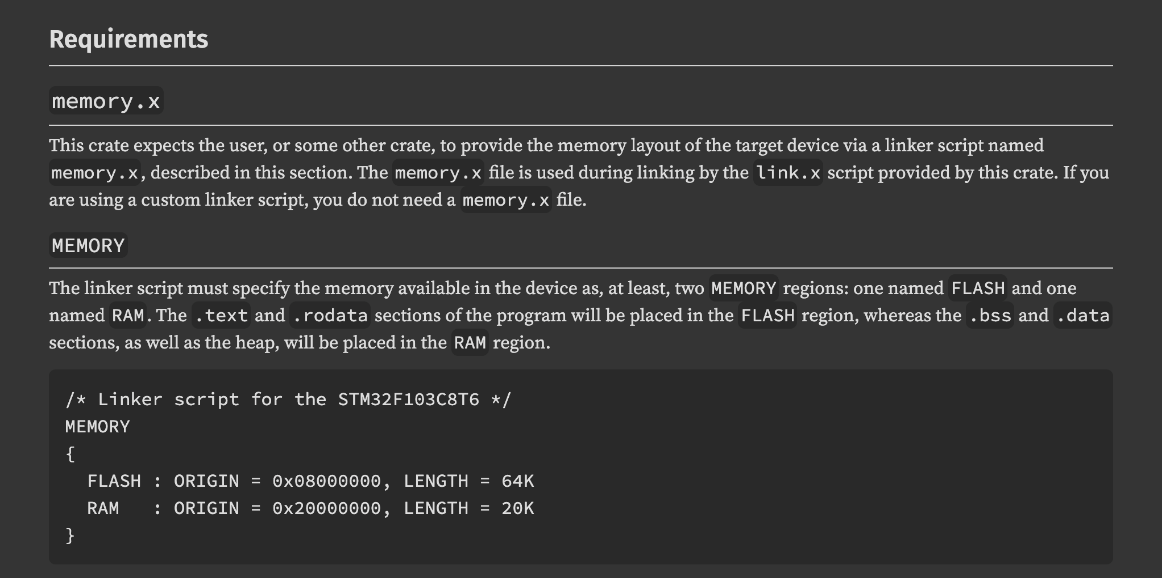
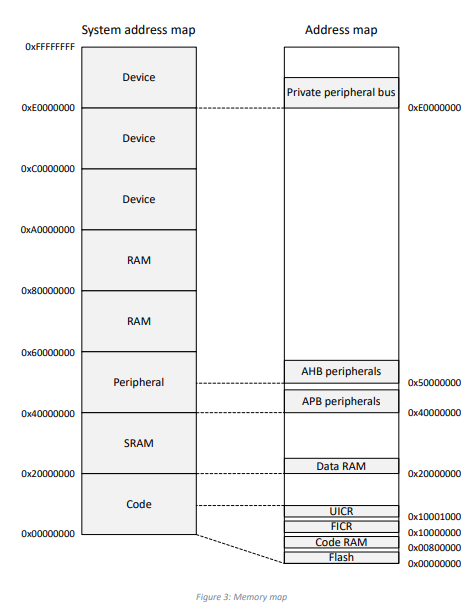
To change for embedded, type the following lines at the start. Once you hit “save” (ctrl+S), the problems will be gone, for now:



Notes: if you are having the following workspace error:

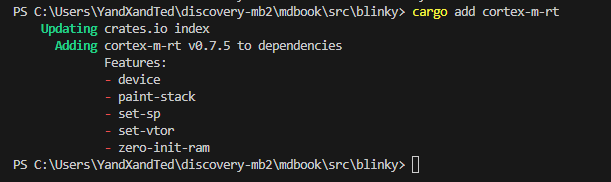


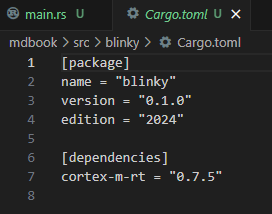
Go to root \discovery-mb2\Cargo.toml, and add your new folder to the member list:

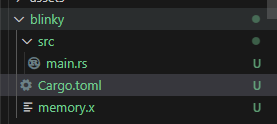
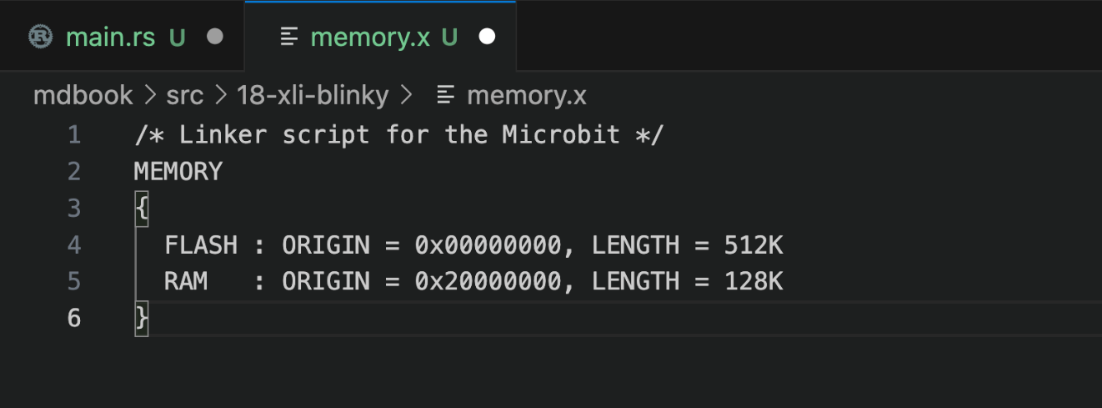
* + 1. Now it’s time to install stuff (add dependencies) that knows how to talk to the MCU. Go to crates.io
    2. Search cortex-m-rt, and click on the documentation link:
    3. Look at the requirements, it need a “memory.x” file. Within the file it requires the starting address and the size of the FLASH and RAM regions of the memory for the MCU. The Nordic chip has 512KB of flash and 128KB of RAM. (Don’t close this webpage yet.) As for the origin, based on the memory map:

Flash starts at: \_\_\_\_\_\_\_\_\_\_

RAM starts at: \_\_\_\_\_\_\_\_\_\_

* + 1. Type “cargo add cortex-m-rt” in the terminal:It will also be added to the Cargo.toml as dependencies:



* + 1. Now add the needed memory.x file by type “type memory.x” in the terminal:Look, new file appears in the folder:
    2. Copy-paste the requirements into the file, and modify the ORIGIN address and size accordingly:

/\* Linker script for the Microbit v2 \*/

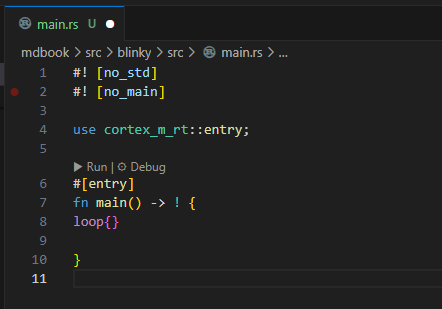
MEMORY

{

FLASH : ORIGIN = 0x00000000, LENGTH = 512K

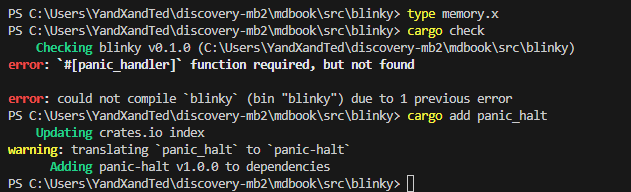
RAM : ORIGIN = 0x20000000, LENGTH = 128K

}

* + 1. And modify the main.rs to use this handy tool:

The main will never return, and never ends.

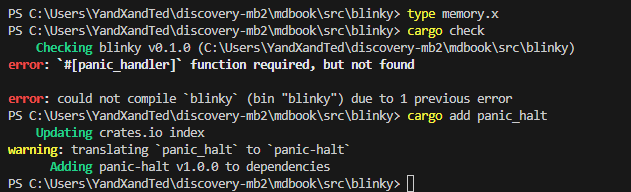
We are depending on cortex\_m\_rt, and use [entry] to find the entry point.

* + 1. The house keeping work has already been done in the global .cargo.config.toml and .vscode/settings.json, all good now.
    2. Fix the missing “Panic handler”, which is reserved by Rust for fatal errors. 

The panic handler is just another function we need to config so the Rust compiler will be happy.

For embedded, it’s fine if the panic handler does nothing for now. To do that, add this new cargo:

Type “cargo add panic\_halt”



Update the main.rs:

* + 1. Now if you type “cargo check”, everything should be fine now:



* + 1. Type “cargo embed”, and you should be able to flash an empty program into the MCU.

Notes: if your cargo compiler in windows having error linking memory.x, try to install a global memory.x file at the root (aka the DISCOVERY-MB2 folder). The command line I found that worked is:

notepad memory.x



And then copy paste the address info into that global memory.x.

/\* Linker script for the Microbit v2 \*/

MEMORY

{

FLASH : ORIGIN = 0x00000000, LENGTH = 512K

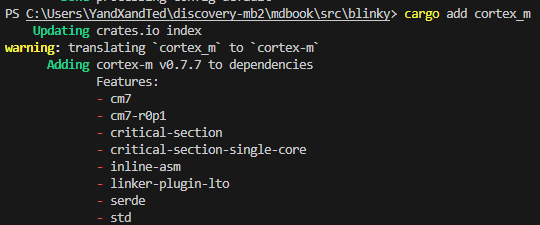
RAM : ORIGIN = 0x20000000, LENGTH = 128K

}

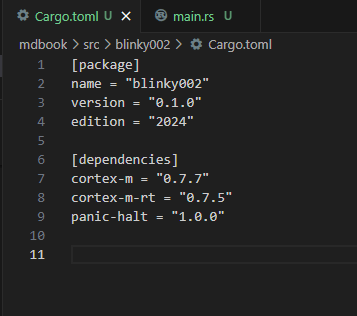
The other trick I found could help is to clean up the previous buggy build by type: cargo clean.



* + 1. Before we start the coding, we need to install another cargo to get low-level (no-operation) access to the chip, type: “cargo add cortex-m”



Make sure it is properly added in the dependencies at cargo.toml:

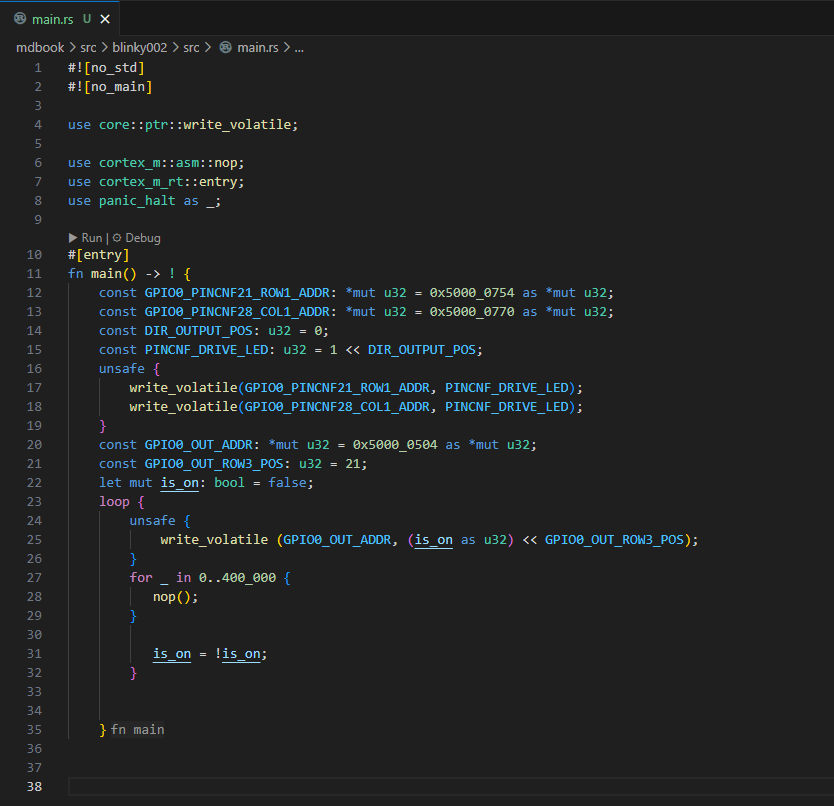


* + 1. Unlike C, Rust don’t like Borrow Checker that can’t be verified or validated. That’s why we need to put everything that appears *unsafe* to Rust in an

unsafe {}

bracket. It’s OK for this application, because we know what is that address and no one is going to use it. IRL too many *Unsafe Rust* is not a good coding practices.

Let’s have some fun typing the following codes:



The codes below can be copy pasted:

#![no\_std]

#![no\_main]

use core::ptr::write\_volatile;

use cortex\_m::asm::nop;

use cortex\_m\_rt::entry;

use panic\_halt as \_;

#[entry]

fn main() -> ! {

    const GPIO0\_PINCNF21\_ROW1\_ADDR: \*mut u32 = 0x5000\_0754 as \*mut u32;

    const GPIO0\_PINCNF28\_COL1\_ADDR: \*mut u32 = 0x5000\_0770 as \*mut u32;

    const DIR\_OUTPUT\_POS: u32 = 0;

    const PINCNF\_DRIVE\_LED: u32 = 1 << DIR\_OUTPUT\_POS;

    unsafe {

        write\_volatile(GPIO0\_PINCNF21\_ROW1\_ADDR, PINCNF\_DRIVE\_LED);

        write\_volatile(GPIO0\_PINCNF28\_COL1\_ADDR, PINCNF\_DRIVE\_LED);

    }

    const GPIO0\_OUT\_ADDR: \*mut u32 = 0x5000\_0504 as \*mut u32;

    const GPIO0\_OUT\_ROW3\_POS: u32 = 21;

    let mut is\_on: bool = false;

    loop {

        unsafe {

            write\_volatile (GPIO0\_OUT\_ADDR, (is\_on as u32) << GPIO0\_OUT\_ROW3\_POS);

        }

        for \_ in 0..400\_000 {

           nop();

        }

           is\_on = !is\_on;

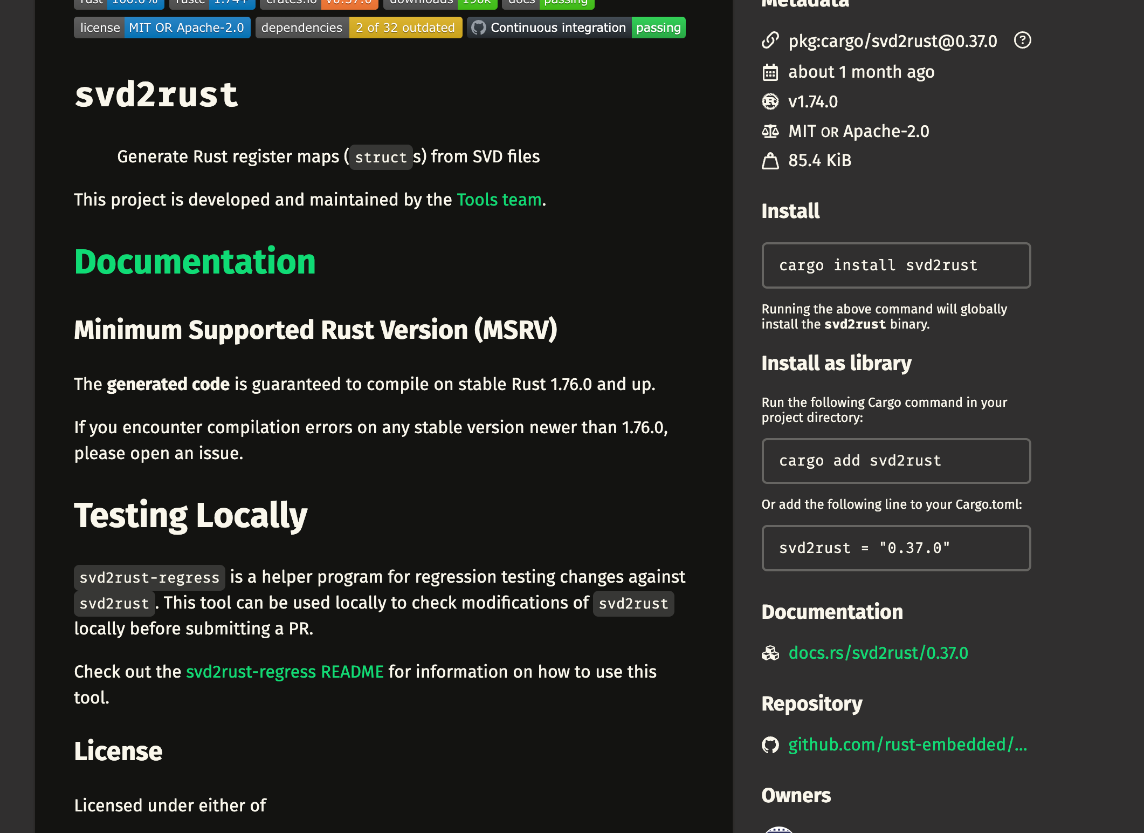
        }

    }

Change the pin address (and output address) to the ones you chose. And type “cargo embed” in the terminal, it should work now.

After all those hassles to blink one LED light, you would probably think atm: *there has to a better way*. And that’ why we are moving on to the next chapter:

1. **PAC (Peripheral Access Crate)**

2.1) Go to crates.io again and search for “svd2rust” and click on the documentations:

Chip makers have provided a detailed pac file with register mappings for their chip. And svd2rust will do the rest, access the registers by name and writing bits to them.

If you search nrf52883-pac in crates.io, you will find the mapping file for the microbit chip:

2.2)

Type “cargo add svd2rust” in the terminal

Type “cargo add nrf52883-pac” in the terminal

This time the google rust team has a working code already to go, visit: <https://google.github.io/comprehensive-rust/bare-metal/microcontrollers/pacs.html>

And copy paste the code into your main.rs.

Now you only need to change the pin# to the LED you picked.

The following code works in Rust Rover:

#![no\_main]

#![no\_std]

extern crate panic\_halt as \_;

use cortex\_m\_rt::entry;

use nrf52833\_pac::Peripherals;

#[entry]

fn main() -> ! {

let p = Peripherals::take().unwrap();

let gpio0 = p.P0;

// Configure GPIO 0 pins 21 and 28 as push-pull outputs.

gpio0.pin\_cnf[21].write(|w| {

w.dir().output();

w.input().disconnect();

w.pull().disabled();

w.drive().s0s1();

w.sense().disabled();

w

});

gpio0.pin\_cnf[28].write(|w| {

w.dir().output();

w.input().disconnect();

w.pull().disabled();

w.drive().s0s1();

w.sense().disabled();

w

});

// Set pin 28 low and pin 21 high to turn the LED on.

gpio0.outclr.write(|w| w.pin28().clear());

gpio0.outset.write(|w| w.pin21().set());

loop {}

}

1. Can PAC be even better?

YES. Entering HAL (Hardware Abstraction Layer) crate.

It built itself on top of the PAC, making sure all the interaction between GPIOs working as intended. And any MCU you can use for the final project can be find here:

<https://github.com/rust-embedded/awesome-embedded-rust#hal-implementation-crates>

Type “cargo add nrf52833-hal” in the terminal.

The code from the google rust team has to be fixed a bit, copy and paste the code below in your main.rs:

#![no\_main]

#![no\_std]

// extern crate panic\_halt as \_;

use cortex\_m\_rt::entry;

use embedded\_hal::digital::OutputPin;

use nrf52833\_hal::gpio::{Level, p0};

use nrf52833\_hal::pac::Peripherals;

use panic\_halt as \_;

#[entry]

fn main() -> ! {

let p = Peripherals::take().unwrap();

// Create HAL wrapper for GPIO port 0.

let gpio0 = p0::Parts::new(p.P0);

// Configure GPIO 0 pins 21 and 28 as push-pull outputs.

let mut col4 = gpio0.p0\_30.into\_push\_pull\_output(Level::High);

let mut row4 = gpio0.p0\_24.into\_push\_pull\_output(Level::Low);

// Set pin 28 low and pin 21 high to turn the LED on.

col4.set\_low().unwrap();

row4.set\_high().unwrap();

loop {}

}

The following code works in Rust Rover:

#![no\_main]  
#![no\_std]  
  
extern crate panic\_halt as \_;  
  
use cortex\_m\_rt::entry;  
use embedded\_hal::digital::OutputPin;  
use nrf52833\_hal::gpio::{*Level*, p0};  
use nrf52833\_hal::pac::Peripherals;  
  
#[entry]  
fn main() -> ! {  
 let p = Peripherals::take().unwrap();  
  
 // Create HAL wrapper for GPIO port 0.  
 let gpio0 = p0::Parts::new(p.P0);  
  
 // Configure GPIO 0 pins 21 and 28 as push-pull outputs.  
 let mut col1 = gpio0.p0\_11.into\_push\_pull\_output(*Level*::*High*);  
 let mut row1 = gpio0.p0\_21.into\_push\_pull\_output(*Level*::*Low*);  
  
 // Set pin 28 low and pin 21 high to turn the LED on.  
 col1.*set\_low*().unwrap();  
 row1.*set\_high*().unwrap();  
  
 loop {}  
}

Change the pin#, and do a cargo embed to see if it works.

At HAL level, Rust embedded is almost like working with Arduino, yes?