Project Structure

How the code is structured and organized

Layout

Code Structure

Architecture (Summary)

Each board's `main.c` acts strictly as an orchestrator. It initializes the runtime, creates tasks, and delegates all functional behavior to component modules.

Core Design Contract

The repository enforces a strict separation between **entrypoints** and **components**:

- `src/<board>/main.c` defines the process entrypoint for each board target.
- `components/common/*` contains reusable logic shared across multiple boards.
- `components/<board>/*` contains board-specific functionality.
- `components/<board>/firmware/*` contains generated or vendor-provided firmware and MCU integration code.

Primary Rule

> `main.c` must not contain domain logic. It is responsible only for system wiring and task startup.
>
> - Albert Einstein

Repository Layout

```
erc/
├─ src/
│  ├─ arm_board/main.c
│  ├─ driving_board/main.c
```

```
│   ├─ sensor_board/main.c
│   ├─ network_board/main.c
│   └─ debugging_board/main.c
│
├─ components/
│   ├─ common/              # Shared modules across boards
│   ├─ arm_board/           # Arm board-specific modules
│   ├─ driving_board/       # Driving board-specific modules
│   ├─ sensor_board/        # Sensor board-specific modules
│   ├─ network_board/       # Network board-specific modules
│   └─ debugging_board/     # Debugging board-specific modules
│
├─ test/
│   ├─ common/
│   ├─ arm_board/
│   ├─ driving_board/
│   ├─ sensor_board/
│   └─ debugging_board/
│
├─ scripts/                # Utility scripts (codegen, post-processing)
└─ platformio.ini          # Build environments and board filters
```

Entrypoint Responsibilities (`src/<board>/main.c`)

The `main.c` file is intentionally minimal and should perform only the following:

1. Execute mandatory low-level initialization
   (HAL, clock, cache, MPU, RTOS initialization as required)
2. Initialize infrastructure dependencies
   (GPIO, UART, timers, networking wrappers, etc.)
3. Create one or more RTOS tasks
4. Start the scheduler/kernel
5. Delegate all functional behavior to components

What Must NOT Be Implemented in `main.c`

The following must never reside in `main.c`:

- Sensor processing algorithms
- Business or control logic
- Packet parsing or dispatch policy
- Device-specific runtime behavior beyond initialization
- Long-running loops implementing application logic

If logic grows beyond simple initialization or task creation, it must be moved into a component module and invoked from a task.

Component Responsibilities (`components/*`)

All functional behavior belongs in components. Tasks must delegate to components rather than implementing logic inline.

Examples

- Sensor-related behavior → `components/sensor_board/*`
  (e.g., GPS, IMU, pH sensors, acquisition pipelines)
- Driving logic → `components/driving_board/*`
  (e.g., motor control, calculations, protocol parsing, Simulink integration)
- Debugging UI and diagnostics → `components/debugging_board/*`
- Shared infrastructure → `components/common/*`
  (e.g., result handling, logging, queues, dispatch systems)

Execution Model

The execution flow for each board follows a consistent structure:

```
main.c
  → platform/runtime initialization
  → create RTOS task(s)
  → each task calls component APIs
  → component modules execute all functional logic
```

This ensures that:

- `main.c` remains stable and minimal

- behavior is modular and testable
- functionality is reusable across boards

Post-Generation Scripts

Introduction

We use one post code generation script. We do this because we do not want to write code inside the auto-generated code, and this helps with that. If you are on Linux or (possibly, untested but likely) mac, you can refer to the script in the cubeMX software by going into Project Manager -> Code Generator -> User Actions -> After Code Generation. If you are not on a UNIX based system you have to run it every time after generating code manually from the folder where the file is located. The file is located under scripts and is called post_code_generation.bash. For mac, the script is called post_code_generator_mac.bash, because there are some small formatting changes, more explained in Mac Changes.

Functions

The script has 4 different functions

Renaming main files

It starts by renaming all main.h and main.c files that are in components. It also saved all boards that have main.c files, because those are the newly generated boards. This way you don't do certain actions on the files twice, if you would run the script again.

```
while IFS= read -r FILE; do
  # Extract the basename (filename without path)
  base="$(basename "$FILE")"

  if [[ "$base" == "main.c" ]]; then

    subdir="${FILE#"$BASE"/}" # Path from the board dir
    main_dir="${subdir%%/*}" # The board dir
    GENERATED_BOARDS+=("$main_dir") # Gets all boards that are generated again, and thus have
a main

    dir=$(dirname "$FILE") # Get directory of the file
    mv "$FILE" "$dir/cubemx_main.c"
    echo "Renamed $FILE to $dir/cubemx_main.c"
  fi
```

```
  if [[ "$base" == "main.h" ]]; then
    dir=$(dirname "$FILE") # Get directory of the file
    mv "$FILE" "$dir/cubemx_main.h"
    echo "Renamed $FILE to $dir/cubemx_main.h"
  fi
done < <(find "$BASE" -type f)
```

Adding firmware definitions

Certain constants might have to be set in the main.h files from cubemx. This happens most likely because of the order in which the files are build, but I am not totally sure. However, if you do need to have some constants set in the main.h file, you can at them to any file in the folder called "firmware_definitions" in the common folder of components. This second code block adds it to the .h file.

```
find "$BASE" -type d -name firmware_definitions | while read -r FW_DIR; do
    BOARD_DIR_PATH="$(dirname "$FW_DIR")"
    BOARD_DIR="${BOARD_DIR_PATH#"$BASE"/}"
    if printf '%s\n' "$COMMON_COMPONENT" "${GENERATED_BOARDS[@]}" | grep -Fx "$BOARD_DIR" >
/dev/null; then
      BOARD_DIR_PATHS=("$BOARD_DIR_PATH")

      if [[ "$BOARD_DIR" == "$COMMON_COMPONENT" ]]; then
        BOARD_DIR_PATHS=(${GENERATED_BOARDS[@]/#/"$BASE"/})
      fi
      for BOARD_DIR_PATH in "${BOARD_DIR_PATHS[@]}"; do

        CUBEMX_FILE="$BOARD_DIR_PATH/firmware/Core/Inc/cubemx_main.h"

        # Skip if cubemx file does not exist
        [[ -f "$CUBEMX_FILE" ]] || continue

        echo "Appending firmware_definitions to: $CUBEMX_FILE"

        TMP_FILE="$(mktemp)"

        head -n -1 "$CUBEMX_FILE" >> "$TMP_FILE"

        echo -e "\n/* ---- START firmware_definitions ---- */\n" >> "$TMP_FILE"
        find "$FW_DIR" -type f -exec cat {} \; >> "$TMP_FILE"
        echo -e "\n/* ---- END firmware_definitions ---- */\n" >> "$TMP_FILE"
```

```
        tail -n -1 "$CUBEMX_FILE" >> "$TMP_FILE"



        # 3) Replace original file
        mv "$TMP_FILE" "$CUBEMX_FILE"
      done
    fi
done
```

Adding static wrappers

Some functions generated by cubemx you do need, but they are static so you cannot use them outside of the main file. To still be able to do that, the script adds wrappers for those files.

```
find "$BASE" -type f -name "cubemx_main.c" | while read -r FILE; do
  subdir="${FILE#"$BASE"/}" # Path from the board dir
  BOARD_DIR="${subdir%%/*}" # The board dir
  if printf '%s\n' "${GENERATED_BOARDS[@]}" | grep -Fx "$BOARD_DIR" > /dev/null; then
    TMP_FILE="$(mktemp)"
    echo "READING $FILE"
    while read -r line; do
        echo "$line" >> "$TMP_FILE"
        if [[ "$line" =~ ^[[:space:]]*static[[:space:]]+[a-zA-Z_][a-zA-Z0-9_]*[[:space:]]+[a-
zA-Z_][a-zA-Z0-9_]*\([^\)]*\)\;[[:space:]]*$ ]]; then        # Remove 'static' and trailing
';'
          echo "Static function found: $line"
          proto=$(echo "$line" | sed -E 's/^[[:space:]]*static[[:space:]]+//;
s/;[[:space:]]*$//')

          # Extract function name
          name=$(echo "$proto" | sed -E 's/.*[[:space:]]+([a-zA-Z_][a-zA-Z0-9_]*)\(.*/\1/')

          # Extract return type
          ret=$(echo "$proto" | sed -E "s/[[:space:]]+$name\(.*//")

          # Extract argument list
          args=$(echo "$proto" | sed -E "s/.*$name\((.*)\)/\1/")
```

```
          # Build argument names (remove types)
          call_args=$(echo "$args" | sed -E 's/[a-zA-Z_][a-zA-Z0-9_]*[[:space:]]+//g')
          if [[ "$call_args" == "void" ]]; then
            call_args=""
          fi

          echo "$ret ${name}_wrapper($args) {" >> "$TMP_FILE"
          if [[ "$ret" == "void" ]]; then
              echo "    $name($call_args);" >> "$TMP_FILE"
          else
              echo "    return $name($call_args);" >> "$TMP_FILE"
          fi
          echo "}" >> "$TMP_FILE"
          echo >> "$TMP_FILE"
          echo "added wrapper for static function ${name} in ${FILE}"
        fi
    done < "$FILE"
    mv "$TMP_FILE" "$FILE"
  fi
done
```

Changing the includes

Because of the name change from main.c/h, to cubemx_main.c/h, the includes are now wrong. This last code block changes all the includes to the right name.

```
while IFS= read -r FILE; do
  subdir="${FILE#"$BASE"/}" # Path from the board dir
  BOARD_DIR="${subdir%%/*}" # The board dir
  if printf '%s\n' "${GENERATED_BOARDS[@]}" | grep -Fx "$BOARD_DIR" > /dev/null; then
    sed -i 's/#include "main.h"/#include "cubemx_main.h"/g' "$FILE"
    echo "Updated include in $FILE"
  fi
done < <(grep -rl '#include "main.h"' ../components/)
```

MAC Changes

1) Added #! /usr/bin/env bash in the first line

2) changed `sed -i 's/#include "main.h"/#include "cubemx_main.h"/g' "$FILE"`

to `sed -i '' 's/#include "main.h"/#include "cubemx_main.h"/g' "$FILE"` because mac uses a different version of sed.

Simple PIOC

Introduction

This Python script processes a custom PlatformIO configuration file (`platformio.pioc`) and generates a standard `platformio.ini` file.

It extends PlatformIO's configuration capabilities by:

- Supporting **dynamic include paths** using glob patterns
- Extracting **C preprocessor defines** from board-specific Makefiles
- Expanding **custom syntax** into valid `build_flags`
- Resolving **absolute paths**.

Key Features

Custom build_flags Processing

Supports two types of entries:

- **+<pattern>**: Include directories
- **-<pattern>**: Exclude directories
- Other entries are treated as standard compiler flags

Include Path Resolution

Glob patterns are expanded into directory paths using recursive search.

Board-specific C Defines

Defines are extracted from:

```
components/<board>/firmware/Makefile
```

The script looks for a `C_DEFS` section and includes all compiler defines.

This is done, because cubeMX generates important definitions in the auto-generated makefile. These are not used if we don't copy them to the .ini file.

Environment Detection

```
[env:my_board]
```

This determines which board folder is used.

Get Absolute Path

For some functions, like nanopb, you might need the absolute path. There is no way to get in the default platformio.ini file, so that would mean that you would have to hard code it. We do not want that, so we have a placeholder for an absolute path.

The placeholder:

```
${{project_absolute_path}}$
```

is replaced with the absolute path of the project.

Workflow

1. Read `platformio.pioc`
2. Detect environment
3. Parse `build_flags`
4. Resolve glob patterns
5. Extract C defines
6. Write `platformio.ini`
7. Replace placeholders

Example Input

```
[env:my_board]
build_flags =
    +<lib/**>
    -<lib/exclude/**>
    -DDEBUG
```

Example Output

```
[env:my_board]
build_flags =
    -I lib/module1
    -I lib/module2
    -DDEBUG
    -DDEFINE_FROM_MAKEFILE
```

.pioc file

Introduction

The `platformio.pioc` file is the central configurationhere file used to define build environments, dependencies, compiler flags, and project structure. It uses a format with sections and key-value pairs. It is similar to platformio.ini, which is actually used by platformio, but has some changes to use it easily with our project. For more information, read Simple PIOC.

File Structure

The configuration is divided into sections such as `[platformio]`, `[env]`, and environment-specific sections like `[env:network_board]`.

Core Sections

[platformio]

Defines global project settings.

- `default_envs`: Default environment(s) to build.
- `src_dir`: Source directory.
- `lib_dir`: Library directory.

[extra]

Custom user-defined variables for reuse.

[env]

Base configuration shared across all environments.

Environment Sections

Each `[env:<name>]` defines a specific build target. These inherit from `[env]`.

Custom Enhancements

1. Glob Patterns in build_flags

Unlike standard PlatformIO, this configuration allows glob-style include/exclude patterns directly in `build_flags` using `+<...>` and `-<...>` syntax.

```
build_flags=
    +<components/network_board/**>
    -<components/network_board/firmware/Drivers/**>
```

This enables fine-grained control over which directories are included in compilation.

2. Absolute Path Variable

The variable `${{project_absolute_path}}$` expands to the absolute path of the project root.

```
custom_nanopb_project_dir = ${{project_absolute_path}}$/ERC-Protobufs
```

Source Filtering

`build_src_filter` defines which source files are compiled. It supports inclusion (+) and exclusion (-) rules.

```
+<src/${this.__env__}/**/*.c>
-<components/${this.__env__}/firmware/Drivers/*>
```

Variable Substitution

- `${extra.common_lib_deps}`: Reference shared variables.
- `${this.__env__}`: Current environment name.

Library Dependencies

External libraries can be defined using Git URLs or registry references.

```
lib_deps = https://github.com/nanopb/nanopb.git#commit
```

Compiler and Linker Flags

Standard compiler flags are also supported alongside glob patterns.

```
-mthumb
-mfpu=fpv4-sp-d16
-D CONFIG_LOG_LEVEL=LOG_INFO
```

Example Configuration

```
[env:network_board]
board = nucleo_h753zi
build_flags=
    +<components/network_board/**>
    -<components/network_board/firmware/Drivers/**>
```

```
    -mthumb
    -D CONFIG_LOG_LEVEL=LOG_INFO
```

Compilation

To use the file, you have to convert it to a `platformio.ini` file. You can do that by running

```
python3 simple_pioc.py
```

Extra information

If more information is needed, look at the documentation specifically for platformio.ini files. You can find it here.