Queensland University of Technology

EGH456 - Embedded Systems

Group 17

Embedded Systems Design for an Electric Vehicle

Assessment 2

Ivan Keh - n11210656
James Prince - n10421262
Jason Tieu - n10972030
Samuel Smith - n11064196

Image generated by ChatGPT with a Photo of the provided Kit [1]

Contents

List of Figures

List of Tables

1 Introduction

Electric vehicles (EVs) are becoming increasingly common as the world seeks cleaner and more efficient alternatives to traditional combustion engines. This shift introduces significant engineering challenges in the design of embedded systems that can safely and efficiently control electric motors under real-time constraints. Unlike internal combustion engines, electric motors require fine-grained, rapid control of torque and speed, and the systems supporting them must be capable of immediately responding to changes in load, velocity, and external conditions.

This project focuses on the development of a real-time embedded system for controlling a three-phase Brushless DC (BLDC) motor and providing live feedback to the user via a touchscreen graphical interface. In automotive applications, such systems must operate under strict timing constraints, where delayed responses to critical events, such as obstacle detection or electrical faults, can lead to unsafe behaviour or physical damage. Ensuring safe and reliable vehicle operation therefore requires deterministic system behaviour, where tasks are executed predictably and safety mechanisms respond within guaranteed time bounds. This places significant demands on embedded software design, particularly in coordinating concurrent processes such as motor control, sensor acquisition, and fault handling under real-time constraints.

The embedded platform selected for this project is the Tiva TM4C1294NCPDT microcontroller, which supports high-performance real-time applications with its ARM Cortex-M4F core and extensive peripheral interfaces. The system architecture is built around FreeRTOS, enabling pre-emptive multitasking for concurrent execution of motor control, sensor acquisition, safety monitoring, and user interface tasks. A modular software structure is employed to ensure maintainability, while UART telemetry and SerialPlot are used to visualise system performance for debugging and validation.

The embedded software must meet several key requirements:

- Real-Time Performance**:** Ensures timely execution of motor control and safety routines under concurrent task loads.
- Motor Control and Monitoring**:** Provides smooth start-up, regulated speed control, and braking using PWM, with real-time feedback on motor RPM and power consumption.
- Safety and Fault Handling**:** Detects over-current, initiating emergency stop procedures and system protection.
- Sensor Integration**:** Acquires and filters signals from current, speed, and one optional environmental sensor, ensuring reliable data even in the presence of noise.
- User Interface Integration**:** Delivers a responsive graphical interface for motor control, parameter configuration, and system status visualisation, including real-time plotting of filtered data.

The final embedded system integrates these features into a cohesive and safety-aware control platform suitable for modern electric vehicle applications. The successful development of this system required coordinated efforts across multiple areas, including motor control, sensor integration, fault handling, user interface design, and system-level integration. Each team member was responsible for specific subsystems and contributed to the overall functionality, robustness, and performance of the final solution.

James Prince developed the motor control subsystem, including PWM regulation, RPM feedback, and UART telemetry. Samuel Smith handled sensor acquisition and filtering for speed and current data. Ivan Keh implemented fault detection and emergency stop logic. Jason Tieu designed the graphical user interface, integrating controls, status indicators, and live data plots. System integration and testing were carried out collaboratively.

2 Design and Implementation

The system was developed using a task-based architecture built on FreeRTOS, allowing the motor control, sensor processing, and user interface components to operate concurrently. Each subsystem was implemented as an independent thread with clearly defined responsibilities, scheduled according to priority and timing requirements. Inter-task communication was managed using FreeRTOS message queues for data sharing and semaphores for mutual exclusion when accessing shared resources. This structure ensured consistent and predictable system behaviour during runtime, particularly under load.

To support development and maintainability, comprehensive code documentation was generated using Doxygen. This automated documentation system provides detailed descriptions of all functions, and data structures, creating extensive reference materials for the codebase. The complete documentation is accessible online at https://james712346.github.io/EGH456Project/, facilitating code review, debugging, and future development efforts.

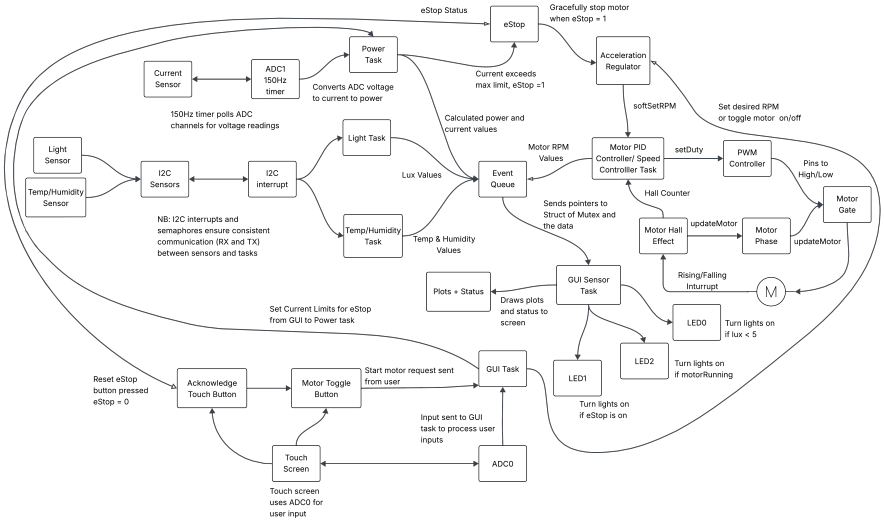


Figure 1: System architecture (Found in Appendix B)

Figure 1 presents an overview of the RTOS task structure and system architecture, highlighting the relationships between tasks and the flow of control and data throughout the system. The following sections detail the implementation and design considerations of each major subsystem, including the motor control logic, digital signal processing for sensors, and graphical user interface integration.

2.1 Shared Resources

Our team implemented a thread-safe data sharing mechanism to enable communication between FreeRTOS tasks without blocking operations. This solution uses a two-tiered structure design combined with mutex protection to ensure data integrity in a concurrent environment.

2.1.1 Data Structure Design

The implementation employs two complementary structures as shown in Figure 2:

1. **struct val** - A simple data container with two fields: `filtered` for processed sensor data and `raw` for unprocessed sensor readings.
2. **struct sharedValues** - A wrapper structure that encapsulates a `val` struct containing the actual data and a mutex handle for thread-safe access control.

This hierarchical approach separates data storage from synchronisation concerns, promoting clean code organisation and maintainability.

```
9  typedef struct {
10     double filtered;
11     volatile double raw;
12 } val;
13
14 typedef struct {
15     SemaphoreHandle_t mutex;
16     val values;
17 } sharedValues;
```

Figure 2: Shared data structures from "/include/shared.h"

2.1.2 Inter-Task Communication

For GUI updates, the system uses a FreeRTOS queue that passes pointers to initialised shared variables. This design allows the GUI task to identify specific shared resources by comparing pointers and execute the appropriate access functions without copying large data structures, improving both performance and memory efficiency.

2.1.3 Access Functions

The shared resource system provides three main access functions, each designed for specific use cases. Figure 3 demonstrates the implementation of the getter function:

```
35 uint8_t getter(sharedValues *dataPoint, val* buff, TickType_t
       blockingTime){
36     if (xSemaphoreTake(dataPoint->mutex, blockingTime) != pdPASS){
37         return -1;
38     }
39     *buff = dataPoint->values;
40     xSemaphoreGive(dataPoint->mutex);
41     return 0;
42 }
```

Figure 3: Example getter function implementation from "/src/shared.c"

Getter Function Retrieves data from shared resources safely. It accepts a pointer to the target `sharedValues` struct, a maximum delay for mutex acquisition, and a pointer to the buffer where retrieved data will be stored. The function safely copies current data values to the provided buffer.

Setter Function Replaces entire data content in shared resources atomically. It takes a pointer to the target `sharedValues` struct, a maximum delay for mutex acquisition, and a `val` struct containing new data to replace existing values. This function updates both filtered and raw values simultaneously.

SetterVal Function Updates individual data fields selectively for fine-grained control. It accepts a pointer to the target `sharedValues` struct, a maximum delay for mutex acquisition, a double value to be assigned, and a boolean flag (1 = update filtered value, 0 = update raw value). This function modifies only the specified field while preserving the other.

2.1.4 Thread Safety and Performance

All access functions implement mutex-based synchronisation with configurable timeout periods. This approach ensures data integrity by preventing race conditions during concurrent access, while timeout mechanisms prevent indefinite blocking and potential deadlocks. The implementation ensures that tasks do not hold onto mutexes for extended periods by requiring tasks to provide pre-allocated buffers for data retrieval or pre-computed values for data updates. This design minimises the critical

section duration, as tasks perform expensive computations outside the mutex-protected region and only briefly acquire the lock for fast memory copy operations. The maximum delay parameter allows each task to specify its tolerance for waiting, enabling priority-based resource access and preventing high-priority tasks from being indefinitely blocked by resource contention. This efficient design maintains system responsiveness while supporting multiple tasks safely accessing shared resources, crucial for the real-time performance requirements of the electric vehicle control system. [2]

2.2 Motor

The motor control subsystem represents one of the most critical components in the overall system architecture, as it serves as the primary output actuator capable of directly impacting user safety in a fully implemented electric vehicle. The potential for serious harm or fatality necessitates that all motor operations be designed with comprehensive safety protocols and fail-safe mechanisms as the primary consideration.
This section covers the complete motor control implementation, including phase switching, RPM calculations, speed control, and safety requirements.

2.2.1 Design and Task Approach

The Motor Subsystem was designed using two FreeRTOS tasks: one for motor control and another for RPM calculation. It incorporated three GPIO interrupt handlers, each responsible for monitoring a specific input and updating one of three bits in a $uint8\_t$ status variable based on the input's state. The motor control task utilised Timer 1A [3] on the micro controller to incrementally adjust the motor speed toward the user-defined target RPM, ensuring a safe and controlled acceleration. Additionally, emergency stop detection and handling were implemented within the Timer 1A interrupt, allowing immediate response to stop conditions without needing to wait for a mutex to be taken/given.

2.2.2 Phase Timings

The phase timings are determined using Hall effect sensors embedded in the motor, which detect the rotor's position relative to the three stator phases. Interrupts are configured on both the rising and falling edges of the H2, M3, and N2 pins [4]. When triggered, these interrupts release a semaphore that activates the motor control FreeRTOS task. This task is responsible for switching the motor phases based on the currently active Hall effect sensor state.

The Hall sensors correspond to the motor phases as follows: H2 corresponds to Phase B, N2 to Phase C, and M3 to Phase A. Phase updates are handled through the provided motor API code [4, 5]. Special consideration is given to edge cases where two Hall effect sensors are active simultaneously; these conditions are detected and correctly resolved within the custom motor control logic.

Figure 4 shows the output of the Hall effect sensors during manual rotation of the motor, demonstrating the sequence of sensor activations and their role in determining rotor position.

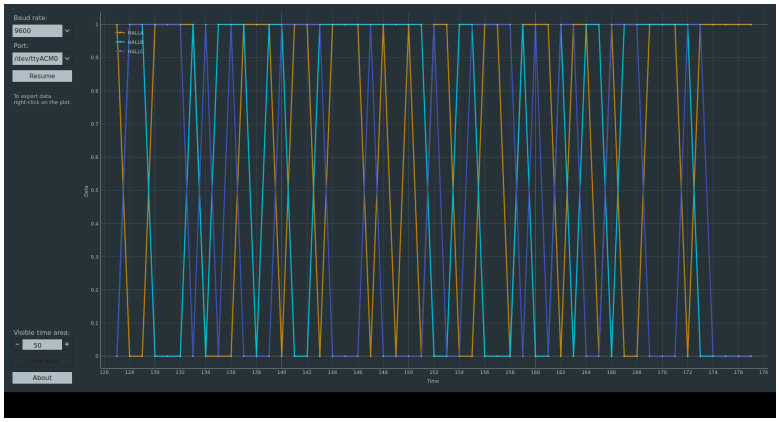


Figure 4: Hall effect sensor outputs (H2 - HALLB, M3 - HALLA, N2 - HALLC) during manual rotation of the motor.

2.2.3 Speed Calculation

Speed calculation is performed within the speed controller task, which runs at a fixed interval. Each time the task executes, it calculates the motor speed based on the number of Hall effect rising edges that occurred since the last execution. These edges are counted by an interrupt service routine, which increments a counter on each rising edge.

The time difference dt is calculated using `xTaskGetTickCount()`, which returns the number of ticks since the system started. The speed controller task stores the tick count from its previous execution and subtracts it from the current tick count to determine dt, the elapsed time in milliseconds.

Given that there are 12 Hall effect triggers per revolution[6], the rotational speed in RPM is calculated using the following equation:

$$\text{RPM} = 60 \times \frac{a}{12} \times \frac{1000}{dt} \tag{1}$$

Where:

- a is the number of Hall effect rising edges counted since the last task execution,
- dt is the time difference between the current and previous executions of the speed controller task, in milliseconds.

2.2.4 PID Controlller

The PID (Proportional–Integral–Derivative) controller is a widely used feedback control mechanism designed to ensure that a system's output follows a desired reference or set point. It achieves this by continuously calculating an error value as the difference between the desired set point and the actual process variable, and applying corrective actions based on three gains [7]:

- **Proportional (K_p):** Produces an output that is directly proportional to the current error. This helps drive the system toward the set point and provides an immediate correction.
- **Integral (K_i):** Accounts for the accumulation of past errors over time. It helps eliminate steady-state error by integrating the error, ensuring the output eventually reaches the set point.
- **Derivative (K_d):** Predicts the future trend of the error by considering its rate of change. It helps dampen the response and reduce overshoot.

Through motor tuning, suitable values for the proportional and integral gains were selected, while a small derivative gain was also introduced to improve stability.

The PID controller was implemented within the speed controller task and executed after each speed calculation. This placement allows the system to respond quickly to any sudden changes in load or target speed. To avoid instability during startup or low-speed operation, the PID calculation was conditionally skipped when the motor was off or operating below 100 RPM. In these cases, a kick-start mechanism was used to help the motor begin spinning before engaging the PID controller, preventing erratic control behaviour.

2.2.5 Safety Acceleration Considerations

Acceleration is managed using Timer 1A on the micro controller, which is configured to run at 50 Hz. On system startup, a step size is calculated by dividing the maximum allowed RPM acceleration by the timer frequency. For this project, the requirement is a maximum acceleration of 500 RPM/s, resulting in step increments of ± 10 RPM per timer interrupt.

These step changes are applied to a variable referred to as the *soft RPM*, which incrementally approaches the user-defined target RPM. The soft RPM is then passed to the PID controller. This approach ensures smooth and controlled acceleration and deceleration, preventing sudden changes in speed that could hurt a passenger.

Figure 5 illustrates the behaviour of the acceleration logic. The plot shows the measured RPM and the soft RPM over time. The staircase-like pattern of the soft RPM highlights the discrete step increments applied at each timer interrupt. The measured RPM follows this curve as the PID controller adjusts the motor output accordingly.

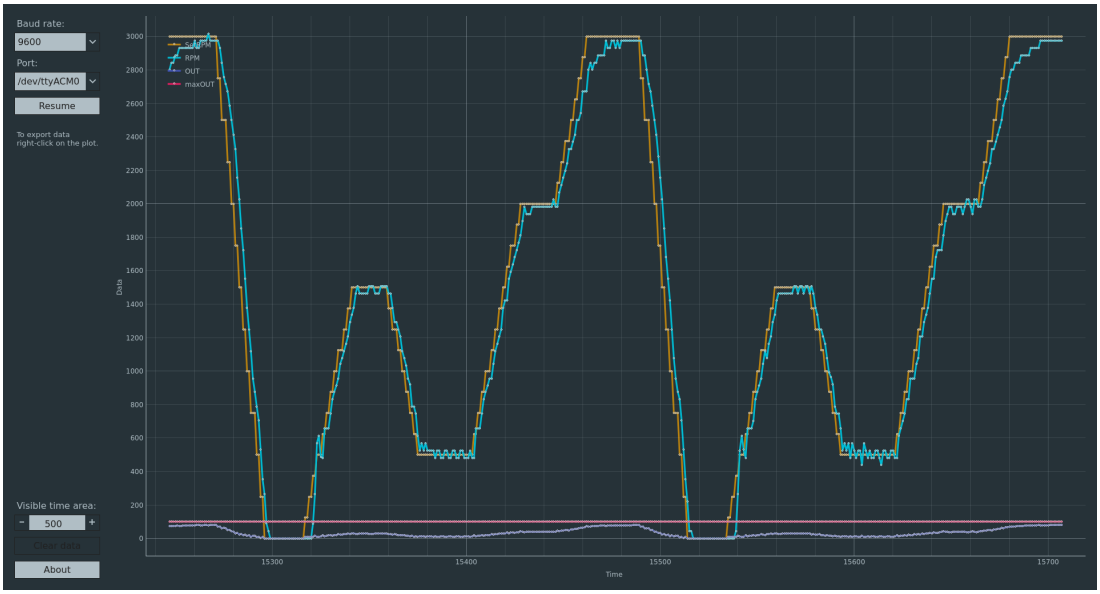


Figure 5: Measured RPM vs. Soft RPM over time

2.2.6 Emergency Stop

The Emergency Stop (e-stop) is a system requirement that demands the motor decelerate to a complete stop at a rate of 1000 RPM/s. Leveraging the same logic used for normal acceleration, the system switches to a more aggressive step size when an e-stop condition is detected. Specifically, within the Timer 1A interrupt, the step value is adjusted to match the required e-stop deceleration, and the target RPM is immediately set to 0.

Because this logic is handled inside the timer interrupt, it avoids the need for mutexes or coordination with other tasks. The code is kept minimal to prevent blocking the rest of the system. As shown in Figure 5 around the 15,500 time mark, the system exhibits a rapid decrease in RPM, with larger downward steps to zero that is clearly contrasting the gradual transitions seen during normal operation.

2.2.7 Interfacing with Additional Tasks

To interface with other tasks in the system, we publish current RPM, set RPM, and output using the shared value structures described earlier. These values are pushed to a FreeRTOS queue whenever they are updated.

For the emergency stop (e-stop), we implemented a dedicated setter and getter function that is shared between the GUI and sensor tasks. This allows tasks to easily and safely trigger or detect whether an e-stop condition has occurred.

2.3 Sensors

Multiple sensors have been implemented in the design to monitor various conditions both inside and outside the system. Temperature and humidity sensors measure the temperature in degrees Celsius and the humidity in Relative Humidity, while a light sensor measures the ambient light levels in lux. The system measures the voltage of two of the three voltage sensors, converts it to a current reading, and calculates the power drawn from the motor. Speed sensing was done using the Hall Effect sensors connected to GPIO pins. This section will summarise the integration of sensors into the system.

2.3.1 Design and Task Approach

The sensor subsystem was designed to collect sensor data using two main sources, ADC and I2C. Light [8], temperature, and humidity were collected from the sensors that interface with I2C, while the speed and current sensors used ADC to gather data necessary for their sensor outputs. The hardware connections for I2C needed to be confirmed to ensure that the correct lines were setup for I2C sensor communication [9]. Once that had been established, testing of the initial communications to each sensor was the nest step towards this task. The light and climate sensors required initialisation before sensor data could be collected. By ensuring that initialisation completed correctly, could the data gathered from the sensors be analysed. Testing of light values was done by flashing a light at the sensor and watching the values change over UART, while temperature and humidity was similarly tested by warming the sensor and breathing on it to see sensor values change over UART.

For the ADC collection, the hardware connections had to be established to ensure that the proper pins were configured for ADC. While hardware connection diagrams were provided showing what pins were connected to the voltage lines, an ADC scan task function was created to poll all of the ADC lines to ensure readings were suitable from a voltage sensor. The datasheet explains that the voltage is bidirectional, so a zero voltage reading would be halfway between the 4095 max ADC voltage reading. Once those lines had been identified using the ADC scan task and the hardware connection diagrams, the formula provided in the motor data sheet [5] was used to convert the voltage readings to current. By multiplying the 24V motor voltage by the current, and multiplying those results by 1.3 (to account for only having 2/3 of the voltage reading), a power measurement was possible.

The speed sensing was achieved by using the three hall effect sensors connected to GPIO pins M, H, and N. These sensors generated interrupts on both the rising and falling edges, with each full revolution of the motor producing 12 transitions. An interrupt service routine (ISR) incremented a transition counter, which was sampled by the speed controller task to calculate RPM. The PID controller was able to use this feedback for precise motor feedback.

2.3.2 Sensor Hardware Overview

The sensor subsystem incorporates multiple devices to monitor environmental conditions, motor behaviour and safety parameters. Table 1 provides a summary of the sensors implemented into the system, their types, interfaces and respective purposes with the system.

Table 1: Sensor Hardware Overview

Sensor	**Type**	**Protocol**	**Purpose**
OPT3001	Ambient Light Sensor	I2C	Detect lighting conditions for day/night
SHT31	Temperature & Humidity	I2C	Environmental monitoring
ADC Channels 0 & 4	Current Sensing (2 Phases)	ADC Interrupts	Monitor motor current using 7mΩ shunts (dual phase, 1.3x correction factor)
Hall Effect Sensors A, B, C	Speed Sensing	GPIO Interrupts	Measure motor RPM using Hall edge counting (12 transitions per revolution)

The sensors that have been selected allow comprehensive monitoring of the environment and motor parameters while balancing system complexity and real-time constraints within the system. The following sections will cover the details surrounding the integration of sensors into the system.

2.3.3 Connection Overview

The OPT3001 ambient light sensor and the SHT31 temperature and humidity sensor [10] shared the I2C bus, each with their own unique device addresses. The micro controller polls each sensor via their own dedicated FreeRTOS sensor task. The ADC-based current sensing and hall effect speed sensing operate independently of the I2C bus. MCU interface connections can be seen in Table 2

Table 2: Connection Overview

Sensor	**Protocol**	**MCU Interface**	**Notes**
OPT3001	I2C	I2C0	Ambient Light
SHT31	I2C	I2C0	Temperature and Humidity
ADC Channels 0 & 4	ADC	ADC1 Ch0, Ch4	Current monitoring
Hall Effect Sensors A, B, C	GPIO	GPIO Ports M, H, N	Speed sensing

To identify the connections required to communicate with the sensors over I2C the booster pack pinout diagram was analysed as seen in Figure 6 showing the pins connected to the data and clock lines [9] [11] [3].

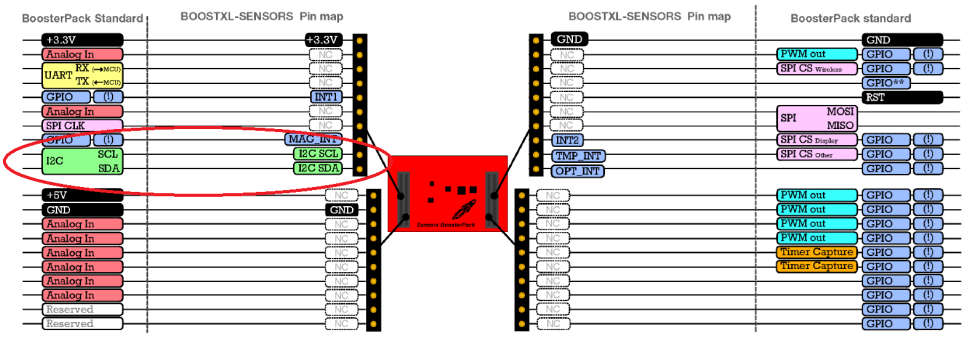


Figure 6: I2C Bus Connection Diagram showing I2C data and clock lines circled in red from BOOSTXL-SENSORS Sensors BoosterPack Plug-in Module datasheet

To properly configure the hall effect sensors and the voltage lines for the motor, the pin-out diagram for the booster pack was analysed, as seen in Figure 7, showing voltage sensors on channels 0 and 4 and Hall effect sensors tied to GPIO ports M, H and N [4].

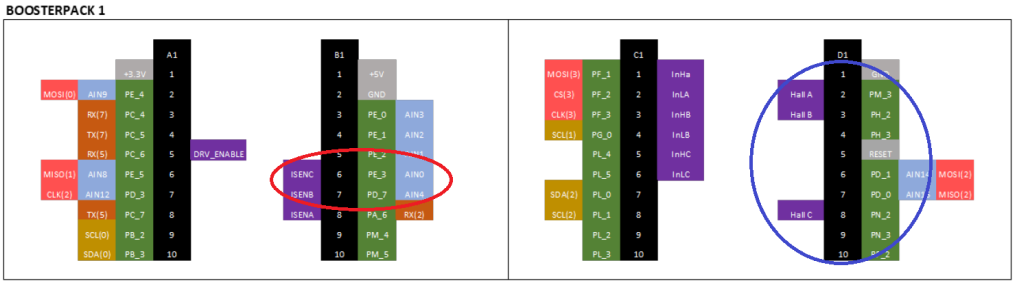


Figure 7: Booster Pack 1 diagram showing motor voltage lines and ports circled in red and Hall effect sensor and ports circled in blue. Sourced from unit support material.

2.3.4 FreeRTOS Sensor Task Design

Each sensor was split up into their own subsequent FreeRTOS tasks [2] to ensure separation of functionality and allow setting of individual sensor timing requirements. Task call frequencies and their scheduling can be seen in Table 3.

Table 3: FreeRTOS Task Overview

Task	Function	Polling Interval	Priority
Light Sensor Task	Polls OPT3001 via I2C	500 ms	2
Temp/Humidity Task	Polls SHT31 via I2C	1000 ms	3
Power Task	Reads ADC Ch0/Ch4, computes current	100 ms	8
Speed Controller Task	Calculates RPM and PID control	100 ms	9

Each task was assigned their RTOS priority based on their call frequency, sensor polling requirements and their importance to system functionality. Light and climate sensors were given lower task priorities compared to the Power and Speed controller tasks due to their critical functions of handling motor functionality and monitoring current limits.

To ensure that I2C read and write communication was consistent, a semaphore was used to ensure that timing requirements were met throughout I2C transmissions.

2.3.5 Sensor API Layer

To assist with the interfacing of sensors, different libraries were used for I2C communication. I2C libraries provided from Texas Instruments (TI) assisted with interfacing with the opt3001 light sensor, with extra I2C drivers from Jesse Haviland [12] were incorporated to assist with the TI libraries. The I2C drivers had to be modified by incorporating semaphores into the read and write functions to assist with timing requirements from the sensor to the sensor task. An I2C driver published by artfulbytes [13] was incorporated and modified to fit the I2C requirements for the SHT31 sensor.

While I2C communication was protected using semaphores, the I2C communication can still fail in reads or writes. In the event of time outs occurring between reads and writes, the system simply retries at the next I2C polling interval. The I2C communications are designed to gracefully degrade in the event of I2C failures, and preserve system operability.

2.3.6 Data Processing, Calibration and Filtering

To filter out the noise from sensor acquisitions, Exponential Moving Average (EMA) filters were incorporated into each sensor. The formula for the EMA can be found here [14]. The formula for the EMA is

$$y[n] = \alpha \cdot x[n] + (1 - \alpha) \cdot y[n - 1] \tag{2}$$

where:

- $y[n]$ is the current filtered value
- $x[n]$ is the current raw sensor reading
- α is the smoothing factor (between 0 and 1)
- $y[n - 1]$ is the previous filtered value

By utilising this filter after sensor data has been acquired, noise can be filtered from sensor samples. This technique was applied to the light sensor, climate sensor, and current sensor. Additionally, the current data was averaged before being filtered again using EMA, ensuring smooth current readings.

For RPM measurements, while no smoothing filters were applied directly, an event based threshold was implemented so that RPM updates were only published if there was a measured change that exceeded 25 RPM. This reduced any unnecessary traffic sent out from the sensors caused by any minor fluctuations.

2.3.7 E-Stop Conditions

To protect the systems motors and users, an emergency stop condition was introduced. The user can define the max current threshold for which the system will start to gracefully shut down the motor. To determine the upper limits for the max current, the motors power supply output current was used as the upper limit. The power supply limit is 1.25A, the current sensor only measures two thirds of the current available, therefore the upper limit that could be set for the motor was 840A.

The user can lower these current thresholds using the GUI to as low as desired. The power task constantly checks the measured current against the max current limit threshold. If the current exceeds this threshold the eStop condition is set. Timer 1A which runs every 250 ms checks for the eStop condition and gracefully shuts down the motor if eStop = 1.

2.3.8 Testing and Validation: Unit Tests

Once communication was established with the sensors via I2C, UART print messages were used to print the values that were being received for each sensor. As seen in Figure 8, the raw data from I2C

was evaluated to ensure that I2C communication was consistent with the converted temperature and humidity readings.

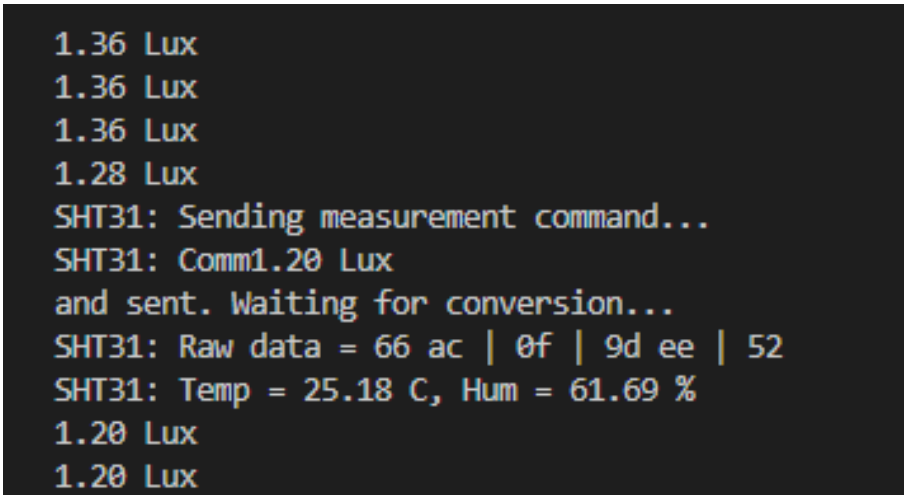


Figure 8: UART print statements from OPT3001 and SHT31 sensor

To validate the current and power readings from the voltage sensors, UART prints were also utilised. Figure 9 shows the ADC values on both ADC channel zero and four, whilst also showing the calculated power. It is to be noted that the power values seen in Figure 9 were from unit testing, are incorrect and are not representative of the final state of the system.

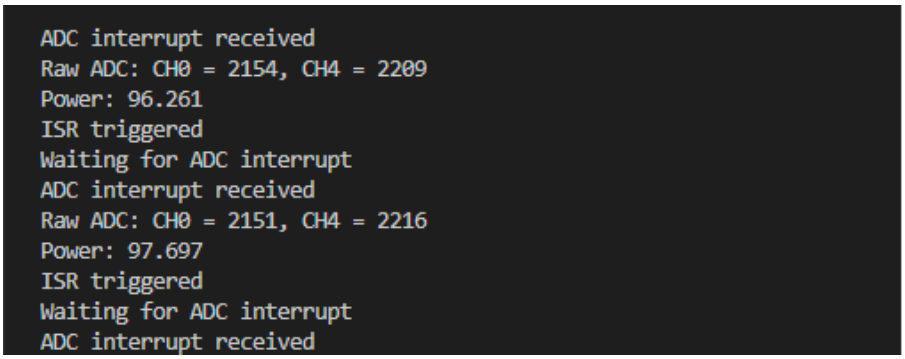
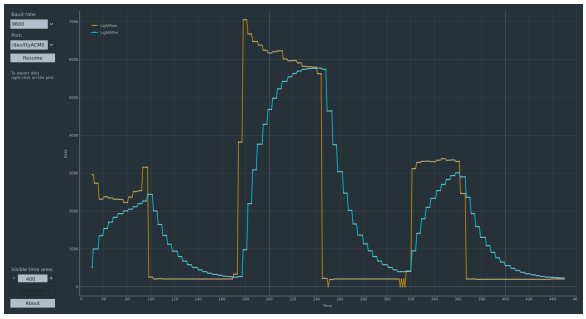


Figure 9: UART print statements showing raw voltage readings (ADC reading between 0 and 4095) and converted power values (these are initial testing values and the power readings are incorrect)

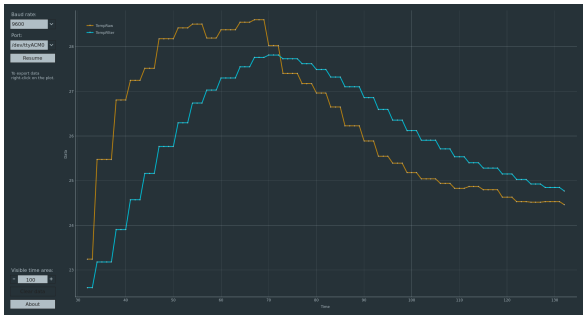
2.3.9 Testing and Validation: Integrated Tests

Once the key subsystems were integrated, the raw and filtered outputs of the sensors were plotted using Tauno Serial Plot[15]. Figure 10 shows that for each different sensor, the EMA filter improves the sensor output versus the raw values with the orange lines representing raw values and the blue lines representing filtered values.

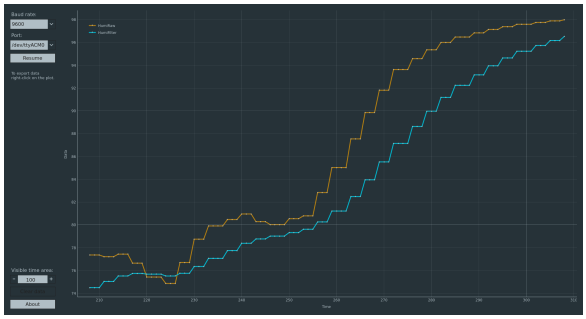
(a) Raw current values and filtered power values.



(b) Raw light values plotted with filtered light values.



(c) Raw temperature values plotted with filtered temperature values.



(d) Raw humidity values plotted with filtered humidity values.

Figure 10: Integrated system test runs showing sensor responses under various scenarios.

2.4 Graphical User Interface (GUI)

The graphical user interface (GUI) was designed to provide an intuitive and responsive means of interaction between the user and the embedded system. Its primary functions include monitoring system status, visualising sensor data in real time, adjusting motor parameters, configuring safety thresholds, providing emergency-stop (e-stop) acknowledgement, and displaying sensor plots. Built using the TivaWare Graphics Library (grlib) [16] and widget toolkit, the GUI operates as a dedicated FreeRTOS task and is fully integrated into the task-based system architecture.

2.4.1 Design and Task Approach

The GUI operates on a layered architecture, separating rendering logic from data processing. It runs at 30 Hz with periodic task delays (vTaskDelay) [2] to balance responsiveness and CPU usage. All rendering is offloaded to a dedicated GUI task, while input data is received asynchronously from other subsystems using FreeRTOS queues and synchronised shared memory (getter() and setter() functions). A separate guiSensorTask filters sensor values and updates canvas widgets for live system feedback.

Touch inputs are handled via the TouchScreenCallbackSet() [16] mechanism and mapped into widget message queues using WidgetPointerMessage [16], allowing seamless UI interaction without directly polling from the ISR.

2.4.2 Screens, Navigation, and Interface Layout

The interface comprises of five main screens: Home, Motor, Status, Plots, and Settings. Each are defined in its own canvas widget hierarchy. Navigation is handled through push buttons with callback functions, which dynamically build the desired screen and assigns it to the root widget. The screen management logic is centralised within a dedicated set of functions responsible for constructing and switching between interface screens, enabling modular and scalable navigation throughout the application. Each screen was designed to balance usability with the limitations of the embedded system:

- Home Screen: Provides a 2x2 button layout for navigating to each functional screen.

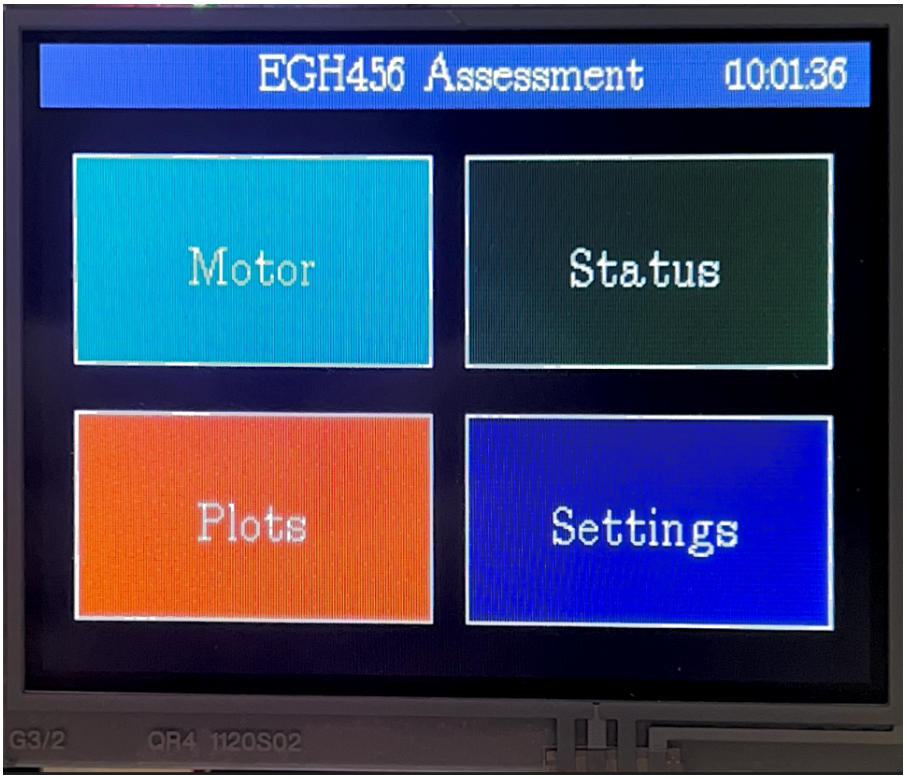


Figure 11: Home Screen Snippet

- Motor Screen: Includes a toggle button for motor on/off control, a live motor status label, and an RPM slider. The slider updates a global RPM setpoint, used by the control task.

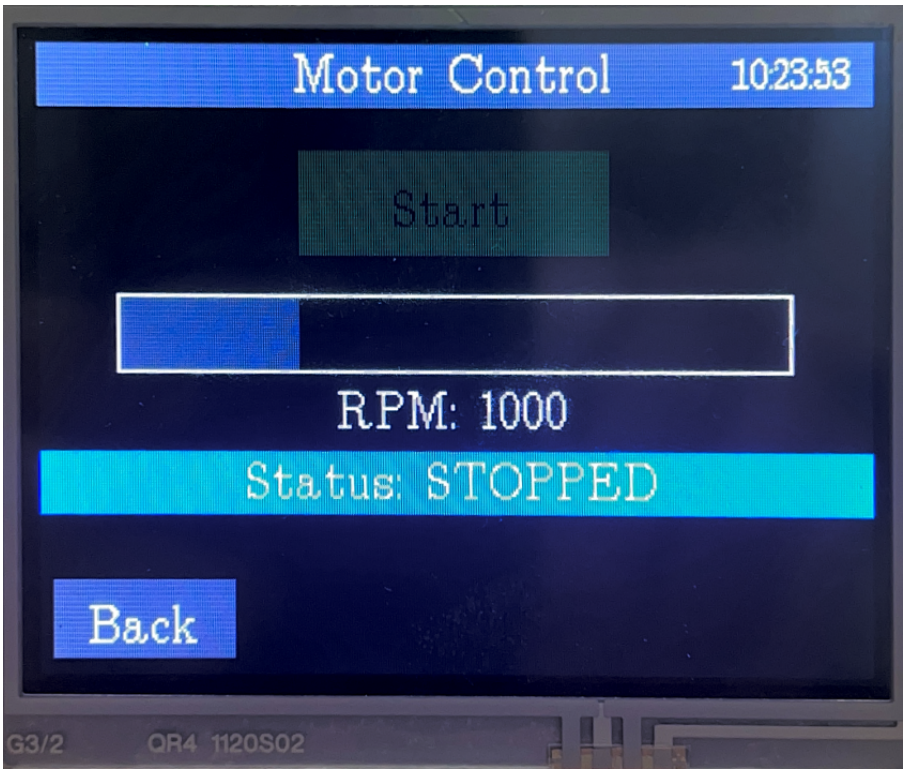


Figure 12: Motor Screen

- Status Screen**:** Displays current system metrics (e.g. actual RPM, system mode, power usage),

motor state indicators, and safety status.

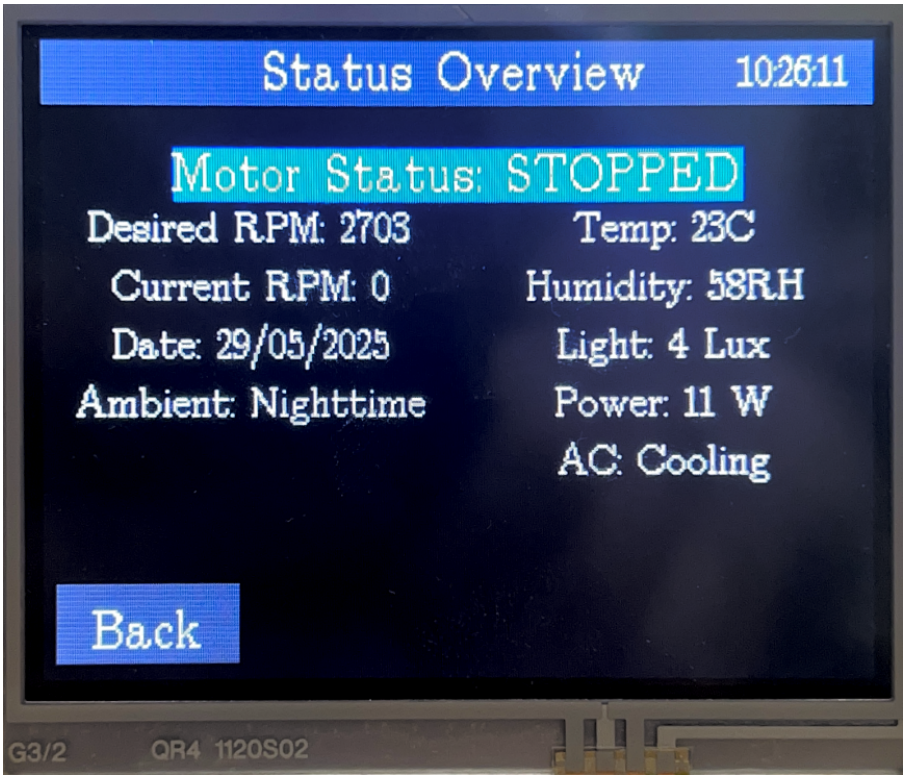


Figure 13: Status Screen

- Plot Screen**:** Graphs filtered sensor data over time, allowing users to switch between RPM, power, light level, and temperature.

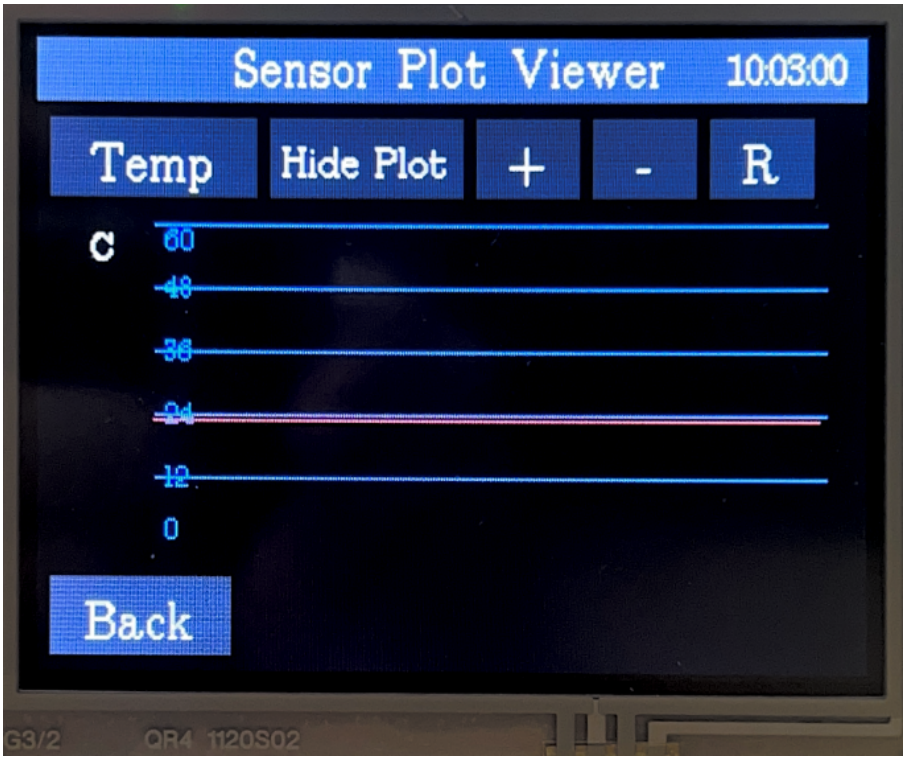


Figure 14: Plot Screen showcasing Temperature

- Settings Screen: Allows configuration of upper/lower thresholds for acceleration, temperature, and current via sliders with live numeric feedback.

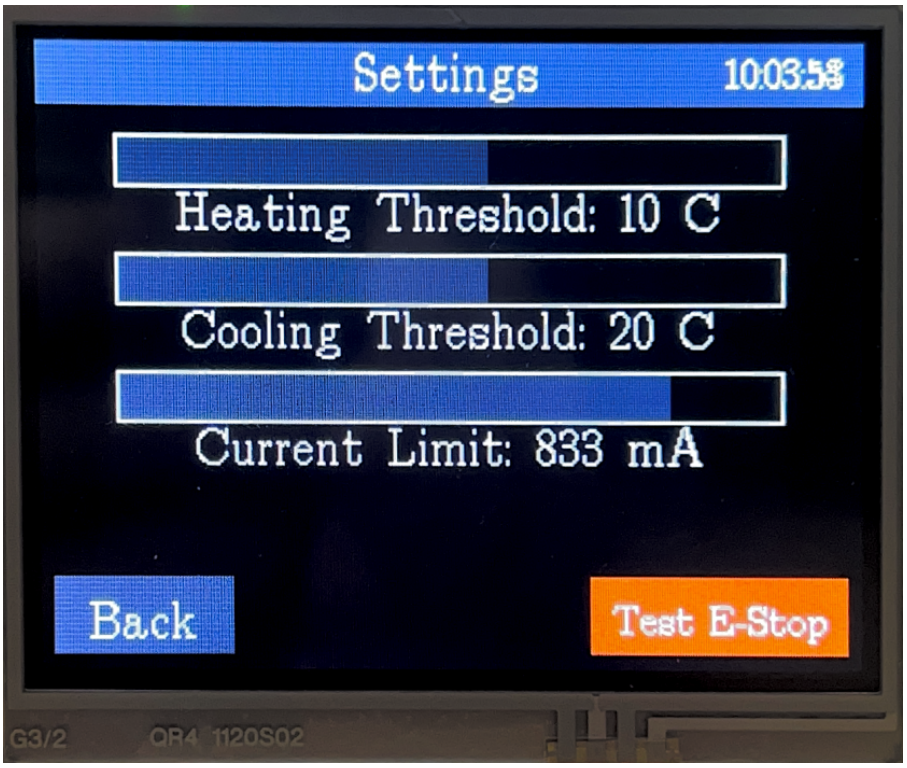


Figure 15: Settings Screen

Slider inputs update global configuration values via callbacks and are styled for clarity using consistent visual formatting. These inputs directly affect the control and safety subsystems at runtime.

2.4.3 Safety Integration

When the control system detects a fault (e.g., over current), it sets a global `eStopFlag`. The GUI reacts by disabling motor controls, changing the toggle button to display **"Acknowledge"** and updating the screen's visual indicators. The user must manually acknowledge the e-stop via the toggle button; this resets the flag and returns the system to a safe, idle state. This ensures that recovery from fault states is deliberate and traceable.

Figure 16: Motor Acknowledge Screen

2.4.4 Real-time Clock Implementation

The GUI displays a real-time clock in the upper-right corner of the Home screen. As the hardware lacks a dedicated real-time clock (RTC), the time is initialised manually to the demonstration day and incremented using a FreeRTOS software timer [2] that triggers once per second. Time is stored in a global structure and rendered using `GrStringDraw()` in the `guiTask`.

2.4.5 Day/Night Detection

Ambient light is monitored continuously using a light sensor. A value below 5 lux is classified as nighttime. When this condition is detected, the GUI displays a "Night Mode" label and activates a simulated headlight LED on the Status screen. This feature updates every second and could be extended to control physical headlights in a full system implementation.

2.4.6 Sensor-specific Features: Temperature and Humidity

The selected additional sensor was the temperature and humidity module (SHT31). The GUI displays real-time temperature (°C) and relative humidity (%RH) values on the *Status* screen, with live updates transmitted via a FreeRTOS queue from the sensor task. Users can configure an upper temperature threshold on the *Settings* screen. If the filtered temperature exceeds this threshold, the AC status message dynamically updates to indicate `"AC: Cooling"` or `"AC: Heating"` accordingly. While humidity is continuously monitored and displayed, it does not currently influence system behaviour and is reserved for future feature expansion.

2.4.7 GUI Sensor Plotting

The Plot screen renders filtered sensor data in real time using a circular buffer. A custom canvas widget calls GUI_DrawGraph() during PlotPanelPaint() to draw line plots for up to 100 points. Users can cycle through six sensor plots, zoom in/out, reset the Y-axis scale, and toggle visibility. The graphs use labelled axes, consistent colours, and dynamically updated units (e.g., RPM, Lux, °C),

supporting fast visual diagnostics.

2.4.8 Hardware and Software Integration

The GUI subsystem interfaces directly with several hardware components, including a 320x240 pixel LCD display and a capacitive touchscreen panel. The display is driven using a standard graphics controller, while touch input is processed through a dedicated touchscreen module that routes events to the graphical widget system. Analog sensor values, such as current or temperature, are acquired through onboard ADC channels and made available to the GUI for plotting and threshold monitoring. The GUI is built using standard modules from the TivaWare software suite [16], including the graphics library, widget toolkit, and low-level driver interfaces. Screen layouts are constructed using canvas, slider, and push button widgets, while serial output via the onboard UART is used to support runtime debugging and event logging throughout GUI operation.

2.4.9 Theoretical concepts and Engineering Choices

The GUI is structured around event-driven programming principles, with a separation between user interface (view), system state (model), and input handling (controller). FreeRTOS tasks follow a producer-consumer model, using queues for communication and avoiding shared memory race conditions through explicit synchronisation. The GUI's low priority ensures that it remains non-intrusive to motor and sensor threads. Modular design patterns (e.g., one function per screen builder) improve code maintainability and facilitate future expansions such as additional screens or new sensor inputs.

3 Results

3.1 Motor

The motor control subsystem demonstrated successful implementation of all key requirements, including real-time phase switching, accurate speed control, safety-compliant acceleration profiles, and reliable emergency stop functionality. Testing was conducted through a combination of hardware validation, UART telemetry logging, and SerialPlot [15] visualisation to verify system performance under various operating conditions.

3.1.1 Phase Switching Performance

Most importantly, the motor demonstrated smooth and quiet operation without whining, stalling, or excessive noise generation. This confirms that the phase switching logic correctly sequences the motor phases in synchronisation with rotor position, preventing the motor from fighting against its own magnetic field. The absence of audible noise or vibration indicates proper timing of phase commutation, which is critical for both motor efficiency and longevity.

3.1.2 PID Controller Performance

The PID controller demonstrated effective motor speed regulation under varying load conditions. Through systematic tuning, optimal gains were established that provided:

- Rapid response to set-point changes without excessive overshoot
- Steady-state accuracy with minimal error between target and actual RPM
- Stability under load variations and external disturbances
- Smooth operation without oscillation

The conditional PID activation (disabled below 100 RPM) successfully prevented instability during startup and low-speed operation. The kick-start mechanism reliably initiated motor rotation before PID engagement, eliminating startup difficulties that occurred in early testing phases. Figure 5 clearly illustrated the PID controller's ability to track the soft RPM reference, with measured RPM following the stepped acceleration profile as designed.

3.1.3 Safety-Compliant Acceleration

The acceleration limiting system successfully enforced the 500 RPM/s maximum acceleration requirement and the 1000 RPM/s Emergency stop deceleration. Timer 1A, operating at 50Hz, applied discrete 10/20 RPM increments to achieve the target acceleration/deceleration rate. Testing confirmed:

- Consistent step timing with 20ms intervals between RPM increments
- Smooth acceleration curves without abrupt speed changes
- Accurate acceleration rates measured at 500 ± 10 and 1000 ± 10 RPM/s

The soft RPM approach effectively prevented sudden motor speed changes that could compromise passenger safety or system stability. Visual confirmation through SerialPlot [15] showed the characteristic staircase pattern of incremental speed changes, validating the controlled acceleration implementation.

3.2 Sensors

This results section demonstrates the performance of the sensor subsystem during system integration and testing. The key outcomes of this section include the verification of sensor accuracy, correct emergency stop activation during over current, and confirmation of stable filtering of sensor outputs. Unit testing and integrated testing was performed to validate this subsystems functionality under real-time conditions.

3.2.1 Sensor Data Validation

Sensor subsystems were validated when integrating the subsystems. This was done by logging both the raw and filtered sensor values. Each sensor produced expected readings when stimulated. Light levels varied when flashing an external light onto the light sensor, temperature and humidity varied with heat and moisture inputs and current readings increased under high motor loads. By applying EMA filtering [14], this successfully reduced noise measurements across all sensors.

3.2.2 E-Stop Trigger Validation

The emergency stop was tested by inducing high current conditions. By loading the motor power supply over its current limit, the e-stop conditions were triggered when the calculated current exceeded the user defined max current limit. Upon e-stop activation, the system began gracefully shutting down the motor via Timer 1A, demonstrating the correct safety response.

3.2.3 Integrated Sensor Plots

Figure 10 show the representative sensor logs that were captured during the systems operation. Both raw and filtered values are plotted and show current, light, temperature, and humidity readings. These plots show the effectiveness of the EMA filtering approach and demonstrate stable sensor acquisitions for the system.

3.3 Graphical User Interface (GUI)

The final embedded system successfully fulfilled most functional requirements as specified in the project brief. Core features, including motor control, safety enforcement, sensor monitoring, and real-time GUI interaction, were tested on physical hardware and found to operate reliably under typical conditions.

Motor control functionality was verified through the GUI's start/stop button and RPM slider, which correctly translated user inputs into changes in motor behaviour. Motor speed was reflected in real time via both GUI labels and UART debug output, confirming the integrity of the feedback loop. Safety features were also validated: exceeding the configured temperature threshold reliably triggered an emergency stop, disabling controls and activating a red fault indicator. The system correctly enforced acknowledgement of fault conditions before allowing the motor to restart, as required.

The sensor plotting interface performed as expected, displaying smooth and timely graphs of motor speed, estimated power, temperature, and ambient light levels. Users were able to switch between plots, and each graph included labelled axes and colour-coded legends to aid interpretation. Additional features such as the real-time clock and day/night classification were also functional; the software-emulated clock maintained consistency during testing, and light sensor readings correctly toggled the simulated headlight LED under low-light conditions. Threshold adjustments made via sliders were reflected immediately in control logic, demonstrating effective inter-task communication and runtime flexibility.

Testing was performed using a combination of interactive GUI checks and UART-based telemetry. Sensor stimuli were simulated manually, for instance, covering the light sensor or applying heat to the temperature sensor, to observe corresponding GUI updates and safety actions. Task behaviour and system state transitions were monitored through log messages and visual feedback, allowing real-time validation of functionality across all modules.

Despite these successes, several limitations were identified. Although the humidity sensor was integrated and displayed in the GUI, it was not used in any control logic, reducing its practical contribution. This highlighted the importance of establishing clear functional roles for each sensor during the design phase. Additionally, the plot screen lacked features such as zooming or dynamic scaling, which would have improved data readability in low-variance scenarios. Some code in the GUI module, particularly in `gui.c`, became tightly coupled to global variables, making it more difficult to refactor or extend. A stronger emphasis on modular encapsulation and abstraction could have improved maintainability.

These challenges offered valuable lessons in the design of real-time embedded systems, particularly in the balance between responsiveness, usability, and maintainability within resource-constrained environments. Future iterations would benefit from more rigorous interface planning and extended functional roles for all integrated components. Ultimately, the project has demonstrated how embedded systems, when carefully designed and tested, can meet the demanding requirements of electric vehicle applications, therefore providing safe, reliable, and responsive control that is essential for modern transport technologies.

4 Conclusion

This project successfully delivered a real-time embedded system for controlling an electric vehicle. The system was developed on a Tiva TM4C1294NCPDT microcontroller using FreeRTOS, integrating motor control, sensor acquisition, safety mechanisms, and a graphical user interface into a cohesive and responsive platform.

The motor control subsystem met key functional and safety requirements. Though phase switching and feedback-driven speed regulation using a PID controller, the motor responded reliably to changes in user input and load. Acceleration and emergency stop profiles were enforced using timer-based logic, ensuring smooth ramp-up and rapid shutdown without compromising system stability. Phase transitions occurred cleanly, and RPM tracking closely followed the target reference values across a range of operating conditions.

Sensor data was acquired through both I2C and ADC interfaces, covering environmental parameters and internal system metrics. Filtering techniques, such as exponential moving averages, significantly improved signal quality, while over-current protection was implemented using real-time power monitoring and an emergency stop trigger. These features were validated through targeted testing and confirmed stable operation in the presence of external disturbances and system noise.

The graphical interface supported real-time interaction with the system, offering visual feedback for motor status, sensor values, and safety alerts. Users could adjust control parameters, monitor system performance, and respond to fault conditions through an intuitive touchscreen interface. Additional features such as day/night detection, live plots with zoom controls, and a software-based clock extended system usability while maintaining real-time responsiveness.

All system components were developed with a focus on modularity, timing accuracy, and fault tolerance. Subsystems communicated through synchronised shared resources and message queues, with mutual exclusion enforced through mutex protection, thus ensuring consistent behaviour under concurrent execution. Testing confirmed that the system met its design goals, with responsive performance and predictable task scheduling across a range of scenarios.

This project demonstrates a practical application of embedded systems engineering principles in a safety-critical context. It highlights the importance of real-time scheduling, reliable sensor integration, and user-focused design in the development of control systems for modern electric vehicles

References

[1] OpenAI, *Chatgpt*, https://chat.openai.com/chat, Large language model, 2025.

[2] FreeRTOS Project, *FreeRTOS Documentation*, API headers, source code documentation, 2025. [Online]. Available: https://www.freertos.org/Documentation/00-Overview.

[3] Texas Instruments, *Tiva™ TM4C1294NCPDT Microcontroller DATA SHEET*, Sections 10, 13, 18 and 25, 2014. [Online]. Available: https://www.ti.com/product/TM4C1294NCPDT.

[4] Queensland University of Technology, *Embedded Control of Electric Vehicle Motor System EGH456 Embedded Systems Supporting Material*, Pages 2–8, 2025. [Online]. Available: https://canvas.qut.edu.au/courses/20807/pages/egh456-assessment-2-design-of-embedded-system-for-electric-vehicle-group-project-details?module_item_id=1889133.

[5] Texas Instruments, *DRV832x 6 to 60-V Three-Phase Smart Gate Driver*, Page 42, 2017. [Online]. Available: https://www.ti.com/product/DRV8323.

[6] maxon, *Ec 32 flat*, 2021. [Online]. Available: https://www.maxongroup.com/medias/sys_master/root/8882562662430/EN-21-292.pdf.

[7] chalti. "Proportional integral derivative controller in control system. "[Online]. Available: https://www.geeksforgeeks.org/proportional-integral-derivative-controller-in-control-system/.

[8] Texas Instruments, *OPT3001 Ambient Light Sensor Datasheet*, Pages 20–23, 2017. [Online]. Available: https://www.ti.com/product/OPT3001.

[9] Texas Instruments, *BOOSTXL-SENSORS Sensors BoosterPack Plug-in Module*, Page 4, 2018. [Online]. Available: https://www.ti.com/lit/ug/slau666b/slau666b.pdf?ts=1748613205413.

[10] J. Huang. "Grove - Temp and Humi Sensor(SHT31). "[Online]. Available: https://wiki.seeedstudio.com/Grove-TempAndHumi_Sensor-SHT31/.

[11] Texas Instruments, *Tiva™ C Series TM4C1294 Connected LaunchPad Evaluation Kit EK-TM4C1294XL User's Guide*, Pages 10–20, 2016. [Online]. Available: https://www.ti.com/tool/EK-TM4C1294XL.

[12] J. Haviland, *I2C Drivers*, Source Code Repository, 2019. [Online]. Available: https://github.com/jhavl.

[13] N. Nilsson. "Vl6180x_vl53l0x_msp430. "[Online]. Available: https://github.com/artfulbytes/vl6180x_vl53l0x_msp430.

[14] G. Hunter. "Exponential moving average (ema) filters." Last updated Apr 11, 2024. [Online]. Available: https://blog.mbedded.ninja/programming/signal-processing/digital-filters/exponential-moving-average-ema-filter/.

[15] T. Erik, *tauno-serial-plotter*, Documentation and Github Repository, 2024. [Online]. Available: https://github.com/taunoe/tauno-serial-plotter.

[16] Texas Instruments, *TivaWAre Graphics Libraryu (grlib)*, Source Code (API headers and function comments), Part of TivaWare Peripheral Driver Library, 2007.

Appendices

A Setter and Getter Functions

```
/**
 * @brief Retrieves a copy of the shared value in a thread-safe manner.
 *
 * @param  dataPoint     Pointer to the shared data structure.
 * @param  buff          Pointer to a buffer where the value will be copied.
 * @param  blockingTime  Maximum time to wait for the mutex.
 *
 * @return 0 on success, -1 on timeout or error acquiring mutex.
 */
uint8_t getter(sharedValues *dataPoint, val* buff, TickType_t blockingTime){
    if (xSemaphoreTake(dataPoint->mutex, blockingTime) != pdPASS){
        return -1;
    }
    *buff = dataPoint->values;
    xSemaphoreGive(dataPoint->mutex);
    return 0;
}

/**
 * @brief Sets the shared value in a thread-safe manner.
 *
 * @param   dataPoint    Pointer to the shared data structure.
 * @param   values      New values to set.
 * @param   blockingTime Maximum time to wait for the mutex.
 *
 * @return 0 on success, -1 on timeout or error acquiring mutex.
 */
uint8_t setter(sharedValues *dataPoint, val values, TickType_t blockingTime){
    if (xSemaphoreTake(dataPoint->mutex, blockingTime) != pdPASS){
        return -1;
    }
    dataPoint->values = values;
    xSemaphoreGive(dataPoint->mutex);
    return 0;
}
```

B System Architecture