

# Storing cryogenic samples in ultra-high vacuum

Reto Trappitsch



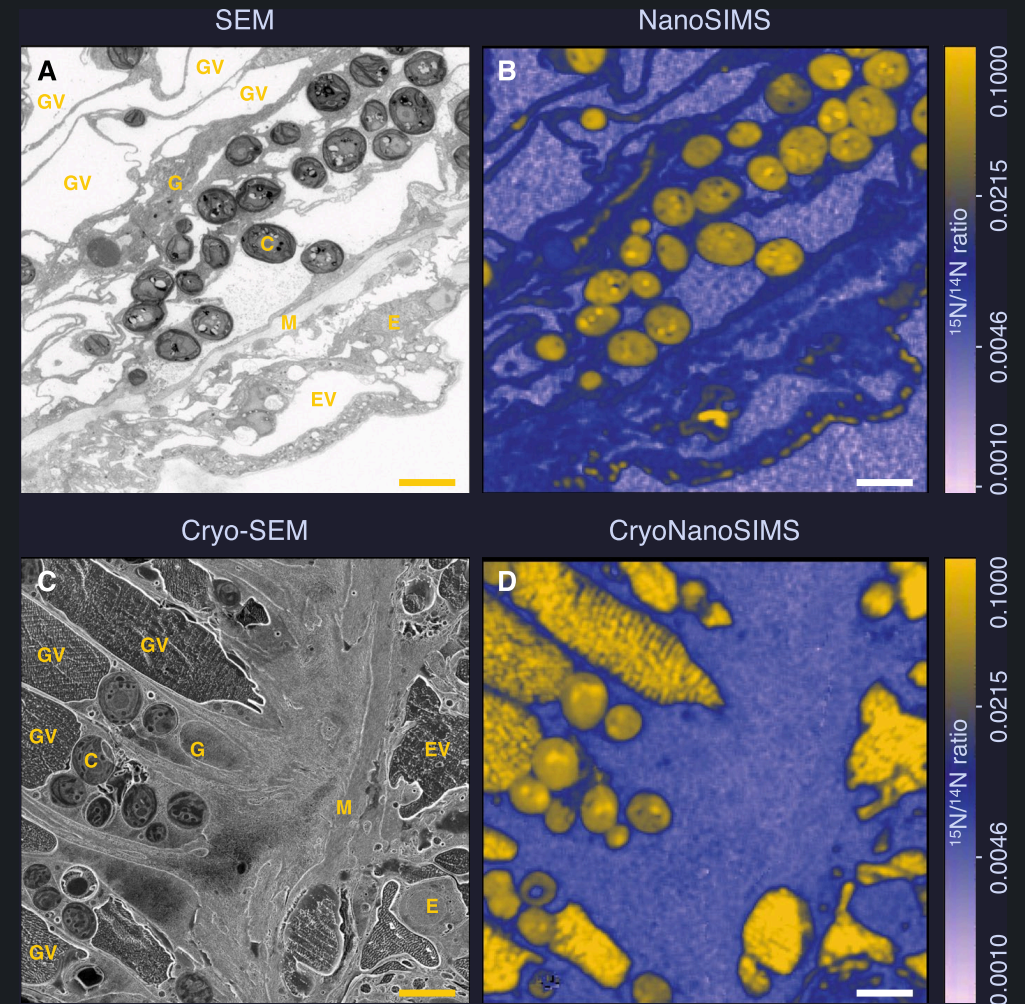
Annual Swiss RSE Day 2026  
August 31, 2026



# Tracing biological processes with mass spectrometry

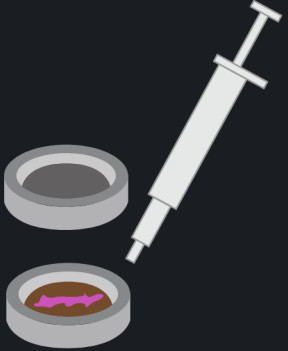
- Biological samples: Water!
- Cannot be analyzed as-is in ultra-high vacuum (UHV)
- Traditional analysis: Substitution of soluble components
- High-pressure freezing:
  - Vitrifies the sample
  - UHV-compatible if kept frozen

**High-pressure vitrification enables analysis of pristine samples.**



# Cryogenic sample preparation workflow

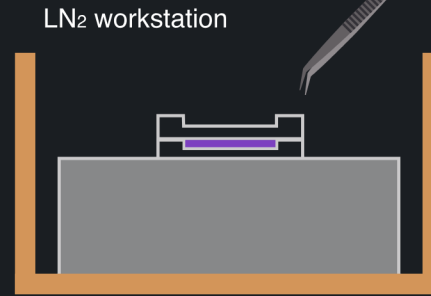
1. Sample loading



2. High pressure freezing



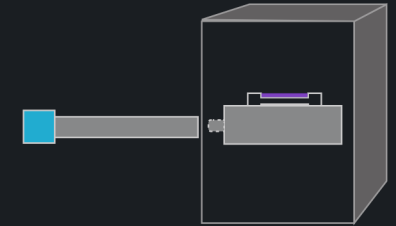
3. Cryo-holder loading



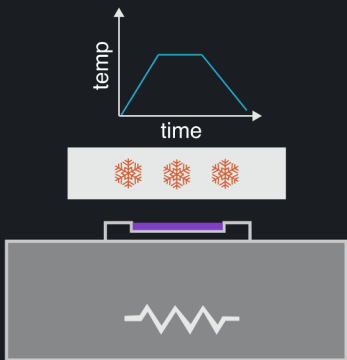
4. Cryo-planing



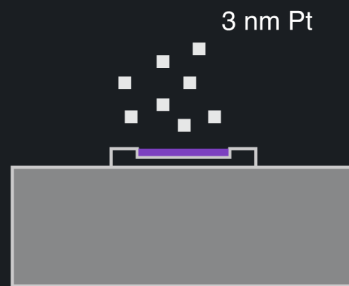
5. Vacuum cryo-transfer



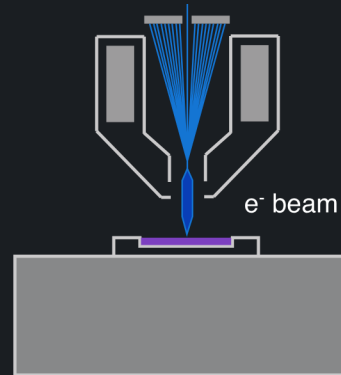
6. Ice sublimation



7. Cryo-coating



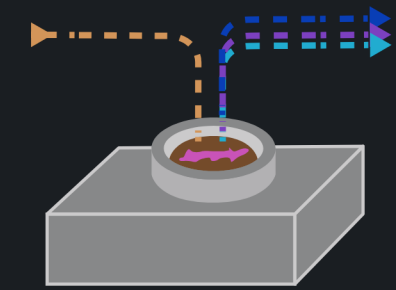
8. Cryo-SEM imaging



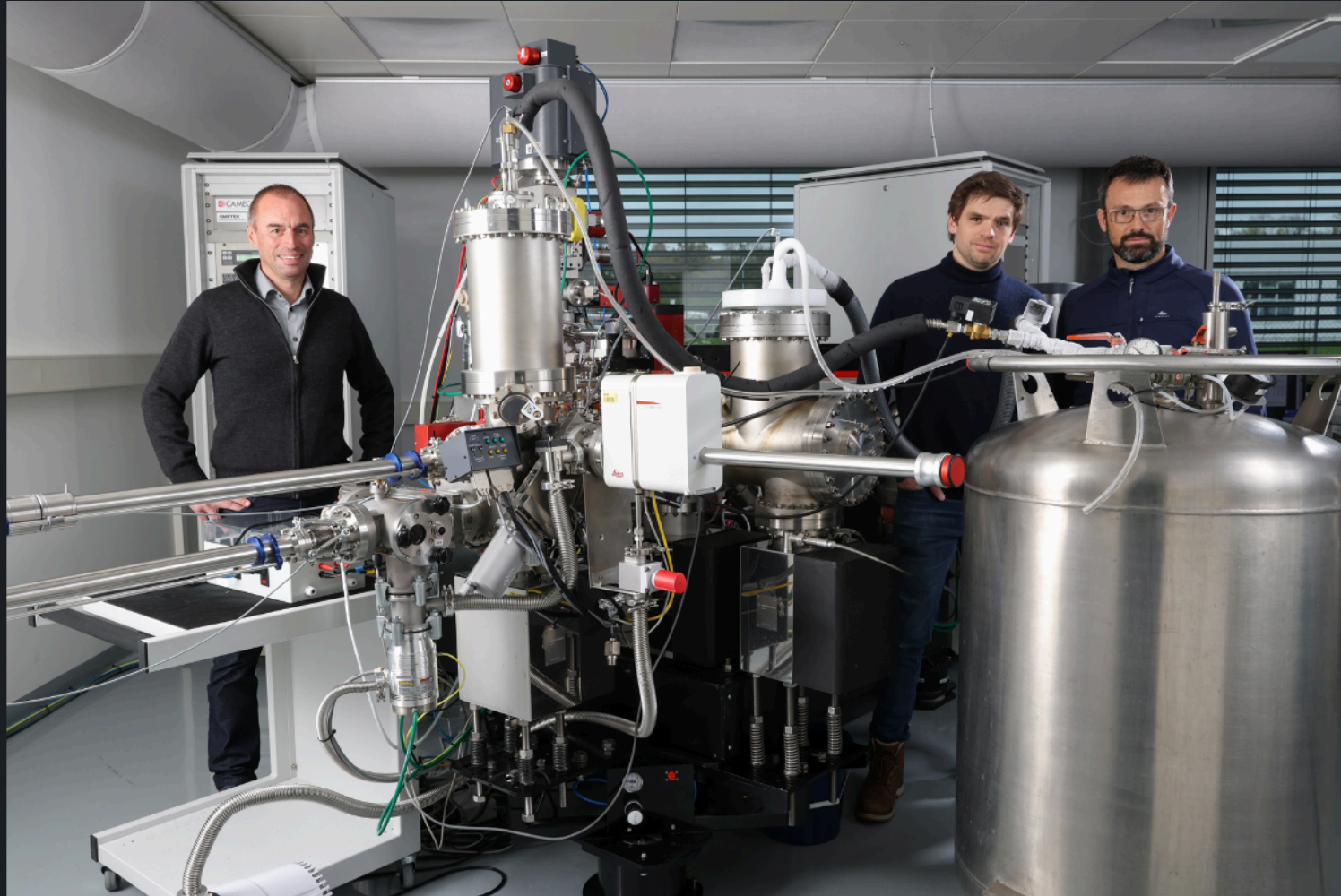
9. Cryo-coating



10. CryoNanoSIMS



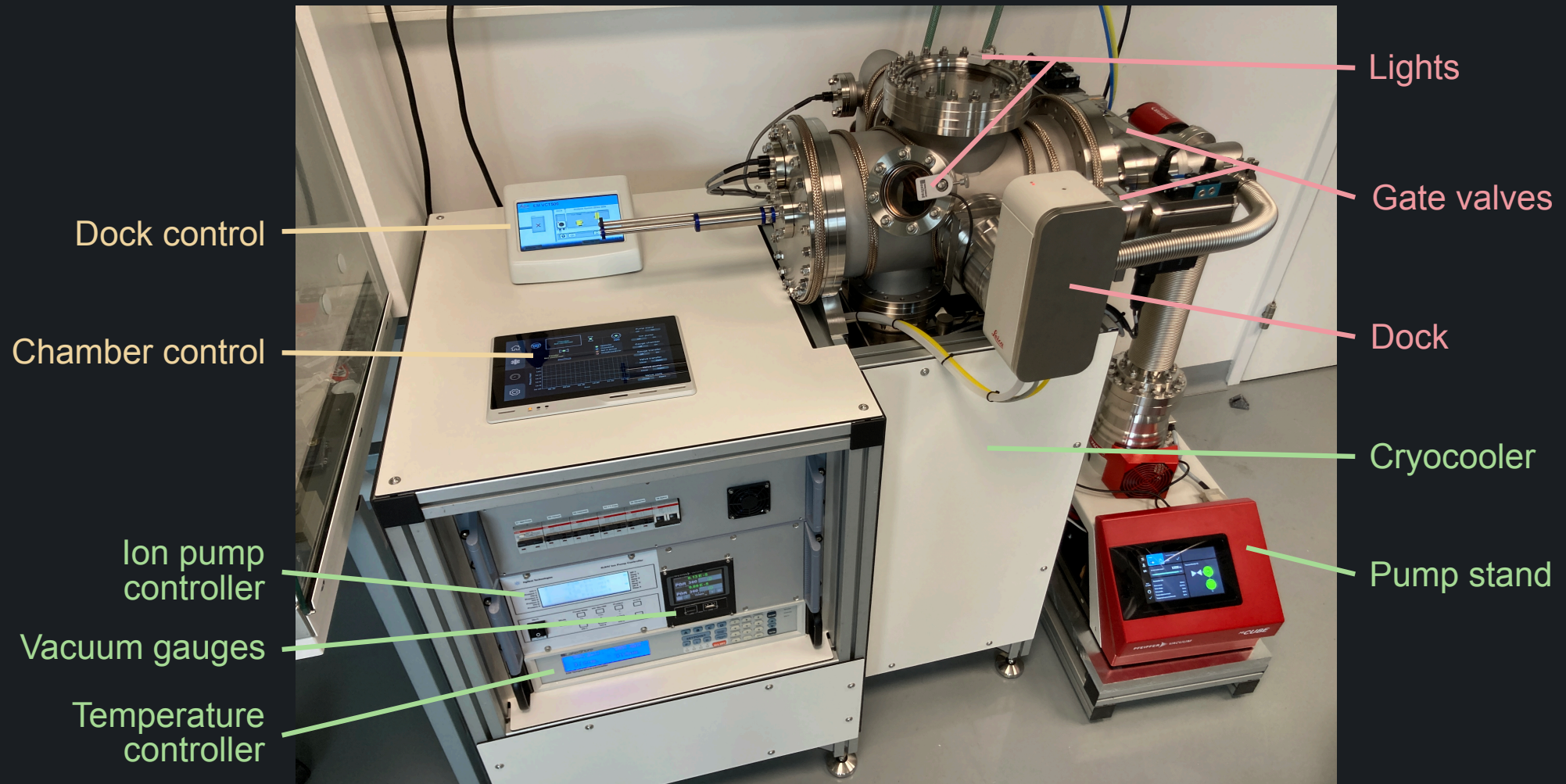
# The CryoNanoSIMS: A unique instrument at EPFL



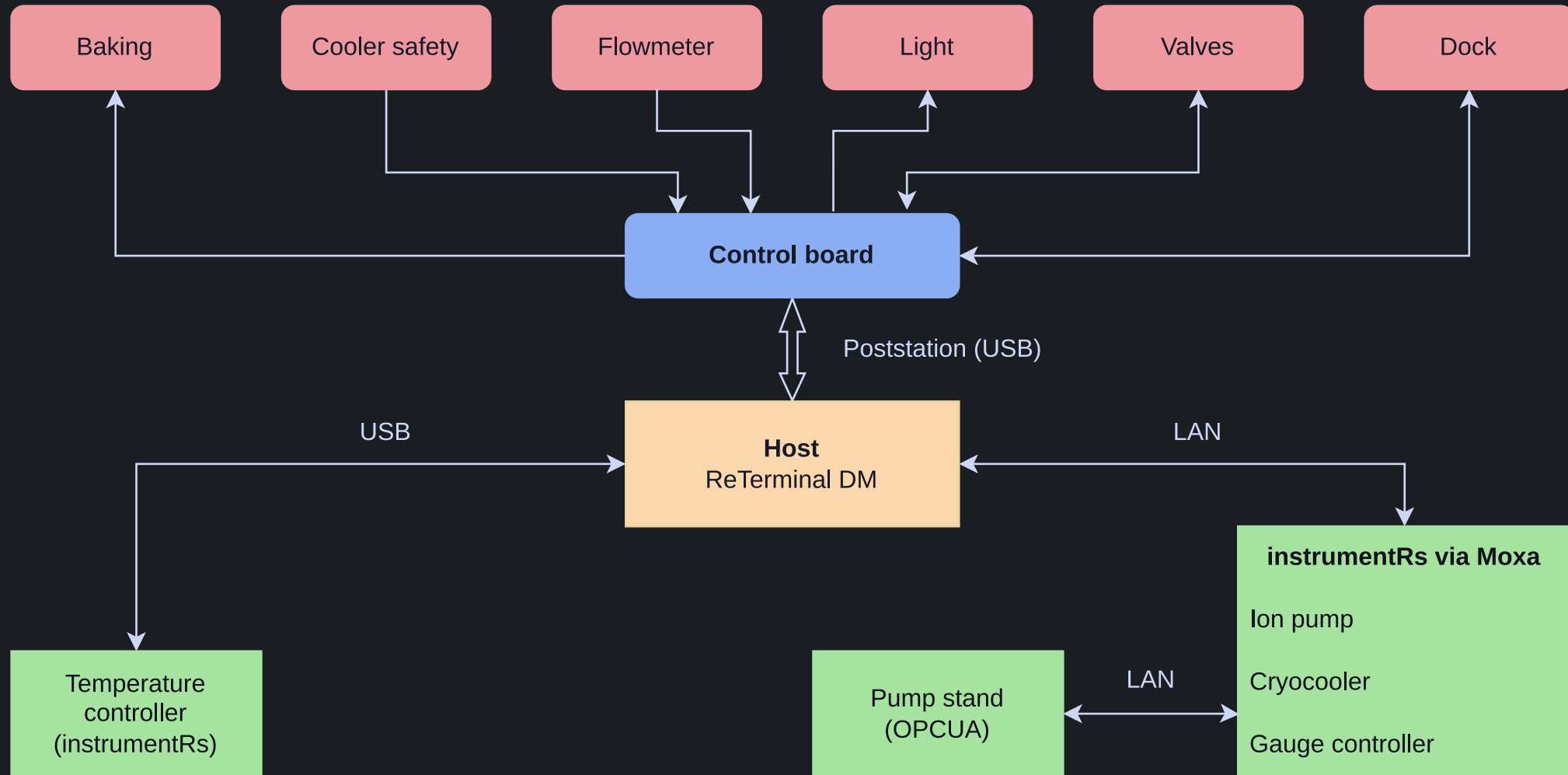
# Cryostorage: Storing samples between analyses



# The components of our home-built storage system

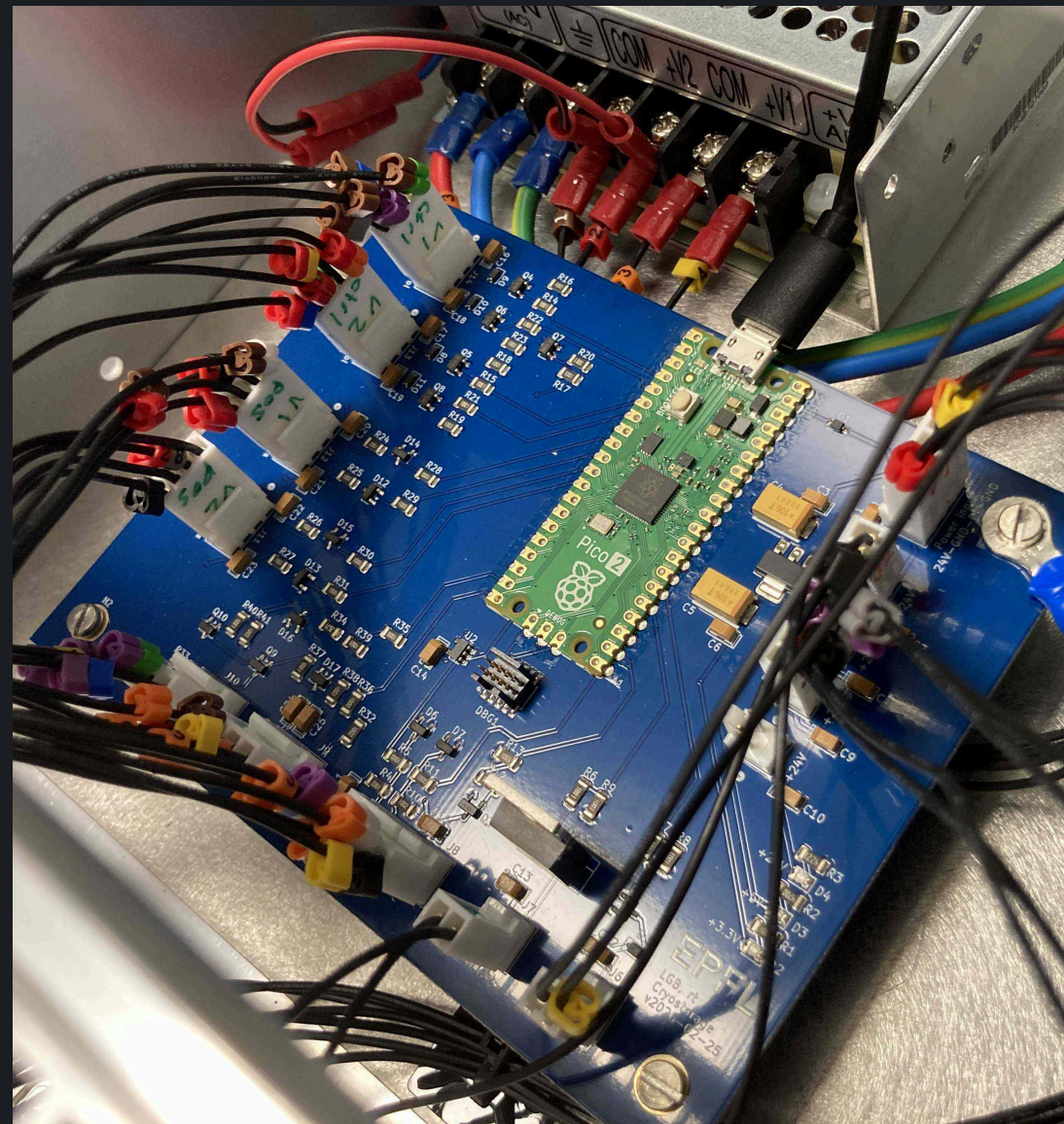


# Using the chamber means controlling all components



# Home-built electronics using RaspberryPi Pico2

- Control lights, valves, baking
- Read flowmeter, valve states
- Rust-based firmware:
  - Async executor `embassy`
  - USB communication with host via `postcard-rpc`
- Hardware safety:
  - Cooler and water flow
  - Transfer valve closing while dock connected is not possible



# Embassy (EMBedded ASync) enables maintainable firmware

- `#![no_std]`
- Supports many MCUs
  - STM32
  - NRF
  - RPi
  - ESP
  - ...
- Provides network and USB stack

```
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    // Initialize the embassy-nrf HAL.
    let p = embassy_nrf::init(Default::default());

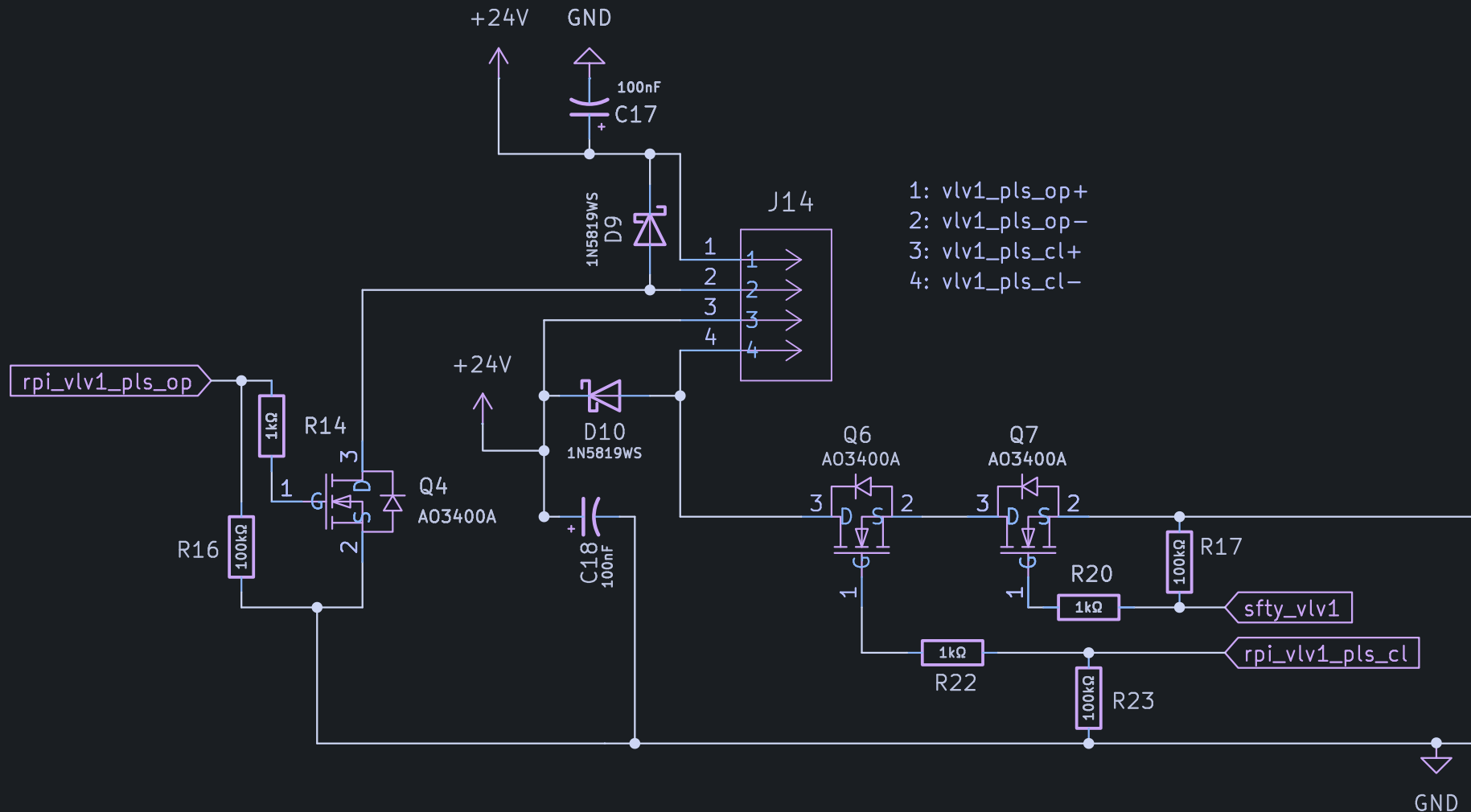
    // Spawned tasks run in the background, concurrently.
    spawner.spawn(blink(p.P0_13.into()).unwrap());

    let mut button = Input::new(p.P0_11, Pull::Up);
    loop {
        // Asynchronously wait for GPIO events
        button.wait_for_low().await;
        info!("Button pressed!");
        button.wait_for_high().await;
        info!("Button released!");
    }
}

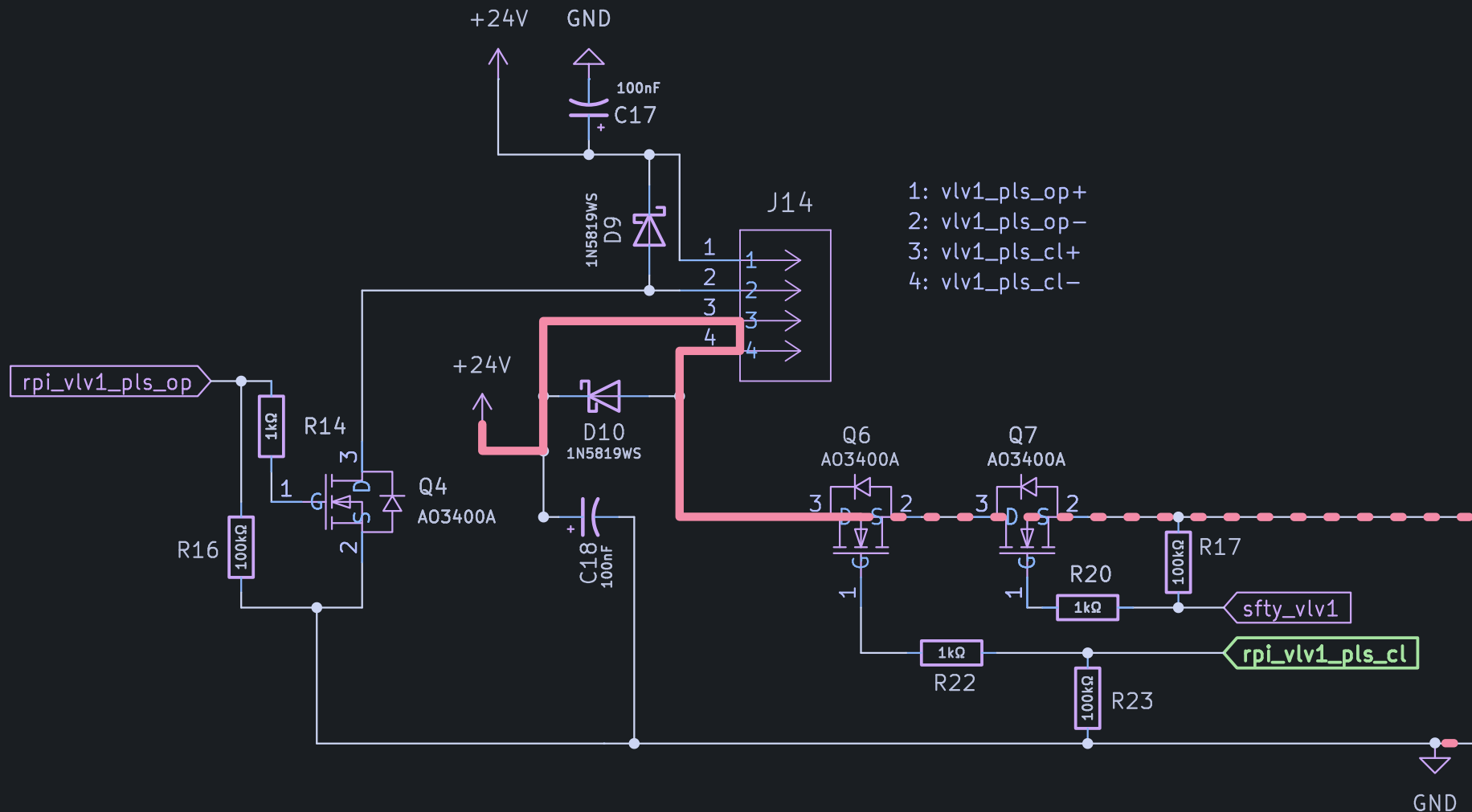
#[embassy_executor::task]
async fn blink(pin: Peri<'static, AnyPin>) {
    let mut led = Output::new(pin, Level::Low, OutputDrive::Standard);

    loop {
        // Timekeeping is globally available, no need to mess with hardware timers.
        led.set_high();
        Timer::after_millis(150).await;
        led.set_low();
        Timer::after_millis(150).await;
    }
}
```

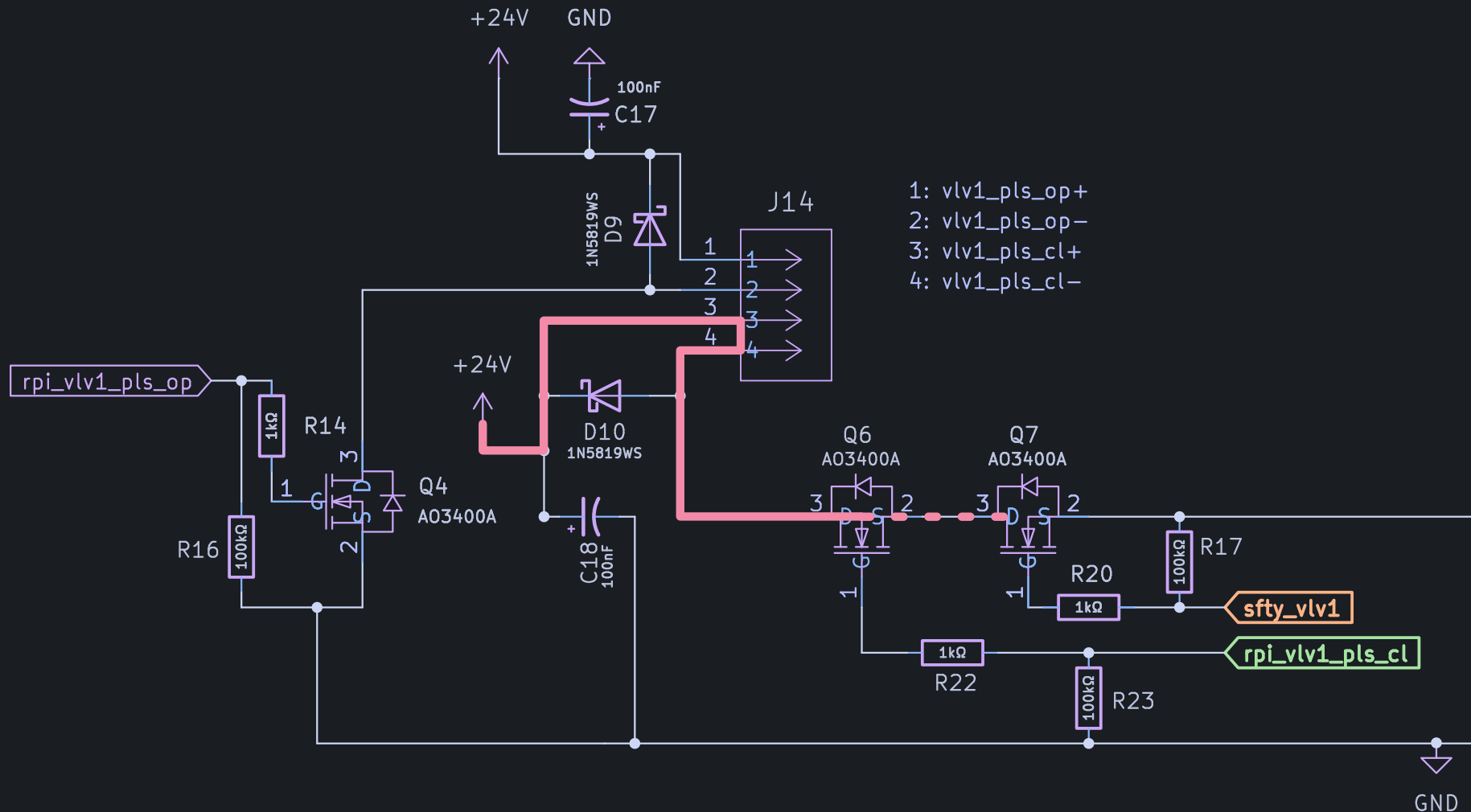
# Hardware safety makes me sleep better at night



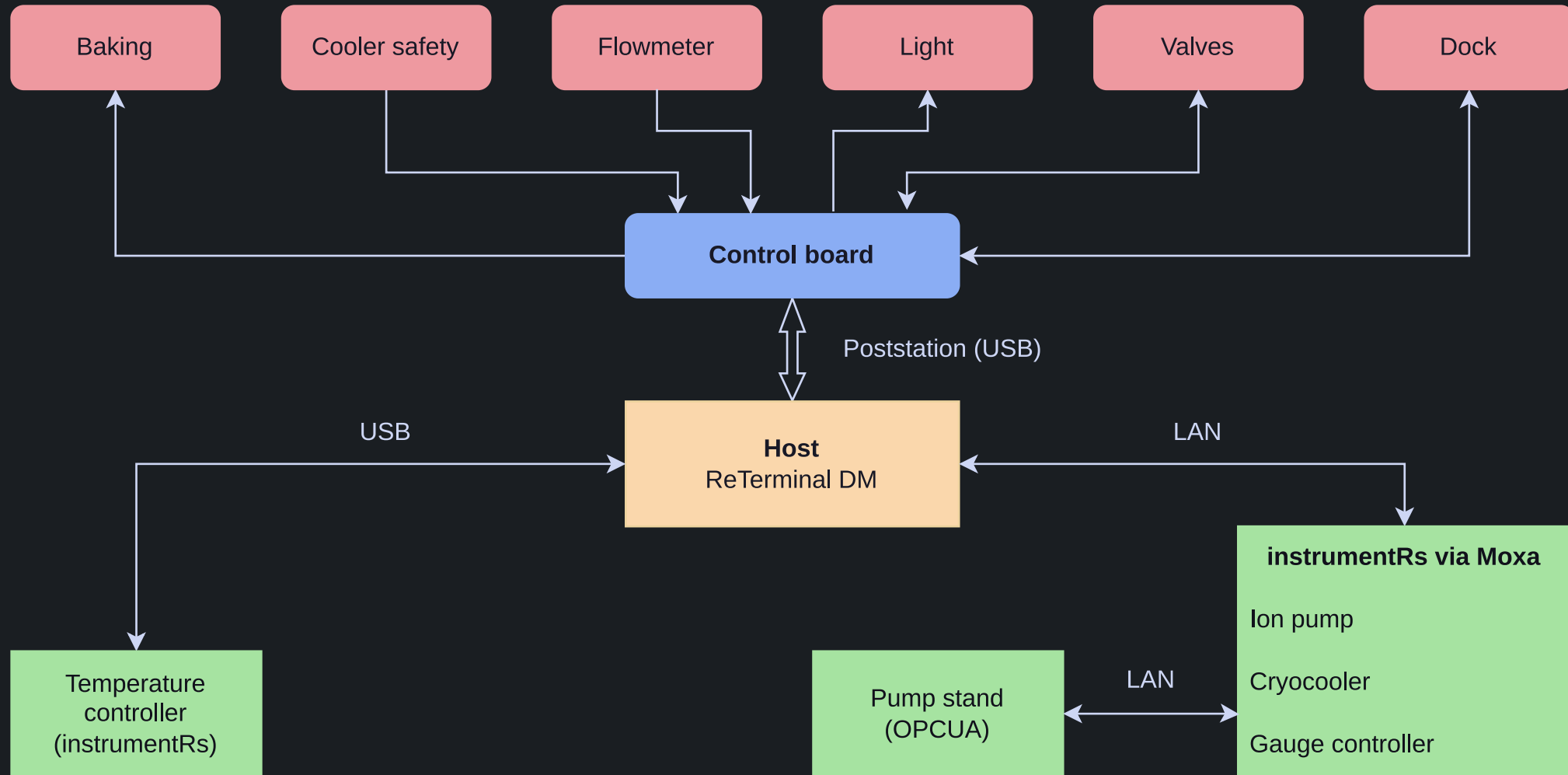
# Closing the valve requires a 24 V pulse controlled by RPi



# Dock connected means Q7 remains open



# What about the connected instruments?



# Talking to the connected instruments

- Serial instruments
  - Mostly connected via Moxa
  - Drivers written with `instrumentRs` framework
  - Regularly polled for status
- Pump stand: `OPCUA` server
  - Event based communication
  - Use existing `opcua` crate



Moxa converts serial ports to TCP/IP interfaces with fixed IPs.

Credit: moxa.com

# Hooking it all together: The host software

- Runs on ReTerminal DM, which is based on a RPi CM4
  - Quad-core Cortex-A72, 1.5GHz
  - Runs RaspberryPi OS
  - UI via 10" touch screen
- Talks to controller via `poststation` server using `postcard-rpc`
- Async host software: `tokio`
  - Listen/poll for changes
  - Execute user commands



ReTerminal DM by SeeedStudio

# A touch interface made with `slint`

- User interface written in `slint`'s own DSL
- Interacts nicely with `tokio`
- Four main screens:
  - Overview of system and samples
  - Temperature info/control
  - Vacuum info/control
  - Settings
- Users can only run workflows!





Sample temperature  
**294 K**

Chamber  
**1.22E-7 mbar**

Transfer  
**4.85E-8 mbar**

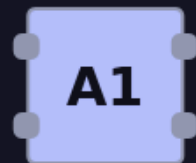


**Start Sample Transfer**

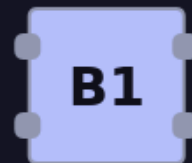
**Vacuum Chamber**

Vent

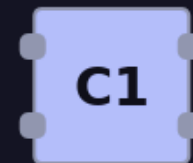
Pump



empty



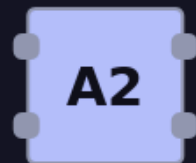
empty



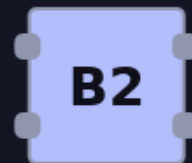
empty



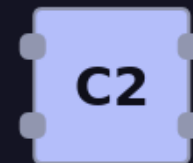
empty



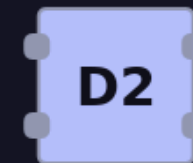
empty



empty



empty



empty





Sample temperature

**294 K**

Bridge temperature

**294 K**

Cooler temperature

**288 K**



**Cryocooler**

Off

On

Water flow



Target temperature

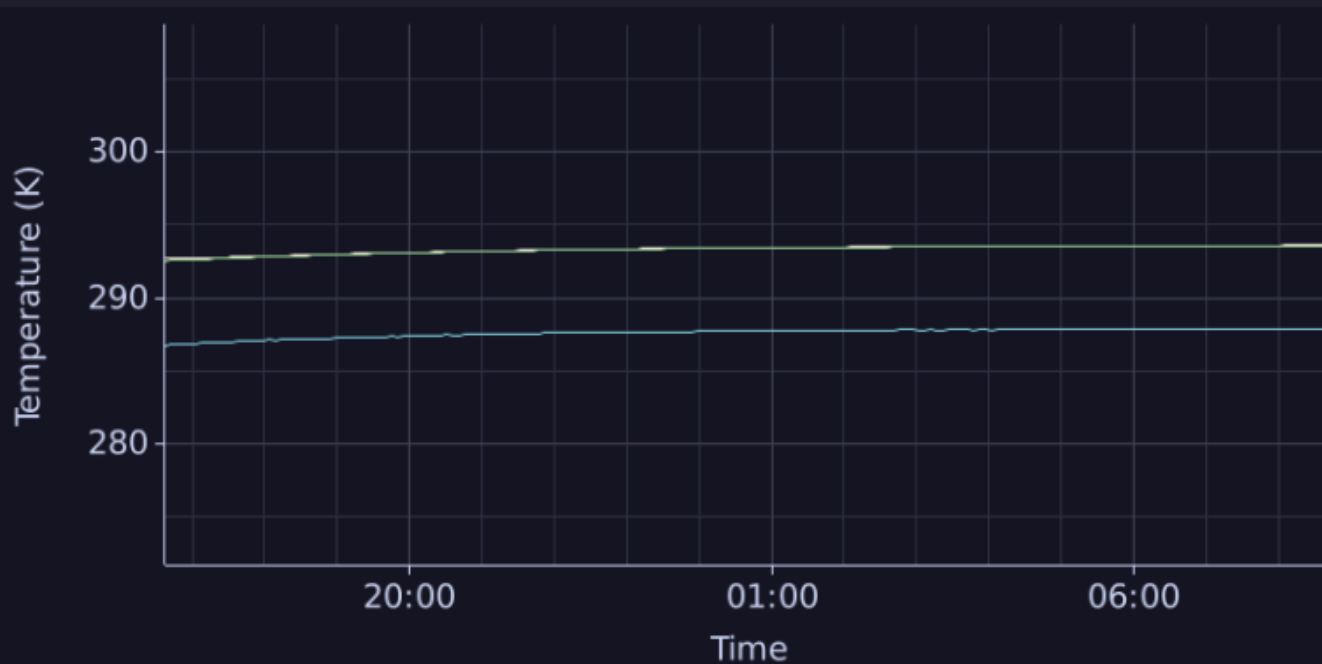
**120 K**

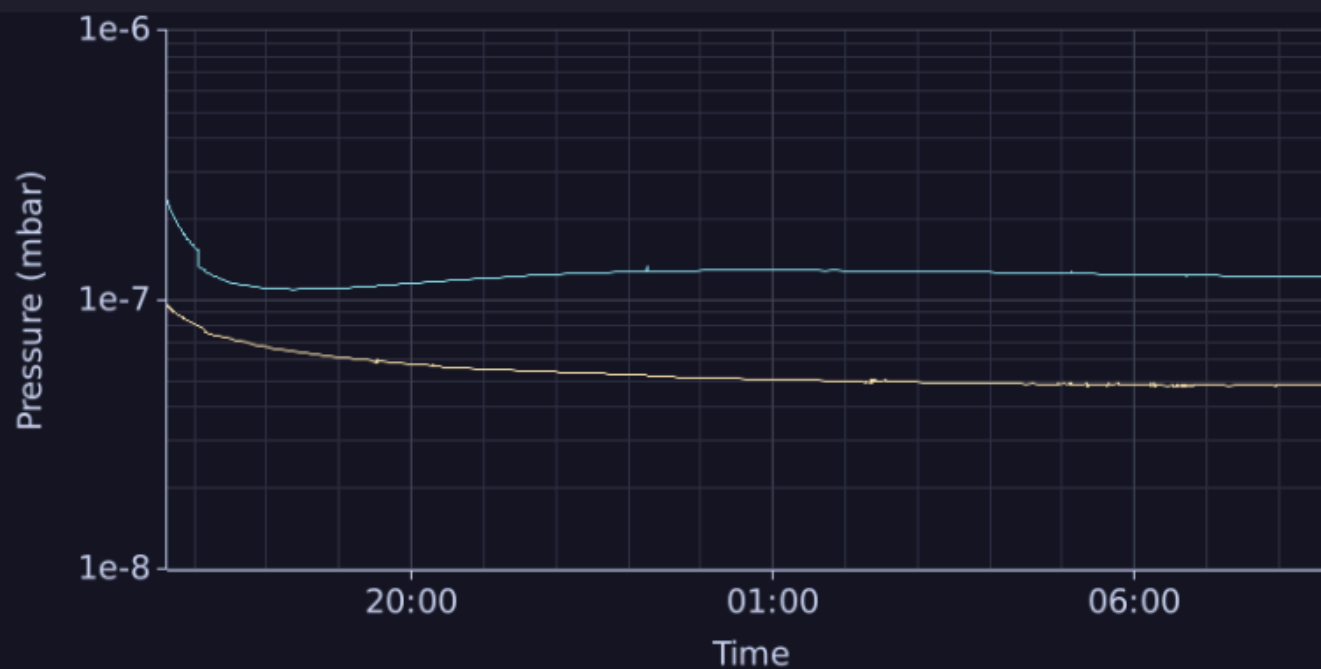
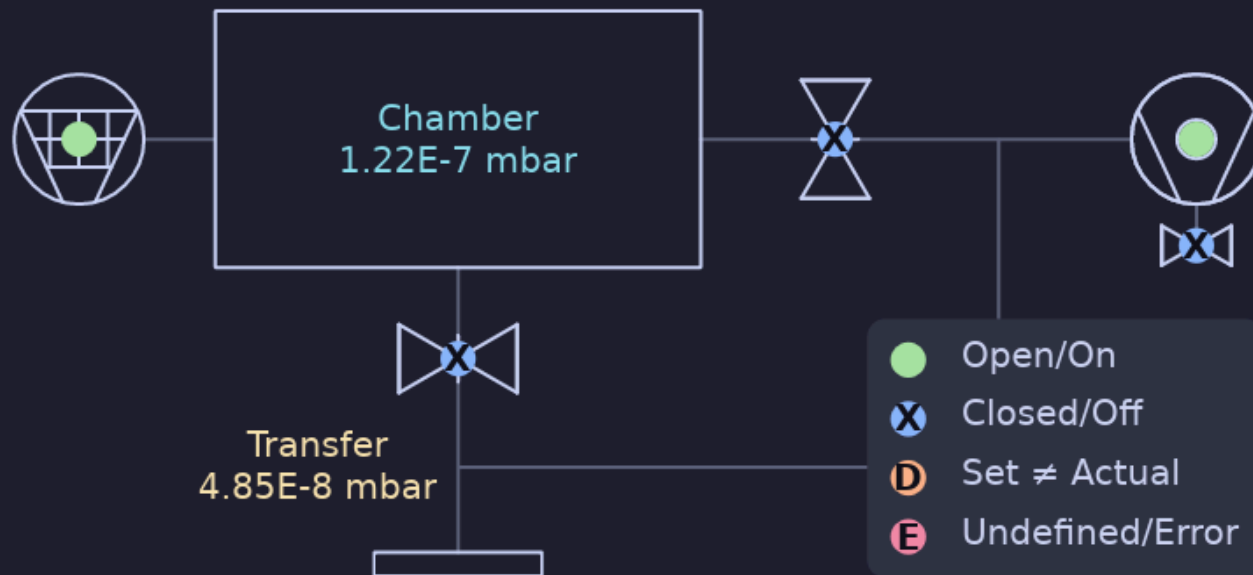
Cooler power

**0 W**



Set target temperature





Pump stand

Off On

Ion pump

Off On

Gauge chamber

Off On

Gauge transfer

Off On

Valve transfer

Close Open

Valve pump

Close Open

Valve vent

Close Open



## Ready for VCT?

No

Yes

Test alarm

Admin mode

Exit program



Log messages:



## Baking

00:00:00

+ d

- h

+ h

-10 min

+10 min

Clear

Stop

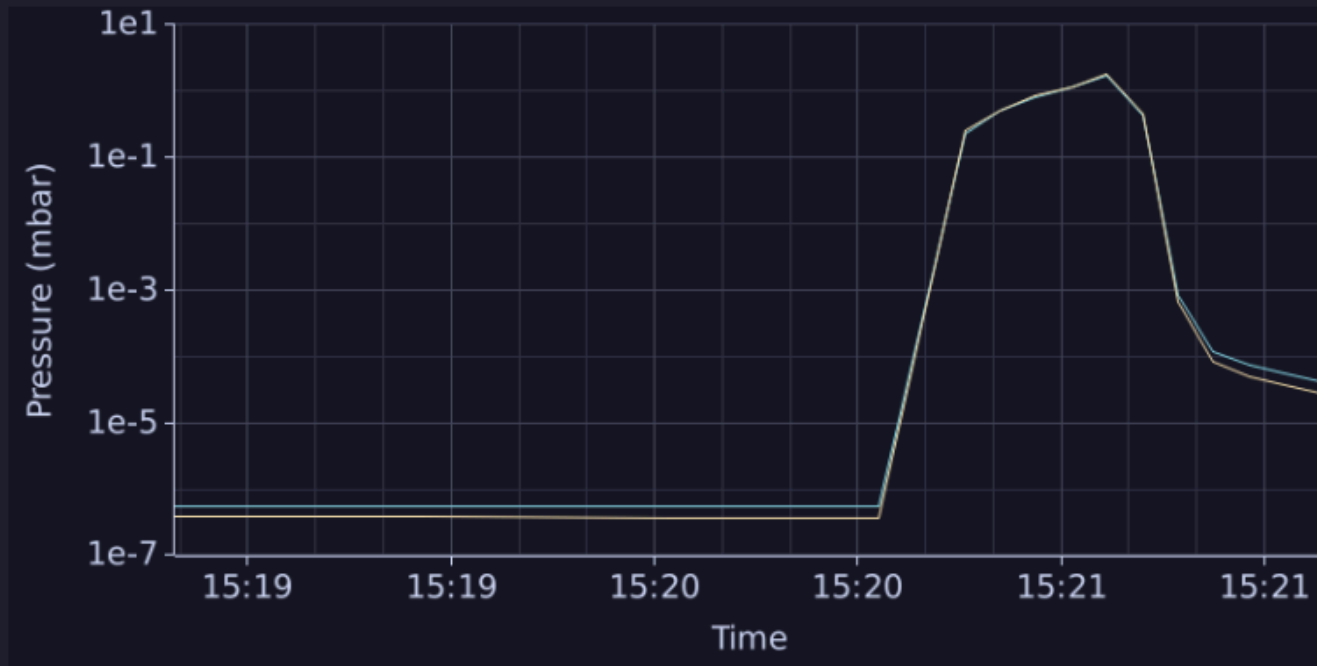
Start

# Pumping the chamber

Pumping the chamber and transfer system. This process can take up to around 3h.

**Waiting for chamber pressure to drop below  $1\text{e-}5$  mbar.**

Remaining time:  
00:39:24



Cancel

# Conclusion: Rust is ideally suited for lab integrations

- Full stack in one language: firmware, comms, and host software
- Type system helps to develop without connected hardware
- Cross-compilation for `aarch64` with `cross-rs`



**Oxidizing the code prevents samples from oxidizing by keeping the cool**

# Do you want to build your own instrument drivers in Rust?

- New rust framework: `instrumentRs` to write instrument drivers
- Inspired by `device-driver` and `instrumentkit`
- Drivers do not depend on interface
- `v0.1.0` developed for cryostorage project
- Preview of `instrumentrs v0.2.0`
  - Specify your instrument in DSL (tbd)
  - Macro derive how new types are transported
  - Traits define the sync (and async) transport

**Get in touch if you are interested!**

**v0.1.0**

- Repo
- Docs

**v0.2.0**

**development**

- Blog series
- Dev repo
- Drivers

# Thank you



**A big thanks to Florent  
Plane and James Munns!**

## Contact

- [reto.trappitsch@epfl.ch](mailto:reto.trappitsch@epfl.ch)
- Matrix
- [blog.galactic-forensics.space](http://blog.galactic-forensics.space)
- Bluesky  
@galactic-forensics.space