Blinky with Rust embedded (v1.0)
Sep 2025 ENR325

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 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.

```
fn main() {
    let z = 13;
    let x = {
        let y = 10;
        dbg!(y);
        z - y
    };
    dbg!(x);
    // dbg!(y);
}
```

```
[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:

```rust
fn main() {
    let x = 10;
    if x == 0 {
        println!("zero!");
    } else if x < 100 {
        println!("biggish");
    } else {
        println!("huge");
    }
}
```

But it can also be used as an expression:

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

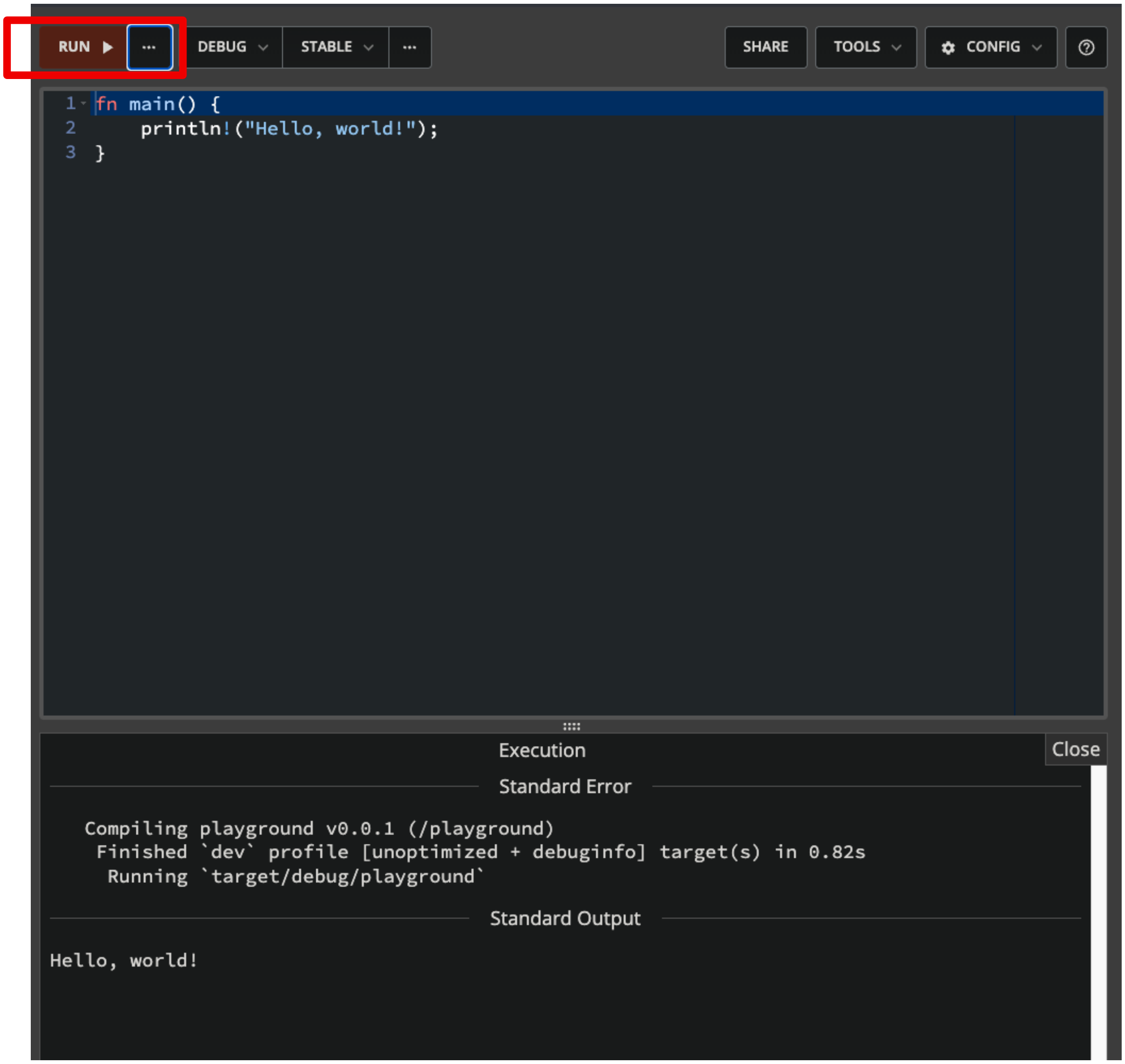
Neet, yeah? Today, we are going to do more “Hello World” in embedded: blink LED lights.

1) Blink LED, bare metal style

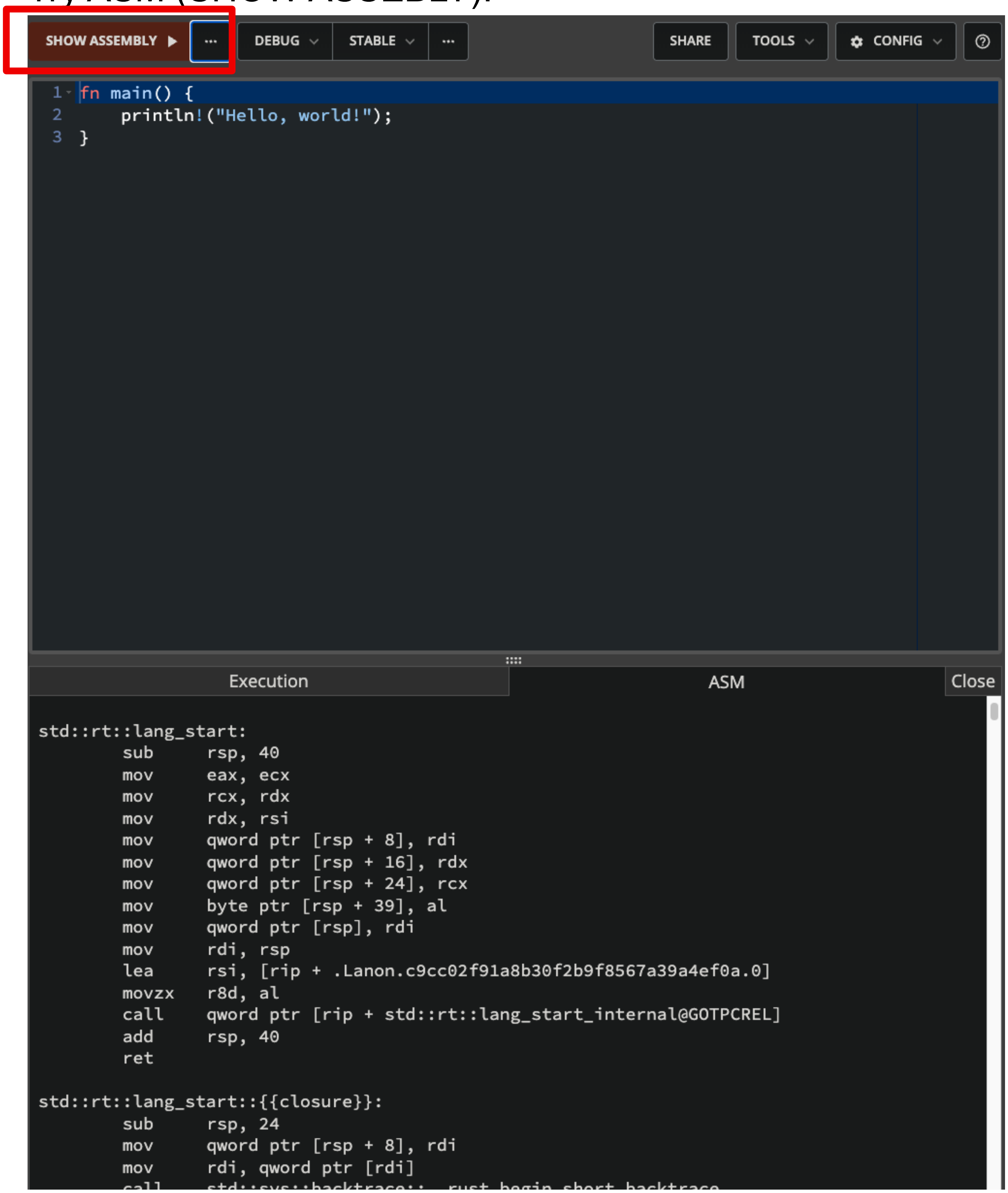
--- It’s called bare metal because it’s the lowest level of software control on an MCU.

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:

1.1) Google “microbit v2 schematics”.

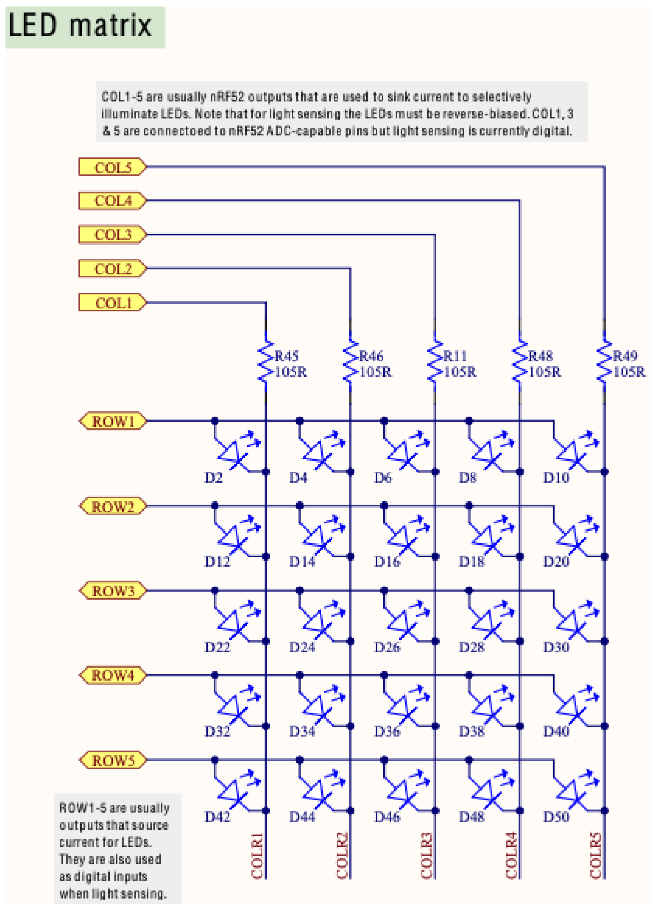
1.2) Go to the github page hosting the hardware info.

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

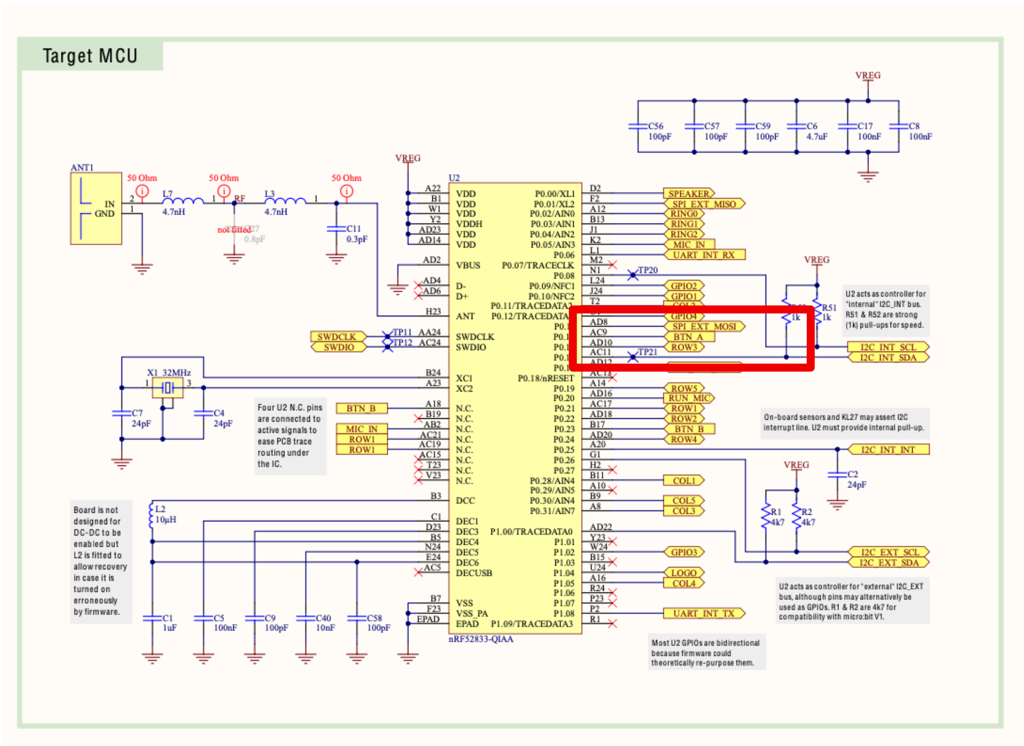
1.3) On their github page, find the schematic pdf, download and save the file on your PC, then open it.

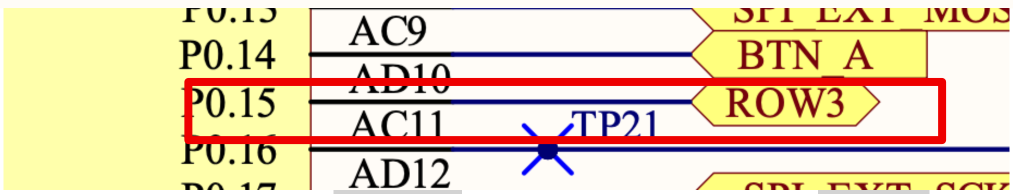
MicroBit_V2.0.0_S_schematic.PDF   Add micro:bit V2.2 schematic and BOM

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.



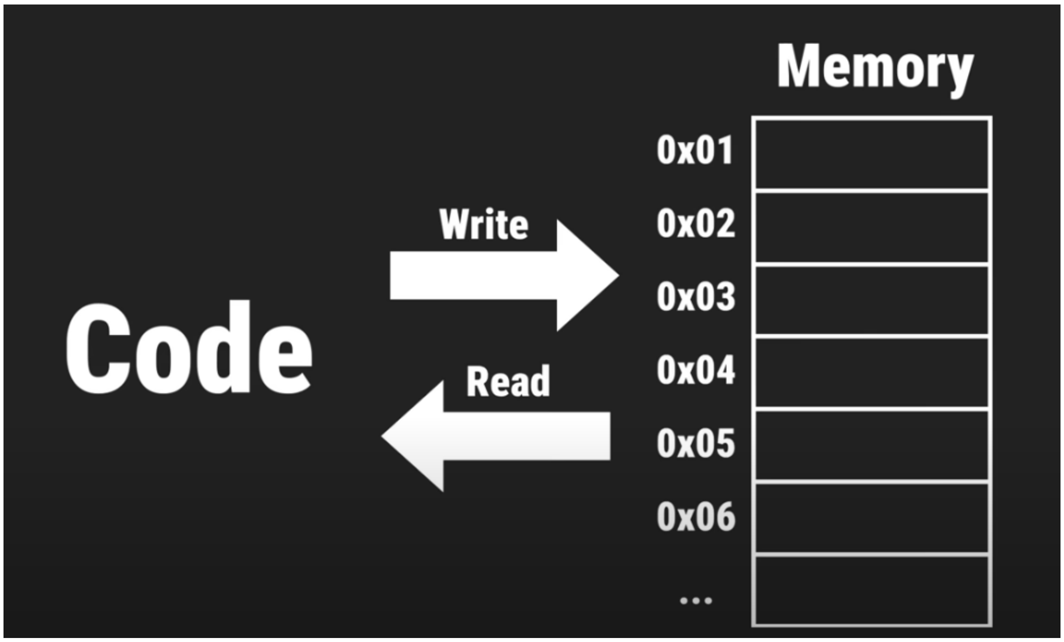


For example, ROW3 was connected to P0.15.

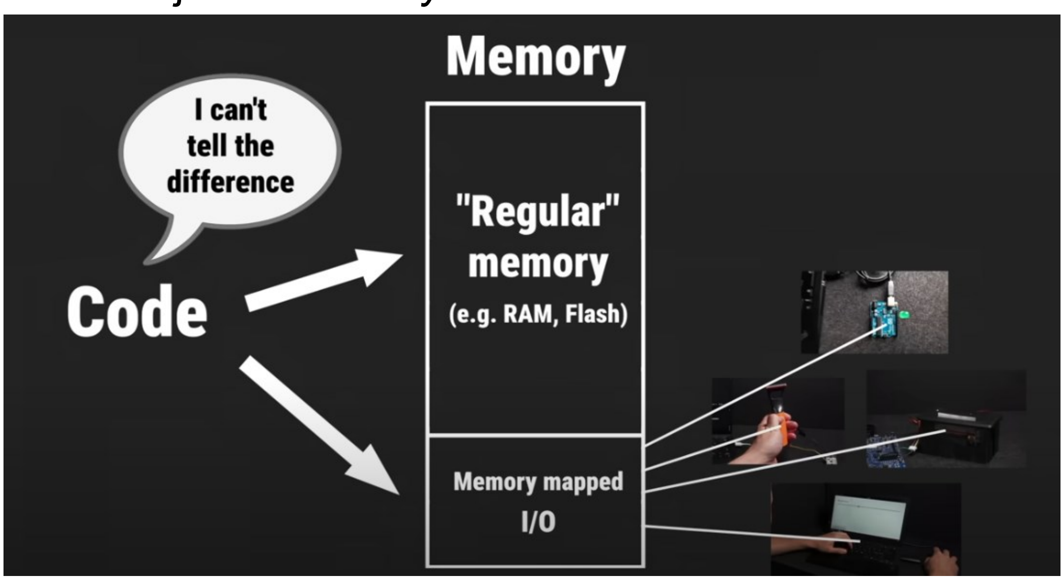
So now, how do we code and drive 3.3V through two pins? We are going to use a *magic* spell called MMIO.

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(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):

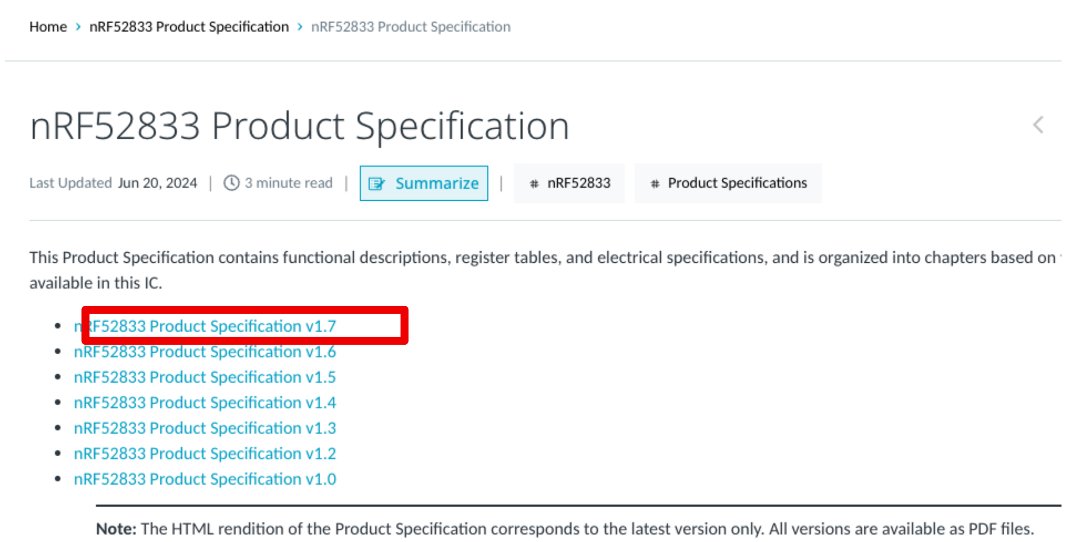


MMIO is just a *Memory* dedicated to a hardware



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.

1.6) 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.


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.

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

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

4452_021 v1.7 228

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 se 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.

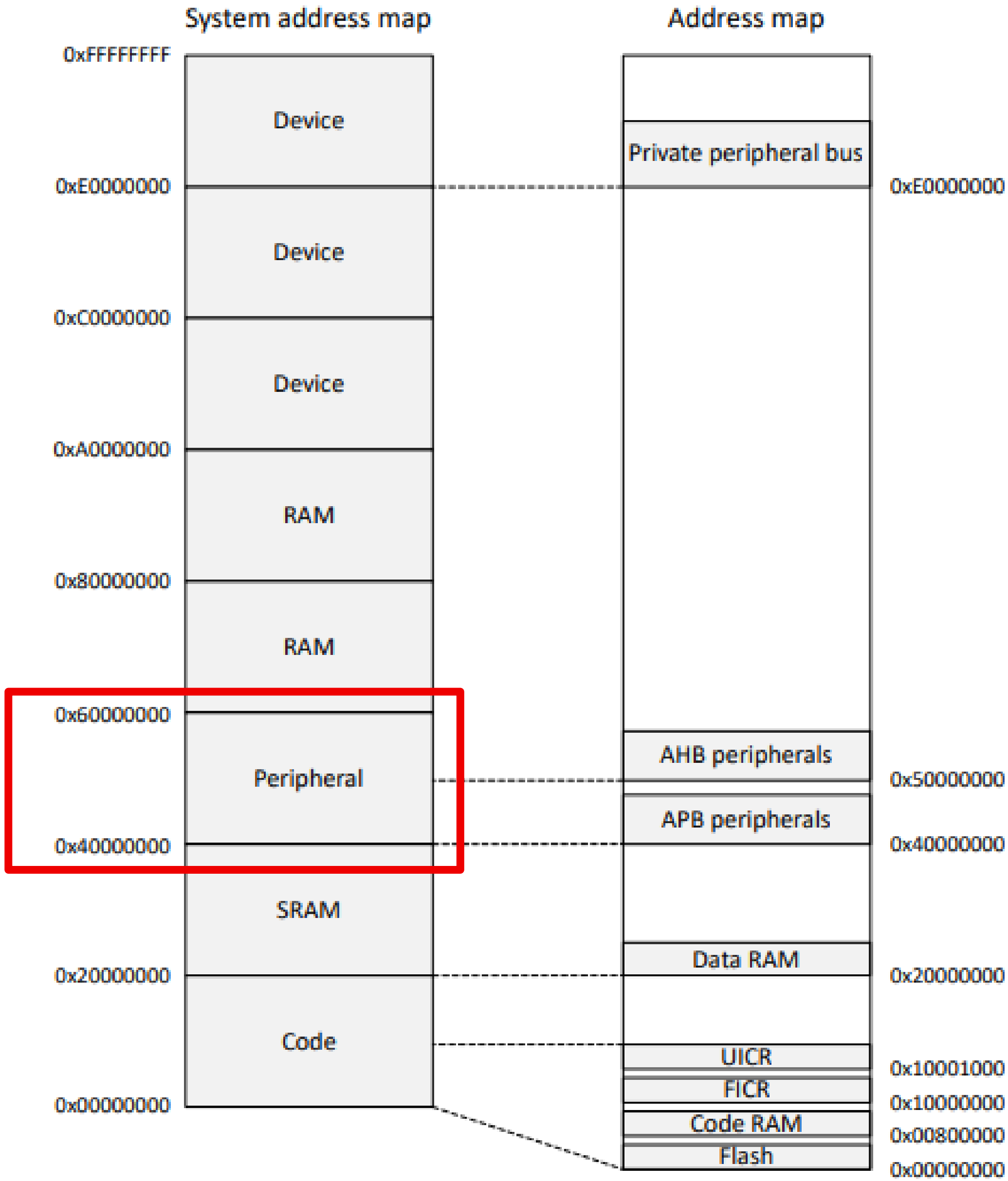


Figure 3: Memory map

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															E	E						D	D	D					C	C	B	A
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			Pin direction. Same physical register as DIR register
			Input	0	Configure pin as an input pin
			Output	1	Configure pin as an output pin
B	RW	INPUT			Connect or disconnect input buffer
			Connect	0	Connect input buffer
			Disconnect	1	Disconnect input buffer
C	RW	PULL			Pull configuration
			Disabled	0	No pull
			Pulldown	1	Pull down on pin
			Pullup	3	Pull up on pin
D	RW	DRIVE			Drive configuration
			S0S1	0	Standard '0', standard '1'
			H0S1	1	High drive '0', standard '1'
			S0H1	2	Standard '0', high drive '1'
			H0H1	3	High drive '0', high 'drive '1''
			D0S1	4	Disconnect '0' standard '1' (normally used for wired-or connections)
			D0H1	5	Disconnect '0', high drive '1' (normally used for wired-or connections)

4452_021 v1.7 265 NORDIC SEMICONDUCTOR

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

i) A defines the direction of the pin. 0 for input and 1 for output.

ii) 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) Time to start your Rust embedded environment in VS code again. Remember *Cargo sth sth* and *probe-rs* we did last time?

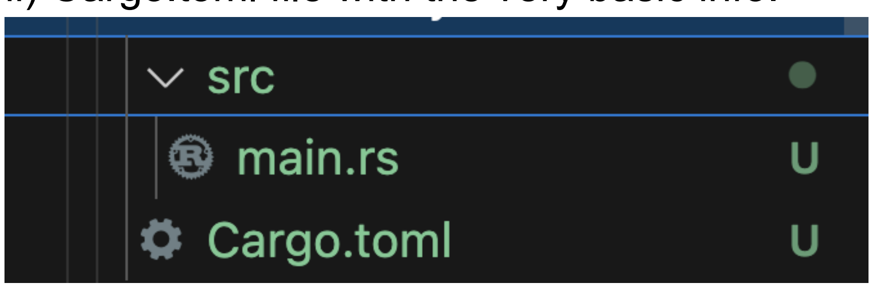
1.7.1) Open the DISCOVERY-MB2 folder.

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.

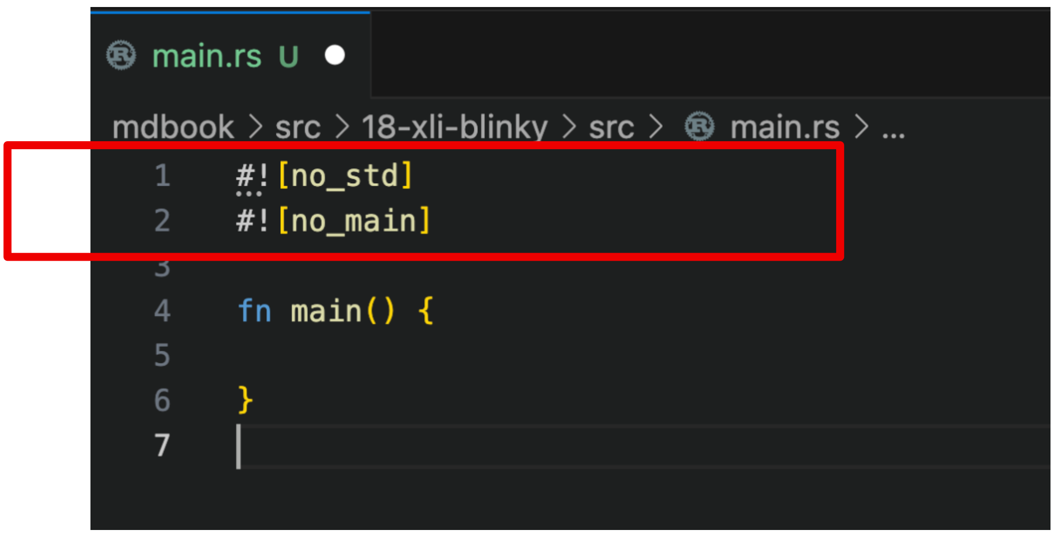
1.7.4) You can also change the name of the new folder by the command "mv <old name> <new name>".

1.7.5) Go to the new folder. You will see there's only minimum number of files:
i) A "hello world" in the src/main.rs.
ii) Cargo.toml file with the very basic info.



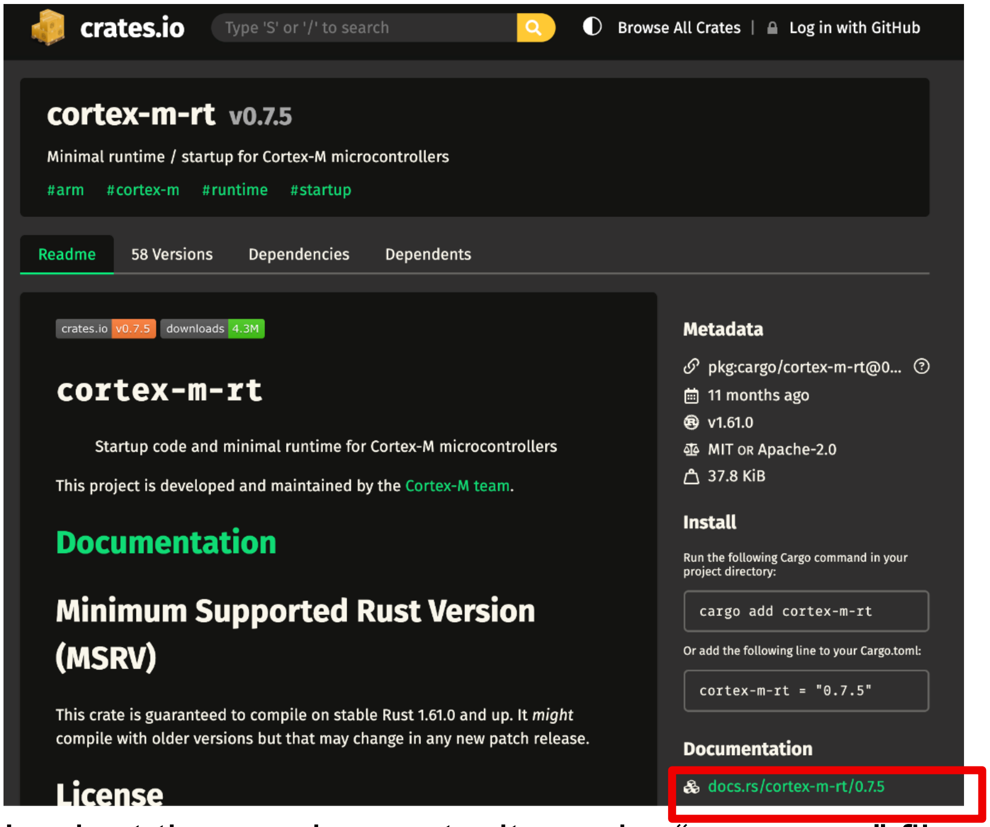
1.7.6) If you open the main.rs, you will see Rust assume we are going to code standard Rust. It will include a standard library, that's why println! works:

To change for embedded, type the following lines at the start:

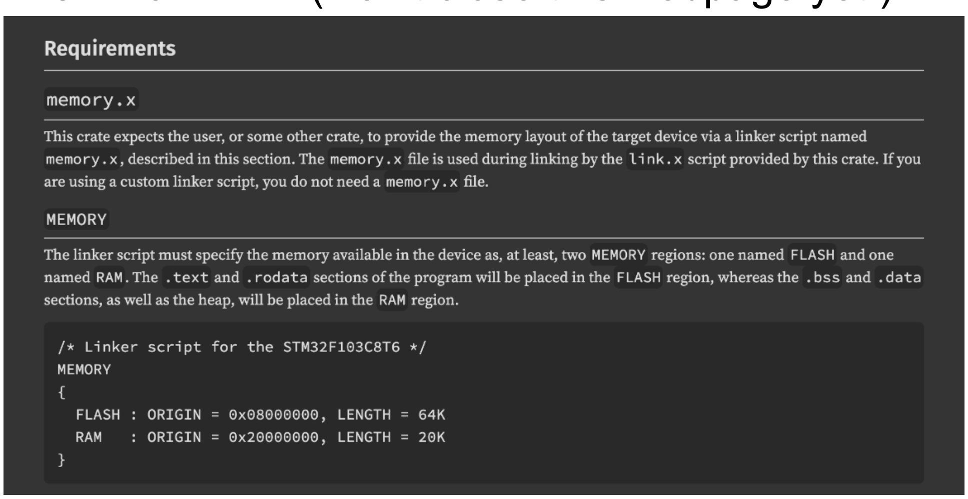


1.7.7) Now it's time to install stuff (add dependencies) that knows how to talk to the MCU. Go to crates.io

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



1.7.9) 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:

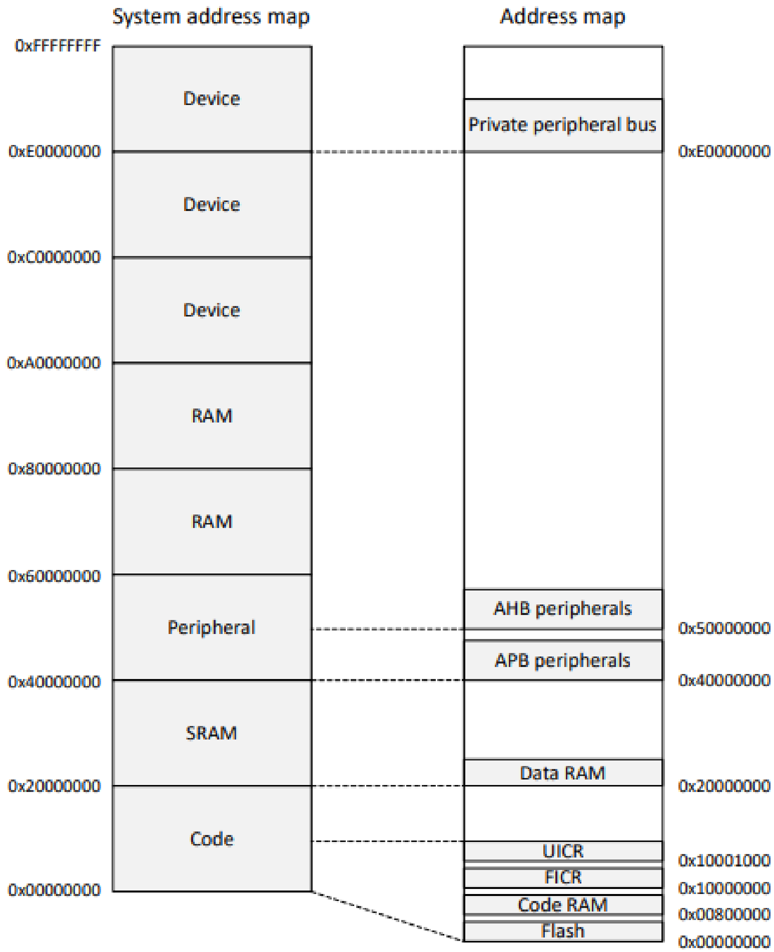


Figure 3: Memory map

Flash starts at: ___________

RAM starts at: ___________

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

```
xili@ML-PH-XL 18-xli-blinky % 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
    Updating crates.io index
     Locking 6 packages to latest Rust 1.90.0 compatible versions
      Adding cortex-m-rt v0.7.5
      Adding cortex-m-rt-macros v0.7.5
      Adding proc-macro2 v1.0.101
      Adding quote v1.0.40
      Adding syn v2.0.106
      Adding unicode-ident v1.0.19
```

It

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

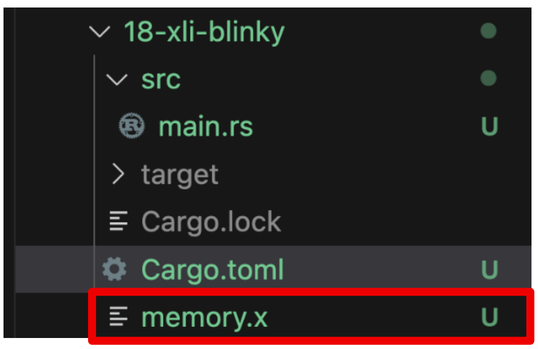
```
mdbook > src > 18-xli-blinky > Cargo.toml > {} package > name
[package]
name = "s18-xli-blinky"
version = "0.1.0"
edition = "2024"

[dependencies]
cortex-m-rt = "0.7.5"
```

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

```
xili@ML-PH-XL 18-xli-blinky % touch memory.x
xili@ML-PH-XL 18-xli-blinky %
```

Look, new file appears in the folder:



1.7.12) 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
/* Linker script for the Microbit */
MEMORY
{
  FLASH : ORIGIN = 0x00000000, LENGTH = 512K
  RAM   : ORIGIN = 0x20000000, LENGTH = 128K
}
```

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

main.rs U ●

mdbook > src > 18-xli-blinky > src > main.rs > main

```rust
#![no_std]
#![no_main]

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {}
}

```

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

The main will never return, and never ends.

1.7.14) Here's some housekeeping work.

i) Every time we run "cargo build", we want to target MCU. We can build a config file by

type "mkdir .cargo", then type "touch .cargo/config.toml", this will create a config file under the .cargo:

```
∨ 18-xli-blinky
  ∨ .cargo
      config.toml    U
```

Now edit the config.toml as follows:

main.rs U   config.toml U ●

mdbook > src > 18-xli-blinky > .cargo > config.toml > ...

```toml
[build]
target = "thumbv7em-none-eabihf"

[target.thumbv7em-none-eabihf]
rustflags = ["-C", "link-arg=-Tlink.x"]
```

The codes below can be copy pasted.

```toml
[build]
target = "thumbv7em-none-eabihf"

[target.thumbv7em-none-eabihf]
rustflags = ["-C", "link-arg=-Tlink.x"]
```

ii) Now we also need to check rust analyzer and tell it only to check for our bare metal target.

Type “mkdir .vscode”

Type “touch .scode/settings.json”

```
xili@ML-PH-XL 18-xli-blinky % mkdir .vscode
xili@ML-PH-XL 18-xli-blinky % touch .vscode/settings.json
xili@ML-PH-XL 18-xli-blinky %
```

Now we created another config file for rust analyzer:

```
18-xli-blinky
  .cargo
    config.toml        U
  .vscode
    settings.json      U
```

Edit it as follows:

```
main.rs U    {} settings.json U
mdbook > src > 18-xli-blinky > .vscode > {} settings.json > rust-analyzer.cargo.target
{
    "rust-analyzer.check.allTargets": false,
    "rust-analyzer.cargo.target": "thumbv7em-none-eabihf"
}
```

The codes below can be copy pasted.

```
{
  "rust-analyzer.check.allTargets": false,
  "rust-analyzer.cargo.target": "thumbv7em-none-eabihf"
}
```

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

```
xili@ML-PH-XL 18-xli-blinky % cargo check
   Compiling cortex-m-rt v0.7.5
    Checking s18-xli-blinky v0.1.0 (/Users/xili/microbit/discovery-mb2/mdbook/src/18-xli-blinky)
error: `#[panic_handler]` function required, but not found

error: could not compile `s18-xli-blinky` (bin "s18-xli-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.

Type “cargo add panic_halt”

```
xili@ML-PH-XL 18-xli-blinky % cargo add panic_halt
    Updating crates.io index
warning: translating `panic_halt` to `panic-halt`
      Adding panic-halt v1.0.0 to dependencies
    Updating crates.io index
     Locking 1 package to latest Rust 1.90.0 compatible version
      Adding panic-halt v1.0.0
xili@ML-PH-XL 18-xli-blinky %
```

Update the main.rs:

main.rs U ×

mdbook > src > 18-xli-blinky > src > main.rs > ...

```rust
#![no_std]
#![no_main]

use cortex_m_rt::entry;
use panic_halt as _;

#[entry]
fn main() -> ! {
    loop {}
}

```

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

```
xili@ML-PH-XL 18-xli-blinky % cargo check
    Checking panic-halt v1.0.0
    Checking s18-xli-blinky v0.1.0 (/Users/xili/microbit/discovery-mb2/mdbook/src/18-xli-blinky)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.04s
xili@ML-PH-XL 18-xli-blinky %
```

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

1.7.18) 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 ×

mdbook > src > s18-xli-blinky > src > main.rs > main

```rust
#![no_std]
#![no_main]

use core::ptr::write_volatile;
use core::arch::asm;
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 {
          unsafe { asm!("nop");}
        }

        is_on = !is_on;
    }

}
```

The codes below can be copy pasted:

```rust
#![no_std]
#![no_main]

use core::ptr::write_volatile;
use core::arch::asm;
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 {
      unsafe { asm!("nop");}
    }

    is_on = !is_on;
  }

}
```

Here're the comments on almost each line generated by Gemini AI.

These lines at the very top are attributes that configure the entire program.

Rust

```
#![no_std]
#![no_main]
```

- `#![no_std]`: This tells the Rust compiler **not to link the standard library (`std`)**. The standard library depends on features of an operating system (like files, networking, and memory allocation) that don't exist in a simple microcontroller environment. Instead, we use the `core` crate, which is a tiny subset of `std` that can run anywhere.
- `#![no_main]`: This attribute informs the compiler that we are **not using the standard `main` function entry point**. In a typical desktop program, the Rust runtime sets things up before calling `main`. Here, we define our own entry point using a special macro.

Rust

```rust
use core::ptr::write_volatile;
use core::arch::asm;
use cortex_m_rt::entry;
use panic_halt as _;
```

- `use core::ptr::write_volatile;`: Imports a special function for writing to a raw memory address. The `_volatile` part is crucial: it tells the compiler not to optimize this write away, ensuring that our instruction to the hardware is always executed.
- `use core::arch::asm;`: Imports the `asm!` macro, which allows us to write inline assembly code. It's used here to create a simple delay.
- `use cortex_m_rt::entry;`: Imports the `entry` macro from the `cortex-m-rt` (Cortex-M Runtime) crate. This macro sets up the interrupt vector table and defines the starting point of our code for Cortex-M microcontrollers like the one on the micro:bit.
- `use panic_halt as _;`: This line imports a "panic handler". If your program encounters an unrecoverable error (a "panic"), this code will simply halt the processor in an infinite loop. The `as _` syntax imports it just for its side-effects without giving it a name we'll use directly.

Rust

```rust
#[entry]
fn main() -> ! {
```

- `#[entry]`: This is the attribute we imported. It marks the `main` function as the true entry point of the program.
- `fn main() -> !`: This declares the main function. The return type `!` is called the "never type". It signifies that this function is not expected to ever return, which makes sense for an embedded program that should run forever in a loop.

Rust

```rust
    const GPIO0_PINCNF21_ROW1_ADDR: *mut u32 = 0x5000_0754 as *mut u32;
    const GPIO0_PINCNF28_COL1_ADDR: *mut u32 = 0x5000_0770 as *mut u32;
```

- **Variables**: These lines declare two constants that are raw, mutable pointers to 32-bit unsigned integers (`*mut u32`).
- **What they do**: They hold the specific memory addresses for the **configuration registers** of Pin 21 and Pin 28 on GPIO Port 0. By writing to these addresses, we can tell the microcontroller *how* to use these pins (e.g., as an input or an output). The addresses like `0x5000_0754` are found in the microcontroller's technical reference manual.

Rust

```rust
const DIR_OUTPUT_POS: u32 = 0;
const PINCNF_DRIVE_LED: u32 = 1 << DIR_OUTPUT_POS;
```

- **Variables**: `DIR_OUTPUT_POS` holds the bit position (`0`) within the configuration register that controls the pin's direction. `PINCNF_DRIVE_LED` creates the actual value we need to write.
- **What it does**: `1 << DIR_OUTPUT_POS` shifts the number `1` left by `0` bits, which just results in the value `1`. According to the datasheet, writing a `1` to bit 0 of a `PIN_CNF` register sets that pin to be an **output**.

Rust

```rust
unsafe {
    write_volatile(GPIO0_PINCNF21_ROW1_ADDR, PINCNF_DRIVE_LED);
    write_volatile(GPIO0_PINCNF28_COL1_ADDR, PINCNF_DRIVE_LED);
}
```

- `unsafe { ... }`: This block is required because we are performing an "unsafe" operation: writing directly to a memory address. Rust cannot guarantee that this is safe, so we must explicitly tell it we know what we're doing.
- `write_volatile(...)`: We call the function we imported earlier to write our configuration value (`1`) to the two pin configuration registers. After this code runs, pins P0.21 and P0.28 are both configured as outputs.

Part 2: The Infinite Blinking Loop

This section contains the logic that runs forever to turn the LED on and off.

Rust

```rust
const GPIO0_OUT_ADDR: *mut u32 = 0x5000_0504 as *mut u32;
const GPIO0_OUT_ROW3_POS: u32 = 21;
let mut is_on: bool = false;
```

- **Variables**:
  - `GPIO0_OUT_ADDR`: A constant pointer to the GPIO Port 0 `OUT` register. This register controls the voltage level (high or low) of all output pins on this port.
  - `GPIO0_OUT_ROW3_POS`: A constant holding the value `21`, which is the bit position for Pin 21 within the `OUT` register.
  - `let mut is_on: bool = false;`: A mutable boolean variable used to keep track of whether the LED should be on or off. We start with it off.

Rust

```rust
loop {
```

- `loop { ... }`: This starts an infinite loop. The code inside will run forever.

Rust

```rust
        unsafe {
            write_volatile (GPIO0_OUT_ADDR, (is_on as u32) << GPIO0_OUT_ROW3_POS);
        }
```

- **What it does**: This is the line that actually turns the pin on or off.
  1. `is_on as u32`: The boolean `is_on` is converted to an integer. `false` becomes `0`, and `true` becomes `1`.
  2. `<< GPIO0_OUT_ROW3_POS`: The `0` or `1` is bit-shifted 21 positions to the left. This creates a 32-bit number where only the 21st bit is set (e.g., `0...010...0`).
  3. `write_volatile(...)`: This 32-bit value is written to the `OUT` register. This sets Pin 21 high (3.3V) or low (0V) while setting all other pins on the port low. When Pin 21 is high and the other pin (P0.28) is low, the LED lights up.

Rust

```rust
        for _ in 0..400_000 {
           unsafe { asm!("nop");}
        }
```

- **What it does**: This is a simple, "busy-wait" delay.
- `for _ in 0..400_000`: This loop repeats 400,000 times.
- `asm!("nop")`: This executes the assembly instruction `nop` (No Operation). It's a command that tells the CPU to do nothing for one clock cycle. By doing this 400,000 times, we pause the program for a fraction of a second.

Rust

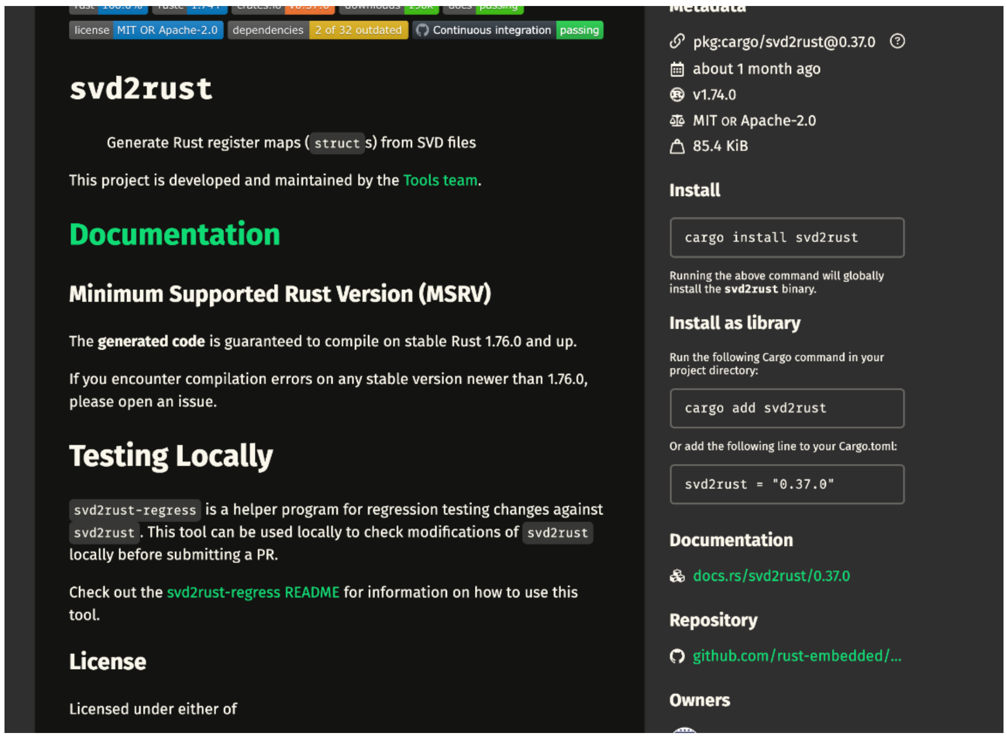
```rust
        is_on = !is_on;
    }
}
```

- `is_on = !is_on;`: At the end of each loop iteration, we flip the value of `is_on`. If it was `true`, it becomes `false`, and vice-versa.
- The program then jumps back to the top of the `loop` and repeats, but this time with the opposite value for `is_on`, causing the LED to change its state.

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:

2) **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.

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 {}
}
```

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?