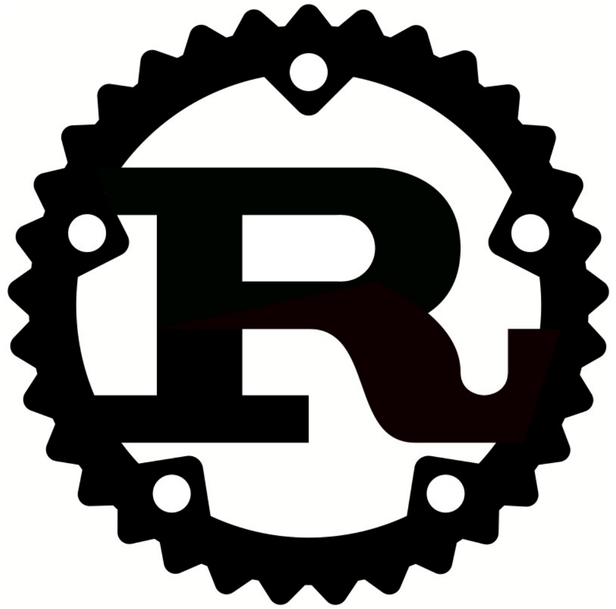
Rust Workshop

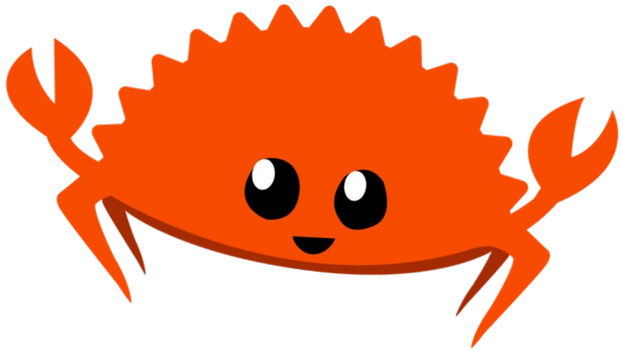


Safe. Fast. Concurrent.
Pick three!

Bitraf, 11. oktober 2023 kl 1800

Mål for workshopen

- Utveksle informasjon om språket Rust og bruken av det
- Bidra til å etablere et miljø for Rust-programmering rundt Bitraf, kanskje primært med vekt på mikrokontrollere
- Bistå med oppsett av toolchain for programutvikling i Rust under Visual Studio Code, både for operativsystem og for embedded



Agenda

- Kort om hva som skiller Rust fra andre programmeringsspråk (jeg kjenner bare noen språk, så her må andre også bidra!)
- Jens vil fortelle litt om Raspberry Pi Pico kontra andre mikrokontrollere (spesielt ESP32)
- Torfinn (og evt. andre) forteller litt om bruk av Rust i komersiell sammenheng
- Hva inngår akkurat nå i mest moderne toolchain for Rust for embedded? Oppskriften som gjennomgås virker for MacOS og Linux, med Picoprobe eller ST-Link V2/V2.1 mot Raspberry Pi Pico, annen ARM microcontroller eller Risc V
- Helt enkle kode-eksempler som illustrerer Rust og step-by-step debugging
- Bistand med oppsett av toolchain for Rust/åpen post

Some cool features in Rust

- Noe av det mest magiske med Rust er at kompilatoren selv legger inn kode som frigir minnet etter data som går ut av scope. Altså ingen *Garbage Collector* og man skriver heller ikke *destructors*
- Objektorientert programmering (i praksis uten arv) støttes med *struct* og *impl*
- Funksjonell programmering støttes bl.a. gjennom *iterators*
- Typesystemet er veldig kraftig og legger til grunn *strict typing*. Dette kommer godt til uttrykk når man f.eks. bruker *std::Collections* crate på egne typer
- Kodeforgrening kan gjøres med *Match*, som lagt overgår *switch/case* i C/C++/C#
- Interfaces defineres og anvendes med *traits*. Polymorfisme/*dynamic dispatch* defineres med *trait objects*
- *Zero Cost Abstractions* er stikkord for veldig mange språklige mekanismer i Rust
- Kompilatoren gir veldig gode feilmeldinger, og programmer som kompilerer virker veldig ofte
- Enhetstesting under utvikling støttes godt i Rusts integrerte test-system
- Pakke- og byggesystemet *cargo* sørger for nedlastning, kompilering og lenking av korrekte versjoner av anvendte crates. Krysskompilering til annen plattform er så enkelt som å spesifisere *target*

Hva sliter begynnere med i Rust?

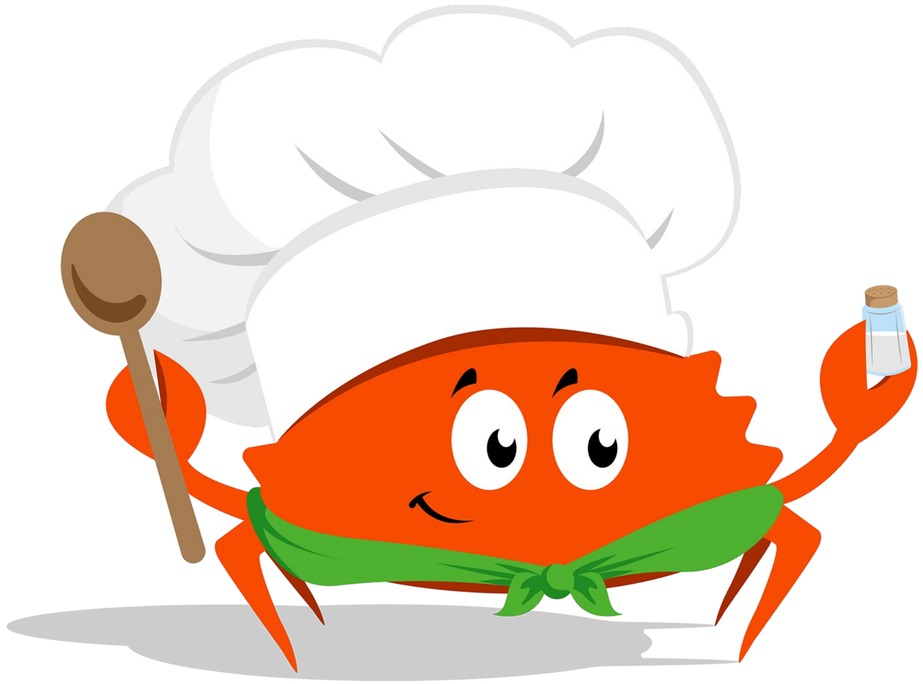
- **Borrow Checker** Rust innfører et eierskapsbegrep som er ulikt alle andre språk. For begynnere er det vanskelig å innse at «helt vanlig kode» innebærer at eierskapet til data flyttes slik at data «plutselig blir utilgjengelig»

- **Mut keyword** Data er i utgangspunktet immutable (uforanderlig) i Rust. Slik er det ikke i de fleste andre språk og det blir gjerne til at begynnere glemmer å spesifisere *mut*

- **Semicolon** Avhengig av hvilket språk man kommer fra, kan begynnere ha vanskelig for å huske at *statements* avsluttes med semikolon (og at *expressions* returnerer uten semicolon).

Hvordan blir Rust-kode annerledes for mikrokontrollere?

- Støttede mikrokontrollere:
  - ARM
  - Risc V
  - ESP32 (proprietær støtte)
- Viktige stikkord:
  - #![no_std]
  - #![no_main]
  - Spesifisering av *panic handling*
- Utnytting av *peripherals* (maskinvarens enheter med funksjonelle egenskaper)
- Ofte må man velge en måte for håndtering av interrupts:
  - Proprietært (ARM har *utviklervennlig* NVIC)
  - RTIC (Real Time Interrupt-driven Concurrency)
  - Embassy (som jeg egentlig ikke kjenner)



Best embedded toolchain right now!

- Standard Visual Studio Code-installasjon

  code.visualstudio.com/download
- Standard Rust-installasjon

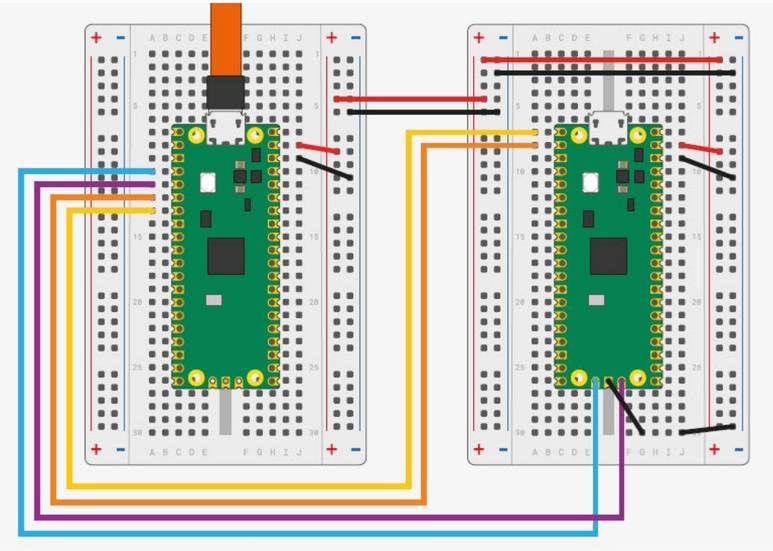
  rust-lang.org/tools/install

  For Linux og MacOS er det en CURL som kjøres i terminal
  curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
- Utvidelser i Visual Studio Code
- Programmeringsprobe
  (Picoprobe eller ST-Link) og *probe-rs*

Nødvendige pluss anbefalte utvidelser til Visual Studio Code

- *Rust-analyzer* Rust-språkstøtte for VS Code
- *CodeLLDB Debugger* (fra maskinkode) for Rust, C++ og andre kompilerte programmeringsspråk
- *Debugger for probe-rs* Debug adapter for VS Code
- *nRF Terminal* En seriell terminal for VS Code
- *crates* Støtteverktøy for Cargo.toml
- *Hex Editor* Muliggjør visning og redigering av minne i en hex-editor

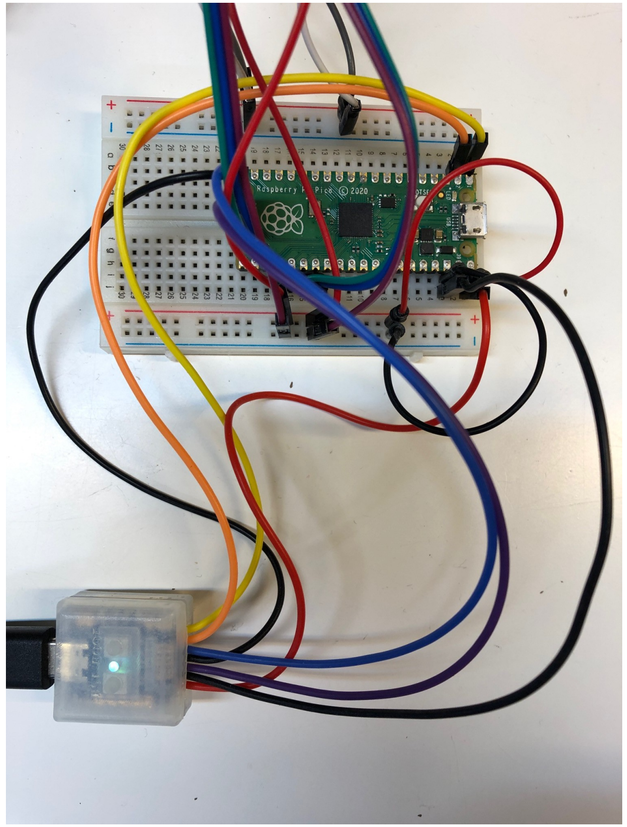
Picoprobe



Enten kan man koble opp en ekstra Raspberry Pi Pico med breadboards og benytte denne som Picoprobe...

...eller man kan lage en dediket dings med utgangspunkt i Tiny2040, lodde på ledninger, designe i FreeCAD og 3D-skrive innkapsling og knapper med Formlabs resin-skriver på Bitraf

(Om man da ikke allerede har en ST-Link er SEGGER J-Link)



Picoprobe brukes som en Serial Wire Debug til USB- og UART-bro. Siste Picoprobe firmware lastes ned fra Raspberry Pi Foundation i UF2-format fra *github.com/raspberrypi/picoprobe/releases*

Oppsett av probe-rs

- Fra terminalvindu, kjør
  - *sudo apt install -y libusb-1.0-0-dev libftdi1-dev libudev-dev libssl-dev*
  - *cargo install probe-rs --features cli*
- Last ned regel-filen *69-probe-rs.rules* fra *probe.rs/files/69-probe-rs.rules* og legg den i */etc/udev/rules.d*
  - Kjør *sudo udevadm control --reload* for å gjøre nye regler aktive
  - Kjør eventuelt *sudo udevadm trigger* for å gjøre nye regler aktive for probe som allerede er koblet til

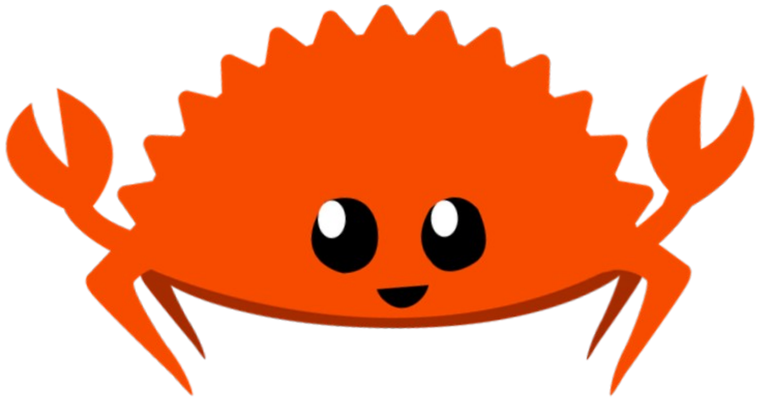
Ganske standard .cargo/config.toml

```
[target.'cfg(all(target_arch = "arm", target_os = "none"))']
runner = "probe-rs --chip RP2040"

rustflags = [
  "-C", "linker=flip-link",
  "-C", "link-arg=--nmagic",
  "-C", "link-arg=-Tlink.x",
  "-C", "inline-threshold=5",
  "-C", "no-vectorize-loops",
]

[build]
target = "thumbv6m-none-eabi"
```

For andre mikrokontrollere må man endre chip-angivelse på toppen og eventuelt mål-arkitekturen på bunnen



Hensiktsmessige filer under .vscode

Merk: Det som legges inn i *chip* og *rust-analyzer.cargo.target* vil bli plukket opp i andre konfigurasjonsfiler!

.vscode/settings.json:

```
{
    "chip": "RP2040",
    "rust-analyzer.cargo.target": "thumbv6m-none-eabi",
    "rust-analyzer.checkOnSave.allTargets": false,
}
```

Filer under .vscode forts.

.vscode/launch.json:

```
{
    "version": "0.2.0",
    "configurations": [
      {
        "preLaunchTask": "${defaultBuildTask}",
        "type": "probe-rs-debug",
        "cwd": "${workspaceRoot}",
        "request": "launch",
        "name": "Debug app",
        "chip": "${config:chip}",
        "coreConfigs": [
          {
            "programBinary": "${workspaceFolder}/target/${config:rust-analyzer.cargo.target}/debug/app"
          }
        ],
      }
  ]
}
```

Filer under .vscode forts.

.vscode/tasks.json:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Cargo build",
      "type": "shell",
      "command": "cargo",
      "args": [
        "flash",
        "--chip",
        "${config:chip}"
      ],
      "problemMatcher": [
        "$rustc"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

Starting a new Rust project in vscode

- Opprett mappe for prosjektet (f.eks. under) *$HOME/projects/embedded/pico*. Du kan enten bruke *cargo new <prosjekt-navn>* for å få opprettet rot-katalog for prosjektet og Rust-spesifikke filer. Alternativt kan du bruke *cargo init* i en allerede eksisterende katalog
- Om nødvendig (hvis du ikke har kompilert til den aktuelle mikrokontrolleren tidligere) installeres kompilerens maskinkode-oversettelses-lag med *rustup target add thumbv6m-none-eabi*
- Nå prosjekter som sikter mot samme mikrokontroller ligger under samme overordnede katalog (*pico*), blir det praktisk å legge *System View Description* som beskriver mikrokontrolleren i det XML-baserte CMSIS-SVD-formatet (rp2040.svd) her. Denne filen bidrar med nyttig kontekst-informasjon til brukergrensesnittet i vscode
- Med ARM mikrokontroller er det god praksis å benytte en linker (til erstatnning for *lld*, som er standard) som sørger for at voksende stack ikke kan spise seg inn i minneområdet til program og data; dette skjer med
  - *cargo install flip-link*
  - *Rustflags = ["-C", "linker=flip-link"]* inkluderes filen *.cargo/config.toml*
- I det følgende beskrives konfigurasjons-filene som minimum må være på plass for kompilere, flashe, debugge og kjøre prosjektet fra vscode

Prosjektmanifestet Cargo.toml i prosjektets rot-katalog

Det er praktisk å kalle alle applikasjoner for app. Da blir det mindre å endre/passe på, fra prosjekt til prosjekt

Cargo.toml:

```
[package]
edition = "2021"
name = "app"
version = "0.2.0"

[dependencies]
cortex-m = "0.7"
cortex-m-rt = "0.7"
embedded-hal = { version = "0.2.5", features = ["unproven"] }
rp-pico = "0.8.0"
panic-halt = "0.2.0"
fugit = "0.3.6"
```

Spesifisering av minne-layout memory.x for Raspberry Pi Pico

```
MEMORY {
    BOOT2 : ORIGIN = 0x10000000, LENGTH = 0x100
    FLASH : ORIGIN = 0x10000100, LENGTH = 2048K - 0x100
    RAM   : ORIGIN = 0x20000000, LENGTH = 256K
}

EXTERN(BOOT2_FIRMWARE)

SECTIONS {
    /* ### Boot loader */
    .boot2 ORIGIN(BOOT2) :
    {
        KEEP(*(.boot2));
    } > BOOT2
} INSERT BEFORE .text;
```

Innhold i memory.x avhenger av mikrokontroller
Man kan typisk finne egnet fil i repository til mikro-kontrollerens Hardware Abstraction Layer

Kode som blinker og skriver til UART
Funksjonaliteten til Picoprobe testes

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

use rp_pico as bsp;
use bsp::entry;
use bsp::hal::{
    pac,
    watchdog::Watchdog,
    clocks::{init_clocks_and_plls, Clock},
    sio::Sio,
    uart::{DataBits, StopBits, UartConfig},
};
use embedded_hal::digital::v2::OutputPin;
use core::fmt::Write;
use fugit::RateExtU32;
use panic_halt as _;

#[entry]
fn main() -> ! {
    let mut pac = pac::Peripherals::take().unwrap();
    let core = pac::CorePeripherals::take().unwrap();
```

Kode forts.

```
let mut watchdog = Watchdog::new(pac.WATCHDOG);
let sio = Sio::new(pac.SIO);

let clocks = init_clocks_and_plls(
    12_000_000_u32, //bsp::XOSC_CRYSTAL_FREQ,
    pac.XOSC,
    pac.CLOCKS,
    pac.PLL_SYS,
    pac.PLL_USB,
    &mut pac.RESETS,
    &mut watchdog,
).ok().unwrap();
let mut delay = cortex_m::delay::Delay::new(core.SYST, clocks.system_clock.freq().to_Hz());

let pins = bsp::Pins::new(
    pac.IO_BANK0,
    pac.PADS_BANK0,
    sio.gpio_bank0,
    &mut pac.RESETS,
);
```

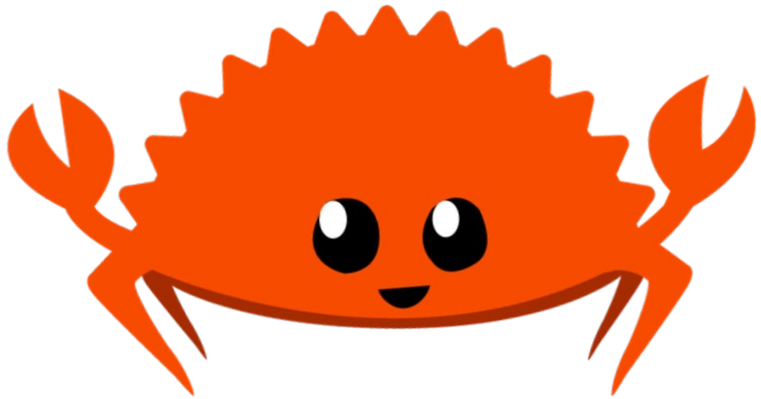
Kode forts.

```
    let mut led_pin = pins.led/* gpio17 */.into_push_pull_output();
    let uart_pins = (pins.gpio0.into_function::<bsp::hal::gpio::FunctionUart>(),
            pins.gpio1.into_function::<bsp::hal::gpio::FunctionUart>());
    let mut uart = bsp::hal::uart::UartPeripheral::new(pac.UART0, uart_pins, &mut pac.RESETS)
        .enable(
            UartConfig::new(9600.Hz(), DataBits::Eight, None, StopBits::One),
            clocks.peripheral_clock.freq(),
        ).unwrap();

    uart.write_full_blocking(b"Program start\r\n");

    loop {
        writeln!(uart, "on!\r").unwrap();
        led_pin.set_high().unwrap();
        delay.delay_ms(250);
        writeln!(uart, "off!\r").unwrap();
        led_pin.set_low().unwrap();
        delay.delay_ms(250);
    }
}
```

Pico W har en veldig spesiell innebygd LED, som ikke kan blinkes like enkelt
Da kan man eventuelt koble en LED til f.eks. GPIO17 som kommentert i koden

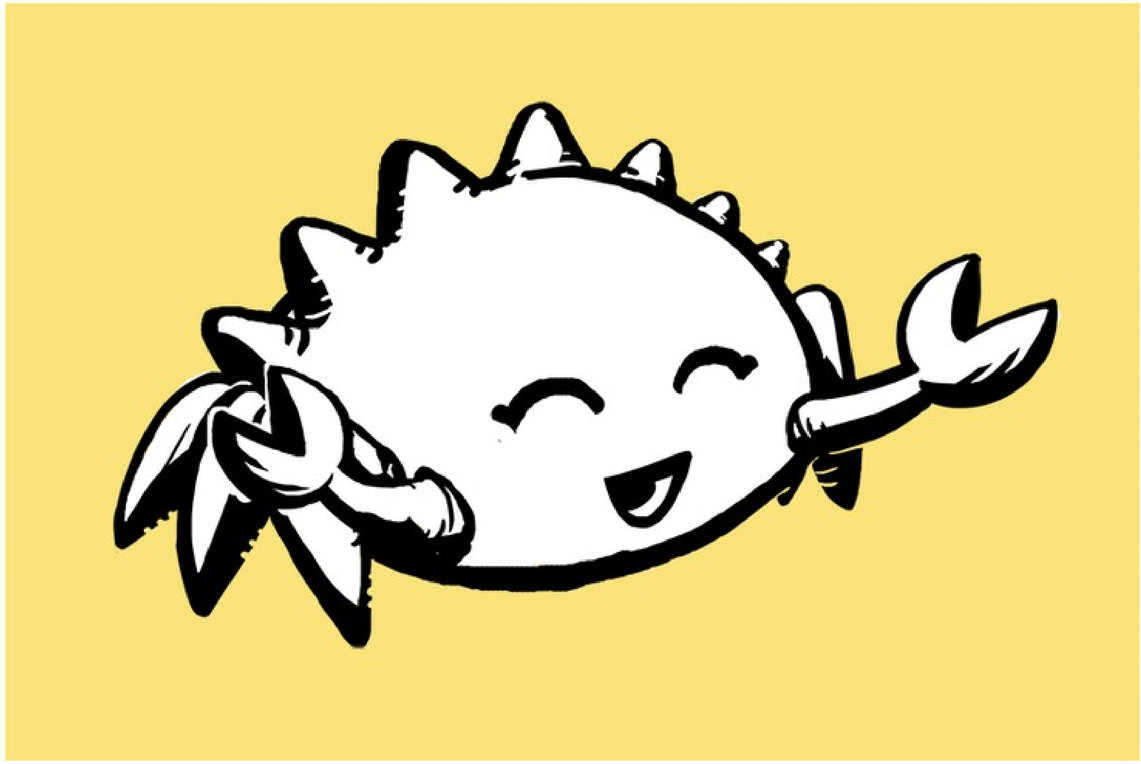


Muligheter og mer informasjon

- Rust gjør det mulig å skrive effektiv kode mot operativsystemer, mot web Web Assembly og mot mikrokontrollere, på samme måte som C og C++, men med de mest moderne språkmekanismer og uten frykt for minnelekkasje
- Når man skriver kompliserte applikasjoner mot embedded, finnes det et par gode sanntids-rammeverk, utover ARM Cortex sitt innebygde NVIC (Nested Vectored Interrupt Controller)
  - RealTime Interupt driven Concurrency *github.com/rtic-rs/rtic*
  - Embassy *embassy.dev*
- Det kan varmt anbefales å benytte gratis opplæringsressurser på internett for å øve programmeringsspråket, standardbiblioteket og viktigste crates
  - *codewars.com*
  - *github.com/rust-lang/rustlings*
  - *exercism.org*
- Sist men ikke minst må Boka nevnes (doc.rust-lang.org/book)!
  Den kan kjøpes eller leses gratis på internett, og gir en dyp presentasjon til språket sammen med funksjonelt slående oppgaver

Det var hva vi hadde forberedt

Takk for oss!



Ekstra

Programmer kan eventuelt kjøres uten programmeringsprobe ved å

- Installere *cargo install elf2uf2-rs --locked*
- Bytte runner i *.cargo/config.toml* med *runner = "elf2uf2-rs -d"*
- Og med Raspberry Pi Pico tilkoblet via USB, kjøre *cargo run*