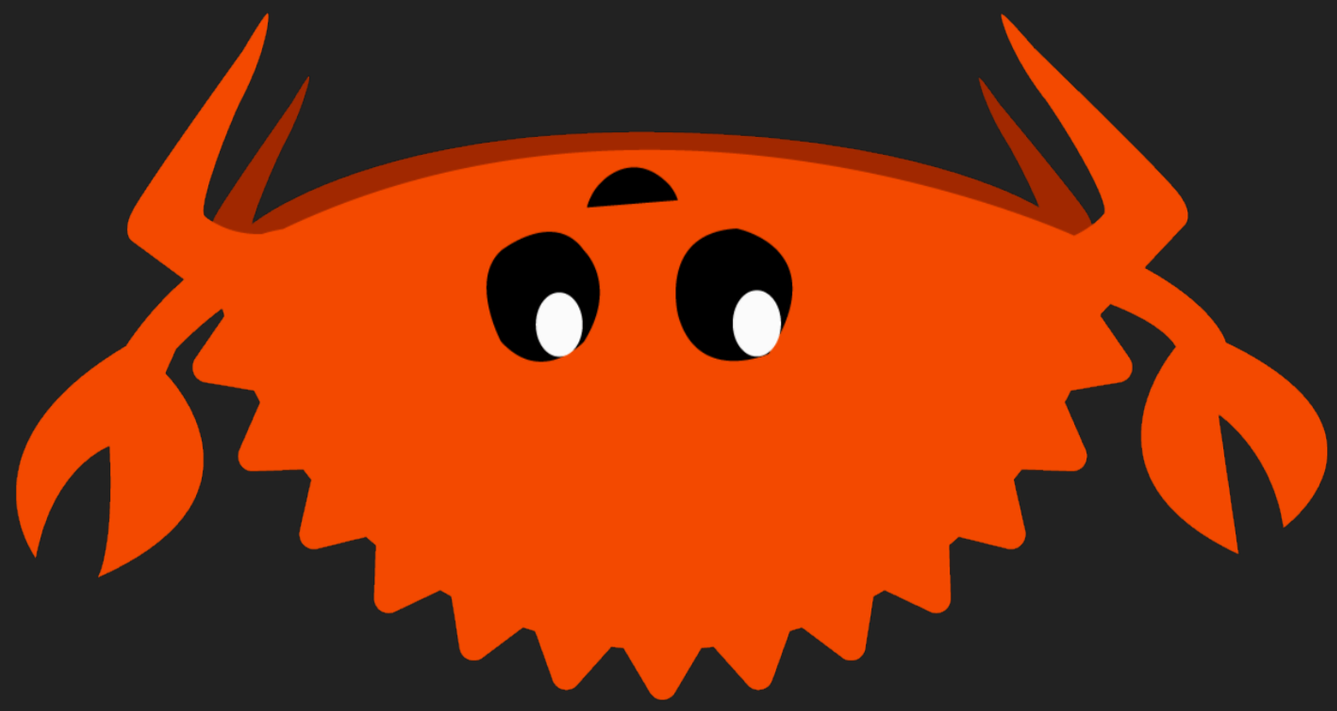
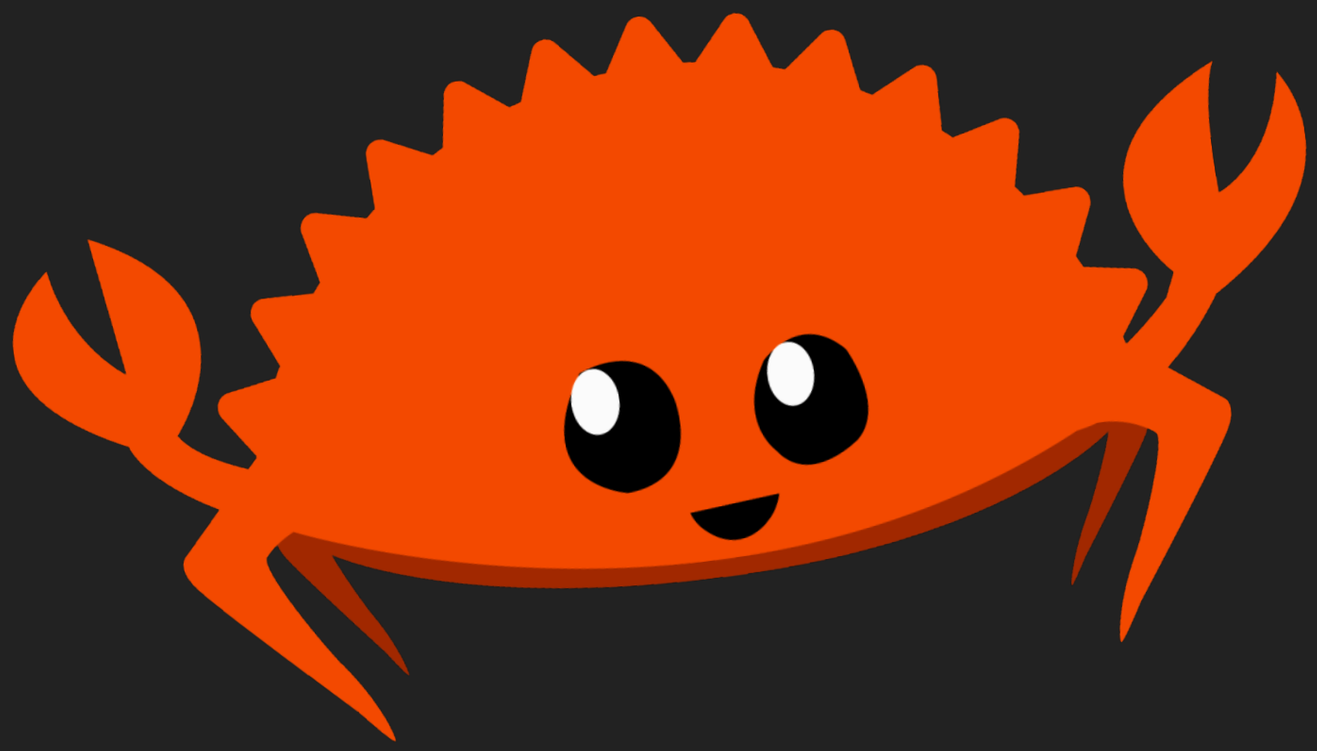
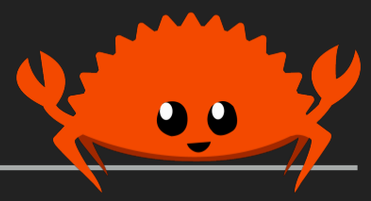




EMBEDDED RUST

INTRODUCTION AND PRACTICAL APPLICATIONS

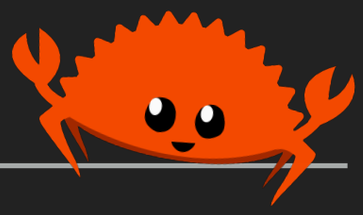


WHAT IS EMBEDDED

- No operating system environment, [no_std]
- Generally no memory allocator
- Constraints on program size, ram size, speed
- Direct access to pretty much everything (core, I/O, peripherals)
- Embedded as in a hardware device, WebAssembly or other? applications are out of scope

WHAT IS A MCU?

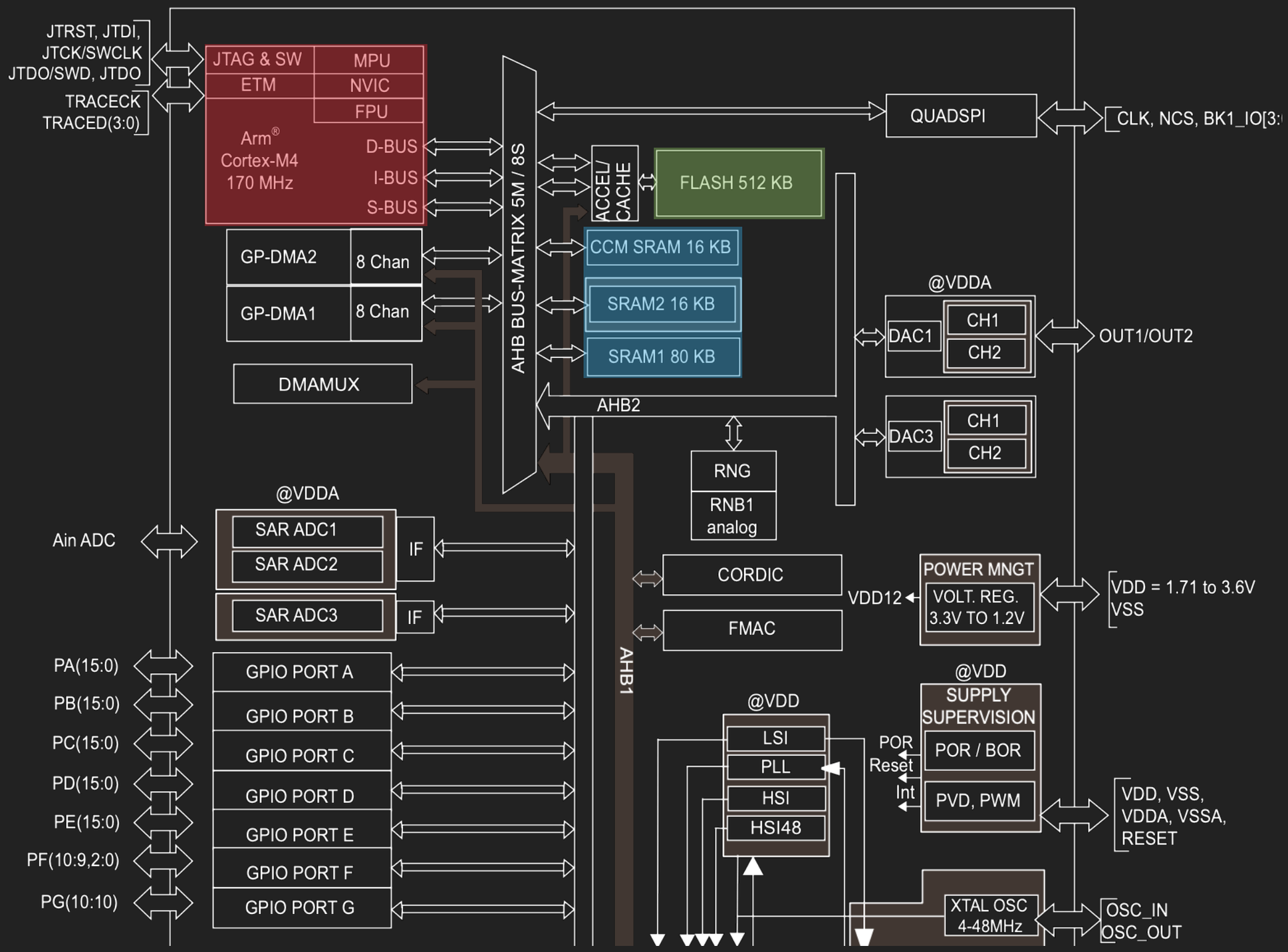
- Computer on a chip
- Contains everything needed to execute a program and interact with outer world
- Size from several mm2
- Often requires only one voltage rail to operate
- Most contain “debugger” interface (SWD, JTAG) that can be used to read/write memory and state.



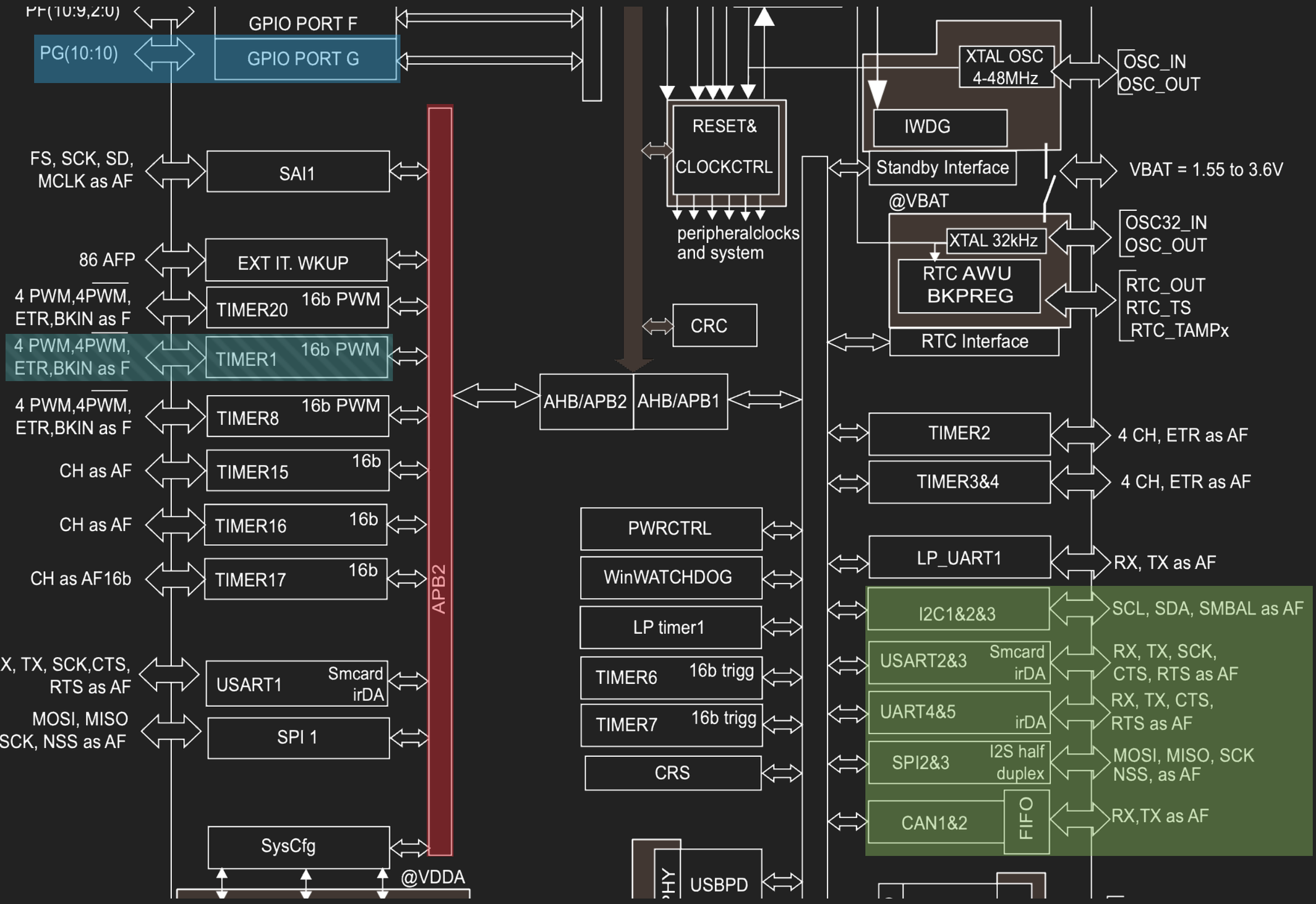
WHERE MICROCONTROLLERS ARE USED?

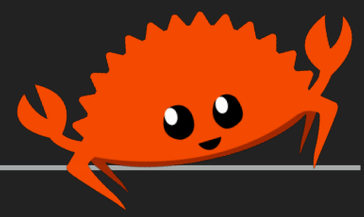
- Everywhere
- Consumer, space, military, medical, automotive, measurement, …
- Inside the Earth (drilling), underwater (submarines), on the water (ocean monitoring), on the ground (∞), in the air (airplanes), in space (satellites, rockets), on another planets (rovers, landers).

HOW PROGRAM IS STORED AND EXECUTED?



HOW TO TALK TO OUTSIDE WORLD

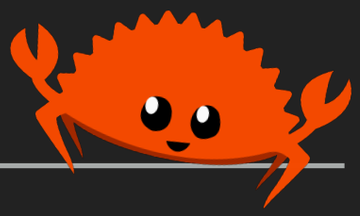




INTERRUPTS

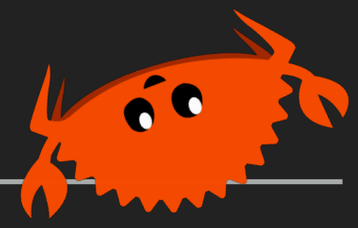
- Asynchronous events that change the execution flow of a program
- Can be from the outside (byte received, input line asserted, etc)
- Or inside (timer expired, software triggered IRQ)

- Push core registers onto a stack and jump to handler
- Basically this is preemptive multitasking baked into the hardware
- All multi-threaded fun is included
- Extremely fast and deterministic (couple 10s of instructions)



WHAT LANGUAGES CAN BE USED?

- C (of course), C++
- Assembler
- Rust
- Lisp
- Forth
- Lua
- Basic
- Pascal
- PL/M
- Surprisingly - Python
- Surprisingly - JavaScript



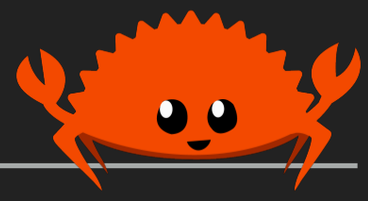
BAD EXAMPLES

```
/**@brief Function for the S110 SoftDevice initialization.
 *
 * @details This function initializes the S110 SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init(void)
{
    uint32_t err_code;

    // Initialize SoftDevice.
    SOFTDEVICE_HANDLER_INIT(NRF_CLOCK_LFCLKSRC_RC_250_PPM_TEMP_1000MS_CALIBRATION, NULL);

    // Enable BLE stack.
    ble_enable_params_t ble_enable_params;
    memset(&ble_enable_params, 0, sizeof(ble_enable_params));
#if (defined(S130) || defined(S132))
    ble_enable_params.gatts_enable_params.attr_tab_size   = BLE_GATTS_ATTR_TAB_SIZE_DEFAULT;
#endif
    ble_enable_params.gatts_enable_params.service_changed = IS_SRVC_CHANGED_CHARACT_PRESENT;
    err_code = sd_ble_enable(&ble_enable_params);
    APP_ERROR_CHECK(err_code);

    // Subscribe for BLE events.
    err_code = softdevice_ble_evt_handler_set(ble_evt_dispatch);
    APP_ERROR_CHECK(err_code);
}
```

```
#define ENABLE_GATE()
#define DISABLE_GATE()
#define DCCAL_ON()
#define DCCAL_OFF()
#define IS_DRV_FAULT()
```

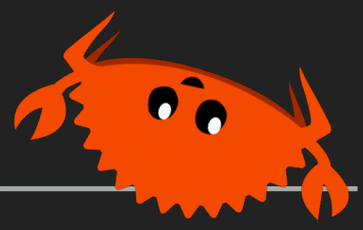
```
palSetPad(GPIOB, 5)
palClearPad(GPIOB, 5)
palSetPad(GPIOB, 1)
palClearPad(GPIOB, 1)
(!palReadPad(GPIOB, 4))
```

```
// ENABLE_GATE
palSetPadMode(GPIOB, 5,
        PAL_MODE_OUTPUT_PUSHPULL |
        PAL_STM32_OSPEED_HIGHEST);

ENABLE_GATE();
```



BLINKY IN RUST

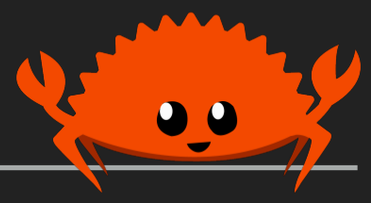
```rust
#![deny(warnings)]
#![deny(unsafe_code)]
#![no_main]
#![no_std]

extern crate cortex_m;
extern crate cortex_m_rt as rt;
extern crate panic_halt;
extern crate stm32g0xx_hal as hal;

use hal::prelude::*;
use hal::stm32;
use rt::entry;

#[entry]
fn main() -> ! {
    let dp = stm32::Peripherals::take().expect("cannot take peripherals");
    let mut rcc = dp.RCC.constrain();
    let gpioa = dp.GPIOA.split(&mut rcc);
    let mut led = gpioa.pa5.into_push_pull_output();

    loop {
        for _ in 0..1_000_000 {
            led.set_low().unwrap();
        }
        for _ in 0..1_000_000 {
            led.set_high().unwrap();
        }
    }
}
```



CLI PROCESSOR

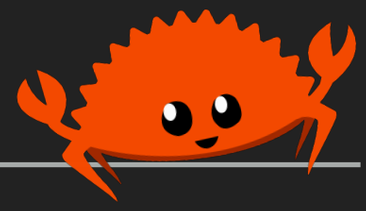
```rust
static mut ARRAY: [u8; 128] = [0u8; 128];
static mut IDX: usize = 0;
static mut LINE_READY: bool = false;
static G_USART0: Mutex<RefCell<Option<hal::usart::Usart>>> = Mutex::new(RefCell::new(None));

#[entry]
unsafe fn main() -> ! {
    let p = setup_peripherals();
    cortex_m::interrupt::free(|cs| *G_USART0.borrow(cs).borrow_mut() = Some(p.usart0));

    loop {
        if *LINE_READY {
            process_cli_input(&*ARRAY[0..*IDX], &mut p);
            *LINE_READY = false;
            *IDX = 0;
        }
        sleep(10.ms());
    }
}
```

```rust
#[interrupt]
unsafe fn USART0_IRQ_Handler() {
    static mut USART0: Option<hal::dma::Channels> = None;

    let usart0 = USART0.get_or_insert_with(|| {
        cortex_m::interrupt::free(|cs| {
            // Move usart here, leaving a None in its place
            G_USART0.borrow(cs).replace(None).unwrap()
        })
    });

    let b = get_byte(usart0);
    *ARRAY[*IDX] = b;
    *IDX += 1;
    if b == '\n' {
        *LINE_READY = true;
    }
}
```



MEET RTIC

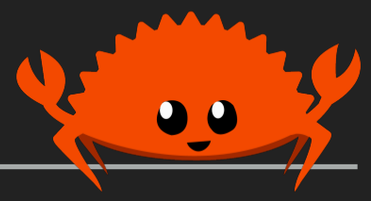
- Compile time analysis of task priorities to prevent dead locks and allow lock-free access to resources (some)
- Hardware based preemptive multitasking
- Supports software tasks and hardware tasks (interrupt handlers), message passing, task scheduling and spawning
- Highly efficient, only one stack for all tasks, no dynamic memory allocation
- All Cortex-M devices supported, RISC-V support is coming
- Amenable to worst case execution time analysis, though tooling for that is not yet ready



IMPROVED CLI PROCESSOR

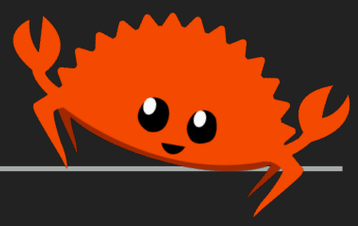
```rust
struct Resources {
    usart0: hal::usart::Usart0,
    cli_p: bbqueue::Producer<u8>,
    cli_c: bbqueue::Consumer<u8>,
}

#[task]
fn init(cx: init::Resources) -> LateResources {
    let p = setup_peripherals();
    let (cli_p, cli_c) = BBBuffer::new().split();
    LateResources {
        usart0: p.usart0,
        cli_p,
        cli_c
    }
}
```

```rust
#[task(resources = ["cli_c"])]
fn cli_processor(cx: cli_processor::Context, s: cli_processor::State) {
    while let Some(rgr) = cx.cli_c.read() {
        process_cli_input(rgr);
        rgr.release();
    }
}

#[task(binds = USART0, resources = ["usart0", "cli_p"])]
fn usart0(cx: usart0::Context) {
    let b = get_byte(cx.usart0);

    let wgr = get_or_create_write_grant(cx.cli_p);
    if b == '\n' {
        wgr.commit();
    } else {
        wgr.append(b);
    }

    cx.spawn.cli_processor();
}
```



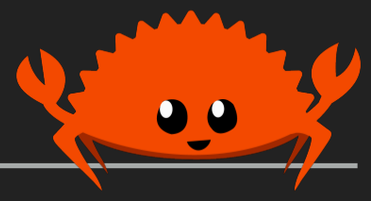
MORE ABOUT RTIC

- Multi core devices is supported, compilation from one source tree for different cores and even different architectures (Cortex-M3 and Cortex-M0 for ex.)
- Tasks support additional event argument that can be used to pass messages. Tasks can schedule themselves, so repeated processes can be created easily.
- static mut variables inside tasks are replaced &mut and guaranteed to be safe to use without runtime overhead
- Task state can be easily stored in a separate struct passed to it as &mut reference



HOT REINIT GPIO->PWM->GPIO

```rust
fn switch_mode_command(bp: &mut BoardPeripherals, args: Args) {
    let cmd = some_or_return!(args.next(), "swmode manual/openloop");
    match cmd {
        "manual" => {
            match bp.switches {
                Some(_) => {
                    rprintln!("Already in manual");
                },
                None => {
                    rprintln!("Switching to manual");
                    let openloop = bp.openloop.take().unwrap();
                    let switches = openloop.deinit();
                    bp.switches = Some(switches)
                }
            }
        }
        "openloop" => {
            match bp.openloop {
                Some(_) => {
                    rprintln!("Already in openloop");
                }
                None => {
                    rprintln!("Switching to openloop");
                    let switches = bp.switches.take().unwrap();
                    let openloop = crate::openloop::OpenLoop::init(bp.clocks.sysclk(), switches);
                    bp.openloop = Some(openloop);
                }
            }
        }
        _ => unknown_command!(cmd)
    }
    command_executed!()
}

```



FLASH WRITE

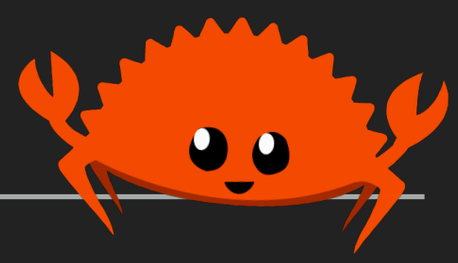
```rust
fn main() -> ! {
    let p = setup_peripherals();
    let mut flash_config = FlashWriter::new(0x0800_7000..(0x0800_7000 + 1u32.kb())).unwrap();
    let b_cfg: BootLoaderConfig = flash_read::<BootLoaderConfig>(flash_config.get_start_address());
    match flash_config.erase(p.flash_regs.borrow_mut()) {
        Ok(_) => {}
        Err(e) => {
            panic!("Flash_config erase failed!")
        }
    }
    match flash_config.write(
        p.flash_regs.borrow_mut(),
        &[BootLoaderConfig {
            can_id: 0x1234567,
            self_crc: 0,
            image_start_address: 0x0800_3000,
            image_crc: 0,
        }],
    ) {
        Ok(_) => {}
        Err(e) => {
            panic!("Flash_config write new config failed!")
        }
    }
    match flash_config.flush(p.flash_regs.borrow_mut()) {
        Ok(_) => {}
        Err(e) => {
            panic!("Flash_config flush failed!")
        }
    }
    let b_cfg_slice: &[BootLoaderConfig] =
        flash_read_slice::<BootLoaderConfig>(flash_config.get_start_address(), 1);
}
```

```rust
#[derive(Debug)]
struct BootLoaderConfig {
    can_id: u32,
    self_crc: u16,
    image_start_address: u32,
    image_crc: u16,
}
```

PIECES FROM WHEEL.ME CODE

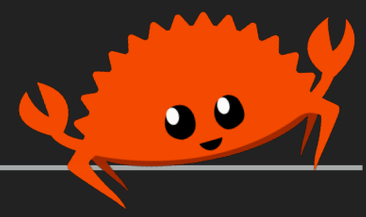
```rust
struct Resources {
    clocks: hal::rcc::Clocks,
    rcc: hal::rcc::Rcc,
    watchdog: IndependedWatchdog,
    bms_state: tasks::bms::BmsState,
    afe_io: tasks::bms::AfeIo,
    adc: Adc<hal::adc::Ready>,
    status_led: PB2<Output<PushPull>>,
    rtt: NonBlockingOutput,
    mcp25625_state: tasks::canbus::Mcp25625State,
    // mcp25625_irq: config::Mcp25625Irq,
    can_tx: config::CanTX,
    can_rx: config::CanRX,
    i2c: config::InternalI2c,
    bq76920: config::BQ769x0,
    // power_blocks: config::PowerBlocksMap,
    exti: Exti,
    button: config::ButtonPin,
    button_state: tasks::button::ButtonState,
    charge_indicator: tasks::led::ChargeIndicator,
    buzzer_pwm_channel: tasks::beeper::BuzzerPwmChannel,
}
```

```rust
struct Resources {
    watchdog: IndependentWatchdog,
    clocks: Clocks,
    delay: DelayCM,
    rtt_down_channel: rtt_target::DownChannel,

    can: can::Can,
    lidar_tx: LidarTx,
    lidar_rx: LidarRx,
    usart2_dma_rcx: Usart2DmaRxContext,
    usart1_dma_rcx: Usart1DmaRxContext,

    imu: Imu,
    imu_prod: ImuRxBufferP,

    lidar_controller: LidarController<'static, LidarCapacity, FRAME_CAPACITY>,
    lidar_heartbeat: HeartBeat<HEARTHBEAT_CHECKS>,
    lidar_consumer: LidarRxConsumer<LidarCapacity>,

    can_transmit_heap: config::can::CanTransmitHeap,
    can_receive_heap: config::can::CanReceiveHeap,

    uavcan_transfer_id: u8,

    lidar_state: state::LidarState,
    imu_state: state::ImuState,

    #[cfg(feature = "carrier")]
    power_gates: peripherals::PowerGates,
    #[cfg(feature = "carrier")]
    adc: ADC,
    mux_serial_dma_rcx: MuxDmaRxContext,
    usart_mux: peripherals::UsartMuxPB9PB12,
    can0_analyzer: canbus::CanAnalyzer,
    mux_serial_tx: MuxSerialTx,
    usart3_tx_producer: Usart3DmaTxProducer,
    usart3_dma_tx_context: Usart3DmaTxContext,
    leds: Leds,
    sop_and_control: SopAndControl,
```



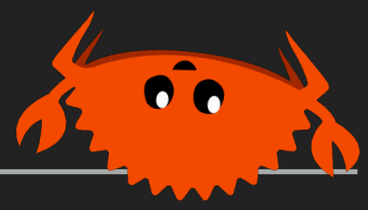
UNATTAINABLE SUPERIORITY

- One call - println!
- Embedded heavily relies on communication with external world for debug, logging and basic interaction
- Each println call on the device consumes space and valuable processor time
- With the help of procedural macros all the calls can be turned into extremely lightweight ones
- This idea can be extended to many other things, like peripheral configuration, error counters and other



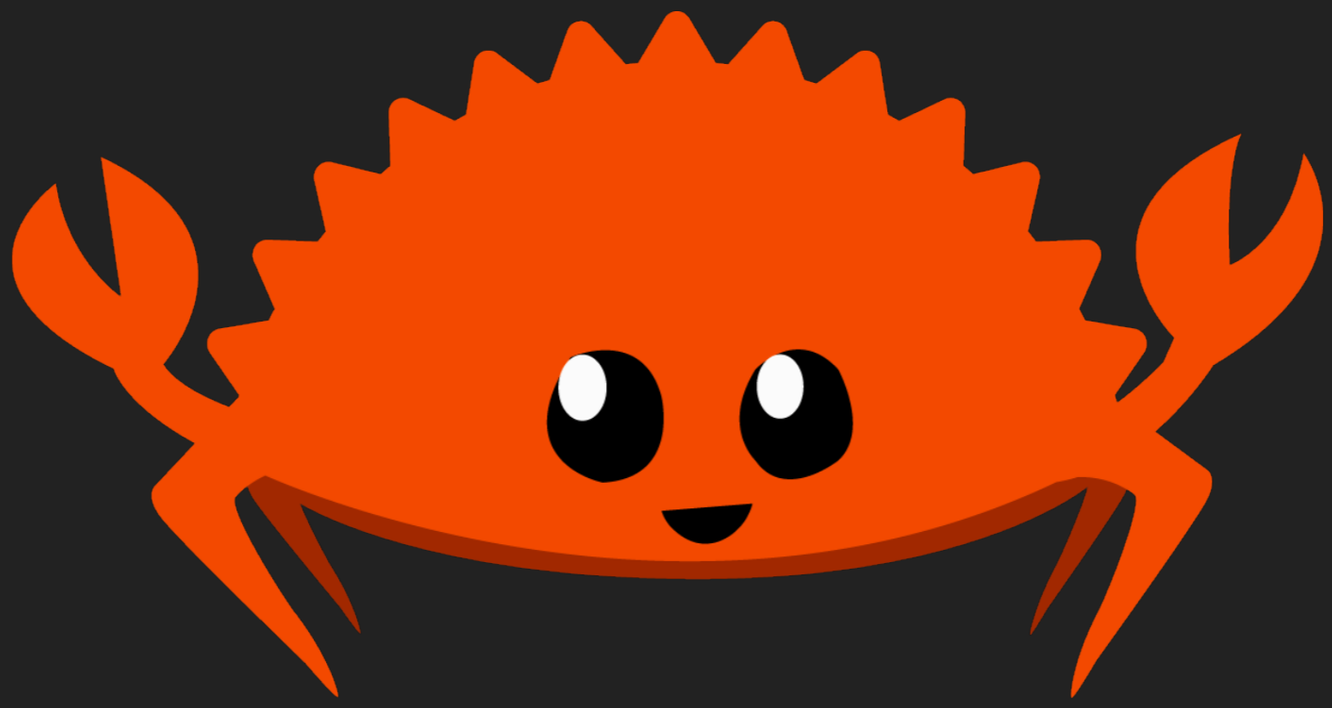
WHAT'S UP WITH RISC-V

- Risc-V is a modern, open and clean machine instruction set architecture (ISA)
- Development started in 2010 at Berkeley by prof. Krste Asanović
- There are support in LLVM
- Probably the next big thing like ARM was, especially with acquisition of the latter
- There are already microcontrollers available, that you can even buy and try (from SiFive)



PAIN POINTS IN EMBEDDED RUST

- Not enough hardware abstraction layers (HALs)
- IDE support is great, but some hardware related things are missing
- Learning material is quickly outdated, not enough people
- Vast amounts of auto generated code from chip manufacturer provided files (SVD) which is sometimes of bad quality and inconsistent
- Ecosystem of HALs is not standardised, how to do so is up to a great debate…
- Some rusty people do not want to abandon C and learn Rust



THANKS FOR ATTENTION!