Ariel OS — The Embedded Rust Software Stack for Microcontroller-based Internet of Things

About us

Kaspar Schleiser *Koen Zandberg*

… part of Ariel OS contributors:

Ariel OS: A library operating system for secure, memory-safe, low-power Internet of Things, written in Rust

Outline

1. Context
2. Rust embedded and the Ecosystem
3. Ariel OS
4. Getting started: From Hello World to Networking
5. Wrapping up

Context: Microcontroller hardware

Restrictions

- Limited processing power
- Low memory
- Single memory: no user- vs kernel-space, no MMU, but probably an MPU

Context: Microcontroller hardware

Restrictions

- Limited processing power
- Low memory
- Single memory: no user- vs kernel-space, no MMU, but probably an MPU

Firmware

- **Baremetal**: no underlying OS with threads, networking, heap allocation…
- **No alloc**: deterministic behaviour, avoid panic and memory fragmentation

Outline

1. Context
2. Rust embedded and its Ecosystem
3. Ariel OS
4. Getting started: From Hello World to Networking
5. Wrapping up

Rust: Language

Memory safety!

Rust: Language

Memory safety!

But we know that by now.

Rust: Language

Crates:

Shared code on crates.io, integrated in the Cargo build system.

Traits:

Interfaces that allow for interoperability between crates.

`no_std`:

No use of the standard library and dynamic memory allocations

Async

Provide asynchronous cooperative multitasking implementations with minimal memory usage

Fosters collaboration across projects

Rust Embedded: Ecosystem

What the ecosystem provides:

`embedded-hal` traits

Provide good generic interfaces for common peripherals across architectures

Rust Embedded: Ecosystem

What the ecosystem provides:

`embedded-hal` traits

Provide good generic interfaces for common peripherals across architectures

Driver crates

Sensor and peripheral drivers as separate crate, make use of the `embedded-hal` traits

Rust Embedded: Ecosystem

What the ecosystem provides:

`embedded-hal` traits

Provide good generic interfaces for common peripherals across architectures

Driver crates

Sensor and peripheral drivers as separate crate, make use of the `embedded-hal` traits

Rust Embedded: Ecosystem

What the ecosystem provides:

`embedded-hal` traits

Provide good generic interfaces for common peripherals across architectures

Driver crates

Sensor and peripheral drivers as separate crate, make use of the `embedded-hal` traits

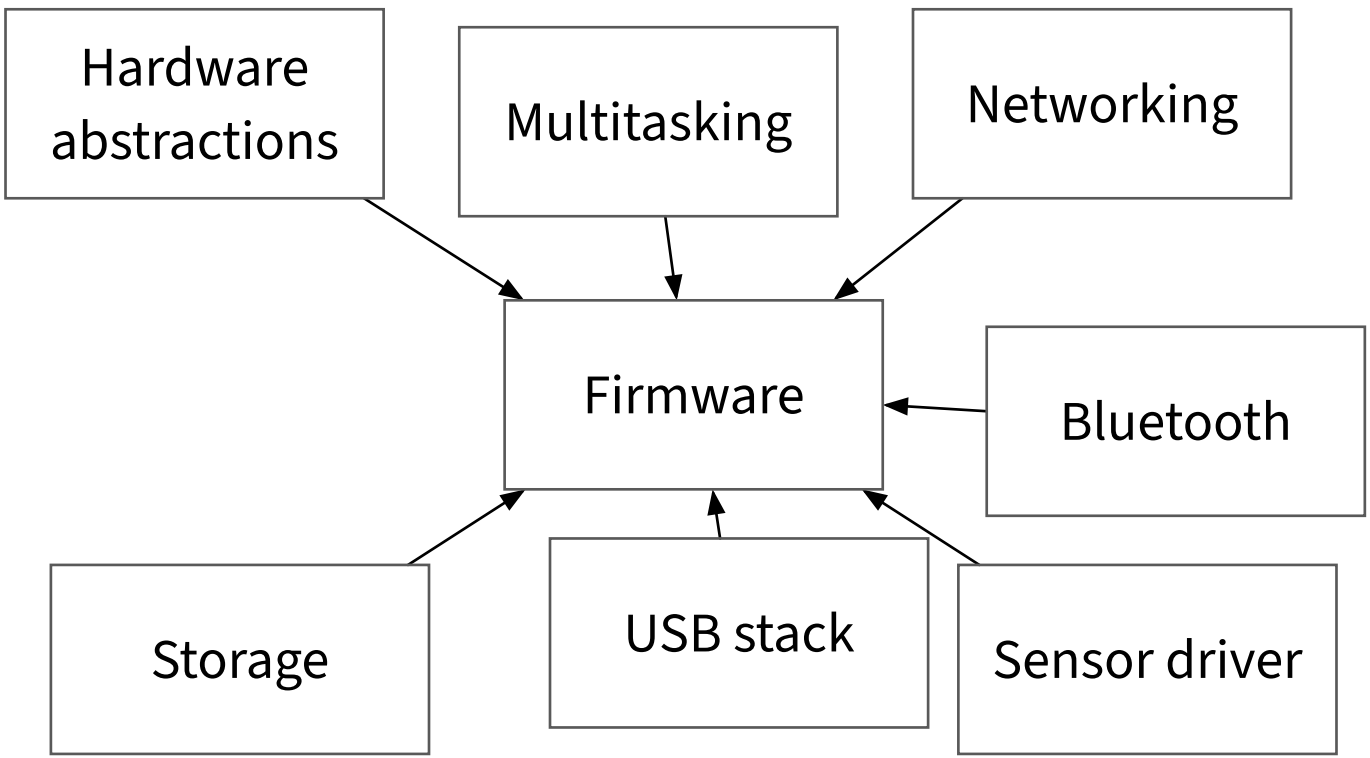
Async framework: Embassy

- Implements hardware abstractions for STM32, nRF and Pi Pico
- Provides a low memory async scheduler

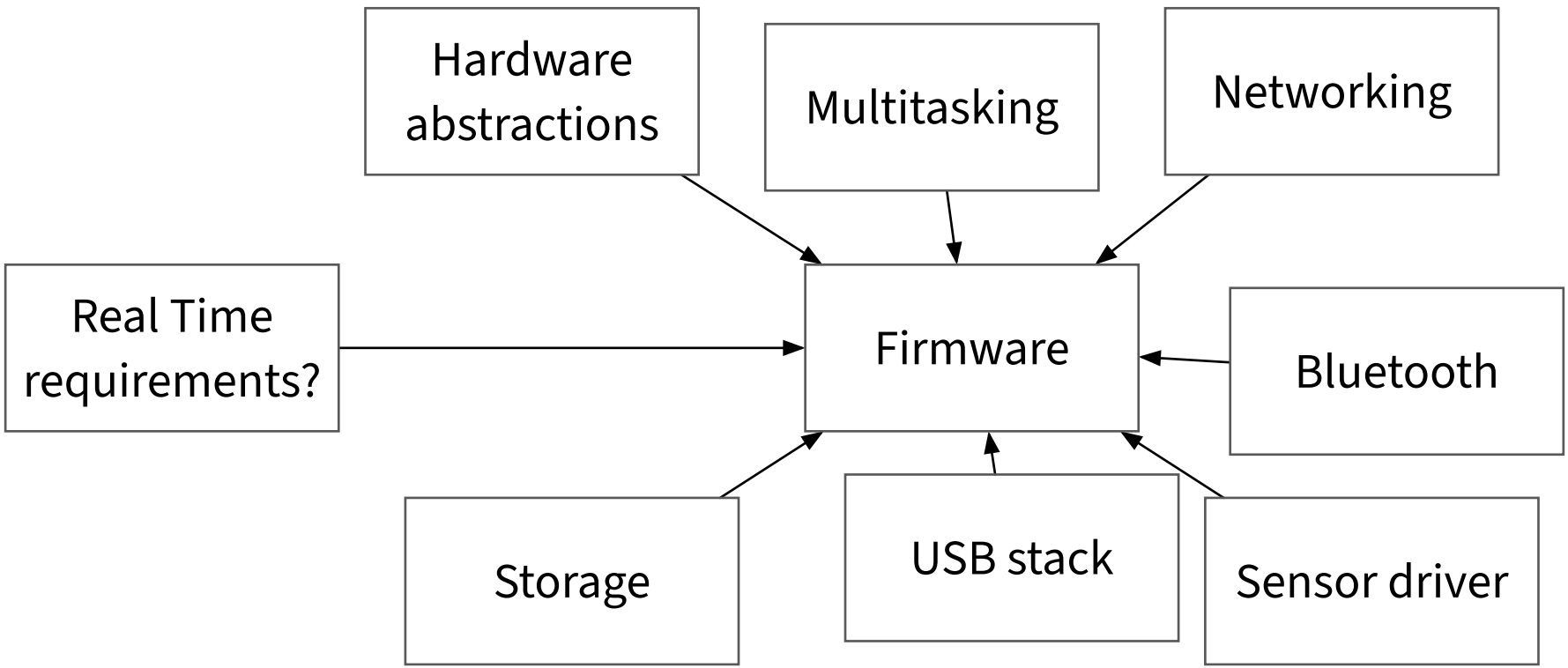
The ecosystem of crates around embedded Rust is huge and growing fast!

Collecting everything

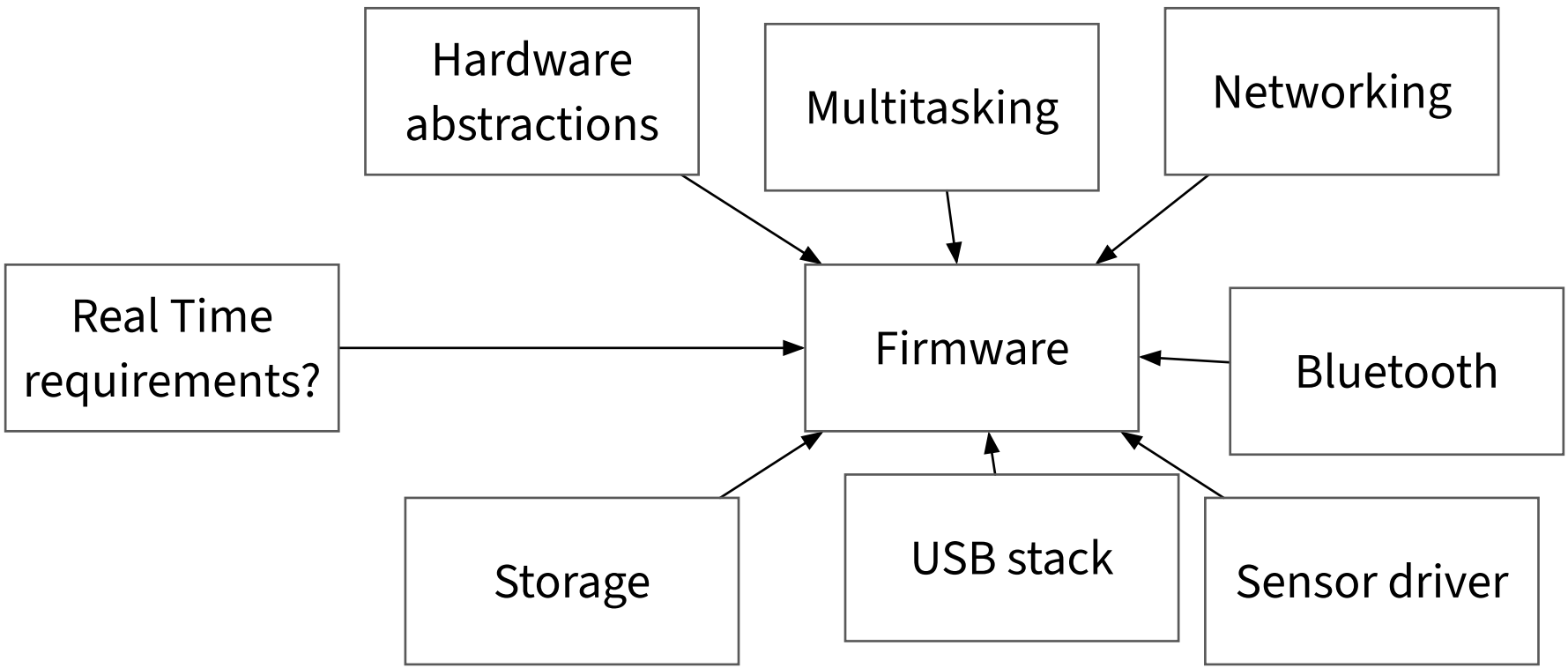
Collecting everything



Collecting everything



Collecting everything

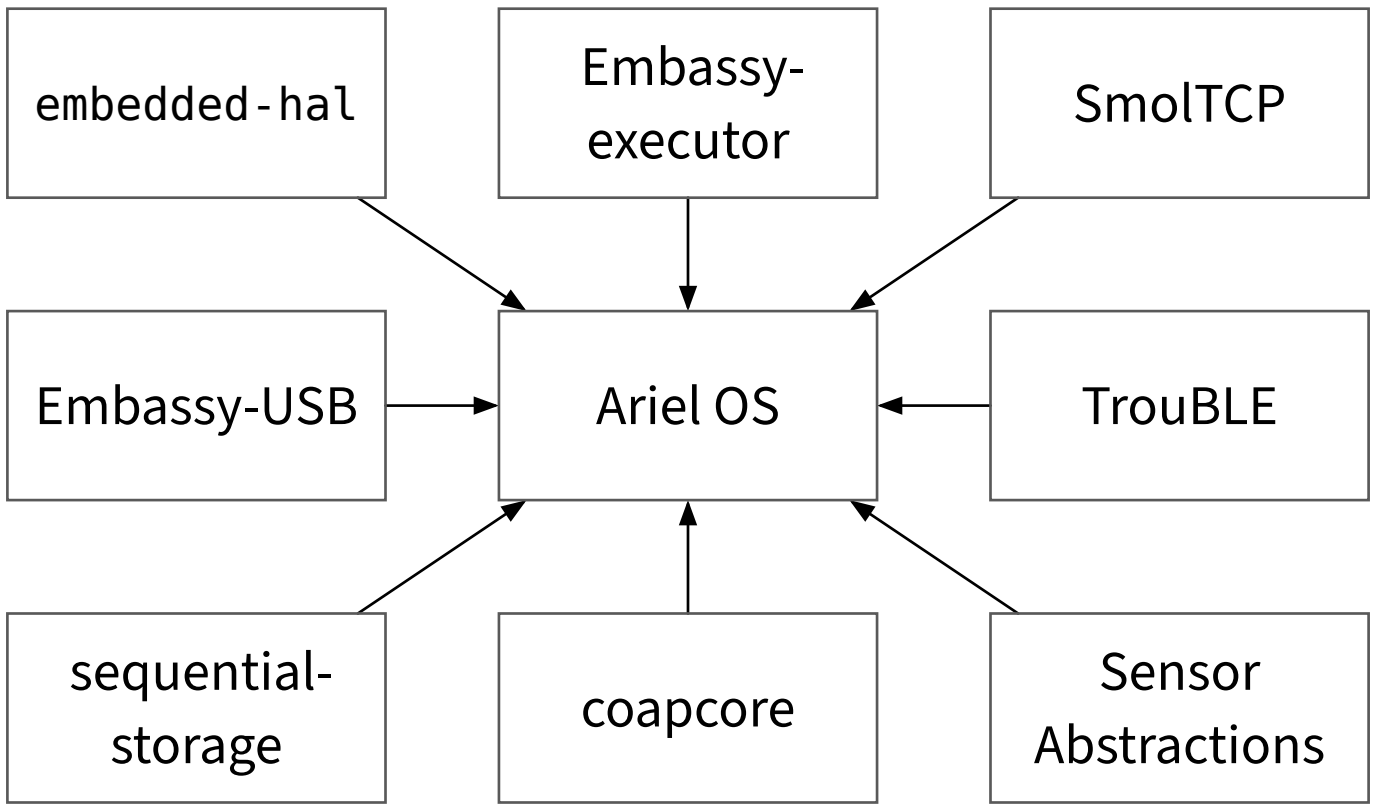


Curating and integrating crates for everything is time consuming

Outline

1. Context
2. Rust embedded and the Ecosystem
3. Ariel OS
4. Getting started: From Hello World to Networking
5. Wrapping up

Ariel OS



Ariel OS integrates crates into a coherent operating system

Ariel OS: Concurrency

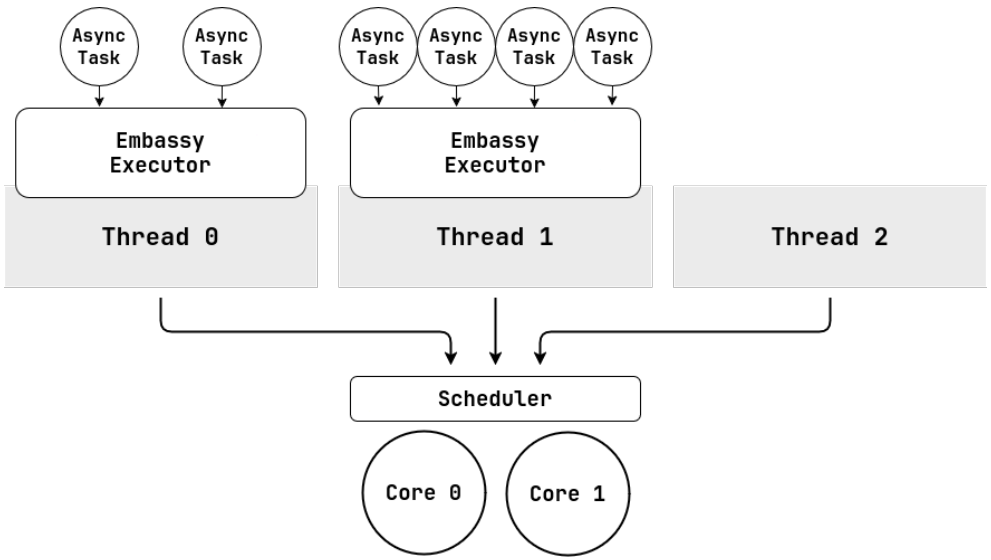
Threading

- Preemptive scheduler
- Priority-based scheduling
- Separate stacks

Async

- Based on the Embassy executor
- Can run inside a thread

Support both preemptive scheduling and low memory async scheduling



Ariel OS: To the Ecosystem

Provide high level functionality

Sensor abstractions

- Enumerate available sensors on a board
- Read out available sensors
- Provides a generic interface to sensors and measurements.

Ariel OS: To the Ecosystem

Provide high level functionality

Sensor abstractions

- Enumerate available sensors on a board
- Read out available sensors
- Provides a generic interface to sensors and measurements.

Structured Board Descriptions

Provide machine-readable descriptions of boards

- Microcontroller information
- Peripherals
- Supported features

Ariel OS: Build system

The configuration space is huge

- Enable features
- Features can conflict
- Catch issues *before* compilation

Ariel OS: Build system

The configuration space is huge

- Enable features
- Features can conflict
- Catch issues *before* compilation

Laze

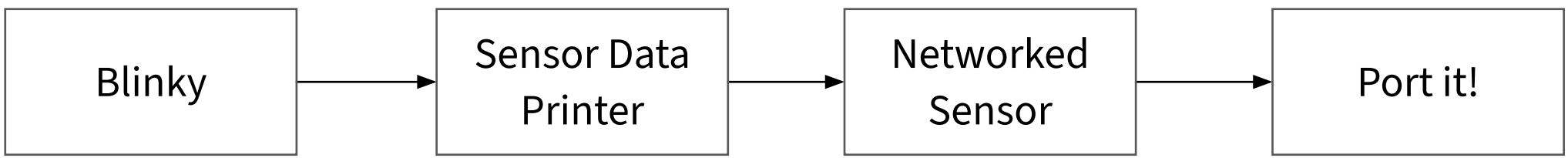
yaml-based declarative build configuration.

- Select required modules
- Abstract away board specifics
- Steers cargo builds
- Runs tasks to flash and inspect builds

Outline

1. Context
2. Rust embedded and the Ecosystem
3. Ariel OS
4. Getting started: From Hello world to Networking
5. Wrapping up

Goal



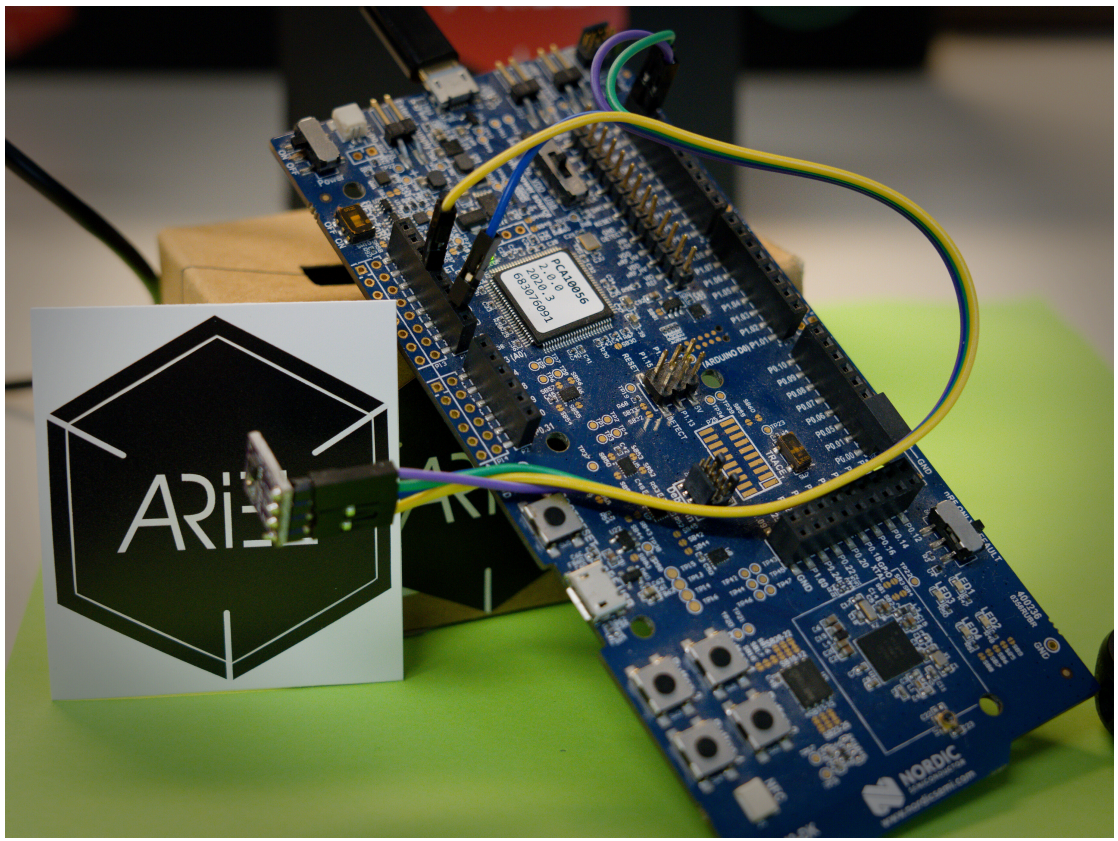
Minimal setup with blinking LED

Connect a sensor

Connect the sensor to the internet

Port the example to another board

Hello World: Hardware



Hello World: Steps

Goal

Show a basic blinking LED

Hardware:

Nordic nRF52840 development kit

Steps

1. Generate the basic project from the template
2. Define LED (GPIO) pin
3. Toggle the led pin in a loop

Hello World: Steps

```Shell
$ cargo generate --git https://github.com/ariel-os/ariel-os-template --name hello-fosdem
```

Project content

```
├─ Cargo.toml
├─ laze-project.yml
└─ src/
   └─ main.rs
```

Hello World: Code

Rust

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

use ariel_os::debug::
  {ExitCode, exit, log::*};

#[ariel_os::task(autostart)]
async fn main() {
    info!("Hello World!");

    exit(ExitCode::SUCCESS);
}
```

Boilerplate

YAML

```yaml
# laze-project.yml
apps:
  - name: hello-fosdem
```

TOML

```toml
# Cargo.toml
[package]
name = "hello-fosdem"
version = "0.1.0"
edition = "2024"
[dependencies]
ariel-os = { path = "..." }
ariel-os-boards = { path = "..." }
```

Hello World: Running

```
1 $ laze build -b nrf52840dk run                                    Shell
2 [...] (compiling & flashing)
3 [INFO ] Hello World! (hello_fosdem hello-fosdem/src/main.rs:8)    } Printed by hardware
4 Firmware exited successfully
```

Hello World: Add the LED

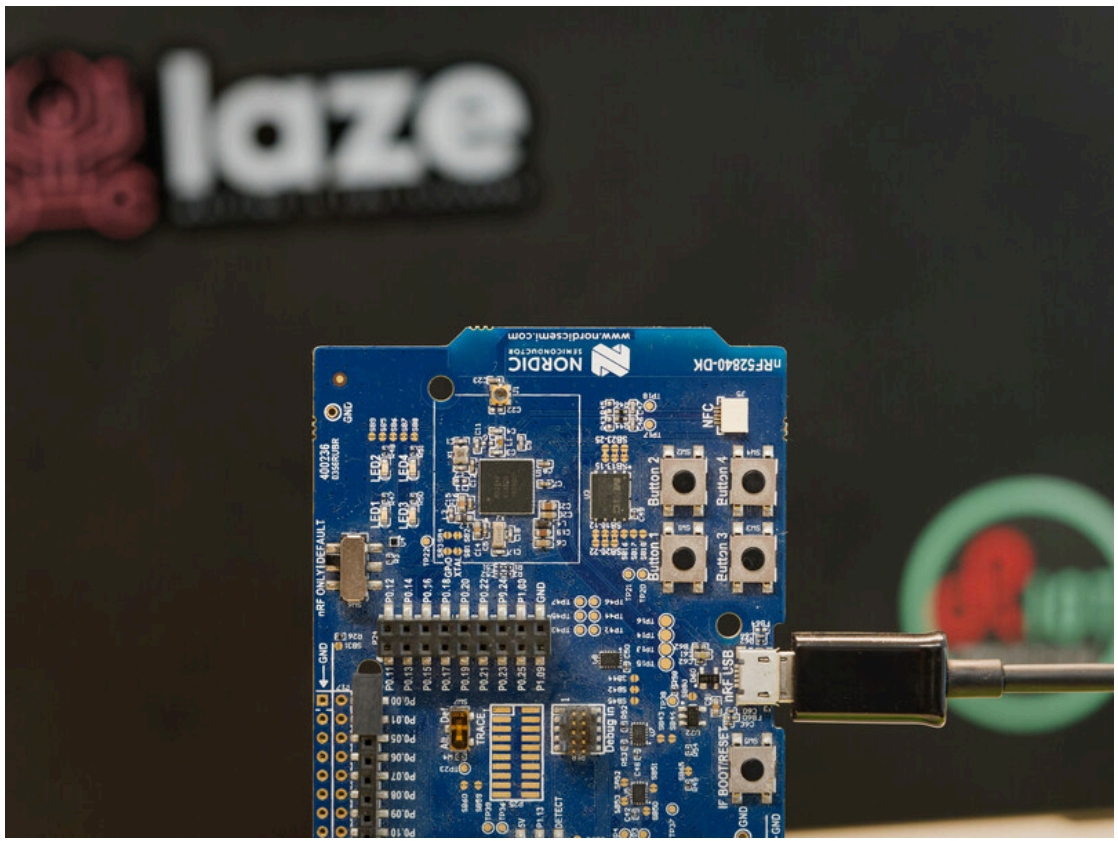
Rust

```rust
define_peripherals!(Peripherals { led0: P0_13 });

#[ariel_os::task(autostart, peripherals)]
async fn main(peripherals: Peripherals) {
    let mut led0 = Output::new(peripherals.led0, Level::Low);

    loop {
        Timer::after_millis(1000).await;
        led0.toggle();
    }
}
```

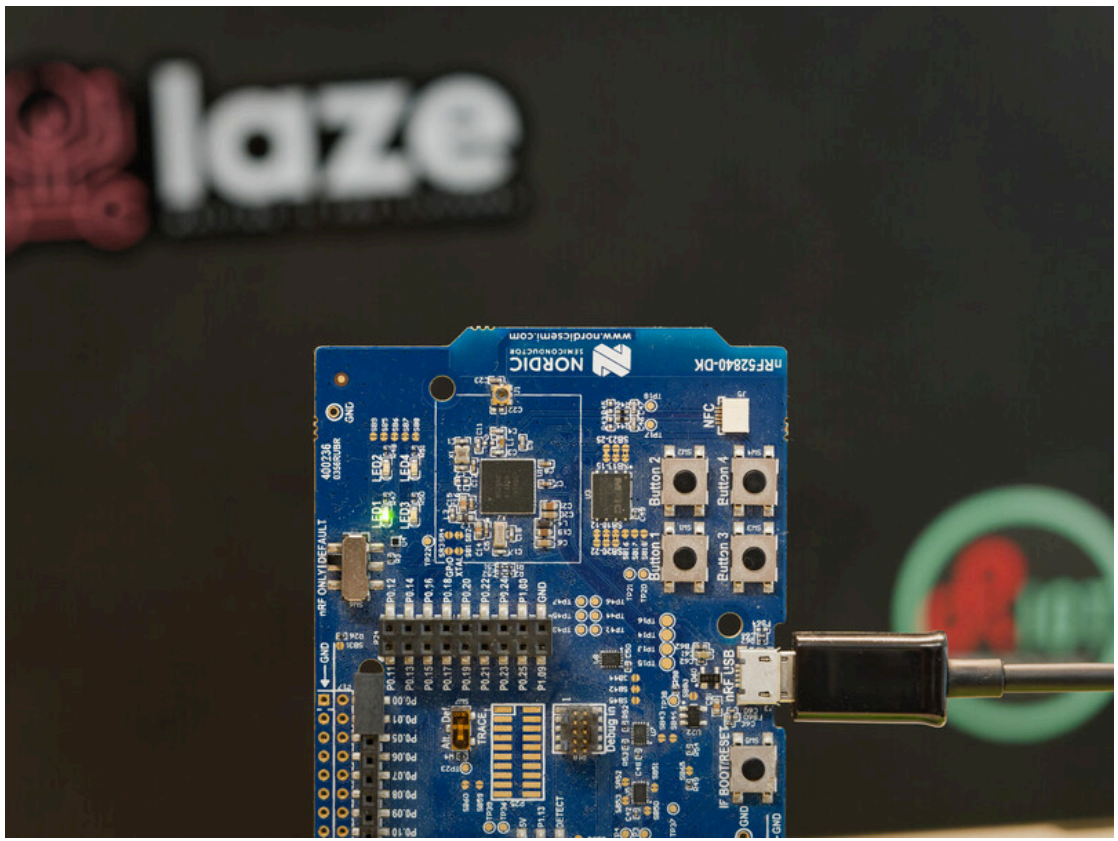
Hello World: Add the LED

```toml
[dependencies]
# add "time" feature
ariel-os = { path = "...", features = ["time"] }
```

Hello World: it blinks



Hello World: it blinks



Sensor Data Printer: Steps

Goal

Add an I2C temperature and humidity sensor

Hardware

SHT31 Temperature & Humidity sensor

Steps

1. Add SHT3x sensor driver crate
2. Define I2C bus in `pins.rs`
3. Read & print out sensor data

Sensor Data Printer: add sensor driver crate

```toml
[dependencies]
# …
embedded-sht3x = { git = "https://gitlab.com/ghislainmary/embedded-sht3x",
                   features = [
                     "async"
                   ]
                 }
```

Sensor Data Printer: define I2C bus/pins

Rust

```rust
mod board {
    use ariel_os::hal::{peripherals, define_peripherals};

    pub type SensorI2c = ariel_os::hal::i2c::controller::TWISPI0;

    define_peripherals!(Peripherals {
        led0: P0_13,
        i2c_sda: P0_26,
        i2c_scl: P0_27,
    });
}
```

Sensor Data Printer: initialize I2C bus

Rust

```rust
#[ariel_os::task(autostart, peripherals)]
async fn main(peripherals: board::Peripherals) {
    let mut led0 = Output::new(peripherals.led0, Level::Low);

    let mut i2c_config = Config::default();
    i2c_config.frequency = const
      { highest_freq_in(Kilohertz::kHz(100)..=Kilohertz::kHz(400)) };

    let i2c_bus = board::SensorI2c::new(peripherals.i2c_sda,
                                        peripherals.i2c_scl,
                                        i2c_config)
```

I2C setup

Sensor Data Printer: perform measurement

Rust

```rust
    // set up sht3x driver
    let mut sensor = Sht3x::new(i2c_bus, DEFAULT_I2C_ADDRESS, Delay);

    loop {
        // Perform a temperature and humidity measurement
        let measurement = sensor.single_measurement().await.unwrap();
        let temp = measurement.temperature.celcius();
        let hum = measurement.relative_humidity;

        info!("temp: {} °C, rel. hum.: {} %\n", temp, hum);

        Timer::after_millis(1000).await;
    }
```

Retrieve Measurement

Sensor Data Printer: that's it

```
$ laze build -b nrf52840dk run
[...] (compiling)
     Running `probe-rs run --protocol=swd --chip nrf52840_xxAA --preverify build/bin/nrf52840dk/cargo/thumbv7em-none-eabihf/release/hello-fosdem`
    Verifying ✔ 100% [####################]  20.00 KiB @  30.90 KiB/s (took 1s)
     Finished in 0.65s
[INFO ] temp: 19.165329 °C, rel. hum.: 39.252308 %
 (hello_fosdem hello-fosdem-2026/src/main.rs:49)
[INFO ] temp: 19.17868 °C, rel. hum.: 39.33318 %
 (hello_fosdem hello-fosdem-2026/src/main.rs:49)
[INFO ] temp: 19.151978 °C, rel. hum.: 39.261463 %
 (hello_fosdem hello-fosdem-2026/src/main.rs:49)
[INFO ] temp: 19.151978 °C, rel. hum.: 39.223316 %
 (hello_fosdem hello-fosdem-2026/src/main.rs:49)
```

Networked Sensor: Steps

Goal

Provide the sensor measurements over a TCP socket

Steps

1. Add TCP send function
2. Call from our main loop
3. Add dependencies
4. Select networking in `laze.yml`

Networked Sensor: TCP send

Rust

```rust
async fn report(s: &str) -> Result<(), &'static str> {
    let host_addr = Ipv4Address::from_str("192.168.1.131").unwrap();
    let stack = net::network_stack().await.unwrap();
    let mut rx_buffer = [0; 256];
    let mut tx_buffer = [0; 256];
    stack.wait_config_up().await;
    let mut socket = TcpSocket::new(stack, &mut rx_buffer, &mut
    tx_buffer);
    socket.connect((host_addr, 4242)).await.map_err(|_| "connect")?;
    socket.write_all(s.as_bytes()).await.map_err(|_| "write_all")?;
    socket.flush().await.map_err(|_| "flush")?;
    Ok(())
}
```

Boilerplate

Create Socket, connect, send

Networked Sensor: Add to the loop

Rust

```rust
loop {
    // ...
    let mut s: String<64> = String::new();
    write!(s, "temp: {temp:.1} °C, rel. hum.: {hum:.1} %\n").unwrap();

    if let Err(e) = report(s.as_str()).await {
        info!("reporting failed: {}", e);
    }
}
```

Format String

Call report

Networked Sensor: Add dependencies

TOML

```toml
[dependencies]
ariel-os = { path = "...",
features = [
  "i2c",
  "tcp",
  "time",
] }
// ...
embedded-io-async = "0.6.1"
heapless = "0.9.2"
```

YAML

```yaml
apps:
  - name: hello-fosdem
    selects:
      - network
```

Networked Sensor: output

```Shell
$ socat -u TCP-LISTEN:4242,fork STDOUT
temp: 20.1 °C, rel. hum.: 45.3 %
temp: 20.1 °C, rel. hum.: 45.2 %
temp: 20.1 °C, rel. hum.: 45.3 %
```

Port it: Steps

Goal

Run the networked sensor on different hardware

Steps

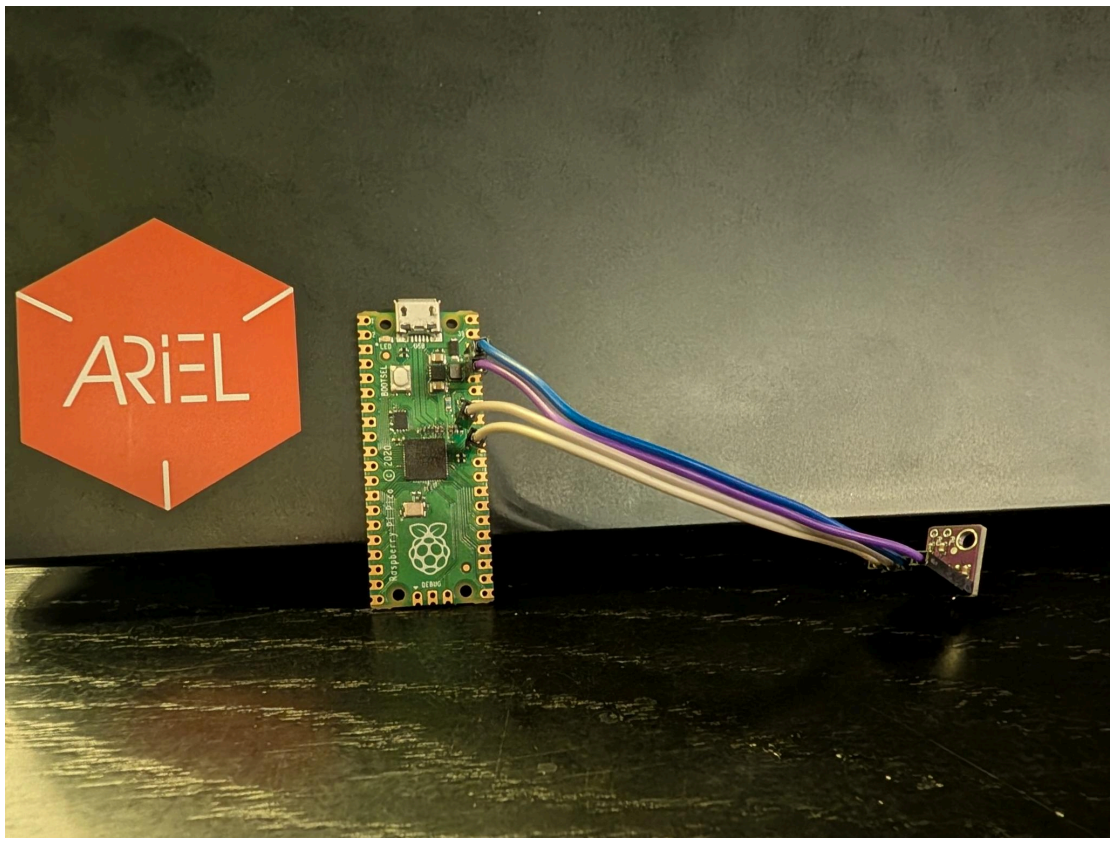
1. define I2C and LED pins for the next board

Port it: defining I2C and LED pins

Rust

```rust
#[cfg(context = "nrf52840dk")]
mod board { /* ... previous version ... */ }

#[cfg(context = "rpi-pico")]
mod board {
    pub type SensorI2c = ariel_os::hal::i2c::controller::I2C0;
    define_peripherals!(Peripherals {
        led0: PIN_25,
        i2c_sda: PIN_12,
        i2c_scl: PIN_13,
    });
}
```

Port it: that’s it



What’s up next

release 0.3.0 landing next week

Adding Bluetooth Low Energy, native “board”, Structured Board Descriptions, …

release after that

secure software updates, better power management, …

Wrapping up

- Ariel OS curates and integrates the embedded Rust ecosystem
- Embedded Rust has never been that easy

You’re now thinking:

Wrapping up

- Ariel OS curates and integrates the embedded Rust ecosystem
- Embedded Rust has never been that easy

You’re now thinking:

- “Why did they use IPv4 in 2026?”

Wrapping up

- Ariel OS curates and integrates the embedded Rust ecosystem
- Embedded Rust has never been that easy

You’re now thinking:

- “Why did they use IPv4 in 2026?”
- “This looks so approachable, I’ll try it!”

Thanks!

Join the action:

https://ariel-os.org