2025 ACC Firmware Documentation

Nov 2025
Rev. 1
Author & Project Head: Franco Heraud

Signed off by: _____

Contents

1 Introduction

1.1 Overview

The Accumulator Cooling Controller (ACC) firmware is responsible for regulating the accumulator thermal management system on the 2025 UWA Motorsport electric vehicle. The microcontroller must acquire measurements from a distributed sensor network, coordinate with the accumulator management system (AMS) over CAN, and drive the cooling hardware while satisfying the Formula SAE safety constraints. This document captures the architectural intent of the firmware and the rationale behind critical design decisions so future contributors can iterate with confidence.

1.2 Document Structure

Section 2 outlines the embedded architecture and execution model that the firmware implements. Sections 4 and 5 cover device drivers and control logic respectively, while Section 6 documents the verification strategy and available tooling. Each subsection is designed to be consumed independently so that developers working on a specific module can quickly locate the relevant background information and implementation notes.

1.3 Requirements

The main functional requirements for the onboard MCU are as follows:

- **Drivers**
  - Must be dual-CAN compatible (one for data-logging and the other for receiving temperature data from the AMS)
  - Preferred to use DMA ADC reads
- **Sensors**
  - Must perform ADC reads on the 4x temperature sensors and 4x pressure sensors and perform the necessary sensor input conversions
  - Must receive additional segment temperature data from the CAN bus
  - Must accept a tachometer input from one of the fans
  - Must perform another ADC read from the power input pins connected to the switch circuit to measure the current power consumption of the synchronous buck controller
- **Control System + Other Considerations**
  - The fan speed must be regulated based off the sensor and tachometer inputs as inputs to cool the accumulator

- Preferred to use a PID controller on top of basic threshold logic for controlling the fan speed
- Sensor and power consumption data must be transmitted over the CAN bus for data-logging purposes

Any feedback + additional notes would be greatly appreciated.

2 Firmware Architecture

2.1 Execution Model

The firmware executes on an STM32F412 microcontroller using the HAL abstraction layer generated via STM32CubeMX. A cooperative scheduler running from the `main()` loop coordinates initialization, periodic polling tasks, and interrupt driven updates. Time-critical paths such as tachometer capture and ADC sampling leverage hardware peripherals with DMA to minimise CPU latency and ensure deterministic response.

2.2 Data Flow

Figure 1 (conceptual) illustrates the primary data paths. Sensor inputs enter the system either as analogue voltages sampled through `ADC2` or as CAN frames received via `FDCAN1`. The data is normalised into the `SensorInputs_t` structure, which is the single source of truth for downstream control modules. Control outputs are subsequently published to the fan drivers (PWM) and to the vehicle network through CAN transmissions.

Figure 1: High level firmware data flow between sensors, processing, and actuators.

2.3 Safety Hooks

All actuator facing code paths must pass through the `acc_safety.c` guard rails. These hooks enforce sanity ranges, fail-safe timeouts, and diagnostics counters. Developers adding new outputs should extend the guard rails instead of bypassing them to maintain the safety case for the accumulator cooling package.

3 Sensors

3.1 Fan RPM reading logic

The main fan RPM calculation logic will utilize a GPIO pin configured as an external interrupt and an internal hardware timer that uses a counter to calculate the pulse frequency in Hz and,

hence, the fan RPM. Firstly, we alter the clock configuration settings to ensure that the input clock frequency used by the timer (TIM2 in this case) is 1 MHz. Where the clock is being divided from the input 8 MHz clock.

Then, if we are using a 32-bit counter resolution, the counter period is $= 2^{32}$ ticks. This is the number of ticks before the timer overflows, which occurs at:

$$T_{overflow} = \frac{2^{32}}{10^6} \approx 71.58 \text{ minutes}$$

Essentially, there is $\Delta t = 1/1\text{MHz} = 1\mu s$ per tick. So we just need to count the number of ticks per digital pulse supplied by the tachometer input and find the pulse period using:

$$T_{pulse} = \Delta t \times (\text{Count Between Pulses})$$

Thus, we derive the pulse frequency in Hz using:

$$f_{pulse} = \frac{1}{T_{pulse}} = \frac{1}{1\mu s \times \text{Count Between Pulses}} = \frac{10^6 \text{ Hz}}{\text{Count Between Pulses}}$$

The core logic implemented through an external interrupt and an internal hardware timer is as follows:

```
const uint8_t PULSES_PER_REVOLUTION = 1;

static volatile uint32_t tach_last_ticks  = 0;
static volatile uint32_t tach_delta_ticks = 0;
static volatile uint8_t  tach_new_period  = 0;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == TACH_IN_Pin) {
        uint32_t now = __HAL_TIM_GET_COUNTER(&htim2);

        uint32_t delta = now - tach_last_ticks;
        tach_last_ticks = now;

        if (delta > 0) {
            tach_delta_ticks = delta;
            tach_new_period  = 1;
        }
    }
}

void Update_Fan_Speed(SensorInputs_t *si) {
  if (!tach_new_period || tach_delta_ticks == 0) si->fan_rpm = 0.0f;
  const float tick_freq_hz = (float)pow(10, 6);
  float pulse_freq_hz = tick_freq_hz / (float)tach_delta_ticks;
```

```
  si->fan_rpm = (pulse_freq_hz * 60) / (float)PULSES_PER_REVOLUTION;
}
```

3.2 Temperature sensor reading logic

Each coolant loop segment is instrumented using thermistors packaged with the fans. The device behaves as a 10 kΩ NTC element forming a voltage divider against a precision pull-up resistor. The ADC reading therefore reflects the thermistor resistance, which in turn maps non-linearly to temperature. The conversion pipeline implemented in `sensor_inputs.c` performs the following steps for every DMA sample:

1. Convert the raw ADC count to an input voltage based on the configured reference voltage and ADC resolution.
2. Derive the thermistor resistance via $R_{\text{therm}} = R_{\text{pull-up}} \left(\frac{V_{\text{ref}}}{V_{\text{adc}}} - 1 \right)$.
3. Apply the Steinhart–Hart approximation using the calibration constants provided in the device data sheet to compute the temperature in Kelvin.
4. Convert to degrees Celsius and clamp the value to the valid operating range reported by the sensor vendor to guard against transient spikes.

The coefficients A, B, and C of the Steinhart–Hart model are stored in firmware so they can be updated as the hardware team refines the sensor selection. A lookup-table shortcut is also supported for low-power modes; the firmware selects between the analytical and table driven approaches based on a compile-time flag.

3.3 Pressure sensor reading logic

Four analogue pressure transducers monitor coolant circuit pressure and the accumulator inlet manifold. The transducers provide ratiometric outputs, therefore the ADC conversion first normalises the reading against the current rail voltage sampled through the housekeeping ADC channel. A linear calibration derived from bench testing translates the normalised value into kilopascals. Calibration data lives in `pressure_calibration.h` so that each chassis can ship with its specific offsets.

4 Device Drivers

4.1 CAN Interface

The CAN driver exposes a double-buffered mailbox API that separates application level frame composition from the low-level HAL interactions. Outgoing frames are enqueued into a ring

buffer and flushed inside the `HAL_CAN_TxMailbox1CompleteCallback` handler. This design prevents the control loop from blocking whenever the transceiver is busy. Received frames are parsed into structured messages defined in `acc_messages.h`. Adding a new CAN topic requires updating the message definition, extending the parser, and documenting the payload format in Section 7.

4.2 PWM Fan Driver

PWM generation is performed by `TIM1` using complementary channels with dead-time insertion disabled. The driver abstracts timer register programming into `FanDriver_SetDutyCycle()` which validates the requested duty cycle against calibrated minimum and maximum values. Interrupt safe updates allow the control algorithm to adjust speeds from either the main loop or PID callbacks without data races.

5 Control Logic

5.1 Thermal Control Strategy

The firmware implements a two-tier strategy. A hysteresis layer prevents rapid toggling of the fans around the thermal limits, while a PID controller refines the duty cycle to track the temperature set-point broadcast by the AMS. The PID loop is computed every 50 ms inside the sensor processing task. The input to the controller is the hottest temperature reported by either the hard-wired thermistors or the CAN-fed segment probes. The output is a duty-cycle request bounded by the fan driver guard rails.

5.2 Fail-safe Behaviour

If the tachometer reports a zero RPM reading while the duty cycle command exceeds 40%, the controller triggers a diagnostic fault and ramps to maximum speed for redundancy. The fault is latched and mirrored on the CAN telemetry stream so that the vehicle controller can initiate backup cooling if required.

6 Testing and Verification

Verification is split between hardware-in-the-loop validation and host-based unit tests. The `Test/` directory now ships with a self-contained Unity/CMock harness that demonstrates how to isolate hardware dependencies. Developers can clone the template and replace the mocked interfaces with the module under test.

6.1 Running the CMock Template

Within `ACC_Firmware/Test/cmock_example` a `Makefile` orchestrates compilation of Unity, the simplified CMock runtime, the generated mock, and the module under test. The example test exercises a fan control hysteresis helper:

```
void test_FanControl_Hysteresis(void) {
    const int sequence[] = {60, 54, 48};
    TemperatureSensor_Read_ExpectAndReturnSequence(sequence, 3);

    TEST_ASSERT_TRUE(FanControl_ShouldEnableCooling());
    TEST_ASSERT_TRUE(FanControl_ShouldEnableCooling());
    TEST_ASSERT_FALSE(FanControl_ShouldEnableCooling());
}
```

Executing `make run` builds the harness and runs the unit tests directly on the developer workstation, removing the need for target hardware during logic development. The mock registers itself with the lightweight CMock runtime to ensure that call expectations are verified automatically during `tearDown()`.

7 CAN Messaging Map

Table 1 summarises the CAN payloads emitted by the ACC controller. Each entry should be kept in sync with the DBC files maintained by the vehicle controls team.

Message ID	Periodicity	Payload	Notes
...	20 ms	Fan duty cycle, fan RPM	Diagnostic flags in byte 7
...	50 ms	Segment temperatures (4x int16)	Values in 0.1°C
...	100 ms	Pressure readings (4x int16)	Values in 0.1 kPa
...	Event	Fault/event code	Published on state transitions

Table 1: Accumulator cooling CAN messages.

8 Change Control

When introducing new peripherals or modifying the control algorithms, please make sure to update the relevant sections and increment the revision metadata at the front of the document. Including diagrams of state machines or timing waveforms greatly assists the test and integration teams in validating behaviour on the vehicle.