How you can contribute to the Rust Embedded Programming Ecosystem

Pramode C.E

February 14, 2018

Objectives

- Learn some core Rust concepts quickly
- Understand how Rust can be used for microcontroller programming
- Find out how you can contribute to embedded development using Rust

Why Rust?

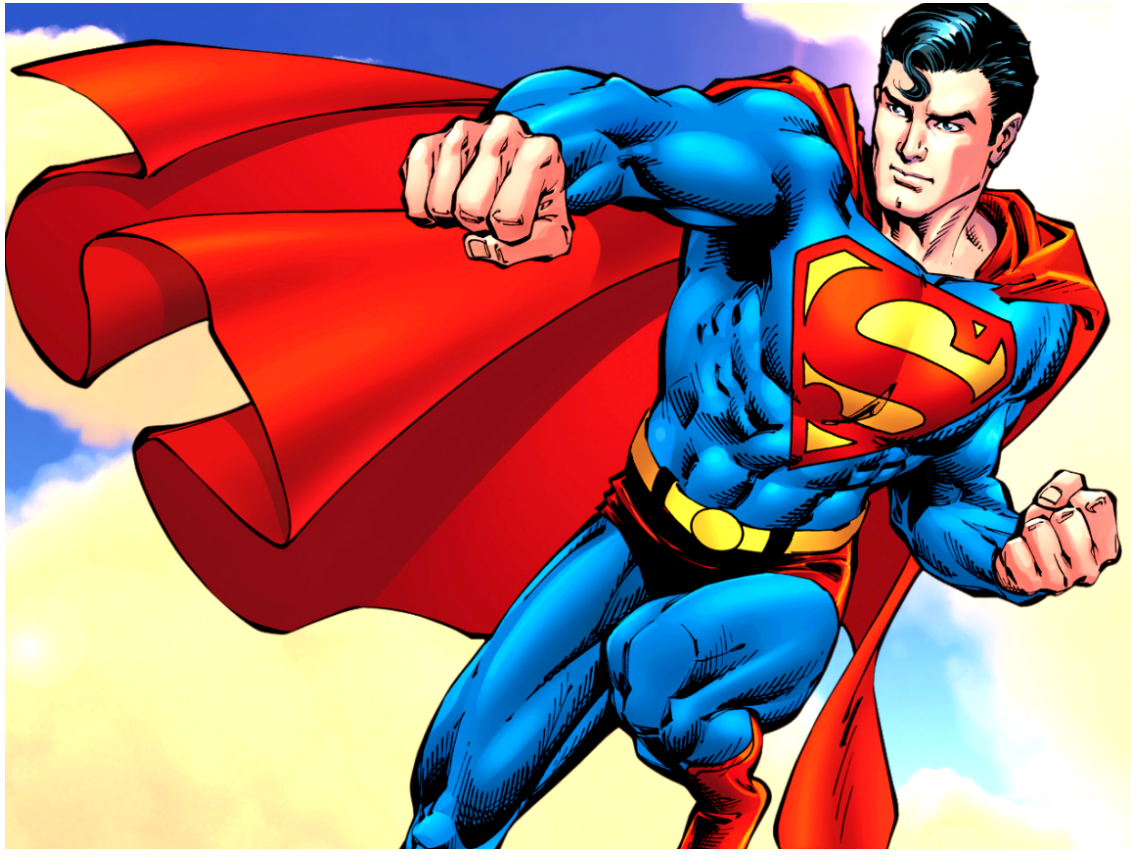

Figure 1: Superpowers!

Superpowers!

- Write safe and efficient code running on systems ranging from:
  - Very small microcontrollers with kilobytes of RAM to
  - Large servers and distributed/parallel computing systems
- Write code that is as efficient as C/C++ (in both memory usage and speed) but without problems like:
  - Buffer overflows
  - Null pointer dereferences
  - Dangling pointers
  - Memory leaks, Use-after-free

Superpowers!

- Write concurrent 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 bit of Rust history

- Started by Graydon Hoare as a personal project in 2006
- Mozilla foundation started sponsoring Rust in 2010
- Rust 1.0 released in May, 2015
- Regular six week release cycles
- Separate “stable” and “nightly” release channels provide access to experimental features without breaking stability.

Installation

Follow the instructions here: https://rustup.rs/

Core tools

- rustc, the compiler
- cargo, the package manager
- rustup, for managing different versions of the toolchain
- clippy, a *lint* tool

C memory safety issues

[code/basics/c-problems]

Rust and memory safety

- Rust achieves memory safety without a garbage collector - there is no significant *runtime* component associated with a Rust program.
- The fundamental ideas associated with memory safety:
  - Ownership and move semantics
  - Borrowing
  - Lifetimes

We will focus on ownership and move semantics.

Rust and memory safety

[code/basics/memory-safety]

Ownership and move semantics

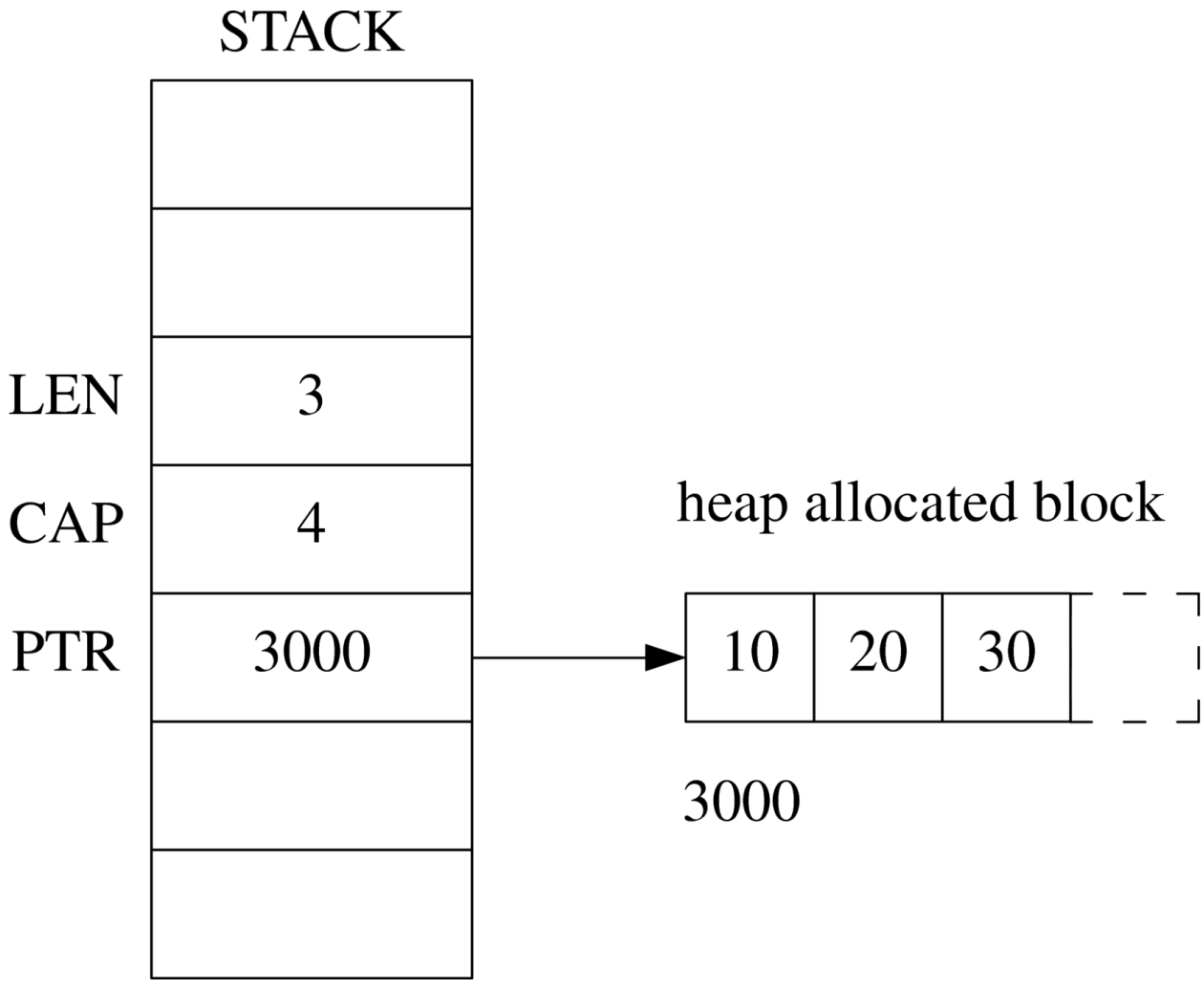

Figure 2: Representing a vector in memory

Ownership and move semantics

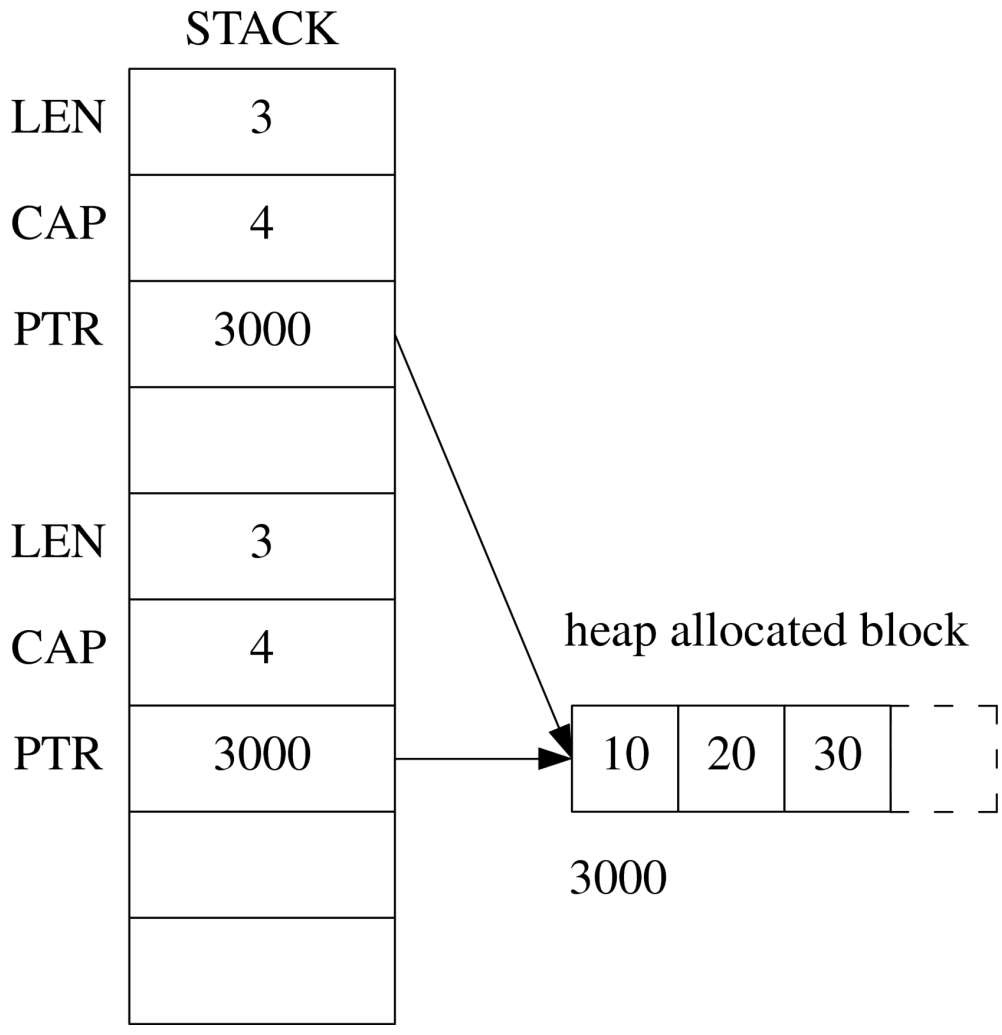

Figure 3: What happens when we try to copy a vector

Ownership and move semantics

- Move semantics guarantees *single ownership*.
- This can be exploited in clever ways to prevent a lot of bugs, even those which are not related to memory management.
- An interesting talk: https://www.video.ethz.ch/events/2017/rust/7fd8ceb9-a60a-4f13-9af4-5bbc6770bb75.html
- The concept of a “borrow” helps us to refer to an entity in memory without actually owning it.

Closures and iterators

[code/basics/closures-iterators]

Structures

[code/basics/closures-iterators]

Generics and Enums

[code/basics/generics-and-enums]

Traits

[code/basics/traits]

Unsafe

[code/basics/unsafe]

Why Rust on microcontrollers?

- Provide safer abstractions without runtime overhead.
- Create re-usable drivers.

Current Status

- 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/)
- 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

- We will focus on Jorge Aparicio's work.
- We will code for ARM Cortex-M micrcocontrollers.

The four levels of abstraction

- 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.

Before we start . . .

- 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 C toolchain (for the linker)
- Nightly Rust
- rustup / xargo
- Detailed instructions: http://pramode.in/2018/01/31/ti-launchpad-with-rust-new-io/

Our platforms

- The TI Stellaris/Tiva Launchpads using ARM Cortex-M4F processors
- The MSP432P401R Launchpad which uses an ARM Cortex-M4F processor
- The STM32F3DISCOVERY board which uses an ARM Cortex-M4F processor

Stellaris/Tiva Launchpad

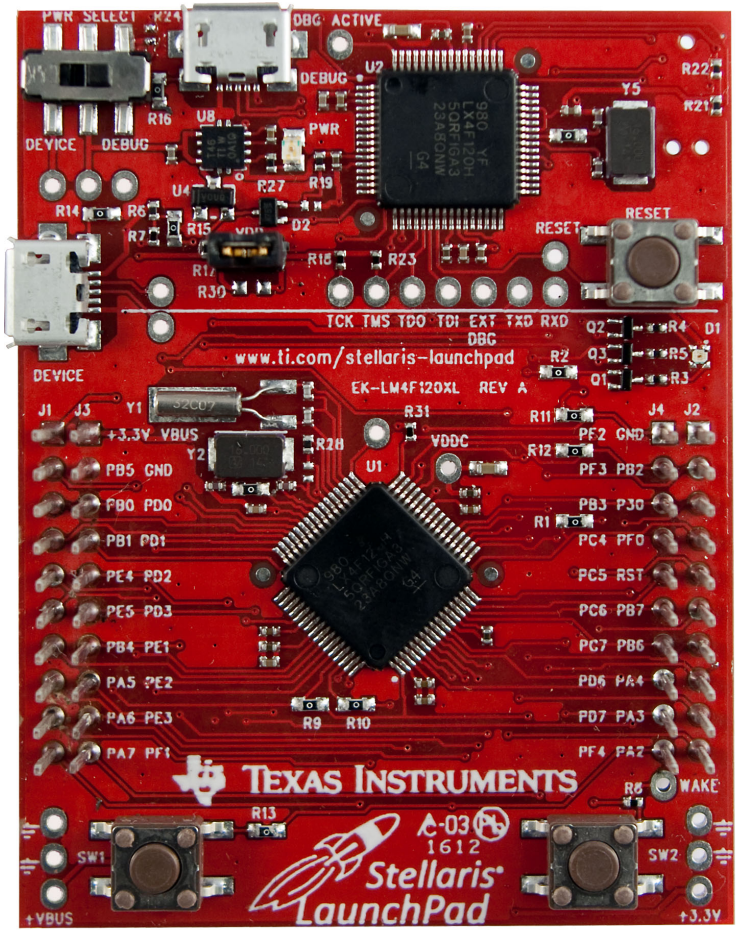

Figure 4: TI Stellaris/Tiva launchpad

The MSP432 Launchpad

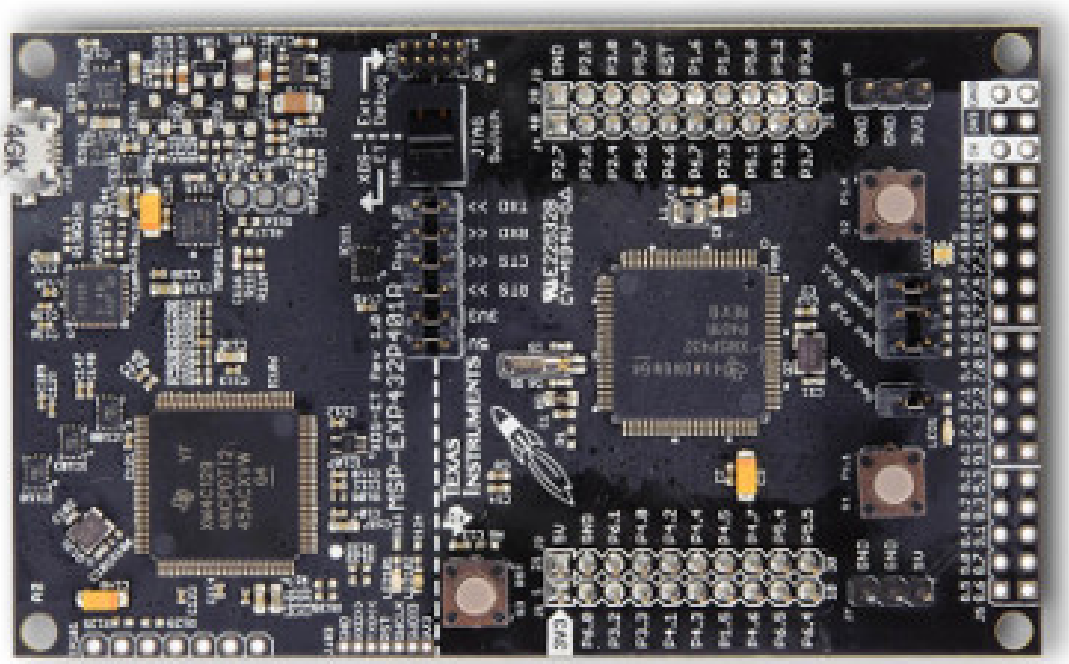

Figure 5: TI MSP432P401 Launchpad

The STM32F3DISCOVERY

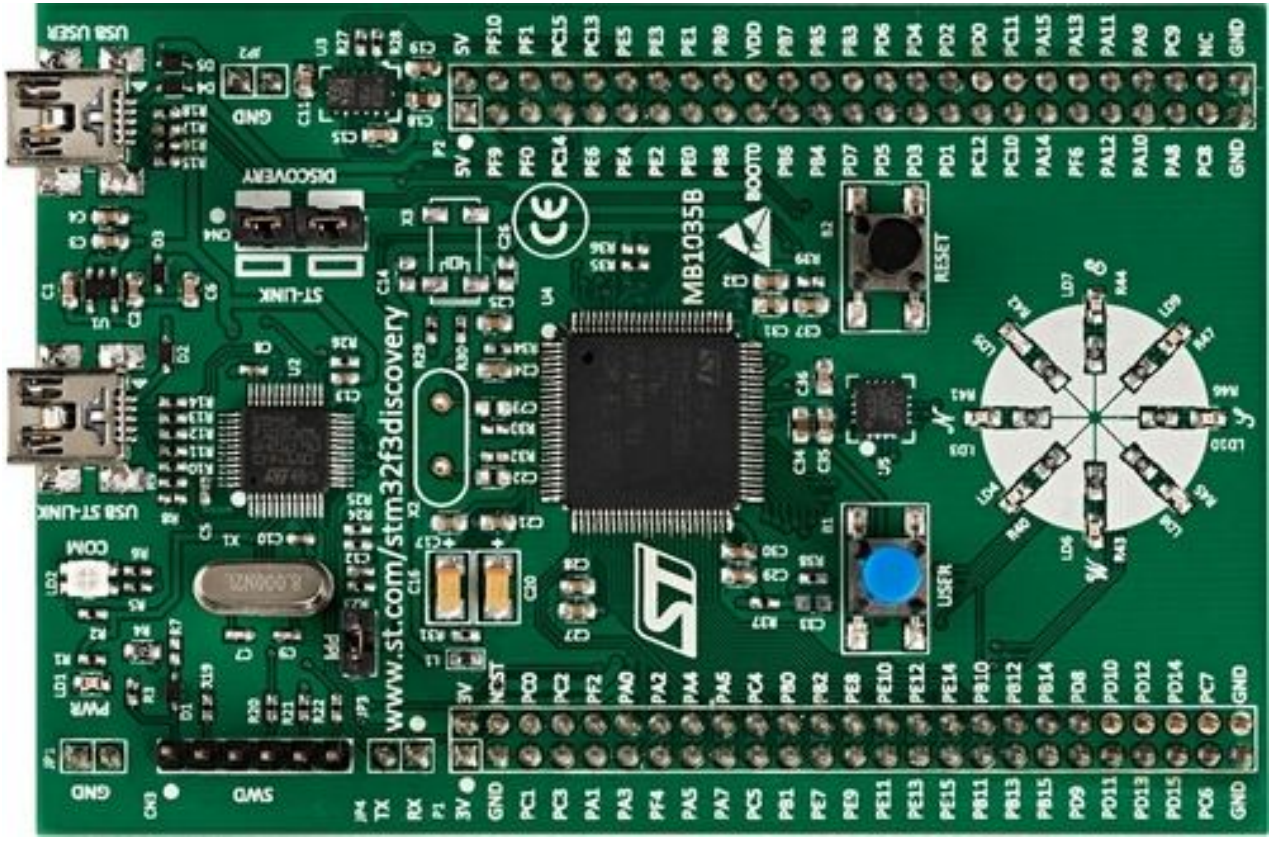

Figure 6: STM32F3DISCOVERY

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 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

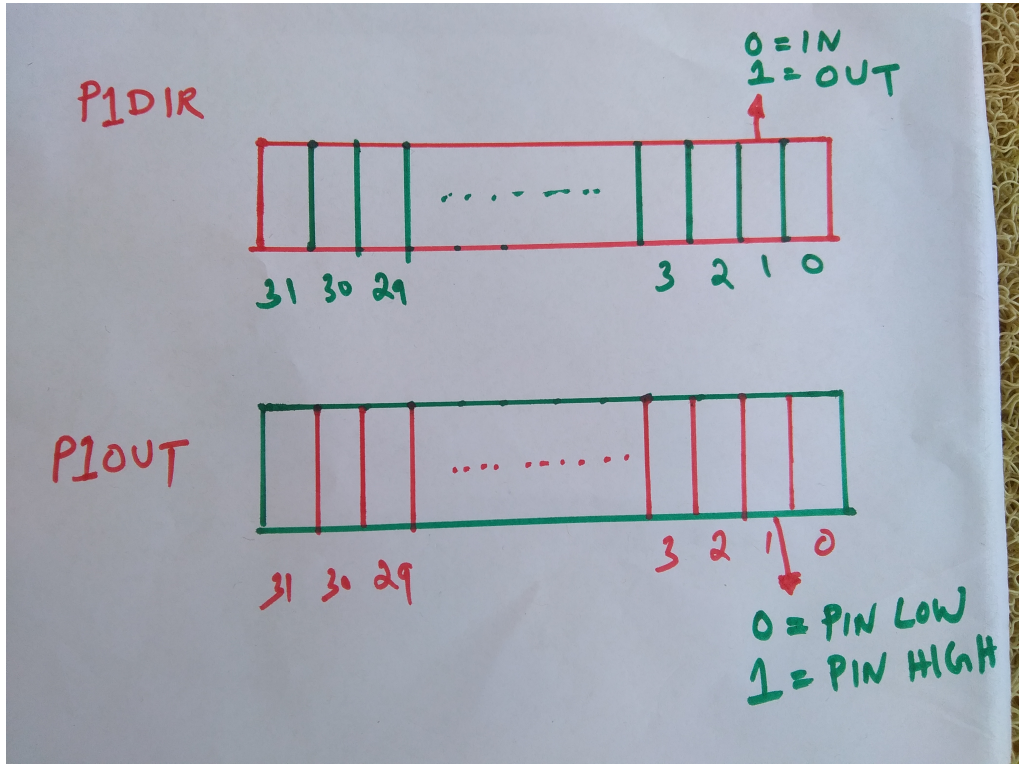

Figure 7: GPIO pin programming

GPIO pin programming: bit numbering

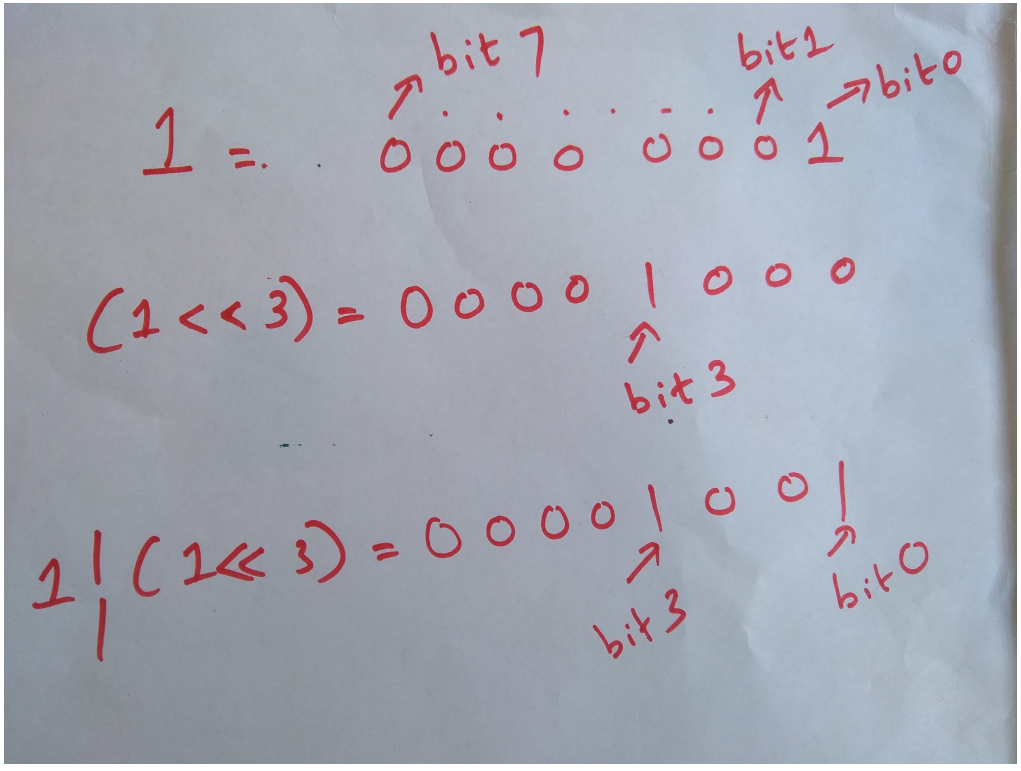

Figure 8: Bit numbering convention

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.

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;

        let x = ptr::read_volatile(
                    RCC_AHBENR as *mut u32);

        ptr::write_volatile(RCC_AHBENR as *mut u32,
                            x | (1 << 21));

        // continued ...
        // check code/stm32f3-raw1
    }
```

Problems

- Low level and error-prone bit manipulations

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;

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());
```

Advantage

- 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 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);

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);

// 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.
- 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]

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