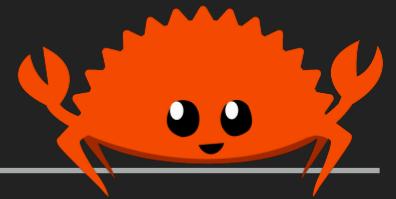


# EMBEDDED RUST

---

INTRODUCTION AND  
PRACTICAL APPLICATIONS



# WHAT IS EMBEDDED

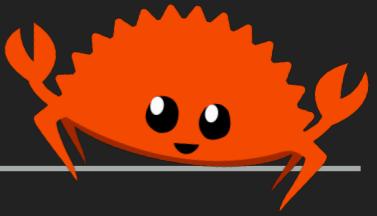
- ▶ No operating system environment, [no\_std]
- ▶ Generally no memory allocator
- ▶ Constraints on program size, ram size, speed
- ▶ Direct access to pretty much everything (core, I/O, peripherals)
- ▶ Embedded as in a hardware device, WebAssembly or other?  
applications are out of scope



# WHAT IS A MCU?

- ▶ Computer on a chip
- ▶ Contains everything needed to execute a program and interact with outer world
- ▶ Size from several mm<sup>2</sup>
- ▶ Often requires only one voltage rail to operate
- ▶ Most contain “debugger” interface (SWD, JTAG) that can be used to read/write memory and state.



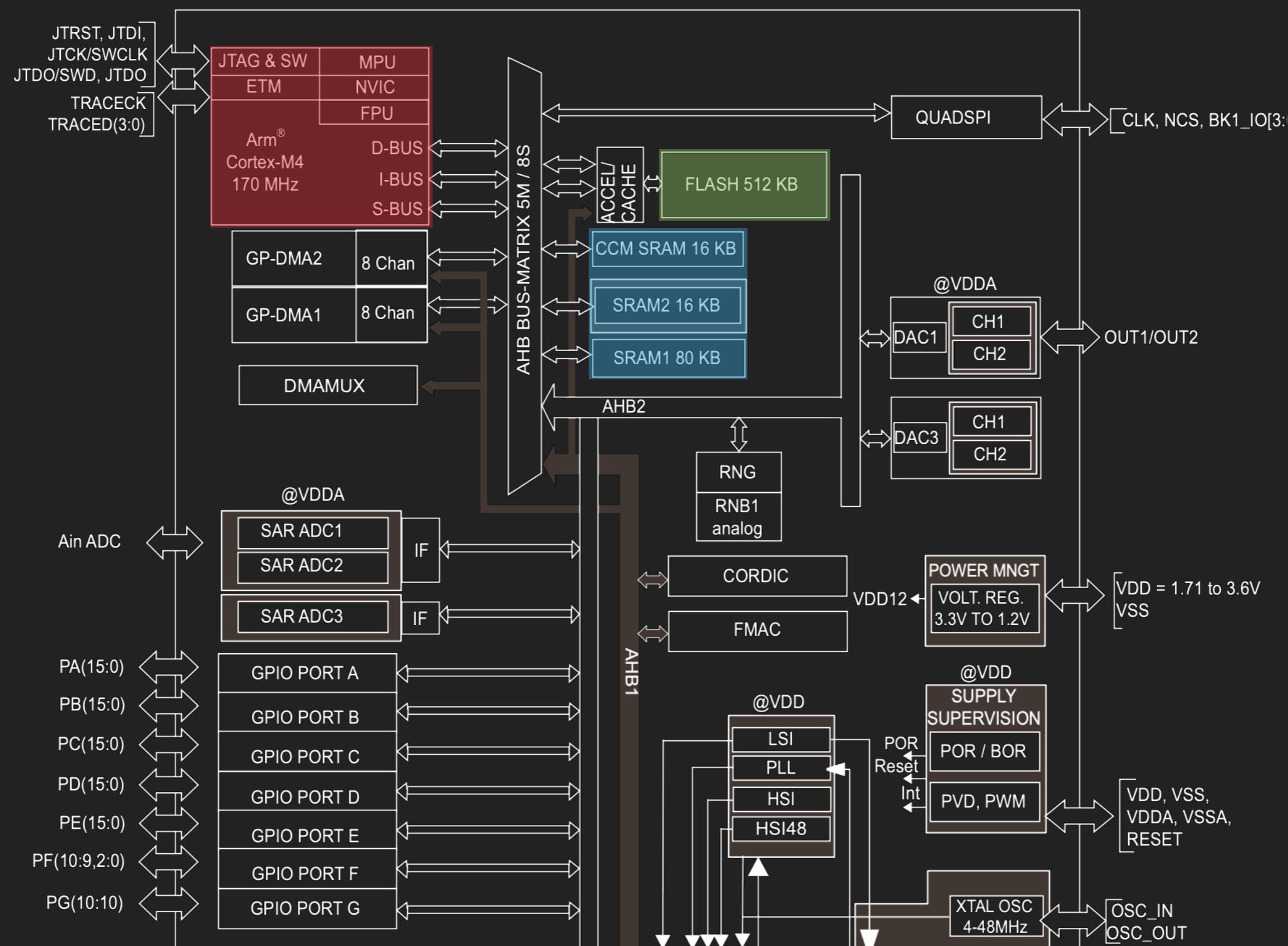


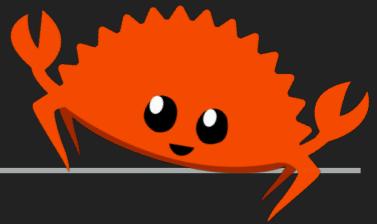
## WHERE MICROCONTROLLERS ARE USED?

- ▶ 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).

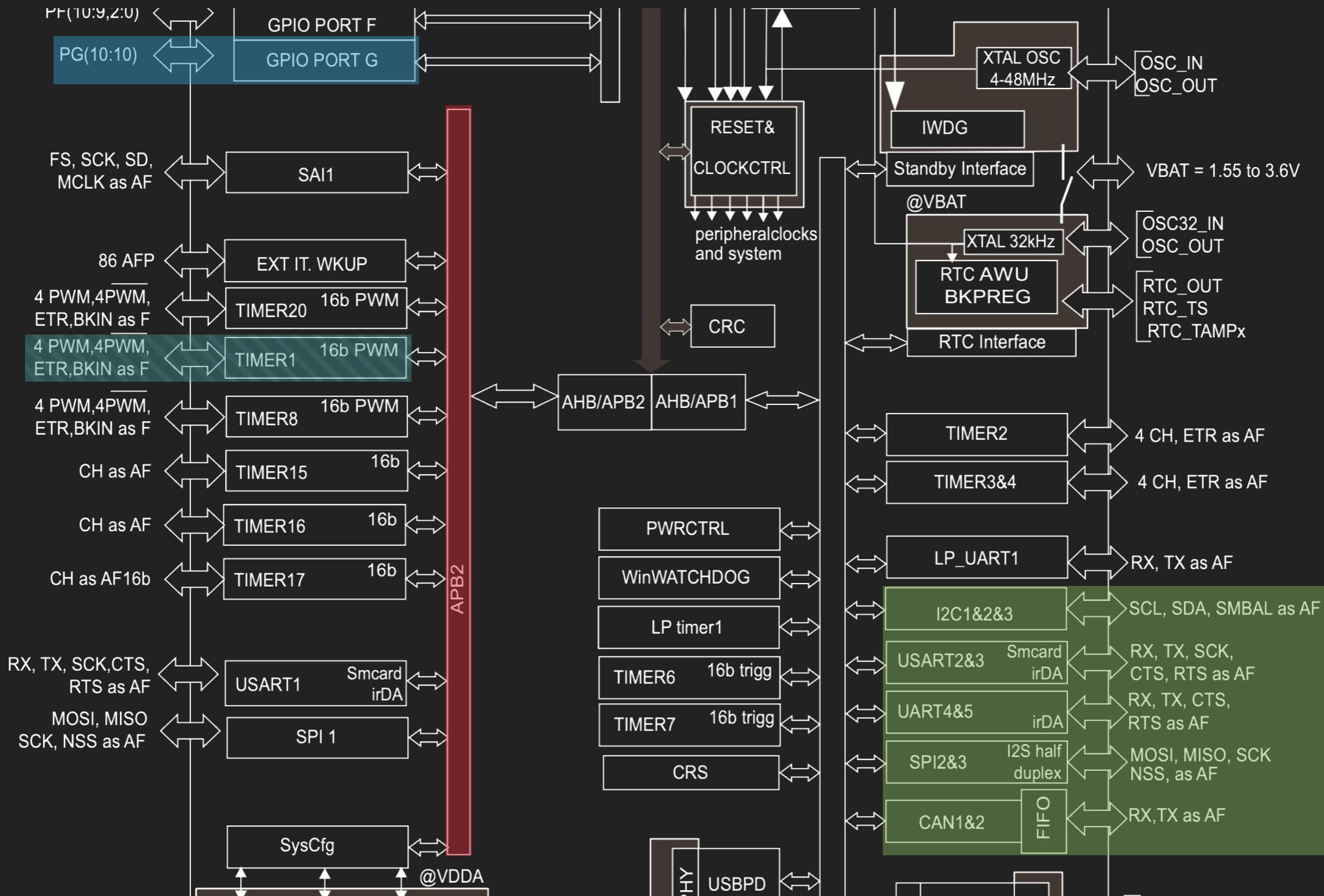


# HOW PROGRAM IS STORED AND EXECUTED?





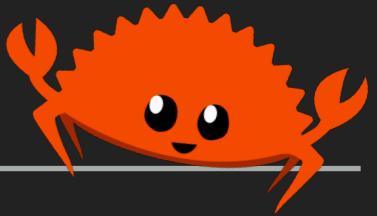
# HOW TO TALK TO OUTSIDE WORLD





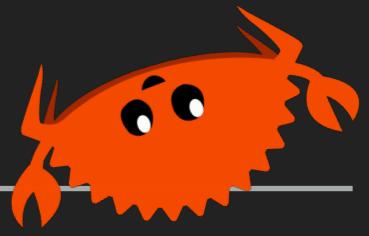
# INTERRUPTS

- ▶ 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)



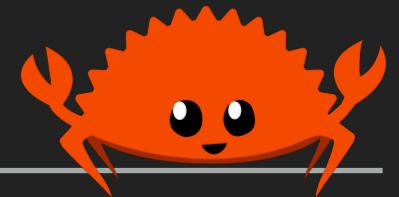
# WHAT LANGUAGES CAN BE USED?

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



# BAD EXAMPLES

```
366  /**@brief Function for the S110 SoftDevice initialization.  
367  *  
368  * @details This function initializes the S110 SoftDevice and the BLE event interrupt.  
369  */  
370  static void ble_stack_init(void)  
371  {  
372      uint32_t err_code;  
373  
374      // Initialize SoftDevice.  
375      SOFTDEVICE_HANDLER_INIT(NRF_CLOCK_LFCLKSRC_RC_250_PPM_TEMP_1000MS_CALIBRATION, NULL);  
376  
377      // Enable BLE stack.  
378      ble_enable_params_t ble_enable_params;  
379      memset(&ble_enable_params, 0, sizeof(ble_enable_params));  
380 #if (defined(S130) || defined(S132))  
381     ble_enable_params.gatts_enable_params.attr_tab_size = BLE_GATTS_ATTR_TAB_SIZE_DEFAULT;  
382 #endif  
383     ble_enable_params.gatts_enable_params.service_changed = IS_SRVC_CHANGED_CHARACT_PRESENT;  
384     err_code = sd_ble_enable(&ble_enable_params);  
385     APP_ERROR_CHECK(err_code);  
386  
387      // Subscribe for BLE events.  
388      err_code = softdevice_ble_evt_handler_set(ble_evt_dispatch);  
389      APP_ERROR_CHECK(err_code);  
390  }  
391  
392  //  
393  //  
394  //  
395  //  
396  //  
397  //  
398  //  
399  //  
400  //  
401  //  
402  //  
403  //  
404  //  
405  //  
406  //  
407  //  
408  //  
409  //  
410  //  
411  //  
412  //  
413  //  
414  //  
415  //  
416  //  
417  //  
418  //  
419  //  
420  //  
421  //  
422  //  
423  //  
424  //  
425  //  
426  //  
427  //  
428  //  
429  //  
430  //  
431  //  
432  //  
433  //  
434  //  
435  //  
436  //  
437  //  
438  //  
439  //  
440  //  
441  //  
442  //  
443  //  
444  //  
445  //  
446  //  
447  //  
448  //  
449  //  
450  //  
451  //  
452  //  
453  //  
454  //  
455  //  
456  //  
457  //  
458  //  
459  //  
460  //  
461  //  
462  //  
463  //  
464  //  
465  //  
466  //  
467  //  
468  //  
469  //  
470  //  
471  //  
472  //  
473  //  
474  //  
475  //  
476  //  
477  //  
478  //  
479  //  
480  //  
481  //  
482  //  
483  //  
484  //  
485  //  
486  //  
487  //  
488  //  
489  //  
490  //  
491  //  
492  //  
493  //  
494  //  
495  //  
496  //  
497  //  
498  //  
499  //  
500  //  
501  //  
502  //  
503  //  
504  //  
505  //  
506  //  
507  //  
508  //  
509  //  
510  //  
511  //  
512  //  
513  //  
514  //  
515  //  
516  //  
517  //  
518  //  
519  //  
520  //  
521  //  
522  //  
523  //  
524  //  
525  //  
526  //  
527  //  
528  //  
529  //  
530  //  
531  //  
532  //  
533  //  
534  //  
535  //  
536  //  
537  //  
538  //  
539  //  
540  //  
541  //  
542  //  
543  //  
544  //  
545  //  
546  //  
547  //  
548  //  
549  //  
550  //  
551  //  
552  //  
553  //  
554  //  
555  //  
556  //  
557  //  
558  //  
559  //  
560  //  
561  //  
562  //  
563  //  
564  //  
565  //  
566  //  
567  //  
568  //  
569  //  
570  //  
571  //  
572  //  
573  //  
574  //  
575  //  
576  //  
577  //  
578  //  
579  //  
580  //  
581  //  
582  //  
583  //  
584  //  
585  //  
586  //  
587  //  
588  //  
589  //  
590  //  
591  //  
592  //  
593  //  
594  //  
595  //  
596  //  
597  //  
598  //  
599  //  
600  //  
601  //  
602  //  
603  //  
604  //  
605  //  
606  //  
607  //  
608  //  
609  //  
610  //  
611  //  
612  //  
613  //  
614  //  
615  //  
616  //  
617  //  
618  //  
619  //  
620  //  
621  //  
622  //  
623  //  
624  //  
625  //  
626  //  
627  //  
628  //  
629  //  
630  //  
631  //  
632  //  
633  //  
634  //  
635  //  
636  //  
637  //  
638  //  
639  //  
640  //  
641  //  
642  //  
643  //  
644  //  
645  //  
646  //  
647  //  
648  //  
649  //  
650  //  
651  //  
652  //  
653  //  
654  //  
655  //  
656  //  
657  //  
658  //  
659  //  
660  //  
661  //  
662  //  
663  //  
664  //  
665  //  
666  //  
667  //  
668  //  
669  //  
670  //  
671  //  
672  //  
673  //  
674  //  
675  //  
676  //  
677  //  
678  //  
679  //  
680  //  
681  //  
682  //  
683  //  
684  //  
685  //  
686  //  
687  //  
688  //  
689  //  
690  //  
691  //  
692  //  
693  //  
694  //  
695  //  
696  //  
697  //  
698  //  
699  //  
700  //  
701  //  
702  //  
703  //  
704  //  
705  //  
706  //  
707  //  
708  //  
709  //  
710  //  
711  //  
712  //  
713  //  
714  //  
715  //  
716  //  
717  //  
718  //  
719  //  
720  //  
721  //  
722  //  
723  //  
724  //  
725  //  
726  //  
727  //  
728  //  
729  //  
730  //  
731  //  
732  //  
733  //  
734  //  
735  //  
736  //  
737  //  
738  //  
739  //  
740  //  
741  //  
742  //  
743  //  
744  //  
745  //  
746  //  
747  //  
748  //  
749  //  
750  //  
751  //  
752  //  
753  //  
754  //  
755  //  
756  //  
757  //  
758  //  
759  //  
760  //  
761  //  
762  //  
763  //  
764  //  
765  //  
766  //  
767  //  
768  //  
769  //  
770  //  
771  //  
772  //  
773  //  
774  //  
775  //  
776  //  
777  //  
778  //  
779  //  
780  //  
781  //  
782  //  
783  //  
784  //  
785  //  
786  //  
787  //  
788  //  
789  //  
790  //  
791  //  
792  //  
793  //  
794  //  
795  //  
796  //  
797  //  
798  //  
799  //  
800  //  
801  //  
802  //  
803  //  
804  //  
805  //  
806  //  
807  //  
808  //  
809  //  
810  //  
811  //  
812  //  
813  //  
814  //  
815  //  
816  //  
817  //  
818  //  
819  //  
820  //  
821  //  
822  //  
823  //  
824  //  
825  //  
826  //  
827  //  
828  //  
829  //  
830  //  
831  //  
832  //  
833  //  
834  //  
835  //  
836  //  
837  //  
838  //  
839  //  
840  //  
841  //  
842  //  
843  //  
844  //  
845  //  
846  //  
847  //  
848  //  
849  //  
850  //  
851  //  
852  //  
853  //  
854  //  
855  //  
856  //  
857  //  
858  //  
859  //  
860  //  
861  //  
862  //  
863  //  
864  //  
865  //  
866  //  
867  //  
868  //  
869  //  
870  //  
871  //  
872  //  
873  //  
874  //  
875  //  
876  //  
877  //  
878  //  
879  //  
880  //  
881  //  
882  //  
883  //  
884  //  
885  //  
886  //  
887  //  
888  //  
889  //  
890  //  
891  //  
892  //  
893  //  
894  //  
895  //  
896  //  
897  //  
898  //  
899  //  
900  //  
901  //  
902  //  
903  //  
904  //  
905  //  
906  //  
907  //  
908  //  
909  //  
910  //  
911  //  
912  //  
913  //  
914  //  
915  //  
916  //  
917  //  
918  //  
919  //  
920  //  
921  //  
922  //  
923  //  
924  //  
925  //  
926  //  
927  //  
928  //  
929  //  
930  //  
931  //  
932  //  
933  //  
934  //  
935  //  
936  //  
937  //  
938  //  
939  //  
940  //  
941  //  
942  //  
943  //  
944  //  
945  //  
946  //  
947  //  
948  //  
949  //  
950  //  
951  //  
952  //  
953  //  
954  //  
955  //  
956  //  
957  //  
958  //  
959  //  
960  //  
961  //  
962  //  
963  //  
964  //  
965  //  
966  //  
967  //  
968  //  
969  //  
970  //  
971  //  
972  //  
973  //  
974  //  
975  //  
976  //  
977  //  
978  //  
979  //  
980  //  
981  //  
982  //  
983  //  
984  //  
985  //  
986  //  
987  //  
988  //  
989  //  
990  //  
991  //  
992  //  
993  //  
994  //  
995  //  
996  //  
997  //  
998  //  
999  //  
1000 //
```



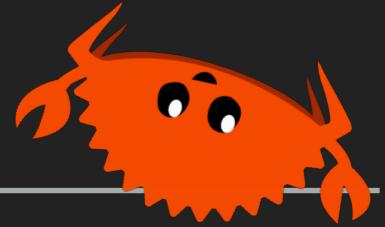
# BLINKY IN RUST

```
1  #![deny(warnings)]
2  #![deny(unsafe_code)]
3  #![no_main]
4  #![no_std]
5
6  extern crate cortex_m;
7  extern crate cortex_m_rt as rt;
8  extern crate panic_halt;
9  extern crate stm32g0xx_hal as hal;
10
11 use hal::prelude::*;

12 use hal::stm32;
13 use rt::entry;

14
15 #[entry]
16 fn main() -> ! {
17     let dp = stm32::Peripherals::take().expect("cannot take peripherals");
18     let mut rcc = dp.RCC.constrain();
19     let gpioa = dp.GPIOA.split(&mut rcc);
20     let mut led = gpioa.pa5.into_push_pull_output();

21
22     loop {
23         for _ in 0..1_000_000 {
24                 led.set_low().unwrap();
25         }
26         for _ in 0..1_000_000 {
27                 led.set_high().unwrap();
28         }
29     }
30 }
```



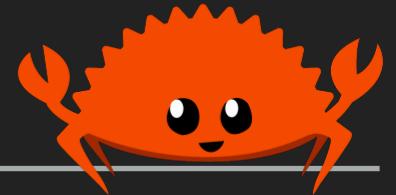
# CLI PROCESSOR

```

1 static mut ARRAY: [u8; 128] = [0u8; 128];
2 static mut IDX: usize = 0;
3 static mut LINE_READY: bool = false;
4 static G_USART0: Mutex<RefCell<Option<hal::usart::Usart>>> = Mutex::new(RefCell::new(None));
5
6 #[entry]
7 unsafe fn main() -> ! {
8     let p = setup_peripherals();
9     cortex_m::interrupt::free(|cs| *G_USART0.borrow(cs).borrow_mut() = Some(p.usart0));
10
11    loop {
12        if *LINE_READY {
13            process_cli_input(&*ARRAY[0..*IDX], &mut p);
14            *LINE_READY = false;
15            *IDX = 0;
16        }
17        sleep(10.ms());
18    }
19 }

1 #[interrupt]
2 unsafe fn USART0_IRQ_Handler() {
3     static mut USART0: Option<hal::dma::Channels> = None;
4
5     let usart0 = USART0.get_or_insert_with(|| {
6         cortex_m::interrupt::free(|cs| {
7             // Move usart here, leaving a None in its place
8             G_USART0.borrow(cs).replace(None).unwrap()
9         })
10    });
11
12    let b = get_byte(usart0);
13    *ARRAY[*IDX] = b;
14    *IDX += 1;
15    if b == '\n' {
16        *LINE_READY = true;
17    }
18 }

```



## MEET RTIC

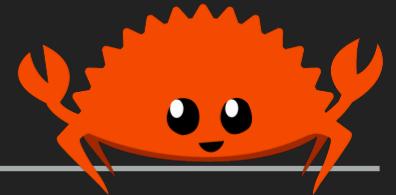
- ▶ Compile time analysis of task priorities to prevent dead locks and allow lock-free access to resources (some)
- ▶ Hardware based preemptive multitasking
- ▶ Supports software tasks and hardware tasks (interrupt handlers), message passing, task scheduling and spawning
- ▶ Highly efficient, only one 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

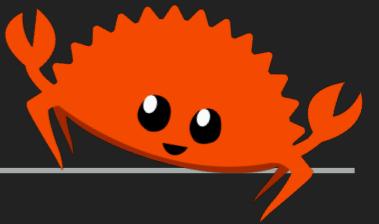
```
1 struct Resources {
2     usart0: hal::usart::Usart0,
3     cli_p: bbqueue::Producer<u8>,
4     cli_c: bbqueue::Consumer<u8>,
5 }
6
7 #[task]
8 fn init(cx: init::Resources) -> LateResources {
9     let p = setup_peripherals();
10    let (cli_p, cli_c) = BBBuffer::new().split();
11    LateResources {
12        usart0: p.usart0,
13        cli_p,
14        cli_c
15    }
16 }
```

```
1  #[task(resources = ["cli_c"])]
2  fn cli_processor(cx: cli_processor::Context, s: cli_processor::State) {
3      while let Some(rgr) = cx.cli_c.read() {
4          process_cli_input(rgr);
5          rgr.release();
6      }
7  }
8
9  #[task(binds = USART0, resources = ["usart0", "cli_p"])]
10 fn usart0(cx: usart0::Context) {
11     let b = get_byte(cx.usart0);
12
13     let wgr = get_or_create_write_grant(cx.cli_p);
14     if b == '\n' {
15         wgr.commit();
16     } else {
17         wgr.append(b);
18     }
19
20     cx.spawn.cli_processor();
21 }
```



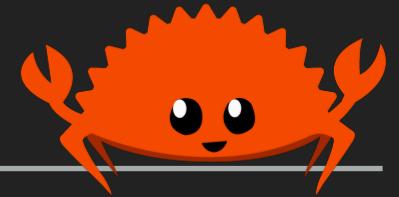
## MORE ABOUT RTIC

- ▶ 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

```
1 fn switch_mode_command(bp: &mut BoardPeripherals, args: Args) {
2     let cmd = some_or_return!(args.next(), "swmode manual/openloop");
3     match cmd {
4         "manual" => {
5             match bp.switches {
6                 Some(_) => {
7                     rprintln!("Already in manual");
8                 },
9                 None => {
10                     rprintln!("Switching to manual");
11                     let openloop = bp.openloop.take().unwrap();
12                     let switches = openloop.deinit();
13                     bp.switches = Some(switches)
14                 }
15             }
16         }
17         "openloop" => {
18             match bp.openloop {
19                 Some(_) => {
20                     rprintln!("Already in openloop");
21                 },
22                 None => {
23                     rprintln!("Switching to openloop");
24                     let switches = bp.switches.take().unwrap();
25                     let openloop = crate::openloop::OpenLoop::init(bp.clocks.sysclk(), switches);
26                     bp.openloop = Some(openloop);
27                 }
28             }
29         }
30         _ => unknown_command!(cmd)
31     }
32     command_executed!()
33 }
34 }
```



# FLASH WRITE

```

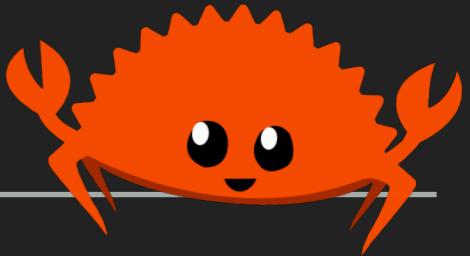
1 fn main() -> ! {
2     let p = setup_peripherals();
3     let mut flash_config = FlashWriter::new(0x0800_7000..(0x0800_7000 + 1u32.kb())).unwrap();
4     let b_cfg: BootLoaderConfig = flash_read::<BootLoaderConfig>(flash_config.get_start_address());
5     match flash_config.erase(p.flash_regs.borrow_mut()) {
6         Ok(_) => {}
7         Err(e) => {
8             panic!("Flash_config erase failed!")
9         }
10    }
11    match flash_config.write(
12        p.flash_regs.borrow_mut(),
13        &[BootLoaderConfig {
14            can_id: 0x1234567,
15            self_crc: 0,
16            image_start_address: 0x0800_3000,
17            image_crc: 0,
18        }],
19    ) {
20        Ok(_) => {}
21        Err(e) => {
22            panic!("Flash_config write new config failed!")
23        }
24    }
25    match flash_config.flush(p.flash_regs.borrow_mut()) {
26        Ok(_) => {}
27        Err(e) => {
28            panic!("Flash_config flush failed!")
29        }
30    }
31    let b_cfg_slice: &[BootLoaderConfig] =
32        flash_read_slice::<BootLoaderConfig>(flash_config.get_start_address(), 1);
}

```

# PIECES FROM WHEEL.ME CODE

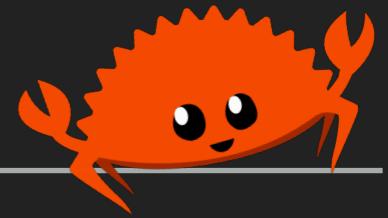
```
struct Resources {
    clocks: hal::rcc::Clocks,
    rcc: hal::rcc::Rcc,
    watchdog: IndependendedWatchdog,
    bms_state: tasks::bms::BmsState,
    afe_io: tasks::bms::AfeIo,
    adc: Adc<hal::adc::Ready>,
    status_led: PB2<Output<PushPull>>,
    rtt: NonBlockingOutput,
    mcp25625_state: tasks::canbus::Mcp25625State,
    // 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,
}
```

```
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<HEARTBEAT_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



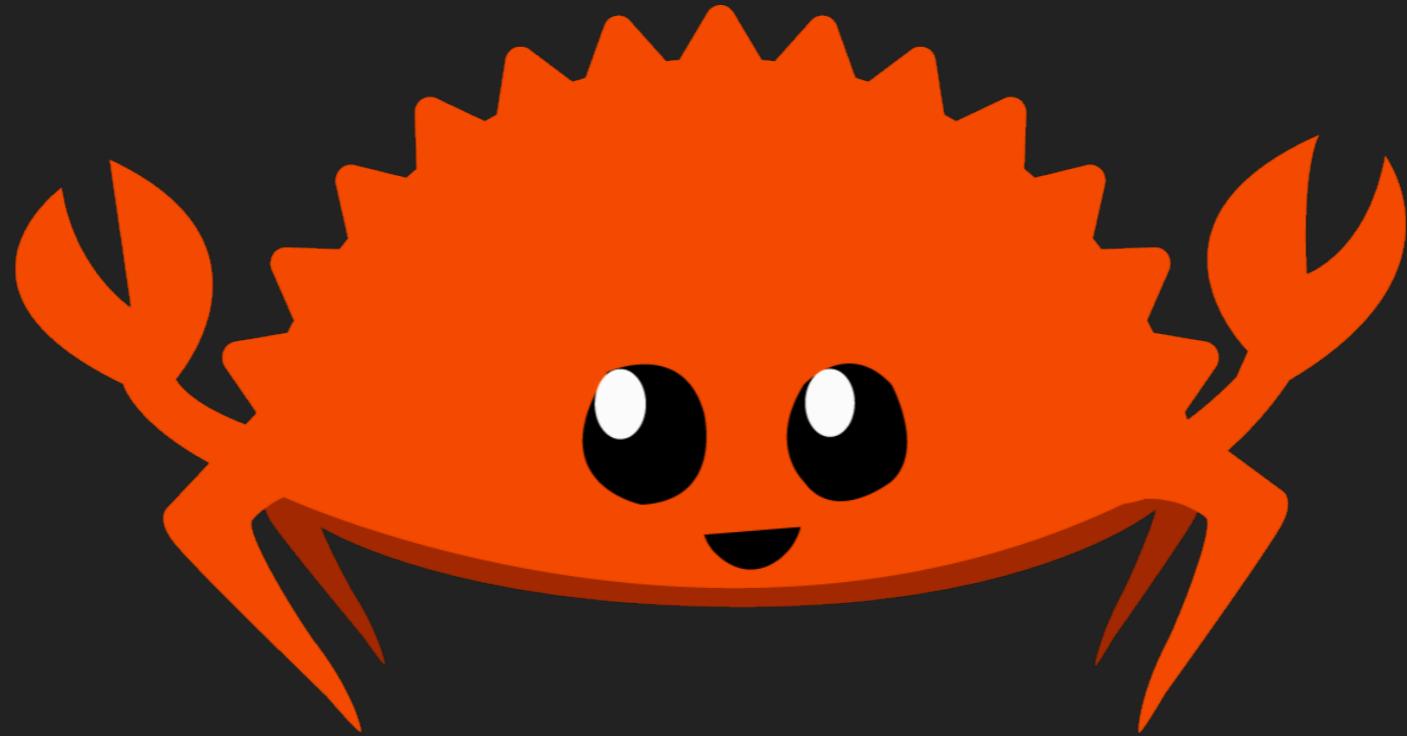
## WHAT'S UP WITH RISC-V

- ▶ 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!