Transmitter Firmware Code Organisation

Bilal Ahmed / Team Lead

December 26, 2025

1 Overview

This document describes the organisation of the transmitter firmware source code for the fibre-optic audio demo kit. The codebase is structured into clearly separated layers:

- **Controller layer** (`controller/`): top-level orchestration and interrupt service routines (ISRs).
- **Driver layer** (`driver/`): hardware-oriented modules for ADC, PWM (Timer0), sampling trigger (Timer1), and mapping.
- **Mapping layer** (`driver/mapping/`): conversion from ADC readings to PWM duty-cycle using a lookup table (LUT).
- **Tools** (`tools/`): host-side utilities used during the build process (e.g. LUT generation).
- **Build system** (`makefile`): defines the build, link, LUT-generation and flash workflow.

The goal of this structure is to separate "what" the system does (controller behaviour and data flow) from "how" it talks to the hardware (drivers), and from purely numerical mapping logic (LUT).

2 Directory Structure

The transmitter firmware source tree is organised as follows:

```
src/
├── controller/
│   ├── controller.c
│   ├── controller.h
│   └── main.c
├── driver/
│   ├── adc/
│   │   ├── adc.c
│   │   └── adc.h
│   ├── mapping/
│   │   ├── adc_lut.inc
│   │   ├── mapping.c
│   │   └── mapping.h
│   ├── pwm/
│   │   ├── timer0.c
│   │   └── timer0.h
│   └── trigger/
│       ├── timer1.c
│       └── timer1.h
├── main.elf
├── main.hex
├── makefile
└── tools/
    ├── gen_lut
    └── gen_lut.c
```

Object files (`*.o`), the linked ELF (`main.elf`) and the Intel HEX image (`main.hex`) are build artefacts generated by `make` and are not treated as source modules.

3 Layer 1: Controller

3.1 `controller/main.c`

Role Program entry point. Contains the `main()` function and nothing hardware-specific.

Responsibilities

- Call the high-level initialisation function `controller_init()`.
- Enter the main superloop (currently an idle loop, with the CPU optionally put into IDLE sleep mode).

Key API

- `int main(void);`
  - Calls `controller_init()` once.
  - Enters `while (1){};` no application logic is run in the foreground; the system is interrupt-driven.

3.2 `controller/controller.h`

Role Public interface for the controller layer. Exposes the top-level initialisation routine to `main.c` and includes the interrupt header.

Responsibilities

- Declare the controller initialisation function.
- Provide a single include point for any future controller-wide definitions.

Key Declarations

- `int controller_init(void);`

3.3 `controller/controller.c`

Role Central orchestrator of the firmware. It:

- Calls all driver initialisation functions.
- Enables global interrupts.
- Owns the time-critical ISRs that implement core functionality:
  - Timer1 Compare Match ISR: starts a new ADC conversion.
  - ADC Conversion Complete ISR: maps the result to a PWM duty-cycle and updates Timer0.

Responsibilities

- Configure the system via:
  - `timer0_init()` (PWM carrier).
  - `adc_init()` (ADC front-end).
  - `timer1_init()` (sampling trigger).
- Enable interrupts using `sei()` once all drivers are initialised.
- Implement:
  - `ISR(TIMER1_COMPA_vect)`: start ADC conversion at each sample tick.
  - `ISR(ADC_vect)`: read latest ADC value and update PWM using the mapping LUT.

Key API (internal to the module)

- `int controller_init(void);`
  - Returns `0` on success, non-zero if any driver initialisation fails.
  - Calls:
    - `timer0_init();`
    - `adc_init();`
    - `timer1_init();`
  - Enables global interrupts with `sei()` on success.
- `ISR(TIMER1_COMPA_vect)`
  - Calls `adc_start()` to begin a new conversion at the configured sample rate.
- `ISR(ADC_vect)`
  - Reads the 10-bit ADC result via the `ADC` data register.
  - Calls `adc_map()` from the mapping module to convert ADC code to an 8-bit PWM value.
  - Calls `set_duty()` from the Timer0 driver to update the PWM duty-cycle.

4 Layer 2: Drivers

The driver layer encapsulates low-level hardware configuration and provides small, clear C APIs to the controller. Each driver has its own header (`.h`) and source (`.c`) pair.

4.1 ADC Driver: `driver/adc/adc.*`

4.1.1 `adc.h`

Role Public interface to the ADC hardware driver.

Responsibilities

- Declare the ADC initialisation and conversion-start functions.
- Declare any shared state used by the controller ISRs (e.g. `adc_last` if present).

Key Declarations

- `int adc_init(void);`
  - Configures ADC reference, channel selection, prescaler and enables ADC interrupts.
- `void adc_start(void);`
  - Starts a single ADC conversion by setting the ADSC bit.

4.1.2 `adc.c`

Role Implementation of the ADC driver.

Responsibilities

- Configure the ADC:
  - Input channel (e.g. ADC0).
  - Voltage reference (e.g. AVCC).
  - Prescaler satisfying ADC clock constraints.
  - ADC enable and interrupt enable.
- Optionally disable digital input on the ADC pin via `DIDR0` to reduce power and noise.
- Provide a simple `adc_start()` wrapper that initiates conversion.

4.2 PWM Driver (Timer0): `driver/pwm/timer0.*`

4.2.1 `timer0.h`

Role Public interface to the Timer0 PWM output driver.

Key Declarations

- `int timer0_init(void);`
  - Configures Timer0 in 8-bit Fast PWM mode.
  - Sets the OC0A pin (e.g. PB3) as an output.
- `void set_duty(uint8_t pwm);`
  - Updates `OCR0A` with a new 8-bit duty-cycle value.

4.2.2 `timer0.c`

Role Implementation of PWM carrier generation.

Responsibilities

- Configure Timer0:
  - Set waveform generation mode to 8-bit Fast PWM (`WGM00` and `WGM01` bits).
  - Configure compare output mode to "clear OC0A on compare match" (`COM0A1` set).
  - Select prescaler (currently no prescale, `CS00` set) to achieve the desired PWM carrier frequency.
- Implement `set_duty()` as a simple write to `OCR0A`.

4.3 Sampling Trigger Driver (Timer1): `driver/trigger/timer1.*`

4.3.1 `timer1.h`

Role Public interface to the Timer1-based sampling trigger.

Key Declarations

- `int timer1_init(void);`

4.3.2 `timer1.c`

Role Implementation of Timer1 configuration for sample timing.

Responsibilities

- Configure Timer1 in CTC mode with top value in `OCR1A`.
- Set the prescaler (e.g. no prescaler) to achieve the desired sample rate.
- Enable the Output Compare Match A interrupt (`OCIE1A`) via `TIMSK1`.
- Leave the ISR itself in the controller layer to keep hardware configuration and data flow separated.

5 Layer 3: Mapping

5.1 `driver/mapping/mapping.h`

Role Public interface for the ADC-to-PWM mapping logic.

Responsibilities

- Declare the mapping function.
- Define the maximum ADC code constant used for clamping (e.g. `ADC_MAX`).

Key Declarations

- `#define ADC_MAX ...` (or equivalent constant):
  - Defines the highest ADC code treated as full-scale in the mapping.
- `uint8_t adc_map(uint16_t adc);`
  - Maps a 10-bit ADC code to an 8-bit PWM value using the LUT, with clamping.

5.2 `driver/mapping/mapping.c`

Role Implementation of the mapping logic and lookup table.

Responsibilities

- Declare the LUT as a `static const` array in program memory:

```
static const uint8_t adc_to_pwm[ADC_MAX + 1] PROGMEM = {
#include "adc_lut.inc"
};
```

- Implement `adc_map()`:
  - Clamp input `adc` to `ADC_MAX`.
  - Use `pgm_read_byte()` to read the corresponding PWM value from flash.

Key API

- `uint8_t adc_map(uint16_t adc);`

5.3 `driver/mapping/adc_lut.inc`

Role Auto-generated include file containing the comma-separated initialiser values for the LUT.

Responsibilities

- Provide a syntactically valid list of `uint8_t` constants to initialise `adc_to_pwm[]`.
- The number of entries must be consistent with `ADC_MAX + 1`.

Generation

- `adc_lut.inc` is not edited by hand; it is generated by the host-side tool `tools/gen_lut` during the build.

6 Tools: LUT Generator

6.1 `tools/gen_lut.c`

Role Host-side C program (compiled with the host compiler, e.g. `gcc`) that generates the mapping LUT at build time.

Responsibilities

- Loop over ADC codes from `0` to `ADC_MAX` inclusive.
- Compute a corresponding PWM value (e.g. linear mapping from `0..ADC_MAX` to `0..255`, or a more complex curve).
- Print the PWM values as comma-separated integers so the output can be included directly into the array initialiser in `mapping.c`.

Key Behaviour

- The program outputs to `stdout`; the Makefile redirects this into `driver/mapping/adc_lut.inc`.
- The mapping logic inside `gen_lut.c` must be kept consistent with the assumptions in the mapping layer.

6.2 `tools/gen_lut`

Role Compiled binary of `gen_lut.c` for the development host. Called by the Makefile as part of the build process.

7 Build System: `makefile`

7.1 Role

The `makefile` defines how to:

- Compile AVR source files to object files.
- Link them into an ELF firmware image.
- Convert the ELF to an Intel HEX file.
- Generate the LUT include file.
- Flash the microcontroller via ISP or bootloader.

7.2 Key Variables

Key variables to be used in the Makefile include:

- `MCU`: target microcontroller (e.g. `atmega164p`).
- `F_CPU`: CPU clock frequency (e.g. `20000000UL`).
- `CC`: AVR compiler executable (e.g. `avr-gcc`).
- `OBJCPY`: object copy utility (e.g. `avr-objcopy`).
- `HOSTCC`: host compiler for `tools/gen_lut` (e.g. `gcc`).
- `CFLAGS`: common compiler flags (e.g. `-mmcu=$(MCU) -DF_CPU=$(F_CPU) -Os -Wall`).
- `OBJS`: list of object files to link (`controller/controller.o`, `driver/adc/adc.o`, etc.).
- `PORT`, `BAUD`: serial port and baud rate for bootloader-based flashing.

7.3 Important Targets

`all` Default target. Builds `main.hex` from source by:

1. Building `tools/gen_lut` (host tool).
2. Generating `driver/mapping/adc_lut.inc`.
3. Compiling all `.c` files into `.o`.
4. Linking into `main.elf`.
5. Converting `main.elf` to `main.hex`.

`tools/gen_lut`

Compiles `tools/gen_lut.c` into the host binary:

```
tools/gen_lut: tools/gen_lut.c
    $(HOSTCC) -o $@ $<
```

`driver/mapping/adc_lut.inc`

Generates the LUT include file:

```
driver/mapping/adc_lut.inc: tools/gen_lut
    ./tools/gen_lut > $@
```

This rule ensures `adc_lut.inc` is regenerated whenever `gen_lut` changes.

main.elf

Links all object files into a single ELF:

```
main.elf: $(OBJS)
    $(CC) $(CFLAGS) -o $@ $(OBJS)
```

main.hex

Converts ELF to HEX for flashing:

```
main.hex: main.elf
    $(OBJCPY) -O ihex -R .eeprom $< $@
```

flashISP

Flash via ISP programmer (e.g. USBasp) using `avrdude`.

flash Flash via serial bootloader (e.g. CH340 + Arduino bootloader) using `avrdude` with `-c arduino` protocol.

clean Remove build artefacts (ELF, HEX, objects, LUT, tools binary).

8 Summary

The codebase is explicitly layered to:

- Keep time-critical ISRs and overall data flow in the controller layer.
- Encapsulate low-level hardware details inside small, testable driver modules.
- Separate numerical mapping logic and LUT generation from both controller and hardware.
- Automate LUT generation via a host-side tool integrated into the Makefile.

This organisation is intended to make the firmware easier to understand, document, and extend, while providing clear module boundaries for different team members to own and develop.