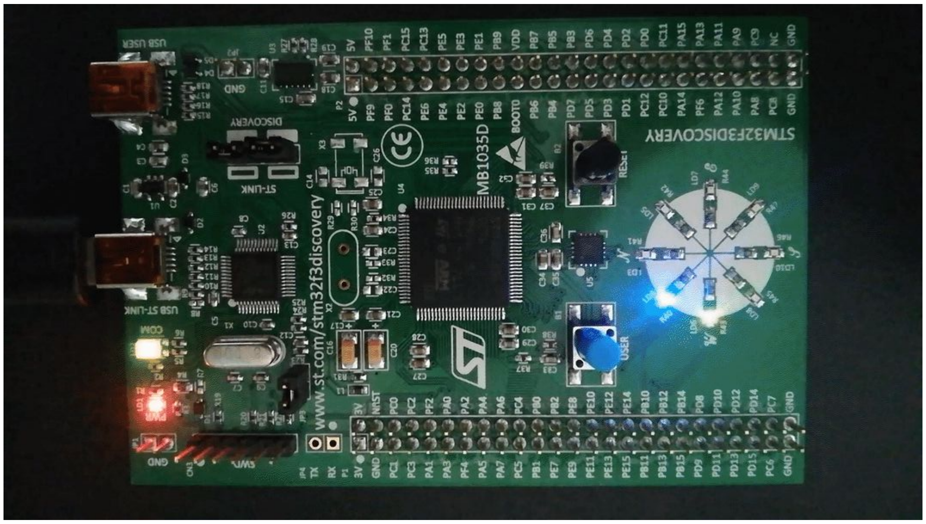
Chapter 5

LED Roulette

LED Roulette

We will implement the LED roulette program in this lecture.



Above is the result of the exercises we will be doing in this lecture.

LED Roulette

- To implement this roulette program we will be using a high level API.
- Main objective of this lecture is to get familiar with Building, Flashing and Debugging.
- We will be following code from this repository **https://github.com/PIAIC-IOT/Quarter2-Online.git**
- Programs we will be writing for microcontrollers are different from our standard programs in two ways.
  - #![no_std]
  - #![no_main]

LED Roulette

#![no_std] : no_std means this program won't use **std** crate.
#![no_main] : no_main means program won't use standard main interface, that's used for argument receiving command line apps.

Note : We named **entry point** "main" here but that could be anything. However signature of entry point function must be **fn() -> !**.

Building Program

Building Program

- After successfully writing code the very first thing is building your program.
- Since we're not building program for our computers but for a different architecture (i.e. MCU), therefore we have to cross compile.
- Cross compiling in Rust is as simple as passing an extra **--target** flag.
- The hectic part is to figuring out **the name of the target**.
- MCU in F3 has a Cortex-M4F processor and cargo knows how to cross compile to the Cortex-M architecture.

Building Program

- Cargo provides 4 different targets that cover the different processor families within that architecture.
  - **thumbv6m-none-eabi**, for the Cortex-M0 and Cortex-M1 processors
  - **thumbv7m-none-eabi**, for the Cortex-M3 processor
  - **thumbv7em-none-eabi**, for the Cortex-M4 and Cortex-M7 processors
  - **thumbv7em-none-eabihf**, for the Cortex-M4F and Cortex-M7F processors
- The one we are interested in is last one (i.e. thumbv7em-none-eabihf) because F3 has Cortex-M4F processor in it.

Building Program

- Before cross compiling we have to download pre-compiled version of the standard library. For that we have to run:

  $ rustup target add thumbv7em-none-eabihf

- We have to download it once after that it will be updated automatically whenever we update rust tool chain.
- All set, we can now compile our program for whatever target we are compiling, using following command:

  $ cargo build --target thumbv7em-none-eabihf

Flashing Program

Flashing Program

Flashing => Moving program to MCU's memory

Every time MCU will power on it will run the program flashed last time.

Steps:

1. Launch OpenOCD
2. Open GDB Server console
3. Connect GDB Server to OpenOCD
4. Load the program

Launching OpenOCD

We saw already what is OpenOCD, now let’s see how it works and how to initiate it.

First we see how to initiate:

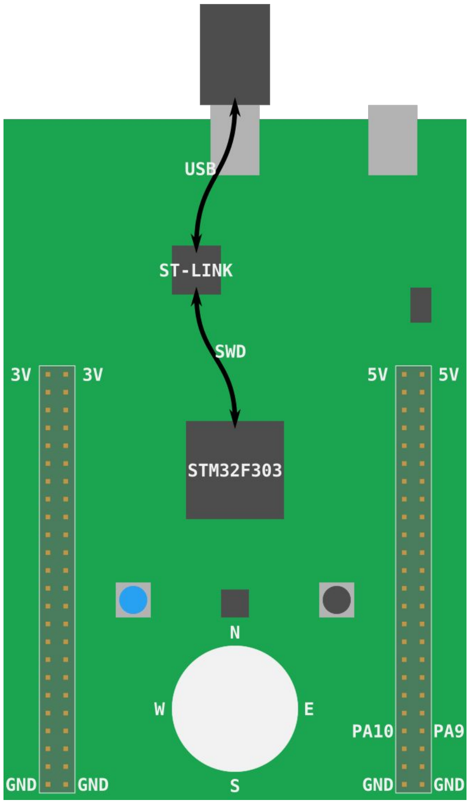
1. Open up a new terminal
2. Change directory to /temp (i.e. ***$ cd /tmp***)
3. Next run this command
   $ openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg

Launching OpenOCD

Now look at the command carefully at previous slide and the image at right.

You might observed that first we are interacting with **st-link** to target **stm32f303**.

ST-LINK opens up a communication channel for us to target.



Initiating GDB Server

These commands only specific to gdb console. They have no meaning to our normal terminal/command prompt.

1. ***$*** *<gdb>* *-q* *target/thumbv7em-none-eabihf/debug/<project-name>*

   <gdb> => System dependent debugging program to read arm binaries. It has 3 ariants **gdb**, **gdb-multiarch*** and **arm-none-eabi-gdb**
   *In our case gdb-multiarch will work.

   target/thumbv7em-none-eabihf/debug/<project-name> => Path to the binaries of projects we want to flash in MCU.

Connecting OpenOCD

Previous command only opens gdb shell. To connect to Openocd GDB Server run the following command.

2. ***(gdb) target remote :3333***

   This **3333** highlights the port number on which GDB server listens requests. And this command connects us to this port.

   ```
   Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
   Info : accepting 'gdb' connection on tcp/3333
   Info : device id = 0x10036422
   Info : flash size = 256kbytes
   ```

   Above is the result of this command.

Loading Program

Once we connected successfully, now final step to flash our code to MCU's persistent memory.

3. ***(gdb) load***

rajabraza@EliteBook-Folio-9470m: /tmp

File Edit View Search Terminal Help

```
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
adapter speed: 950 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800019c msp: 0x10002000
```

Debugging Program

Debugging Program

After **load** command our program stopped at **entry point**.

There are some helpful commands used for debugging purpose.

- break
- continue

break => break command is used to break program at certain point.
For example: if we want to stop program at the beginning of main function then we will run: ***(gdb) break main***

continue => this command will take us from one breakpoint to another breakpoint.

Debugging Program

For viewing line by line execution of program we can switch to GDB Text User Interface (TUI), for that run:
(gdb) layout src *(output on next slide)*

For disabling TUI mode we can run:
(gdb) tui disable

For resetting the program to initial state while debugging run:
(gdb) monitor reset halt

For terminating GDB session
(gdb) quit

Debugging Program

```
src/main.rs
    4
    5       use aux::{entry, prelude::*, Delay, Leds};
    6
B+  7       #[entry]
    8       fn main() -> ! {
    9           let  y;
    10          let x = 42;
    11           y = x;
    12
    13          // infinite loop; just so we don't leave this stack frame
>   14          loop {}
    15      }
    16      // fn main() -> ! {
    17      //     let (mut delay, mut leds): (Delay, Leds) = aux::init();
    18
    19      //     let half period = 500 u16;
remote Remote target In: chapter05 exercises::  cortex m rt main                L14   PC: 0x8000198
(gdb) step
chapter05 exercises::  cortex m rt main () at src/main.rs:10
(gdb) print  y
$1 = -536810104
(gdb) step
(gdb) print x
$2 = 42
(gdb) print &x
$3 = (i32 *) 0x10001ffc
(gdb) step
(gdb) print  y
$4 = 42
(gdb)
```

Debugging Program

Further commands:

- **step** : for moving to next line
- **print** : for printing values of variables
- **Info locals** : for printing all values of variables at once
- **Clear shell** : for clearing TUI screen

LED Program

LED Program

- So far we have learnt the basics of the embedded applications life-cycle.
- Now we will head to the application we started with desire of, LED Roulette.
- For this program we will use some abstrations, which are already written in the **library**.
- The library function will provide us **LEDs** and **Delays.**
- Library gives us two functions to use with LEDs; **on()** and **off()**
- Also it gives us a function for Delay; **delay_ms()**

Alias

Summary