Aivo Pütsep

Programmable Boost Controller

Metropolia University of Applied Sciences

Bachelor of Engineering

Electronics Degree Programme

Bachelor’s Thesis

21 May 2023

Abstract

Author:	Aivo Pütsep
Title:	Programmable Boost Controller
Number of Pages:	55 pages + 1 appendix
Date:	21 May 2023
Degree:	Bachelor of Engineering
Degree Programme:	Electronics
Professional Major:	
Supervisor:	Janne Mäntykoski, Senior Lecturer

This research centers on the development of a programmable boost controller, a device to effectively regulate the intake air pressure within a turbocharged internal combustion engine. The purpose of this study was to engineer a system with the possibility to choose from different power levels the engine is producing by adjusting the boost output of the turbocharger. A device like that is of significant interest to automotive enthusiasts, who continuously seek ways to enhance their vehicle's performance.

The programmable boost controller was designed using a combination of different components and control algorithms. One vehicle with a turbocharged internal combustion engine served as a testing platform. The integration and following testing of the boost controller took place within this car, a 1992 Audi S4, allowing for an in-depth analysis of its performance and impact on the engine's overall power output.

The results of this study showed effective boost control and interface performance, although minor issues were identified. These findings emphasize the need for further enhancements in future iterations.

This research contributes to the ongoing research within automotive performance tuning. The creation and successful implementation of this programmable boost controller provides a promising path for further advancements in engine performance optimization.

Keywords: ESP32, PID, Wastegate, Pressure, Controller

Contents

List of Abbreviations

CAD:	Computer-aided design
CRC:	Cyclic redundancy check
DAQ:	Data acquisition
ECU:	Engine control unit
GPIO:	General purpose input/output
LED:	Light-emitting diode.
MAC:	Modern air control
MAF:	Mass Air Flow
OLED:	Organic light-emitting diode
OTA:	Over-the-air
PCB:	Printed circuit board
PID:	A proportional–integral–derivative controller
PWM:	Pulse-width modulation
WG:	Wastegate

1 Introduction

A Programmable Boost Controller is a device that allows for the regulation of pressure generated by the turbocharger. This enables to effectively manage the power output of the engine.

The Programmable Boost Controller was created as an alternative to buying an electronic boost controller which can result in a significant expense and yet be limited in functionality and features. These limitations can be avoided, by researching and developing a device by oneself.

The system's development involved making an easy-to-use human interface with a screen, addressable LEDs, and buttons. The goal was to have a boost pressure control system that was controlled by a PID algorithm. The Boost Controller directly controls a solenoid with works with the wastegate system. Subsequently, it was thought that the device should establish communication with an onboard Engine Control Unit (ECU) using the CAN Bus protocol.

The development began with researching the required components and determining if any components that had bed acquired before can be re-utilized. Some components that were initially considered suitable for the project were revised later in the process. The first microcontroller was an Arduino Nano, which seemed appropriate, yet gave complications with the OLED screen refresh rate. This was resolved by incorporating a more powerful development board (ESP32), which also gives further development opportunities in the future. Moreover, as ESP32 features built-in wireless communication features such as Bluetooth and Wi-Fi. It was decided that an ESP32 microcontroller should be used as the main processing unit.

Once the main idea was clear, the prototyping started by laying out the components on a breadboard. Simultaneously the development began on the wiring schematics. Firmware development also began and the initial prototype of the device was created.

A pneumatic test bench was built to test the Boost Controller in a controlled environment. Suitable plastic enclosures were designed, and 3D printed.

The first working prototype was installed on a turbocharged car, which gave the capability to gather vital data early in development. Although due to evident limitations encountered on the constructed test bench, the collected data yielded artificial results to some degree. The data acquisition was done with an oscilloscope, Excel, and Arduino IDE.

The initial prototype proved to be successful and became a functioning boost controller, with minor limitations. During the continued development, a second prototype incorporated some additional features such as CAN Bus compatibility and an overall reduction in the dimensions by implementing printed circuit boards (PCBs).

2 Theory and Components

The aim of the project was to have a working device that helps regulate a turbocharged internal combustion engine’s intake air pressure level. The power output of the engine is almost proportional to the amount of air forced into the engine. Conversely the intake pressure can be used to restrict the power output of the engine.

This programmable boost controller is designed for vehicles with high-performance engines that have been upgraded for additional output.

The electronic boost controller works by operating a MAC Solenoid via a 12V PWM [12.] signal that controls the wastegate linear actuator of the turbocharger. The wastegate has a mechanical built-in spring that acts as a passive boost regulator which is related to the spring stiffness. The higher the stiffness of the spring, the more pressure can be accumulated in the intake compressor before the wastegate valve is pushed open by the intake pressure.

The car in which the boost controller was mounted has a spring stiffness that limits the boost pressure up to 0.6 Bar (gauge pressure), after which the wastegates valve is pushed open by the engine's intake pressure. In this case, the turbocharger's wastegate regulates the pressure only up to 0.6 Bar(g). By incorporating an electronic boost controller, this limitation can be avoided, and higher inlet pressures can be achieved.

The MAC solenoid was programmed to open at 0.6 Bar (PWM 100%), thereby opening the negative pressure line and keeping the wastegate valve closed after 0.6 Bar no matter the spring pressure. Then before reaching the desired pressure (Setpoint), the PID Controller steps in and gradually decreases and increases the PWM signal to get a controlled and steady pressure output without oscillations and reducing the overshoot to a minimum.

On the car the Boost Controller was installed the MAC Solenoid is connected as follows

- Port 1: To the turbo compressor inlet hose
- Port 2: To the Wastegate lower chamber
- Port 3: To the turbo compressor outlet hose

When the MAC Solenoid is in the OFF position or disconnected from the Boost Controller (Figure 1) the air path is from Port 3 to Port 2, this means that the boost pressure is only being controlled by the mechanical spring which in our case is 0.6 Bar (g) pressure.

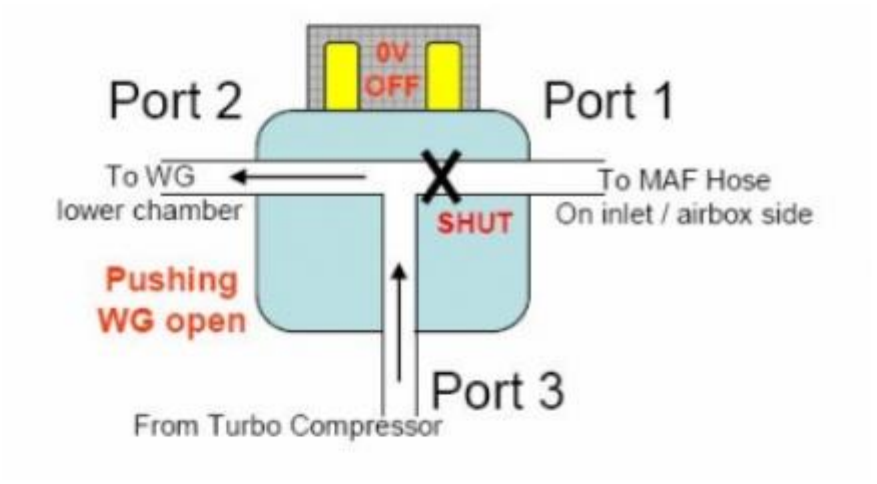


Figure 1. Mac Solenoid OFF position [1].

When the MAC solenoid is in the active or "ON position (Figure 2), it also provides an air path from Port 2 to Port 1. In this scenario then ambient air pressure is presented to the lower wastegate chamber and so the mechanical spring continues to keep the wastegate closed. Another way to think of this is that Port 1 provides the path of least resistance for air to travel out of the lower wastegate chamber. [1.]

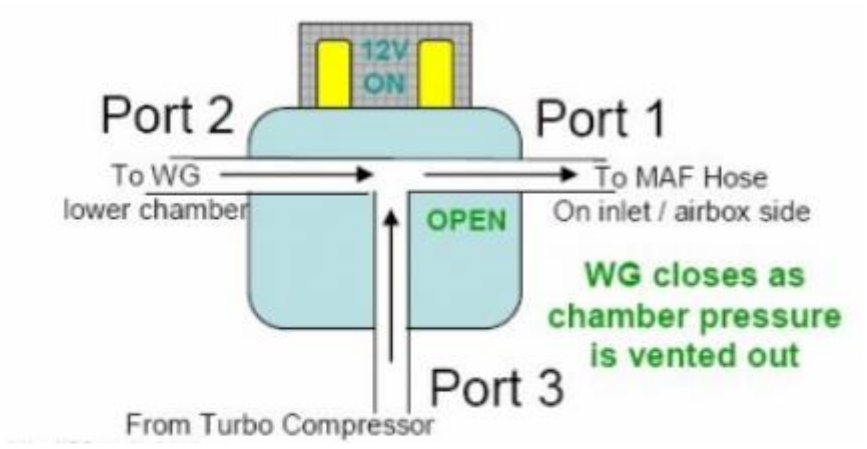


Figure 2. Mac Solenoid ON position [1].

When the MAC solenoid changes into a closed state during the inlet pressures above 0.6 Bar(g), the wastegate valve opens as in normal operation, diverting the exhaust gasses from the turbine of the turbocharger, reducing the boost pressure. By adjusting the wastegate actuator, the electronic boost controller can increase or decrease the boost pressure past 0,6 Bar (Figure 3), depending on the user's needs.

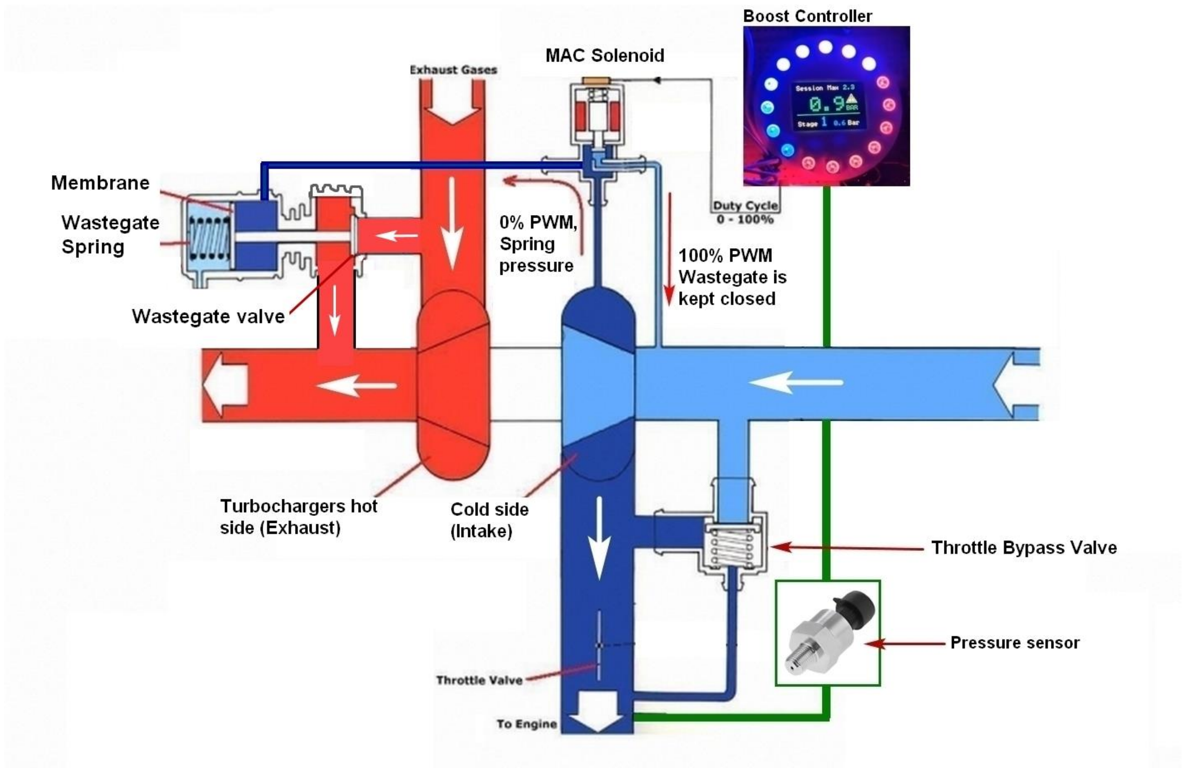


Figure 3. Overview of the Pressure control system [2].

Theoretically, the electronic boost controllers can efficiently optimize the boost pressure for different engine operating conditions. For example, during the first and second gear, the boost pressure can be reduced to improve traction. Conversely, during higher gears and high-load conditions, the boost pressure can be increased to improve power output and performance. This option can be added in future iterations.

An electronic boost controller can help prevent engine damage by preventing over-boosting, which can cause the engine gaskets to fail or in the worst case to ‘’lean out’’ the engine causing detonation in the engine. By controlling the boost pressure more precisely, the electronic boost controller can help ensure that the turbocharger operates within safe limits, reducing the risk of damage or failure. [1.]

2.1 ESP32 Microcontroller

As stated above the ESP32 Chip (Figure 4) was chosen to be the main control unit of the device. The ESP32 was chosen over the preliminary choice of Arduino Nano, as it did not provide the desired refresh rate on the OLED screen. Various solutions were tried to resolve the problem. But none really worked. Switching to the ESP32 however, instantly fixed the problem.

The ESP32 has many more advantages over the Arduino Nano like faster processing power (dual-core processor with clock speeds up to 240 MHz), more memory (able to run more complex programs and store extra data on EEPROM), and more GPIO pins, making it a preferred choice for this project. The ESP32 features built-in Bluetooth and WIFI, which offers more possibilities for future development, for example the developed firmware supports Over-The-Air (OTA) firmware updates. [3.]

The ESP32 Specifications [3, p. 1-4.]:

- Microcontroller: Tensilica Xtensa LX6 dual-core 32-bit processor, running at up to 240 MHz
- Operating Voltage: 2.2V to 3.6V
- Wi-Fi: 802.11b/g/n/e/i, with support for WPA/WPA2/WPA3 and WPS encryption
- Bluetooth: Bluetooth Low Energy (BLE) and Classic Bluetooth with support for BLE Mesh networking
- RAM: 520 KB SRAM
- Flash Memory: 4MB or 8MB
- GPIO: thirty-four programmable pins, with support for digital input/output, PWM, and interrupt handling
- ADC: 12-bit SAR ADC up to 18 channels
- DAC: 8-bit resolution with two channels
- Peripherals: I2C, SPI, UART, I2S, and SDIO interfaces
- Operating Temperature: -40°C to +125°C

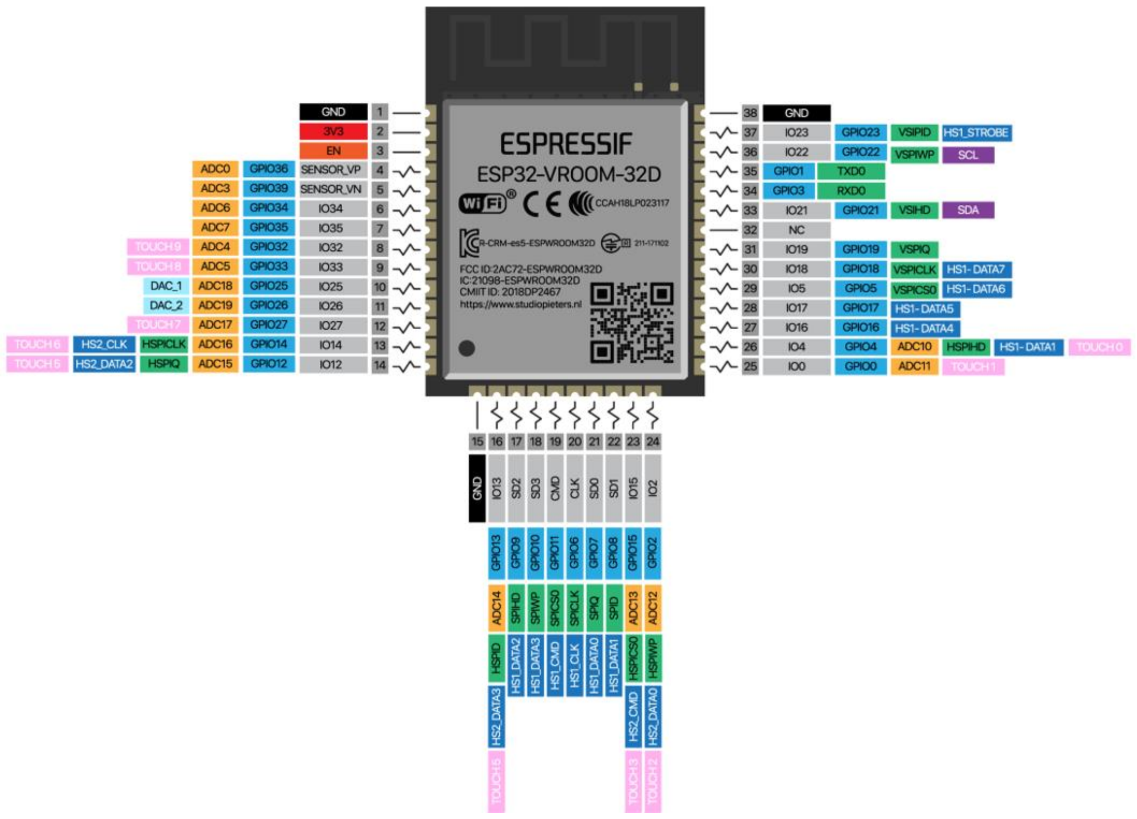

ESPRESSIF
ESP32-VROOM-32D
GND 1
3V3 2
EN 3
ADC0 GPIO36 SENSOR_VP 4
ADC3 GPIO39 SENSOR_VN 5
ADC6 GPIO34 IO34 6
ADC7 GPIO35 IO35 7
TOUCH9 ADC4 GPIO32 IO32 8
TOUCH8 ADC5 GPIO33 IO33 9
DAC_1 ADC18 GPIO25 IO25 10
DAC_2 ADC19 GPIO26 IO26 11
TOUCH7 ADC17 GPIO27 IO27 12
TOUCH6 HS2_CLK HSPICLK ADC16 GPIO14 IO14 13
TOUCH5 HS2_DATA2 HSPIQ ADC15 GPIO12 IO12 14
38 GND
37 IO23 GPIO23 VSPID HS1_STROBE
36 IO22 GPIO22 VSPIWP SCL
35 GPIO1 TXD0
34 GPIO3 RXD0
33 IO21 GPIO21 VSPIHD SDA
32 NC
31 IO19 GPIO19 VSPIQ
30 IO18 GPIO18 VSPICLK HS1-DATA7
29 IO5 GPIO5 VSPICS0 HS1-DATA6
28 IO17 GPIO17 HS1-DATA5
27 IO16 GPIO16 HS1-DATA4
26 IO4 GPIO4 ADC10 HSPIHD HS1-DATA1 TOUCH0
25 IO0 GPIO0 ADC11 TOUCH1
15 GND
16 IO13 GPIO13 ADC14 HSPID HS2_DATA3 TOUCH6
17 SD2 GPIO9 SPIHD HS1_DATA2
18 SD3 GPIO10 SPIWP HS1_DATA3
19 CMD GPIO11 SPICS0 HS1_CMD
20 CLK GPIO6 SPICLK HS1_CLK
21 SD0 GPIO7 SPIQ HS1_DATA0
22 SD1 GPIO8 SPID HS1_DATA1
23 IO15 GPIO15 ADC13 HSPICS0 HS2_CMD TOUCH3
24 IO2 GPIO2 ADC12 HSPIWP HS2_DATA0 TOUCH2

Figure 4. Pinout of an ESP32 Microcontroller chip [4].

While the ESP32 chip is used in many development boards, a Wemos D1S (Figure 5) development board was chosen due to its compact size. The design of the boost controller enclosure was mainly based on the dimensions of the instrument cluster and the development board. Consequently, smaller dimensions were favoured in the development process. The Wemos D1S board is only slightly larger than the microcontroller itself making it ideal for this project. In Figure 6 we can see that the case circumference of the boost controller is not much larger than the D1S development board.

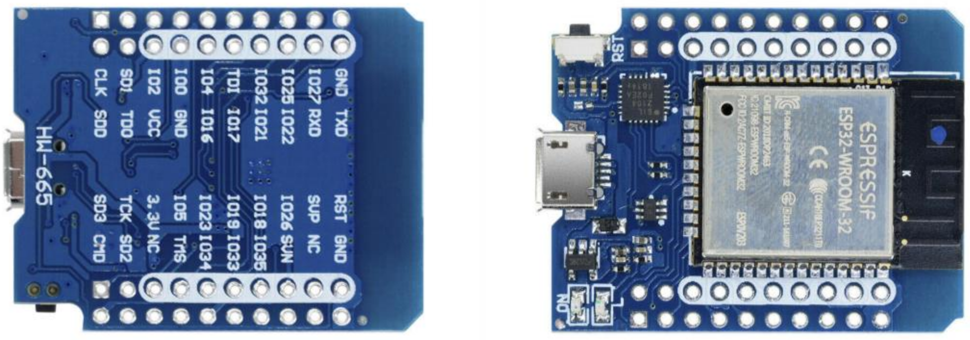


Figure 5. Wemos D1S ESP32 Development board [5].

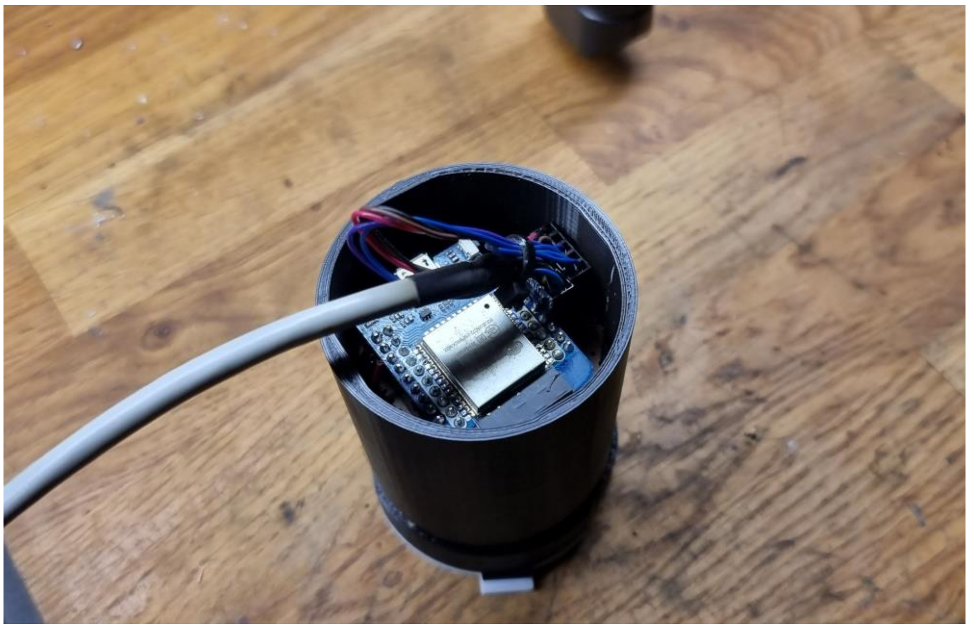


Figure 6. The Wemos D1S ESP32 inside of the plastic enclosure.

2.2 Pressure Sensors

A pressure sensor is a device that senses fluid or gas pressure and converts it into an electrical Analog signal. First, it was chosen to use a 0-3.3 Bar(g) sensor as seen in Figure 7, but later when testing on a car, it proved not to be the best choice. It did not show the negative pressure, therefore it was decided to use another sensor from a previous project, which can be seen in Figure 8.

The accuracy of a pressure sensor depends on many factors, like the design of the diaphragm, the quality of the sensing element, wiring and connector, and also the calibration of the sensor is a important factor. [6.]

The pressure sensor that was used, its specifications are [7]:

- Pressure Range: -1…3 Bar(g)
- Material: Plastic
- Output: 0.1 V… 4.0 V linear voltage output.
- Working Temperature: -40 … +125 ºC
- Works for oil, fuel, water, or air pressure.
- Accuracy: ±1%

Figure 7. Initially chosen pressure sensor, later revised [6].

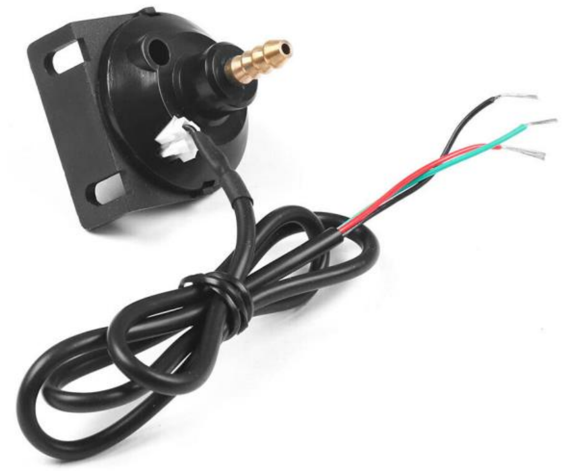


Figure 8. The chosen pressure sensor for the project [7].

The pressure sensor that was chosen had a more or less linear output, as it simplifies the interpretation of its readings. Table 1 depicts the corresponding output voltage (V) for the given pressure levels (Bar(g)). Figure 9 provides a visual representation of the data presented in Table 1. A linear trend can be seen which confirms that the pressure sensor's output is indeed linear. This means that as pressure increases or decreases, the voltage output changes accordingly.

Table 1. The pressure sensors voltage vs pressure [7].

Voltage	Pressure BAR
0.1	-1
0.2	-0.8
0.4	-0.6
0.6	-0.4
0.8	-0.2
1.0	0
1.2	0.2
1.4	0.4
1.6	0.6
1.8	0.8
2.0	1
2.2	1.2
2.4	1.4
2.6	1.6
2.8	1.8
3.0	2
3.2	2.2
3.4	2.4
3.6	2.6
3.8	2.8
4.0	3

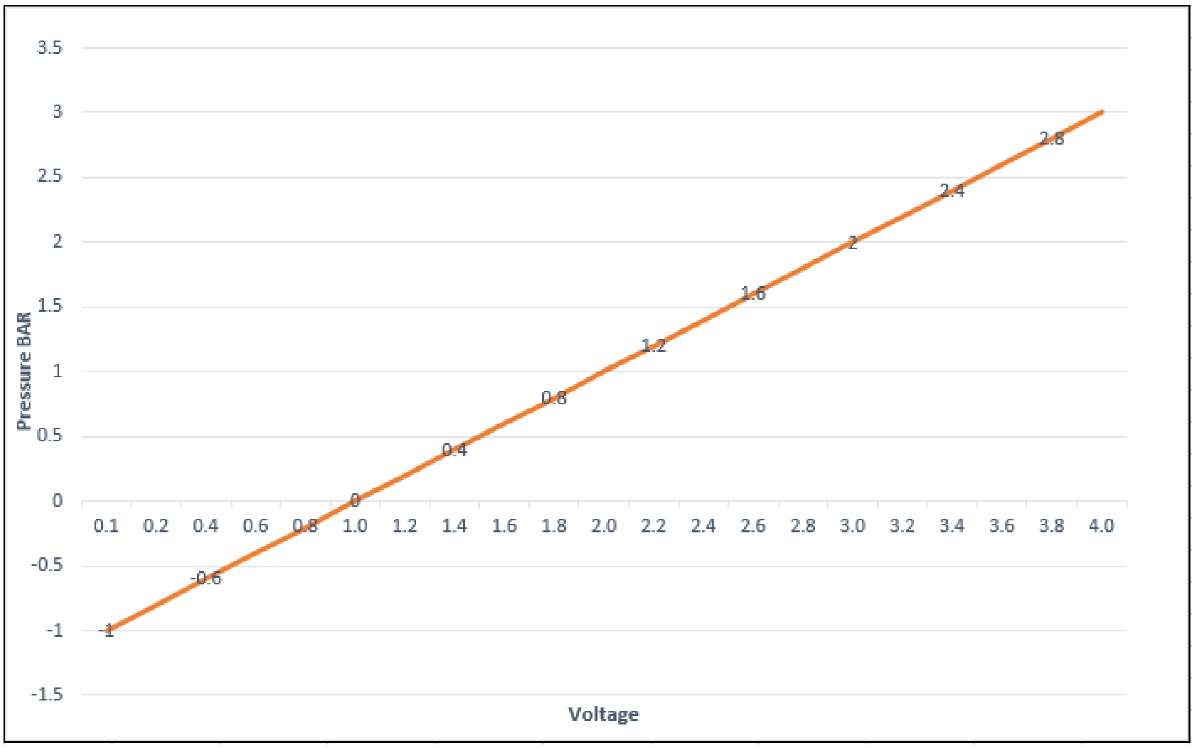


Figure 9. The pressure sensor pressure over voltage relationship [7].

2.3 MAC Solenoid

A MAC (Modern Air Control) Solenoid (Figure 10) is a type of valve commonly used in various industrial applications. The main use of this kind of valve is to control the flow of different gases.

The name MAC Solenoid comes from a company that first developed the valve in the 1940s.

Those solenoids are can be made using different materials like steel, brass or stainless steel. They are available in many sizes and configurations for various applications and requirements.

The working of a MAC solenoid is simple. When an electrical current is applied to the coil, the valve is pulled opens, and when no current is induced, the valve is pushed to a closed position with the built-in spring.

The specific solenoid that was used has an internal 25.4 Ω coil that provides air control up to 10 Bar, which is more than required, as for current purposes more than 3 Bar(g) pressure levels are not expected.

The electronic solenoid works with a 30Hz 14.2V PWM signal. PWM can be used to accurately control the speed and position of the actuator. The following section outlines Pulse-Width Modulation (PWM). [8.]

Figure 10. MAC Solenoid [8].

2.4 Pulse Width Modulation

PWM (Pulse Width Modulation) is a technique used in Electronics. With a PWM signal, the amount of power delivered to a component or device can be controlled by switching the state ON (HIGH) and OFF (LOW) at a specific frequency, and the average power output can be determined by the ratio of ON time to OFF time, which is known as the Duty Cycle.

Duty cycle refers to the percentage of time when the signal state is in the ON state compared to the total time of one complete cycle. When a signal is switched from HIGH to LOW with a specific frequency and duty cycle a square wave is formed.

A square wave is one of the common forms of PWM signals. A duty cycle of 50% would mean the signal is half the time HIGH and half the time LOW in one complete cycle. A duty cycle of 25% would mean that the signal is a quarter of the time HIGH and three-quarters of the time LOW in one cycle.

A further explanation can be seen in Figure 11, where the analog signal value is mapped from 0 to 255. That means that the digital representation of the analog signal ranges from 0 to 255. Each increment in the digital value corresponds to an increment of approximately 0.39% of the total analog range. This mapping allows for the conversion between analog and digital signals, enabling the microcontroller to interpret and process analog data using its 8-bit binary system.

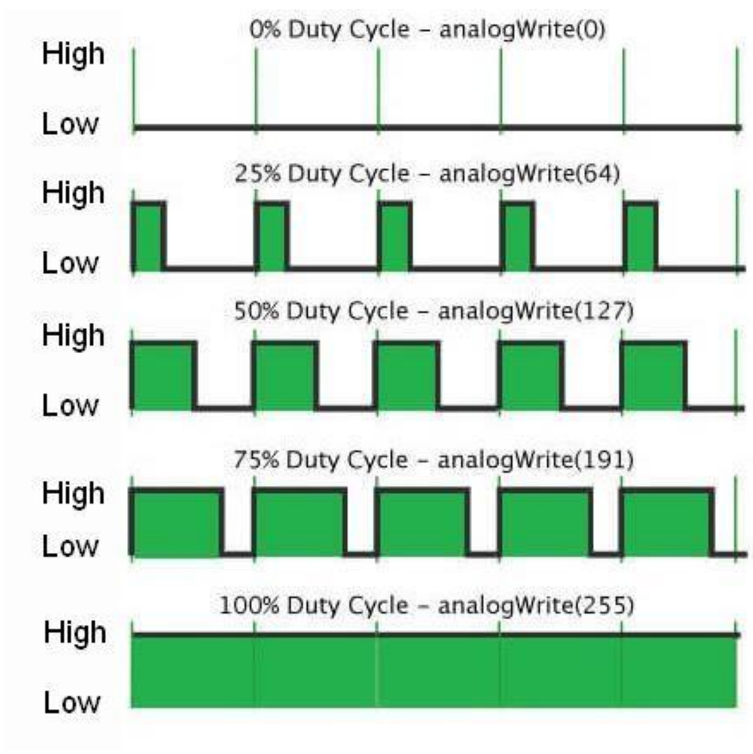

0% Duty Cycle - analogWrite(0)
High
Low
25% Duty Cycle - analogWrite(64)
High
Low
50% Duty Cycle - analogWrite(127)
High
Low
75% Duty Cycle - analogWrite(191)
High
Low
100% Duty Cycle - analogWrite(255)
High
Low

Figure 11. Duty Cycle vs Percentage vs Analog value [9].

When the duty cycle of the square wave is adjusted, the amount of power delivered can be easily controlled without changing the frequency or amplitude of the signal. The frequency determines how fast the signal is switched ON and OFF (HIGH and LOW). Along with the frequency and duty cycle, the timing, intensity, and duration of the signal can be established. In the current implementation, when the duty cycle of the signal is adjusted, the MAC solenoid can be easily controlled, allowing for precise control of the actuator position and movement. [9.]

2.5 OLED Screen and Addressable LEDs

2.5.1 OLED Screen

A large emphasis was set to find a good quality screen, as it is the main component users would interact with. Good-quality small OLED screens have emerged in recent years. In the search for a screen that met the size requirements, a small OLED screen with dimensions of 96x64 pixels (Figure 12) was used. This screen utilizes RGB OLED technology, which theoretically supports up to 65,000 colors. The display is built on SSD1331 technology.

OLED displays, including this color OLED screen, are known for their low power consumption (only 25 mA when all pixels are white) and good brightness levels compared to LCD screens. The communication protocol employed by this display is SPI, which means that, in addition to the VCC and GND connections, it requires five additional GPIO pins. Fortunately, there are multiple libraries available to facilitate the usage of this screen. A library called <SSD_13XX.h> was successfully implemented and performed well. [10.]

Specifications of the screen [10].

• Driver Chip	SSD1331
• Interface:	SPI
• Resolution:	96×64
• Display Size:	0.95inch
• Colours:	Over 65 000
• Viewing Angle:	120°
• Operating Temp:	-20~70
• Operating Voltage:	5V

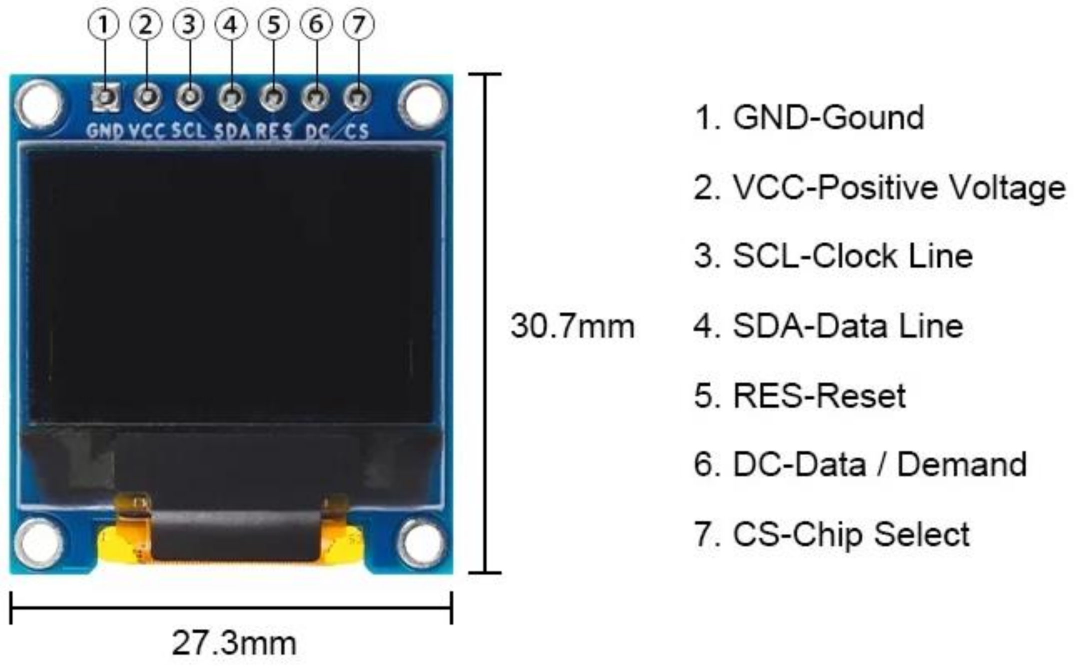


Figure 12. The OLED Screen used in this project [10].

2.5.2 Addressable LEDs

For getting a more interactive user experience addressable LEDs called Neopixels were used in the project (Figure 13). The WS2812 Integrated Light Source, also known as the NeoPixel by Adafruit, represents a significant breakthrough in the pursuit of a straightforward, expandable, and cost-effective full-colour LED solution. Within a compact surface-mount package, red, green, and blue LEDs are seamlessly integrated with a driver chip, all controlled through a single wire. These LEDs offer versatility, allowing them to be used independently, linked together in extended sequences, or combined into captivating and innovative configurations.

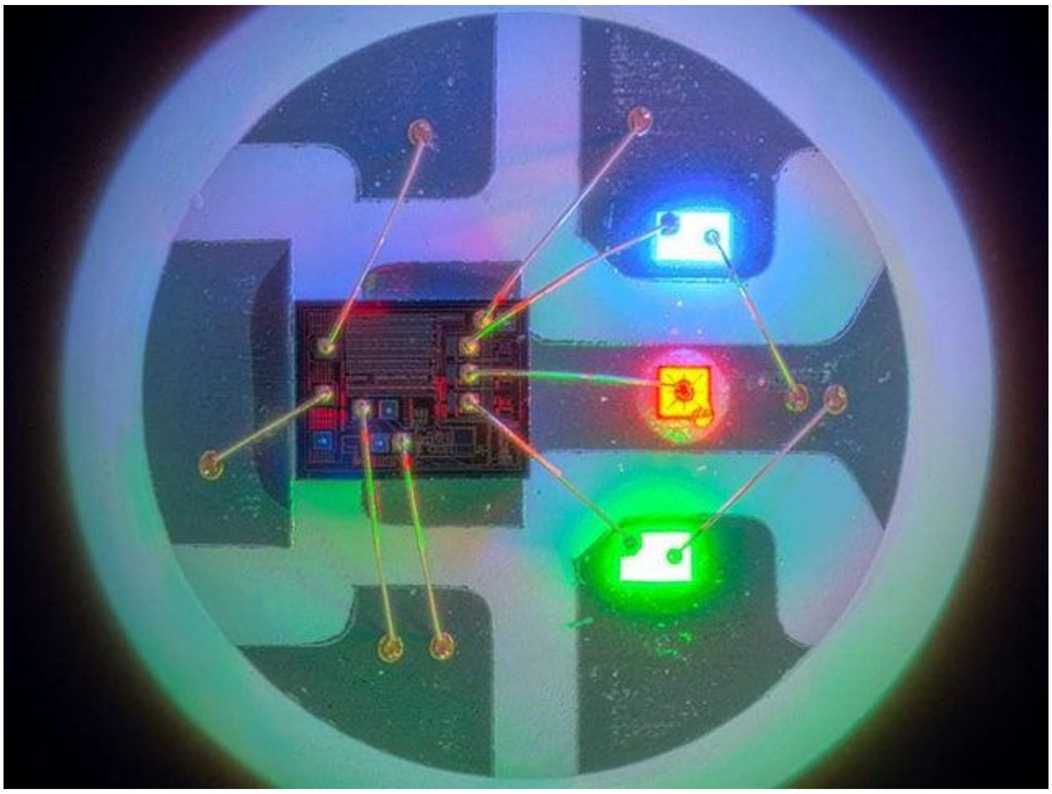


Figure 13. Close-up of an addressable LED [11].

The versatility of addressable LEDs enables the creation of captivating lighting effects and animations. With the ability to control each LED individually, you can achieve dynamic and interactive lighting shows. These LEDs are available in various form factors, including strips, rings, and matrices, providing flexibility for different applications. [11.]

In this specific application, a 16 LED ring with WS2812 LEDs was utilized (Figure 14).

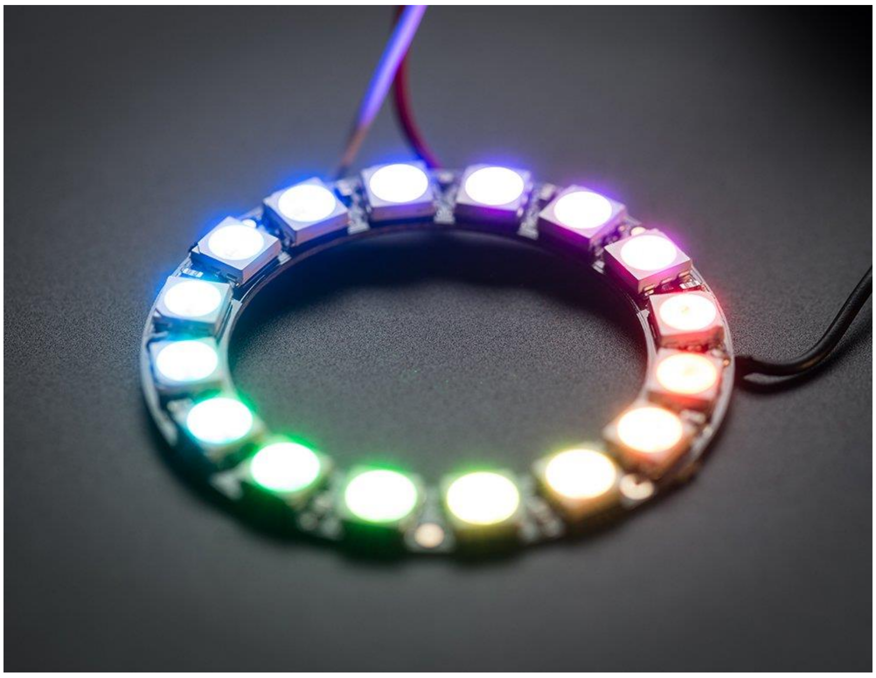


Figure 14. Neopixel Led ring that was used in the project [12].

It was wanted that the LEDs and screen also display the current boost pressure and the negative pressure, with LEDs lighting up accordingly. Also, a small startup animation was implemented to enhance the visual experience. Additionally, flashing LEDs were incorporated to indicate when the maximum selected pressure level has been reached. Furthermore, the LED ring offers additional functionality within the menu system, allowing the user to engage in more immersive interaction.

Later in further iterations, smaller addressable LEDs should be used, called APA102s, which would even further enhance the experience and look much better to the eye. Their footprint is much smaller, so the lighting would be gapless. The connection only has an extra clock pin but uses the same FastLed Library.

2.6 PID Controller

A Proportional Integral Derivative (PID) controller is a precise and automatically optimized control system utilized to regulate various parameters such as temperature, pressure, and speed to desired setpoints. The fundamental mechanism employed by PID controllers revolves around control loop feedback. The PID controller calculates the error by determining the difference between the actual value and the desired value and adjusting the controlling parameters accordingly. This error is continuously evaluated until the process comes to a halt.

A practical example that highlights the effectiveness of PID controllers is Industrial Batch Temperature Control. In closed systems where maintaining constant heat is challenging, PID controllers are employed to regulate the heat supply, ensuring the overall operations are commercially viable and cost-effective.

The proportional component measures the error between the desired value and the actual value, playing a key role in generating corrective responses. The integral component calculates the accumulated sum of past error values and integrates them to determine the integral term. Once the error is eliminated from the system, the integral component ceases to increase. The derivative component anticipates future error values based on current values and is particularly useful in systems with rapid rate changes. The collective operation of these three components gives rise to the name Proportional Integral Derivative (PID) controller.

To implement a stable Pulse Width Modulation (PWM) output, a PID controller was incorporated into the code, using the library " PID_v1.h".
PID calculation (Figure 15) involves three key components: proportional (P), integral (I), and derivative (D). The goal of a PID controller is to compute an output value that minimizes the difference between the desired setpoint and the measured value.

The PID formula:

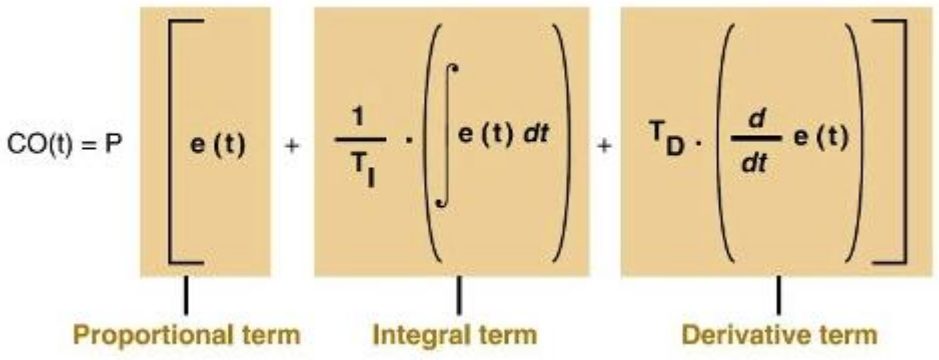


Figure 15. PID Formula [13].

Output = Kp *(error + (1/Ti) * ∫(error)dt + Td * d(error)/dt) [13].

Proportional (P): The proportional term is directly proportional to the current error. It calculates the correction value by multiplying the error by a proportional gain constant (Kp).

Integral (I): The integral term accumulates past errors over time and addresses steady-state errors. It integrates the error over time (∫(error)dt) and scales it by the inverse of the integral time constant (Ti).

Derivative (D): The derivative term predicts the future trend of the error based on its current rate of change. It calculates the rate of change of the error (d(error)/dt) and multiplies it by the derivative time constant (Td).

The controller output is the sum of these three components, and the gains Kp, Ti, and Td need to be appropriately tuned for optimal control performance. [13.]

2.7 Mosfet

A range of MOSFETs and transistors were suitable for this project. The initially chosen Power Transistor functioned satisfactorily. But it disappeared during the testbench phase. Subsequently, a small pre-assembled MOSFET PCB (Figure

16) was chosen, Its Schematic we can see on Figure 23. These PCBs have been found to be convenient for use and have proved to be excellent for prototyping purposes.

The MOSFET board featured two MOSFETs connected in parallel to boost the capacity for handling current. Specifically, it utilized two AOD4184A MOSFETs, which proved to be effective for this project. The specs [14]. for the compact MOSFET PCB provided specifications stating a maximum current capacity of 15 A, which surpassed the requirements. It was interesting to note that when inspecting the datasheet [15.] each MOSFET individually could handle 12 A, leading to a belief that together they should be capable of handling up to 24 A. Nevertheless, both were sufficient for this purpose regarding their current handling. [14.]

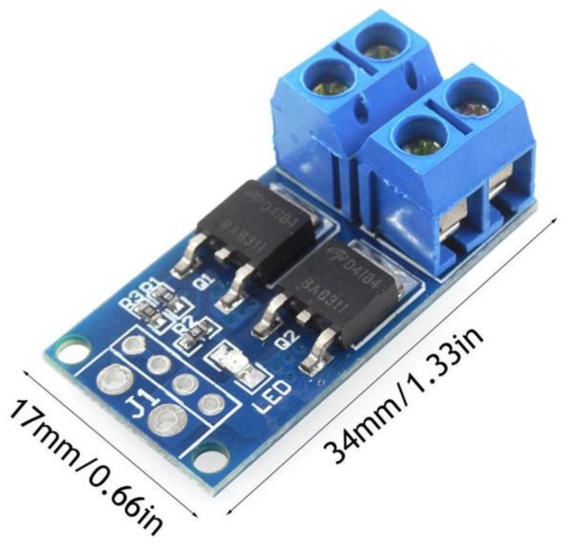


Figure 16. PCB with 2 MOSFETs [14].

2.8 CAN Bus

CAN bus was added to have communication between the car's ECU and the boost controller, enabling communication between the Programmable Boost

Controller and a car’s ECU. This would allow the boost controller to read various data directly from the ECU, such as intake pressure, intake air temperature, voltage, etc. With that eliminates the need for an extra pressure sensor. The Programmable Boost Controller can also send information to the ECU through the CAN bus interface. [17.]

The theory behind CAN bus is based on the principle of distributed control, where each ECU has its own intelligence and can make decisions based on the data it receives from other ECUs on the network, this means that adding a boost controller to the existing network should cause no problems (Figure 17).

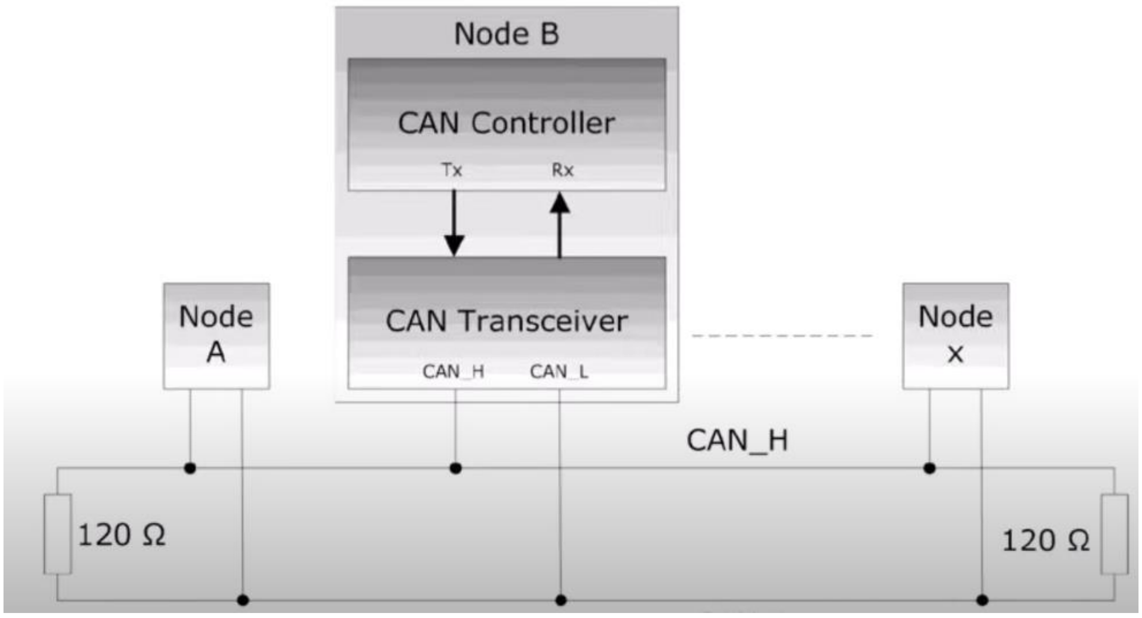


Figure 17. CAN Bus network model [16].

In a CAN bus system, data is transmitted in packets, each containing an identifier, a data field, and a few control bits (Figure 18). The identifier tells the receiving ECU what type of data is being sent, while the data field contains the actual information being transmitted. The control bits provide error checking and other functions.

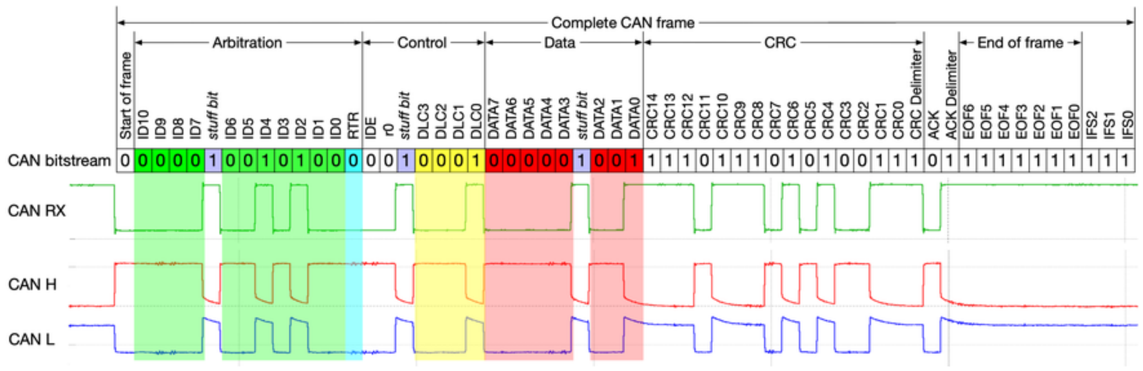


Figure 18. Complete CAN Bus frame [17].

To ensure that the CAN bus network operates correctly, the resistance of the wiring and connectors must be within the specified range. The CAN bus protocol specifies a maximum resistance of 60 Ω for the entire network, including the wiring and connectors. This resistance is achieved by using two 120 Ω termination resistors, one at each end of the network, which effectively creates a 60 Ω resistance between the two ends. [17.]

In this project a MCP2515 CAN Bus module was used (Figure 19), this is a simple module that supports CAN protocol version 2.0B and can be used for communication speeds up to 1Mbps. Its schematic we can see in figure 22. The controller has an integrated SPI interface for communication with the ESP32. [18.]

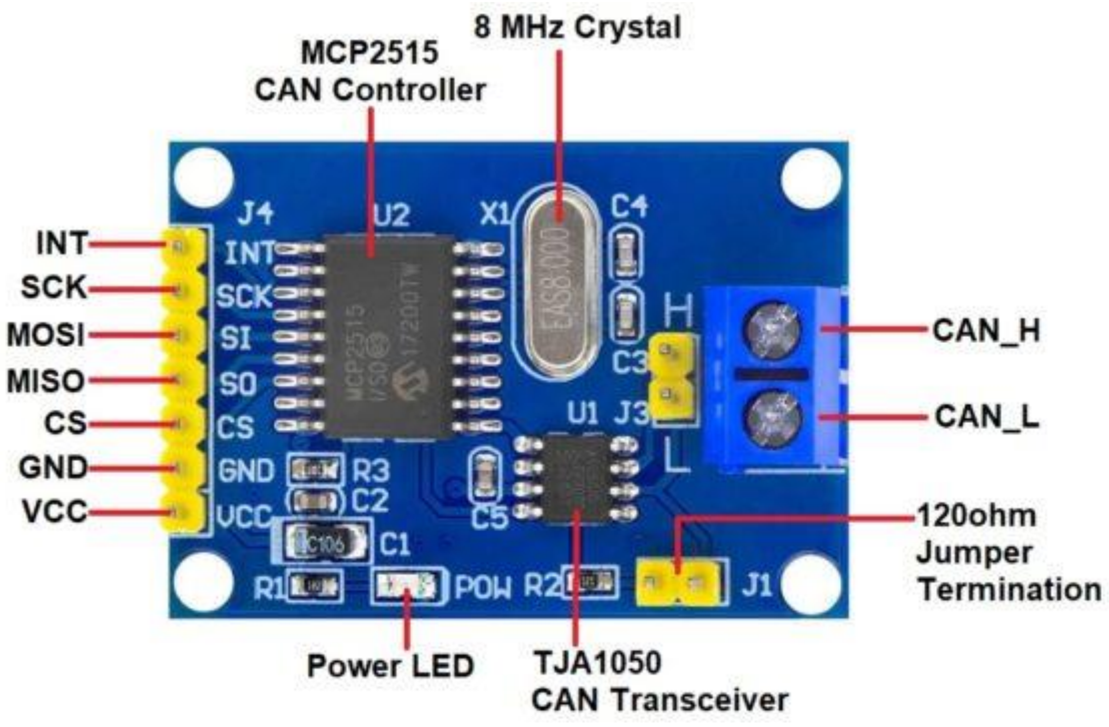


Figure 19. A small CAN bus PCB that was used [18].

3 Design and Implementation

The menu of the device should be user-friendly and easy to navigate. To achieve this a simple diagram was created (Figure 18). Each submenu is identified by an Event number, which corresponds to the code's build structure. Although there are libraries available with templates for creating menus but using those would probably require a longer learning curve than simply writing the code oneself. Therefore, it was decided to keep code writing as simple as possible and mostly rely on the existing code-writing knowledge.

The menu system depicted in Figure 20 serves as a reference, illustrating its initial structure. It should be noted that numerous updates have been implemented since then, resulting in some differences in the current appearance of the actual menu system.

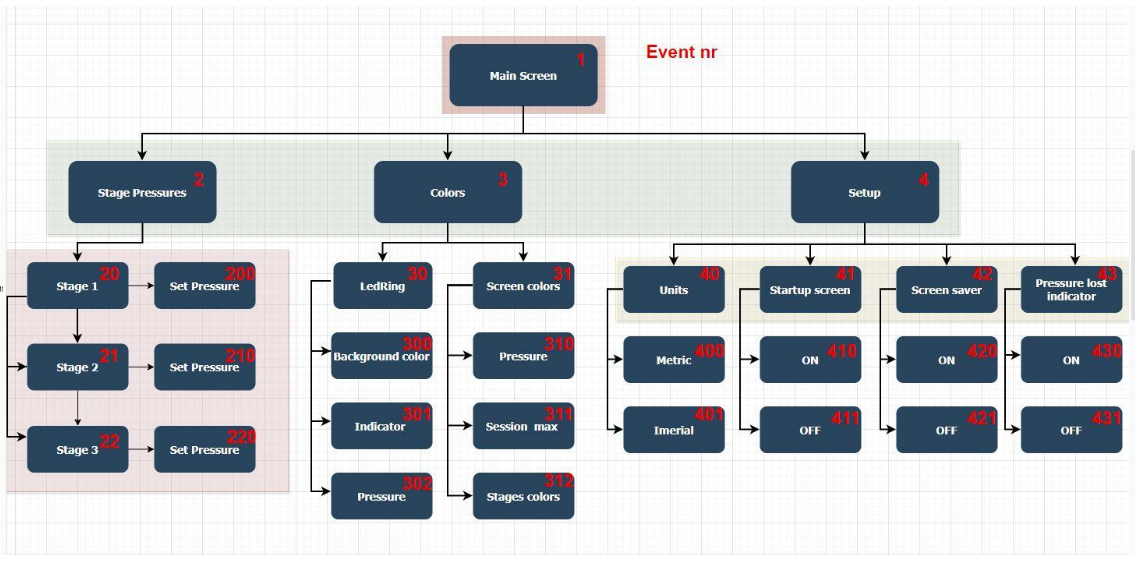
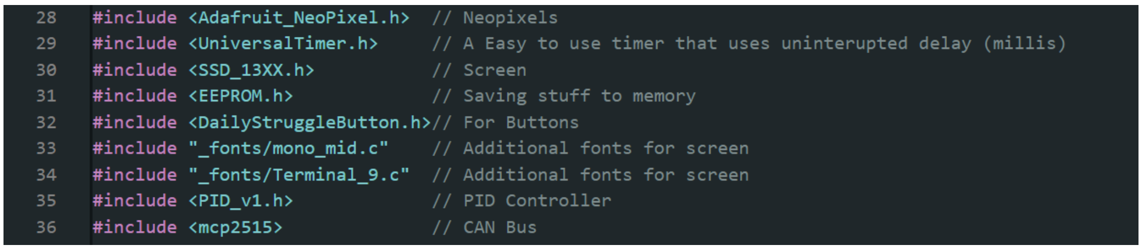


Figure 20. First diagram of the device's menu

3.1 Coding

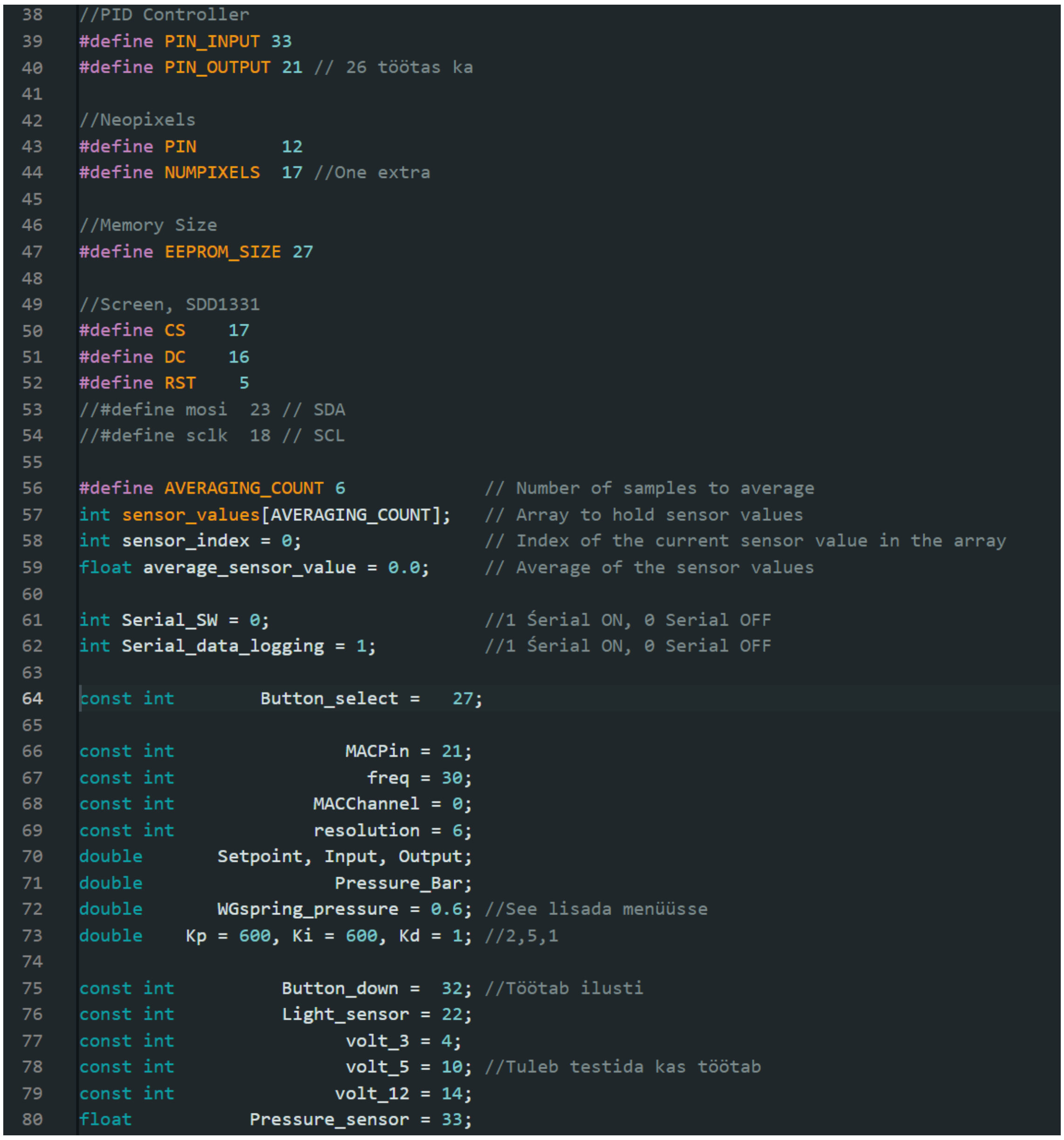
The Code language that was written was C++, and the interface was Arduino IDE. The writing began by researching what libraries to use and how to manage to get them interacting all together. The libraries that were used can be seen in Listing 1.



```
#include <Adafruit_NeoPixel.h>  // Neopixels
#include <UniversalTimer.h>     // A Easy to use timer that uses uninterupted delay (millis)
#include <SSD_13XX.h>           // Screen
#include <EEPROM.h>             // Saving stuff to memory
#include <DailyStruggleButton.h>// For Buttons
#include "_fonts/mono_mid.c"    // Additional fonts for screen
#include "_fonts/Terminal_9.c"  // Additional fonts for screen
#include <PID_v1.h>             // PID Controller
#include <mcp2515>              // CAN Bus
```

Listing 1. Libraries that were used in the code

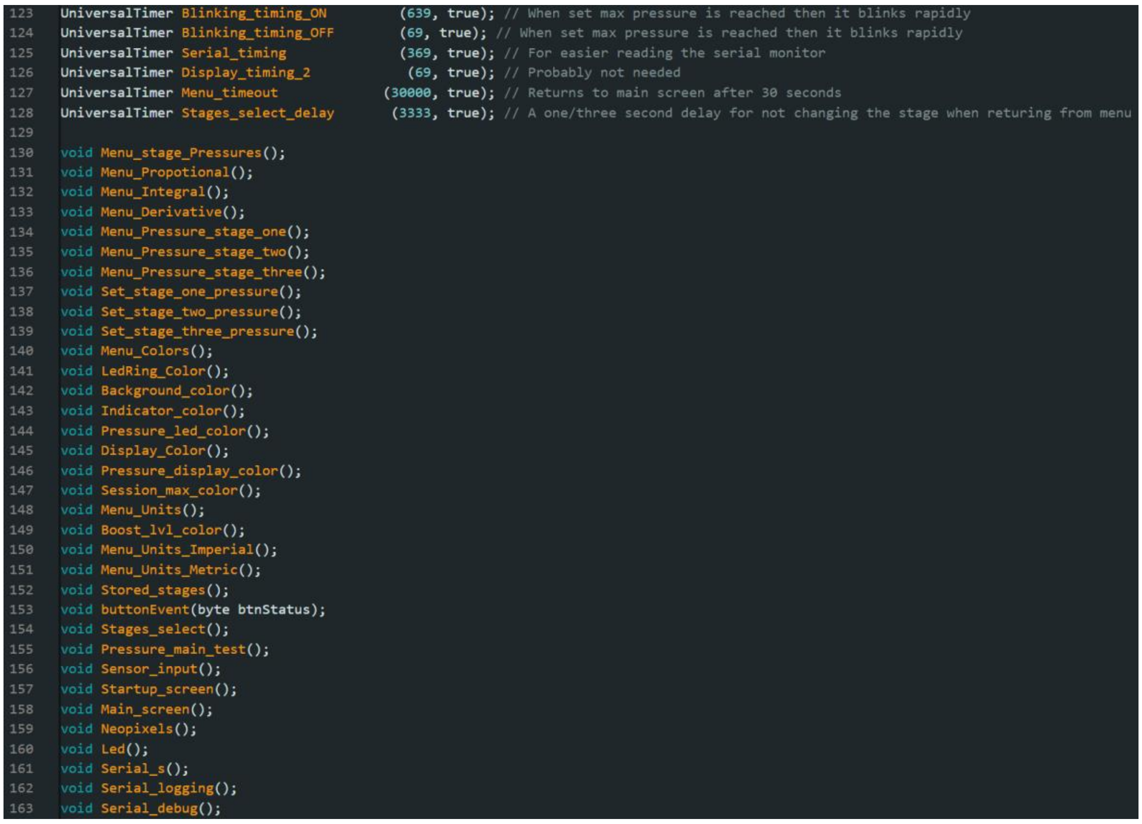
Next, the GPIOs were defined (General Purpose Input/Output, GPIO refers to a set of pins or connectors on a microcontroller) that were used can be seen in Listing 2.



```
//PID Controller
#define PIN_INPUT 33
#define PIN_OUTPUT 21 // 26 töötas ka

//Neopixels
#define PIN        12
#define NUMPIXELS  17 //One extra

//Memory Size
#define EEPROM_SIZE 27

//Screen, SDD1331
#define CS    17
#define DC    16
#define RST    5
//#define mosi  23 // SDA
//#define sclk  18 // SCL

#define AVERAGING_COUNT 6             // Number of samples to average
int sensor_values[AVERAGING_COUNT];   // Array to hold sensor values
int sensor_index = 0;                 // Index of the current sensor value in the array
float average_sensor_value = 0.0;     // Average of the sensor values

int Serial_SW = 0;                    //1 Śerial ON, 0 Serial OFF
int Serial_data_logging = 1;          //1 Śerial ON, 0 Serial OFF

const int        Button_select =   27;

const int                MACPin = 21;
const int                  freq = 30;
const int             MACChannel = 0;
const int             resolution = 6;
double       Setpoint, Input, Output;
double                  Pressure_Bar;
double       WGspring_pressure = 0.6; //See lisada menüüsse
double    Kp = 600, Ki = 600, Kd = 1; //2,5,1

const int          Button_down =  32; //Töötab ilusti
const int          Light_sensor = 22;
const int                volt_3 = 4;
const int                volt_5 = 10; //Tuleb testida kas töötab
const int               volt_12 = 14;
float           Pressure_sensor = 33;
```

Listing 2. GPIOs that were used on the microcontroller

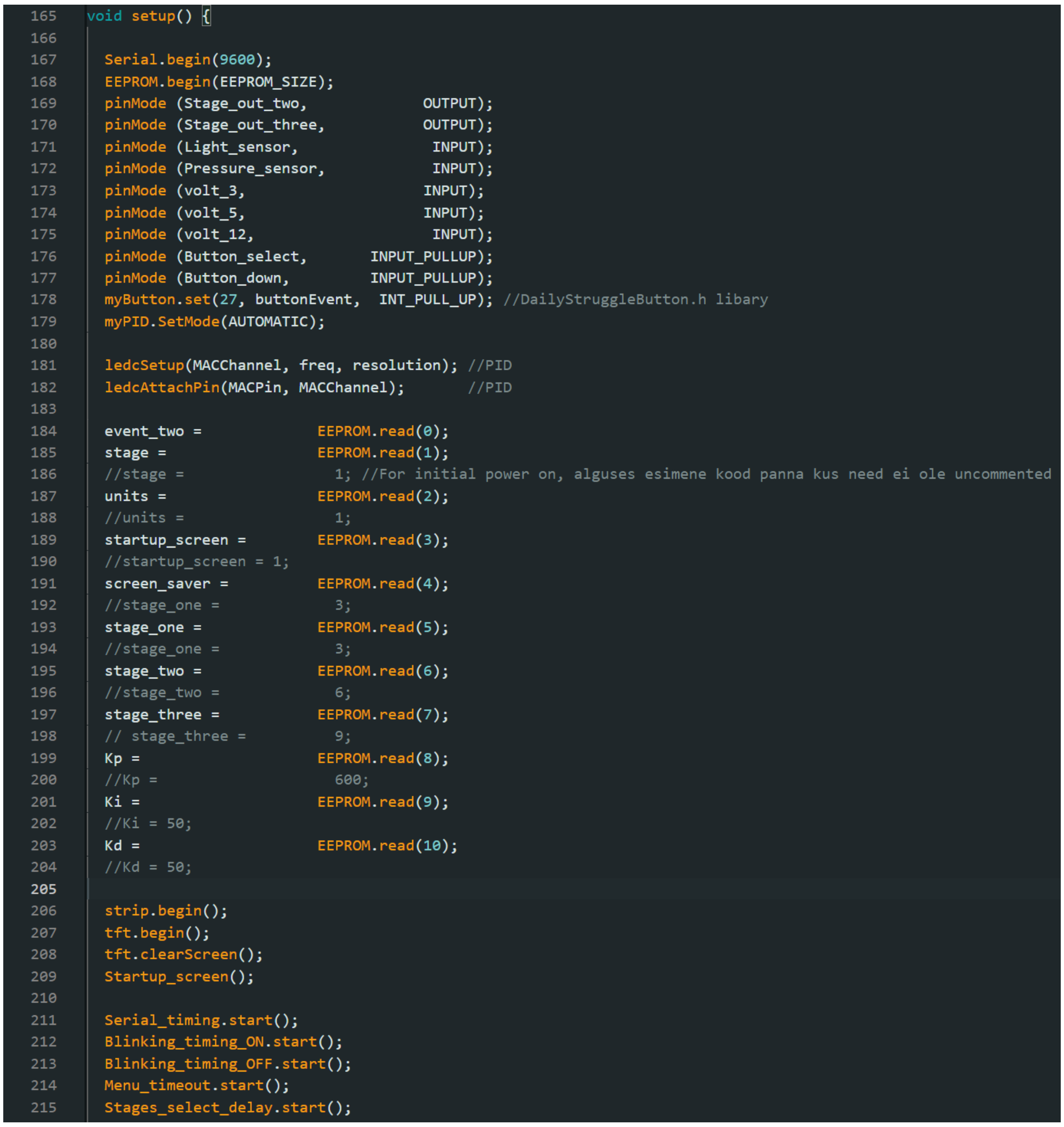
Following that, timers were added as non-blocking delays. Additionally, the 'millis' function was utilized in certain cases. Timers were added with a Timer library to simplify and enhance code readability. After that functions were developed to enhance the understanding and clarity of the existing functions (Listing 3). Although not all of them are utilized in the latest version, these functions have been prepared and are available for implementation per requirement.



```
UniversalTimer Blinking_timing_ON          (639, true); // When set max pressure is reached then it blinks rapidly
UniversalTimer Blinking_timing_OFF         (69, true); // When set max pressure is reached then it blinks rapidly
UniversalTimer Serial_timing               (369, true); // For easier reading the serial monitor
UniversalTimer Display_timing_2             (69, true); // Probably not needed
UniversalTimer Menu_timeout              (30000, true); // Returns to main screen after 30 seconds
UniversalTimer Stages_select_delay        (3333, true); // A one/three second delay for not changing the stage when returing from menu

void Menu_stage_Pressures();
void Menu_Propotional();
void Menu_Integral();
void Menu_Derivative();
void Menu_Pressure_stage_one();
void Menu_Pressure_stage_two();
void Menu_Pressure_stage_three();
void Set_stage_one_pressure();
void Set_stage_two_pressure();
void Set_stage_three_pressure();
void Menu_Colors();
void LedRing_Color();
void Background_color();
void Indicator_color();
void Pressure_led_color();
void Display_Color();
void Pressure_display_color();
void Session_max_color();
void Menu_Units();
void Boost_lvl_color();
void Menu_Units_Imperial();
void Menu_Units_Metric();
void Stored_stages();
void buttonEvent(byte btnStatus);
void Stages_select();
void Pressure_main_test();
void Sensor_input();
void Startup_screen();
void Main_screen();
void Neopixels();
void Led();
void Serial_s();
void Serial_logging();
void Serial_debug();
```

Listing 3. Timers and Functions in the Code

Subsequently, in Setup, the input and output pins were assigned, EEPROM addresses were allocated, and serial communication was initiated. Additionally, the PID controller was initialized, and the execution of the initial functions commenced, including the starting animation (seen in Listing 4).

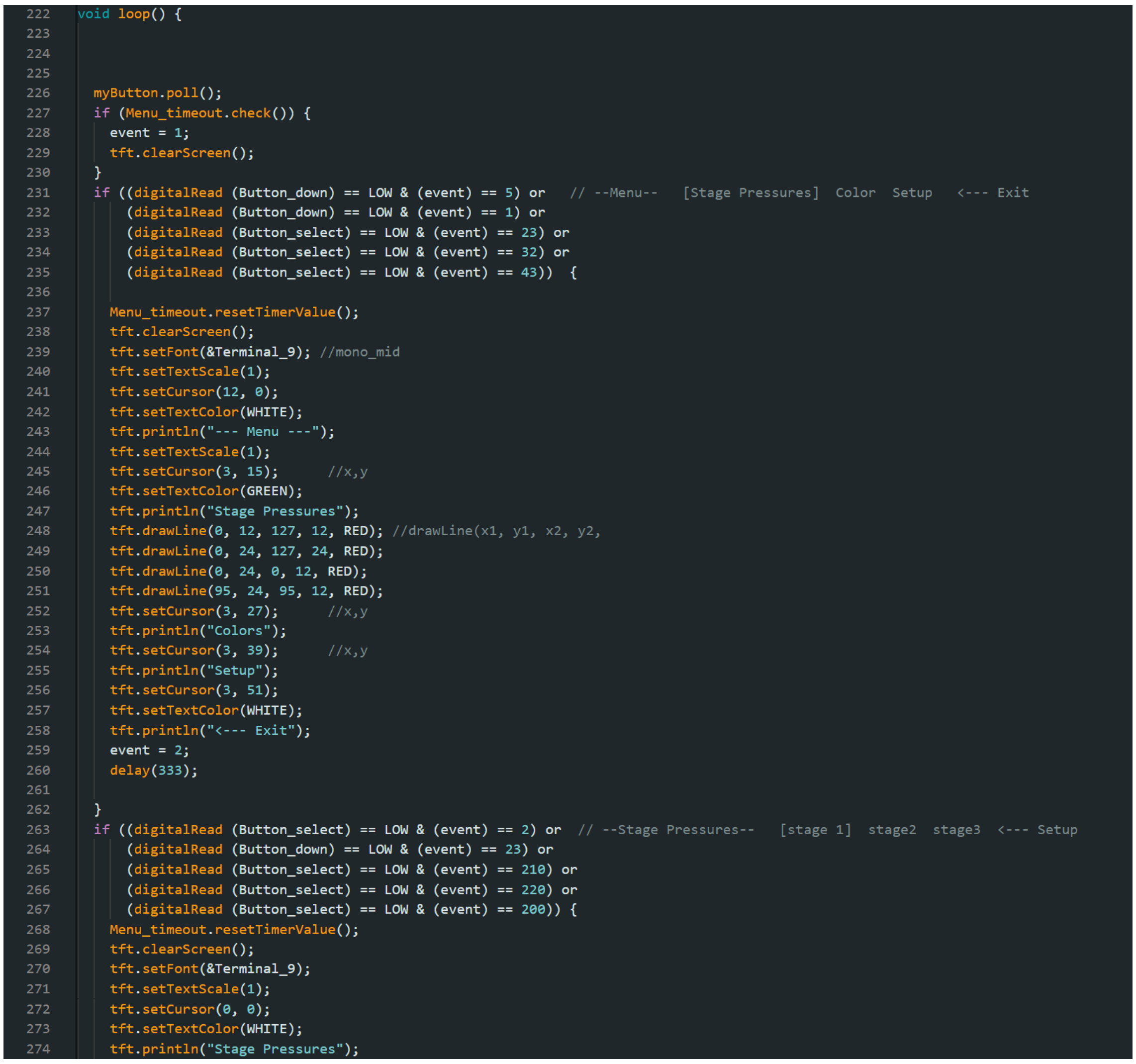


```
void setup() {

  Serial.begin(9600);
  EEPROM.begin(EEPROM_SIZE);
  pinMode (Stage_out_two,             OUTPUT);
  pinMode (Stage_out_three,           OUTPUT);
  pinMode (Light_sensor,               INPUT);
  pinMode (Pressure_sensor,            INPUT);
  pinMode (volt_3,                    INPUT);
  pinMode (volt_5,                    INPUT);
  pinMode (volt_12,                    INPUT);
  pinMode (Button_select,       INPUT_PULLUP);
  pinMode (Button_down,         INPUT_PULLUP);
  myButton.set(27, buttonEvent,  INT_PULL_UP); //DailyStruggleButton.h libary
  myPID.SetMode(AUTOMATIC);

  ledcSetup(MACChannel, freq, resolution); //PID
  ledcAttachPin(MACPin, MACChannel);       //PID

  event_two =             EEPROM.read(0);
  stage =                 EEPROM.read(1);
  //stage =                 1; //For initial power on, alguses esimene kood panna kus need ei ole uncommented
  units =                 EEPROM.read(2);
  //units =                 1;
  startup_screen =        EEPROM.read(3);
  //startup_screen = 1;
  screen_saver =          EEPROM.read(4);
  //stage_one =             3;
  stage_one =             EEPROM.read(5);
  //stage_one =             3;
  stage_two =             EEPROM.read(6);
  //stage_two =             6;
  stage_three =           EEPROM.read(7);
  // stage_three =          9;
  Kp =                    EEPROM.read(8);
  //Kp =                    600;
  Ki =                    EEPROM.read(9);
  //Ki = 50;
  Kd =                    EEPROM.read(10);
  //Kd = 50;

  strip.begin();
  tft.begin();
  tft.clearScreen();
  Startup_screen();

  Serial_timing.start();
  Blinking_timing_ON.start();
  Blinking_timing_OFF.start();
  Menu_timeout.start();
  Stages_select_delay.start();
```

Listing 4. Setup part of the code

Next, the loop was initiated (Listing 5). Here it can be seen how the menu was constructed. The menu was primarily constructed using events and conditional statements (if and if-else). This method of implementation proved to work well. Still, it is possible that a better alternative can be developed in the future. During the intial testing the code was performing as required, therefore this method was kept as is.

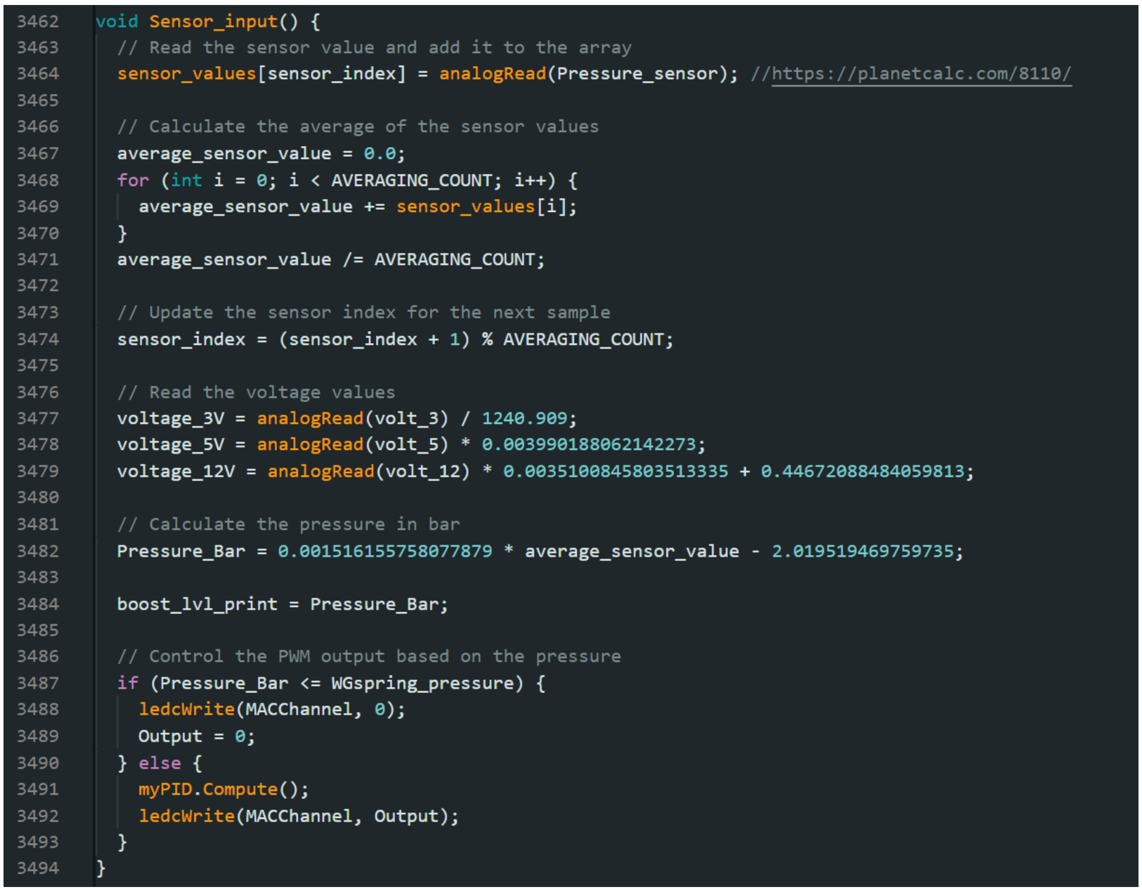
The loop starts by using a timer, every time a button is pressed a 30-second timer starts and when the time runs out it automatically switches back to the main screen, in this case, the main screen is ”Event 1’’.



```
void loop() {



  myButton.poll();
  if (Menu_timeout.check()) {
    event = 1;
    tft.clearScreen();
  }
  if ((digitalRead (Button_down) == LOW & (event) == 5) or   // --Menu--   [Stage Pressures]  Color  Setup   <--- Exit
      (digitalRead (Button_down) == LOW & (event) == 1) or
      (digitalRead (Button_select) == LOW & (event) == 23) or
      (digitalRead (Button_select) == LOW & (event) == 32) or
      (digitalRead (Button_select) == LOW & (event) == 43))  {

    Menu_timeout.resetTimerValue();
    tft.clearScreen();
    tft.setFont(&Terminal_9); //mono_mid
    tft.setTextScale(1);
    tft.setCursor(12, 0);
    tft.setTextColor(WHITE);
    tft.println("--- Menu ---");
    tft.setTextScale(1);
    tft.setCursor(3, 15);      //x,y
    tft.setTextColor(GREEN);
    tft.println("Stage Pressures");
    tft.drawLine(0, 12, 127, 12, RED); //drawLine(x1, y1, x2, y2,
    tft.drawLine(0, 24, 127, 24, RED);
    tft.drawLine(0, 24, 0, 12, RED);
    tft.drawLine(95, 24, 95, 12, RED);
    tft.setCursor(3, 27);      //x,y
    tft.println("Colors");
    tft.setCursor(3, 39);      //x,y
    tft.println("Setup");
    tft.setCursor(3, 51);
    tft.setTextColor(WHITE);
    tft.println("<--- Exit");
    event = 2;
    delay(333);

  }
  if ((digitalRead (Button_select) == LOW & (event) == 2) or  // --Stage Pressures--   [stage 1]  stage2  stage3  <--- Setup
      (digitalRead (Button_down) == LOW & (event) == 23) or
      (digitalRead (Button_select) == LOW & (event) == 210) or
      (digitalRead (Button_select) == LOW & (event) == 220) or
      (digitalRead (Button_select) == LOW & (event) == 200)) {
    Menu_timeout.resetTimerValue();
    tft.clearScreen();
    tft.setFont(&Terminal_9);
    tft.setTextScale(1);
    tft.setCursor(0, 0);
    tft.setTextColor(WHITE);
    tft.println("Stage Pressures");
```

Listing 5, Beginning part of the ‘’Loop’’

Afterward, the creation of various functions was initiated, one of which involves the computation of sensor inputs. An averaging formula was included which will be further discussed in the following paragraphs relating to Measurements and Testing. Additionally, to aid future debugging, the distinct voltages of 3.3 V, 5 V, and 14.2 V could be observed. Subsequently, a straightforward equation was formulated to calibrate the sensor output. For this task, an online calculator [19] was utilized. This webpage offers a convenient tool to determine the equation of a line based on two points, presented in slope-intercept and parametric forms. The sensor was calibrated with analog reading and corresponding pressure levels in a used range against another sensor. As a result, the calibration process was completed effortlessly. All of that can be seen in Listing 6. The whole code can be seen on a shared Dropbox link. [20.]



```
void Sensor_input() {
  // Read the sensor value and add it to the array
  sensor_values[sensor_index] = analogRead(Pressure_sensor); //https://planetcalc.com/8110/

  // Calculate the average of the sensor values
  average_sensor_value = 0.0;
  for (int i = 0; i < AVERAGING_COUNT; i++) {
    average_sensor_value += sensor_values[i];
  }
  average_sensor_value /= AVERAGING_COUNT;

  // Update the sensor index for the next sample
  sensor_index = (sensor_index + 1) % AVERAGING_COUNT;

  // Read the voltage values
  voltage_3V = analogRead(volt_3) / 1240.909;
  voltage_5V = analogRead(volt_5) * 0.003990188062142273;
  voltage_12V = analogRead(volt_12) * 0.0035100845803513335 + 0.44672088484059813;

  // Calculate the pressure in bar
  Pressure_Bar = 0.001516155758077879 * average_sensor_value - 2.019519469759735;

  boost_lvl_print = Pressure_Bar;

  // Control the PWM output based on the pressure
  if (Pressure_Bar <= WGspring_pressure) {
    ledcWrite(MACChannel, 0);
    Output = 0;
  } else {
    myPID.Compute();
    ledcWrite(MACChannel, Output);
  }
}
```

Listing 6. Sensor input calculations

3.2 Electronics

Most of the elements required were already present on smaller printed circuit boards (PCBs). The initial step involved linking these components, which were initially carried out on a breadboard. After ensuring the successful functioning of all parts, everything was assembled using a perf board. The connection was established using jumper wires and various other components. Though, this first version lacked compatibility with the Controller Area Network (CAN) Bus. Therefore, it was decided to create another prototype that would incorporate the CAN Bus functionality and a light sensor. Pair of round PCBs were designed (Figure 24) (Some headers already soldered) to connect all the components. These PCBs were ordered from a website called JLCPCB.com. The PCBs came unassembled and were built after arrival. Data logging was initiated, and all the data was collected for later review.

3.2.1 Schematics

The online and freely accessible PCB design tool called Easy EDA was used to create the schematic [21.]. The user interface of this online tool is quite similar to other PCB design softwares that are available. As I am familiar with various PCB Design tools, I found this online tool to be easy to use and navigate. Additionally, the saved data is conveniently stored in a cloud, making it accessible from various devices. This feature was particularly beneficial as I often switch between different computers while working on different projects. Within the tool, numerous components were readily available for immediate use, although a few were absent. Fortunately, custom pinouts and footprints for those missing components were easily created. EasyEDA was seamlessly integrated with the PCB printing website, facilitating the direct placement of orders for the PCBs without any issues.

In the main schematic (Figure 21), various components are present. The R1 120 Ohm resistor is for the CAN Bus Network to terminate the communication bus. R6 is associated with the light sensor to enable the lowering of the screen and LEDs brightness in low-light conditions. Also, the main schematic includes four resistors (R2, R3, R4, R5) that are utilized for voltage measurements for later debugging purposes.

As they are used for measuring voltages of 5V and 14.2V (The car's operating voltage). These resistors function as voltage dividers, allowing the voltage to be scaled down to levels between 0V and 3.3V. As the microcontroller's analog inputs maximum voltage is 3.3 volts and that corresponds to a 4095 digital reading.

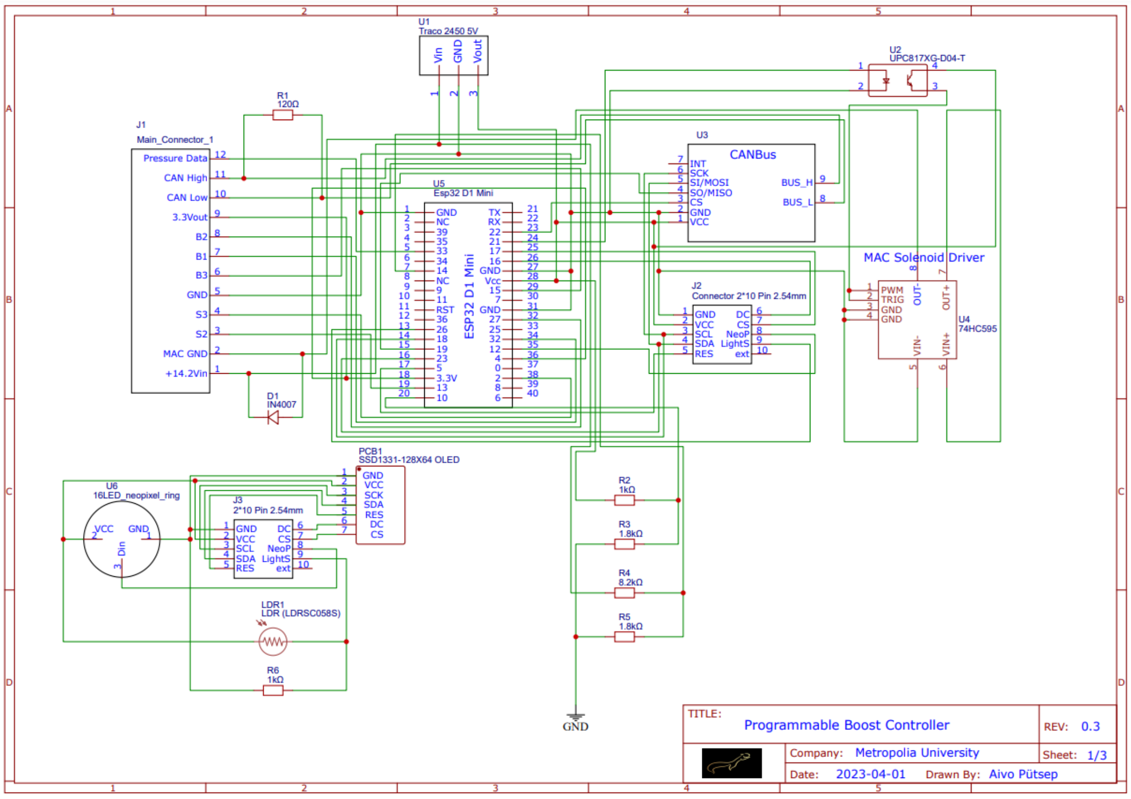


Figure 21. Main Schematic

A flyback diode was used in parallel with the solenoid, it provides a path for the inductive current to circulate when the power is switched off. The diode allows the energy stored in the magnetic field to dissipate gradually and prevents voltage spikes from occurring. This protection ensures the smooth operation of the solenoid and protects other components in the circuit.

And for added safety, it was decided to incorporate an optocoupler into the circuit, which provides electrical isolation between the MOSFETs and the microcontroller output pin. By using this component, noise can be further reduced and gain protection against high voltages or transients can be implemented.

In the initial first version of the prototype, the optocoupler was not included, and the circuit still functioned. Yet now that it has been implemented in the second prototype, the effects can be observed in real-life, and the benefits of using the optocoupler in terms of improved noise reduction and enhanced protection against voltage spikes or transients can be assessed.

The theoretical voltage spike or swing in a 12V DC car system can vary depending on the specific circumstances and the nature of the transient event. Voltage spikes or transients in automotive systems can typically range from a few volts to several tens of volts.
Transient voltage spikes can be caused by inductive kickback from ignition systems, alternator field collapse, or sudden changes in electrical loads such as switching on or off high-powered devices. These spikes can potentially damage sensitive electronic components if they are not properly protected.

The final component is a voltage regulator called the Traco TSR-1. [22, p. 2] Its purpose is to convert the input voltage of 14.2V DC to a stable 5V voltage. Initially, an LM7805 was considered, but it became hot even without reaching the maximum load. Upon calculating the current draw of the components, it was realized that especially when the Neopixels would be at full brightness and displaying white, the current draw approaches the 1A threshold where the LM35 would require a heatsink. Given the limited available space, it was decided to use the Traco TSR-1 voltage regulator instead. It is far more efficient and does not require additional external components such as capacitors.

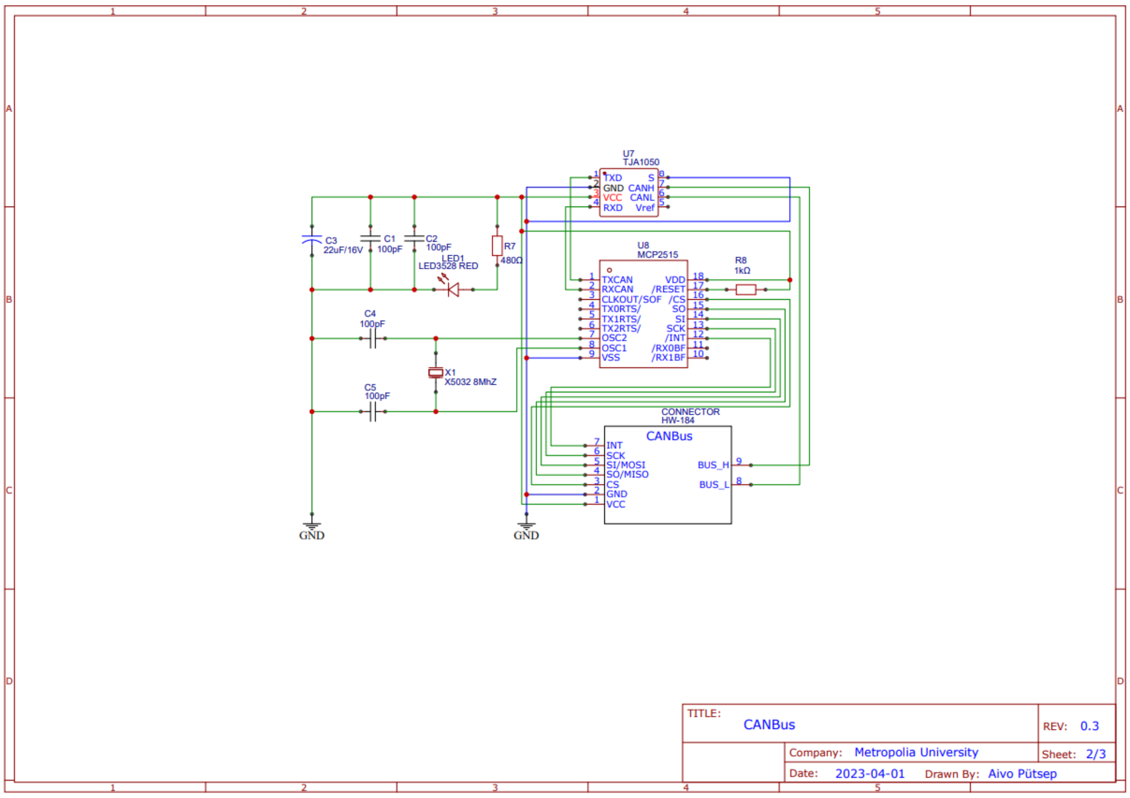


Figure 22. CAN Bus schematic

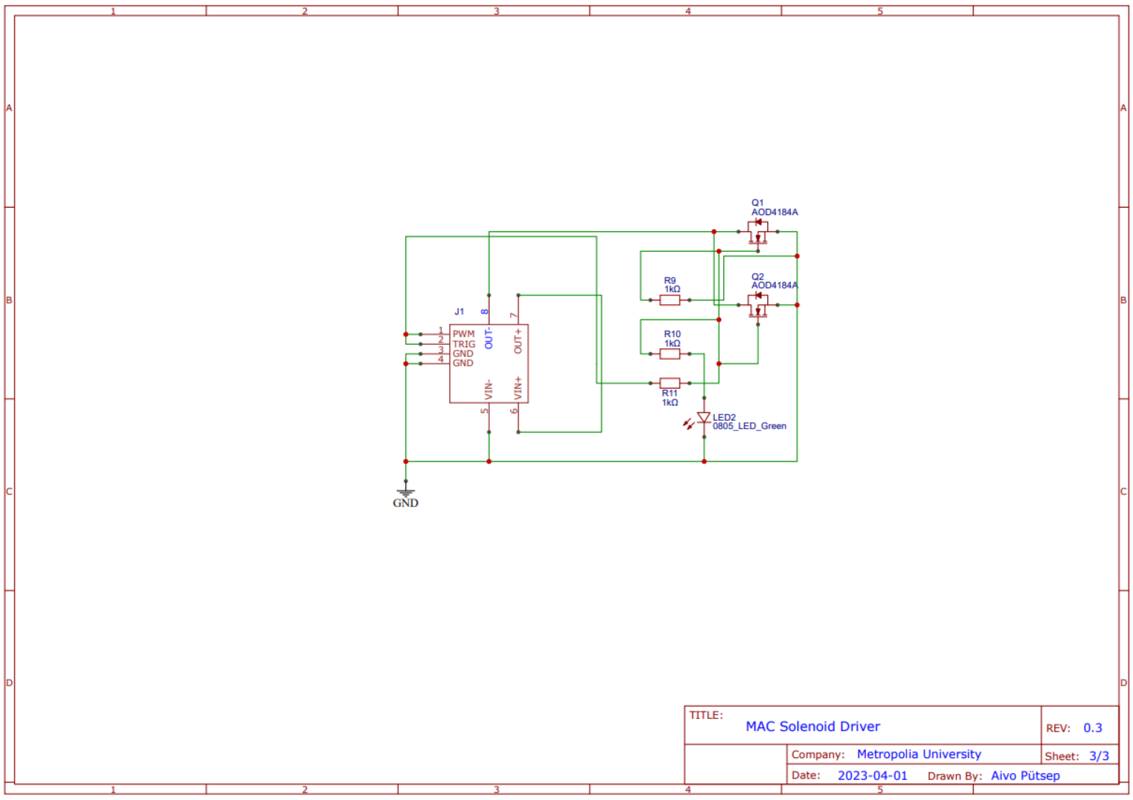


Figure 23. MAC Solenoid Driver schematic

3.2.2 Assembling on a Breadboard

The assembling of the breadboard was mostly straightforward, the initial version was actually done with another ESP32 development board called an ESP32 DEVKIT v1. Later when the smaller development board was received, it was switched to that one, it worked almost the same, only some GPIOs needed to be rearranged. The breadboard connections can be seen in Figure 24. In this picture, the light sensor and the CAN Bus PCB are not connected.
Several components can be seen, including the microcontroller, MOSFET board, two buttons, and a screen surrounded by Neopixels.

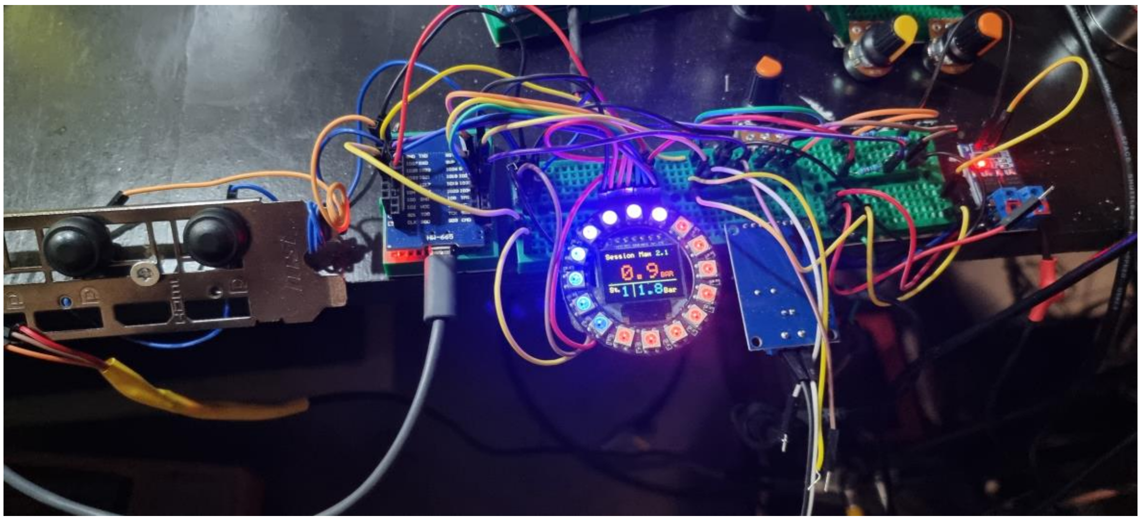


Figure 24. Most of the components connected to a breadboard

3.2.3 PCB

The printed circuit board (PCB) was created with the online PCB design tool in EasyEDA [21.] and it was ordered through a webpage (Figure 25) called JLCPCB.

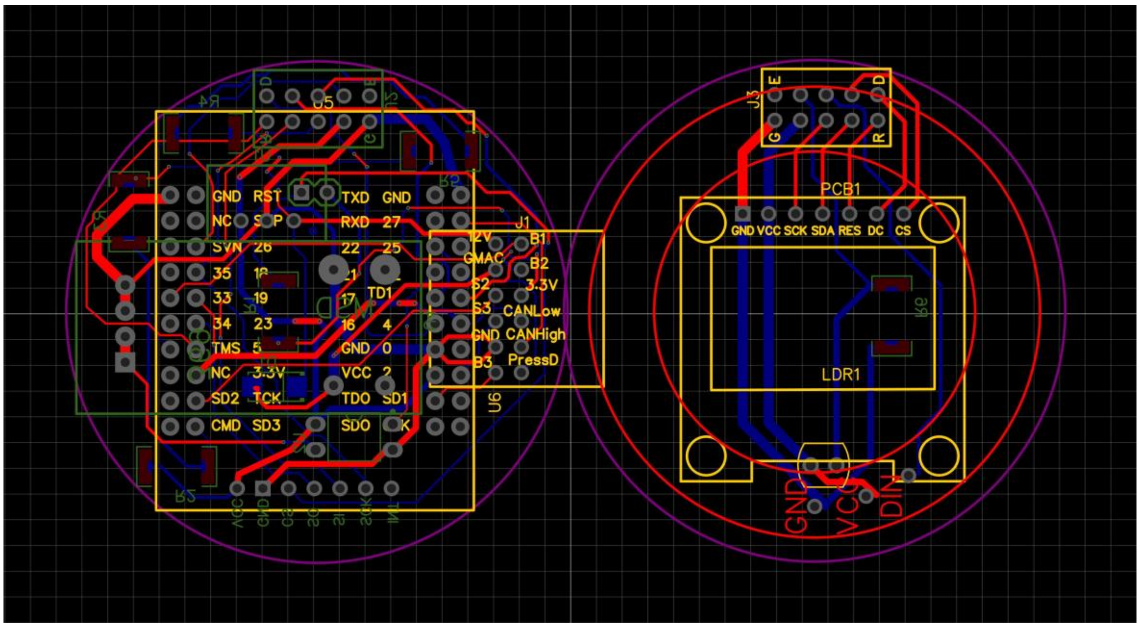


Figure 25. The overview of the PCB Design in EasyEda

After making the order it took some time and the PCBs arrived by post (Figure 26). The quality of those boards was particularly good, and an order will be placed there again in the future.

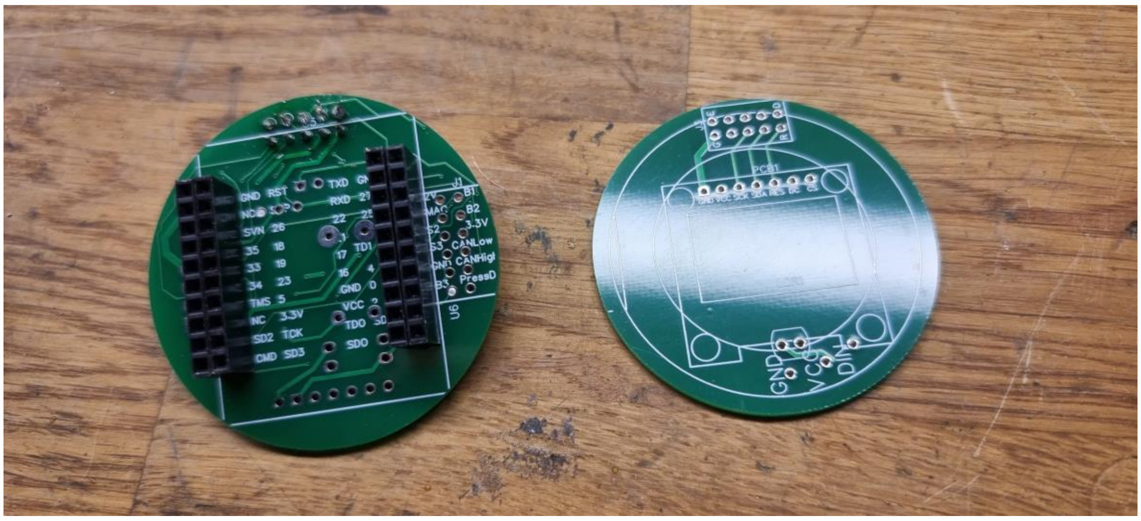


Figure 26. PCBs that were received from JLCPCB.com

3.3 Case design

3.3.1 3D Modelling and Printing

To ensure that all components were housed securely a case was required for the project (Figure 27 and Figure 28). The goal was to create a case that the front of the device would also look decent in the instrument cluster and its appearance would not make it feel out of place. The first prototype was mainly soldered together using small wires and a perf board connecting all the components. This meant that the first case dimensions had to be bigger than using a regular PCB design.

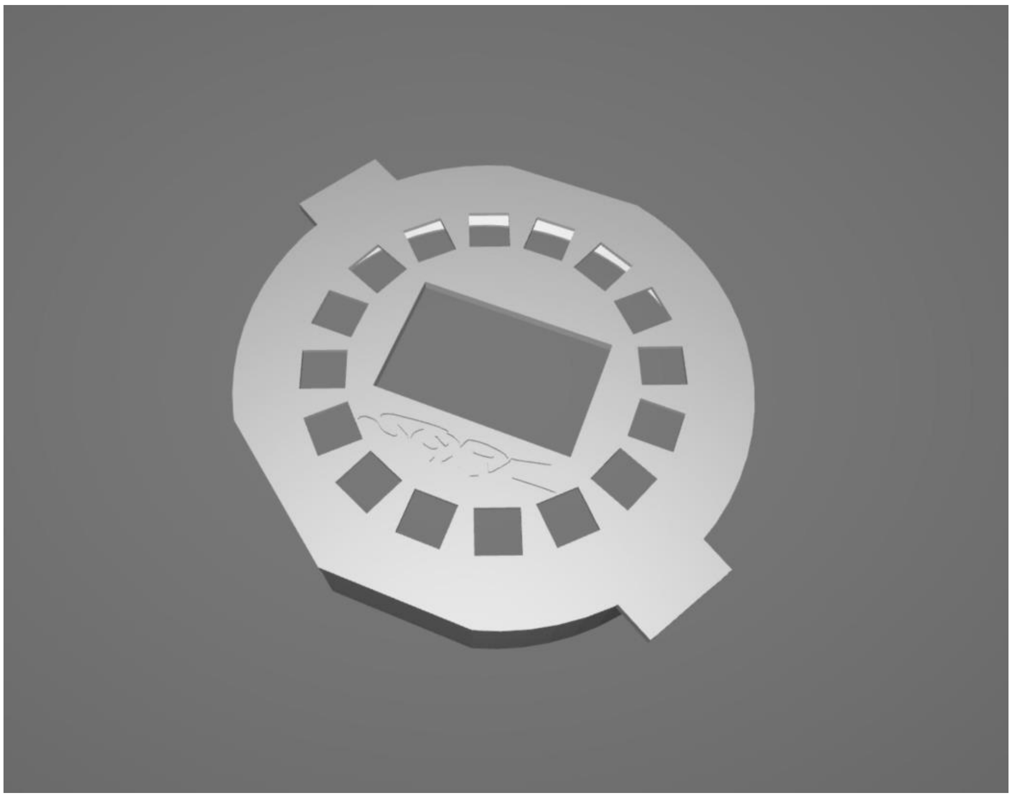


Figure 27. The 3D model of the front face of the boost controller

The initial case prototype was created using CAD software and printed using a plastic called PLA, with has a maximum temperature threshold of 60-65°C before it begins to soften and deform. In future iterations, as the device is mounted to a car, witch cabin temperature in summer can get past those thresholds, the case should be printed with ABS or PETG to provide higher temperature resistance.

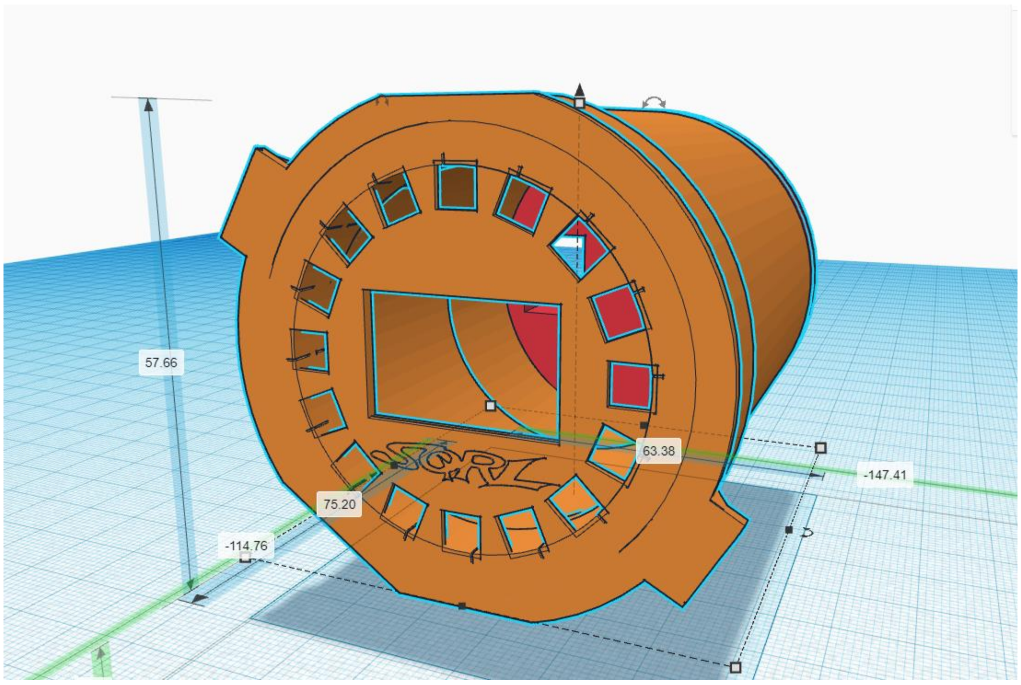


Figure 28. Another view of the casing with dimensions in mm.

3.4 Assembling the first prototype

Like said above, the first prototype was connected using a perf board and wires (Figure 29). Here a Traco TSR1 and a BD907 power transistor can be seen. The DB907 eventually was not used.

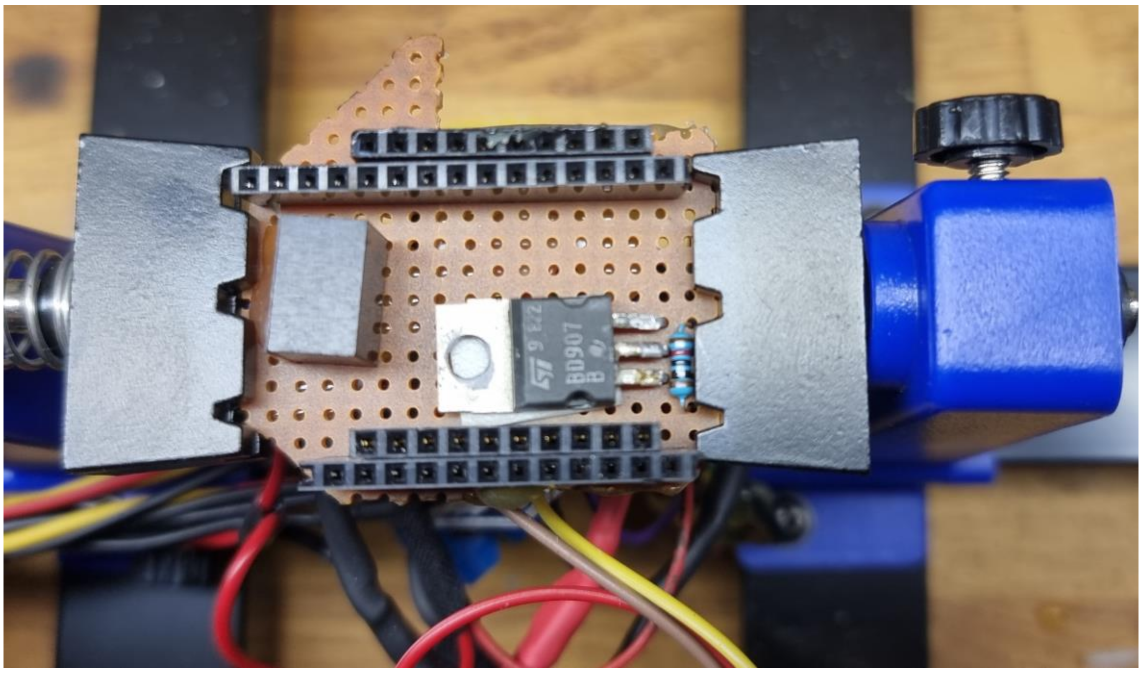


Figure 29. The first prototype with some components soldered

For the microcontroller and the screen connectors 2.45mm headers were used to easily switch components when needed (Figure 29 and Figure 30), connecting everything like that took some time but, in the end, the first working was managed to be produced, was later installed on a car (Figure 31).

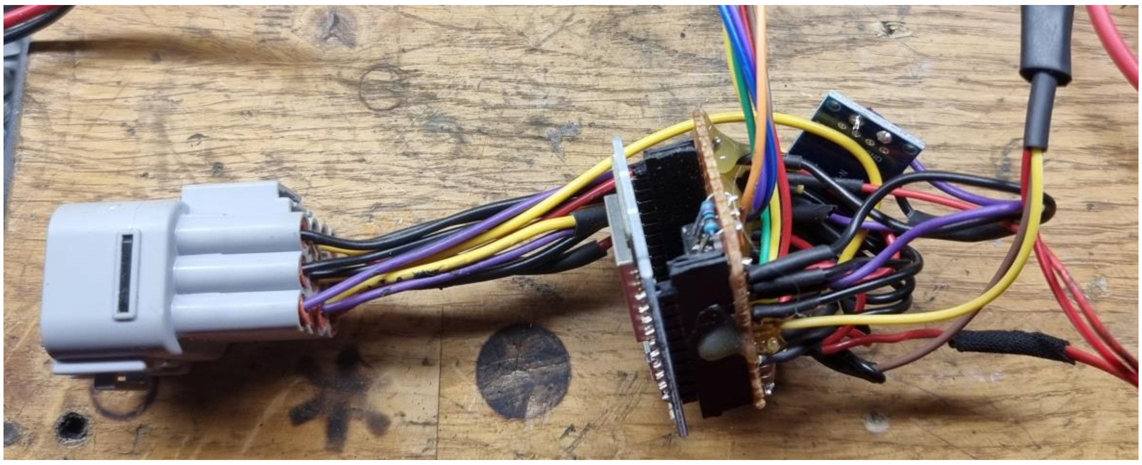


Figure 30. The first prototypes side-view

After everything was soldered together the components were installed inside of the casing.

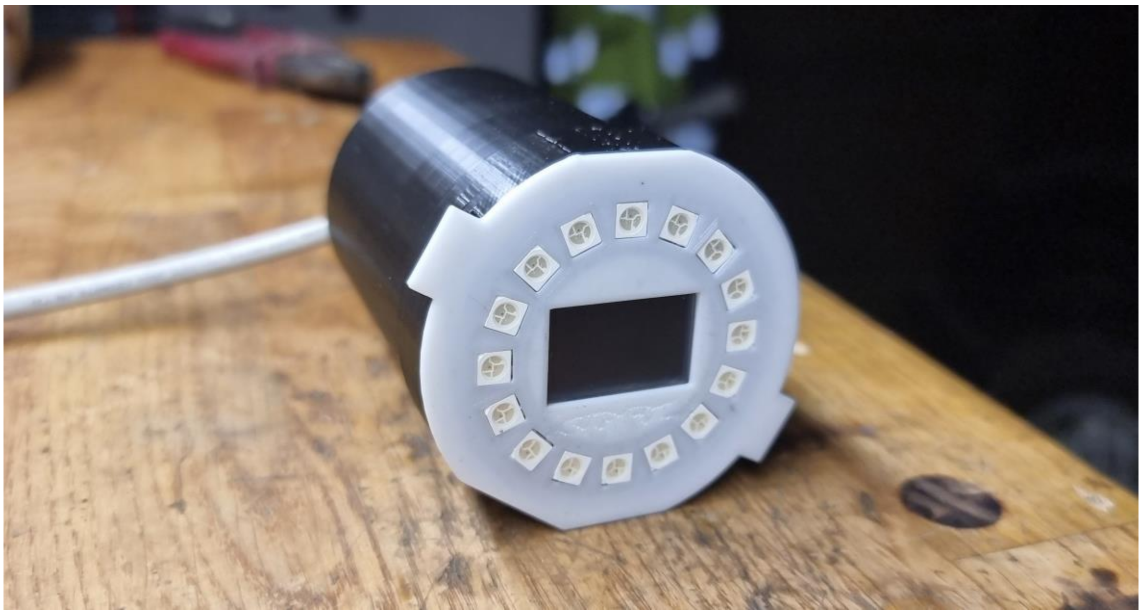


Figure 31. Printed casing with the components installed.

3.5 Assembling the second prototype

As the first prototype was a little bigger in its dimensions and lacked the optocoupler, light sensor and CAN Bus Feature it was decided that a second prototype should be made. The assembled device can be seen in Figures 32, 33, and 34.

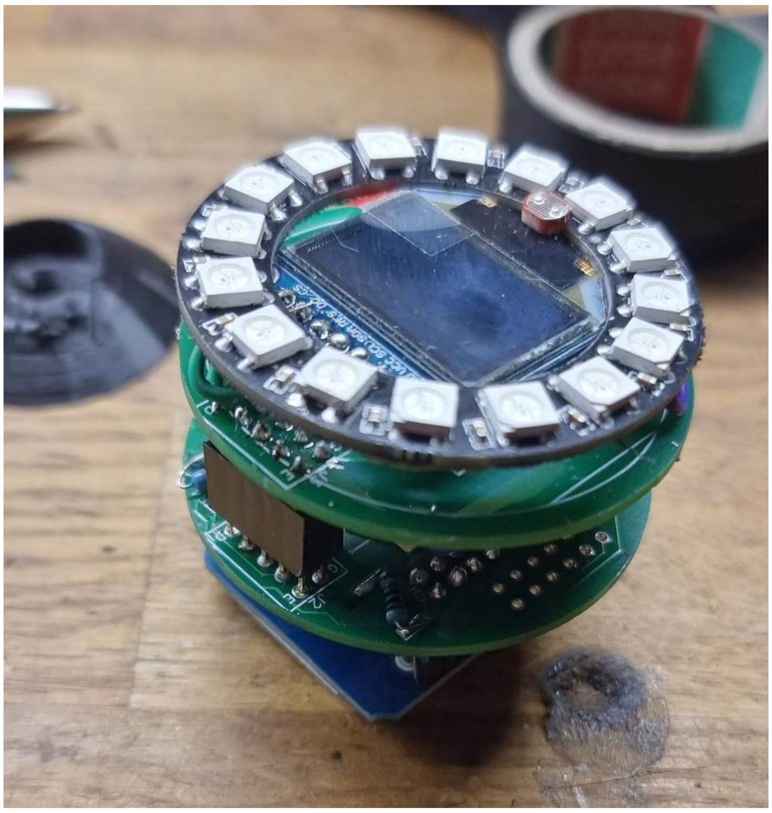


Figure 32. Top view of the second prototype

Regular 2.54mm female/male headers were again utilized to connect the two PCBs and the microcontroller, enabling an easy swap when required later. In this prototype, the wrong footprints for the resistors were mistakenly used, but it did not matter significantly because regular thru-hole resistors could now be used and soldered directly to the pads.

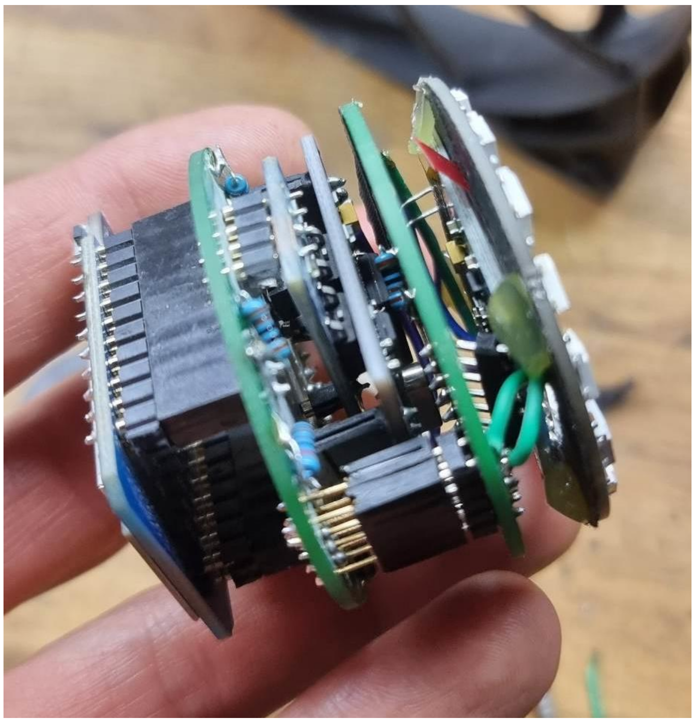


Figure 33. Side view of the second prototype

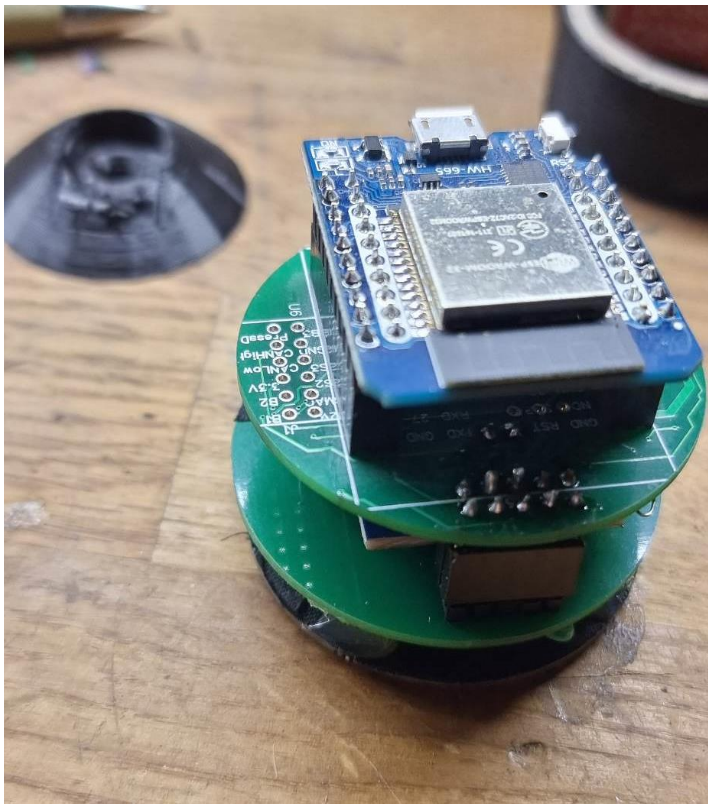


Figure 34. Bottom view of the second prototype

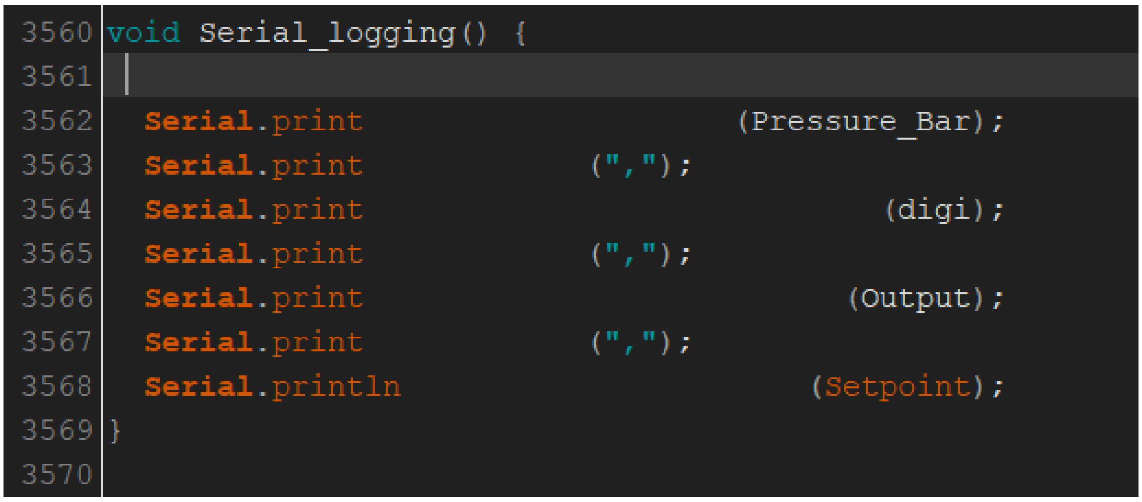
4 Measurements and Testing

The testing was conducted using two different approaches. Initially, bench testing was carried out to assess the performance of the system under controlled conditions. Subsequently, the system was tested in the actual vehicle it was intended for. In this section, the two testing approaches will be discussed.

For data acquisition Excel Data Streamer plugin, Arduino IDE Serial Plotter and an Oscilloscope was used. For bench testing the Oscilloscope was used to see what the controller is doing, how does it modulate the PWM signal and is everything calibrated correctly.

In the code a separate serial output function was written to have four channel output, CH1 for Boost pressure, CH2 for Analog output values, CH3 for PWM and CH4 was set to Setpoint. The code itself is straight forward, worked the same way when wanting to use the Arduino IDE Serial plotter, the values only needed to be separated by commas and they should be written in one line as it can be seen from Listing 7.

After some experimentation, it was found that Excel worked best with a baud rate of 9600 bits/s. Higher bitrates gave some unusable readings.



```
void Serial_logging() {

  Serial.print                    (Pressure_Bar);
  Serial.print            (",");
  Serial.print                            (digi);
  Serial.print            (",");
  Serial.print                          (Output);
  Serial.print            (",");
  Serial.println                      (Setpoint);
}

```

Listing 7. Serial output code for data logging with Excel

4.1 Testing on a Bench

The testing bench included a compressor for imitating the turbocharger, the Microcontroller and other components were assembled to a breadboard, a pressure sensor and a MAC solenoid were added. And for good measure, also an Analog pressure sensor was added Figure 35.

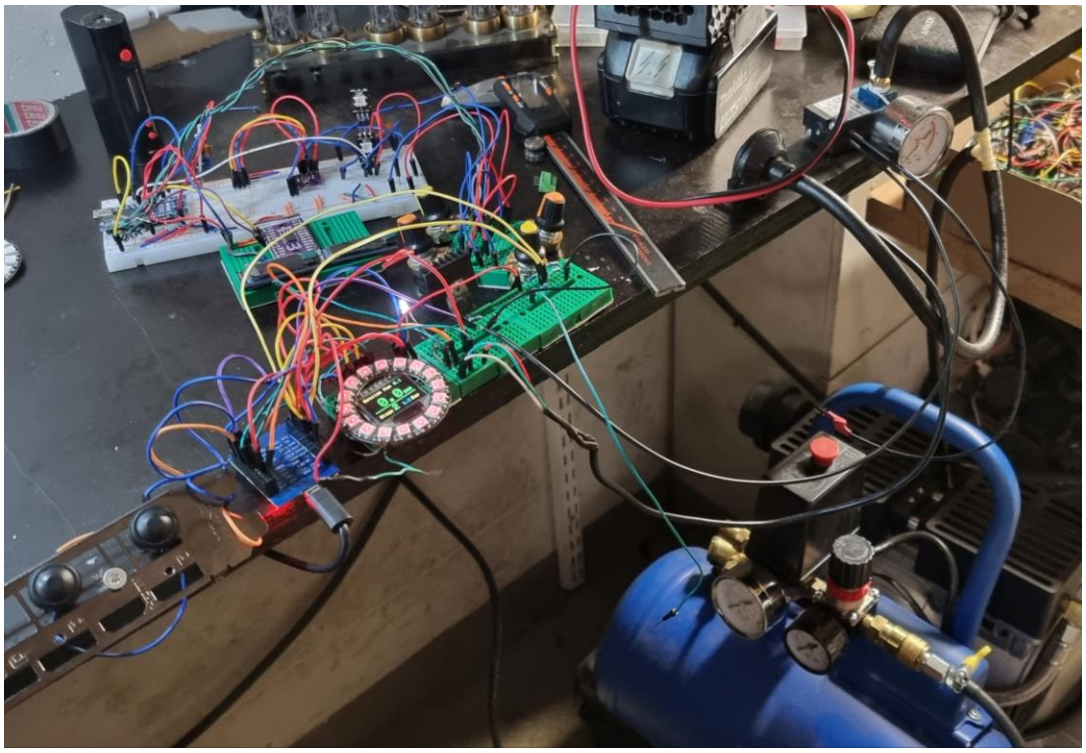
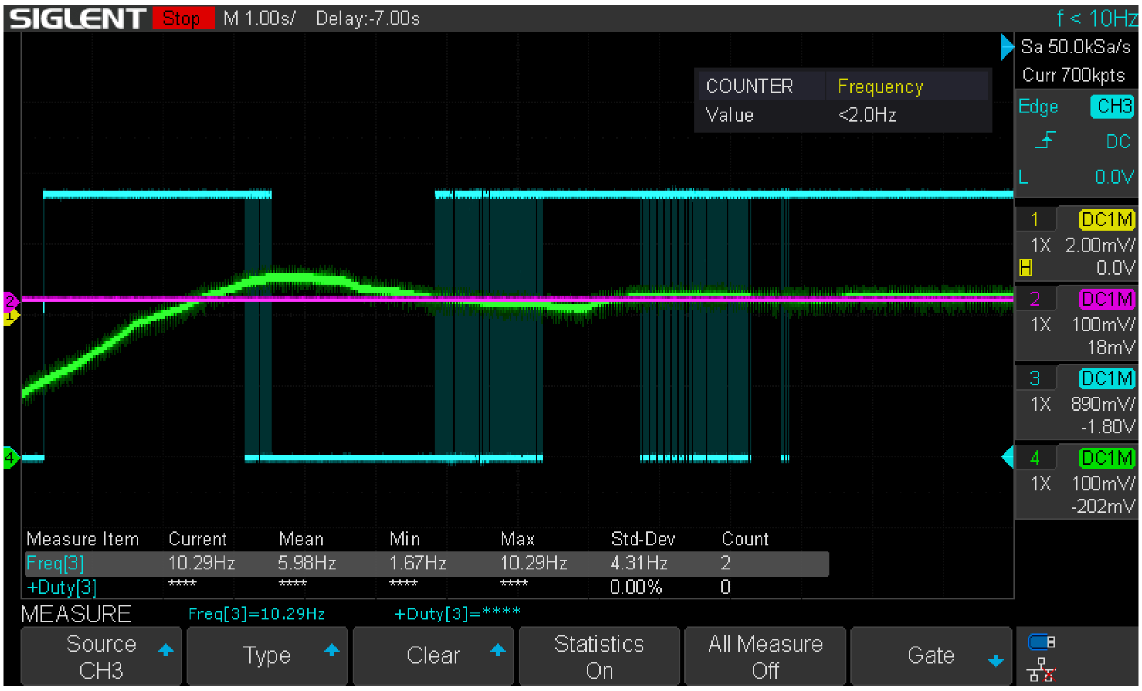


Figure 35. An overview of the bench testing setup

Upon testing, it became obvious rather quickly that in order to properly test the system, a wastegate valve would be needed, similar to what a car has. As a workaround, to manually test the output PWM the pressure sensor was replaced with a potentiometer to simulate the boost pressure. This allowed to observe the PWM signal and get a rough idea of how the PID controller was functioning. But this approach had limitations. For instance, the pressure could not be released quickly enough to accurately determine the extent of the pressure overshoot and how the PID controller was functioning to minimize it. Despite these limitations, the test would still be considered successful as new data were able to be gathered and it was confirmed that the PID was functioning and sending a proper signal to the MAC solenoid through the MOSFET. In order to monitor the system, an oscilloscope probe was connected to the MAC solenoid's output pin, another probe was connected to the pressure sensor, and an additional line was set up to monitor the setpoint pressure. It can be seen in Figure 36. As the Purple line represents the setpoint, the green line represents the intake pressure, and the cyan line represents us the duty cycle. From the graph, when the pressure reaches 0.6 bar the duty cycle goes to 100%, as it should, and when it reaches the setpoint it tries to counteract by lowering the PWM and eventually going to 0% PWM signal, after with the pressure decreases and the PWM signal increases again till it stabilizes the boost pressure.



```
COUNTER   Frequency
Value     <2.0Hz
```

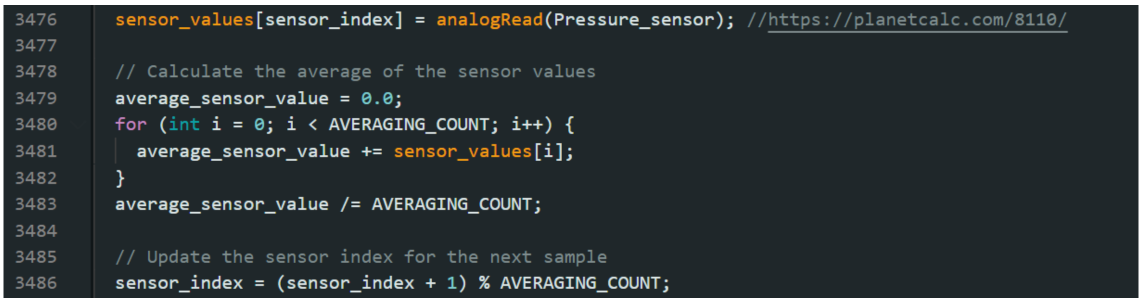
Measure Item	Current	Mean	Min	Max	Std-Dev	Count
Freq[3]	10.29Hz	5.98Hz	1.67Hz	10.29Hz	4.31Hz	2
+Duty[3]	****	****	****	****	0.00%	0

```
Freq[3]=10.29Hz    +Duty[3]=****
```

Figure 36. Oscilloscope output

When the Arduino serial plotter output was observed, it was noticed that the pressure that the controller was reading was fluctuating more than expected, indicating potential noise issues. It appeared that improved grounding might have been beneficial for removing unnecessary noise from the analog input. The primary issue was that the analog reading itself was fluctuating, which implied a need for the implementation of an averaging function to the sensor's input pin would be beneficial.

Taking all of those results into consideration an averaging function was added to the code in Listing 8.



```
sensor_values[sensor_index] = analogRead(Pressure_sensor); //https://planetcalc.com/8110/

// Calculate the average of the sensor values
average_sensor_value = 0.0;
for (int i = 0; i < AVERAGING_COUNT; i++) {
  average_sensor_value += sensor_values[i];
}
average_sensor_value /= AVERAGING_COUNT;

// Update the sensor index for the next sample
sensor_index = (sensor_index + 1) % AVERAGING_COUNT;
```

Listing 8: Part of the code where the averaging is done to minimize the input oscillations.

This part of the code creates an array to hold the last 6 sensor values and then it calculates the average of those 6 values before the pressure value is calculated. The sensor_index variable monitors the current position in the array, and the % operator is used to go back in fto the beginning of the array once the end is reached.

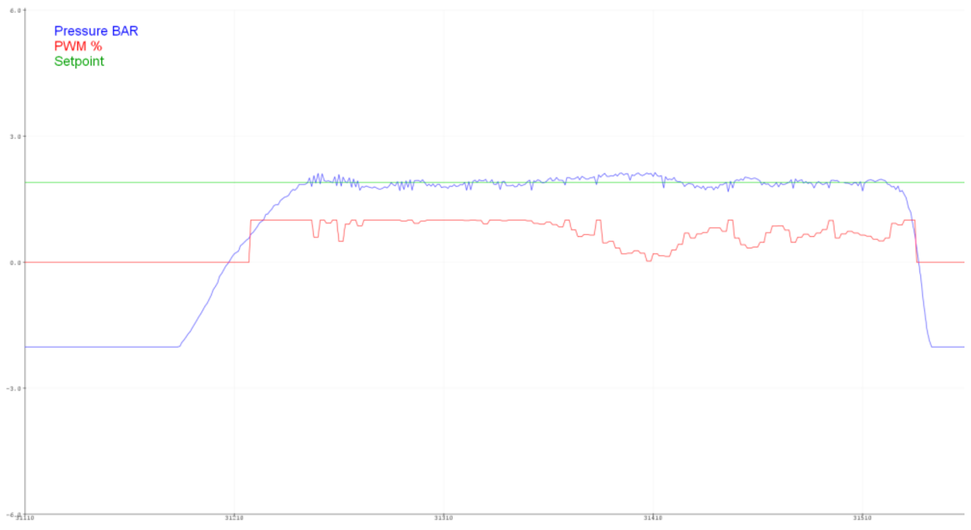


Figure 37, Data before averaging function

In Figure 37, the input without the averaging function can be observed, and in Figure 38, the implemented averaging is displayed. It is noticeable that the pressure line is much smoother, and on the device’s screen, the numbers no longer fluctuate. Adding this averaging function both in acquiring and displaying the data proved to be a good iteration.

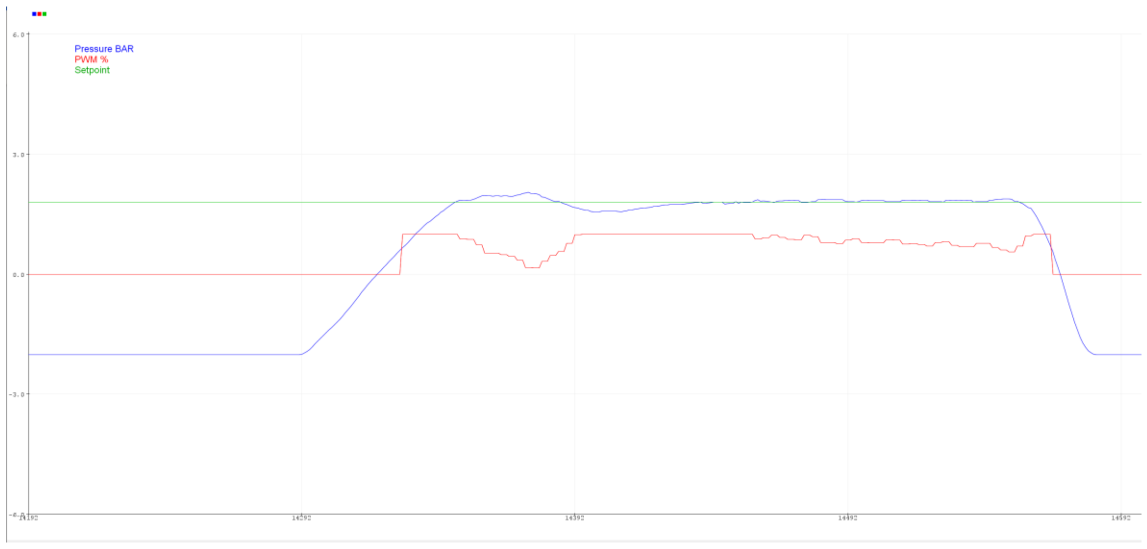


Figure 38, Data after averaging function

4.2 Testing on a car

In order to test the Boost Controller on a car, it was first necessary to hardwire the boost controller’s connector to the car. Some wires had to be connected to the car's wiring harness, while others were connected to the engine control unit. We needed to have +12V when the ignition is turned on, Ground, Output wires for the MAC solenoid, stage wires for the ECU, CAN Bus wires for the ECU which were twisted to eliminate any interference, pressure sensor wires and the buttons were to be connected.

In Figure 39 a quick sketch can be seen that was made for connecting the pins to a connector to have a fast and a clear overview on what pin is what. In Figure 40 the connector itself can be seen, the male connector part is connected to the car, and the female connector part is connected to the boost controller.

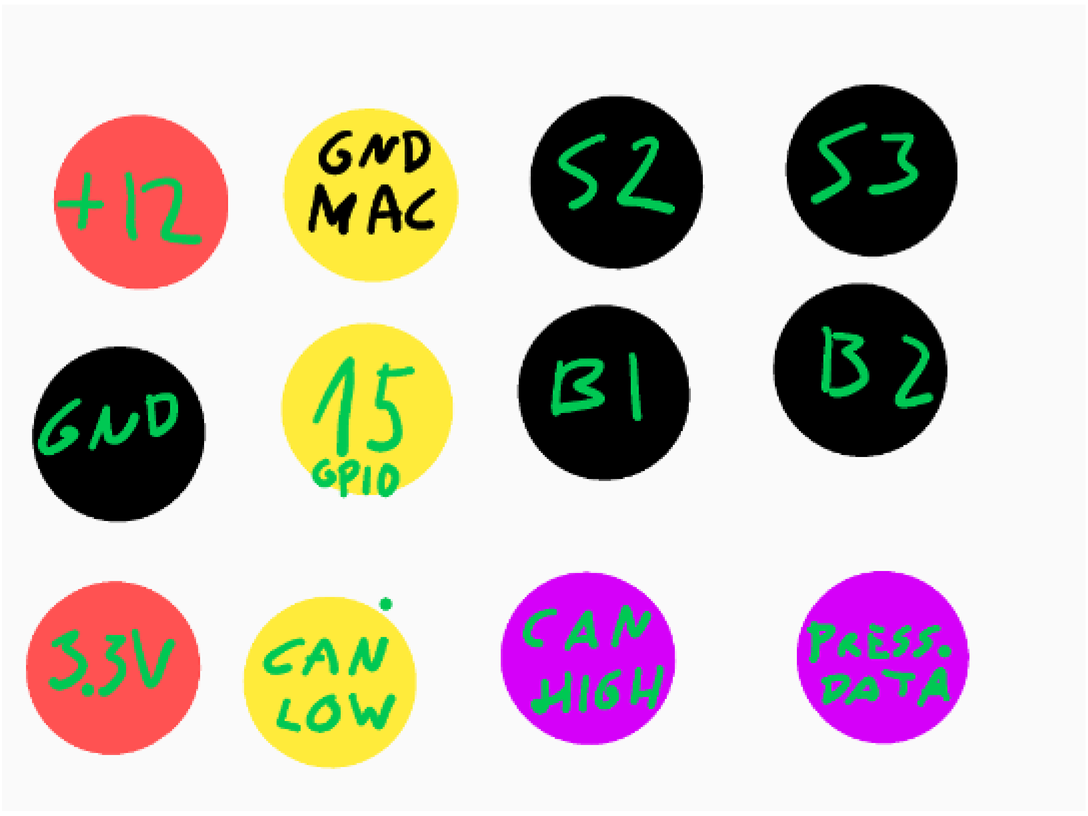


Figure 39. Boost controller’s connector pinout with wire colors (front view)

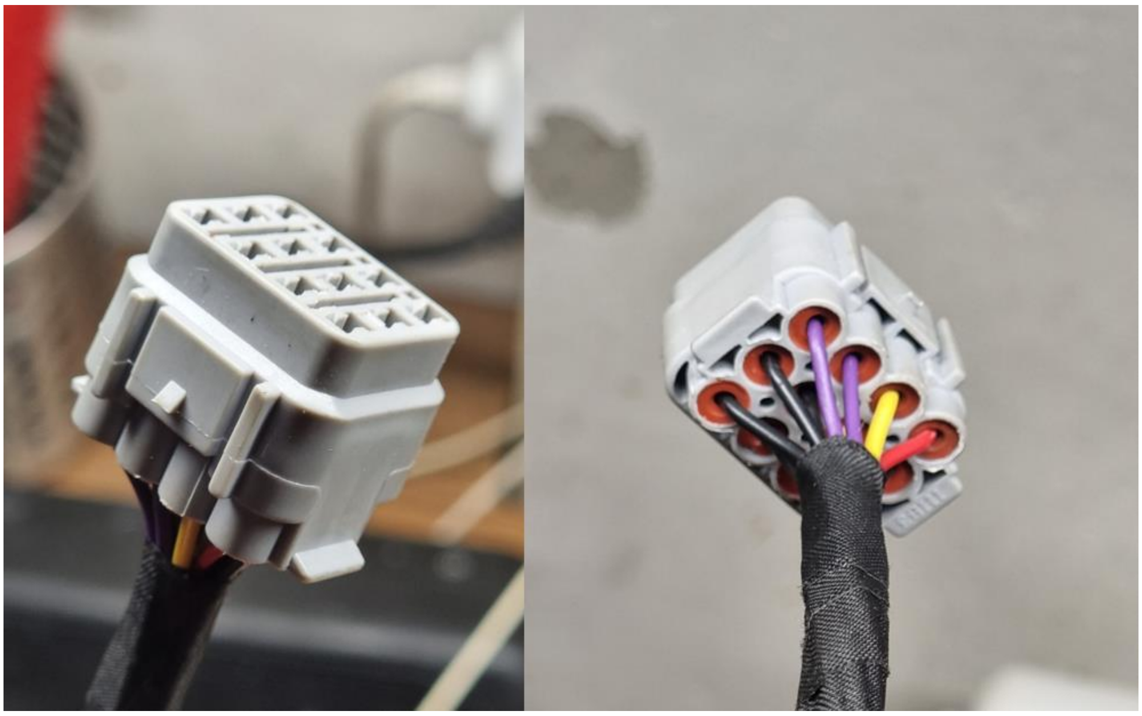


Figure 40. This is the connector that was used in the project

In Figure 41 and Figure 42 we can see the instrument cluster where the controller was mounted, the boost controller replaced the Clock that was mounted in there before.

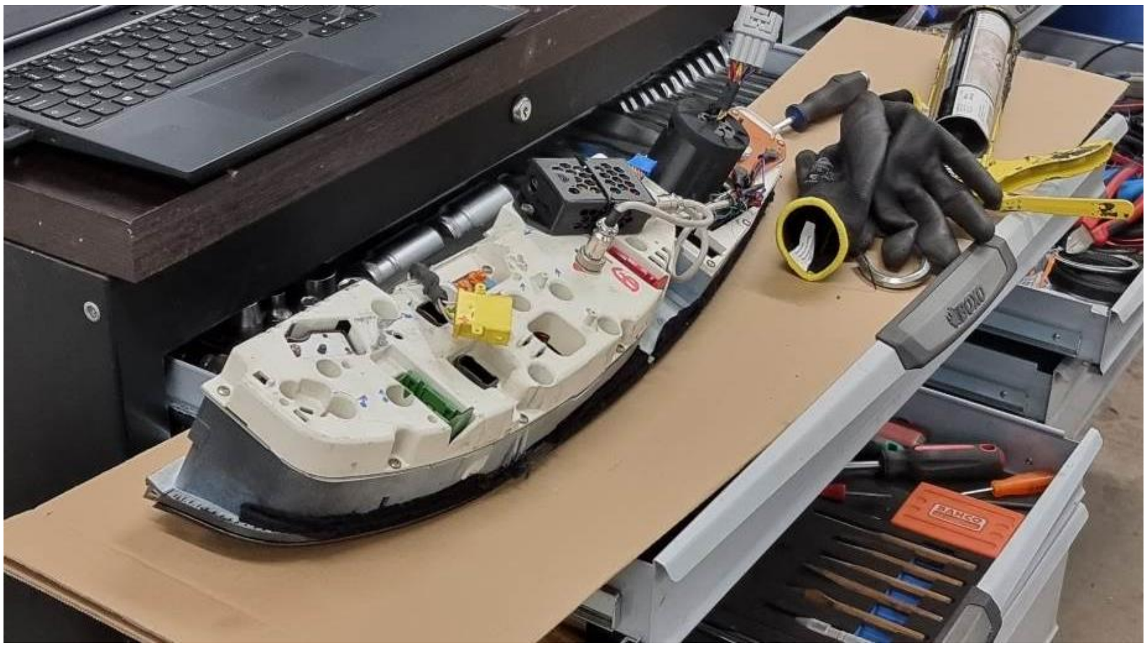


Figure 41. The cars instrument cluster the first prototype was installed

The boost controller was a direct fit as precise measurements were taken before.

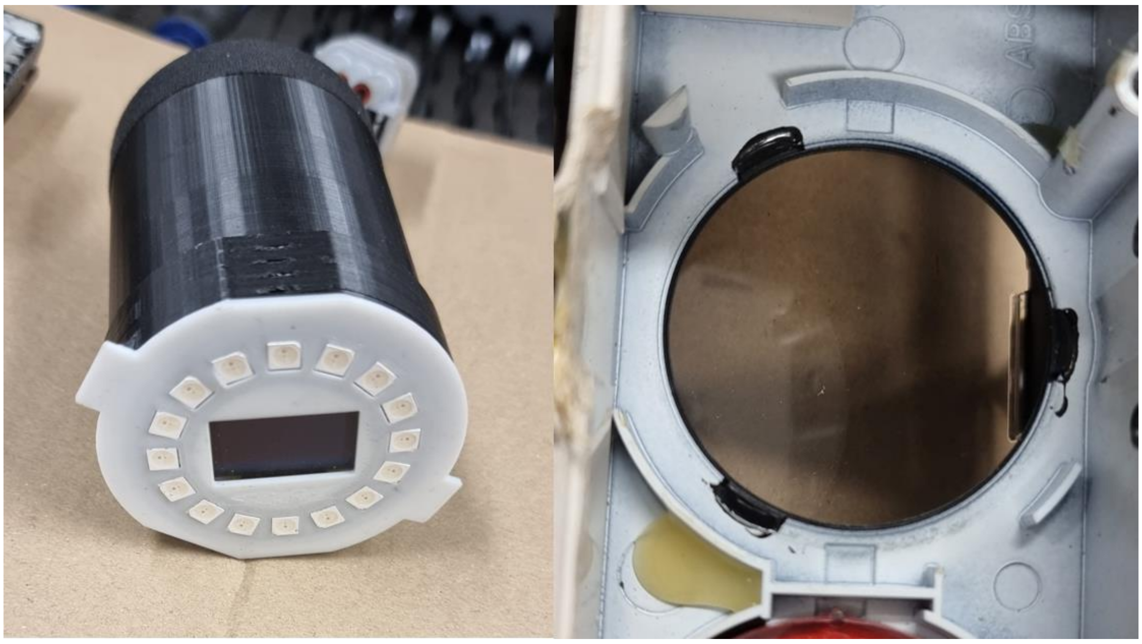


Figure 42. The boost controller and the space in the instrument cluster it was mounted

Three buttons, SELECT, DOWN, and a third one, UP, were added for future iterations. The wiper Lever had Three buttons so why not use all of them, initially the boost controller was coded to have two buttons.

All the necessary wires were located behind the instrument cluster wiring harness, making the connection straightforward.

Once the device was installed in a car, the real testing began. A functioning turbocharger with observable boost levels was at our disposal, which could now, hopefully, be altered on the fly using the Boost Controller.

In the image displayed below (Figure 43), we can observe the operation of the controller in a car. The duty cycle is depicted by the grey line in the graph. However, difficulty was encountered in representing it as "HIGH" and "LOW" as duty cycle in 30Hz. Instead, it was presented as a percentage scale, where 100% equals 1 and 50% equals 0.5 and 0% equals 0. This approach allows for the convenient observation of our graph. The orange line represents the setpoint (Stage) of the boost pressure. And lastly, the blue line represents the Intake boost pressure.

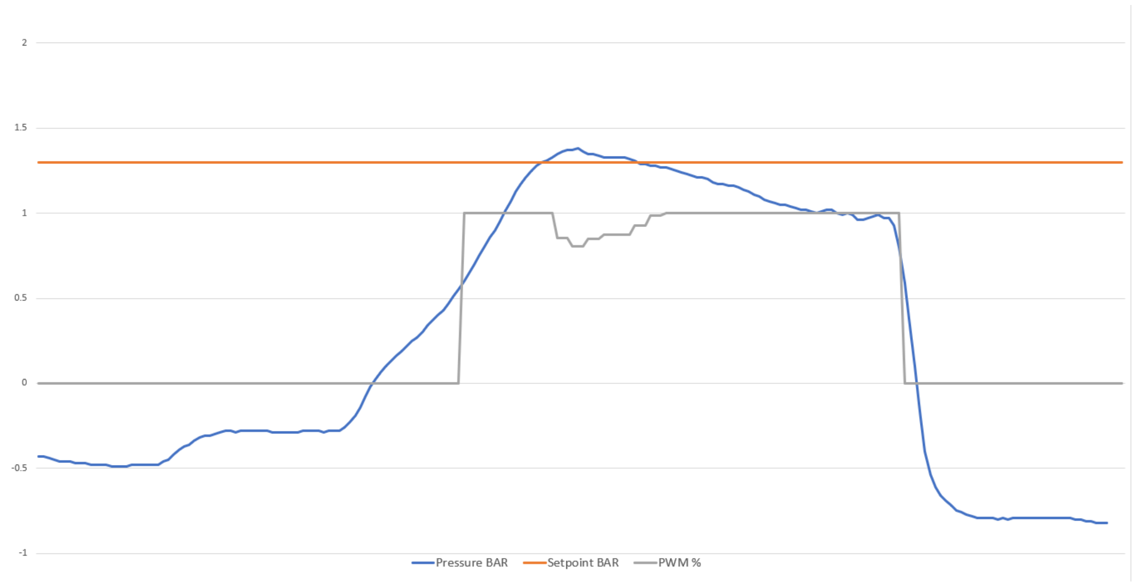


Figure 43. The graph we got from the boost controller.

As evident from the graph, when the pressure is low, the duty cycle is at 0%. After it reaches 0.6 bar the PWM goes to 100% as it should. As the pressure approaches the setpoint, the duty cycle should gradually decrease eventually stabilizing at the desired pressure level. But we did observe a noticeable overshoot and after that, the boost pressure gradually decreased much more than it should. It was only when the pressure hit 1 bar it began to achieve stability. This outcome was undesirable, indicating a need for further tuning to the PID controller.

The code has been prepared for changing the PID values directly from the menu, but this feature is intended for future iterations. So, the PID values were changed in the code, as it allowed us to tune how the boost controller was controlling the boost pressure.

PID Tuning:

To get the best performance out of the PID controller, tuning was needed. The tuning process began by increasing the Proportional and Integral values (Using the same values) until the output became slightly unstable. To counteract the unstable behavior, the Derivative value was increased until steadiness was more or less achieved. After that, fine-tuning of those values started, mostly by reducing the P value and increasing the I. The overall goal was to use as high values as possible while still having a stable response. Then all values were decreased to have a safety margin to prevent oscillation and overshoot.

Following the PID tuning, a much better outcome was achieved, which can be observed in Figure 44. The Blue line represents the Boost pressure as the red line is the PWM signal in percentage and the green line is the setpoint.

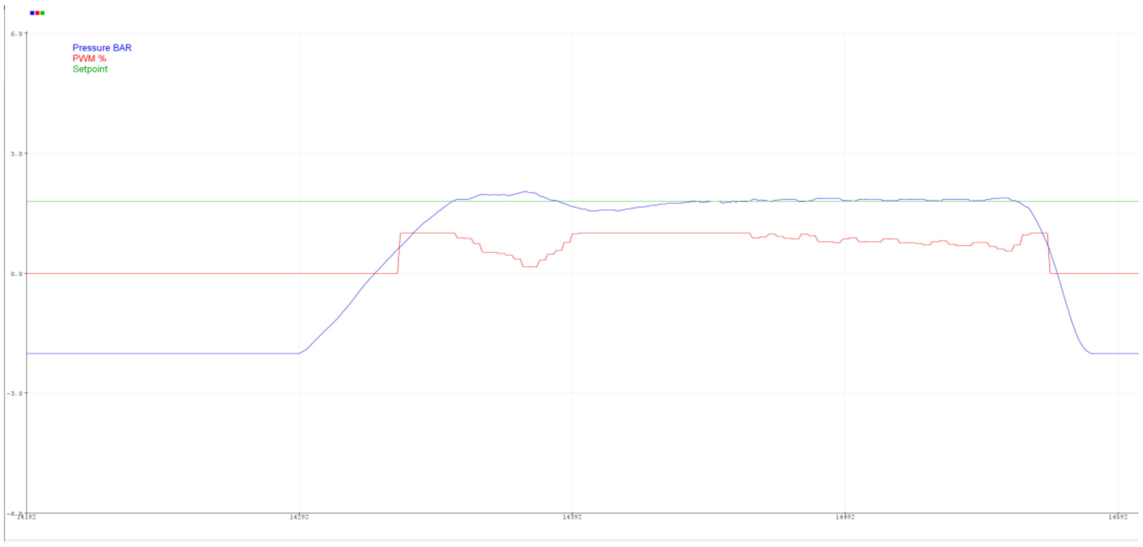


Figure 44. Pressure control after the PID Adjustments

5 Results and Discussion

5.1 Results

First, the results of the first prototype are going to be discussed.
The results were surprisingly good, even with the first PID values the Controller did a semi-decent job on controlling the boost levels. The interface worked right away. The Controller itself was a direct fit to the instrument cluster. But we also encountered some problems and made a couple of observations. Firstly, a recalibration of the pressure sensor was still needed somehow, as the calibration was slightly off, which was amusing because it functioned on the bench. Secondly, the screen brightness was not enough when viewing the screen in direct sunlight. The LEDs brightness was too much in certain situations, especially when the desired pressure was reached the white light blinking was a little too much. On the display, some of the text was excessively small and some modifications had to be made in the code. Both the LEDs issue and the text size can be easily adjusted in the code. But, to improve the screen brightness, the best solution would be obtaining a higher-quality display.

Now, let us discuss the results of the second prototype. Sadly, these results were not as satisfactory. Upon initially connecting all the components and powering on the device, we encountered a few issues. The screen displayed a partially fragmented image, and the LEDs failed to turn on. Shortly after, the screen shut down completely. Although resetting the device allowed it to start again, the screen once again disappeared after a few seconds. Disappointment was felt as the implemented CAN Bus feature was eagerly awaited to be tested.

To troubleshoot the problem, we disassembled everything and only connected the screen to ensure there was no interference between the components. However, the outcome remained unchanged. While testing the connections, they appeared to be fine. The belief held is that when the screen was integrated into the PCB, the pin holes did not align perfectly, necessitating a slight bend. Unfortunately, during the bending process, probably too much pressure was exerted on the screen, potentially causing damage.

5.2 Discussion

For the third prototype, all the components should be incorporated into a single PCB. This would result in a much smaller footprint, consequently reducing the overall size of the device and of course, making it resemble a proper PCB.

As for the PCB I made, I missed out on including test points and a ground loop. These errors slowed down the troubleshooting process. Also, the track width on the PCB should have been bigger on certain places. Maybe even those mistakes added to the fact that the second prototype did not work correctly.

When the device was first powered on, a fragmented image was displayed on the screen, and the LED ring was not lit up. The absence of test points made it difficult to identify the exact cause of the issue promptly. It seemed that damage may have been sustained by the screen cable or the screen itself when the pins were bent to achieve a correct 90-degree angle. As said before the bending was done because the holes for the screen did not align perfectly, contributing to the problem. But this experience provided valuable insights into component placement and PCB design for upcoming iterations.

For the next PCB, we gained knowledge like:

- Component placement: Strategically placing the components to minimize signal interference and optimize thermal dissipation. And making sure later everything lines up.
- Routing: Separating high-speed signals from noisy signals to minimize interference.
- Power distribution: Plan and implement a proper power distribution network, ensuring sufficient current-carrying capacity and low impedance paths for a stable voltage supply.
- Grounding: Implement a solid ground plane and connect it effectively to avoid ground loops and minimize noise
- Design for testability: Include test points and access to critical signals for easy debugging and testing during the prototype phase.

Future Plans:

In the future, the Screen and LEDs should be changed. Datalogging should be accessible through Bluetooth. The screen should be much brighter so that it would also be visible on a sunny day. And LEDs should be small APA102s. The casing material should be ABS or PETG plastic. And the device itself should look aesthetically good.

References

1 Quattroword, Audi S2 Wastegate Frequency Valve (WGFV) – N75, 2009. [Online]. https://forums.quattroworld.com/s4s6/msgs/20987.phtml [cited May 20th, 2023]

2 S4NoMore, Bimmerpost. Thread 959674, Post #2, 2014. [Online Forum]. https://f30.bimmerpost.com/forums/showthread.php?t=959674 [cited May 20th, 2023]

3 Espressif Systems, ESP32 Datasheet, [Page 1-4], 2016. [Online]. https://www.esp32.dk/esp32_datasheet_en.pdf [cited May 20th, 2023]

4 Achim Pieters, ESP32 – Pinout, 2020. [Online]. https://www.studiopieters.nl/esp32-pinout/ [cited May 21st, 2023]

5 Maker Portal, Shop, Esp32 D1 Mini, 2021. [Online]. https://makersportal.com/shop/esp32-d1-mini-bluetoothwifi-board [cited Feb 21st, 2023]

6 Dataq Instruments, Pressure sensor, 2020, [Online Datasheet]. https://www.dataq.com/resources/pdfs/datasheets/pressure-sensor-ds.pdf [cited May 21st, 2023]

7 Dyno Racing, Pressure sensor, 2023, [Online]. https://tinyurl.com/ysrcevbv [cited May 27th, 2023]

8 Ultraperformance, MAC Boost Control Solenoid (3 Port), 2023. [Online]. https://www.ultraperformance.co.uk/mac-boost-control-solenoid-3-port [cited May 27th, 2023]

9 Circuitgeeks, Basics of Arduino PWM, 2022. [Online]. https://www.circuitgeeks.com/arduino-pwm/ [cited May 27th, 2023]

10 Elecrow, 0.95-inch RGB OLED Screen, 2023. [Online]. https://www.elecrow.com/0-95-inch-oled-module-96-64-rgb-display-7pin-oled-module-with-ssd1331-spi-port.html [cited May 27th, 2023]

11 Adafruit, Neopixels Insights, 2017. [Online]. https://learn.adafruit.com/sipping-power-with-neopixels/insights [cited May 27th, 2023]

12 Esp-Tech, Neopixel Ring 16x WS2812 5050 RGB LED, 2023. [Online]. https://www.exp-tech.de/en/modules/led-controller/5066/neopixel-ring-16-x-ws2812-5050-rgb-led [cited May 27th, 2023]

13 Vance Vandoren, Understanding PID Control and loop tuning fundamentals, 2016. [Online]. https://www.controleng.com/articles/understanding-pid-control-and-loop-tuning-fundamentals/ [cited May 27th, 2023]

14 Elecrow, High-power Mosfet module, 2023. [Online]. https://www.elecrow.com/high-power-mosfet-trigger-switch-drive-module.html [cited May 28th, 2023]

15 Alpha & Omega Semiconductor, AOD4184A, 2009. [Online Datasheet]. https://alltransistors.com/adv/pdfdatasheet_aosemi/aod4184a.pdf [cited May 28th, 2023]

16 Wilfried Voss, CAN Bus and SAE j1939 Bus Voltage, 2019. [Online]. https://copperhilltech.com/blog/can-bus-and-sae-j1939-bus-voltage/ [cited May 28th, 2023]

17 Wikipedia, CAN Bus, 2023 [Online].
https://en.wikipedia.org/wiki/CAN_bus [cited May 28th, 2023]

18 How2electronics, Arduino CAN Bus Tutorial | Interfacing MCP2515 CAN Module with Arduino, 2023. [Online].
https://how2electronics.com/interfacing-mcp2515-can-bus-module-with-arduino/ [cited May 28th, 2023]

19 Timur, How to find the parametric equation, 2021. [Online].
https://planetcalc.com/8110/ [cited May 28th, 2023]

20 Dropbox, Programmable Boost Controller code, 2023. [Online].
https://www.dropbox.com/s/l5bcdn1ly0tedaa/DPBC.txt?dl=0 [cited May 28th, 2023]

21 EasyEDA, PCB Design tool, 2023. [Online]. https://easyeda.com/editor [cited May 31st, 2023]

22 Traco Power, TSR 1, Series 1A, 2022. [Online].
https://www.tracopower.com/int/tsr1-datasheet [cited May 31st, 2023]

Appendix 1

Design of a printed circuit board

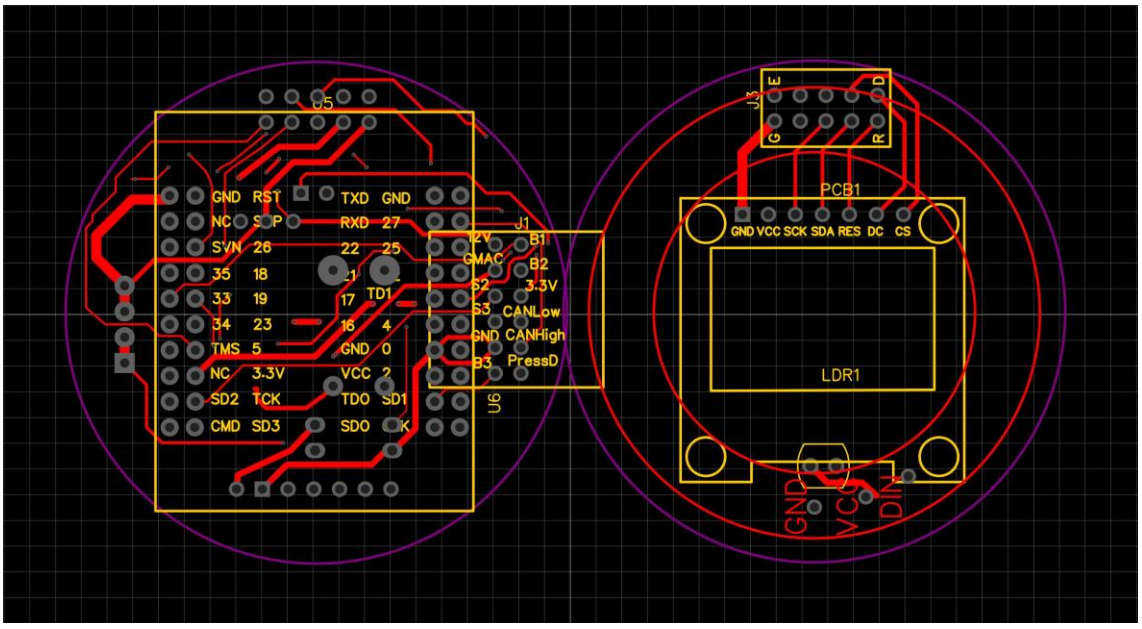


Figure 45. Top Layer

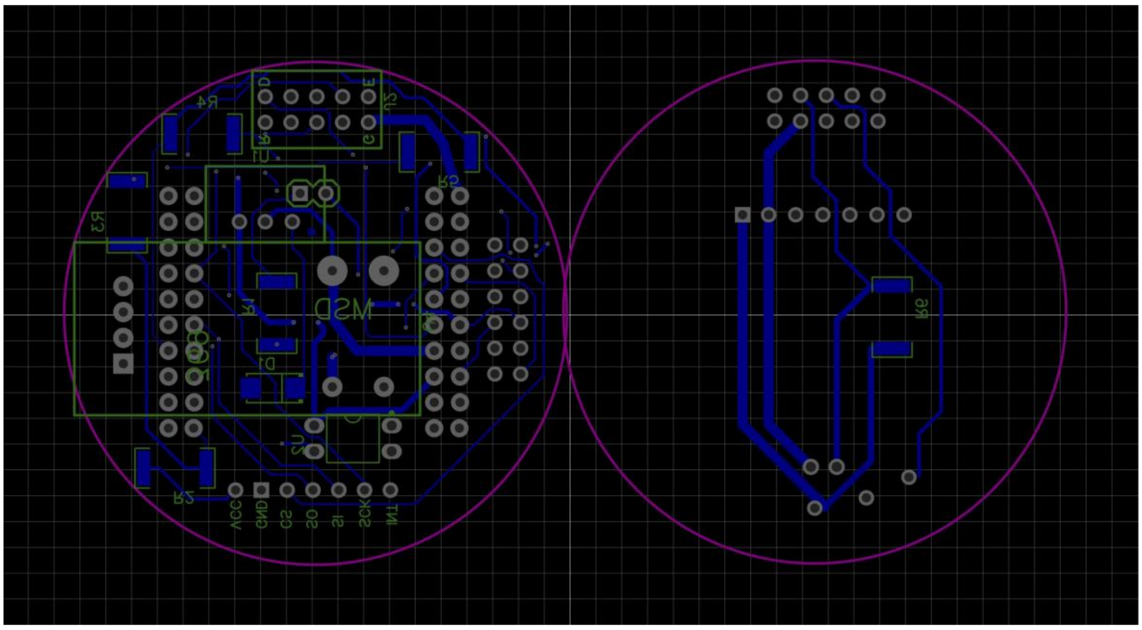


Figure 46. Bottom layer