Continuing Education Center

DesignNews

Embedded Software using RUST

DAY 4 : Interfacing to Peripherals in Rust

Sponsored by

Webinar Logistics

- Turn on your system sound to hear the streaming presentation.
- If you have technical problems, click “Help” or submit a question asking for assistance.
- Participate in ‘Group Chat’ by maximizing the chat widget in your dock.

THE SPEAKER

Jacob Beningo

Visit 'Lecturer Profile'

Beningo Embedded Group - President

Focus: Embedded Software Consulting

An independent consultant who specializes in the design of real-time, microcontroller based embedded software.
He has published two books:

- Reusable Firmware Development
- MicroPython Projects
- Embedded Software Design

Writes a weekly blog for DesignNews.com focused on embedded system design techniques and challenges.

Visit www.beningo.com to learn more …

Visit 'Lecturer Profile' in your console for more details.

Course Sessions

- Introduction to Rust for Embedded Systems
- ”Hello Rust!”, using QEMU
- “Hello Rust!”, using the STM32F3
- Interfacing to Peripherals in Rust
- Becoming a Rust Expert

Continuing Education Center

Sponsored By

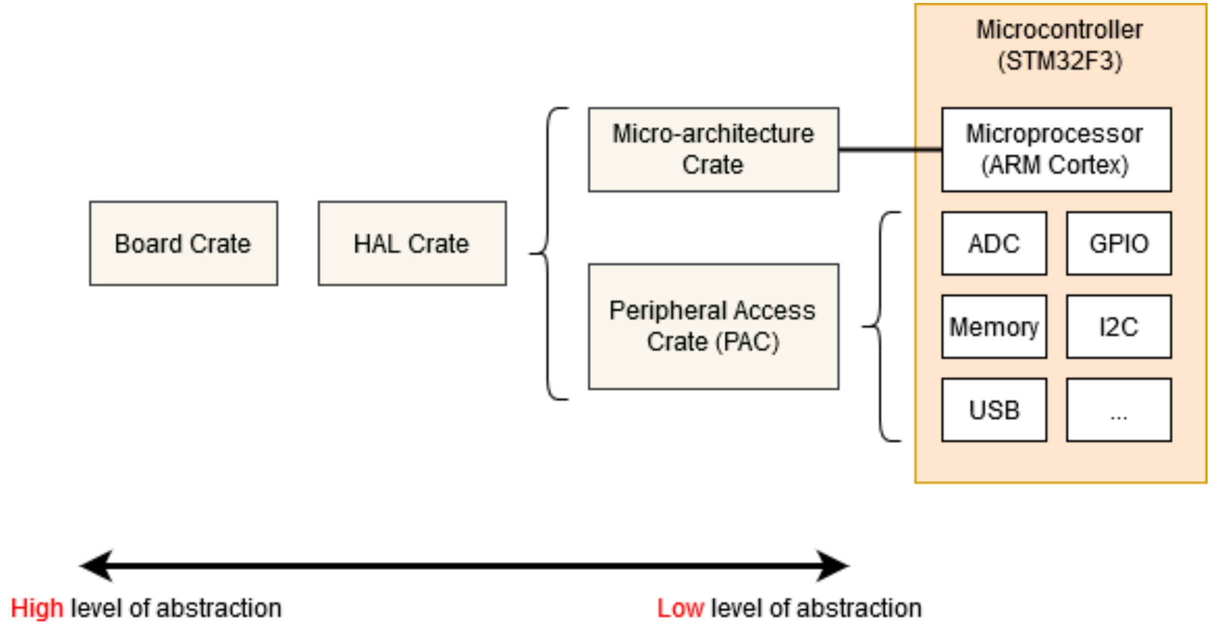
1 More on Crates

Crates

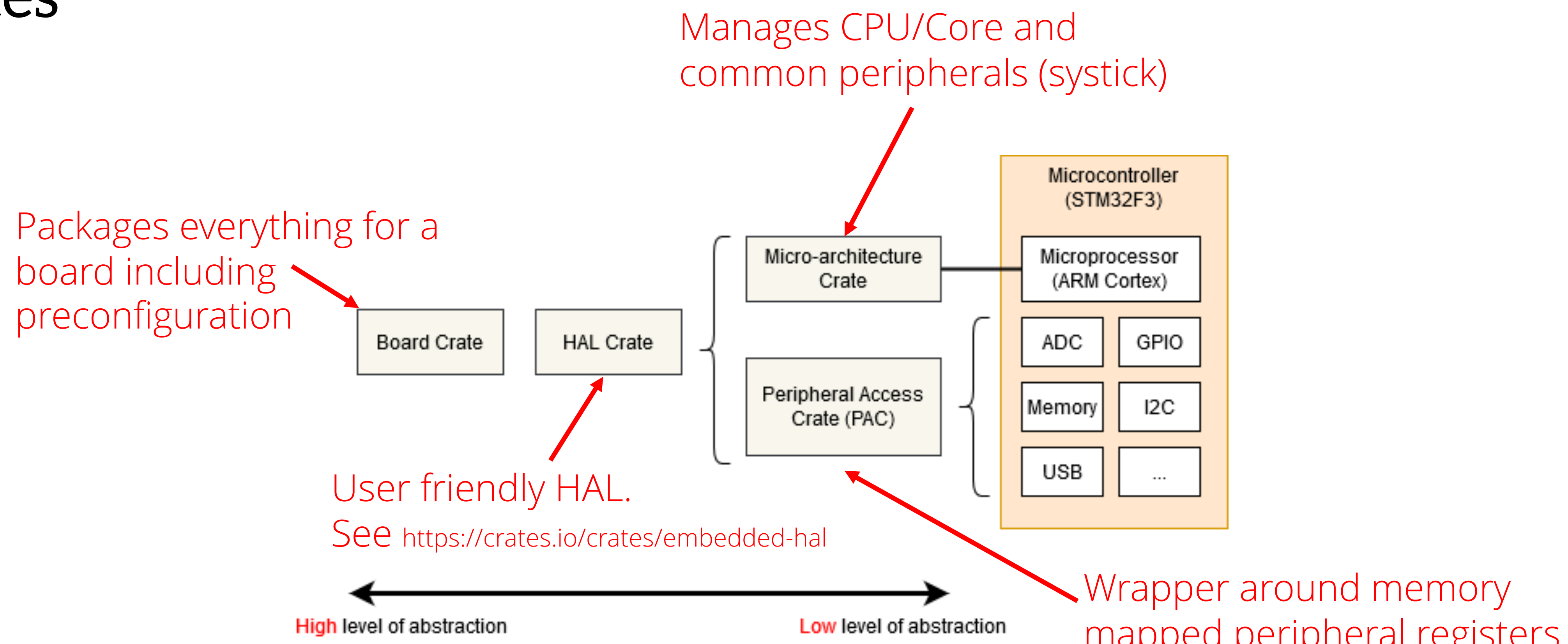
A crate is a compilation unit in Rust.

Rust treats each *.rs file as a crate file.

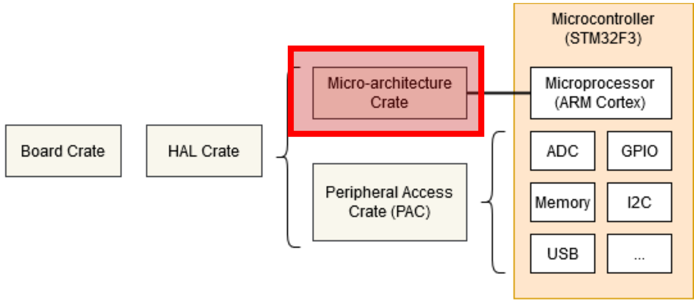
There are several types of crates for embedded developers.



Crates

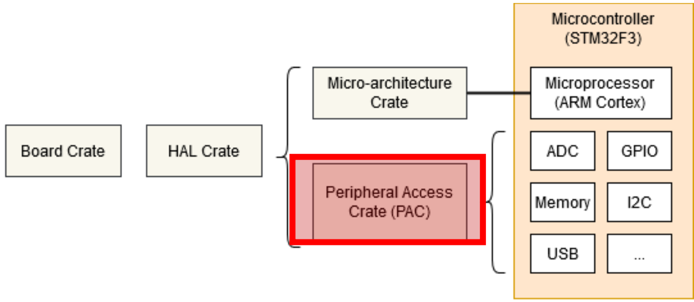

Manages CPU/Core and common peripherals (systick)
Packages everything for a board including preconfiguration
Microcontroller (STM32F3)
Micro-architecture Crate
Microprocessor (ARM Cortex)
Board Crate
HAL Crate
ADC
GPIO
Peripheral Access Crate (PAC)
Memory
I2C
USB
...
User friendly HAL.
See https://crates.io/crates/embedded-hal
High level of abstraction
Low level of abstraction
Wrapper around memory mapped peripheral registers

Microarchitecture Crate




```rust
#![no_std]
#![no_main]
use cortex_m::peripheral::{syst, Peripherals};
use cortex_m_rt::entry;
use panic_halt as _;

#[entry]
fn main() -> ! {
    let peripherals = Peripherals::take().unwrap();
    let mut systick = peripherals.SYST;
    systick.set_clock_source(syst::SystClkSource::Core);
    systick.set_reload(1_000);
    systick.clear_current();
    systick.enable_counter();
    while !systick.has_wrapped() {
        // Loop
    }

    loop {}
}
```

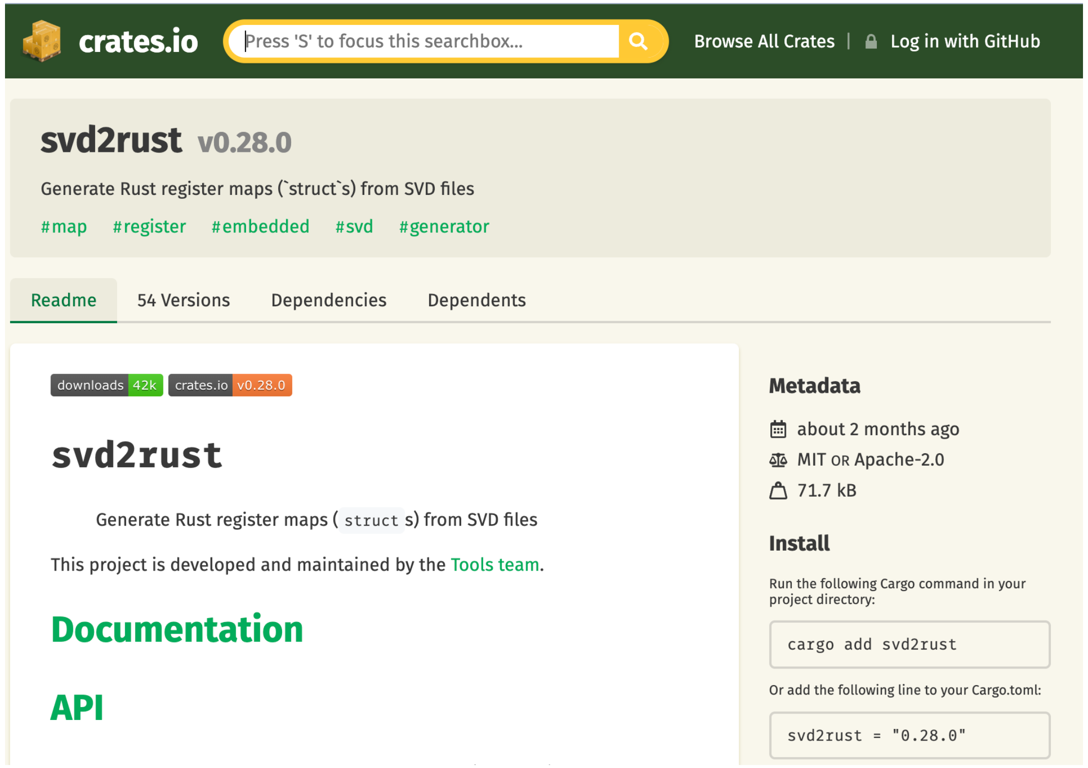
Peripheral Access Crate



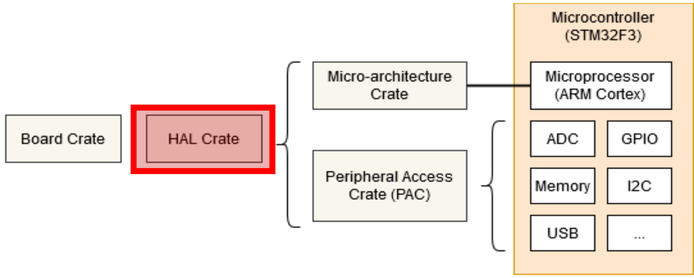

```rust
#![no_std]
#![no_main]

use panic_halt as _; // panic handler

use cortex_m_rt::entry;
use tm4c123x;

#[entry]
pub fn init() -> (Delay, Leds) {
    let cp = cortex_m::Peripherals::take().unwrap();
    let p = tm4c123x::Peripherals::take().unwrap();

    let pwm = p.PWM0;
    pwm.ctl.write(|w| w.globalsync0().clear_bit());
    // Mode = 1 => Count up/down mode
    pwm._2_ctl.write(|w| w.enable().set_bit().mode().set_bit());
    pwm._2_gena.write(|w| w.actcmpau().zero().actcmpad().one());
    // 528 cycles (264 up and down) = 4 loops per video line (2112 cycles)
    pwm._2_load.write(|w| unsafe { w.load().bits(263) });
    pwm._2_cmpa.write(|w| unsafe { w.compa().bits(64) });
    pwm.enable.write(|w| w.pwm4en().set_bit());
}
```

Peripheral Access Crates

Autogenerate your own creates



HAL Crate




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

use panic_halt as _; // panic handler

use cortex_m_rt::entry;
use tm4c123x_hal as hal;
use tm4c123x_hal::prelude::*;
use tm4c123x_hal::serial::{NewlineMode, Serial};
use tm4c123x_hal::sysctl;

#[entry]
fn main() -> ! {
    let p = hal::Peripherals::take().unwrap();
    let cp = hal::CorePeripherals::take().unwrap();

    // Wrap up the SYSCTL struct into an object with a higher-layer API
    let mut sc = p.SYSCTL.constrain();
    // Pick our oscillation settings
    sc.clock_setup.oscillator = sysctl::Oscillator::Main(
        sysctl::CrystalFrequency::_16mhz,
        sysctl::SystemClock::UsePll(sysctl::PllOutputFrequency::_80_00mhz),
    );
    // Configure the PLL with those settings
    let clocks = sc.clock_setup.freeze();

    // Wrap up the GPIO_PORTA struct into an object with a higher-layer API.
    // Note it needs to borrow `sc.power_control` so it can power up the GPIO
    // peripheral automatically.
    let mut porta = p.GPIO_PORTA.split(&sc.power_control);
```

```rust
    // Activate the UART.
    let uart = Serial::uart0(
        p.UART0,
        // The transmit pin
        porta
            .pa1
            .into_af_push_pull::<hal::gpio::AF1>(&mut porta.control),
        // The receive pin
        porta
            .pa0
            .into_af_push_pull::<hal::gpio::AF1>(&mut porta.control),
        // No RTS or CTS required
        (),
        (),
        // The baud rate
        115200_u32.bps(),
        // Output handling
        NewlineMode::SwapLFtoCRLF,
        // We need the clock rates to calculate the baud rate divisors
        &clocks,
        // We need this to power up the UART peripheral
        &sc.power_control,
    );

    loop {
        writeln!(uart, "Hello, World!\r\n").unwrap();
    }
}
```

What crate would you expect to use the most when developing your applications?

- Microarchitecture crate
- PAC
- Embedded HAL
- Board
- other

2 LED Example

Example Setup

Clone the STM32F3 Rust Example Directory

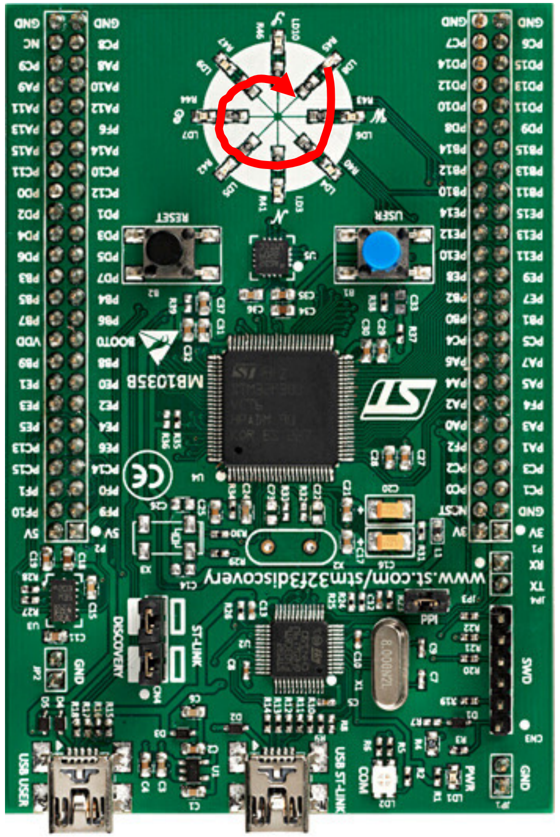
- https://github.com/rust-embedded/discovery

Project 05-led-roulette

- 01-background
- 02-requirements
- 03-setup
- 04-meet-your-hardware
- 05-led-roulette
- 06-hello-world
- 07-registers
- 08-leds-again
- 09-clocks-and-timers
- 10-serial-communication
- 11-usart
- 12-bluetooth-setup
- 13-serial-over-bluetooth
- 14-i2c
- 15-led-compass
- 16-punch-o-meter
- WIP-async-io-the-future

LED Roulette

```rust
#![deny(unsafe_code)]
#![no_main]
#![no_std]

use aux5::{Delay, DelayMs, LedArray, OutputSwitch, entry};

#[entry]
fn main() -> ! {
    let (mut delay, mut leds): (Delay, LedArray) = aux5::init();

    let ms = 50_u8;
    loop {
        for curr in 0..8 {
            let next = (curr + 1) % 8;

            leds[next].on().ok();
            delay.delay_ms(ms);
            leds[curr].off().ok();
            delay.delay_ms(ms);
        }
    }
}
```



LED Roulette aux5

```rust
//! Initialization code

#![no_std]

pub use panic_itm; // panic handler

pub use cortex_m_rt::entry;

pub use stm32f3_discovery::{leds::Leds, stm32f3xx_hal, switch_hal};
pub use switch_hal::{ActiveHigh, OutputSwitch, Switch, ToggleableOutputSwitch};

use stm32f3xx_hal::prelude::*;
pub use stm32f3xx_hal::{
    delay::Delay,
    gpio::{gpioe, Output, PushPull},
    hal::blocking::delay::DelayMs,
    pac,
};

pub type LedArray = [Switch<gpioe::PEx<Output<PushPull>>, ActiveHigh>; 8];

pub fn init() -> (Delay, LedArray) {
    let device_periphs = pac::Peripherals::take().unwrap();
    let mut reset_and_clock_control = device_periphs.RCC.constrain();

    let core_periphs = cortex_m::Peripherals::take().unwrap();
    let mut flash = device_periphs.FLASH.constrain();
    let clocks = reset_and_clock_control.cfgr.freeze(&mut flash.acr);
    let delay = Delay::new(core_periphs.SYST, clocks);
```

```rust

    // initialize user leds
    let mut gpioe = device_periphs.GPIOE.split(&mut reset_and_clock_control.ahb);
    let leds = Leds::new(
        gpioe.pe8,
        gpioe.pe9,
        gpioe.pe10,
        gpioe.pe11,
        gpioe.pe12,
        gpioe.pe13,
        gpioe.pe14,
        gpioe.pe15,
        &mut gpioe.moder,
        &mut gpioe.otyper,
    );

    (delay, leds.into_array())
}
```

What do you think is the most important component to crate creation?
- Function naming
- Encapsulation
- Dependency management
- Other

CEC Continuing Education Center

Sponsored By Digi-Key Corporation

3 Exceptions and Interrupts

BENINGO EMBEDDED GROUP

Exceptions and Interrupts

Exceptions, and interrupts, are a hardware mechanism by which the processor handles asynchronous events and fatal errors (i.e., executing an invalid instruction). Exceptions imply preemption and involve exception handlers, subroutines executed in response to the signal that triggered the event.

Exceptions and Interrupts

```rust
#[exception]
fn SysTick() {
    static mut COUNT: u32 = 0;

    // `COUNT` has transformed to type `&mut u32` and it's safe to use
    *COUNT += 1;
}
```

```rust
#[interrupt]
fn TIM2() {
    static mut COUNT: u32 = 0;

    // `COUNT` has type `&mut u32` and it's safe to use
    *COUNT += 1;
}
```

Exceptions and Interrupts

```
#[exception]
fn DefaultHandler(irqn: i16) {
    // custom default handler
}
```

```
#[exception]
fn HardFault(ef: &ExceptionFrame) -> ! {
    if let Ok(mut hstdout) = hio::hstdout() {
        writeln!(hstdout, "{:#?}", ef).ok();
    }

    loop {}
}
```

How do you feel about rust so far?

- Great
- Good
- Undecided
- Not impressed
- Ready for C++ class

Sponsored By

4 Going Further

Rust Resources

- Rust Website
- Rust Book
- Rust for Embedded Book
- Learning Rust for Embedded Systems
- Rust By Example
- RTIC: Real-Time Interrupt Driven Concurrency

Thank you for attending

Please consider the resources below:

- www.beningo.com
  - Blog, White Papers, Courses
  - Embedded Bytes Newsletter
    - http://bit.ly/1BAHYXm
  - Embedded Software Design
    - https://www.beningo.com/embedded-software-design/

From www.beningo.com under

- Blog > CEC – Embedded Software using Rust

Thank You

Sponsored by