Beginner workshop

https://embedded-trainings.ferrous-systems.com/

Please do the setup steps

- if you haven't already
  - https://embedded-trainings.ferrous-systems.com/preparations.html
  - https://embedded-trainings.ferrous-systems.com/tooling-check.html

- starter code and slides are here
  - https://github.com/ferrous-systems/embedded-trainings-2020

Agenda

- `no_std` programs
- Embedded Rust tooling
- Using a Hardware Abstraction Layer
- Using the Radio on the nRF52840 to solve a puzzle

The hardware

- nRF52840 Development Kit
  - USB port J2: J-Link debugger
  - Connect a cable to it
- nRF52840 Dongle
  - No on-board debugger

nRF52840

- ARM Cortex-M4F processor
- 1 MB of Flash
- 256 KB of RAM
- USBD: USB 2.0 Full-Speed device
- RADIO: IEEE 802.15.4 and Bluetooth Low Energy compatible

Parts of a `no_std` program

- Book: section 3.1
- Folder: `beginner/apps`
- File: `src/bin/hello.rs`
- `#![no_std]`: `std` API is not available but `core` is
- `#![no_main]`: custom entry point
- divergent `main` function

Cross compiling

- Book: section 3.2
- Folder: `beginner/apps`
- File: `src/bin/hello.rs`
- `cargo build --bin hello`
- Compilation target is in `.cargo/config.toml`
- Output ELF is in `target/thumbv7em-none-eabi/debug`

Analysis: Binary size

- Book: section 3.3
- Folder: `beginner/apps`
- File: `src/bin/hello.rs`
- Do NOT measure file size
- `cargo size --bin hello`
  - First+second column is size in Flash
  - Second+third column is static RAM usage

Running a program

- Book: section 3.4
- Folder: `beginner/apps`
- File: `src/bin/hello.rs`
- Custom Cargo runner: `probe-run`
- ✅ Click "Run" button in VS code
  - (or run `cargo run --bin hello` if not using VS Code)
- on a breakpoint the Cargo runner prints a stack backtrace and exits
- ✅ Try changing the log statement and re-running the program

Panicking behavior

- Book: section 3.5
- Folder: `beginner/apps`
- File: `src/bin/panic.rs`
- No default behavior in `no_std` programs
- Must pick one
  - Use a panic handler crate like `panic-log`
  - Or write a `#[panic_handler]` function
- ✅ try changing `panic_log`'s `#[panic_handler]` function

Hardware Abstraction Layer (HAL) - LED

- Book: section 3.6
- Folder: `beginner/apps`
- File: `src/bin/led.rs`
- Run: `cargo doc -p dk --open` -- HAL API documentation
- `Led.on` and `Led.off` control the LEDs
- ✅ try turning on/off different LEDs
- ✅ try uncommenting the `set_log_level` statement

HAL - Timer

- Book: section 3.7
- Folder: `beginner/apps`
- File: `src/bin/blinky.rs`
- `Timer.wait` can be used to create delays
- ☑ try changing the delay value

Using the dongle

- Book: section 3.8
- Folder: `boards/dongle`
- ✅ Disconnect the DK board for now
- ✅ Press reset button on the Dongle to put it in bootloader mode
- The Dongle will pulsate its red LED in bootloader mode
- Run: `dongle-flash loopback.hex`
- Run: `serial-term` to display the Dongle's logs
- ✅ check for interference; use `change-channel` if there is

Radio out

- Book: section 3.9
- Folder: `beginner/apps`
- File: `src/bin/radio-send.rs`
- ✅ reconnect the DK
- Check `serial-term` for new output
- LQI: Link Quality Indicator. Higher = better
- ✅ Try:
  - Using a different Channel
  - Changing the TX power
  - Increasing the distance between the DK and the dongle

Radio in

- Book: section 3.10
- Folder: `beginner/apps`
- File: `src/bin/radio-recv.rs`
- The Dongle responds to each incoming packet
- The response contains the received data but reversed
- ✅ Try: inserting a delay between `send` and `recv_timeout`

Reflashing the Dongle

- Book: section 3.11
- Press the reset button on the Dongle to put it in bootloader mode
- Run: `dongle-flash puzzle.hex`
- Run: `serial-term`
- Check: `serial-term` output should have "`app=puzzle.hex`"
- Also note that the channel has changed

Radio puzzle

- Book: section 3.11
- Folder: `beginner/apps`
- File: `src/bin/radio-puzzle.rs`
- Dongle holds a string encrypted via single-letter substitution
- Your task is to decrypt it
- Dongle's response depends on packet size
  - 0 bytes: answers with encrypted string
  - 1 byte: mapping from plaintext letter to the ciphertext letter
  - Else: answers with "correct" if the packet contained the decrypted string

Radio puzzle help

- Book: section 3.12
- Suggested steps
  1. Send a 1 letter packet to the radio to get a feel for how the mapping works
  2. Get familiar with the dictionary API. Do some insertions and look ups
  3. Get mappings from the radio and insert them into the dictionary
  4. Get familiar with the buffer API; plaintext will go in a separate buffer
  5. Retrieve the ciphertext from the Dongle; get familiar with iterating it
  6. Do the reverse mapping to decrypt the message
  7. Send plaintext to the Dongle for confirmation
- There are incremental solutions to these steps in `src/bin`

Things for you to check out

- Book: section 3.13
- 802.15.4 experiments: energy detection, collision avoidance and WiFi coexistence
  - See section 3.14 of the workbook for details
- Memory safe interrupt handling
  - Check the concurrency chapter of the embedded Rust book
  - Check the Real-Time Interrupt-driven Concurrency (RTIC) framework