

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.

0) 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 compiler, 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.

```
1 fn main() {
2     let z = 13;
3     let x = {
4         let y = 10;
5         dbg!(y);
6         z - y
7     };
8     dbg!(x);
9     // dbg!(y);
10 }
```

[src/main.rs:5:9] y = 10
[src/main.rs:8:5] x = 3

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

```
1 fn main() {
2     let x = 10;
3     if x == 0 {
4         println!("zero!");
5     } else if x < 100 {
6         println!("biggish");
7     } else {
8         println!("huge");
9     }
10 }
```

But it can also be used as an expression:

```
1 fn main() {
2     let x = 1000;
3     let size = if x < 20 { "small" } else if x < 100 { "biggish" } else { "huge" };
4     println!("number size: {}", size);
5 }
```

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-](https://play.rust-lang.org/?version=stable&mode=debug&edition=2024)
[lang.org/?version=stable&mode=debug&edition=2024](https://play.rust-lang.org/?version=stable&mode=debug&edition=2024)

Instead of RUN:

The image shows a code editor interface with a dark theme. At the top, there is a toolbar with buttons: 'RUN' (with a play icon), a button with three dots, 'DEBUG' (with a dropdown arrow), 'STABLE' (with a dropdown arrow), and another button with three dots. To the right of these are buttons for 'SHARE', 'TOOLS' (with a dropdown arrow), 'CONFIG' (with a gear icon and a dropdown arrow), and a help icon (question mark). The main area of the editor contains a code snippet:

```
1 fn main() {  
2     println!("Hello, world!");  
3 }
```

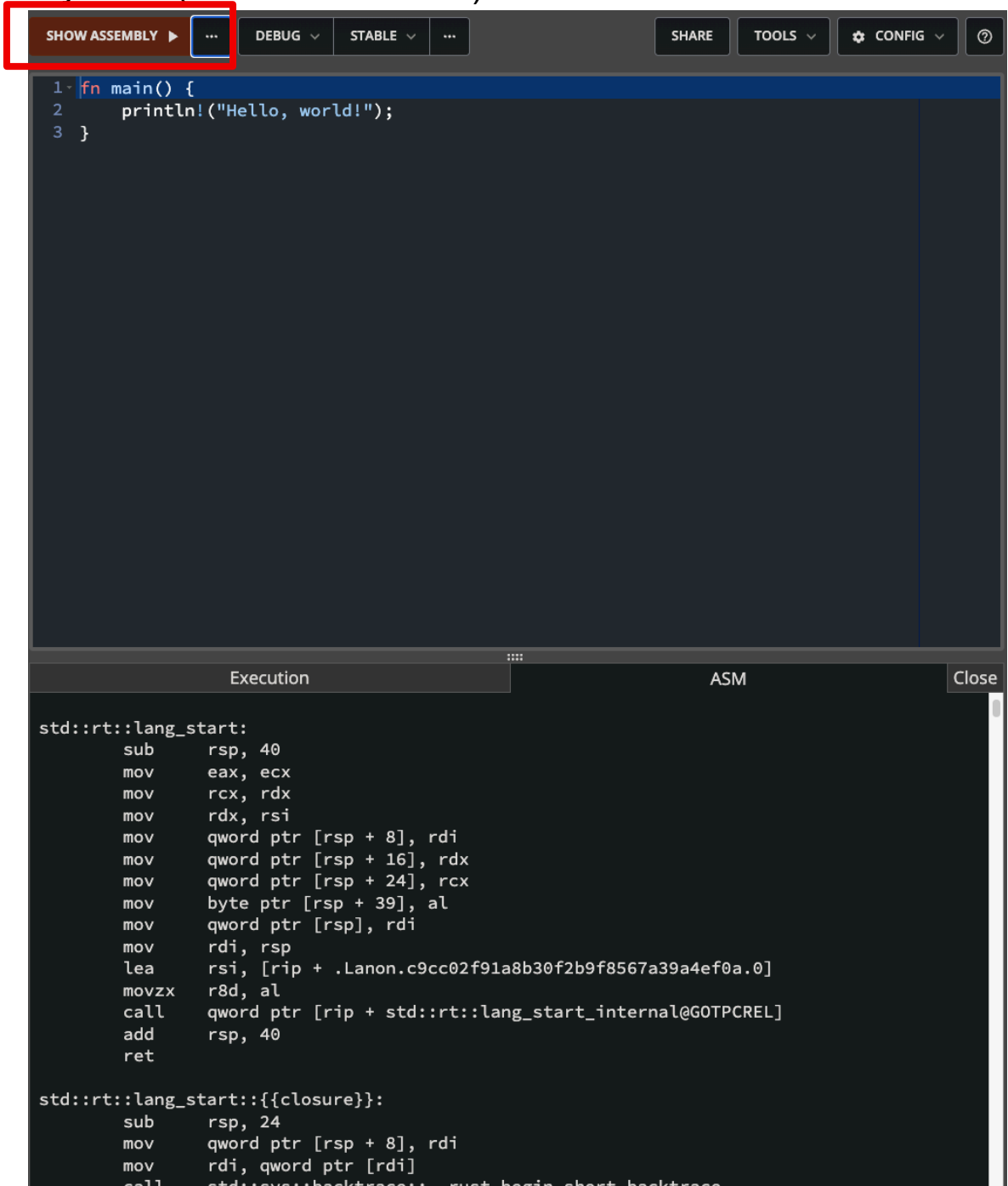
Below the code editor is a panel with a tab labeled 'Execution' and a 'Close' button. This panel is divided into two sections: 'Standard Error' and 'Standard Output'. The 'Standard Error' section contains the following text:

```
Compiling playground v0.0.1 (/playground)  
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.82s  
Running `target/debug/playground`
```

The 'Standard Output' section contains the text:

```
Hello, world!
```

Try ASM (SHOW ASSEMBLY):



The screenshot shows a Rust IDE interface. At the top, a toolbar contains several buttons: 'SHOW ASSEMBLY' (highlighted with a red box), a three-dot menu, 'DEBUG', 'STABLE', and another three-dot menu. To the right are 'SHARE', 'TOOLS', 'CONFIG', and a help icon. Below the toolbar is a code editor with the following Rust code:

```
1 fn main() {  
2     println!("Hello, world!");  
3 }
```

Below the code editor is a panel with two tabs: 'Execution' and 'ASM'. The 'ASM' tab is selected, showing the assembly code for the 'main' function. The assembly code is as follows:

```
std::rt::lang_start:  
    sub     rsp, 40  
    mov     eax, ecx  
    mov     rcx, rdx  
    mov     rdx, rsi  
    mov     qword ptr [rsp + 8], rdi  
    mov     qword ptr [rsp + 16], rdx  
    mov     qword ptr [rsp + 24], rcx  
    mov     byte ptr [rsp + 39], al  
    mov     qword ptr [rsp], rdi  
    mov     rdi, rsp  
    lea     rsi, [rip + .Lanon.c9cc02f91a8b30f2b9f8567a39a4ef0a.0]  
    movzx   r8d, al  
    call    qword ptr [rip + std::rt::lang_start_internal@GOTPCREL]  
    add     rsp, 40  
    ret  
  
std::rt::lang_start::{{closure}}:  
    sub     rsp, 24  
    mov     qword ptr [rsp + 8], rdi  
    mov     rdi, qword ptr [rdi]  
    call    std::sys::backtrace::rust_begin_short_backtrace
```

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

- 1.1) Google "microbit v2 schematics".
- 1.2) Go to the github page hosting the hardware info.

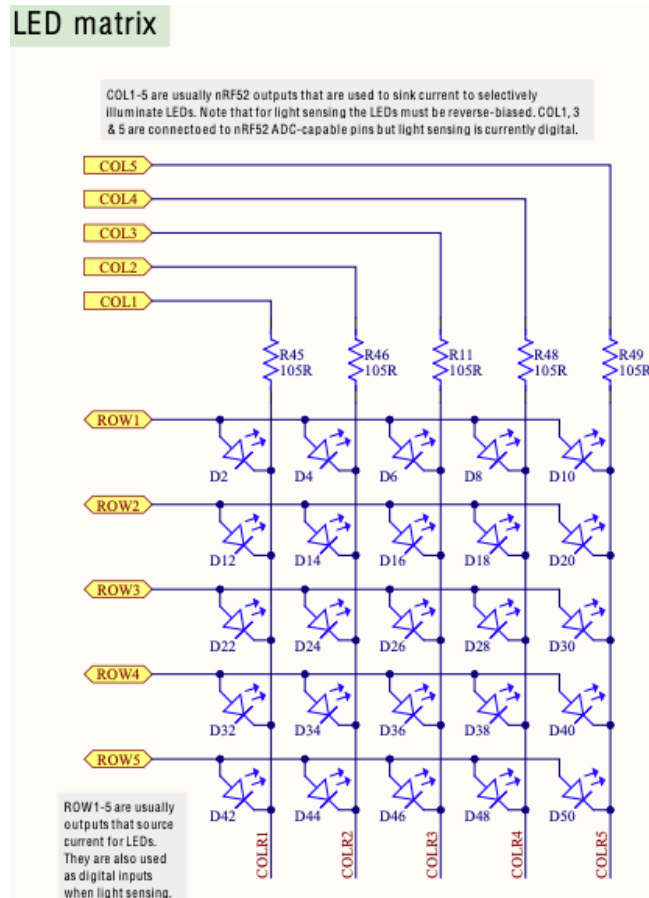
github.com/microbit-foundation/microbit-v2-hardware

- 1.3) On their github page, find the schematic pdf, download and save the file on your PC, then open it.

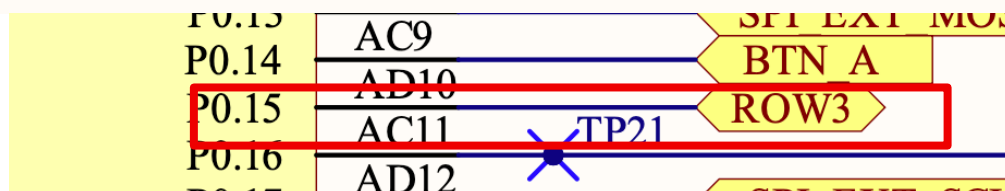
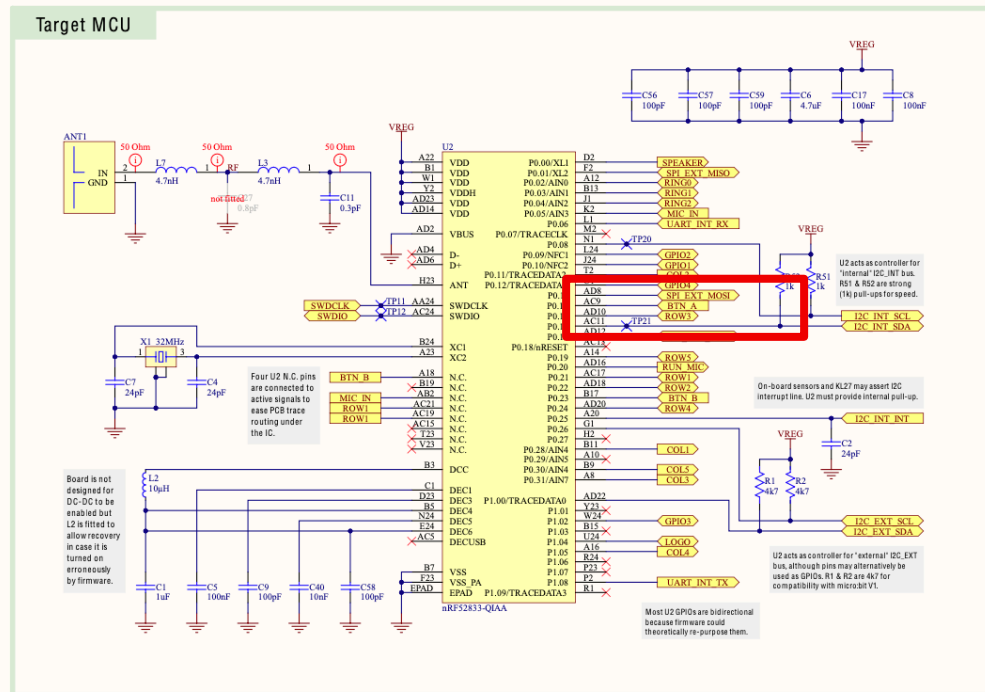
MicroBit_V2.0.0_S_schematic.PDF

Add micro:bit V2.2 schematic and BOM

- 1.4) 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.

1.5) Memory-Mapped Input/Output (MMIO)

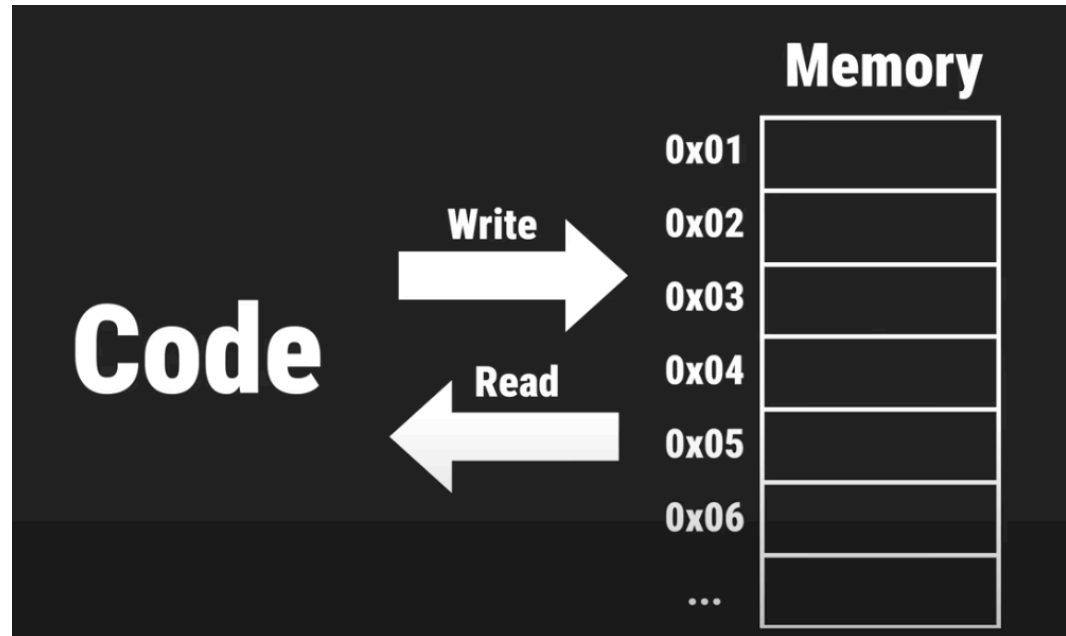
For background knowledge, here's a good blog post:

[https://ctf.re/kernel/pcie/tutorial/dma/mmio/tlp/2024/03/26/pcie-part-2/#:~:text=Memory%20Mapped%20Input/Output%20\(a,bbreve,to%20and%20from%20the%20device.](https://ctf.re/kernel/pcie/tutorial/dma/mmio/tlp/2024/03/26/pcie-part-2/#:~:text=Memory%20Mapped%20Input/Output%20(a,bbreve,to%20and%20from%20the%20device.)

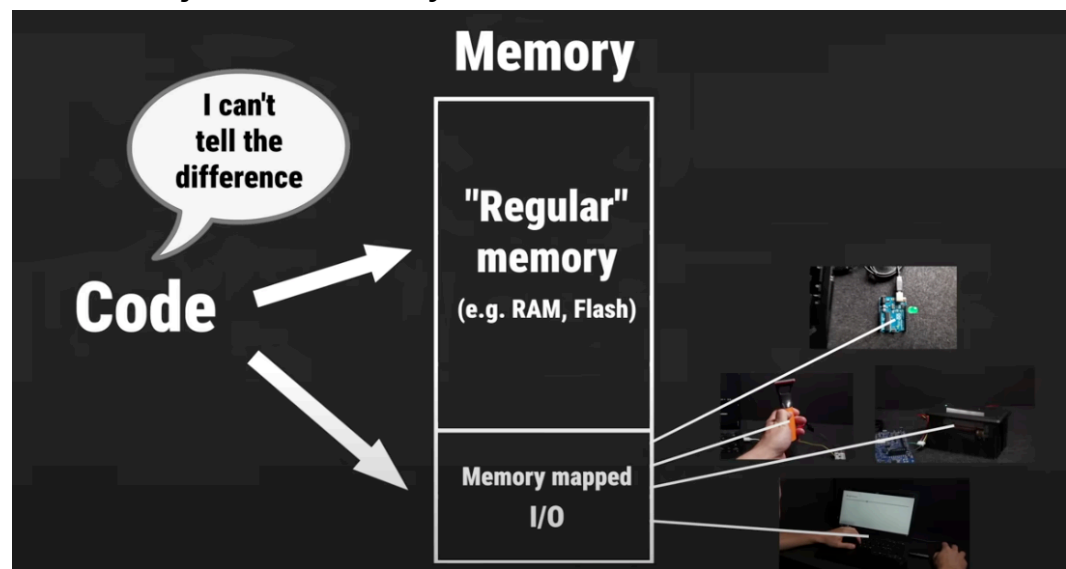
Or an even better YouTube video:

<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.

- 1.6) 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.

docs.nordicsemi.com/bundle/ps_nrf52833/page/keyfeatures_html5.html

Home > nRF52833 Product Specification > nRF52833 Product Specification

nRF52833 Product Specification

Last Updated Jun 20, 2024

3 minute read

Summarize

nRF52833

Product Specifications

This Product Specification contains functional descriptions, register tables, and electrical specifications, and is organized into chapters based on available in this IC.

- **nRF52833 Product Specification v1.7**
- nRF52833 Product Specification v1.6
- nRF52833 Product Specification v1.5
- nRF52833 Product Specification v1.4
- nRF52833 Product Specification v1.3
- nRF52833 Product Specification v1.2
- nRF52833 Product Specification v1.0

Note: The HTML rendition of the Product Specification corresponds to the latest version only. All versions are available as PDF files.

Download the PDF file and open it.

nRF52833

Product Specification
v1.7

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

6.8 GPIO — General purpose input/output.	228
6.8.1 Pin configuration registers	229
6.8.2 Registers	231

The GPIO port peripheral implements up to 32 pins, PIN0 through PIN31. Each of these pins can be individually configured in the PIN_CNF[n] registers (n=0..31).

The following parameters can be configured through these registers:

- Direction
- Drive strength



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

6.8.2 Registers

Instances

Instance	Base address	Description
GPIO	0x50000000	General purpose input and output This instance is deprecated.
P0	0x50000000	General purpose input and output, port 0
P1	0x50000300	General purpose input and output, port 1

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

Register	Offset	Description
OUT	0x504	Write GPIO port
OUTSET	0x508	Set individual bits in GPIO port
OUTCLR	0x50C	Clear individual bits in GPIO port
IN	0x510	Read GPIO port
DIR	0x514	Direction of GPIO pins
DIRSET	0x518	DIR set register
DIRCLR	0x51C	DIR clear register
LATCH	0x520	Latch register indicating what GPIO pins that have met the criteria set in the LATCH registers
DETECTMODE	0x524	Select between default DETECT signal behavior and LDETECT mode
PIN_CNF[14]	0x738	Configuration of GPIO pins
PIN_CNF[15]	0x73C	Configuration of GPIO pins
PIN_CNF[16]	0x740	Configuration of GPIO pins

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.

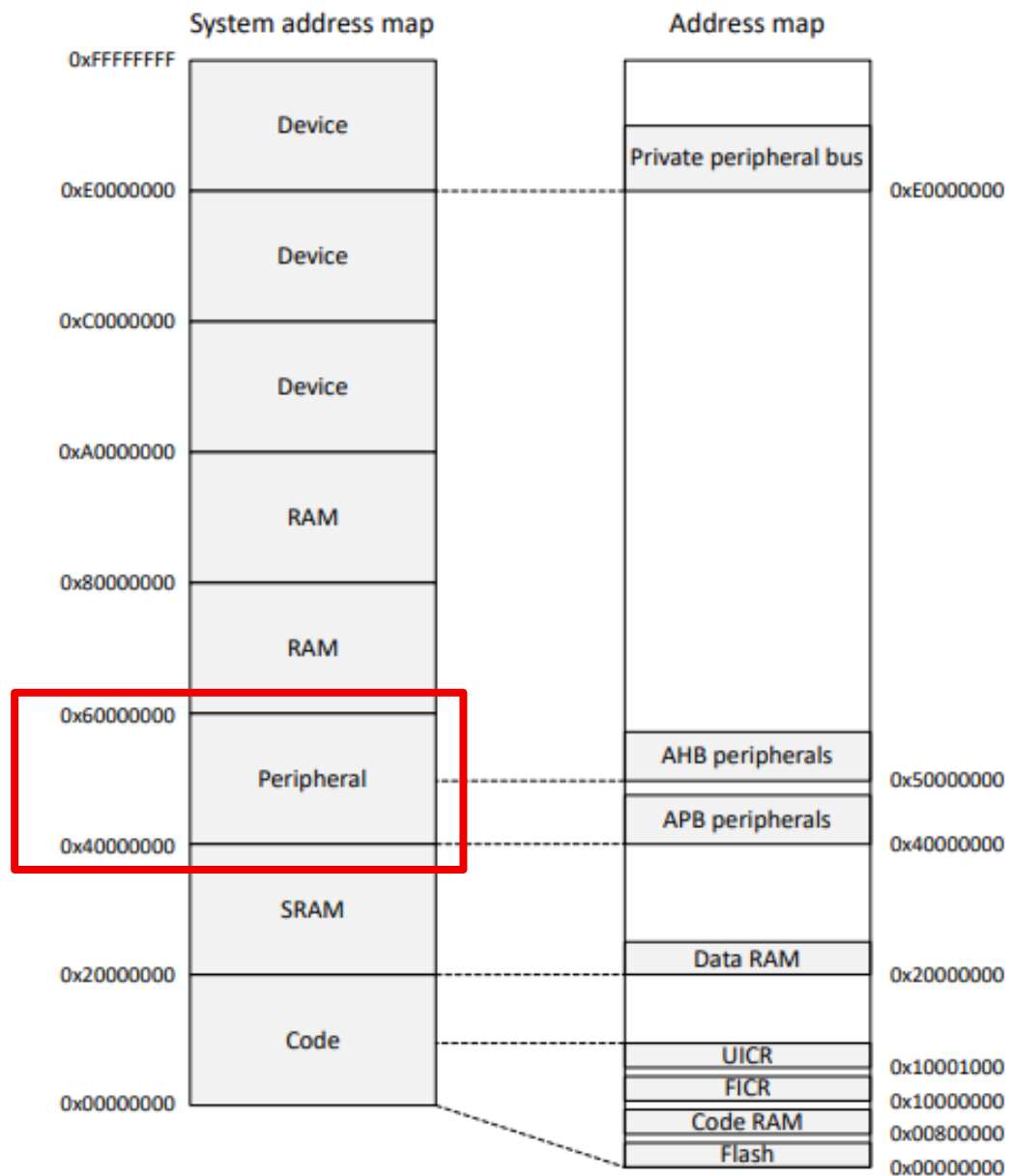


Figure 3: Memory map

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).

6.8.2.25 PIN_CNF[15]

Address offset: 0x73C

Configuration of GPIO pins

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID																																
Reset 0x00000002	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
ID	R/W	Field	Value ID	Value	Description																											
A	RW	DIR	Input	0	Pin direction. Same physical register as DIR register																											
			Output	1	Configure pin as an output pin																											
B	RW	INPUT	Connect	0	Connect or disconnect input buffer																											
			Disconnect	1	Connect input buffer																											
C	RW	PULL	Disabled	0	Disconnect input buffer																											
			Pulldown	1	Pull configuration																											
			Pullup	3	No pull																											
D	RW	DRIVE	S0S1	0	Drive configuration																											
			H0S1	1	Standard '0', standard '1'																											
			S0H1	2	High drive '0', standard '1'																											
			H0H1	3	Standard '0', high drive '1'																											
			D0S1	4	High drive '0', high drive '1'																											
D0H1	5	Disconnect '0' standard '1' (normally used for wired-or connections)																														
					Disconnect '0', high drive '1' (normally used for wired-or connections)																											

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

- A defines the direction of the pin. 0 for input and 1 for output.
- 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:

6.8.2.1 OUT

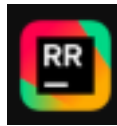
Address offset: 0x504

Write GPIO port

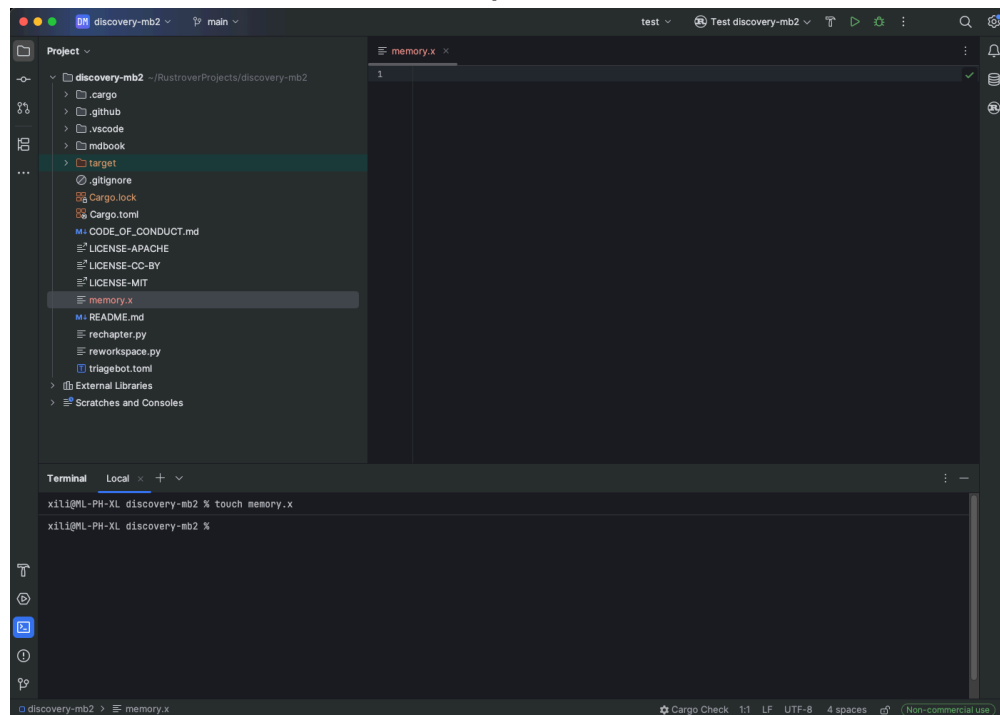
Bit number				31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ID				f	e	d	c	b	a	z	y	x	w	v	u	t	s	r	q	p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a			
Reset 0x00000000				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
ID	R/W	Field	Value ID	Value		Description																																
A	RW	PIN0				Pin 0																																
			Low	0		Pin driver is low																																
			High	1		Pin driver is high																																
B	RW	PIN1				Pin 1																																
			Low	0		Pin driver is low																																
			High	1		Pin driver is high																																
C	RW	PIN2				Pin 2																																
			Low	0		Pin driver is low																																
			High	1		Pin driver is high																																
D	RW	PIN3				Pin 3																																
			Low	0		Pin driver is low																																
			High	1		Pin driver is high																																
E	RW	PIN4				Pin 4																																
			Low	0		Pin driver is low																																
			High	1		Pin driver is high																																

The address is _____.

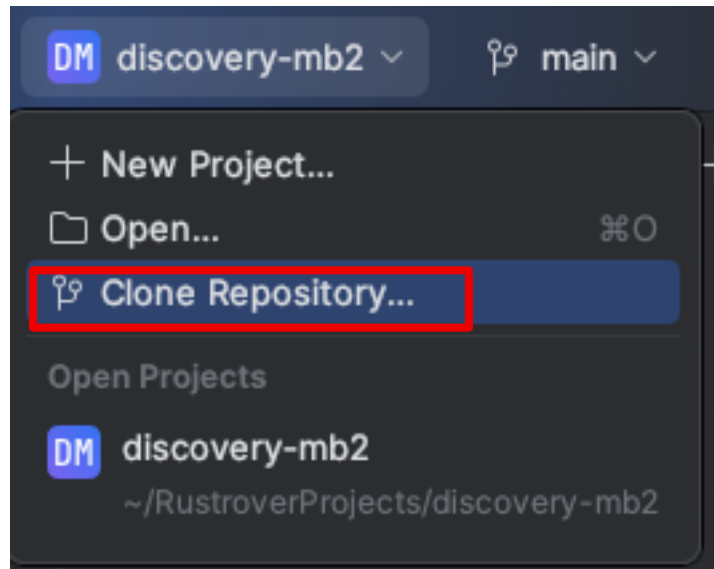
Now we need to look for the value sent to this address, e.g., for PIN4 it's E, Bit number 4, and 1 will drive it high (3.3V). Write down the Bit number for your two PINs. Now we know (to config and to control,) the specific address we need to send to, and the specific value we need to send. What's next?



1.7) Rust Rover warm-ups for windows:



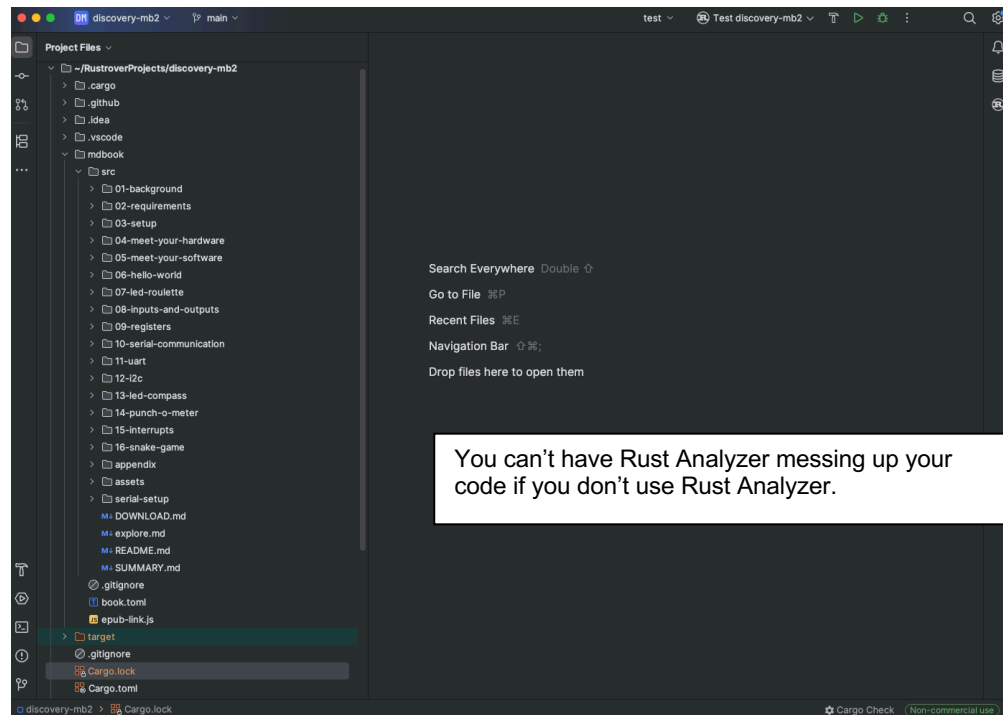
Time to start your Rust embedded environment in Rust Rover. Let's start from the beginning and:



The address for the repo is:

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

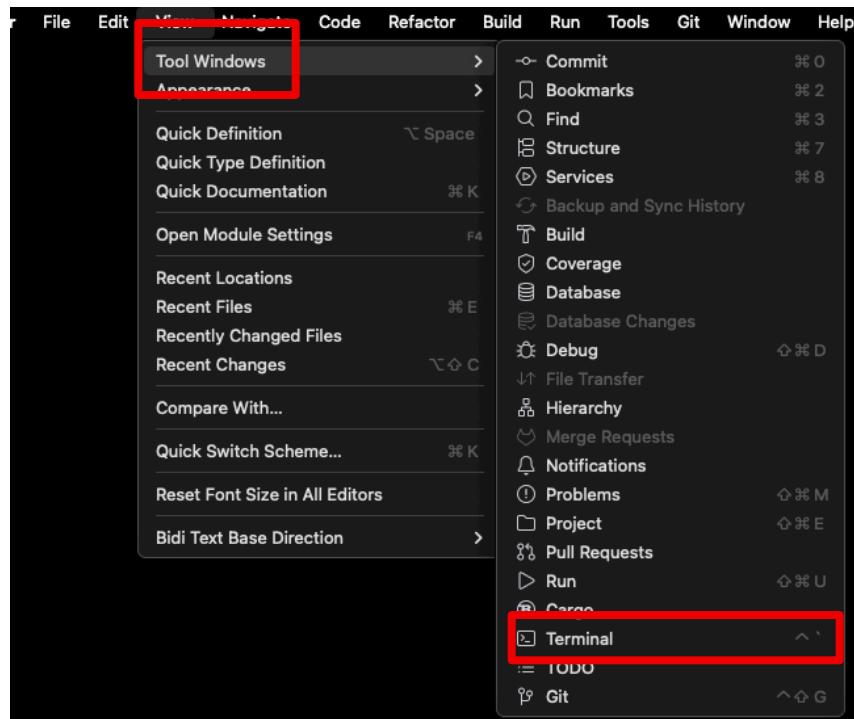
Rust Rover will ask where you would like to put the folder in your PC. Pick a spot and start cloning. If everything works well, you would see the DISCOVERY-MB2 folder shown up as follows:



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:

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src> cd 06-hello-world
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\06-hello-world> cargo check
    Compiling cortex-m v0.7.7
    Checking nb v1.1.0
    Compiling bare-metal v0.2.5
    Checking stable_deref_trait v1.2.0
    Checking byteorder v1.5.0
    Checking void v1.0.2
    Checking nb v0.1.3
    Checking vcell v0.1.3
    Compiling cortex-m-rt v0.7.5
    Checking embedded-hal v0.2.7
    Checking hash32 v0.3.1
    Checking volatile-register v0.2.2
    Compiling portable-atomic v1.11.1
    Compiling heapless v0.8.0
    Checking bitfield v0.13.2
    Compiling typenum v1.18.0
    Compiling az v1.2.1
    Checking cfg-if v1.0.3
    Checking half v2.6.0
    Compiling nrf52833-pac v0.12.2
    Checking usb-device v0.3.2
    Compiling fixed v1.29.0
    Checking bytemuck v1.23.2
    Checking critical-section v1.2.0
    Compiling nrf-hal-common v0.18.0
    Checking nrf-usb v0.3.0
    Checking embedded-dma v0.2.0
    Checking embedded-io v0.6.1
    Checking cast v0.3.0
    Checking rand_core v0.6.4
    Checking embedded-hal v1.0.0
    Checking embedded-storage v0.3.1
    Compiling nrf52833-hal v0.18.0
    Checking tiny-led-matrix v1.0.2
    Checking panic-halt v1.0.0
    Checking microbit-common v0.15.1
    Checking microbit-v2 v0.15.1
    Checking hello-world v0.1.0 (C:\Users\YandXandTed\discovery-mb2\mdbook\src\06-hello-world)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 28.57s
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\06-hello-world> 
```

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

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\06-hello-world> probe-rs list
The following debug probes were found:
[0]: BBC micro:bit CMSIS-DAP -- 0d28:0204:9906360200052820541965c4bfd8d2bf000000000e052820 (CMSIS-DAP)
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\06-hello-world> cargo embed
Compiling nb v1.1.0
Compiling byteorder v1.5.0
Compiling void v1.0.2
Compiling stable_deref_trait v1.2.0
Compiling nb v0.1.3
Compiling vcell v0.1.3
Compiling bare-metal v0.2.5
Compiling hash32 v0.3.1
Compiling embedded-hal v0.2.7
Compiling volatile-register v0.2.2
Compiling bitfield v0.13.2
Compiling heapless v0.8.0
Compiling portable-atomic v1.11.1
Compiling cfg-if v1.0.3
Compiling cortex-m v0.7.7
Compiling half v2.6.0
Compiling cortex-m-rt v0.7.5
Compiling usb-device v0.3.2
Compiling typenum v1.18.0
Compiling az v1.2.1
Compiling bytemuck v1.23.2
Compiling critical-section v1.2.0
Compiling nrf52833-pac v0.12.2
Compiling nrf-usb v0.3.0
Compiling embedded-dma v0.2.0
Compiling embedded-storage v0.3.1
Compiling fixed v1.29.0
Compiling embedded-io v0.6.1
Compiling rand_core v0.6.4
Compiling cast v0.3.0
Compiling embedded-hal v1.0.0
Compiling tiny-led-matrix v1.0.2
Compiling panic-halt v1.0.0
Compiling nrf-hal-common v0.18.0
Compiling nrf52833-hal v0.18.0
Compiling microbit-common v0.15.1
Compiling microbit-v2 v0.15.1
Compiling hello-world v0.1.0 (C:\Users\YandXandTed\discovery-mb2\mdbook\src\06-hello-world)
Finished 'dev' profile [unoptimized + debuginfo] target(s) in 31.48s
Config default
Target C:\Users\YandXandTed\discovery-mb2\target\thumbv7em-none-eabi\debug\hello-world
Erasing ✓ 100% [#####] 20.00 KiB @ 31.42 KiB/s (took 1s)
Programming ✓ 100% [#####] 20.00 KiB @ 19.17 KiB/s (took 1s)
Finished in 1.69s
Done processing config default
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\06-hello-world> 
```

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

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

```

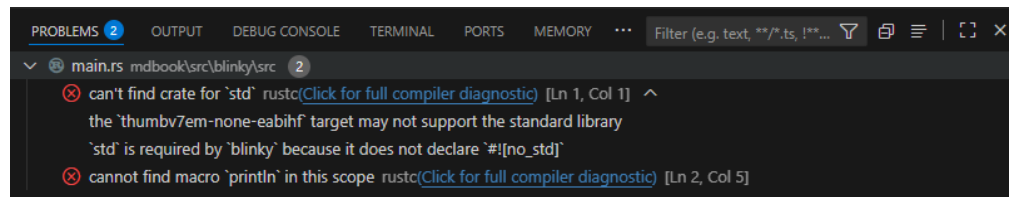
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\06-hello-world> cd..
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src> cargo new blinky
    Creating binary (application) `blinky` package
    Adding `blinky` as member of workspace at `C:\Users\YandXandTed\discovery-mb2`
note: see more `Cargo.toml` keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src> cd blinky
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky> 

```

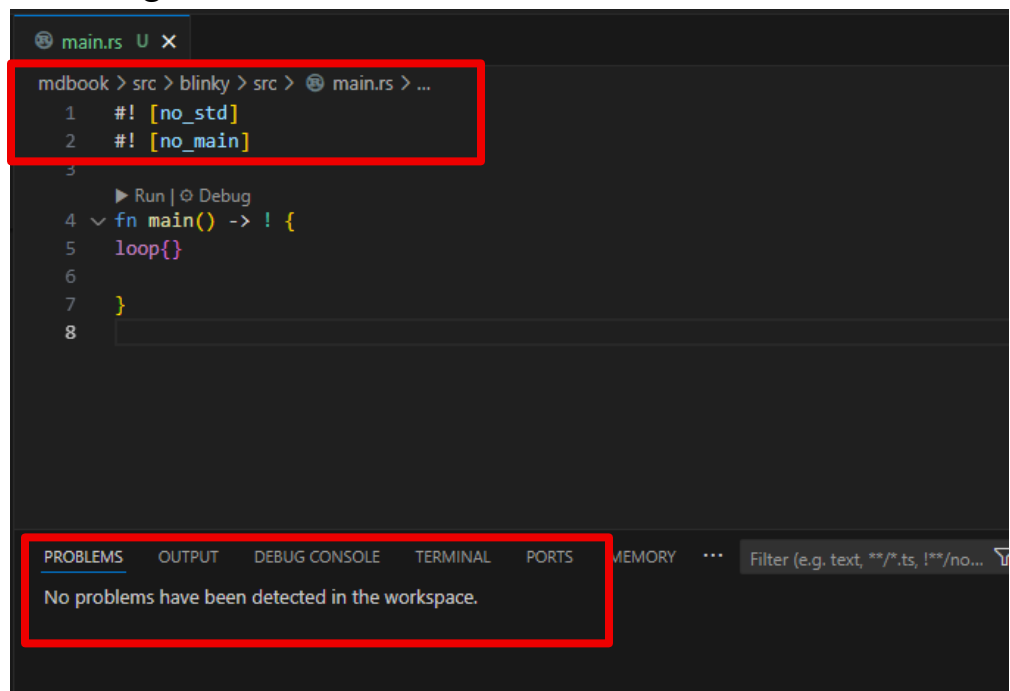
The new project folder “blink” is red, that’s ok.



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:

```
error: current package believes it's in a workspace when it's not:
current:  C:\Users\VandXandTed\discovery-mb2\mdbook\src\blinkyPAC\Cargo.toml
workspace: C:\Users\VandXandTed\discovery-mb2\Cargo.toml

this may be fixable by adding 'mdbook\src\blinkyPAC' to the 'workspace.members' array of the manifest located at: C:\Users\VandXandTed\discovery-mb2\Cargo.toml
Alternatively, to keep it out of the workspace, add the package to the 'workspace.exclude' array, or add an empty '[workspace]' table to the package's manifest.
PS C:\Users\VandXandTed\discovery-mb2\mdbook\src\blinkyPAC> []
```

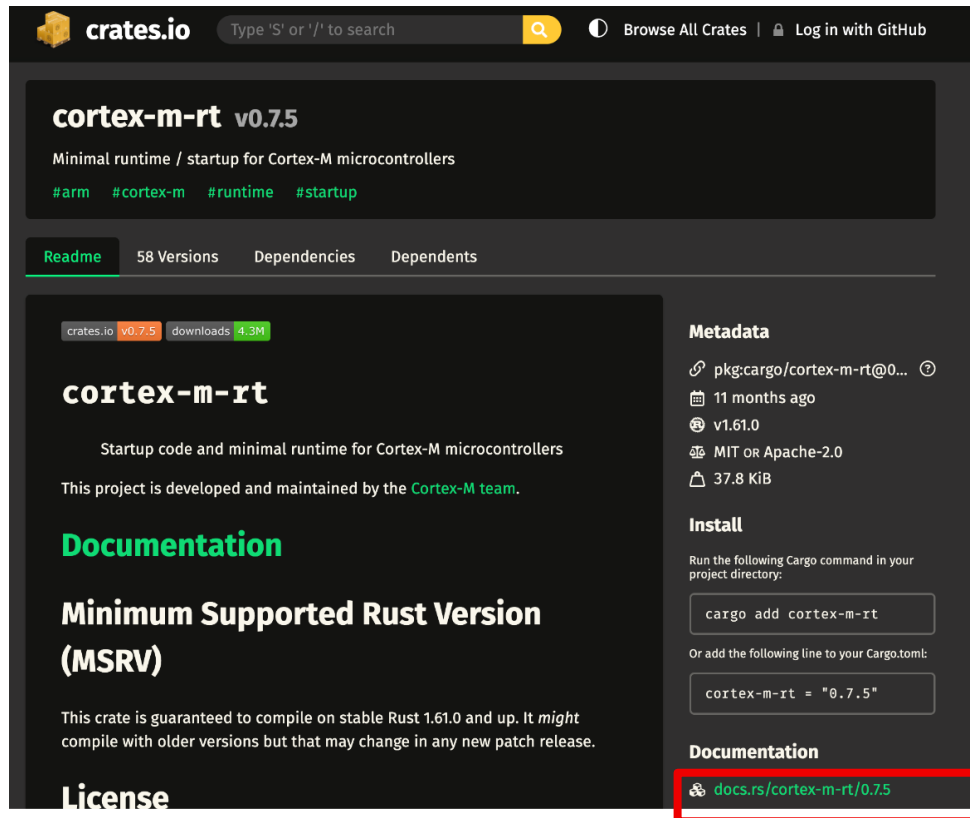
Go to root \discovery-mb2\Cargo.toml, and add your new folder to the member list:

```
main.rs U Cargo.toml M
Cargo.toml
3  members = [
4      "mdbook/src/01-meet-your-software",
5      "mdbook/src/02-creating-a-new-project",
6      "mdbook/src/06-hello-world",
7      "mdbook/src/07-led-roulette",
8      "mdbook/src/08-inputs-and-outputs",
9      "mdbook/src/09-registers",
10     "mdbook/src/11-uart",
11     "mdbook/src/12-i2c",
12     "mdbook/src/13-led-compass",
13     "mdbook/src/14-punch-o-meter",
14     "mdbook/src/15-interrupts",
15     "mdbook/src/16-snake-game",
16     "mdbook/src/appendix/3-mag-calibration", "mdbook/src/blinky",
17     "mdbook/src/serial-setup",
18     "mdbook/src/blinky",
19     "mdbook/src/blinky002",
20     "mdbook/src/blinkyPAC",
21 ]
22
23 [profile.release]
24 codegen-units = 1
25 debug = false
```

1.7.5) Now it's time to install stuff (add dependencies) that knows how to talk to the MCU. Go to crates.io

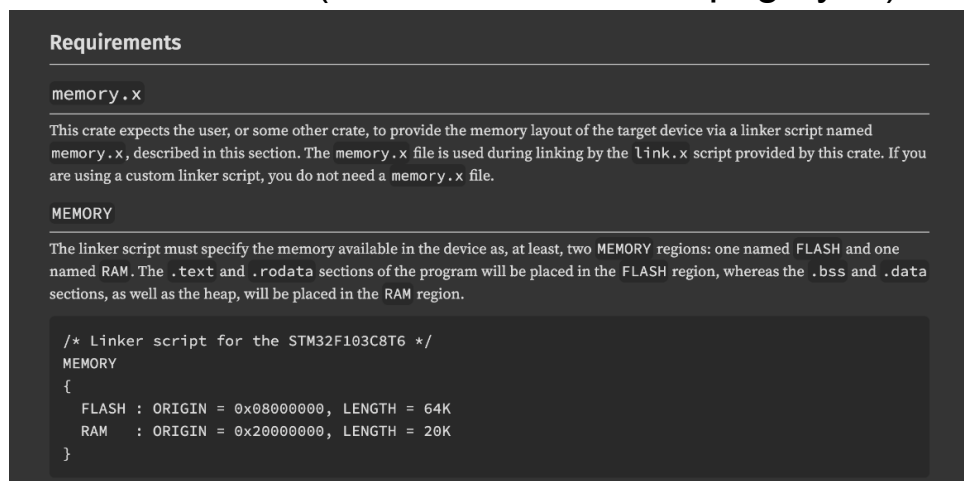
 crates.io

1.7.6) Search cortex-m-rt, and click on the documentation link:



The screenshot shows the crates.io page for the `cortex-m-rt` crate, version `v0.7.5`. The page includes a search bar at the top, a description of the crate as a "Minimal runtime / startup for Cortex-M microcontrollers", and tabs for "Readme", "58 Versions", "Dependencies", and "Dependents". The "Readme" tab is active, showing the crate's name, description, and a "Documentation" link. The "Documentation" link is highlighted with a red box. The "Metadata" section on the right shows the package name, version, license, and size. The "Install" section provides instructions on how to add the crate to a project. The "Documentation" section at the bottom right shows the link to the documentation page.

1.7.7) 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.)



The screenshot shows the "Requirements" section of the `cortex-m-rt` documentation. It describes the `memory.x` file and provides an example linker script for the STM32F103C8T6 MCU. The script defines the FLASH and RAM regions with their origins and lengths.

```
/* Linker script for the STM32F103C8T6 */
MEMORY
{
  FLASH : ORIGIN = 0x08000000, LENGTH = 64K
  RAM   : ORIGIN = 0x20000000, LENGTH = 20K
}
```

As for the origin, based on the memory map:

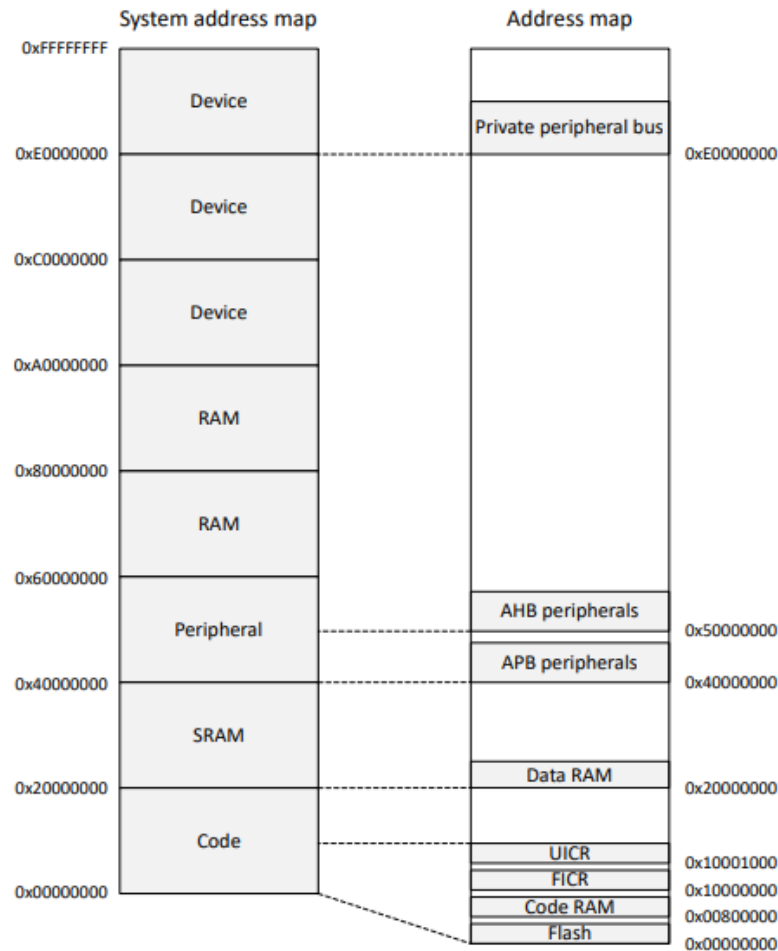


Figure 3: Memory map

Flash starts at: _____

RAM starts at: _____

1.7.8) Type “**cargo add cortex-m-rt**” in the terminal:

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blink> cargo add cortex-m-rt
Updating crates.io index
Adding cortex-m-rt v0.7.5 to dependencies
Features:
- device
- paint-stack
- set-sp
- set-vtor
- zero-init-ram
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blink> █
```

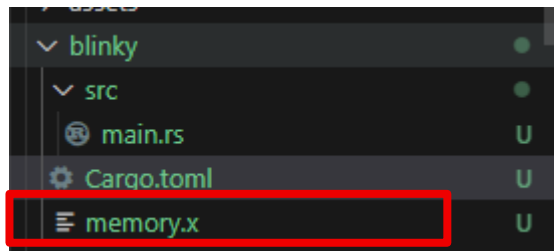
It will also be added to the Cargo.toml as dependencies:

```
main.rs U Cargo.toml U X
mdbook > src > blinky > Cargo.toml
1 [package]
2 name = "blinky"
3 version = "0.1.0"
4 edition = "2024"
5
6 [dependencies]
7 cortex-m-rt = "0.7.5"
8
```

1.7.9) Now add the needed memory.x file by type “type memory.x” in the terminal:

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky> type memory.x
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky> █
```

Look, new file appears in the folder:

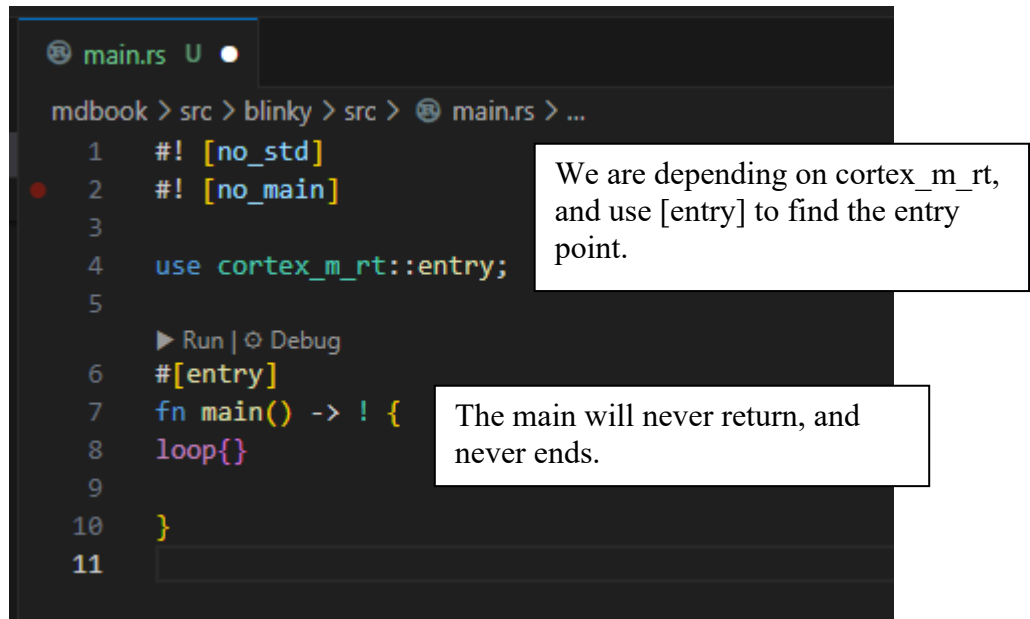


1.7.10) Copy-paste the requirements into the file, and modify the ORIGIN address and size accordingly:

```
main.rs U memory.x U
mdbook > src > 18-xli-blinky > memory.x
1 /* Linker script for the Microbit */
2 MEMORY
3 {
4     FLASH : ORIGIN = 0x00000000, LENGTH = 512K
5     RAM : ORIGIN = 0x20000000, LENGTH = 128K
6 }
```

```
/* Linker script for the Microbit v2 */
MEMORY
{
    FLASH : ORIGIN = 0x00000000, LENGTH = 512K
    RAM : ORIGIN = 0x20000000, LENGTH = 128K
}
```

1.7.11) And modify the main.rs to use this handy tool:



```
main.rs U ●
mdbook > src > blinky > src > main.rs > ...
1  #! [no_std]
2  #! [no_main]
3
4  use cortex_m_rt::entry;
5
6  #[entry]
7  fn main() -> ! {
8      loop{}
9
10 }
11
```

We are depending on cortex_m_rt, and use [entry] to find the entry point.

The main will never return, and never ends.

1.7.12) The house keeping work has already been done in the global `.cargo.config.toml` and `.vscode/settings.json`, all good now.

1.7.13) Fix the missing “Panic handler”, which is reserved by Rust for fatal errors.

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky> cargo check
Checking blinky v0.1.0 (C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky)
error: `#[panic_handler]` function required, but not found

error: could not compile `blinky` (bin "blinky") due to 1 previous error
```

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`”

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky> cargo add panic_halt
Updating crates.io index
warning: translating `panic_halt` to `panic-halt`
Adding panic-halt v1.0.0 to dependencies
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky>
```


Update the main.rs:

```
main.rs U X
mdbook > src > 18-xli-blinky > src > main.rs > ...
1  #![no_std]
2  #![no_main]
3
4  use cortex_m_rt::entry;
5  use panic_halt as _;
6
7  #[entry]
8  fn main() -> ! {
9      loop {}
10 }
11
```

1.7.14) Now if you type “**cargo check**”, everything should be fine now:

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky> cargo check
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.17s
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky> 
```

1.7.15) 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

```
\discovery-mb2> 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`.

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky002> cargo clean
Removed 190 files, 73.0MiB total
```

1.7.16) 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`”

```
PS C:\Users\YandXandTed\discovery-mb2\mdbook\src\blinky> cargo add cortex_m
Updating crates.io index
warning: translating `cortex_m` to `cortex-m`
Adding cortex-m v0.7.7 to dependencies
Features:
- cm7
- cm7-r0p1
- critical-section
- critical-section-single-core
- inline-asm
- linker-plugin-lto
- serde
- std
```

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

```
Cargo.toml U x  main.rs U
mdbook > src > blinky002 > Cargo.toml
1  [package]
2  name = "blinky002"
3  version = "0.1.0"
4  edition = "2024"
5
6  [dependencies]
7  cortex-m = "0.7.7"
8  cortex-m-rt = "0.7.5"
9  panic-halt = "1.0.0"
10
11
```

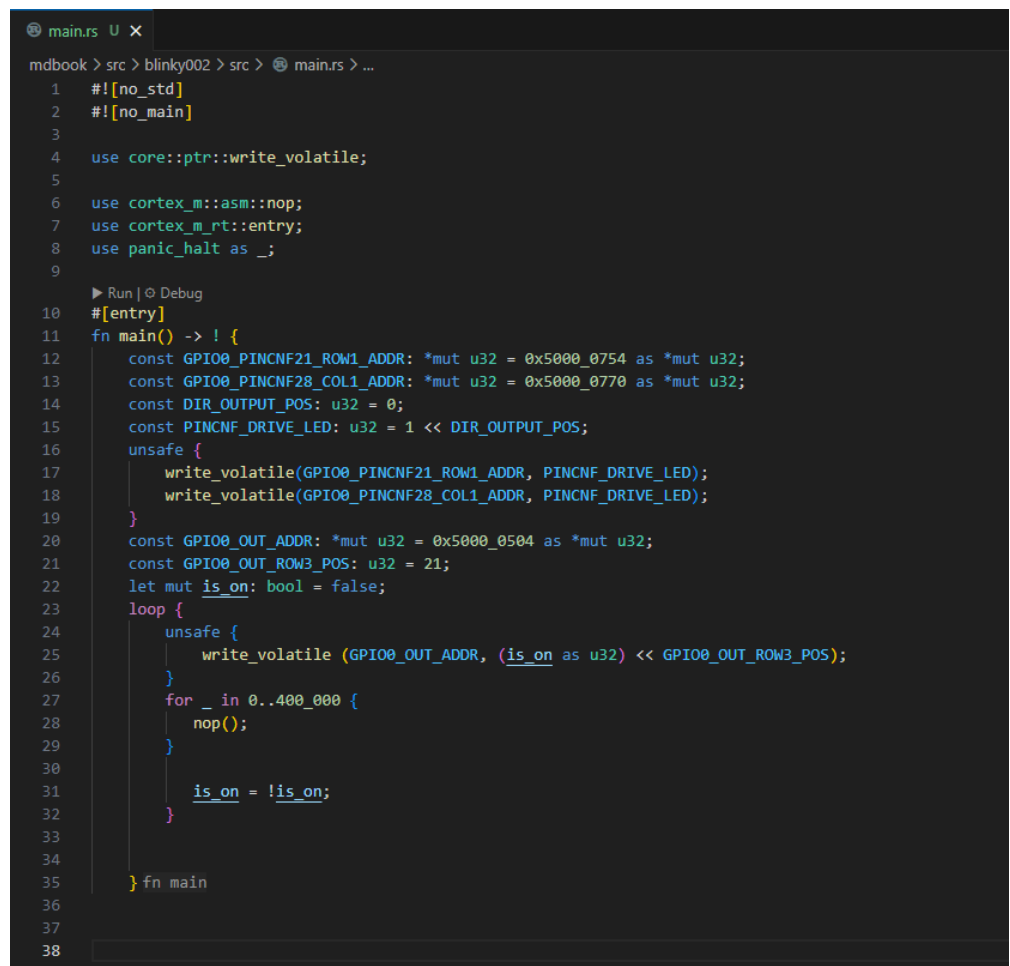
1.7.17) Unlike C, Rust don't like Borrow Checker that can't be verified or validated. That's why we need to put everything that appears *unsafe* to Rust in an

`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:



```
main.rs U x
mdbook > src > blinky002 > src > main.rs > ...
1  #![no_std]
2  #![no_main]
3
4  use core::ptr::write_volatile;
5
6  use cortex_m::asm::nop;
7  use cortex_m_rt::entry;
8  use panic_halt as _;
9
10 ▶ Run | ⏏ Debug
11 #[entry]
12 fn main() -> ! {
13     const GPIO0_PINCNF21_ROW1_ADDR: *mut u32 = 0x5000_0754 as *mut u32;
14     const GPIO0_PINCNF28_COL1_ADDR: *mut u32 = 0x5000_0770 as *mut u32;
15     const DIR_OUTPUT_POS: u32 = 0;
16     const PINCNF_DRIVE_LED: u32 = 1 << DIR_OUTPUT_POS;
17     unsafe {
18         write_volatile(GPIO0_PINCNF21_ROW1_ADDR, PINCNF_DRIVE_LED);
19         write_volatile(GPIO0_PINCNF28_COL1_ADDR, PINCNF_DRIVE_LED);
20     }
21     const GPIO0_OUT_ADDR: *mut u32 = 0x5000_0504 as *mut u32;
22     const GPIO0_OUT_ROW3_POS: u32 = 21;
23     let mut is_on: bool = false;
24     loop {
25         unsafe {
26             write_volatile(GPIO0_OUT_ADDR, (is_on as u32) << GPIO0_OUT_ROW3_POS);
27         }
28         for _ in 0..400_000 {
29             nop();
30         }
31         is_on = !is_on;
32     }
33 }
34 fn main
35
36
37
38
```

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();
        }

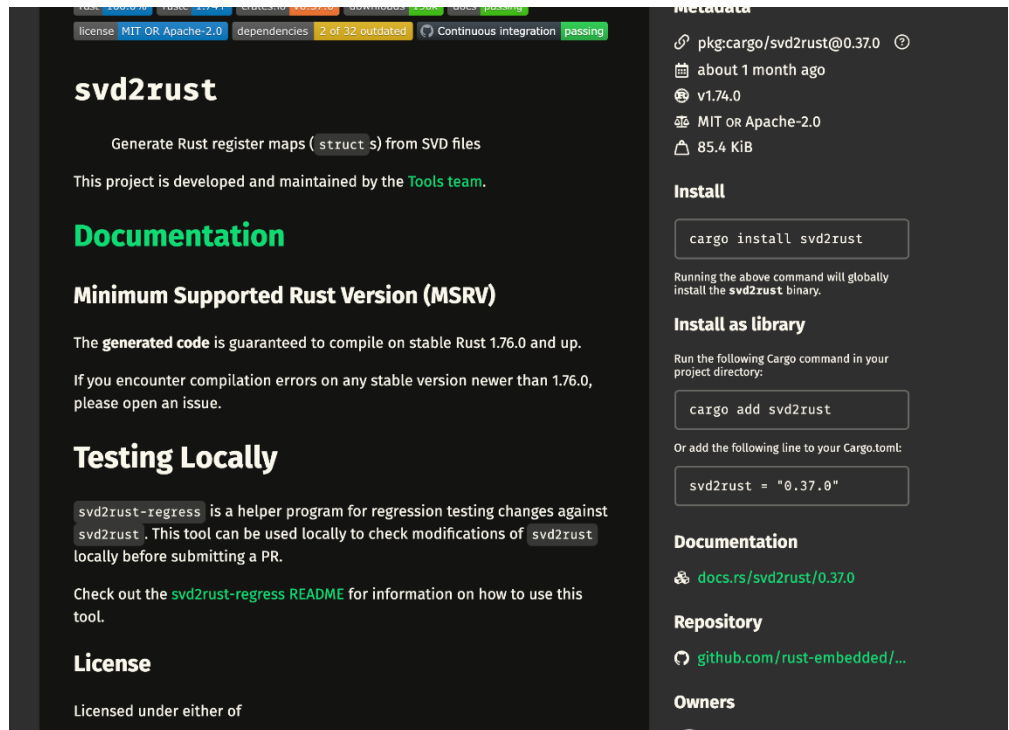
        is_on = !is_on;
    }
}
```

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's why we are moving on to the next chapter:

2) PAC (Peripheral Access Crate)

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



The screenshot shows the crates.io page for the **svd2rust** crate. The page is dark-themed and contains the following sections:

- svd2rust**: The crate name.
- Generate Rust register maps (structs) from SVD files**: A brief description of the crate's purpose.
- This project is developed and maintained by the Tools team.**: Information about the maintainers.
- Documentation**: A link to the documentation.
- Minimum Supported Rust Version (MSRV)**: A section stating that the generated code is guaranteed to compile on stable Rust 1.76.0 and up, and that users should open an issue if they encounter compilation errors on any stable version newer than 1.76.0.
- Testing Locally**: A section explaining that `svd2rust-regress` is a helper program for regression testing changes against `svd2rust`, and that users should check out the `svd2rust-regress README` for information on how to use this tool.
- License**: A section stating that the crate is licensed under either of the MIT OR Apache-2.0 licenses.
- Metadata**: A section containing the following information:
 - pkg:cargo/svd2rust@0.37.0**: The crate's identifier.
 - about 1 month ago**: The crate's last update time.
 - v1.74.0**: The crate's version number.
 - MIT OR Apache-2.0**: The crate's license.
 - 85.4 KiB**: The crate's size.
- Install**: A section with a button that says `cargo install svd2rust`.
- Install as library**: A section with a button that says `cargo add svd2rust`.
- Documentation**: A link to `docs.rs/svd2rust/0.37.0`.
- Repository**: A link to `github.com/rust-embedded/...`.
- Owners**: A section listing the crate's owners.

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 {}
}

```

3) 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?