Embedded Rust

Introduction

Following the excellent Rust Discovery tutorial that can be found on the net, I wanted to start my own embedded Rust project from scratch for the STM32F401RE. That way, I got more familiar with the STM32 rust crate and with cargo (the discovery tutorial uses a HAL). So, this is my summary of creating a simple project and getting it to run on the MCU. I scratched all of this stuff together from the discovery tutorial, reading code and the Rust documentation so to save myself some trouble in the future I though I better write it down. Maybe you, the reader, can use this stuff too. Enjoy!

Setup

First, some packages need to be installed. These are **arm-none-eabi-gdb** and **openocd**. Next Rust needs to be updated using **rustup**.

```
rustup target add thumbv7em-none-eabihf
```

Next, I created a new project with cargo **cargo new test_stm32f4**. That's all for the basic setup.

Dependencies

As I am using the stm32f4 crate and the cortex-m crates, these need to be added to the **Cargo.toml** of the project, below is the full file:

Cargo.toml

```
[package]
name = "test_stm32f4"
version = "0.1.0"
authors = ["m1etz"]
edition = "2018"

[dependencies]
panic-halt = "0.2.0"
cortex-m = "0.7.0"
cortex-m-rt = "0.6.13"
```

```
[dependencies.stm32f4]
version = "0.12.1"
features = ["stm32f401", "rt"]
```

All of these dependencies are needed, which I will show later in the actual code. Next, a file for the linker is needed to tell the linker about the MCU memory. This file, **memory.x** is shown in the box below. I copied this from github after a quick google search. If you do not have this file and you try to compile, you will be notified as it is necessary for the compilation to succeed.

memory.x

```
MEMORY
{
    /* NOTE K = KiBi = 1024 bytes */
    FLASH : ORIGIN = 0x08000000, LENGTH = 256K
    RAM : ORIGIN = 0x20000000, LENGTH = 40K
}   /* original file featured some comments below ... */
```

Coding and Building

So, finally, the code:

main.rs

```
#![no_main]
#![no_std]
// the peripheral driver API for the stm32f401re
use stm32f4::stm32f401;
// entry point for the application
use cortex_m_rt::entry;

// the panic handler, we strictly need this
#[allow(unused_extern_crates)]
extern crate panic_halt; // panic handler

fn delay() {
    for _i in 0..100000 { // set this to ~25000 for a faster blink
        // do nothing.
    }
}

#[entry]    // the entry point of the application
fn main() -> ! {

    // get peripherals
    let mut peripherals = stm32f401::Peripherals::take().unwrap();
```

```
    // take RCC and GPIOA, the peripherals we will work with
    let rcc = &peripherals.RCC;
    let gpioa = &peripherals.GPIOA;

    // clock gate
    rcc.ahb1enr.write(|w| w.gpioaen().set_bit());
    // pin to output
    gpioa.moder.modify(|_, w| w.moder5().output());

    loop {      // toggle
        // set the bit
        gpioa.odr.modify(|_, w| w.odr5().set_bit());
        delay();
        // clear the bit
        gpioa.odr.modify(|_, w| w.odr5().clear_bit());
        delay();
    } // main loop

} // fn main end
```

Nothing really special here. We activate the clock for GPIOA in RCC, then set the mode of GPIOA Pin 5 to output. After that we set the bit, wait and clear the bit, wait again. The classic Hello World for a microcontroller. Now, the two macros at the beginning, **no_std** prevents Rust from loading the standard library, which is not possible for bare metal applications, so we need to tell Rust that. The **no_main** indicates we are not using the standard rust main interface, according to some documentation. After a little more research, I found out that if this main interface is used in a no_std Rust application, the nightly build must be used. So in my case, it saved me the trouble to go for the nightly toolchain.

Build and Run

Here, another file can/needs to be created: **.cargo/config**, for setting cargo options. Here is the file:

```
[target.thumbv7em-none-eabihf]
runner = "arm-none-eabi-gdb -q"
rustflags = [
  "-C", "link-arg=-Tlink.x",
]
```

This sets the target and what happens when we do a **cargo run**. So now, for the actual part where everything is loaded to the MCU.

```
# build the project
cargo build --target thumbv7em-none-eabihf
# note: there is a configuration you can use so you don't need the
# "--target" didn't bother with that here though.
```

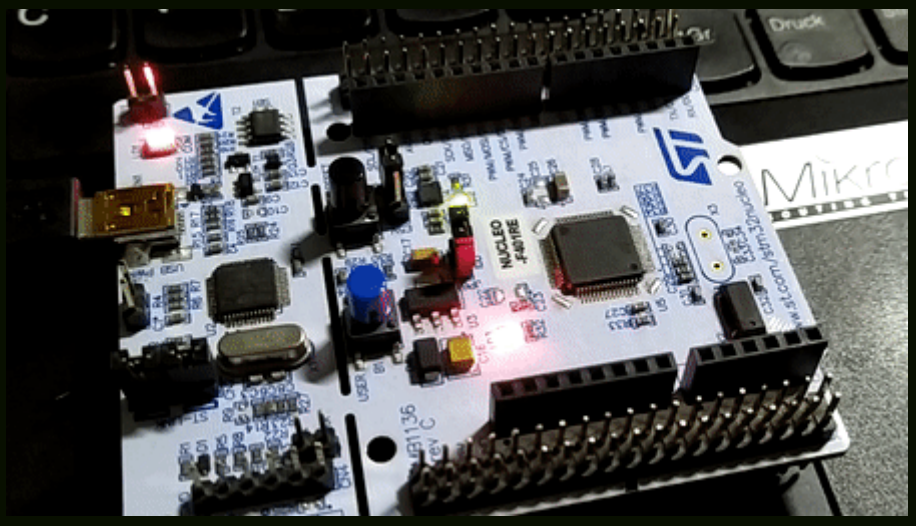
```
# start openocd
openocd \
    -f interface/stlink-v2-1.cfg \
    -f target/stm32f4x.cfg
```

```
# run the project
cargo run --target thumbv7em-none-eabihf
# note: there is a configuration you can use so you don't need the
# "--target" didn't bother with that here though.
```

At this point, there should be a GDB session open.

```
# connect to the target
target remote :3333
# load the program
load
# set a breakpoint in main
break main
# now run
continue
```

... and now the LED blinks (the little green one)! Mission accomplished!



But I'm not gonna stop at the Hello World program, that's just straight up boring. Let's continue with something that's at least a bit more interesting. Let's modify the .cargo/config file, so we can automate a couple of things. I scratched this together from some Stackoverflow posts:

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

```
[run]
target = "thumbv7em-none-eabihf"
[target.thumbv7em-none-eabihf]
runner = "arm-none-eabi-gdb -q"
rustflags = [
    "-C", "link-arg=-Tlink.x",
]
```

Now, a **cargo build** and **cargo run** don't need the --target option anymore! Cool, so let's measure some voltages with the ADC next:

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

use stm32f4::stm32f401;
use cortex_m_rt::entry;

#[allow(unused_extern_crates)]
extern crate panic_halt; // panic handler

fn delay() {
    for _i in 0..10000 {
        // do nothing.
    }
}

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

    // value measured by the ADC
    let mut adc_measure : u16;

    // get peripherals
    let peripherals = stm32f401::Peripherals::take().unwrap();

    // unwrapped peripherals we need.
    let rcc = &peripherals.RCC;
    let gpioa = &peripherals.GPIOA;
    let adc = &peripherals.ADC1;

    // clock gate, GPIOA and ADC1
    rcc.ahb1enr.write(|w| w.gpioaen().set_bit());
    rcc.apb2enr.write(|w| w.adc1en().set_bit());

    // pin to ADC analog in
    gpioa.moder.modify(|_, w| w.moder0().analog());
    // pin to output
    gpioa.moder.modify(|_, w| w.moder5().output());

    // adc on
    adc.cr2.modify(|_,w| w.adon().set_bit());
```

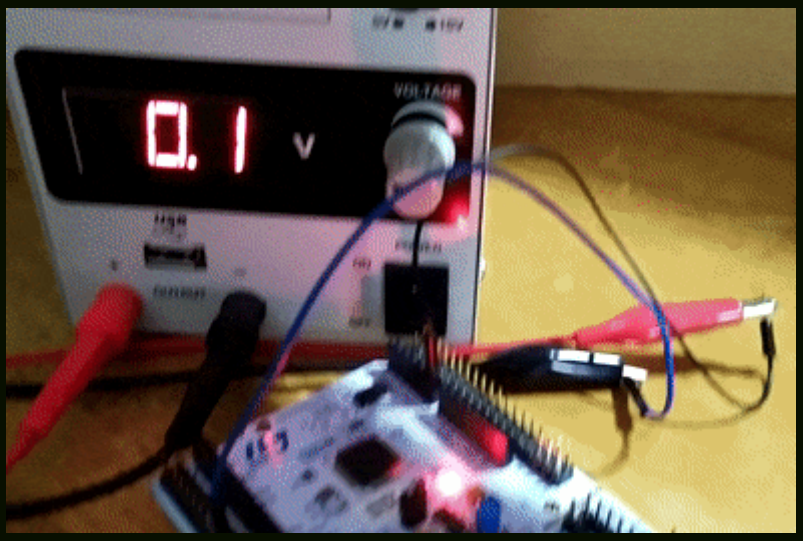
```
    loop {      // toggle

        // start a single conversion
        adc.cr2.modify(|_,w| w.swstart().start());
        // wait until end of conversion
        while adc.sr.read().eoc().is_not_complete() {}
        // read the measurement value when done, resets EOC
        adc_measure = adc.dr.read().data().bits();
        // set LED if measurement larger than MAX_MEASURE/2
        if adc_measure > 0x200 {
            gpioa.odr.modify(|_, w| w.odr5().set_bit());
        } else { // LED off otherwise
            gpioa.odr.modify(|_, w| w.odr5().clear_bit());
        }
        delay(); // wait a little bit before the next conversion

    } // main loop

} // fn main>
```

After a cargo build and cargo run, I connected to the target, loaded the program and ran it, as outlined below. Now that the program is loaded we can do some ADC measurements. Of course, this is just the simplest ADC setup one can think of, in practice you would probably want to put the ADC in continuous conversion mode and check it using a timer interrupt at a chosen interval, but for now, this shall suffice. So, what's happening in the code? Well, we take a measurement, wait until it is completed (End Of Conversion = EOC is set). After that, we check if the measurement exceeds a threshold, if yes, we turn on the LED, if not turn it off. The threshold voltage is about 0.5 volts, below you can see me using my dirt-cheap power supply to test the program. As you can see, it works fine. Yay!



Conclusion and Summary

Yeah, after a little reading and some playing around my project came together, which will now serve as a template for the future. Note that there is also a ARM Cortex-M quickstart project that can be found on github that is supposed to give you a template you can start with. However, to learn a little bit more and get to know some details, I thought it would be more fun to start from scratch. I will look into the quickstart project some other time, as more experienced "Rustaceans" created it. For what I wanted to achieve, this tutorial/template will suffice. As far as the interfaces in the stm32f4 crate go, I think they're great. Seems you can program an MCU in a readable, clean way with this crate and I'm looking forward to do more complex projects with it.

<= back