Are We Embedded Yet?

Implementing tiny HTTP server on a microcontroller

whoami

Maor Malka -

- Electronics Engineer and Maker
- Currently working as a Digital Design Engineer @ ARBE
- Doing Embedded Code , Hardware Design and Logic Design for the past 11 Years
- Trying to add Rust as a part of my toolbox

Agenda

- Microcontrollers quick glance
- The challenge with rust
- Use case
- Creating the Driver
- Creating the build environment
- Debugging
- Results/Demo
- Conclusions

What are microcontrollers?

Microcontrollers are compact, self-contained computing devices.

They consist of a processor memory,peripherals, and sometimes additional specialize hardware, all integrated onto a single chip.

Designed for real-time operations- Most microcontrollers run either a Bare metal environment running a basic task loop or an interrupt driven loop; or use an RTOS which will ensure consistent, deterministic behaviour

They are engineered for low power consumption, making them suitable for battery-operated and energy-efficient devices.

And yes, they are everywhere.

This one.

What types are there?

Microcontrollers come in many many variants, and is a constant battle between performance,power and price

Here are some example of the possible processor architectures you can find in microcontrollers used today:

8 Bit: AVR (classic arduino), PIC (microchip), 8051

16 Bit: MSP430(TI), dsPIC (microchip)

32 Bit: ARM(M series),RISC-V, MIPS

Memory is scarce, and can vary as low as 8KB of FLASH and 512 bytes of RAM

You can still buy one time programmable ones, as cheap as 0.03$ per piece!

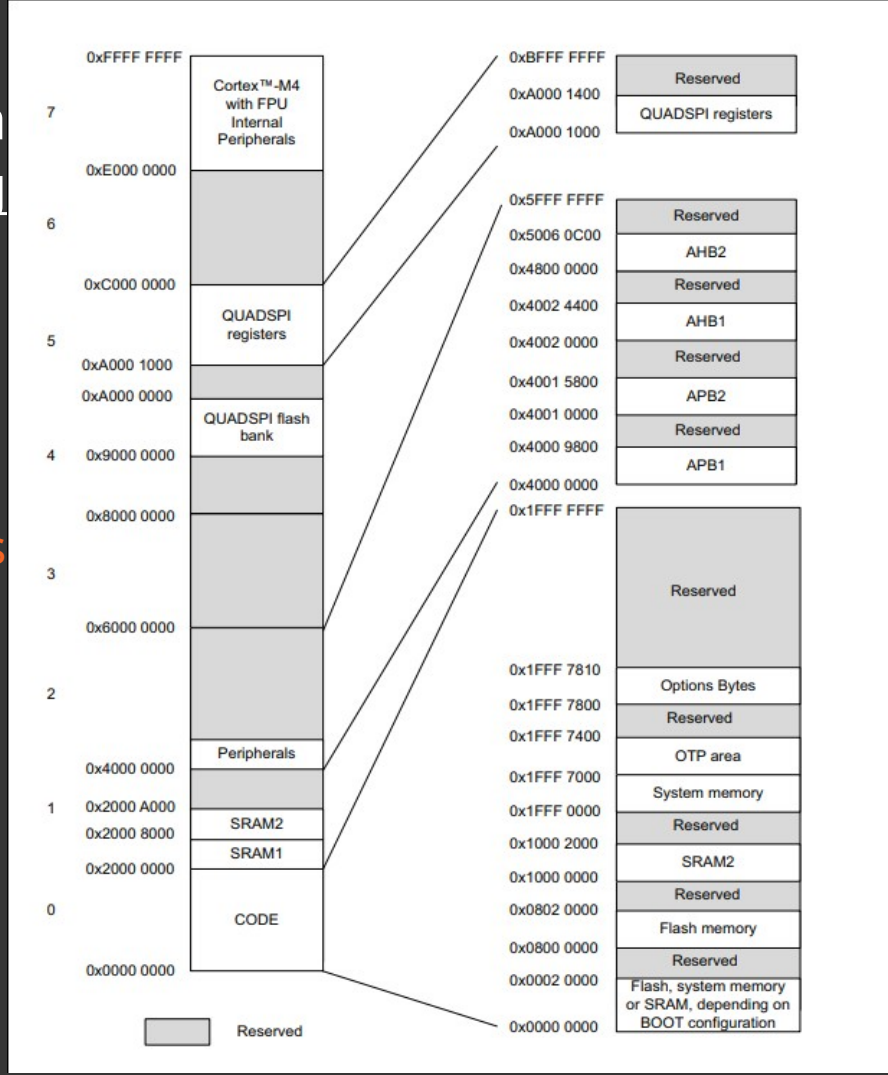
What About Rust?

Most microcontrollers implement a single address space for Everything

This includes Your Code, RAM, and Peripherals.

That means that to access the perierpals we read and write to certain address spaces

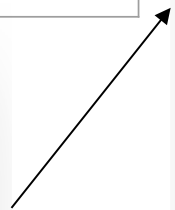
Why is that a problem?



Systick, located at 0xE000_E010

Offset	Name	Description	Width
0x00	**SYST_CSR**	**Control and Status Register**	**32 bits**
0x04	**SYST_RVR**	**Reload Value Register**	**32 bits**
0x08	**SYST_CVR**	**Current Value Register**	**32 bits**
0x0C	**SYST_CALIB**	**Calibration Value Register (Read only)**	**32 bits**

```rust
#[repr(C)]
struct SysTick {
    pub csr: u32,
    pub rvr: u32,
    pub cvr: u32,
    pub calib: u32,
}
let systick = unsafe { &mut *(0xE000_E010 as *mut SysTick) };
let time = unsafe { core::ptr::read_volatile(&mut systick.cvr) };
```



```rust
use volatile_register::{RW, RO};

pub struct SystemTimer {
    p: &'static mut RegisterBlock
}

#[repr(C)]
struct RegisterBlock {
    pub csr: RW<u32>,
    pub rvr: RW<u32>,
    pub cvr: RW<u32>,
    pub calib: RO<u32>,
}

impl SystemTimer {
    pub fn new() -> SystemTimer {
        SystemTimer {
            p: unsafe { &mut *(0xE000_E010 as *mut RegisterBlock) }
        }
    }

    pub fn get_time(&self) -> u32 {
        self.p.cvr.read()
    }

    pub fn set_reload(&mut self, reload_value: u32) {
        unsafe { self.p.rvr.write(reload_value) }
    }
}

pub fn example_usage() -> String {
    let mut st = SystemTimer::new();
    st.set_reload(0x00FF_FFFF);
    format!("Time is now 0x{:08x}", st.get_time())
}
```

BUT WHAT ABOUT THE BORROW CHECKER?

How to handle borrow checking on the peripherals?

- The User Can borrow access to the peripherals
- Any additional borrow will cause a panic.

Why not make peripherals a global mutable static variable?

```
struct Peripherals {
    serial: Option<SerialPort>,
}
impl Peripherals {
    fn take_serial(&mut self) -> SerialPort {
        let p = replace(&mut self.serial, None);
        p.unwrap()
    }
}
static mut PERIPHERALS: Peripherals = Peripherals {
    serial: Some(SerialPort),
};
```

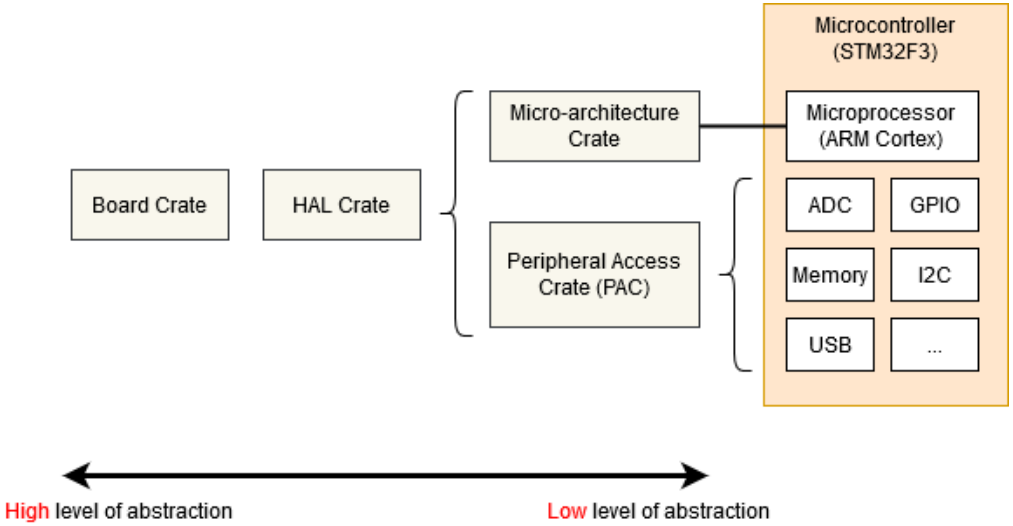
HAL 9000

The methodology of using all of these peripherals will require us to write a lot of code to access all these peripherals.

Luckly, most microcontroller designers also provide an SVD file, which describes all the peripherals, their addresses, and fields.

This SVD file can be converted to the Peripheral Access Crate, which creates the necessary structs and mapping to allow for proper access to the peripherals.

This however is not enough as you'll still need a lot of manual register writes to perform tasks on the peripherals, and thus, a Hardware Abstraction Library (HAL) is written to allow the user to perform basic operations on the peripherals.



Use case

I wanted try and create a somewhat useful project using embedded rust to show several aspects

- ➔ **Reuse**
  The ability to use cargo to save a lot of time making code blocks
- ➔ **Size**
  Even when using rust we can still fit in small sized microcontrollers
- ➔ **Safety**

  Leveraging rust to ensure no funny business is done unknowingly

The Task?

Implement a working “web server” on an STM32.

Why? To allow GUI usage without needing additional SW*

The Problem?

Our microcontroller (STM32l412):

- **Has no networking hardware.**
- **Has no file system support.**
- **Has only 128 KB of Flash and 40 KB of RAM.**

#![no_std]

- Given the Space limitations, it is very common to use #![no_std] in embedded rust projects.
- This implies that we need to additional steps to get basic functions of rust working (such as Vec,Panics)
- This also implies that any crate we choose to use has to support running in #![no_std] mode.

So what are we doing?

The plan is to use the USB interface of the STM32 microcontroller to create a compliant CDC-NCM device, AKA- USB-Ethernet Dongle

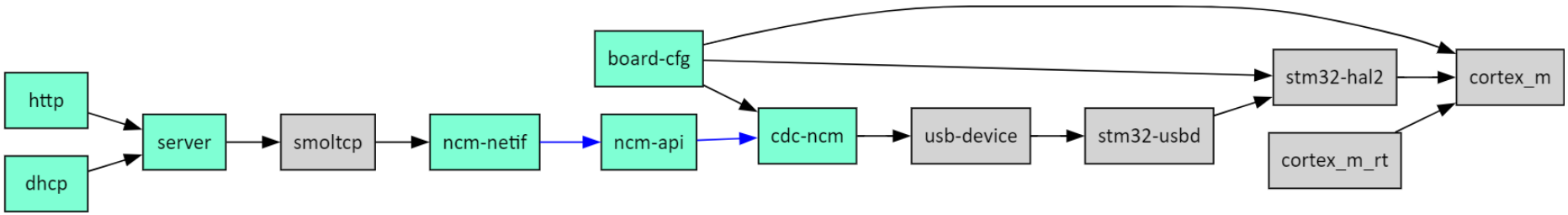
Next, we will need to get a TCP/IP stack working on the board and create a network interface adapter that will support our USB interface

Finally, We will create a server with two sockets:

- A HTTP socket which will serve the website and handle GET/POST requests
- A DHCP socket which will give the computer which the device is connected to an IP.

Kind of like this

Gray: Crates
Lightblue: Code we need to write



Creating the driver

- CDC-NCM Driver
  - Writing an CDC-NCM descriptors as per the USB spec supported by usb-device crate
  - Building an API handler to parse the incoming USB traffic and convert it to ethernet packets and vise-versa
  - As packets arrive in small chunks (64 bytes), we need to buffer and slowly parse the messages, this was done using a concurrent queue
- Smoltcp netif
  - Smoltcp expects the crate user to create a network interface and connect it by providing tx/rx tokens it will consume
  - This was implemented by creating a concurrent queue between the NCM-API handler and the NETIF

The Bare Metal build environment

- Cross-compiler toolchain
- Runtime
- Memory.x (Linker Script)
- Build.rs (to force rustc to use the linkers script)
- Our Main.rs which contains:
  - Entry point for our code
  - A Panic Handler
  - Global Allocator
  - Hardware Exception Handlers

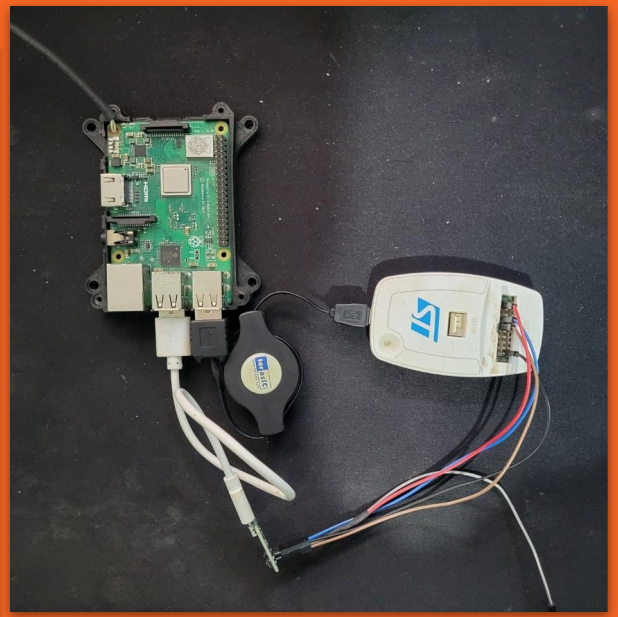
Debugging

Debugging is super easy; If you have the right tools.

As rust supports GDB, we can easily setup a GDB server using openOCD which interfaces to the board via a jtag debugger

As the microcontroller uses a modern ARM core, we also have access to the RTT (real-time trace)

That coupled with *ferrous-systems* **defmt** allows us to get logs and even panics printed out to our console!



```
maorm@LAPTOP-O524TOIN:~/projects/stamrustwkspace/stamrust$ nc 192.168.1.44 8765 | defmt-print -e target/thumbv7em-none-eabihf/debug/stamrust
0 INFO  starting server...
└─ stamrust::__cortex_m_rt_main @ src/main.rs:203
1000 DEBUG seconds:1 loops: 48881
└─ stamrust::finalize_perfcounter @ src/main.rs:236
2000 DEBUG seconds:2 loops: 48895
└─ stamrust::finalize_perfcounter @ src/main.rs:236
3000 DEBUG seconds:3 loops: 48894
└─ stamrust::finalize_perfcounter @ src/main.rs:236
4000 DEBUG seconds:4 loops: 48895
└─ stamrust::finalize_perfcounter @ src/main.rs:236
5000 DEBUG seconds:5 loops: 48894
└─ stamrust::finalize_perfcounter @ src/main.rs:236
6000 DEBUG seconds:6 loops: 48895
└─ stamrust::finalize_perfcounter @ src/main.rs:236
7000 DEBUG seconds:7 loops: 48895
└─ stamrust::finalize_perfcounter @ src/main.rs:236
```

Safety?

- Global variables
  - As common as sand in embedded projects
  - Being forced to use a mutex
- Concurrency
  - Can’t do it otherwise
  - Interrupt driven ready
- Compilation/Panic gotchas
  - overflow/underflow
  - Invalid api sizes

Demo Time!

Conclusions

Are the goals covered?
Yes! we have a working project, which is fast, light and safe!
However... are we embedded?

Professional safe environments
Automotive - ISO26262, ferrous systems has created a certified rustc which can be used on automotive systems
Aerospace -DO178 has not yet been adopted
Medical - IEC 60601 has not yet been adopted
Military?

Embedded engineers are a tough nut.
Experience and insanity will make it very difficult to shift embedded software engineers to attempt new languages other than C (or C++ on embedded linux).

The customer-vendor problem
As long as customers do not request usage of rust, the support and toolchains will be stuck in open source support

RTOS in rust
- Embassy
- RTIC
- lilos

RTOS in C
- FreeRTOS (21 Years)
- AzureRTOS (27 Years)
- uC/OS (33 Years)

Thank you!

questions?