ESPRESSIF

Introduction

⚠️ This material is based on unstable crates. It worked at the time of writing but following it today may not result in compiling code! However, it can be used as inspiration for getting started with Rust on Espressif. You can join the esp-rs community on Matrix for all technical questions and issues! The community is open to everyone.

Content of this material

This is Ferrous Systems' *Embedded Rust on Espressif* training material. It is divided into two workshops: introductory and advanced. The introductory trail will introduce you to basics of embedded development and how to make the embedded board interact with the outside world - reacting to commands and sending sensor data.

The advanced course takes it from there to dive deeper into topics like interrupt handling, low-level peripheral access and writing your own drivers.

The board

An Espressif Rust Board is mandatory[1] for working with this book - emulators like QEMU are not supported.

The board design and images, pin layout and schematics can be also found in this repository.

If you subscribed to one of the trainings, a board will be provided to you directly by Espressif. Some exercises also require wireless internet access.

Our focus lies primarily on the ESP32-C3 platform, a RISC-V based microcontroller with strong IoT capabilities, facilitated by integrated Wi-Fi and Bluetooth 5 (LE) functionality as well as large RAM + flash size for sophisticated applications. A substantial amount of this course is also applicable for Xtensa the other architecture Espressif uses, in particular the ESP32-S3. For low-level access the general principles apply as well, but actual hardware access will differ in various ways - refer to the technical reference manuals (C3, S3) or other available technical documents as needed.

Rust knowledge

Basic Rust like The Rust Book Chapters 1 - 6, Chapter 4 Ownership does not need to be fully understood.

[1] It is possible to follow the intro part with ESP32-C3-DevKitC-02 but we do not recommend it. It is inherently easier to follow the training when using the same hardware.

Preparations

This chapter contains informations about the course material, the required hardware and an installation guide.

Icons and Formatting we use

We use Icons to mark different kinds of information in the book:

- ✅ Call for action
- ❗ Warnings, Details that require special attention
- 🔎 Knowledge that dives deeper into a subject, but which you are not required to understand to proceed.
- 💬 Descriptions for Accessibility

> Example note: Notes like this one contain helpful information

Required Hardware

- Rust ESP Board, available on Mouser, Aliexpress. Full list of vendors.
- USB-C cable suitable to connect the board to your development computer
- Wi-Fi access point connected to the Internet

No additional debugger/probe hardware is required.

Ensuring a working setup

❗ As of March 2022 we are not providing complete setup instructions for MS Windows.

❗ If you are participating in a training led by Ferrous Systems, we urge you to do prepare for the workshop by following the instructions in this chapter least one business day in advance to verify you're ready to go by the time it starts. Please contact us should you encounter any issues or require any kind of support.

❗ If you are using a ESP32-C3-DevKitC-02 a few pins and slave adresses are different, since the board is similar but not the same. This is relevant for the solutions in advanced/i2c-sensor-reading/ and advanced/i2c-driver/, where the pins and slave addresses for the ESP32-C3-DevKitC-02 are commented out.

Companion material

- Official esp-rs book

Checking the hardware

Connect the Espressif Rust Board to your computer. Verify a tiny red control LED lights up.

The device should also expose its UART serial port over USB:

Windows: a USB Serial Device (COM port) in the Device Manager under the Ports section

Linux: a USB device under `lsusb`. The device will have a VID (vendor ID) of `303a` and a PID (product ID) of `1001` -- the `0x` prefix will be omitted in the output of `lsusb`:

```
$ lsusb | grep USB
Bus 006 Device 035: ID 303a:1001 Espressif USB JTAG/serial debug unit
```

Another way to see the device is to see which permissions and port is associated to the device is to check the `/by-id` folder:

```
$ ls -l /dev/serial/by-id
lrwxrwxrwx 1 root root .... usb-
Espressif_USB_JTAG_serial_debug_unit_60:55:F9:C0:27:18-if00 -> ../../ttyACM0

(If you are using a ESP32-C3-DevKitC-02 the command is `$ ls /dev/ttyUSB*` )
```

macOS: The device will show up as part of the USB tree in `system_profiler`:

```
$ system_profiler SPUSBDataType | grep -A 11 "USB JTAG"

USB JTAG/serial debug unit:

  Product ID: 0x1001
  Vendor ID: 0x303a
  (...)
```

The device will also show up in the `/dev` directory as a `tty.usbmodem` device:

```
$ ls /dev/tty.usbmodem*
/dev/tty.usbmodem0
```

Software

Follow the steps below for a default installation of the ESP32-C3 platform tooling.

🔎 Should you desire a customized installation (e.g. building parts from source, or add support for Xtensa/ESP32-S3), instructions for doing so can be found in the Installing Rust chapter of the *Rust on ESP* Book.

Rust toolchain

✅ If you haven't got Rust on your computer, obtain it via https://rustup.rs/

Furthermore, for ESP32-C3, a specific *nightly* version of the Rust toolchain is currently required.

✅ Install nightly Rust and add support for the target architecture using the following console commands:

```
$ rustup install nightly-2022-03-10
$ rustup component add rust-src --toolchain nightly-2022-03-10
```

🔎 Rust is capable of cross-compiling to any supported target (see `rustup target list`). By default, only the native architecture of your system is installed. To build for the Xtensa architecture (*not* part of this material), a fork of the Rust compiler is required as of January 2022.

Espressif toolchain

Several tools are required:

- `cargo-espflash` - upload firmware to the microcontroller
- `ldproxy` - Espressif build toolchain dependency

✅ Install them with the following command:

```
$ cargo install cargo-espflash ldproxy
```

Toolchain dependencies

Debian/Ubuntu

```
$ sudo apt install llvm-dev libclang-dev clang
```

macOS

(when using the Homebrew package manager, which we recommend)

```
$ brew install llvm
```

Troubleshooting

- Python 3 is a required dependency. It comes preinstalled on stock macOS and typically on desktop Linux distributions. An existing **Python 2** installation with the `virtualenv` add-on pointing to it is known to potentially cause build problems.
- Error `failed to run custom build command for libudev-sys v0.1.4` or `esp-idf-sys v0.30.X`:

  At time of writing, this can be solved by

  1. running this line from the `esp-rs` container:

  `apt-get update \ && apt-get install -y vim nano git curl gcc ninja-build cmake libudev-dev python3 python3-pip libusb-1.0-0 libssl-dev \ pkg-config libtinfo5`

  2. restarting the terminal
  3. If this is not working, try `cargo clean`, remove the `~/.espressif` folder and reinstall according to esp instructions.

  ⚠️ In step 2, do not clone the `https://github.com/espressif/esp-idf.git` repository. For this training, we are using a git tag.

  Instead, do the following:

```
git clone --recursive --depth 1 --shallow-submodules
git@github.com:espressif/esp-idf.git --branch "v4.4.1" esp-idf-v4.4
cd esp-idf-v4.4
./install.sh esp32c3
. ./export.sh
```

If you change terminal, you will need to source the `export.sh` file:

```
source ~/esp/esp-idf-v4.4/export.sh
```

4. On Ubuntu, you might need to change your kernel to `5.19`. Run `uname -r` to obtain your kernel version.

Docker

❗ Please **note** the Docker container provides an alternative option to **compile** the Rust exercises in. It is meant for users that have experience with virtualized environments. Be aware that we cannot provide help for Docker specific issues during the training.

An alternative environment to **compile** the Rust exercises in is to use Docker. In this repository there is a `Dockerfile` with instructions to install the Rust toolchain & all required packages. This virtualized environment is designed to only compile the binaries for the espressif target. Other commands, e.g. using `cargo-espflash`, still need to be executed on the host system.

✅ Install `Docker` for your operating system.

To build the Docker image run the following command from the root folder:

```
$ docker image build --tag esp --file .devcontainer/Dockerfile .
```

Building the image takes a while depending on the OS & hardware (20-30 minutes).

To start the new Docker container run:

```
$ docker run --mount
type=bind,source="$(pwd)",target=/workspace,consistency=cached -it esp /bin/bash
```

This starts an interactive shell in the Docker container. It also mounts the local repository to a folder named `/workspace` inside the container. Changes to the project on the host system are reflected inside the container & vice versa.

Using this Docker setup requires certain commands to run inside the container, while other have to be executed on the host system. It's recommended to keep two terminals open, one connected to the Docker container, one on the host system.

- in the container: compile the project
- on the host: use the `cargo-espflash` sub-command to flash the program onto the embedded hardware

Additional Software

VS Code

One editor with good Rust support is VS Code which is available for most platforms. When using VS Code we recommend the following extensions to help during the development.

- `Even Better TOML` for editing TOML based configuration files
- `Rust Analyzer` to provide code completion & navigation

There are a few more useful extensions for advanced usage

- `lldb` a native debugger extension based on LLDB
- `crates` to help manage Rust dependencies

VS Code & Devcontainer

One extension for VS Code that might be helpful to develop inside a Docker container is `Remote Containers`. It uses the same `Dockerfile` as the Docker setup, but builds the image and connects to it from within VS Code. Once the extension is installed VS Code recognizes the configuration in the `.devcontainer` folder. Use the `Remote Containers - Reopen in Container` command to connect VS Code to the container.

Workshop repository

The entire material can be found at https://github.com/ferrous-systems/espressif-trainings.

✅ Clone and change into the workshop repository:

```
$ git clone "https://github.com/ferrous-systems/espressif-trainings.git"
$ cd espressif-trainings
```

❗ Windows users may have problems with long path names. Follow these steps to substitute the path:

```
git clone https://github.com/ferrous-systems/espressif-trainings.git
subst r:\ espressif-trainings
cd r:\
```

Repository contents

- `advanced/` - code examples and exercises for the advanced course
- `book/` - markdown sources of this book
- `common/` - code shared between both courses
- `common/lib/` - support crates
- `common/lib/esp32-c3-dkc02-bsc` - board support crate (bsc) for the `ESP32-C3-DevKitC-02` board
- `common/vendor/` - third party crates that have been forked to add required support, pending upstream merges
- `extra/` - tools not required for this training which might still be useful
- `intro/` - code examples and exercises for the introduction course

A word on configuration

We use toml-cfg throughout this workshop as a more convenient and secure alternative to putting credentials or other sensitive information directly in source code: the settings are stored in a file called `cfg.toml` in the respective package root instead

This configuration contains exactly one section header which has the same name as your package (`name = "your-package"` in `Cargo.toml`), and the concrete settings will differ between projects:

```toml
[your-package]
user = "example"
password = "h4ckm3"
```

If you copy a `cfg.toml` to a new project, remember to change the header to `[name-of-new-package]`.

Hello, board!

You're now ready to do a consistency check.

✅ Connect the USB-C port of the board to your computer and enter the hardware check directory in the workshop repository:

```
espressif-trainings$ cd intro/hardware-check
```

To test Wi-Fi connectivity, you will have to provide your network name (SSID) and password (PSK). These credentials are stored in a dedicated `cfg.toml` file (which is `.gitignore` d) to prevent accidental disclosure by sharing source code or doing pull requests. An example is provided.

✅ Copy `cfg.toml.example` to `cfg.toml` (in the same directory) and edit it to reflect your actual credentials:

❗ The 5GHz band is not supported according to ESP32-C3 documentation, you need to ensure you are using a WiFi with active 2.4GHz band.

```
$ cp cfg.toml.example cfg.toml
$ $EDITOR cfg.toml
$ cat cfg.toml

[hardware-check]
wifi_ssid = "Your Wifi name"
wifi_psk = "Your Wifi password"
```

✅ Build, run and monitor the project, substituting the actual serial device name for `/dev/SERIAL_DEVICE`:

```
$ cargo espflash --release --monitor /dev/SERIAL_DEVICE

Serial port: /dev/SERIAL_DEVICE
Connecting...

Chip type:         ESP32-C3 (revision 3)
(...)
Compiling hardware-check v0.1.0
Finished release [optimized] target(s) in 1.78s

[00:00:45] ########################################     418/418     segment
0x10000

Flashing has completed!
(...)
rst:0x1 (POWERON),boot:0xc (SPI_FAST_FLASH_BOOT)
(...)
(...)
(...)
I (4427) bsc::wifi: Wifi connected!
```

The board LED should turn yellow on startup, and then, depending on whether a Wifi connection could be established, either turn red (error) or blink, alternating green and blue. In case of a Wifi error, a diagnostic message will also show up at the bottom, e.g.:

```
Error: could not connect to Wi-Fi network: ESP_ERR_TIMEOUT
```

Extra information about building, flashing and monitoring

If you want to try to build without flashing, you can run:

```
cargo build --target riscv32imc-esp-espidf
```

This can save a lot of time as you do not need to re-flash the program in its entirety and flashing can take up quit some time.

If `cargo espflash --release --monitor /dev/YOUR_SERIAL_DEVICE` has been successful, you can exit with `ctrl+C`, and run the monitor the device without flashing anew with the following command:

```
espmonitor /dev/YOUR_SERIAL_DEVICE
```

Troubleshooting

Build errors

```
error[E0463]: can't find crate for `core`
= note: the `riscv32imc-esp-espidf` target may not be installed
```

You're trying to build with a `stable` Rust - you need to use `nightly`. this error message is slightly misleading - this target *cannot* be installed. It needs to be built from source, using `build-std`, which is a feature available on nightly only.

```
error: cannot find macro `llvm_asm` in this scope
```

You're using an incompatible version of nightly - configure a suitable one using `rust-toolchain.toml` or `cargo override`.

```
CMake Error at .../Modules/CMakeDetermineSystem.cmake:129 (message):
```

Your Espressif toolchain installation might be damaged. Delete it and rerun the build to trigger a fresh download:

```
$ rm -rf ~/.espressif
```

```
Serial port: /dev/tty.usbserial-110
Connecting...

Unable to connect, retrying with extra delay...
Unable to connect, retrying with default delay...
Unable to connect, retrying with extra delay...
Error: espflash::connection_failed

× Error while connecting to device
╰─▶ Failed to connect to the device
help: Ensure that the device is connected and the reset and boot pins are not
being held down
```

The board is not accessible with USB-C cable. A typical connection error looks like this:

Workarounds:

1. press and hold boot button on the board, start flash command, release boot button after flashing process starts
2. use a hub.

Source.

Connecting to Wifi

- You will get an `ESP_ERR_TIMEOUT` error also in case your network name or password are incorrect, so double-check those.

Intro Workshop

Project organization

The esp-rs crates

Unlike most other embedded platforms, Espressif supports the Rust standard library. Most notably this means you'll have arbitrary-sized collections like `Vec` or `HashMap` at your disposal, as well as generic heap storage using `Box`. You're also free to spawn new threads, and use synchronization primitives like `Arc` and `Mutex` to safely share data between them. Still, memory is a scarce resource on embedded systems, and so you need to take care not to run out of it - threads in particular can become rather expensive.

Services like WiFi, HTTP client/server, MQTT, OTA updates, logging etc. are exposed via Espressif's open source IoT Development Framework, esp-idf. It is mostly written in C and as such is exposed to Rust in the canonical split crate style:

- a `sys` crate to provide the actual `unsafe` bindings (esp-idf-sys)
- a higher level crate offering safe and comfortable Rust abstractions (esp-idf-svc)

The final piece of the puzzle is low-level hardware access, which is again provided in a split fashion:

- esp-idf-hal implements the hardware-independent embedded-hal traits like analog/digital conversion, digital I/O pins, or SPI communication - as the name suggests, it also uses `esp-idf` as a foundation
- if direct register manipulation is required, esp32c3 provides the peripheral access create generated by svd2rust.

More information is available in the ecosystem chapter of the `esp-rs` book.

Build toolchain

🔎 As part of a project build, `esp-idf-sys` will download esp-idf, the C-based Espressif toolchain. The download destination is configurable; to save disk space and download time, all examples/exercises in the workshop repository are set to use one single `global` toolchain, installed in `~/.espressif`. See the `ESP_IDF_TOOLS_INSTALL_DIR` parameter in `esp-idf-sys`'s README for other options.

Package layout

On top of the usual contents of a Rust project created with `cargo new`, a few additional files and parameters are required. The examples/exercises in this workshop are already fully configured, and for creating new projects it is recommended to use the cargo-generate wizard based approach.

🔎 The rest of this page is optional knowledge that can come in handy should you wish to change some aspects of a project.

`Cargo.toml`

This workshop is written around the `native` build system. Alternatively you may use `PlatformIO` / `pio`, however this is currently being deprecated.

```
[features]
default = ["native"]
native = ["esp-idf-sys/native"]
```

Some build dependencies must be set:

```
[build-dependencies]
embuild = "0.28"
anyhow = "1"
```

Additional configuration files

- `build.rs` - Cargo build script. Here: sets environment variables required for building.
- `.cargo/config.toml` - sets the target architecture and controls build details. This is the place to override `ESP_IDF_TOOLS_INSTALL_DIR` if you wish to do so.
- `sdkconfig.defaults` - overrides `esp-idf` specific parameters such as stack size or log level.

Generating new projects

We're now going to use `cargo-generate` (a generic project wizard) to set up our first application.

Most other exercises in this workshop already provide a project skeleton and don't require using `cargo-generate`.

✅ Change to the `intro` directory and run `cargo generate` with the `esp-idf` template:

```
$ cd intro
$ cargo generate --git https://github.com/esp-rs/esp-idf-template cargo
```

You'll be prompted for details regarding your new project. When given a choice between several options, navigate using cursor up/down and select with the Return key.

The first message you see will be: `⚠Unable to load config file: /home/$USER/.cargo/cargo-generate.toml`. You see this error because you do not have a favorite config file, but you don't need one and you can ignore this warning.

🔎 You can create a favorite config file that will be placed in `$CARGO_HOME/cargo-generate`, and override it with `-c, --config <config-file>`.

If you make a mistake, hit `Ctrl+C` and start anew.

✅ Configure your project:

(These items may appear in a different order)

- Project Name: `hello-world`
- Rust toolchain: `nightly`
- MCU: `esp32c3`
- ESP-IDF native build version: `4.4`
- STD support: `true`

We're going to build using the `native` variant of the Espressif build system.

✅ Enable the native build system by opening `Cargo.toml` in your new `hello-world` project and adding `"native"` as default feature:

```
[features]
default = ["native"] # add this line
native = ["esp-idf-sys/native"]
```

🔎 `.cargo/config.toml` contains local settings (list of all settings) for your package. `Cargo.toml` contains dependencies import all your dependencies.

Optional, but recommended: To save disk space and download time, set the toolchain directory to global - otherwise each new project/workspace will have its own instance of the toolchain installed on your computer.

✅ Open `hello-world/.cargo/config.toml` and add the following line to the bottom of the `[env]` section. Leave everything else unchanged.

```
[env]
# ...
ESP_IDF_TOOLS_INSTALL_DIR = { value = "global" } # add this line
```

✅ Open `hello-world/rust-toolchain.toml` and change the file to look like this:

```
[toolchain]

channel = "nightly-2022-03-10" # change this line
```

✅ Run your project by using the following command out of the `hello_world` directory.

```
$ cd hello-world
$ cargo espflash --release --monitor /dev/SERIAL_DEVICE
```

✅ The last lines of your output should look like this:

```
(...)
I (268) cpu_start: Starting scheduler.
Hello, world!
```

Extra tasks

- If your main function exits, you have to reset the microcontroller to start it again. What happens when you put an infinite loop at the end instead? Test your theory by flashing a looping program.
- Can you think of a way to prevent what you're now seeing? (click for hint:[1])

Troubleshooting

- `⛔ Git Error: authentication required`: your git configuration is probably set to override `https` github URLs to `ssh`. Check your global `~/.git/config` for `insteadOf` sections and disable them.
- `Error: Failed to generate bindings`: add `default = ["native"]` to `Cargo.toml`
- if you're using the deprecated `pio` build system, an initial git commit of your project will be required for a successful build.
- if `cargo espflash` is stuck on `Connecting...`, you might have another monitor process still running (e.g. from the initial `hardware-check` test). Try finding and terminating it. If this doesn't help, disconnect and reconnect the board's USB cable.

[1] yield control back to the underlying operating system by `sleep`ing in a loop instead of busy waiting. (use `use std::thread::sleep`)

HTTP and HTTPS client

In this exercise, we'll write a small client that retrieves data over a HTTP connection to the internet. Then we will upgrade it into an HTTPS client.

HTTP client

The goal of this exercise is to write a small HTTP client that connects to a website.

Setup

✅ Go to `intro/http-client` directory.

✅ Open the prepared project skeleton in `intro/http-client`.

✅ Add your network credentials to the `cfg.toml` as in the hardware test.

✅ Open the docs for this project with the following command:

```
$ cargo doc --open
```

`intro/http-client/examples/http_client.rs` contains the solution. you can run it with the following command:

```
cargo espflash --release --example http_client --monitor $SERIALDEVICE
```

Making a connection

By default only unencrypted HTTP is available, which rather limits our options of hosts to connect to. We're going to use `http://neverssl.com/`.

In `esp-idf`, HTTP client connections are managed by `http::client::EspHttpClient` in the `esp-idf-svc` crate. It implements the `http::client::Client` trait from `embedded-svc`, which defines functions for HTTP request methods like `GET` or `POST`. This is a good time to have a look at the documentation you opened with `cargo doc --open` for `http::client::EspHttpClient` and see instantiation methods at your disposal.

✅ Add the url `http://neverssl.com/` to the main function. This is the address we will query.

✅ Create a new `EspHttpClient` with default values. Look for a suitable constructor in the documentation.

Calling HTTP functions (e.g. `get(url)`) on this client returns an `EspHttpRequest`, which must be turned into a `Writer` to reflect the client's option to send some data alongside its request.

After this optional send step the `Writer` can be turned into a `Response` from which the received server output can be read:

The `get` function uses as_ref(). This means that instead of being restricted to one specific type like just `String` or just `&str`, the function can accept anything that implements the `AsRef<str>` trait - that is, any type where a call to `.as_ref()` will produce an `&str`. This works for `String` and `&str`, but also the `Cow<str>` enum type which contains either of the previous two.

```
let request = client.get(url.as_ref())?;
// the parameter passed to `into_writer` is the number of bytes
// the client intends to send
let writer = request.into_writer(0)?;
let response = writer.submit()?;
```

The parameter passed to `into_writer` is the number of bytes the client intends to send. Here we are not trying to send anything.

A successful response has a status code in the 2xx range. Followed by the raw html of the website.

✅ Verify the connection was successful.

✅ Return an Error if the status is not in the 2xx range.

```

match status {
        200..=299 => {
        }
        _ => anyhow::bail!("unexpected response code: {}", status),
    }
```

The status error can be returned with the Anyhow, crate which contains various functionality to simplify application-level error handling. It supplies a universal `anyhow::Result<T>`, wrapping the success (`Ok`) case in T and removing the need to specify the Err type, as long as every error you return implements `std::error::Error`.

✅ Turn your `response` into a `embedded_svc::io::Read` reader by calling `response.reader()` and read the received data chunk by chunk into a `u8` buffer using `reader.do_read(&mut buf)`. `do_read` returns the number of bytes read - you're done when this value is `0`.

✅ Report the total number of bytes read.

✅ Log the received data to the console. Hint, the response in the buffer is in bytes, so you might need a method to convert from bytes to `&str`.

Extra Tasks

✅ Handle 3xx, 4xx and 5xx status codes each in a separate match arm

✅ Write a custom `Error` enum to represent these errors. Implement the `std::error::Error` trait for your error.

Troubleshooting

- `missing WiFi name/password`: ensure that you've configured `cfg.toml` according to `cfg.toml.example` - a common problem is that package name and config section name don't match.

```
# Cargo.toml
#...
[package]
name = "http-client"
#...

# cfg.toml
[http-client]
wifi_ssid = "..."
wifi_psk = "..."
```

- `Guru Meditation Error: Core 0 panic'ed (Load access fault). Exception was unhandled.` This may caused by an `.unwrap()` in your code. Try replacing those by question marks.

HTTPS CLIENT

You will now make changes to your http client files so that it also works for encrypted connections.

`intro/http-client/examples/http_client.rs` contains the solution. You can run it with the following command: (It won't build unless you have completed the first step of the exercise.)

```
cargo espflash --release --example https_client --monitor $SERIALDEVICE
```

To establish a secure, encrypted HTTPS connection, we first need to add some certificates so a server's identity can be verified.

✅ Enable basic TLS certificate support in your project's `sdkconfig.defaults` by deleting the existing `CONFIG_MBEDTLS...` lines and adding:

```
CONFIG_MBEDTLS_CERTIFICATE_BUNDLE=y
CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_CMN=y
```

Now, we create a custom client configuration to use an `http::client::EspHttpClientConfiguration` which enables the use of these certificates and uses default values for everything else:

```

let mut client = EspHttpClient::new(&EspHttpClientConfiguration {
        use_global_ca_store: true,
        crt_bundle_attach: Some(esp_idf_sys::esp_crt_bundle_attach),

        ..Default::default()
    }
```

✅ Initialize your HTTP client with this new configuration and verify HTTPS works by downloading from a `https` resource e.g. `https://espressif.com/`. the download will show as raw html in the terminal output.

Troubleshooting (repeated from previous section)

- `missing WiFi name/password`: ensure that you've configured `cfg.toml` according to `cfg.toml.example` - a common problem is that package name and config section name don't match.

```toml
# Cargo.toml
#...
[package]
name = "http-client"
#...

# cfg.toml
[http-client]
wifi_ssid = "..."
wifi_psk = "..."
```

A simple HTTP server

We're now turning our board into a tiny web server that upon receiving a `GET` request serves data from the internal temperature sensor.

Setup

You can find a prepared project skeleton in `intro/http-server/`. It includes establishing a WiFi connection, but you must configure it to use your network's credentials in `cfg.toml`.

`intro/http-server/examples/https-server.rs` contains a solution. You can run it with the following command:

```
cargo espflash --release --example http_serve --monitor $SERIALDEVICE
```

Serving requests

To connect to your board with your browser, you need to know the board's IP address.

✅ Run the skeleton code in `intro/http-server`. The output should yield the board's IP address like this:

```
I (3862) esp_netif_handlers: sta ip: 192.168.178.54, mask: ..
server awaiting connection
```

The `sta ip` is the "station", the WiFi term for an interface connected to an access point. This is the address you'll put in your browser (or other http client like `curl`).

> esp-idf tries to register the hostname `espressif` in your local network, so often `http://espressif/` instead of `http://<sta ip>/` will also work.
>
> You can change the hostname by setting `CONFIG_LWIP_LOCAL_HOSTNAME` in `sdkconfig.defaults`, e.g.: `CONFIG_LWIP_LOCAL_HOSTNAME="esp32c3"`

Sending HTTP data to a client involves:

- creating an an instance of `EspHttpServer`
- looping in the main function so it doesn't terminate - termination would result in the server going out of scope and subsequently shutting down
- setting a separate request `handler` function for each requested path you want to serve content. Any unconfigured path will result in a `404` error. These handler functions are realized inline as Rust closures via:

```
server.handle_get(path, |request, response| {
    // the `response` needs to write data to the client
    let mut writer = response.into_writer(request);

    // construct a response
    let some_buf = ...;

    // now you can write your desired data
    writer.write_all(&some_buf);

    // once you're done the handler expects a `Completion` as result,
    // this is achieved via:
    Ok(())
});
```

✅ Create a `EspHttpServer` instance using a default `esp_idf_svc::http::server::Configuration`. The default configuration will cause it to listen on port 80 automatically.

✅ Verify that a connection to `http://<sta ip>/` yields a `404` (not found) error stating `This URI does not exist`.

✅ Write a request handler for requests to the root path (`"/"`). The request handler sends a greeting message at `http://<sta ip>/`, using the provided `index_html()` function to generate the HTML String.

Dynamic data

We can also report dynamic information to a client. The skeleton includes a configured `temp_sensor` that measures the board's internal temperature.

✅ Write a second handler that reports the chip temperature at `http://<sta ip>/temperature`, using the provided `temperature(val: f32)` function to generate the HTML String.

Hints

- If you want to send a response string, it needs to be converted into a `&[u8]` slice via `a_string.as_bytes()`
- The temperature sensor needs exclusive (mutable) access. Passing it as owned value into the handler will not work (since it would get dropped after the first invocation) - you can fix this by making the handler a `move ||` closure, wrapping the sensor in an `Arc<Mutex<_>>`, keeping one `clone()` of this `Arc` in your main function and moving the other into the closure.

Troubleshooting

- `httpd_txrx: httpd_resp_send_err` can be solved by restarting, or `cargo clean` if nothing happens.
- Make sure computer and rust board are using the same wifi network.

IoT using MQTT

How does MQTT work

❗ This exercise requires an MQTT server. If you're participating in a Ferrous Systems training, login credentials for a server operated by Espressif will be made available in the workshop, otherwise you can use one listed at https://test.mosquitto.org/ or install one locally.

To conclude the introductory course, let's add some IoT functionality to the board. Our goal here is have it send out real-time updates of sensor values without having to poll repeatedly, like we would with an HTTP server, and also receive commands to change the board LED color.

This can be modeled using a publish-subscribe architecture, where multiple clients publish messages in certain channels/topics, and can also subscribe to these topics to receive messages sent by others. Dispatching of these messages is coordinated by a message broker - in our case, this is done an MQTT server.

MQTT messages

An MQTT message consists of two parts - topic and payload.

The topic serves the same purpose as an email subject or a label on a filing cabinet, whereas the payload contains the actual data. The payload data format is not specified, although JSON is common.

🔎 The most recent version of the MQTT standard (MQTT 5) supports content type metadata.

When sending a MQTT message, a Quality of Service (QoS) parameter needs to be defined, indicating delivery guarantees:

- at most once
- at least once
- exactly once.

For the purpose of this exercise it does not matter which quality you choose.

MQTT topics

MQTT topics are UTF-8 strings representing a hierarchy, with individual levels separated by a `/` slash character. A leading slash is supported but not recommended. Some example topics are:

```
home/garage/temperature
beacons/bicycle/position
home/alarm/enable
home/front door/lock
```

Here a sensor would periodically publish the garage temperature which then gets broadcast to every subscriber, just as the bicycle beacon publishes its GPS coordinates. The `alarm` and `lock` topics serve as a command sink for specific devices. However, nothing prevents additional subscribers from listening in on these commands, which might provide useful for auditing purposes.

🔎 Topics starting with `$` are reserved for statistics internal to the broker. Typically the topic will begin with `$SYS`. Clients cannot publish to these topics.

❗ Since all workshop participants will be sharing the same MQTT server, some measures are required to prevent crosstalk between different projects. The exercise skeleton will generate a unique, random ID (in the `UUID v4` format) for each repository checkout. You can also manually generate your own online. Your UUID should be used as leading part of the message topics sent between computer and board, roughly resembling this pattern:

```
6188eec9-6d3a-4eac-996f-ac4ab13f312d/sensor_data/temperature
6188eec9-6d3a-4eac-996f-ac4ab13f312d/command/board_led
```

Subscribing to topics

A client sends several subscribe messages to indicate they're interested in receiving certain topics. Wildcards are optionally supported, both for a single hierarchy level and as a catch-all:

- `home/garage/temperature` - subscribes only to this specific topic
- `home/#` - the hash character is used as multi-level wildcard and thus subscribes to every topic starting with `home/` - `home/garage/temperature`, `home/front door/lock` and `home/alarm/enable` would all match, but `beacons/bicycle/position` won't. The multi-level wildcard must be placed at the end of a subscription string.
- `home/+/temperature` - the plus character serves as single-level wildcard to subscribe to `home/garage/temperature`, `home/cellar/temperature`, etc.

MQTT Exercise: Sending Messages

Setup

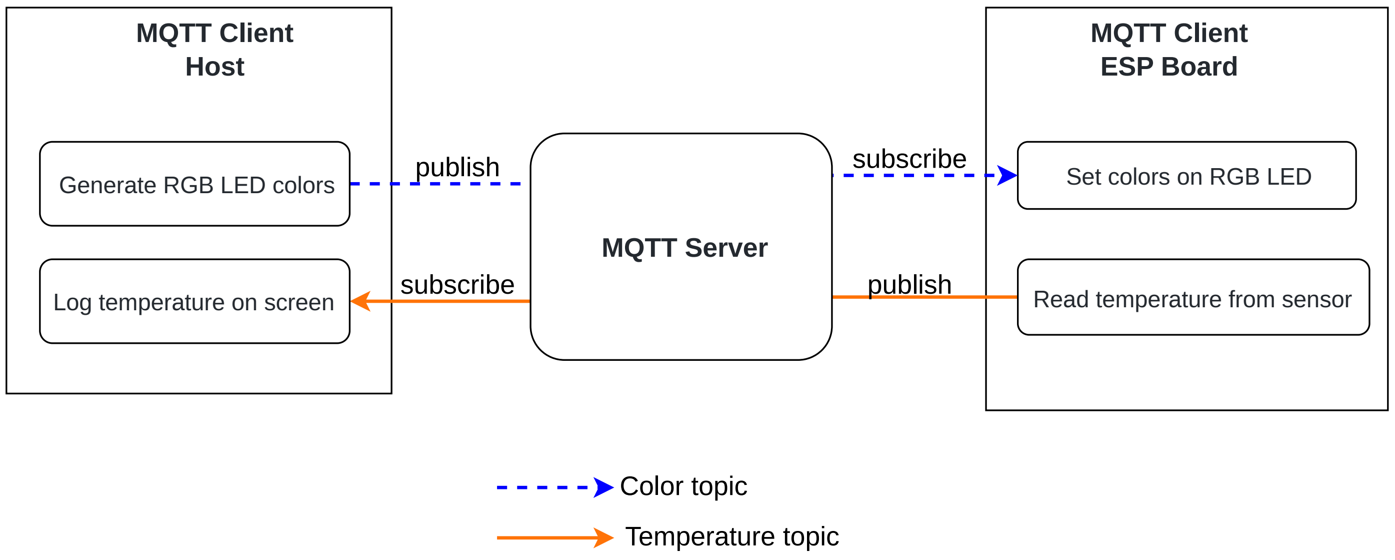
✅ You can find a prepared project skeleton in `intro/mqtt/exercise`.

✅ In `intro/mqtt/host_client` you can find a host run program that mimics the behavior of a second client. Run it in a separate terminal using the `cargo run` command. Find more information about the host client below.

The client also generates random RGB colors and publishes them in a topic. **This is only relevant for the second part of this exercise**.

❗ Similar to the http exercises you need to configure your connection credentials in `cfg.toml` for both programs. Besides WiFi credentials you'll also need to add MQTT server details. Check each `cfg.toml.example` for required settings. Remember the name between brackets in the the `cfg.toml` file is the name of the package in `Cargo.toml`.

The structure of the exercises is as below. In this part, we will focus on the Temperature topic.



Tasks

✅ Create an `EspMqttClient` with a default configuration and an empty handler closure.

✅ Send an empty message under the `hello_topic` to the broker. Use the `hello_topic(uuid)` utility function to generate a properly scoped topic.

✅ Verify a successful publish by having a client connected that logs these messages. The `host_client` implements this behavior. You should run it in another terminal.

✅ In the loop at the end of your main function, publish the board temperature on `temperature_data_topic(uuid)` every second. Verify this, too.

Establishing a connection

Connections are managed by an instance of `esp_idf_svc::mqtt::client::EspMqttClient`. It is constructed using

- a broker URL which in turn contains credentials, if necessary
- a configuration of the type `esp_idf_svc::mqtt::client::MqttClientConfiguration`
- a handler closure similar to the http server exercise

```
let mut client = EspMqttClient::new(broker_url,
    &mqtt_config,
    move |message_event| {
        // ... your handler code here - leave this empty for now
        // we'll add functionality later in this chapter
    })?;
```

Support tools & crates

To log the sensor values sent by the board, a helper client is provided under `intro/mqtt/host_client`. It subscribes to the temperature topic.

The `mqtt_messages` crate (located in `common/lib`) supports handling messages, subscriptions and topics:

Functions to generate topic strings

- `color_topic(uuid)` - creates a topic to send colors that will be published to the board.

- `hello_topic(uuid)` - test topic for initially verifying a successful connection
- `temperature_data_topic(uuid)` - creates a whole "temperature" topic string

Encoding and decoding message payloads

The board temperature `f32` float is converted to four "big endian" bytes using `temp.to_be_bytes()`.

```
// temperature
let temperature_data = &temp.to_be_bytes() as &[u8]; // board
let decoded_temperature = f32::from_be_bytes(temperature_data); // workstation
```

Publish & Subscribe

`EspMqttClient` is also responsible for publishing messages under a given topic. The `publish` function includes a `retain` parameter indicating whether this message should also be delivered to clients that connect after it has been published.

```
let publish_topic = /* ... */;
let payload: &[u8] = /* ... */ ;
client.publish(publish_topic, QoS::AtLeastOnce, false, payload)?;
```

Troubleshooting

- `error: expected expression, found .` when building example client: update your stable Rust installation to 1.58 or newer
- MQTT messages not showing up? make sure all clients (board and workstation) use the same UUID (you can see it in the log output)
- Make sure the `cfg.toml` file is configured properly. The `example-client` has a `dbg!()` output at the start of the program, that shows `mqtt` configuration. It should output the content of your `cfg.toml` file.
- `error: expected expression, found .` while running the host-client can be solved with `rustup update`

MQTT Exercise: Receiving LED Commands

✅ Subscribe to `color_topic(uuid)`

✅ Run `host_client` in parallel in it's own terminal. The `host_client` publishes board LED `color` roughly every second.

✅ Verify your subscription is working by logging the information received through the topic.

✅ React to the LED commands by setting the newly received color to the board with `led.set_pixel(/* received color here */)`.

Encoding and decoding message payloads

The board LED commands are made of three bytes indicating red, green and blue. - `enum ColorData` contains a topic `color_topic(uuid)` and the `BoardLed` - it can convert the `data()` field of an `EspMqttMessage` by using `try_from()`. The message needs first to be coerced into a slice, using `let message_data: &[u8] = &message.data();`

```
// RGB LED command

if let Ok(ColorData::BoardLed(color)) = ColorData::try_from(message_data) { /* set
new color here */ }
```

Publish & Subscribe

`EspMqttClient` is not only responsible for publishing but also for subscribing to topics.

```
let subscribe_topic = /* ... */;
client.subscribe(subscribe_topic, QoS::AtLeastOnce)
```

Handling incoming messages

The `message_event` parameter in the handler closure is of type `Result<Event<EspMqttMessage>`. Since we're only interested in processing successfully received messages, we can make use of deep pattern matching into the closure:

```
let mut client =
        EspMqttClient::new(
            broker_url,
            &mqtt_config,
            move |message_event| match message_event {
                Ok(Received(msg)) => process_message(msg, &mut led),
                _ => warn!("Received from MQTT: {:?}", message_event),
            },
        )?;
```

In the processing function, you will handle `Complete` messages. Use Rust Analyzer to generate the missing match arms or match any other type of response by logging an `info!()`.

```
    match message.details() {
        // all messages in this exercise will be of type `Complete`
        // the other variants of the `Details` enum are for larger message
payloads
        Complete => {

            // Cow<&[u8]> can be coerced into a slice &[u8] or a Vec<u8>
            // You can coerce it into a slice to be sent to try_from()
            let message_data: &[u8] = &message.data();
            if let Ok(ColorData::BoardLed(color)) =
ColorData::try_from(message_data) {
                // set the LED to the newly received color

            }
        }
        // Use Rust Analyzer to generate the missing match arms or match an
incomplete message with a log message.

    }
```

Hints

- Use a logger to see what you are receiving, for example: `info!("{}", color);` or `dbg!(color)`.

Extra tasks

Implement MQTT with hierarchical topics

- work on this if you have finished everything else. We don't provide a full solution for this, as this is to test how far you get on your own.

Check `common/lib/mqtt-messages`:

- Implement the same procedure but by using MQTT hierarchy. Subscribe subscribing to all "command" messages, combining `cmd_topic_fragment(uuid)` with a trailing `#` wildcard.
- Use `enum Command` instead of `enum ColorData`. `enum Command` represents all possible commands (here: just `BoardLed`).
- `RawCommandData` stores the last part of a message topic (e.g. `board_led` in `a-uuid/command/board_led`). It can be converted into a `Command` using `try_from`.

```
// RGB LED command
let raw = RawCommandData {
    path: command,
    data: message.data(),
};
```

Check the `host-client`:

- you will need to replace `color` with `command`. For example like this

```
let command = Command::BoardLed(color)
```

- in the `process_message()` function you will need to parse the topic.

```
match message.details() {
    Complete => {
        // all messages in this exercise will be of type `Complete`
        // the other variants of the `Details` enum
        // are for larger message payloads

        // Cow<str> behaves a lot like other Rust strings (&str, String)
        let topic: Cow<str> = message.topic(token);

        // determine if we're interested in this topic and
        // dispatch based on its content
        let is_command_topic: bool = /* ... */;
        if is_command_topic {
            let raw = RawCommandData { /* ... */ };
            if let Ok(Command::BoardLed(color)) = Command::try_from(raw) {
                // set the LED to the newly received color
            }

        },
        _ => {}
        }
    }
}
```

Hints!

- Since you will be iterating over a MQTT topic, you will need to `split()` on a string returns an iterator. You can access a specific item from an iterator using `nth()`.
- The solution implementing hierarchy can be run with `cargo espflash --release --example solution2 --monitor /dev/tty.usbmodem0`, while the solution without can be run with `cargo espflash --release --monitor /dev/tty.usbmodem0` or `cargo espflash --release --example solution1 --monitor /dev/tty.usbmodem0`

Other tasks

- leverage `serde_json` to encode/decode your message data as JSON.
- Send some messages with a large payload from the host client and process them on the microcontroller. Large messages will be delivered in parts instead of `Details::Complete`:

```
InitialChunk(chunk_info) => { /* first chunk */},
SubsequentChunk(chunk_data) => { /* all subsequent chunks */ }
```

You do not need to differentiate incoming chunks based on message ID, since at most one message will be in flight at any given time.

Troubleshooting

- `error: expected expression, found .` When building host client: update your stable Rust installation to 1.58 or newer
- MQTT messages not showing up? make sure all clients (board and workstation) use the same UUID (you can see it in the log output)

Advanced Workshop

In this course we're going to dive deeper into topics that are embedded-only and/or close to the hardware, especially focussing on lower level i/o. Unlike in the first part, we'll not just use the higher level abstractions where for example something like pin configurations are hidden away. Instead you'll learn how to configure them yourself. You're also going to learn how to write directly into registers and how to find out which register is needed in the first place. We'll talk about ownership issues and memory safety issues in the context of exercises.

This part consists of three exercises:

In the first one you'll learn how to handle a button interrupt, in the second you'll read sensor values from sensors via the I²C bus. Once you have used the drivers we have prepared, you'll learn how to get started writing your own. This is a necessary skill as Rust drivers are usually not provided by manufacturers.

Preparations

Please go through the preparations chapter to prepare for this workshop.

Reference

If you're new to embedded programming read our reference where we explain some terms in a basic manner.

Lower level I/O: How to manipulate Registers

In general there are two ways to write firmware for the ESP32-C3. One is the bare-metal using only `[no_std]` Rust, and the other using `[std]` Rust and C-Bindings to the esp-idf. `[no_std]` Rust refers to Rust not using the standard library, only the core library, which is a subset of the standard library that does not depend on the existence of an operating system.

What do the ecosystems look like?

`[std]` Rust and the esp-idf

The most established way to use Rust on ESP32-C3 is using C bindings to the esp-idf. We can use Rust's standard library when going this route, as we can use an operating system: FreeRTOS. Being able to use the standard library comes with benefits: We can use all types no matter if they are stack or heap allocated. We can use threads, Mutexes and other synchronization primitives.

The esp-idf is mostly written in C and as such is exposed to Rust in the canonical split crate style:

- a `sys` crate to provide the actual `unsafe` bindings (esp-idf-sys)
- a higher level crate offering safe and comfortable Rust abstractions (esp-idf-svc)

The final piece of the puzzle is low-level hardware access, which is again provided in a split fashion:

- esp-idf-hal implements the hardware-independent embedded-hal traits like analog/digital conversion, digital I/O pins, or SPI communication - as the name suggests, it also uses `esp-idf` as a foundation

More information is available in the ecosystem chapter of the `esp-rs` book.

This is the way that currently allows the most possibilities on Espressif chips if you want to use Rust. Everything in this course is based on this approach.

We're going to look at how to write values into Registers in this ecosystem in the context of the Interrupt exercise.

Bare metal Rust with `[no_std]`

As the name bare metal implies, we don't use an operating system. Because of this, we can't use language features that rely on one. The core library is a subset of the standard library that excludes features like heap allocated types and threads. Code that uses only the core library is labelled with `#[no_std]`. `#[no_std]` code can always run in a `std` environment, but the reverse is not true. In Rust the mapping from Registers to Code works like this:

Registers and their fields on a device are described in *System View Description* (SVD) files. `svd2rust` is used to generate *Peripheral Access Crates* (PACs) from these SVD files. The PACs provide a thin wrapper over the various memory-mapped registers defined for the particular model of micro-controller you are using.

Whilst it is possible to write firmware with a PAC alone, some of it would prove unsafe or otherwise inconvenient as it only provides the most basic access to the peripherals of the microcontroller. So there is another layer, the *Hardware Abstraction Layer* (HAL). HALs provide a more user friendly API for the chip, and often implement common traits defined in the generic `embedded-hal`.

Microcontrollers are usually soldered to some *Printed Circuit Board* (or just *Board*), which defines the connections that are made to each pin. A *Board Support Crate* (BSC, also known as a *Board Support Package* or BSP) may be written for a given board. This provides yet another layer of abstraction and might, for example, provide an API to the various sensors and LEDs on that board - without the user necessarily needing to know which pins on your microcontroller are connected to those sensors or LEDs.

Although a PAC for the ESP32-C3 exists, bare-metal Rust is highly experimental on ESP32-C3 chips, so for now we will not work with it on the microcontroller directly. We will write a partial sensor driver in this approach as driver's should be platform agnostic.

I²C

Introduction

The Inter-Integrated Circuit is a serial protocol (usually shortened to I²C or I2C), which allows multiple peripheral (or slave) chips to communicate with one or more controller (or master) chips. Many devices can be connected to the same I²C bus and messages can be sent to a particular device by specifying its I²C address. The protocol requires two signal wires and can only be used for short-distance communications within one device.

One of the signal lines is for data (SDA) and the other is for the clock signal (SCL). The lines are pulled-high by default with some resistors fitted somewhere on the bus. Any device on the bus (or even multiple devices simultaneously) can 'pull' either or both of the signal lines low. This means that no damage occurs if two devices try and talk on the bus at the same time - the messages are merely corrupted (and detectably so).

An I²C *transaction* consists of one or more *messages*. Each *message* is comprised of a *start symbol*, some *words*, and finally a *stop symbol* (or another *start symbol* if there is a follow-on message). Each word is eight bits, followed by an ACK (0) or NACK (1) bit which is sent by the recipient to indicate whether the word was received and understood correctly. The first word indicates both the 7-bit address of the device the message is intended for, plus a bit to indicate if the device is being *read from* or being *written to*. If there is no device of that address on the bus, the first word will naturally have a 'NACK' after it (because there is no device driving the SDA line low to generate an 'ACK' bit) and so you know there is no device present.

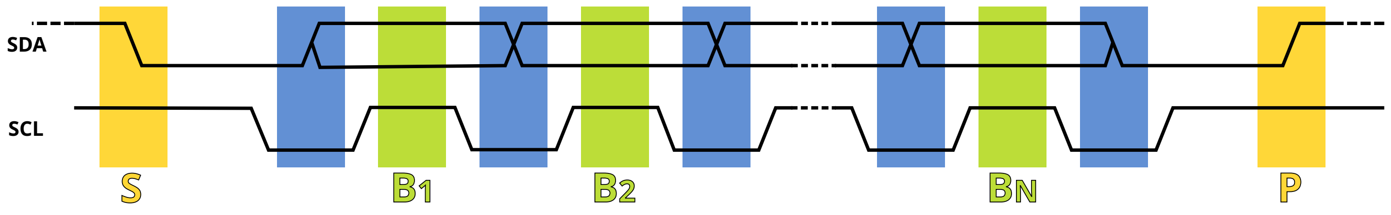
The clock frequency of the SCL line is usually 400 kHz but slower and faster speed are supported (standard speed are 100kHz-400kHz-1MHz).In our exercise, the configuration will be 400 kHz (`<MasterConfig as Default>::default().baudrate(400.kHz().into())`).

To read three bytes from an EEPROM device, the sequence will be something like:

Step	Controller Sends	Peripheral Sends
1.	START	
2.	Device Address + W	
3.		ACK
4.	High EEPROM Address byte	
5.		ACK
6.	Low EEPROM Address byte	

Step	Controller Sends	Peripheral Sends
7.		ACK
8.	START	
9.	Device Address + R	
10.		ACK
11.		Data Byte from EEPROM Address
12.	ACK	
13.		Data Byte from EEPROM Address + 1
14.	ACK	
15.		Data Byte from EEPROM Address + 2
16.	NAK (i.e. end-of-read)	
17.	STOP	

I²C signal image



A sequence diagram of data transfer on the I²C bus:

- S - Start condition
- P - Stop condition
- B_1 to B_N - transferring of one bit
- SDA changes are allowed when SCL is low (blue), otherwise there will be a start or stop condition generated.

Source & more details: Wikipedia.

I²C Sensor Reading Exercise

In this exercise you will learn how to read out sensors on the I²C bus.

The board has two sensors that can be read via the I²C bus: `

Peripheral	Part number	Reference	Crate	Address
IMU	ICM-42670-P	Datasheet	Link	0x68
Temperature and Humidity	SHTC3	Datasheet	Link	0x70

The task is to use an existing driver from crates.io to read out the temperature and humidity sensor over I²C. After that, a second sensor will be read out over the same I²C bus using `shared-bus`. The driver for the second sensor is available locally in `common/`.

Part 1: Reading Temperature & Humidity

Create an instance of the of the Humidity sensor SHTC3 and read and print the values for humidity and temperature every 600 ms.

`i2c-sensor-reading/examples/part_1.rs` contains a working solution of Part 1. To run the solution add `--example part_1` to your run command:

```
$ cargo espflash --release --example part_1 --monitor /dev/SERIAL_DEVICE
```

`i2c-sensor-reading/src/main.rs` contains skeleton code, that already contains necessary imports for this part.

Steps:

✅ Go to the `i2c-sensor-reading/` folder and open the relevant documentation with the following command:

```
$ cargo doc --open
```

✅ Define two pins, one as SDA and one as SCL.

Signal	GPIO
SDA	GPIO10

Signal	GPIO
SCL	GPIO8

✅ Create an Instance of the I²C peripheral with the help of the documentation you generated. This requires a baud rate: You can use 400kHz, a default value.

✅ Use the `shtcx` driver crate, make an instance of the SHTC3 sensor passing the I²C instance into them. Check the documentation for guidance.

✅ To check if the sensor is addressed correctly, read it's device ID and print the value.

Expected Output:

```
Device ID: 71
```

✅ Make a measurement, and read the sensor values and print them. Check the documentation for guidance on sensor methods.

Expected Output:

```
TEMP: [local temperature] °C
HUM: [local humidity] %
```

❗ Some sensors need some time to pass between measurement and reading value. ❗ Watch out for the expected units!

Hints

- There are methods that turn the sensor values into the desired unit.

Part 2: Reading Accelerometer data.

Using a bus manager, implement the second sensor. Read out its values and print the values from both sensors.

Continue with your own solution from part one. Alternatively you can start with the provided partial solution of Part 1: `i2c-sensor-reading/examples/part_1.rs`.

`i2c-sensor-reading/examples/part_2.rs` contains a working solution of Part 2. You can consult it if you need help.

Steps

✅ Import the driver crate for the ICM42670p.

```
use imc42670p;
```

✅ Create an instance of the sensor.

✅ Why does passing the same I²C instance to two sensors not work, despite both being on the same I²C bus?

▶ Answer

✅ Import the bus manager crate.

```
use shared_bus;
```

✅ Create an instance of a simple bus manager. Make two proxies and use them instead of the original I²C instance to pass to the sensors.

✅ Read & print the device ID from both sensors.

Expected Output:

```
Device ID SHTC3: 71
Device ID ICM42670p: 96
```

✅ Start the ICM42670p in low noise mode.

✅ Read the gyroscope sensor values and print them with 2 decimal places alongside the temperature and humidity values.

Expected Output:

```
GYRO: X: 0.00 Y: 0.00 Z: 0:00
TEMP: [local temperature] °C
HUM: [local humidity] %
```

I²C Driver Exercise - Easy Version

We're not going to write an entire driver, merely the first step: the `hello world` of driver writing: reading the device ID of the sensor. This version is labelled easy, because we explain the code fragments, and you only have to copy and paste the fragments into the right place. Use this version if you have very little previous experience with Rust, if these workshops are your first in the embedded domain, or if you found the hard version too hard. You can work in the same file with either version.

`i2c-driver/src/icm42670p.rs` is a gap text of a very basic I²C IMU sensor driver. The task is to complete the file, so that running `main.rs` will log the device ID of the driver. This gap text driver is based on the version of the same name that lives in common, but provides a little bit more functionality.

`i2c-driver/src/icm42670p_solution.rs` provides the solution to this exercise. If you want to run it, the imports need to be changed in `main.rs` and `lib.rs`. The imports are already there, you only need to comment the current imports out and uncomment the solutions as marked in the line comments.

Driver

Instance of the Sensor

To use a peripheral sensor first you must get an instance of it. The sensor is represented as a struct that contains both its device address, and an object representing the I²C bus itself. This is done using traits defined in the `embedded-hal` crate. The struct is public as it needs to be accessible from outside this crate, but its fields are private.

```rust
#[derive(Debug)]
pub struct ICM42670P<I2C> {
    /// The concrete I²C device implementation.
    i2c: I2C,

    /// Device address
    address: DeviceAddr,
}

// ...
```

We add an `impl` block that will contain all the methods that can be used on the sensor instance. It also defines the Error Handling. In this block we also implement an instantiating method. Methods can also be public or private. This method needs to be accessible from outside, so it's labelled `pub`. Note that written this way, the sensor instance takes ownership of the I²C bus.

```
impl<I2C, E>ICM42670P<I2C>
where
    I2C: i2c::WriteRead<Error = E> + i2c::Write<Error = E>,
{
    /// Create a new instance of the ICM42670P.
    pub fn new(i2c: I2C, address: DeviceAddr) -> Result<Self, E> {
        Ok(Self{ i2c, address })
    }
// ...
```

Device address

- the device's addresses are available in the code:

```
AD0 = 0b110_1000, // or 0x68
AD1 = 0b110_1001, // or 0x69
```

- This I²C device has two possible addresses - 0x68 and 0x69. We tell the device which one we want it to use by applying either 0V or 3.3V to the AP_AD0 pin on the device. If we apply 0V, it listens to address 0x68. If we apply 3.3V it listens to address 0x69. You can therefore think of pin AD_AD0 as being a one-bit input which sets the least-significant bit of the device address. More information is available in the data sheet, section 9.3

Representation of Registers

The sensor's registers are represented as enums. Each variant has the register's address as value. The type `Register` implements a method that exposes the variant's address.

```
#[derive(Clone, Copy)]
pub enum Register {
    WhoAmI = 0x75,
}

impl Register {
    fn address(&self) -> u8 {
        *self as u8
    }
}
```

read_register() and write_register()

We define a read and a write method, based on methods provided by the `embedded-hal` crate. They serve as helpers for more specific methods and as an abstraction that is adapted to a sensor with 8-bit registers. Note how the `read_register()` method is based on a `write_read()` method. The reason for this lies in the characteristics of the I²C protocol: We first need to write a command over the I²C bus to specify which register we want to read from. Helper methods can remain private as they don't need to be accessible from outside this crate.

```

impl<I2C, E>ICM42670P<I2C>
where
    I2C: i2c::WriteRead<Error = E> + i2c::Write<Error = E>,
{
    //...
    fn write_register(&mut self, register: Register, value: u8) -> Result<(), E> {
        let byte = value as u8;
        self.i2c
            .write(self.address as u8, &[register.address(), byte])
    }

    fn read_register(&mut self, register: Register) -> Result<u8, E> {
        let mut data = [0];
        self.i2c
            .write_read(self.address as u8, &[register.address()], &mut data)?;
        Ok(u8::from_le_bytes(data))
    }
}
```

✅ Implement a public method that reads the `WHOAMI` register with the address `0x0F`. Make use of the the above `read_register()` method.

✅ Optional: Implement further methods that add features to the driver. Check the documentation for the respective registers and their addresses. Some ideas:

```
* switching the the gyroscope sensor or the accelerometer on
* starting measurements
* reading measurements
```

🔎 General info about peripheral registers

Registers can have different meanings; in essence they are **a location that can store a value**.

In this specific context we are using an external device (since it is a sensor, even if it is on the same PCB). It is addressable by I2C and we are reading and writing to its register addresses. The addresses each identify a unique location that contains some information. In this case, we want the address for the location that contains the current temperature, as read by the sensor.

You can find the register map of the ICM-42670 in section 14 should you want to try to get other interesting data from this sensor.

Writing an I²C Driver - Hard

We're not going to write an entire driver, merely the first step: the `hello world` of driver writing: reading the device ID of the sensor. This version is labelled hard, because you have to come up with the content of the methods and research information in the `embedded-hal` and data-sheets yourself. You can work in the same file with either version.

`i2c-driver/src/icm42670p.rs` is a gap text of a very basic I²C IMU sensor driver. The task is to complete the file, so that running `main.rs` will log the device ID of the driver. The this gap text driver is based on the version of the same name that lives in common, but provides a little bit more functionality.

`i2c-driver/src/icm42670p_solution.rs` provides the solution to this exercise. If you want to run it, the imports need to be changed in `main.rs` and `lib.rs`. The imports are already there, you only need to comment the current imports out and uncomment the solutions as marked in the line comments.

Driver API

Instance of the Sensor

✅ Create a struct that represents the sensor. It has two fields, one that represents the sensor's device address and one that represents the `I²C` bus itself. This is done using traits defined in the `embedded-hal` crate. The struct is public as it needs to be accessible from outside this crate, but its fields are private.

✅ Implement an instantiating method in the `impl` block. This method needs to be accessible from outside, so it's labelled `pub`. The method takes ownership of the I²C bus and creates an instance of the struct you defined earlier.

Device address

✅ This I²C device has two possible addresses, find them in the data sheet, section 9.3.

🔎 We tell the device which one we want it to use by applying either 0V or 3.3V to the AP_AD0 pin on the device. If we apply 0V, it listens to address 0x68. If we apply 3.3V it listens to address 0x69. You can therefore think of pin AD_AD0 as being a one-bit input which sets the least-significant bit of the device address.

✅ Create an enum that represents both address variants. The values of the variants need to be in binary representation.

Representation of Registers

✅ Create an enum that represents the sensor's registers. Each variant has the register's address as value. For now you only need the WhoAmI register. Find its address in the data sheet.

✅ Implement a method that exposes the variant's address as `u8`.

read_register() and write_register()

✅ Check out the write and write_read function in the embedded-hal. Why is it `write_read` and not just `read`?

▶ Answer

✅ Define a `read_register` and a `write_register` method for the sensor instance. Use methods provided by the `embedded-hal` crate. They serve as helpers for more specific methods and as an abstraction that is adapted to a sensor with 8-bit registers. This means that the data that is written, as well as the data that is read is an unsigned 8-bit integer. Helper methods can remain private as they don't need to be accessible from outside this crate.

✅ Implement a public method that reads the `WHOAMI` register with the address `0x0F`. Make use of the the above `read_register()` method.

✅ Optional: Implement further methods that add features to the driver. Check the documentation for the respective registers and their addresses. Some ideas: * switching the the gyroscope sensor or the accelerometer on * starting measurements * reading measurements

General info about how registers work

- Registers are small amounts of storage, immediately accessible by the processor. The registers on the sensor are 8 bits.
- They can be accessed by their address
- You can find register maps in the section 14.
- Returning a value with MSB and LSB (most significant byte and least significant byte) is done by shifting MSB values, and OR LSB values.

```rust
let GYRO_DATA_X: i16 = ((GYRO_DATA_X1 as i16) << 8) | GYRO_DATA_X0 as i16;
```

If you need hints and inspiration on what to implement, you can check the icm42670p in the common/lib folder.

Interrupts

An interrupt is a request for the processor to interrupt currently executing code, so that the event can be processed in a timely manner. If the request is accepted, the processor will suspend its current activities, save its state, and execute a function called an interrupt handler to deal with the event. Interrupts are commonly used by hardware devices to indicate electronic or physical state changes that require time-sensitive attention, for example pushing a button.

The fact that interrupt handlers can be called at any time provides a challenge in embedded Rust: It requires the existence of statically allocated mutable memory that both the interrupt handler and the main code can refer to and it also requires that this memory is always accessible.

Challenges

Flash Memory

Flash memory does not fulfill this requirement as it is out of action for example during write operations. Interrupts that occur during this time will go unnoticed. In our example this would result in no reaction when the button is pushed. We solve this by moving the the interrupt handler into RAM.

Statically Mutable Memory

In Rust such memory can be declared by defining a `static mut`. But reading and writing to such variables is always unsafe, as without precautions race conditions can be triggered.

How do we handle this problem?

In our example, the ESP-IDF framework provides a `Queue` type which handles the shared-mutable state for us. We simply get a `QueueHandle` which unique identifies the particular `Queue` being used. However, the main thread is given this `QueueHandle_t` at run-time, and so we still need a small amount of shared-mutable state in order to share the `QueueHandle_t` with the interrupt routine. We use an `Option<QueueHandle_t>`, which we statically initialize to `None`, and later replace with `Some(queue_handle)` when the queue has been created by ESP-IDF.

In the interrupt routine Rust forces us to handle the case where the `static mut` is still `None`. If this happens we can either return early, or we can `unwrap()` the value, which will exit the program with an error if the value was not previously set to `Some(queue_handle)`.

There is still a risk that `main()` might be in the processing of changing the value of the variable (i.e. changing the `QueueHandle_t` value) just as the interrupt routine fires, leaving it in an inconsistent or invalid state. We mitigate this by making sure we only set the value once, and we do so before the interrupt is enabled. The compiler cannot check that this is safe, and so we must use the `unsafe` keyword when we read or write the value.

Read more about this in the Embedded Rust Book

`unsafe {}` blocks:

This code contains a lot of `unsafe {}` blocks. As a general rule, `unsafe` does not mean that the contained code is not memory safe, it means, that Rust can't make safety guarantees in this place and that it is in the responsibility of the programmer to ensure memory safety. For example Calling C Bindings is per se unsafe, as Rust can't make any safety guarantees for the underlaying C Code.

Building the Interrupt Handler

The goal of this exercise is to handle the interrupt that fires if the `BOOT` button is pushed. This exercise involves working with C bindings to the ESP-IDF and other unsafe operations, as well as non-typical rust documentation. In a first step we will go line by line to build this interrupt handler.

You can find a skeleton code for this exercise in `advanced/button-interrupt/src/main.rs.` You can find the solution for this exercise in `advanced/button-interrupt/examples/solution.rs`

Tasks

1. Configure the button (GPIO 9) with a c struct `gpio_config_t` the following settings:
   - input mode
   - pull up
   - interrupt on positive edge

The struct has the following fields:

- `pin_bit_mask`: represents the Pin number, the value 1 shifted by the number of the pin.
- `mode`: sets the mode of the pin, it can have the following settings:
  - `gpio_mode_t_GPIO_MODE_INPUT`
  - `gpio_mode_t_GPIO_MODE_OUTPUT`
  - `gpio_mode_t_GPIO_MODE_DISABLE` // disable gpio
  - `gpio_mode_t_GPIO_MODE_OUTPUT_OD` // open drain output
  - `gpio_mode_t_GPIO_MODE_INPUT_OUTPUT` // input and output
  - `gpio_mode_t_GPIO_MODE_INPUT_OUTPUT_OD` // open drain input and output
- `pull_up_en`: true.into(), if the GPIO is pulled up,
- `pull_down_en`: true.into(), if the GPIO is pulled down,
- `intr_type`: sets the interrupt type, it can have the following settings:
  - `gpio_int_type_t_GPIO_INTR_ANYEDGE` // interrupt at any edge
  - `gpio_int_type_t_GPIO_INTR_DISABLE` // interrupt disabled
  - `gpio_int_type_t_GPIO_INTR_NEGEDGE` // interrupt at negative edge

- `gpio_int_type_t_GPIO_INTR_POSEDGE` // interrupt at positive edge

They are constants with numbers representing the bit that must be set in the corresponding register.

2. Write the configuration into the register with `unsafe extern "C" fn gpio_config`. This needs to happen in the unsafe block. To make these FFI calls we can use the macro `esp!($Cfunktion)`.

3. Install a generic GPIO interrupt handler with `unsafe extern "C" fn gpio_install_isr_service`. This function takes `ESP_INTR_FLAG_IRAM` as argument.

4. Create a `static mut` that holds the queue handle we are going to get from `xQueueGenericCreate`. This is a number that uniquely identifies one particular queue, as opposed to any of the other queues in our program. The queue storage itself if managed by the Operating System.

```
static mut EVENT_QUEUE: Option<QueueHandle_t> = None;
```

5. Create the event queue using `pub unsafe extern "C" fn xQueueGenericCreate`. This lets us safely pass events from an interrupt routine to our main thread.

```
EVENT_QUEUE = Some(xQueueGenericCreate(QUEUE_SIZE, ITEM_SIZE, QUEUE_TYPE_BASE));
```

6. Add a function which that will be called whenever there is a GPIO interrupt on our button pin. We put this function in a special block of RAM (`iram0`), so it will still be available even if the external flash is busy doing something else (like filesystem work). The function needs to get the queue handle from `EVENT_QUEUE` and call the `xQueueGiveFromISR` function with a `std::ptr::null_mut()` - the objects in our queue are of size zero, so we don't actually need a 'thing' to put on the queue. Instead, the act of pushing a 'nothing' is enough to wake up the other end!

```
#[link_section = ".iram0.text"]
unsafe extern "C" fn button_interrupt(_: *mut c_void) {
    xQueueGiveFromISR(EVENT_QUEUE.unwrap(), std::ptr::null_mut());
}
```

If the interrupt fires, an event is added to the queue.

7. Pass the function we just wrote to the generic GPIO interrupt handler we registered earlier, along with the number of the GPIO pin that should cause this function to be executed.

```
esp!(gpio_isr_handler_add(
    GPIO_NUM,
    Some(button_interrupt),
    std::ptr::null_mut()
))?;
```

8. Inside a loop, wait until the queue has an item in it. That is, until the `button_interrupt` function puts something in the queue.

```
let res = xQueueReceive(EVENT_QUEUE.unwrap(), ptr::null_mut(), QUEUE_WAIT_TICKS);
```

9. Handle the value of `res`, so that "Button pushed!" is logged, if the button is pushed.

10. Run the program and push the `BOOT` button, so see how it works!

Random LED color on pushing a button

✅ Modify the code so the RGB LED light changes to different random color upon each button press. The LED should not go out or change color if the button is not pressed for some time.

Continue by adding to your previous solution or the code from `advanced/button-interrupt/src/main.rs` You can find the solution for this exercise in `advanced/button-interrupt/examples/solution_led.rs`

Solving Help

- The necessary imports are already made, if you enter `cargo --doc --open` you will get helping documentation regarding the LED.
- The LED's part number is WS2812RMT.
- It's a programmable RGB LED. This means there aren't single pins to set for red, green and blue, but that we need to instantiate it to be able to send `RGB8` type values to it with a method.
- The board has a hardware random number generator. It can be called with `esp_random()`.
- Calling functions from the `esp-idf-sys` is unsafe in Rust terms and requires an `unsafe()` block. You can assume that these functions are safe to use, so no other measures are required.

Step by Step Guide to the Solution

1. Initialize the LED peripheral and switch the LED on with an arbitrary value just to see that it works.

```
let mut led = WS2812RMT::new()?;

 let arbitrary_color = RGB8::new(20, 0, 20);
 led.set_pixel(arbitrary_color).unwrap(); // remove this line after you tried
it once
```

2. Light up the LED only when the button is pressed. You can do this for now by exchanging the print statement.

```
1 => {
     led.set_pixel(arbitrary_color)?;

     },
 _ => {},
```

3. Create random RGB values by calling `esp_random()`.
   - This function is unsafe.
   - It yields u32, so it needs to be cast as u8.

```
unsafe {
//...
1 => {
    let r = esp_random() as u8;
    let g = esp_random() as u8;
    let b = esp_random() as u8;

    let color = RGB8::new(r, g, b);
    led.set_pixel(color)?;

    },
_ => {},
```

4. Optional: If you intend to reuse this code in another place, it makes sense to put it into its own function. This lets us explore in detail, which parts of the code need to be in `unsafe` blocks.

```
    fn random_light(led: &mut WS2812RMT) {

        let mut color = RGB8::new(0, 0, 0);
        unsafe {
            let r = esp_random() as u8;
            let g = esp_random() as u8;
            let b = esp_random() as u8;

            color = RGB8::new(r, g, b);
        }

        led.set_pixel(color).unwrap();
    }

    unsafe {
        // ...
        match res {
                1 => {
                    // Generates random rgb values
                    random_light(&mut led);

                },
                _ => {},
            };
        }
    }
}
```

Reference

GPIO

GPIO is short for General Purpose Input Output. GPIOs are digital (or sometimes analogue) signal pins that can be used as interfaces to other systems or devices. Each pin can be in various states, but they will have a default state on power-up or after a system reset (usually a harmless one, like being a digital input). We can then write software to change them into the appropriate state for our needs.

We'll introduce a couple of concepts related to GPIOs:

Pin Configurations

GPIOs can be configured one of several different ways. The options available can vary depending in the design of the chip, but will usually include:

Floating: A floating pin is neither connected VCC nor Ground. It just floats around at whatever voltage is applied. Note though, that your circuit should externally pull the pin either low or high, as CMOS silicon devices (such as microcontrollers) can be fail to work correctly if you leave a pin higher than the 'low voltage threshold' or `Vtl`, but lower than the 'high voltage threshold' or `Vth` for more than a few microseconds.

Push-Pull-Output: A pin that is configured as push–pull output can then be set to either drive a high voltage on to the pin (i.e. connect it to VCC), or a low voltage on to the pin (i.e. connect it to Ground). This is useful for LEDs, or buzzers or other devices that use small amounts of power.

Open-Drain-Output: Open Drain outputs switch between "disconnected" and "connected to ground". It is expected that some external resistor will weakly pull the line up to VCC. This type of output is designed to allow multiple devices to be connected together - the line is 'low' if any of the devices connected to the line drive it low. If two or more devices drive it low at the same time, no damage occurs (connecting Ground to Ground is safe). If none of them drive it low, the resistor will pull it high by default.

Floating-Input: A pin where the external voltage applied can be read in software, as either a `1` (usually if the voltage is above some threshold voltage) or a `0` (if it isn't). The same warnings apply as per the 'Floating' state.

Pull-Up-Input: Like a Floating-Input, except an internal 'pull-up' resistor weakly pulls the line up to VCC when nothing external is driving it down to Ground. Useful for reading buttons and other switches, as it saves you from needing an external resistor.

Active high/low

A digital signal can be in two states: `high` and `low`. This is usually represented by the voltage difference between the signal and ground. It is arbitrary which of these voltage levels represents which logic states: So both `high` and `low` can be defined as an active state.

For example: An active high pin has voltage when the logic level is active. And active low pin has voltage when the logic level is set to inactive.

In embedded Rust abstractions show the logic level and not the voltage level. So if you have an active low pin connected to an LED, you need to set it to inactive in order for the LED to light up.

Chip Select

Chip Select is a binary signal to another device that can switch that device on or off, either partially or entirely. It is usually a signal line connected to a GPIO, and commonly used to allow multiple devices to be connected to the same SPI bus - each device only listens when its Chip Select line is active.

Bit Banging

For protocols such as I2C or SPI we usually use peripherals within the MCU to convert the data we want to transmit into signals. In some cases, for example if the MCU does not support the protocol or if a non-standard form of the protocol is used, you need to write a program that turns the data into signals manually. This is called bit-banging.