An Overview of the Embedded Rust Ecosystem

Or, Why are there so many crates, and what do they do?

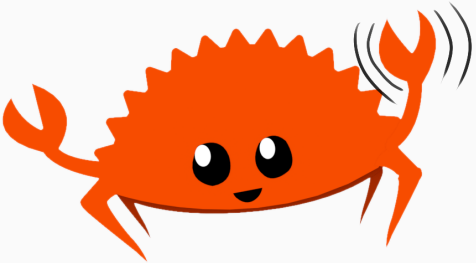
Frans Skarman aka. TheZoq2

July 18, 2020

Who am I?

Frans Skarman (@TheZoq2)

- Rust evangelist
- Maintainer of `stm32f1xx_hal`
- My desk is covered in rust



Who are you?

- Embedded rust

Who are you?

- Embedded rust
- Arduino

Who are you?

- Embedded rust
- Arduino
- Embedded C

Who are you?

- Embedded rust
- Arduino
- Embedded C
- Rust user

Goals

There are lots of embedded crates

How do they all fit together?

Goals

There are lots of embedded crates

How do they all fit together?

Explained through showing the need for them

Goals

There are lots of embedded crates

How do they all fit together?

Explained through showing the need for them

The initial code is not something you'd write

The lowest level

- Micro controllers control I/O through “registers”
- Typically memory mapped
- Read the datasheet for instructions

An example: turning on an LED

Using an `stm32f1` processor

An example: turning on an LED

Using an `stm32f1` processor

Start by reading the 1000 page datasheet:

An example: turning on an LED

Using an `stm32f1` processor

Start by reading the 1000 page datasheet:

Process:

- Power up the GPIO peripheral in the RCC peripheral
- Configure pin as output
- In the correct mode
- Set output to High

An example: turning on an LED

Configure pin 13 as output

- Control register at offset 0x04
- Offset from start of GPIOx
- Write 0b10 in bit 20, 21
- Write 0b01 in bit 23, 22

Same deal for the output value.

9.2.1 Port configuration register low (GPIOx_CRL) (x=A..G)

Address offset: 0x00

Reset value: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:30, 27:26, 23:22, 19:18, 15:14, 11:10, 7:6, 3:2 **CNFy[1:0]:** Port x configuration bits (y= 0 .. 7)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table*.
In input mode (MODE[1:0]=00):
00: Analog mode
01: Floating input (reset state)
10: Input with pull-up / pull-down
11: Reserved
In output mode (MODE[1:0] > 00):
00: General purpose output push-pull
01: General purpose output Open-drain
10: Alternate function output Push-pull
11: Alternate function output Open-drain

Bits 29:28, 25:24, 21:20, 17:16, 13:12, 9:8, 5:4, 1:0 **MODEy[1:0]:** Port x mode bits (y= 0 .. 7)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table*.
00: Input mode (reset state)
01: Output mode, max speed 10 MHz.
10: Output mode, max speed 2 MHz.
11: Output mode, max speed 50 MHz.

The rust code

```
startup_gpio()

const GPIOC_START: usize = 0x4001_1000;
const CRH_OFFSET: usize = 0x04;
const ODR_OFFSET: usize = 0x0C;

const OUTPUT_MODE: u32 = 0x01;
const PUSH_PULL: u32 = 0x10;

unsafe {
    *((GPIOC_START + CRH_OFFSET) as *mut u32) =
        (OUTPUT_MODE << 20) | (PUSH_PULL << 22);

    *((GPIOC_START + ODR_OFFSET) as *mut u32) =
        1 << 13;
}
```

The rust code

```
startup_gpio()

const GPIOC_START: usize = 0x4001_1000;
const CRH_OFFSET: usize = 0x04;
const ODR_OFFSET: usize = 0x0C;

const OUTPUT_MODE: u32 = 0x01;
const PUSH_PULL: u32 = 0x10;

unsafe {
    *((GPIOC_START + CRH_OFFSET) as *mut u32) =
        (OUTPUT_MODE << 20) | (PUSH_PULL << 22);

    *((GPIOC_START + ODR_OFFSET) as *mut u32) =
        1 << 13;
}
```

The rust code

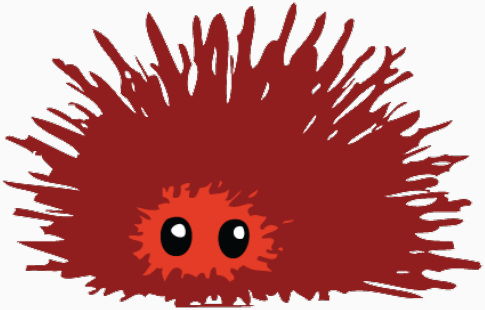
```
startup_gpio()

const GPIOC_START: usize = 0x4001_1000;
const CRH_OFFSET: usize = 0x04;
const ODR_OFFSET: usize = 0x0C;

const OUTPUT_MODE: u32 = 0x01;
const PUSH_PULL: u32 = 0x10;

unsafe {
    *((GPIOC_START + CRH_OFFSET) as *mut u32) =
        (OUTPUT_MODE << 20) | (PUSH_PULL << 22);

    *((GPIOC_START + ODR_OFFSET) as *mut u32) =
        1 << 13;
}
```

The rust code

```
startup_gpio()

const GPIOC_START: usize = 0x4001_1000;
const CRH_OFFSET: usize = 0x04;
const ODR_OFFSET: usize = 0x0C;

const OUTPUT_MODE: u32 = 0x01;
const PUSH_PULL: u32 = 0x10;

unsafe {
    *((GPIOC_START + CRH_OFFSET) as *mut u32) =
        (OUTPUT_MODE << 20) | (PUSH_PULL << 22);

    *((GPIOC_START + ODR_OFFSET) as *mut u32) =
        1 << 13;
}
```

The rust code

- Very unsafe
- Very unergonomic



SVD files

- Published by microcontroller manufacturers
- Describes the function of each register
- `svd2rust` generates rust crates

Peripheral Access Crates (PACs)

- Result of patches + `svd2rust`
- *Mostly* safe interface
- Adds a lot of zero cost abstraction
- Prevents re-use of peripherals

Peripheral Access Crates (PACs)

```
startup_gpio()

// Set pin 13 as a push pull output
dp.GPIOC.crh.modify(|_r, w| {
    w
        .mode13().output()
        .cnf13().push_pull()
});

// Actually change the output
dp.GPIOC.odr.modify(|_r, w| {
    w.odr13().high()
});
```

Peripheral Access Crates (PACs)

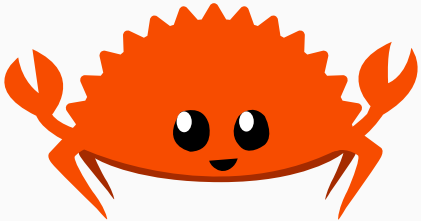
```
startup_gpio()

// Set pin 13 as a push pull output
dp.GPIOC.crh.modify(|_r, w| {
    w
        .mode13().output()
        .cnf13().push_pull()
});

// Actually change the output
dp.GPIOC.odr.modify(|_r, w| {
    w.odr13().high()
});
```

Peripheral Access Crates (PACs)

```
startup_gpio()

// Set pin 13 as a push pull output
dp.GPIOC.crh.modify(|_r, w| {
    w
        .mode13().output()
        .cnf13().push_pull()
});

// Actually change the output
dp.GPIOC.odr.modify(|_r, w| {
    w.odr13().high()
});
```

Peripheral Access Crates (PACs)

- Much higher level interface
- Zero cost abstraction
  - (In release mode)
- Most unsafety is now fixed



Peripheral Access Crates (PACs)

Some remaining issues

- No check for peripheral dependencies
- Or correct intialisation

$\Rightarrow$ Still requires through reading of the datasheet

Hardware Abstraction Layers (HALs)

A high level interface around the PAC

Hardware Abstraction Layers (HALs)

```
// Set up common structs (What startup_gpio() did)
let mut rcc = dp.RCC.constrain();

// Acquire and configure the GPIOC peripheral
let mut gpioc = dp.GPIOC.split(&mut rcc.apb2);

// Set the pin mode
let mut led = gpioc.pc13.into_push_pull_output(crh);
// Turn the led on
led.set_high()
```

Hardware Abstraction Layers (HALs)

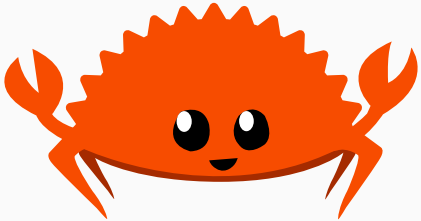
```
// Set up common structs (What startup_gpio() did)
let mut rcc = dp.RCC.constrain();

// Acquire and configure the GPIOC peripheral
let mut gpioc = dp.GPIOC.split(&mut rcc.apb2);

// Set the pin mode
let mut led = gpioc.pc13.into_push_pull_output(crh);
// Turn the led on
led.set_high()
```

Hardware Abstraction Layers (HALs)

```
// Set up common structs (What startup_gpio() did)
let mut rcc = dp.RCC.constrain();

// Acquire and configure the GPIOC peripheral
let mut gpioc = dp.GPIOC.split(&mut rcc.apb2);

// Set the pin mode
let mut led = gpioc.pc13.into_push_pull_output(crh);
// Turn the led on
led.set_high()
```

Hardware Abstraction Layers (HALs)

- High level interface
- Type state ensures correct initialisation

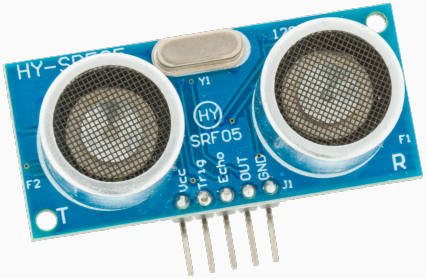


External Hardware

Using our HAL to talk to other hardware

HY-SRF05 ultrasonic distance sensor

- Send a pulse on one pin
- Wait for pulse on another
- Length of return pulse is proportional to distance



External Hardware (Implementation)

```
fn measure_distance(
    timer: Timer,
    (trig, echo): (PA0<Output>, echo: PA1<Input>)
) -> f32 {
    // Start pulse
    trig.set_high();
    timer.wait(10.ms());
    trig.set_low();

    // Wait for return pulse to start
    while echo.is_low() {}
    timer.start_measurement();
    // Wait for pulse to end
    while echo.is_high() {}
    return duration_to_distance(timer.elapsed())
}
```

External Hardware (Implementation)

```
fn measure_distance(
    timer: Timer,
    (trig, echo): (PA0<Output>, echo: PA1<Input>)
) -> f32 {
    // Start pulse
    trig.set_high();
    timer.wait(10.ms());
    trig.set_low();

    // Wait for return pulse to start
    while echo.is_low() {}
    timer.start_measurement();
    // Wait for pulse to end
    while echo.is_high() {}
    return duration_to_distance(timer.elapsed())
}
```

External Hardware (Implementation)

```
fn measure_distance(
    timer: Timer,
    (trig, echo): (PA0<Output>, echo: PA1<Input>)
) -> f32 {
    // Start pulse
    trig.set_high();
    timer.wait(10.ms());
    trig.set_low();

    // Wait for return pulse to start
    while echo.is_low() {}
    timer.start_measurement();
    // Wait for pulse to end
    while echo.is_high() {}
    return duration_to_distance(timer.elapsed())
}
```

External Hardware (Usage)

```
// Set up structs
let (rcc, gpioa) = initial_setup()

// Configure our pins
let trig = gpioc.pa0.into_push_pull_output(gpioa.crh);
let echo = gpioc.pa1.into_input(gpioa.crh);
// Configure our timer
let timer = dp.TIM1.setup();

// Use the driver
measure_distance(timer, (trig, echo))
```

External Hardware (Usage)

```
// Set up structs
let (rcc, gpioa) = initial_setup()

// Configure our pins
let trig = gpioc.pa0.into_push_pull_output(gpioa.crh);
let echo = gpioc.pa1.into_input(gpioa.crh);
// Configure our timer
let timer = dp.TIM1.setup();

// Use the driver
measure_distance(timer, (trig, echo))
```

External Hardware (Usage)

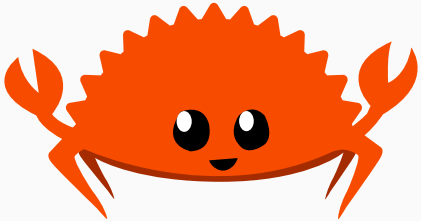
```
// Set up structs
let (rcc, gpioa) = initial_setup()

// Configure our pins
let trig = gpioc.pa0.into_push_pull_output(gpioa.crh);
let echo = gpioc.pa1.into_input(gpioa.crh);
// Configure our timer
let timer = dp.TIM1.setup();

// Use the driver
measure_distance(timer, (trig, echo))
```

External Hardware

We just made our first “driver”

Types ensure that things are configured correctly

Ownership ensures pins and timers aren’t shared

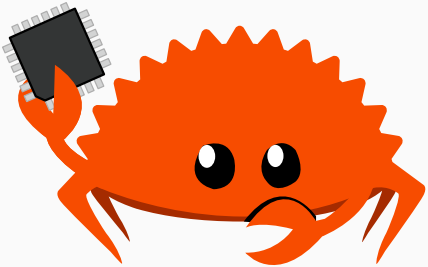


Another HAL

HAL for another microcontroller

Probably does similar things

With a slightly different API



Another HAL

- Our driver used our types
- It is not compatible with the other HAL

Embedded HAL

- Provides traits for common peripherals
  - GPIO
  - Timers
  - SPI
  - UART
  - etc...
- HALs implement them
- Drivers use them
- Driver re-use across the ecosystem!

HAL Usage Example

```
use embedded_hal::{InputPin, OutputPin, Timer};

fn measure_distance<I, O>(
    timer: Timer,
    (trig, echo): (O, I)
) -> f32
where
    I: InputPin,
    O: OutputPin,
    T: Timer,
{
    // Same as before
}
```

Misc. Crates

Real-Time Interrupt-driven Concurrency (RTIC)

- A concurrency framework for building real time systems
  - Tasks
  - Task communication
  - Real time analysis
  - etc.
- Previously known RTFM (Real Time For the Masses)
- See the previous talk by Emil Fresk

Board Support Crates (BSP)

Provides utilities for breakout boards

- Silkscreen pinouts
- Drivers for peripherals
- etc.

`cortex-x` crates

Cortex-<m,a,r> is the core of many arm microcontrollers

Some relevant crates:

- `cortex-m`: PAC for core peripherals (timers, debugger etc.)
- `cortex-m-rt`: minimal runtime (entry points, exceptions, ...)
  - included with `rt` feature in PACs
- `cortex-m-semihosting`: Debugging convenience

Conclusion

- `embedded-hal` is the core of ecosystem
- HALs implement embedded-hal traits
- usually by using PACs
- drivers use `e-h` traits to talk to external devices

Conclusion

- `embedded-hal` is the core of ecosystem
- HALs implement embedded-hal traits
- usually by using PACs
- drivers use `e-h` traits to talk to external devices

Other crates exist

- RTIC provides concurrency
- svd2rust generates PACs

Where should you start?

HALs are nice!

Starting from scratch?

- Pick a microcontroller with a good HAL

Where should you start?

HALs are nice!

Starting from scratch?

- Pick a microcontroller with a good HAL

Specific hardware in mind?

- Look for a HAL
- If none exist
  - Check for a PAC
  - Or make one using svd2rust if svd files are available
  - Build a HAL on top

Where should you start?

HALs are nice!

Starting from scratch?

- Pick a microcontroller with a good HAL

Specific hardware in mind?

- Look for a HAL
- If none exist
  - Check for a PAC
  - Or make one using svd2rust if svd files are available
  - Build a HAL on top

Similar steps for drivers

Resources

- A collection of most rust-embedded related things
  - `https://github.com/rust-embedded/awesome-embedded-rust`
- embedded-wg chatroom
- Community discord

Thank you for listening!