Embedded Rust

Building a logic analyzer from scratch

Overview

- Why use embedded rust?
- Ecosystem overview
- Getting started building a embedded rust application
- Use embedded HAL
- Some demos
- Short Q / A session

Why use Rust for embedded?

- Usually the domain of the C programming language
  - There are other alternatives like C++
- Portability
  - Can work on very constrained devices
  - Targets a wide range of embedded platforms (cortex-m, RISC-V, AVR, ...)
  - Currently cortex-m and RISC-V have the best support
  - Rust is able to target anything llvm
- Low profile / no runtime
  - Dynamic memory allocation is optional
  - No garbage collection
  - Predictable performance
  - Small code size

Why use Rust for embedded?

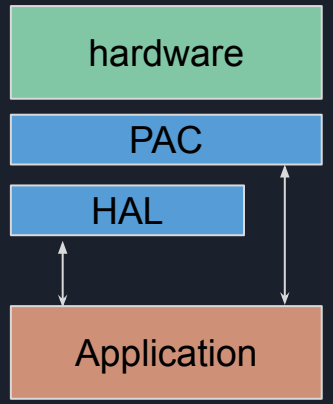
- Interoperability
  - Excellent C FFI, no overhead
  - Talk to existing vendor libraries / drivers
- Higher level abstractions (compared to C)
  - Advancements in language design in the last 50 years (Traits, ...)
  - The borrow checker
  - Safely encapsulate unsafe peripheral access
- Excellent Tooling
  - Build with cargo (not strictly mandatory)
  - Generate whole project documentation with rustdoc
  - Shout out to rust-analyzer (https://rust-analyzer.github.io/)
  - Compiler error messages

Why use Rust for embedded?

- Ecosystem
  - Support for many microcontrollers
  - New drivers almost weekly

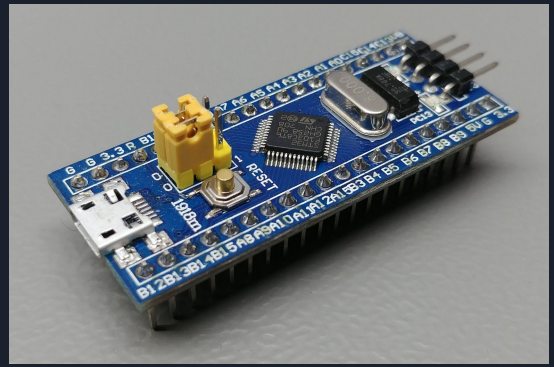
Rust embedded ecosystem

- Lots of Crates / libraries, different categories
- no_std crates
  - Libraries compatible with no_std rust
  - e.g. serde, bitflags, ...
- Peripheral access Crates (PAC)
  - Slim abstractions over hardware
  - Usually automatically generated by a tool (like svd2rust)
- Embedded HAL Crates
  - Higher level abstractions over hardware
  - User interfaces defined in embedded-hal Crate (https://github.com/rust-embedded/embedded-hal)
  - Provides hardware independent interface (digital IO, timers, USB, analog IO, ...)



Let’s get started

- Our target: “stm32 bluepill” development board
  - Dirt cheap, almost 4$ a piece
  - I already have a full box of them
- Features
  - Plenty of general purpose IO (GPIO)
  - USB controller & connector
  - Enough timers
- Would not recommend
  - Missing programmer / debugger
  - USB connector is really bad
  - Wrong resistor on USB-DP pin
  - For beginners, look at the STM32F3DISCOVERY board



Let’s get started

- Toolchain
  - Rust toolchain -> install from https://rustup.rs/
  - Rust target for cortex-m3 -> *rustup target add thumbv7m-none-eabi*
  - Gdb for embedded arm -> Package called *arm-none-eabi-gdb*
  - Done?!
- Project setup
  - Copy https://github.com/rust-embedded/cortex-m-quickstart
  - Modify .cargo/config, in our case already correct
  - Modify memory.x (configure flash and RAM size)
    - `FLASH : ORIGIN = 0x08000000, LENGTH = 128K`
    - `RAM : ORIGIN = 0x20000000, LENGTH = 20K`

main.rs

- Exclude rust standard library

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

  - No system calls / operating system APIs
  - No dynamic memory allocation

- Implement panic

```
use panic_halt as _;
```

  - Part of minimal “runtime”
  - Implements the panic handler

main.rs

- Entry point

```
use cortex_m_rt::entry;
#[entry]
fn main() -> ! {
      asm::nop();
      loop {
      // your code goes here
      }
}
```

  - Procedural macro, generates necessary pre init
  - Exception handlers
  - Reset memory (zero .bss section)
  - Calls main()

Demo time!

USB communication

- Add HAL for stm32f1xx devices to dependencies
  - Found at https://github.com/rust-embedded/awesome-embedded-rust
  - embedded-hal -> Interface for generic hardware interaction
  - usb-device -> Extends embedded-hal with usb interface
  - stm32f1xx-hal -> Implements hal & usb
- Import hal in main.rs
  - Otherwise it will not be linked -> compilation fails
  - Using anything from stm32f1xx_hal should be enough

USB communication

- Acquire access to peripherals

```
use stm32f1xx_hal::pac;
fn main() -> ! {
      let peripherals = pac::Peripherals::take().unwrap();
      let core_peripherals = pac::CorePeripherals::take().unwrap();
      let mut rcc = peripherals.RCC.constrain();
      let mut flash = peripherals.FLASH.constrain();
…
```

  - PAC: Peripheral access Crate
    - Raw access to peripherals
  - HAL: Hardware abstraction layer
    - Safe(ish) access to generic peripherals

USB communication

- Clocks, clocks everywhere!

```
let clock_cfg = rcc.cfgr
      .use_hse(8.mhz())
      .sysclk(48.mhz())
      .pclk1(24.mhz())
      .freeze(&mut flash.acr);
assert!(clock_cfg.usbclk_valid());
```

  - Usually generated by some vendor (GUI) tool
  - Rust: High level builder pattern
  - compiles down to a few loads & stores to registers
  - Allows us to check if it works with required peripherals!

USB communication

- USB reset

```
let mut gpioa = peripherals.GPIOA.split(&mut rcc.apb2);
let mut dp = gpioa.pa12.into_push_pull_output(&mut gpioa.crh);
let dm = gpioa.pa11;
dp.set_low().unwrap();
delay(clock_cfg.sysclk().0 / 100);
let dp = dp.into_floating_input(&mut gpioa.crh);
```

  - Acquire pin a12, convert to HAL output pin
  - Set low
  - Delay execution
  - Set to floating again

USB communication: set up usb device

- Static variables for access in interrupt

```
static mut USB_BUS: Option<UsbBusAllocator<usb::UsbBusType>> = None;
static mut USB_DEV: Option<UsbDevice<usb::UsbBusType>> = None;
```

  - Usb stack runs in USB interrupt context

- Set up generic USB peripheral

```
let usb = usb::Peripheral { usb: peripherals.USB, pin_dm: dm, pin_dp: dp};
let bus = usb::UsbBus::new(usb);
let usb_dev = UsbDeviceBuilder::new(USB_BUS.as_ref().unwrap(),
UsbVidPid(0xdead,
0xbeef)).manufacturer("ruabmbua").product("rlogic").device_class(0x03).build();
```

USB communication: set up usb device

- Enable interrupts

```
unsafe {
      NVIC::unmask(Interrupt::USB_HP_CAN_TX);
      NVIC::unmask(Interrupt::USB_LP_CAN_RX0);
}
```

- Keep cpu idle

```
loop {
      asm::wfi();
}
```

USB communication: set up usb device

- Run USB stack in interrupt

```
fn usb_interrupt() {
      let usb_dev = unsafe { USB_DEV.as_mut().unwrap() };
      if !usb_dev.poll(&mut []) {
            return;
      }
}
#[interrupt]
fn USB_HP_CAN_TX() { usb_interrupt();}
#[interrupt]
fn USB_LP_CAN_RX0() { usb_interrupt();}
```

  - Interrupt procedural macro provided by pac::interrupt
  - Automatically adds interrupt vector

Demo time!

USB communication: protocol

- Commands sent to device
  - Start, frequency parameter
  - Stop
  - Use USB interrupt transfers
- Device to PC channel
  - Potentially a lot of data
  - Use 64 byte USB packets
  - Pack logic levels into bits
  - Use USB bulk transfers

USB communication: protocol

- Use tagged enum for commands

```
pub enum Command { Stop, Start(Hertz) }
```

- Encode command opcode as byte

```
#[repr(u8)]
enum CommandOp {Stop = 0x00, Start, _End,}
```

USB communication: protocol

- Decode opcode

```
pub fn parse(bytes: &[u8]) -> Option<Command> {
    if bytes[0] >= CommandOp::_End as u8 {return None;}
    let op: CommandOp = unsafe { core::mem::transmute(bytes[0]) };
```

  - Check if opcode < max opcode
  - Transmute byte to CommandOp enum

USB communication: protocol

- Extract arguments, pack into Command

```
match op {
      CommandOp::Stop => Some(Command::Stop),
      CommandOp::Start => {
            let val = bytes[1..5].try_into().unwrap();
            Some(Command::Start(u32::from_le_bytes(val).hz()))
      }
      _ => unreachable!(),
}
```

  - Match on opcode
  - In case of Start, read byte 1 to 4 as little endian

Capture timer

- Need to capture input pins in specific interval
- One of the following modes:
  - Running (at frequency)
  - Not running
  - Uninitialized

```
enum CaptureTimer {
   Uninit,
   Enabled(CountDownTimer<TIM2>),
   Disabled(TIM2),
}
```

Capture timer

- Start command

```
fn start(&mut self, freq: Hertz, clocks: &Clocks, apb1: &mut APB1) {
      let mut other = CaptureTimer::Uninit;
      mem::swap(self, &mut other);
      if let CaptureTimer::Disabled(tim) = other {
            let mut timer = Timer::tim2(tim, clocks, apb1)
                  .start_count_down(freq);
            timer.listen(Event::Update);
            *self = CaptureTimer::Enabled(timer);
      } else {
            mem::swap(self, &mut other);
      }
}
```

  - Borrow checker problem
  - Can not move members out of self
  - Apply mem::swap() trick

Capture timer

- Stop command

```
fn stop(&mut self) {
      let mut other = CaptureTimer::Uninit;
      mem::swap(self, &mut other);
      if let CaptureTimer::Enabled(mut count_down) = other {
            count_down.unlisten(Event::Update);
            *self = CaptureTimer::Disabled(count_down.release());
      } else {
            mem::swap(self, &mut other);
      }
}
```

Demo time

Code is at https://github.com/ruabmbua/rlogic

https://www.rust-lang.org/what/embedded

https://github.com/rust-embedded/awesome-embedded-rust

https://rust-embedded.github.io/book/

https://docs.rs/embedded-hal/0.2.4/embedded_hal/index.html