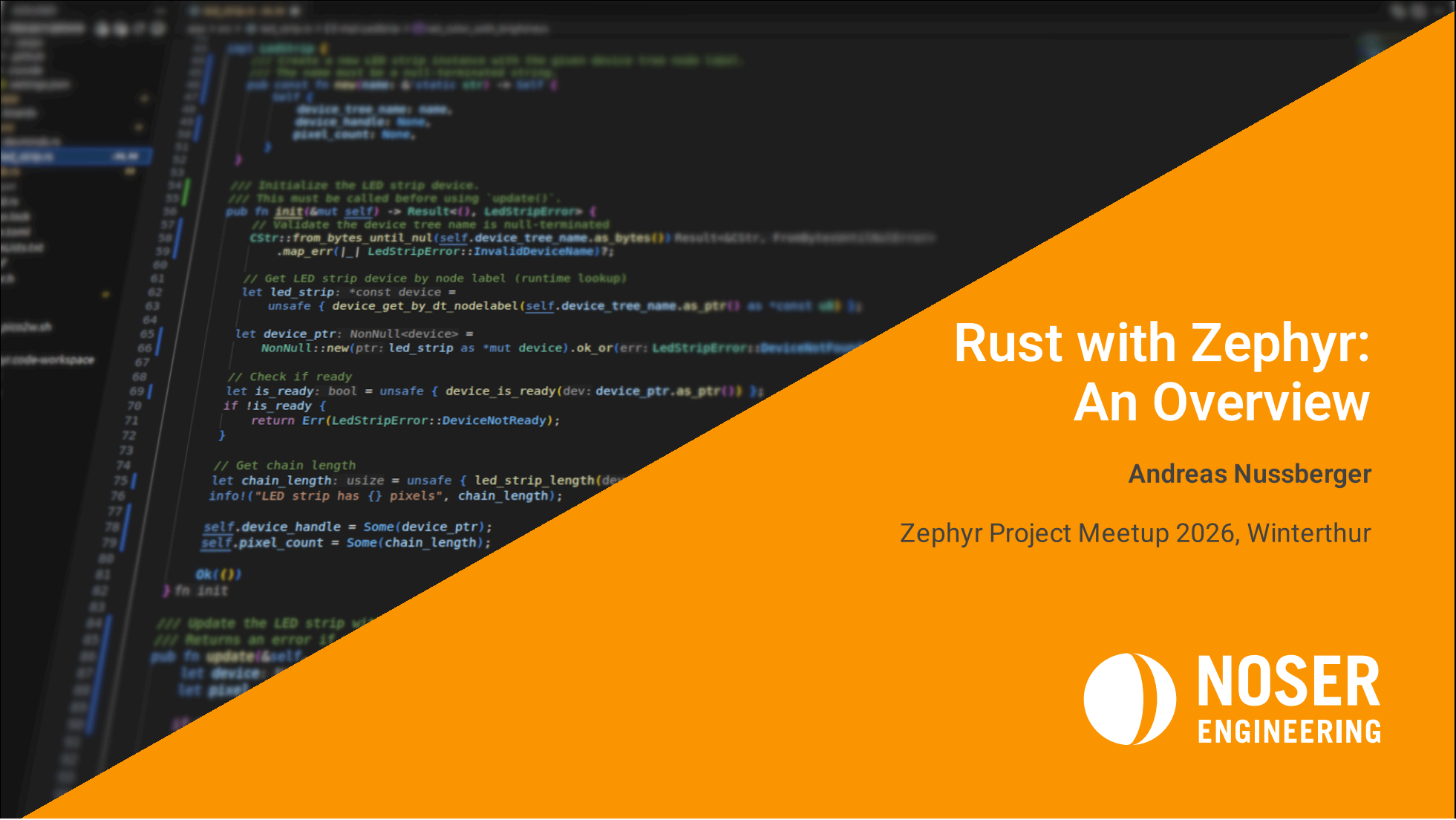


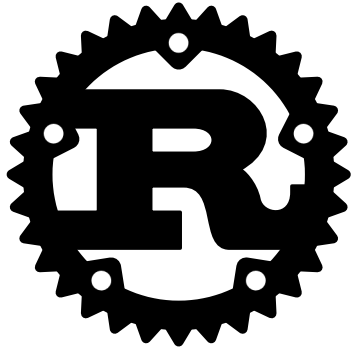
Rust with Zephyr: An Overview

Andreas Nussberger

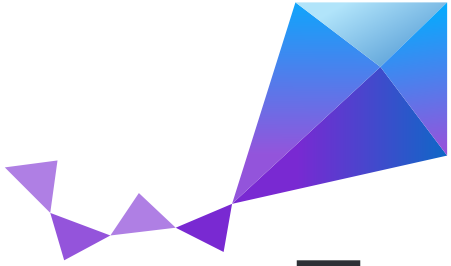
Zephyr Project Meetup 2026, Winterthur

NOSER ENGINEERING

Agenda



- Introduction
- How to use Rust with Zephyr
- How to use C from Rust
- Zephyr, Rust and AI
- Summary



Zephyr®

NOSER ENGINEERING

Andreas Nussberger

andreas.nussberger@noser.com

- **Expert Software Engineer** and **IoT Team Lead**
  at *Noser Engineering AG*
- **Trainer and founder** of *devminds GmbH*
- Education: **Dr. sc. ETH Zürich** (Computer Vision)
- **Highlights**
  - Senior developer in aviation, medical and security industry
  - Leading and pushing research projects in aviation industry
  - PhD thesis: Aerial Object Tracking from an Airborne Platform

Why Zephyr?

because it’s

GREAT

Why Rust?

because it’s

GREAT

Why Rust?

Because:

- **The White House told you so!**
- **Rust support in Linux Kernel is here to stay!**
  - lkml.org/lkml/2025/12/13/38
- **Technical reasons:**
  - Borrow checker
  - `match` syntax
  - Traits
  - Error handling
  - Tooling: toolchains, package management, testing, documentation, …
  - …

FEBRUARY 26, 2024

PRESS RELEASE: Future Software Should Be Memory Safe

ONCD ▸ BRIEFING ROOM ▸ PRESS RELEASE

Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

From Miguel Ojeda <>
Subject [PATCH] rust: conclude the Rust experiment
Date Sat, 13 Dec 2025 01:00:42 +0100

The Rust support was merged in v6.1 into mainline in order to help determine whether Rust as a language was suitable for the kernel, i.e. worth the tradeoffs, technically, procedurally and socially.

At the 2025 Linux Kernel Maintainers Summit, the experiment has just been deemed concluded [1].

• • •

But the experiment is done, i.e. Rust is here to stay.

NOSER ENGINEERING

Rust language support in Zephyr

> Initial support for Rust was added in Zephyr version 4.1.0 (March 7th 2025)!

Official documentation:

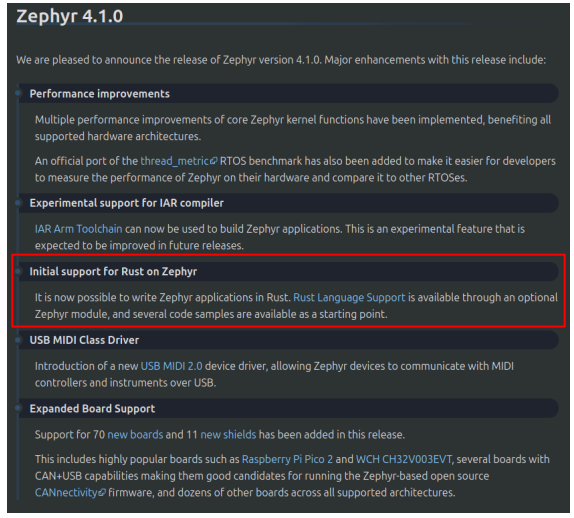
docs.zephyrproject.org/latest/develop/languages/rust/index.html

Zephyr Rust language support on GitHub:

github.com/zephyrproject-rtos/zephyr-lang-rust

Rust in Zephyr overview by David Brown:

youtube.com/watch?v=zAqJbdRZ0eM



Zephyr 4.1.0

We are pleased to announce the release of Zephyr version 4.1.0. Major enhancements with this release include:

Performance improvements

Multiple performance improvements of core Zephyr kernel functions have been implemented, benefiting all supported hardware architectures.

An official port of the thread_metric RTOS benchmark has also been added to make it easier for developers to measure the performance of Zephyr on their hardware and compare it to other RTOSes.

Experimental support for IAR compiler

IAR Arm Toolchain can now be used to build Zephyr applications. This is an experimental feature that is expected to be improved in future releases.

Initial support for Rust on Zephyr

It is now possible to write Zephyr applications in Rust. Rust Language Support is available through an optional Zephyr module, and several code samples are available as a starting point.

USB MIDI Class Driver

Introduction of a new USB MIDI 2.0 device driver, allowing Zephyr devices to communicate with MIDI controllers and instruments over USB.

Expanded Board Support

Support for 70 new boards and 11 new shields has been added in this release.

This includes highly popular boards such as Raspberry Pi Pico 2 and WCH CH32V003EVT, several boards with CAN+USB capabilities making them good candidates for running the Zephyr-based open source CANnectivity firmware, and dozens of other boards across all supported architectures.

Add Rust language support to Zephyr

Rust is an optional module which needs to be enabled:

```
west config manifest.project-filter +zephyr-lang-rust
west update
```

This will create:

`zephyr-workspace/modules/lang/rust`

> ⚠ You might want to pin a more recent hash of the `zephyr-lang-rust` module in your `west.yml` as the Rust GPIO interface, for example, changed significantly!

zephyr-workspace/modules/lang/rust/

```
├── docgen/
├── docs/
├── etc/
├── Kconfig/
├── samples/
├── tests/
├── zephyr/
├── zephyr-build/
├── zephyr-macros/
├── zephyr-sys/
├── ci-manifest.yml
├── CMakeLists.txt
├── LICENSE
├── dt-rust.yaml
├── main.c
└── README.rst
```

Create an initial Rust project in Zephyr

- Follow the official documentation: Rust Language Support
- Check the samples: github.com/zephyrproject-rtos/zephyr-lang-rust/tree/main/samples

Folders and files

- `app` : contains the Rust application
- `lib.rs` : the Rust source code
- `target/` : the Rust build folder
- `Cargo.toml` : the Rust project file
- `CMakeLists.txt` : build interface to Zephyr
- `build/` : the Zephyr build folder

Example structure

zephyr-workspace/zephyr-project/

```
├── app/
│   ├── src
│   │   └── lib.rs
│   ├── target/
│   ├── Cargo.lock
│   ├── Cargo.toml
│   ├── CMakeLists.txt
│   └── prj.conf
├── build/
└── west.yml
```

"Hello world" in Rust with Zephyr

- `#![no_std]`: avoids loading the standard library
  - See: A no_std Rust Environment
- `#[unsafe(no_mangle)]` disables standard symbol name mangling
  - See: The no_mangle attribute
- `extern "C"` publicly exports `rust_main` with C ABI (application binary interface)
  - See: External blocks: ABI
- The `log` module is provided by `zephyr-lang-rust` to call the `info!`, `warning!` or `error!` log macros from Rust

zephyr-workspace/zephyr-project/app/src/lib.rs

```rust
#![no_std]

use log::info;

#[unsafe(no_mangle)]
extern "C" fn rust_main() {
    unsafe {
        zephyr::set_logger().unwrap();
    }

    info!("Hello world from Rust on {}",
          zephyr::kconfig::CONFIG_BOARD);
}
```

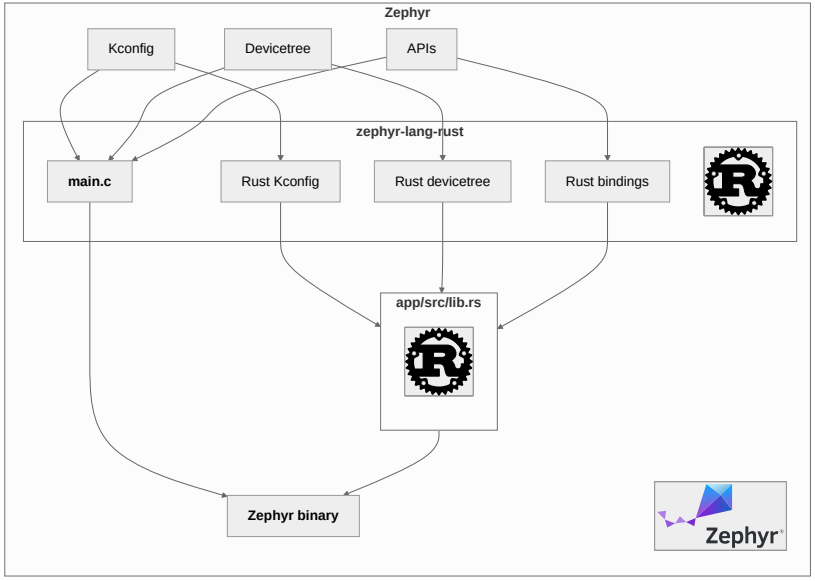
Rust with Zephyr insights

Build workflow

- Zephyr builds are triggered using `west` (CMake)
- CMake triggers `cargo` (Rust package manager)

```
cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(my_app)
rust_cargo_application() # provided by zephyr-lang-rust
```

- The Rust application is compiled as static library
- CMake compiles the `main.c` from `zephyr-lang-rust` and links the static Rust library to create the final binary



Note: in the background Zephyr Kconfig targets have to be mapped to LLVM target triples…

NOSER ENGINEERING

Rust with Zephyr insights

main.c

zephyr-workspace/modules/lang/rust/main.c

```c
#include <zephyr/kernel.h>

#ifdef CONFIG_RUST

extern void rust_main(void);

...

int main(void)
{
  rust_main();
  return 0;
}

...

#endif
```

lib.rs

zephyr-workspace/zephyr-project/app/src/lib.rs

```rust
#![no_std]

use log::info;

#[unsafe(no_mangle)]
extern "C" fn rust_main() {
    unsafe {
        zephyr::set_logger().unwrap();
    }

    info!("Hello world from Rust on {}",
          zephyr::kconfig::CONFIG_BOARD);
}
```

Rust with Zephyr insights

zephyr-lang-rust crates

- **zephyr-build**: exposes *Kconfig* and *Devicetree* to Rust via project `build.rs`:

```
fn main() {
  zephyr_build::dt_cfgs();
}
```

- **zephyr-macros**: Rust macros e.g. for spawning Zephyr threads:

```
#[zephyr::thread(stack_size = 128)]
fn zephyr_rust_thread() {
  // do something...
}
```

- **zephyr-sys**: generates the low-level bindings to the Zephyr API using bindgen

- **zephyr**: contains the actual Rust abstractions for Zephyr features:
  - device/flash
  - device/gpio
  - embassy
  - logging
  - sync
  - sys
  - …

Check the official: API documentation

Selected Zephyr abstractions available in Rust

- `device/flash` : Rust wrappers for flash controllers
- `device/gpio` : Rust wrappers for GPIOs

```rust
use zephyr::raw::ZR_GPIO_OUTPUT_ACTIVE;

#[unsafe(no_mangle)]
extern "C" fn rust_main() {
    let mut led1 =
      zephyr::devicetree::aliases::led1::get_instance()
      .unwrap();

    if !led1.is_ready() {
        loop {}
    }

    led1.configure(ZR_GPIO_OUTPUT_ACTIVE);

    led1.toggle_pin();
}
```

- `embassy` : support for using Embassy (Rust embedded Framework) with Zephyr
- `logging` : logging for Rust

```rust
use log::{error, info, warn};

#[unsafe(no_mangle)]
extern "C" fn rust_main() {
    unsafe {
        zephyr::set_logger().unwrap();
    }

    info!("This is an info message");
    warn!("This is a warning message");
    error!("This is an error message");
}
```

Selected Zephyr abstractions available in Rust

- `sync` : synchronization primitives such as Mutex
- `thread` : Zephyr threads for Rust

```rust
#[unsafe(no_mangle)]
extern "C" fn rust_main() {
    ...
    let blink_thread = run_blinky();
    blink_thread.start();
}

#[cfg(dt = "aliases::led1")]
#[zephyr::thread(stack_size = 128)]
fn run_blinky() {
    warn!("Starting blinky thread...");
    ...
}
```

- `timer` : Zephyr timers for Rust

```rust
#[unsafe(no_mangle)]
extern "C" fn rust_main() {
    ...
    let counter = SpinMutex::new(0);
    let callback = Callback {
        call: |counter: &SpinMutex<u32>| {
            let mut val = counter.lock().unwrap();
            *val += 1;
            info!("Timer fired {} times", *val);
        },
        data: counter,
    };
    let callback_timer = StoppedTimer::new().start_callback(
      callback, NoWait, Duration::millis(10000));
    // Make sure timer stays in scope!
    ...
}
```

- `work` : Zephyr work queues for Rust

NOSER ENGINEERING

Example project

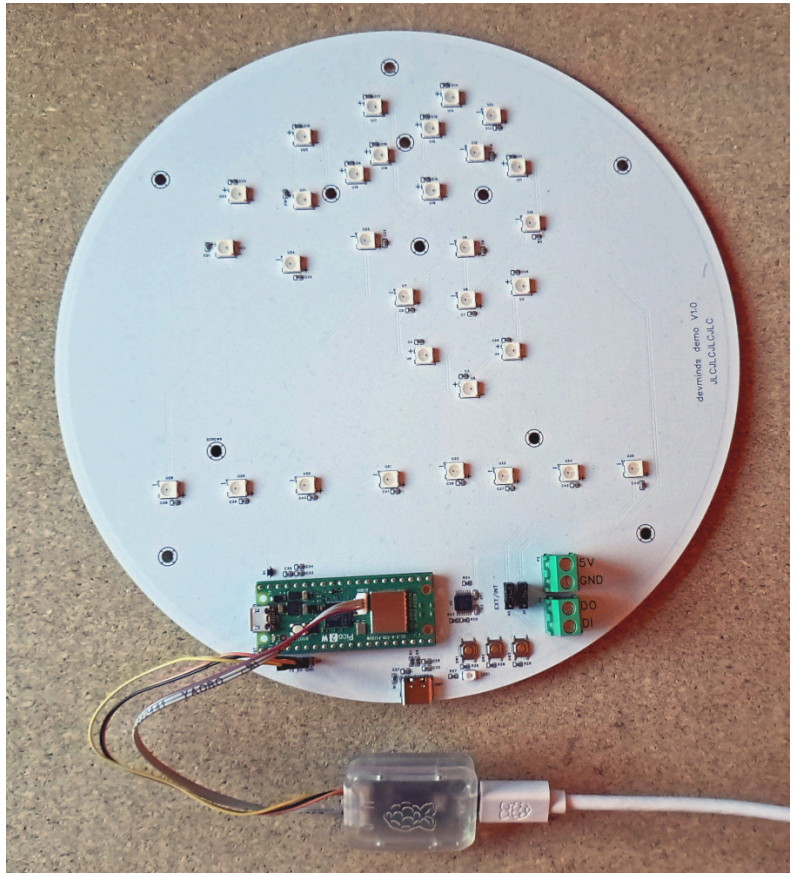
Developed by devminds GmbH for training purposes

Hardware

- Based on Raspberry Pi Pico 2W
- Hardware on development board:
  - Status LED
  - WS2812 compatible LED strip
  - GPIOs

Development environment

- Raspberry Pi Debug Probe
- VSCode
- Debian based Linux as host OS



NOSER ENGINEERING

Get ready for Raspberry Pico 2W development

- Add Rust build target for Pico 2W:

```
rustup target add thumbv8m.main-none-eabi
```

- Setup custom OpenOCD from Raspberry Pi for Pico 2W, e.g. by executing the provided pico_setup.sh:

```
wget -O pico_setup.sh https://raw.githubusercontent.com/raspberrypi/pico-setup/master/pico_setup.sh
chmod +x pico_setup.sh
sudo ./pico_setup.sh
```

- Enable `udev` rules for OpenOCD:

```
sudo cp ZEPHYR_SDK_PATH/sysroots/x86_64-pokysdk-linux/usr/share/openocd/contrib/60-openocd.rules /etc/udev/rules.d
sudo udevadm control --reload
```

- Apply PR #112 to `zephyr-lang-rust` to fix Rust IDE integration. Otherwise, `rust-analyzer` will not work correctly e.g. within VSCode.

Rust blinky

Write Rust code

- Create `rust_main()`
- Get `led1` from device tree
- Configure `led1` as output
- Toggle `led1` in loop

Flash the program

```
west build -p auto \
  -b rpi_pico2/rp2350a/m33/w app
west flash \
  --openocd /usr/local/bin/openocd
```

zephyr-workspace/zephyr-rust-project/rust-app/src/lib.rs

```rust
#![no_std]

use zephyr::raw::ZR_GPIO_OUTPUT_ACTIVE;
use zephyr::time::{sleep, Duration};

#[unsafe(no_mangle)]
extern "C" fn rust_main() {
    let mut led1 =
      zephyr::devicetree::aliases::led1::get_instance().unwrap();
    if !led1.is_ready() {
        loop {}
    }
    led1.configure(ZR_GPIO_OUTPUT_ACTIVE);
    const BLINKY_DELAY: Duration = Duration::millis_at_least(1000);
    loop {
        led1.toggle_pin();
        sleep(BLINKY_DELAY);
    }
}
```

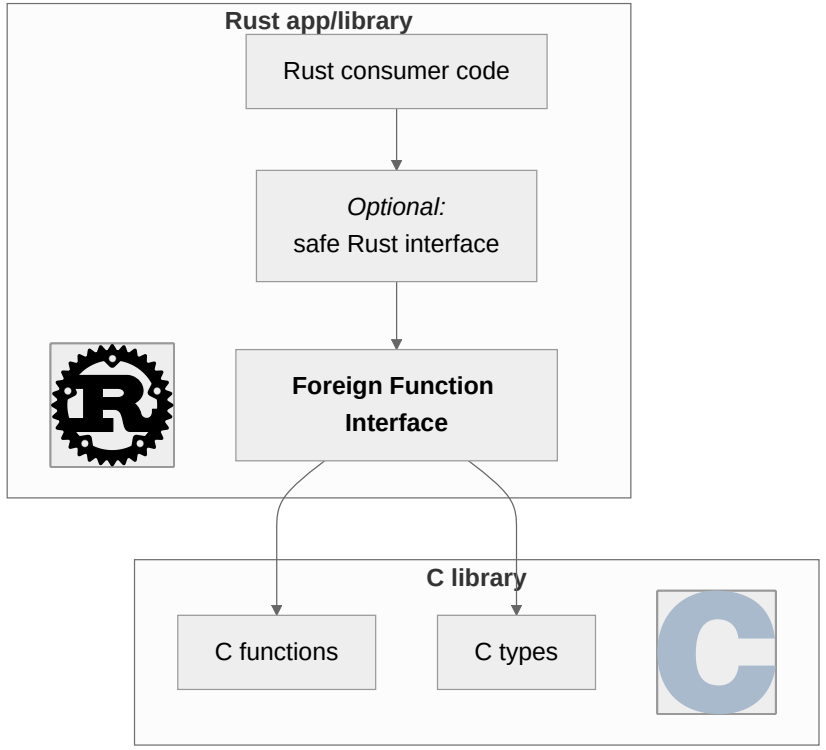

NOSER ENGINEERING

How to use the Zephyr API from Rust?

- **Rust allows to call C code:**
  - A little C with your Rust
  - Interoperability with C
- C allows to call Rust code:
  - A little Rust with your C

Rust allows to call C functions through the:
Foreign Function Interface

We can implement the FFI bindings: **manually**, or **automatically** using bindgen!



NOSER ENGINEERING

Manual Rust FFI bindings

- `c_library_name.h` : header file of the C library to be used
- `manual_bindings.rs` : the manually created FFI bindings for the C library in Rust
- `safe_interface.rs` : optional safe interface to use the C library from Rust

Example usage:

```rust
fn main() {
    let data = b"Hello";
    let result = send(data);
    println!("Send result: {}", result);
}
```

safe_interface.rs

```rust
pub fn send(src: &[u8]) -> bool {
    unsafe {
        // Assumption: 0 = success, else = error
        send(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

manual_bindings.rs

```rust
#[link(name = "c_library_name")]
unsafe extern "C" {
    fn send(buffer: *const u8, size: size_t) -> c_int;
}
```

c_library_name.h

```c
int send(const char* buffer, size_t size);
```

Example: Rust interface for Zephyr LED strips

Getting started

- Check the LED strip Zephyr documentation
- The example project uses `WS2812` compatible LED strips
- Create a `wrapper.h` where required C source code is included:

```
extern int errno; // This is somehow not available ...
#include <zephyr/autoconf.h> // Get Kconfig settings
#include <zephyr/drivers/led_strip.h>
```

- Create Rust bindings for the `led_strip` using `build.rs`, the `wrapper.h` and bindgen

zephyr-workspace/zephyr-project/build.rs

```
fn main() {
    ...
    let bindings = bindgen::Builder::default()
        // The input header
        .header("wrapper.h")
        // We are in a no_std environment
        .use_core()
        // Set target platform
        .clang_arg(format!("--target={}", zephyr_target))
        // Add required Zephyr include paths
        ...
        // Generate bindings for static functions
        .wrap_static_fns(true)
        .wrap_static_fns_path(&bindgen_wrapper_file)
        // Enable led_strip related bindings
        .allowlist_item("led_strip_.*")
        // Enable device related bindings
        .allowlist_function("device_.*")
        // Enable device tree device symbols
        .allowlist_var("__device_dts_ord_.*")
        ...
}
```

Automatically generated bindings using bindgen

build/rust/target/thumbv8m.main-none-eabi/debug/build/rustapp-xxx/out/bindings.rs

```
...
unsafe extern "C" {
    #[doc = " @brief\t\tMandatory function to update an LED strip with the given RGB array.\n\n @param dev\t\tLED str
    #[link_name = "led_strip_update_rgb__extern"]
    pub fn led_strip_update_rgb(
        dev: *const device,
        pixels: *mut led_rgb,
        num_pixels: usize,
    ) -> ::core::ffi::c_int;
}
...
unsafe extern "C" {
    #[doc = " @brief\tMandatory function to get chain length (in pixels) of an LED strip device.\n\n @param dev\tLED
    #[link_name = "led_strip_length__extern"]
    pub fn led_strip_length(dev: *const device) -> usize;
}
...
```

Rust wrapper for Zephyr LED strips

zephyr-workspace/zephyr-project/app/src/led_strip.rs

```rust
pub struct LedStrip {
    device_tree_name: &'static str,
    device_handle: *const device,
    pixel_num: usize,
}

impl LedStrip {
    pub fn new(name: &'static str) -> Self {
        LedStrip {
            device_tree_name: name,
            pixel_num: 0,
            device_handle: core::ptr::null(),
        }
    }

    pub fn init(&mut self) -> Result<(),
      LedStripError> {
       ...
    }
...
```

zephyr-workspace/zephyr-project/app/src/led_strip.rs - continued

```rust
...
    pub fn update(&mut self, pixels: &[LedColor])
      -> Result<(), LedStripError> {
        if self.pixel_num == 0 {
            return Err(LedStripError::NotInitialized);
        }
        if pixels.len() > self.pixel_num {
            return Err(LedStripError::PixelDataTooLarge);
        }
        // Update the LED strip with new pixel data
        let rc = unsafe {
            led_strip_update_rgb(
                self.device_handle,
                pixels.as_ptr() as *mut LedColor,
                pixels.len(),
            )
        };
        match rc {
            0 => Ok(()),
            _ => return Err(LedStripError::UpdateFailed(rc)),
        }
    }
}
```

Using the Zephyr LED strip Rust wrapper

- `LedStrip::new()`: creates a new `LedStrip` device object from the given zero terminated devicetree name
- `.init().unwrap()`: initializes the LED strip and panics on failure
- `.update()`: updates the LED strip with the given pixel value array

Note: some code is omitted for simplicity, e.g. initializing the logger!

zephyr-workspace/zephyr-project/app/src/lib.rs

```rust
#[unsafe(no_mangle)]
extern "C" fn rust_main() {
    ...

    // Initialize LED strip and panic on error
    let mut led_strip = LedStrip::new("led_strip\0");
    led_strip.init().unwrap();

    // Initialize all pixels bright white
    let mut pixels = [LedColor { r: 255, g: 255, b: 255 }; 31];

    // Update LED strip and show potential failures
    if let Err(LedStripError::UpdateFailed(code)) =
      led_strip.update(&pixels) {
        warn!("Update failed with code: {}", code);
    }
}
```

Final Rust application

The resulting application uses:

- **Logging** interface from `zephyr-lang-rust`
- **GPIO** interface from `zephyr-lang-rust`
- Zephyr **threads** from `zephyr-lang-rust`
- Custom `led_strip` module based on:
  - Automatic Rust bindings for the Zephyr `led_strip` C interface generated by 🌐 bindgen
  - Manual Rust wrapper implementation

Zephyr, Rust and AI

Some personal notes about using AI with Zephyr and Rust:

- ⊠ Very useful during debugging build issues and warnings (e.g. device tree problems, etc)
  - Benefit of using Zephyr / Rust: all sources are part of the workspace and therefore accessible by AI
- ⊟ Not that useful in getting Rust working with Zephyr
- ⊞ Useful once Rust ist working with Zephyr to program in Rust

Summary

Zephyr with Rust is a very powerful combination!

Which I would unfortunately not yet recommend for production use …

… therefore:

get your hands on, experiment, and start contributing! 🚀

Thank you for your attention

andreas.nussberger@noser.com

This presentation is powered by **sli.dev**: *Presentation Slides for Developers*