An introduction to “bare-metal” embedded systems programming using Rust

Pramode C.E

February 14, 2018

Why Rust?

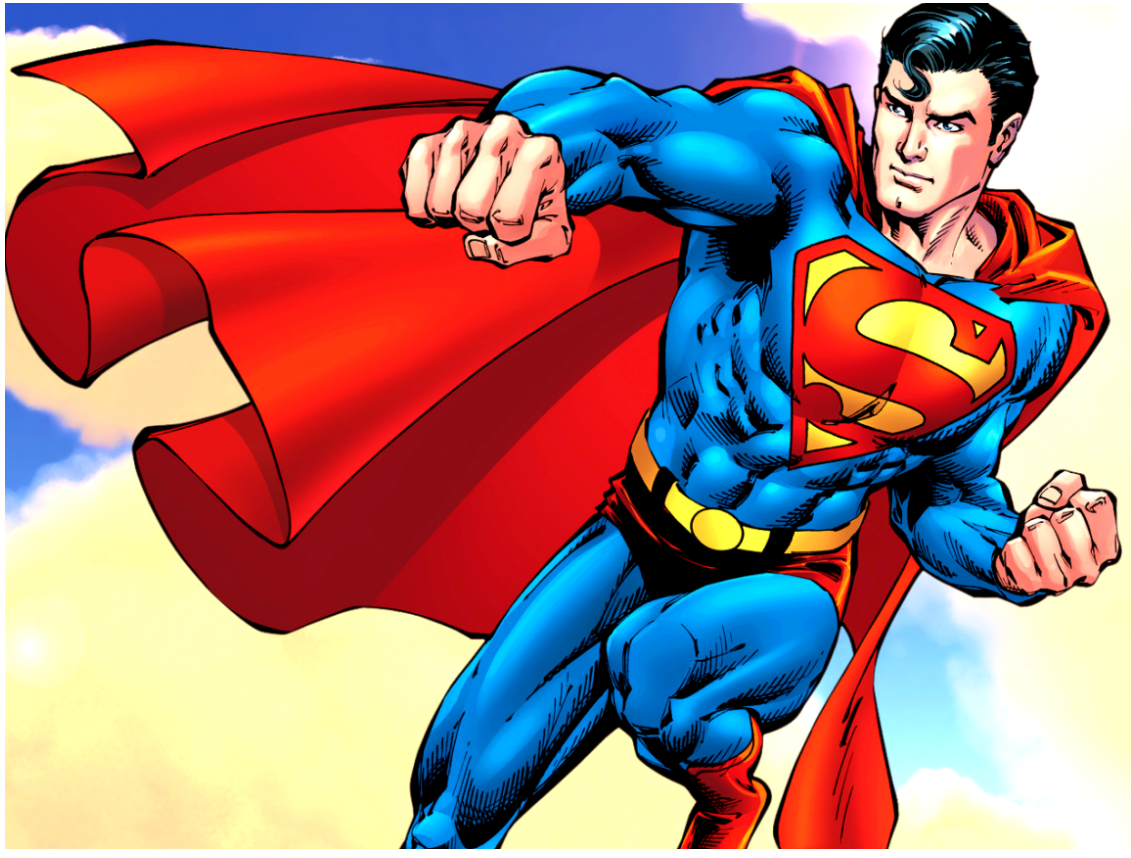

Figure 1: Superpowers!

- Ownership / Move semantics makes it possible to write peripheral access API's which can't be used incorrectly.
- The trait system together with crates.io opens up the possibility of writing highly re-usable code.

- Write concurrent (interrupt-driven) programs without worrying about data races.
- Use productive, high-level abstractions like sum types, pattern matching, iterators, closures, generics etc even on very low end systems.

A quick review of Rust concepts relevant to this talk

```rust
// Ownership and Move Semantics
fn main() {
    let v1 = vec![1,2,3];
    let v2 = v1;
    println!("{:?}", v1);
}
```

```rust
// Ownership and Move Semantics
fn foo(v: Vec<i32>) -> i32 {
    v[0]
}

fn main() {
    let v = vec![1,2,3];
    let _t = foo(v); // vector gets moved
    println!("{:?}", v);
}
```

```rust
// Closures
fn main() {
    let f = |x| x * x ;

    println!("{}", f(10));
}
```

Zero cost abstractions

```rust
// Compile: rustc -O add.rs
const N: u64 = 1000000000;
fn main() {
    let r = (0..N).fold(0, |sum, i| sum+i);
    println!("{}", r);
}
```

Unsafe

```
fn main() {
    let p1 = 0 as *mut i32;

    *p1 = 100;

    // similar to writing, in C:
    // int *p; p = 0;
    // *p = 100;
}
```

Unsafe

```
fn main() {

    unsafe {
        let p1 = 0 as *mut i32;

        *p1 = 100;
        // no compile time error.
        // But program crashes at run time!
    }

}
```

Rust for “bare-metal” embedded systems programming

- Rust uses LLVM for code generation.
- Excellent support for ARM microcontrollers.
- AVR, MSP430, RISC-V support in the growing phase.

Major directions

- Jorge Aparicio (http://blog.japaric.io/) is working on some very interesting abstractions for embedded development:
  - An I/O framework and a hardware abstraction layer. (http://blog.japaric.io/brave-new-io/)
  - A framework for Real Time systems development called RTFM (http://blog.japaric.io/tags/rtfm/)

Major directions

- The TockOS project (https://www.tockos.org/) is developing an embedded operating system for low-memory, low-power applications. The OS kernel is written in Rust.

Our focus

- Jorge Aparicio's work
- ARM Cortex-M micrcocontrollers

Demo - Self balancing Robot!

Figure 2: Self-balancing Robot, coded in Rust by Jorge Aparicio!

- ARM Cortex-M3 (no FPU) @ 8 Mhz
- 4Kb RAM, 100 bytes flash, no heap
- Gyro, Accelerometer, Kalman filter, PID controller, Logging (bluetooth) @ 500Hz

Demo - Madgwick's orientation filter

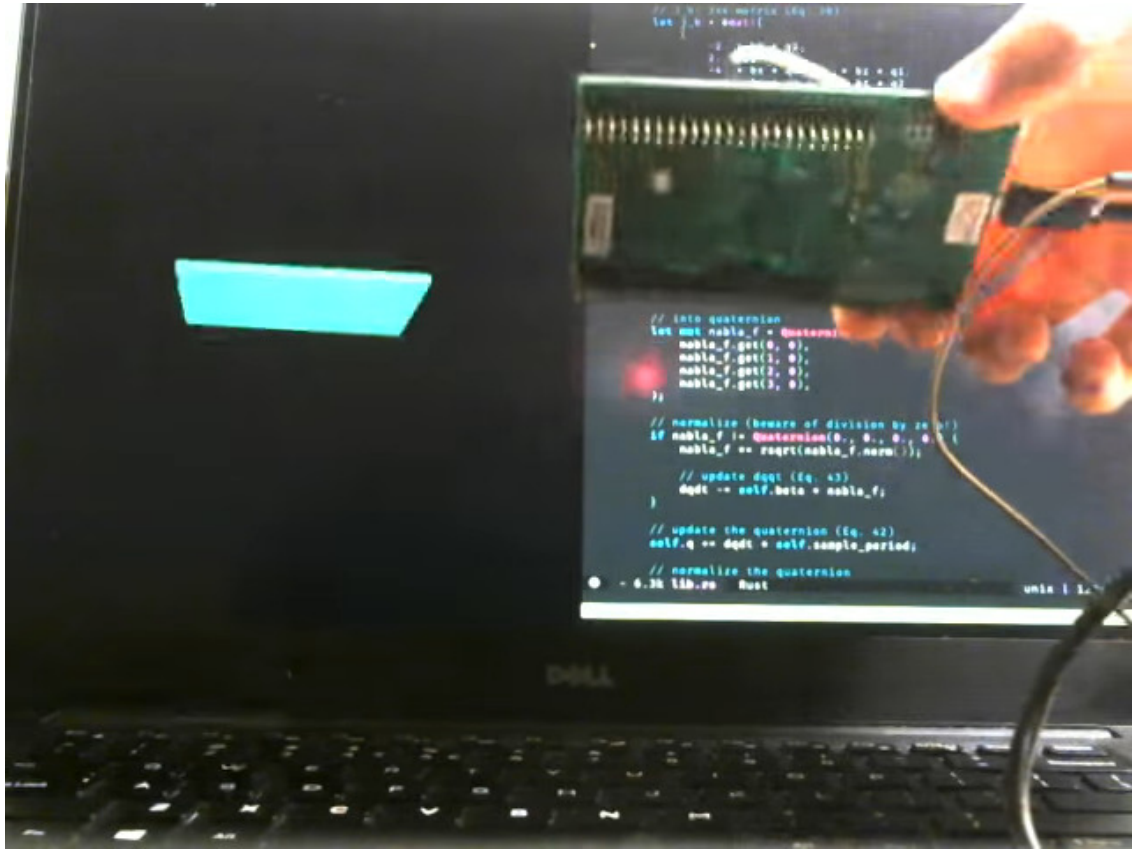

Figure 3: Madgwick's orientation filter by Jorge Aparicio!

Demo - Ethernet driver and CoAP server

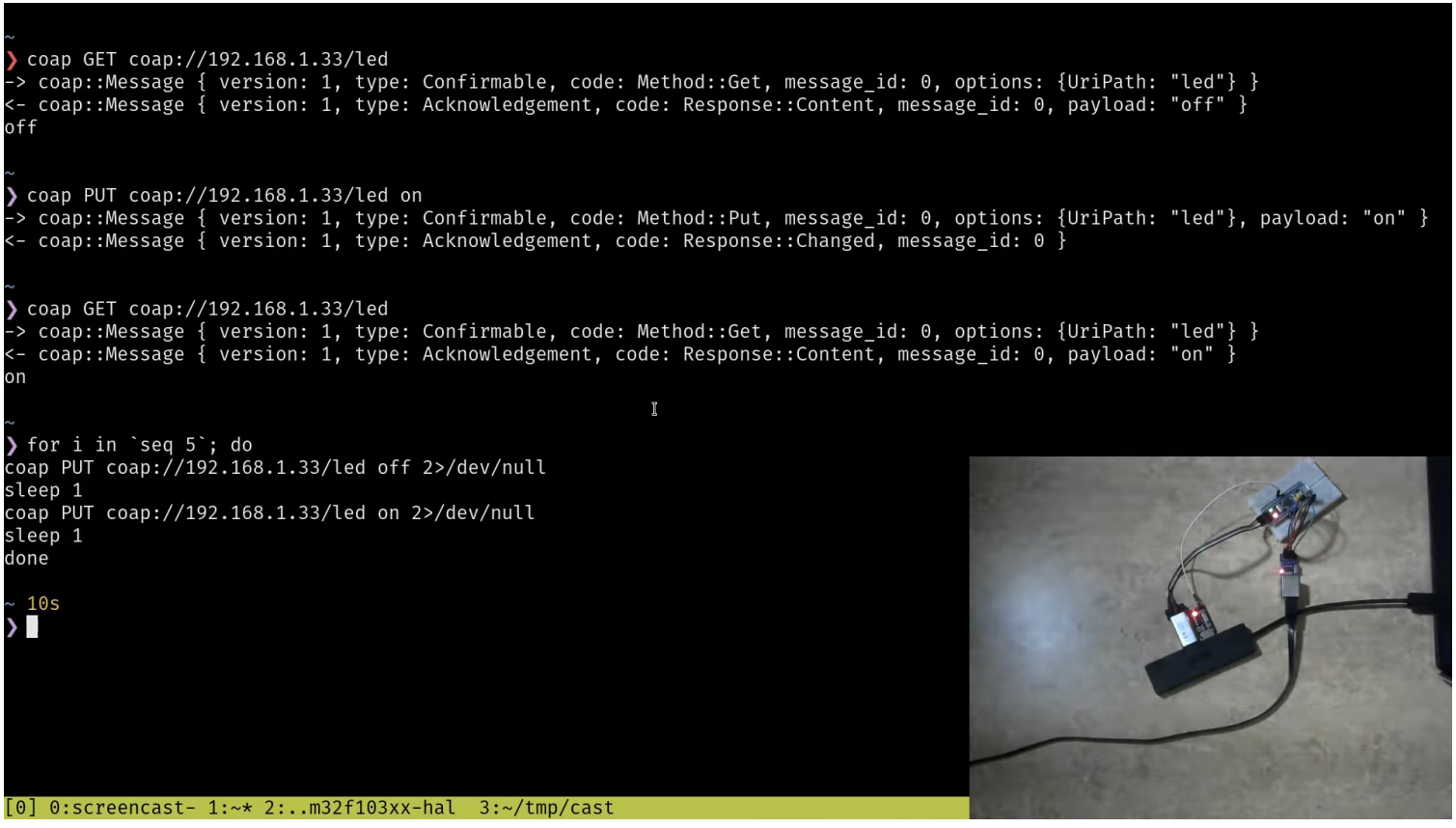

Figure 4: Ethernet driver by Jorge Aparicio!

The four levels of abstraction when programming microcontrollers in Rust (using Jorge's framework)

- Direct register programming using raw pointers.
- Using functions autogenerated by svd2rust (https://github.com/japaric/svd2rust).
- Using functions provided by a Hardware Abstraction Layer.
- Using functions provided by a Board Support crate.

Nightly Rust

- Bare-metal embedded systems programming requires “Nightly Rust”; things will break unexpectedly. (http://railwayelectronics.blogspot.in/2018/01/i-recently-picked-up-embedded-project.html)
- The standard library is not available when doing bare-metal embedded programming.

Installing the toolchain

- GNU Binutils (for the linker)
- Nightly Rust
- rustup / xargo

- st-flash or openocd
- Detailed instructions: https://japaric.github.io/discovery/

Some platforms

- The STM32F3DISCOVERY board which uses an ARM Cortex-M4F processor
- The “blue pill” board using STM32F103 processor
- The TI Stellaris/Tiva Launchpads using ARM Cortex-M4F processors

The STM32F3DISCOVERY

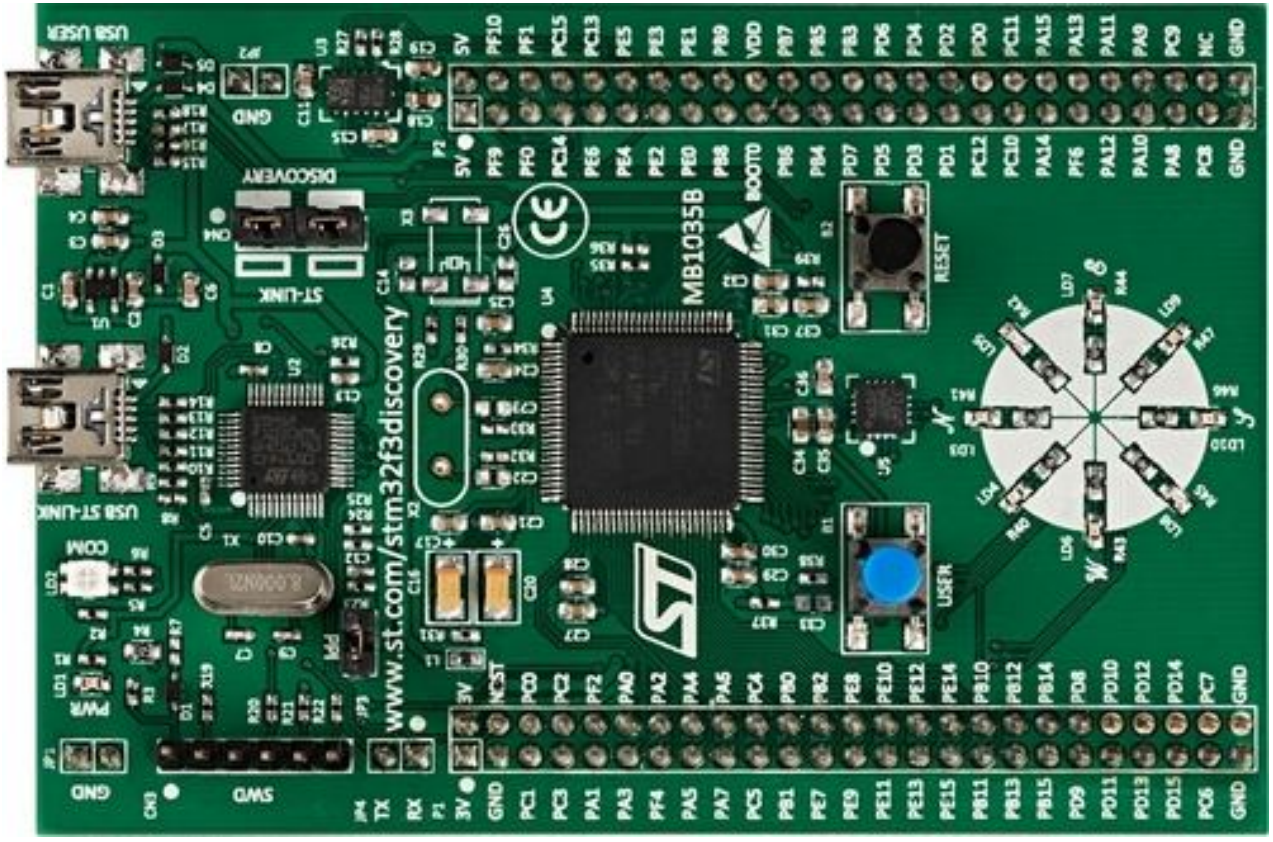

Figure 5: STM32F3DISCOVERY

The blue pill board

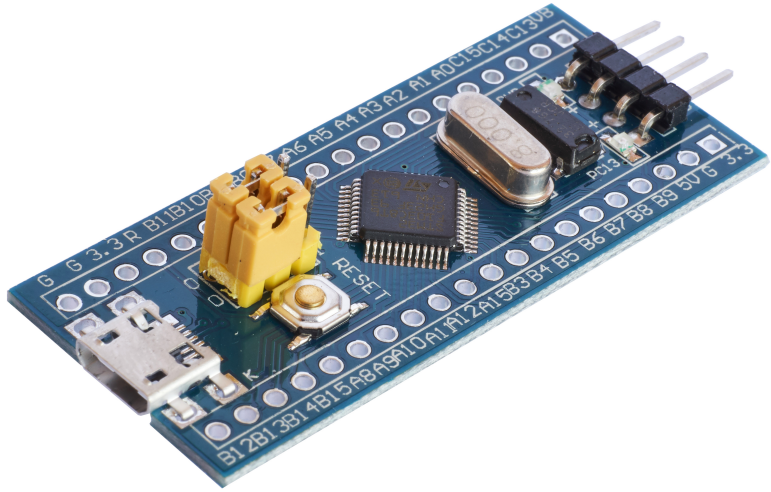

Figure 6: STM32F103 blue pill

ST-Link for the blue pill board

Figure 7: ST-Link

Stellaris/Tiva Launchpad

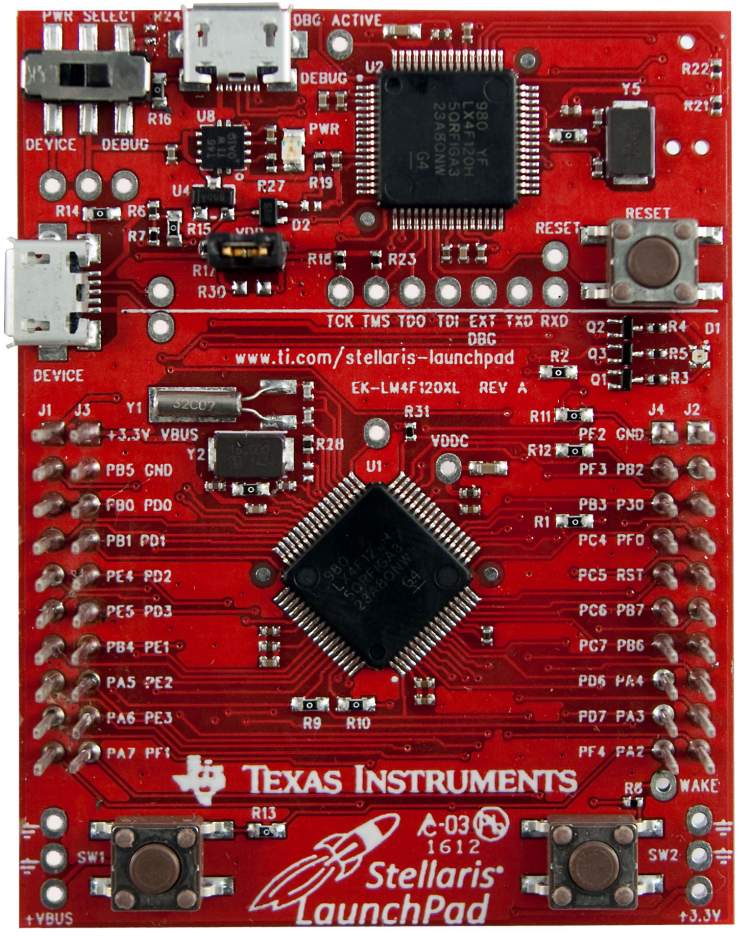

Figure 8: TI Stellaris/Tiva launchpad

Peripherals

- General Purpose I/O (GPIO) pins
- Timer / Counter units
- Serial interfaces: USART, SPI, I2C
- Analog to Digital Convertors
- Pulse width modulation

Programming a peripheral

- Each peripheral has dozens of registers (memory mapped locations) associated with it.
- You program the peripheral by writing special bit patterns to these registers.
- The technical reference manual describing these can easily run into more than 1000 pages!

GPIO pin programming

- GPIO pins are grouped into PORTS. Each port (say PortA, PortB) has usually at least 8 pins associated with it.

GPIO pin programming

- GPIO ports have registers associated with them:
  - Setting the direction of each pin (IN/OUT)
  - Setting/Clearing each pin (configured as OUTPUT)
  - Reading the digital logic level on the pins (configured as INPUT)

GPIO pin programming: bit numbering

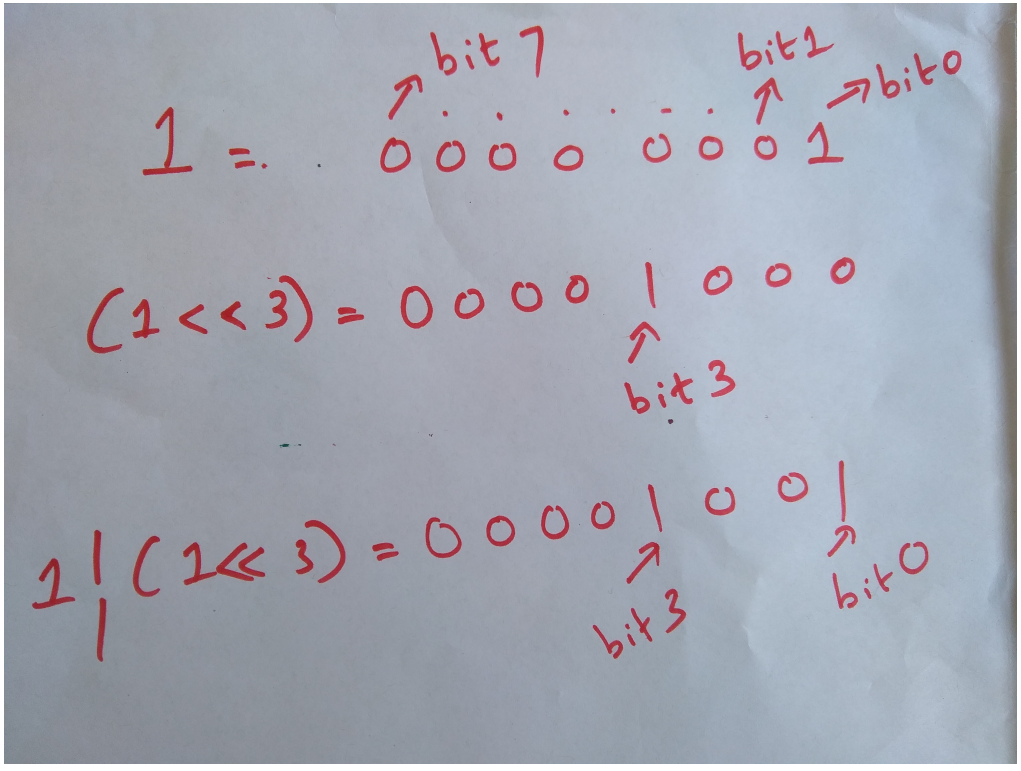

Figure 9: Bit numbering convention

GPIO pin programming

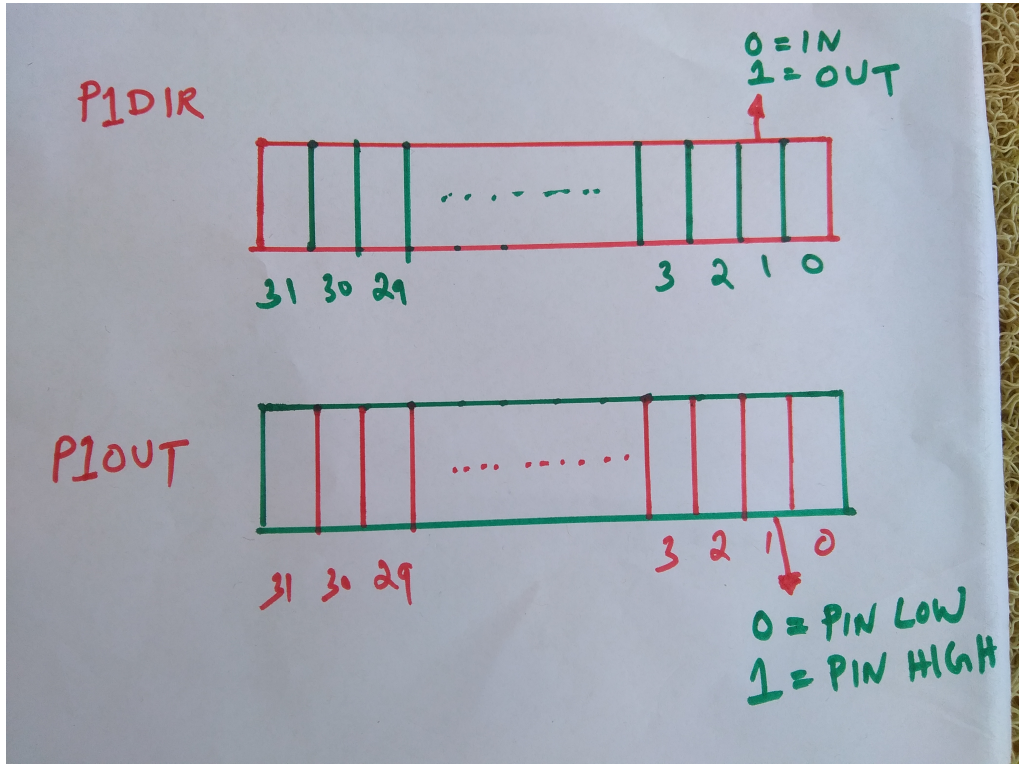

Figure 10: GPIO pin programming

GPIO pin programming

- Enable the port
- Configure the pin as input or output
- Set/Clear the pin (for output)
- Read the pin (for input)

GPIO pin programming: STM32F3DISCOVERY board

- Board has 8 LED's connected to PE (Port E) pins. Two of these are on PE9 and PE11.
- First, bit 21 of AHBENR has to be SET to enable PE (Port E).
- The mode register (MODER) has two bits reserved for each pin, the rightmost two bits for the 0th pin.

GPIO pin programming: STM32F3DISCOVERY board

- If the two bits in MODER have value “01”, corresponding pin is a digital OUTPUT pin.
- Writing a “1” to bits 0 to 15 of GPIOE_BSRR results in corresponding pin getting a logic HIGH on it. Writing a “1” to bits 16 to 31 of GPIOE_BSRR results in the (N - 16)th pin (where N is the bit number) getting a logic LOW on it.

Level 1: Direct GPIO pin programming using raw pointers: STM32F3DISCOVERY board

```
unsafe {
        const RCC_AHBENR: u32 = 0x40021000 + 0x14;
        const GPIOE_BSRR: u32 = 0x48001018;
        const GPIOE_MODER: u32 = 0x48001000;

        /* Continued ... */
```

Direct GPIO pin programming using raw pointers: STM32F3DISCOVERY board

```
        let x = ptr::read_volatile(
                    RCC_AHBENR as *mut u32);
        ptr::write_volatile(RCC_AHBENR as *mut u32,
                            x | (1 << 21));
        ptr::write_volatile(GPIOE_MODER as *mut u32,
                            (1 << 18));
        // Turn on the LED (red) on PE9
        ptr::write_volatile(GPIOE_BSRR as *mut u32,
                            (1 << 9));

    }
```

Level 2: Auto-generating register access functions using svd2ust

- svd2rust (https://github.com/japaric/svd2rust)
- Input: XML file describing peripherals/registers/bit fields (called an "SVD" file: http://www.keil.com/pack/doc/CMSIS/SVD/html/index.html)
- Output: A Rust library with data structures and functions for accessing the peripheral registers.

A sample “svd” file

[https://raw.githubusercontent.com/m-labs/dslite2svd/master/svd/tm4c123x.xml]

```
<name>GPIOC</name>
        <description>GPIO Port C</description>
        <value>2</value>
      </interrupt>
      <interrupt>
        <name>GPIOD</name>
        <description>GPIO Port D</description>
        <value>3</value>
      </interrupt>
```

Auto-generated Rust file

[https://raw.githubusercontent.com/m-labs/dslite2svd/master/crates/tm4c123x/src/lib.rs]

```
impl super::LOAD {
        #[doc = r" Modifies the contents of the register"]
        #[inline]
        pub fn modify<F>(&self, f: F)
        where
        for<'w> F: FnOnce(&R, &'w mut W) -> &'w mut W,
        {
            let bits = self.register.get();
            let r = R { bits: bits };
            let mut w = W { bits: bits };
            f(&r, &mut w);
            self.register.set(w.bits);
        }
```

Code written using this interface

[code/stm32f3-svd2rust]

```
// Put ON LED's on PE9, PE11
let p = Peripherals::take().unwrap();
let gpioe = p.GPIOE;
let rcc = p.RCC;

/* continued ...*/
```

Code written using this interface

```
rcc.ahbenr.modify(|r, w| w.iopeen().set_bit());
gpioe.moder.write(|w| w.moder11().output()
                       .moder9().output());
gpioe.bsrr.write(|w| w.bs9().set()
                      .bs11().set());
```

Why this approach is good

- Old, C style: peripheral registers as global "variables" which can be used in *any* way you like.
- This approach: peripheral registers are resources whose usage is constrained in certain ways.
- The abstractions have NO runtime cost!

Why this approach is good

- No error prone bit-twiddling!
- svd2rust automates a very tedious process.

A bug!

```
// Put ON LED's on PE9, PE11
let p = Peripherals::take().unwrap();
let gpioe = p.GPIOE;
let rcc = p.RCC;

rcc.ahbenr.modify(|r, w| w.iopeen().set_bit());
gpioe.moder.write(|w| w.moder11().output());
gpioe.bsrr.write(|w| w.bs9().set()
                      .bs11().set());
```

A bug!

We still don't have a sufficiently high-level representation of a "peripheral" in our code!

The “embedded-hal” to the rescue

[code/stm32f3-hal1]

```
let p = stm32f30x::Peripherals::take().unwrap();
let mut rcc = p.RCC.constrain();
let mut gpioe = p.GPIOE.split(&mut rcc.ahb);

/* continued */
```

The “embedded-hal” to the rescue

```
let mut pe9: PE9<Output<PushPull>> = gpioe.pe9
            .into_push_pull_output(&mut gpioe.moder,
                                   &mut gpioe.otyper);
let mut pe11: PE11<Output<PushPull>> = gpioe.pe11
            .into_push_pull_output(&mut gpioe.moder,
                                   &mut gpioe.otyper);

pe9.set_high();
pe11.set_high();
```

The “embedded-hal” to the rescue

```
let mut pe8: PE8<Input<Floating>> = gpioe.pe8;

pe8.set_high(); // compile time error!
```

The “embedded-hal” to the rescue

```
let mut pe8: PE8<Input<Floating>> = gpioe.pe8;

// compile time error!
let mut pe8_1: PE8<Output<PushPull>> = gpioe.pe8
        .into_push_pull_output(&mut gpioe.moder,
                               &mut gpioe.otyper);
```

Move semantics to the rescue!

The “embedded-hal” to the rescue

```
// pe9 is currently push-pull output
let mut pe9_1 = pe9.into_floating_input(
                    &mut gpioe.moder,
                    &mut gpioe.pupdr);

// compile time error!
pe9.set_high();
```

Move semantics to the rescue, once again!

The “embedded-hal” to the rescue

- Peripherals are represented by statically typed entities in code.
- The type system is used to encode attributes of a peripheral like: is it a digital I/O pin, is it input/output etc . . .
- Single Ownership enforced by move semantics helps in preventing resource conflicts.

Zero-cost abstractions

- The *embedded-hal* doesn't have any overhead!
- Code written using the HAL is as compact as raw register manipulation code!
- You can verify this by dis-assembling the generated machine code using *arm-none-eabi-objdump*.

Coding with the help of a board support crate

[code/stm32f3-board1]

```
let p = stm32f30x::Peripherals::take().unwrap();
let mut rcc = p.RCC.constrain();
let gpioe = p.GPIOE.split(&mut rcc.ahb);

/*continued...*/
```

Coding with the help of a board support crate

```
// 8 LED's on the board!
let mut leds = Leds::new(gpioe);

for led in leds.iter_mut() {
        led.on();
}
```

Coding with the help of a board support crate

Running LED's: [code/stm32f3-board2]

Generating a random bitstream using a linear feedback shift register

[code/iterators/stm32f3-svd2rust]

Using “embedded-hal” traits for writing generic drivers

- Very little code sharing in the embedded systems community.
- Solution: define a broad interface (expressed as *traits* in Rust) and write driver code which uses *only* this interface.

Using “embedded-hal” traits for writing generic drivers

- Distribute the driver code on crates.io.
- Create implementations of this interface for various microcontrollers!
- Have Fun!!

An example: driver for a simple ADC (MCP3008)

[code/mcp3008-example]

An example embedded-hal trait

```rust
/// Single digital output pin
pub trait OutputPin {
    /// Sets the pin low
    fn set_low(&mut self);

    /// Sets the pin high
    fn set_high(&mut self);
}
```

An embedded-hal implementation of this trait

```rust
// linux-embedded-hal implementation
pub struct Pin(pub sysfs_gpio::Pin);

impl hal::digital::OutputPin for Pin {
    fn set_low(&mut self) {
        self.0.set_value(0).unwrap()
    }

    fn set_high(&mut self) {
        self.0.set_value(1).unwrap()
    }
}
```

The MCP3008 ADC

- 10 bit ADC with 8 input channels
- Communicates over the SPI bus

The MCP3008 ADC

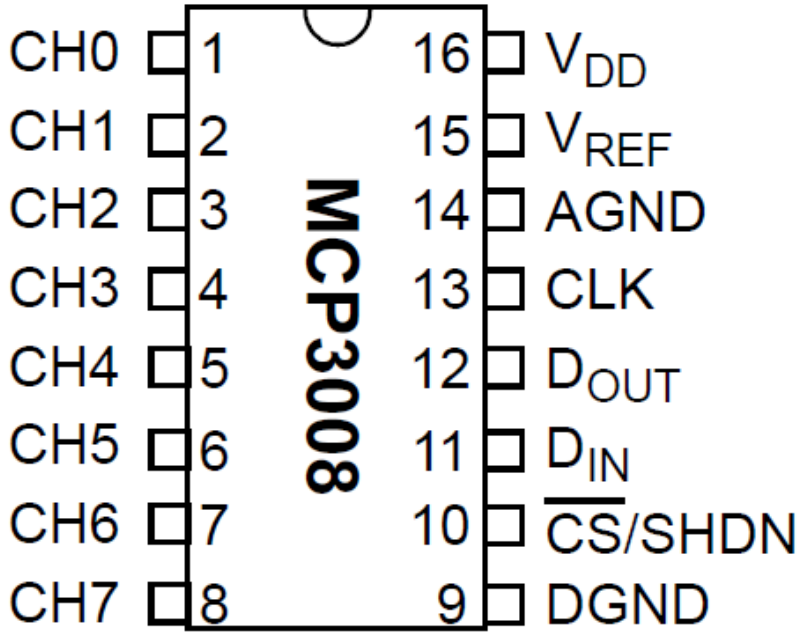

Figure 11: MCP3008 ADC

The MCP3008 ADC

- CLK pin (pin 13) of MCP3008 connected to pin configured as Clock of the microcontroller's SPI peripheral.
- DOUT (pin 12) of the MCP3008 connected to the MISO pin of the microcontroller's SPI peripheral.

The MCP3008 ADC

- DIN (pin 11) of the MCP3008 connected to the MOSI pin of the microcontroller's SPI perihperal.
- CS (pin 10) of the MCP3008 connected to the digital I/O pin of the microcontroller which is configured as the chip select output.

The MCP3008 driver - code walkthrough

```
extern crate embedded_hal as hal;

use hal::blocking::spi::Transfer;
use hal::spi::{Mode, Phase, Polarity};
use hal::digital::OutputPin;
```

The MCP3008 driver - code walkthrough

```
// The Transfer trait
pub trait Transfer<W> {
    /// Error type
    type Error;

    /// Sends `words` to the slave.
    /// Returns the `words` received from the slave
    fn transfer<'w>(&mut self,
        words: &'w mut [W]) ->
            Result<&'w [W], Self::Error>;
}
```

The MCP3008 driver - code walkthrough

```rust
/// SPI mode
pub const MODE: Mode = Mode {
    phase: Phase::CaptureOnFirstTransition,
    polarity: Polarity::IdleLow,
}
```

The MCP3008 driver - code walkthrough

```
/// MCP3008 driver
pub struct Mcp3008<SPI, CS> {
    spi: SPI,
    cs: CS,
}
```

The MCP3008 driver - code walkthrough

```
impl<SPI, CS, E> Mcp3008<SPI, CS>
    where SPI: Transfer<u8, Error = E>,
          CS: OutputPin
{
    /// Creates a new driver from an SPI peripheral and
    /// a chip select digital I/O pin.
    pub fn new(spi: SPI, cs: CS) ->
        Result<Self, E> {
            let mcp3008 = Mcp3008 { spi: spi, cs: cs };
            Ok(mcp3008)
    }
}
```

The MCP3008 driver - code walkthrough

```
impl<SPI, CS, E> Mcp3008<SPI, CS>
    where SPI: Transfer<u8, Error = E>,
          CS: OutputPin
{
    pub fn new(spi: SPI, cs: CS) ->
        Result<Self, E> { }

    /// Read a MCP3008 ADC channel and
    /// return the 10 bit value as a u16
    pub fn read_channel(&mut self, ch: Channels8) ->
        Result<u16, E> {  }

}
```

The MCP3008 driver - code walkthrough

```
self.cs.set_low();
let mut buffer = [0u8; 3];
buffer[0] = 1;
buffer[1] = ((1 << 3) | (ch as u8)) << 4;

self.spi.transfer(&mut buffer)?;

self.cs.set_high();
let r = (((buffer[1] as u16) << 8) |
         (buffer[2] as u16)) & 0x3ff;
Ok(r)
```

The MCP3008 driver - code walkthrough

```
// An application program will use
// "read_channel" like this.

let m = Mcp3008::new(x, y);
let r = m.read_channel(Channels8::CH0);
```

The MCP3008 driver - code walkthrough

```
pub enum Channels8 {
    CH0,
    CH1,
    CH2,
    CH3,
    CH4,
    CH5,
    CH6,
    CH7,
}
```

Application code walkthrough

```
// Runs on a Raspberry Pi using the
// linux-embedded-hal

extern crate linux_embedded_hal as hal;
extern crate adc_mcp3008;

use std::thread;
use std::time::Duration;

use adc_mcp3008::{Mcp3008, Channels8};
use hal::spidev::{self, SpidevOptions};
use hal::{Pin, Spidev};
use hal::sysfs_gpio::Direction;
```

Application code walkthrough

```
/* Configure SPI */

let mut spi = Spidev::open("/dev/spidev0.0").unwrap();
let options = SpidevOptions::new()
    .bits_per_word(8)
    .max_speed_hz(1_000_000)
    .mode(spidev::SPI_MODE_0)
    .build();
spi.configure(&options).unwrap();
```

Application code walkthrough

```
// Configure the Digital I/O Pin
// to be used as Chip Select

let ncs = Pin::new(25);
ncs.export().unwrap();
while !ncs.is_exported() {}
ncs.set_direction(Direction::Out).unwrap();
ncs.set_value(1).unwrap();
```

Application code walkthrough

```
let mut mcp3008 = Mcp3008::new(spi, ncs).unwrap();
```

Application code walkthrough

```
loop {
    let val = mcp3008.read_channel(Channels8::CH0);
    println!("{:?}", val);
    thread::sleep(Duration::from_millis(1000));
}
```

How you can become a contributor!

- Use *svd2rust* for auto-generating peripheral access functions for processors which currently don't have this facility. [example: https://crates.io/crates/msp432p401r]
- Write *embedded-hal* based drivers for all kinds of devices. [example: https://github.com/japaric/l3gd20]
- Implement the *embedded-hal* for a processor for which no implementation exists.

The Rust community (India)

- *Rust India* channel on telegram
- rust-lang.in will be up soon
- https://twitter.com/rustlangindia
- https://github.com/rustindia/
  - Rust for Undergrads: https://github.com/rustindia/Rust-for-undergrads

Beginner-level learning material

- https://doc.rust-lang.org/book/second-edition/ (official book)
- http://intorust.com/ (screencasts)

Advanced resources

- O'Reilly book: *Programming Rust* (http://shop.oreilly.com/product/0636920040385.do)
- Stanford CS140e: an amazing course which teaches you to write an OS kernel (in Rust) for the Raspberry Pi!

Resources on learning about Rust on microcontrollers

- http://blog.japaric.io/
- Fearless concurrency in your microcontroller (Rustfest 2017) (https://www.video.ethz.ch/events/2017/rust/c8682842-9e92-4563-aa9d-d49439e4d2ab.html)
- Rusty Robots (FOSDEM 2018) (https://fosdem.org/2018/schedule/event/rusty_robots/)

Thank you! You can contact me at: mail@pramode.net