National University of Science and Technology POLITEHNICA Bucharest
Faculty of Electronics, Telecommunications and Information Technology

Electronic Module for the Dashboard of a Formula Student Racing Car

Diploma Thesis

submitted in partial fulfillment of the requirements for the Degree of Engineer in the domain *Electronics and Telecommunications*, **study program** *Applied Electronics (ETC-ELAeng)*

Year 2024

Table of contents

List of figures

List of tables

List of abbreviations

CRT = Cathodic Ray Tube
LCD = Liquid Crystal Display
ECU = Engine Control Unit
OTA = Over The Air
LED = Light Emitting Diode
MTBF = Mean Time Between Failures
EDR = Event Data Recorder
NRZ = Non Return to Zero
DMP = Digital Motion Processor
PCB = Printed Circuit Board
SC IN = Safety Circuit Input
SC OUT = Safety Circuit Output
SDA = Serial Data
SCL = Serial Clock
DTR = Data Terminal Ready
RTS = Request To Send
SPI = Serial Peripheral Interface
I2C = Inter-Integrated Circuit
MISO = Master In Slave Out
MOSI = Master Out Slave In
CLK = Clock
CAN = Controlled Area Network
UART = Universal Asynchronous Receiver Transmitter
SD card = Secure Digital Card
VS = Visual Studio
IDE = Integrated Development Environment
FS = Formula Student
ADC = Analog to Digital Converter
HTTP = Hyper Text Transfer Protocol
POR = Power On Reset
VFS = Virtual File System
SSID = Service Set Identifier
RPM = Revolutions Per Minute
PIO = Programmable Input Output
LDO = Low Drop Out

Introduction

And God said, "Let there be light," and there was light. God saw that the light was good, and he separated the light from the darkness. God called the light "day," and the darkness he called "night." And there was evening, and there was morning—the first day. [2]

So, from the biblical point of view, this is how everything has started, yet, from a scientific point of view, we still do not have a concrete answer. After the Big Bang, galaxies and stars created the Universe as we know it. We, as human species, were lucky to find ourselves in a galaxy that has a bright star, the Sun as we call it, that would light up the path in front of us. And so, the light became part of the surrounding nature, and we became one of the species on Earth that can see the light, which was a big upside for us, since this allowed the human species to evolve.

However, from the point of view of an engineer, when there are upsides, must be also some downsides. In this case, one of the downside of the light is that we became too dependent on it. We need light to see, and we need to see in order to not be vulnerable to predators that can see better than us in the dark. However, this is not a problem in the modern society, since we built cities that would protect us from the predators, and yet we still need light to write, read, and to do any kind of activity. The sense of sight became one of the most important senses that we have, and we are using it more than any other senses. When looking into surveys about people ranking their senses based on which one they think is the most used, it seems that on average 80% of them are ranking the sense of sight on the first place, with the second most important being the hearing sense. This was studied in the survey from Figure 1.

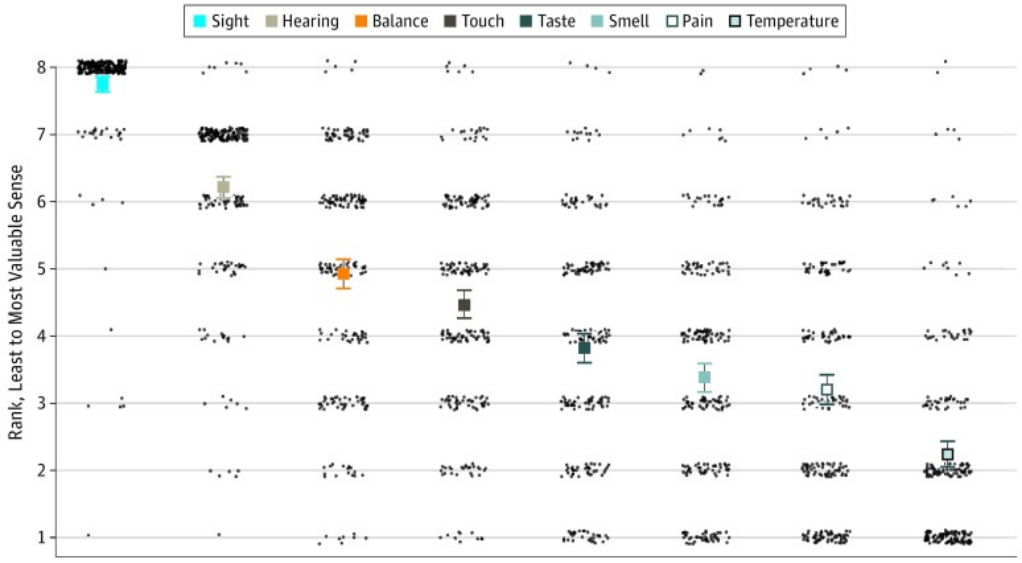


Figure 1: Human sense ranking survey. [1]

This explains why the human machine interaction was developed in such way to be based on the visual and auditory systems of the people. In terms of what a machine is, it can be anything from a simple mechanical device to a complex computer system. I will refer in the further text to the computer systems as machines. At first, the machines were mechanical and

people interacting with it had to see the machine, then mechanically operate it in order to set the input parameters. Afterwards, the machine would compute the output, which was the result of the input parameters, and display it on an area that was specially designated for this purpose. This area would contain digits that would show the results or the results could be printed on a piece of paper.

It was the first step in the human-machine interaction, and the way that the machine could communicate with the human was through light, with the specially designated area which will be later called the display. The mechanical machines succeded the electronic computers that were based on vacuum tubes at around 1943. This kind of machines would usually have front panels that had lights and switches through which the user could operate the machine and monitor its results. This was a good step forward, but it was a limited one, because the computer could not show complex graphical information on such front panel.

The next phase was in human-machine interaction was the cathodic ray tubes (CRT) displays mass adoption. Connected to such electronic computers, the CRT displays could show more complex graphical information, which was more than just the lights on the front panel. Right now many possibilities were unlocked. The user could see text, numbers, and even simple graphical shapes. Games that were developed during that time took advantage of this new technology. Pong was released on 29 November 1972, and it was the first commercially successful video game[3]. Pong is a table tennis sports game featuring simple two-dimensional graphics.

However, the CRT displays were bulky, heavy and consumed a lot of power. These drawbacks made them unsuitable for devices that were not connected to the power grid, such as cars. At first the automotive industry had a bias towards analog gauges, because they were considered to be more reliable than their electronic counterparts. But, this was not here to stay, because electronic devices became more robust, specially after World War II. The solid state diodes were introduced in the alternator, which became a standard part of any vehicle. It is used to charge the battery of the car, as well as to provide power to the electronic devices from it. The alternator was a key component in a combustion powered car, since it would transform the mechanical energy from the engine into electrical energy.

Liquid crystal displays (LCD) were invented afterwards, and they became a better alternative for the automotive industry, since they were thinner and used less power than the CRT displays. Even tough there were a few models of cars with CRT, they were typically high-end models, that were not adopted by the mass market. In the 1990s the LCD was introduced in the dashboard of the cars to display simple information such as the radio frequencies or the time. Toyota was one of the car companies that manufactured a full LCD dashboard for the x80 Chaser model. The idea of having a digital cluster was perceived as a luxury feature and was not standardized, however it was a trending idea among the manufacturers to use an LCD for the clock and the odometer. The mechanical odometer was based on a set of gear ratios that would turn the numbers on the display when the car was moving, whereas the LCD one was based controlled directly by the computer of the car, namely the Engine Control Unit (ECU). The innovative idea of introducing such a computer in a car was a necessary move since the software could adjust the timing of the engine, whereas a mechanical system could not, and this would limit the overall performance of the car. Coming back to the LCD, the first limitation was that it was monochromatic, and the user could not see the information in color.

During this time, the idea of introducing color for the LCD was also developed, and first introduced for hand held televisions in 1988 in Japan [4]. The upside of the monochromatic versions of the LCDs was that they had better visibility in direct sunlight and this was a key feature for the automotive industry. It took a decade until the color version started to be introduced in the dashboard of the cars. In 2000, models of the BMW 7 series was equipped with a color LCD as the main human machine interface. This was a revolution in terms of ways of interacting with the car because the user could select radio stations by means of a

rotary knob, not by buttons. Therefore, several buttons could be removed from the dashboard, making it look cleaner and more aesthetic.

The problem with the LCD was that it did need to be backlit in order to be visible in the dark, and this would become a greater problem for the color versions of it. A solution that was quickly adopted by the manufacturers was to use light emitting diodes (LED) in order to provide backlight for the LCD. It was a matter of time until the LCD became LED display, however, make no mistake that the LED display is the same in terms of technology as the LCD, but it was a way of differentiating the two technologies of backlighting the display, neon lights versus LEDs.

In 2012 Tesla introduced the Model S, which had a 17 inch touch screen LED display in the center of the dashboard, removing completely the physical buttons from it. This allowed the engineers to change the interface by means of software updates over the air (OTA). The paradigm now changed from implementing a hardware based interface to a software based one that is highly reconfigurable and customizable and the owner of the car could benefit from the new interface and features without having to buy a new car or having to modify the interior with aftermarket parts.

Figure 2: Car dashboard design with CRT, LCD and LED displays

The electronic module of the dashboard represents the all the electronic components that are used to power up the module, to control the human - machine interface that is shown on the LCD and to store the data that is read from the CAN bus of the formula car. As any of the modern vehicles on the road, the formula car has a CAN through which all the electronic modules are communicating, however, in this thesis, we will focus only on the dashboard module and on the ECU module. The ECU module is the one that acquires the data from the sensors of the car, processes it and then formats it in CAN packets and sends it via the bus.

Coming to the analyzed problems regarding the development of the dashboard module, one of them would be the ability to reconfigure. What if one would not think about all the

information that the pilot needs to see on the dashboard when planning it? What if other data is demanded by the team to be shown on the display in order to fit a new need that was not foreseen? The solution for this is to use an LCD that would show a graphical user interface (GUI) that can be reconfigured easily, and this to be done by software, because a new hardware implementation would be costly and time-consuming.

The possibility of using LCD in the dashboard of the cars, in general, was a great step forward in terms of the ability to reconfigure. Where it is still not fully accessible to the pilot himself, this feature is easily accessible to the engineers. Then a question still stands, what if the LCD screen is not as redundant as an analog gauge one. Based on the studies made by the Department of Electronics and Telecommunications of University of Florence, the actual mean time between failures MTBF of TFT-LCD screens is very high, around 738.003 hours which is around 35 years [5]. This is a good indicator that the overall reliability of the TFT-LCD screen is above the one of the overall of one of the car, which is around 10 years.

Another design aspect would be the integration of the LCD in the context of the dashboard for the formula student car. Besides the components that must be implemented on the dashboard because the rule book of the competition demands it, there are few other components added to the dashboard to make it more user-friendly, such as a button to change the display page. Therefore, the overall design is clean and simple consisting of the LCD, taking most of the space, and the buttons that control different functions of the car. As seen in the Figure 3, a red push button is integrated in the dashboard for the pilot to be able to cut ignition off in case of emergency, it is mandated by the rulebook. The neutral LED is also mandated by the rulebook, but it cannot be seen in Figure 3, since the FS car depicted there has an electric power train, which does not include a gearbox.

Figure 3: Formula Student dashboard example

But in this context, would it not be possible for the dashboard electronics to include some ways of recording itself the data that is shown on the display? This is a question that is also explored int this thesis. The problem is similar to the one faced by the aviation industry, where a non-volatile memory is embedded in the aircraft's black box. In case of any aviation accident, the black box is recovered, and the data is analyzed post-factum in order to determine what went wrong with the aircraft and why did it crash. This is ignoring the fact that the aircraft can be crashed by another aircraft or by an unindentified flying object, which the black box is not aware of, but this phenomenon is quite rare in the aviation. However, in the automotive industry, the crashes are usually happening between cars and objects, because when car's propulsion fails, the driver is able to recover the cars steering and braking systems

by mechanical means, and stop it safely.

The European Council made a proposal in this sense in 2021 to include a black box type of device in passenger cars that would be produced starting with 2024. The device would be named Event Data Recorder (EDR) and would store the data moments before the crash and after it, in order for the authorities to determine the circumstances of the accident in a data based manner. This kind of device is a novelty one, and therefore such device was not yet implemented in cars. This paper analyzes the problem and proposes a solution for non-volatile data storage directly in the dashboard of the car. The reader would ask himself why such solution is proposed, and this is because the dashboard of the car already receives all the necessary data, needed to be stored, in the black box, via the CAN bus, so it acts as a data aggregator in its format with a link on the CAN bus to read CAN packets from the car's ECU. So, the question is, why not store this data that is shown on the car's dashboard? Also, the analyzed solution includes details such as functional safety and placement of the dashboard in the car, such that it is in a position that would not affect its integrity in a crash in a way that data integrity is at risk. This paper also includes an analysis of the security of the data from the point of view of the CIA triad: Confidentiality, Integrity, Authenticity.

The thesis is structured in the following way: the first chapter presents the hardware component of the dashboard, this containing the LCD integration with the FS car and with the ESP32 microcontroller via UART protocol, the accelerometer circuit, the ESP32 programming and power delivery circuitry, CAN communication circuit, Hybrid selector circuit and the SD card command and control circuit. It will detail all the aspects regarding the physical implementation of the project and also present all the technical considerations that were taken when designing the PCB. In the hardware component chapter I will also discuss the power delivery aspect of the overall design and how much is the expected power consumption

In the second chapter I will present the software component of the dashboard. The project was split into components that will be detailed and as in the hardware case, explained thoroughly. The software was based on ESP-IDF framework, which includes the FreeRTOS kernel. Each module will be a standalone task which will be declared in the main. The software tasks include mainly the drivers for CAN communication, display communication over UART, SPI driver for the SD card data logging, I2C driver for communication with the accelerometer, and the other software parts that take part in the overall architecture.

In the third chapter the testing methodology of the hardware, as well as the software is presented. All the results of the tests are detailed as well as the steps taken to debug the hardware as well as the software and to obtain the results. Also, some conclusions will be taken that include what should be improved in the future work.

The last chapter includes the conclusions of the thesis based on the results presented in the previous chapter. Also the thesis will be resumed in a few sentences, and it will be decided if the overall goal was achieved or not.

Chapter 1
Hardware component

1.1 Dimnension constraints

The Formula Student car is a single seat car, as the Formula 1 cars are, however, it is not as long and does not include such a powerful engine. The Formula Student car is designed to be compact and lightweight, therefore competitive for the FS Autocross Event. During this event the car must be able to take hairpin turns and slaloms, where, in the case of slaloms, the cones must be put in a straight line with a spacing of 7.5m up to 12m between them. A narrower car offers the pilot the advantage of being able to swerve faster between the cones. This is why the design of the car is compact, and the dashboard must be integrated in such a way to respect this spacing constraints.

There is also the problem of the pilot being able to jump outside the car in case of an emergency. The rulebook of the competition manadates that the pilot must be able to jump outside the car in less than 5 seconds. Therefore, the width of the dashboard must allow for the knees of the pilot to be able to move freely in order for him to flex them and jump outside the car. There is a second technique, the pilot can press his arms against the body of the car and push himself outside the car, but this is a hard thing to achieve if he is not powerful. Two cut-outs are implemented in the dashboard of this year's car in order to allow the knees of the pilot to pass easily through as seen in the Figure 1.1.

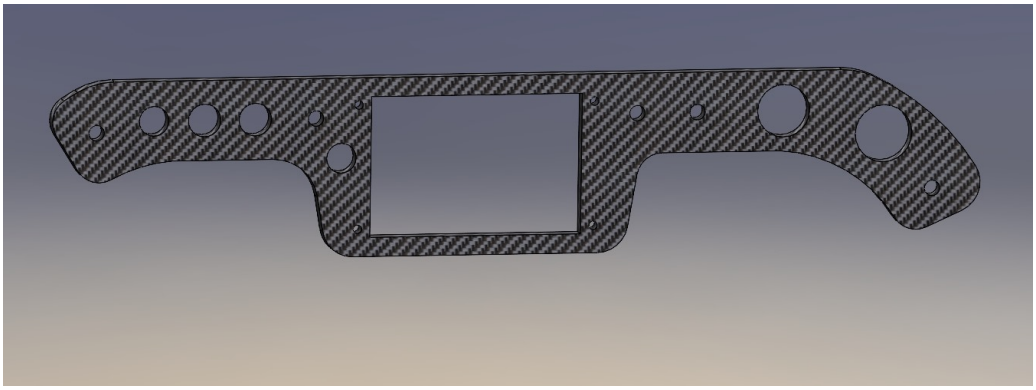


Figure 1.1: Formula Student preliminary DR05 dashboard

This is why a 3.5 inch LCD was chosen as the main display for the dashboard. It respects the constraint of having approximatively 100 mm width and 55 mm height. The chosen LCD is GEN4-ESP32-35 from 4D Systems. Its technology is TFT, which has a reliability expectation of at least 35 years, as stated in the introduction. The LCD has a resolution of 320x240 pixels, which is enough for the pilot to see all the needed graphical information. It will be connected to the ESP32 microcontroller via a 30 pin ribbon cable.

This was not the only constraint that was taken into consideration when choosing the LCD, it was also the know how of the UPBDrive Formula Student team that integrated in the past such LCDs from 4D Systems in the dashboard of the previous cars, and there was a good experience with them, since the team knew that it was a doable project.

_	Specification	Value
	Diagonal Display Size (in)	3.5
	Resolution	320x480
	Brightness (cd/m2)	320
	Backlight LEDs	1×6 Parallel
	Touch	Non-Touch
	Controller	ESP32
	Operating Ambient Temperature	-20°C to +65°C
	Storage Temperature	-30°C to +80°C
	Connector	30-pin FPC
	Display Viewing Area (mm)	73.44 x 48.96
	Module Dimensions (mm)	101.1 x 56.6 x 12.4
	RoHS?	Yes
	Country of origin	Australia
	Product Weight (g)	37
	REACH	Yes
	CE Mark	Yes
	Supply Voltage	4.0V – 6.0V DC
	Supply Current	183 mA
	SD Card Slot	Yes – RAW or FAT16 Support. SPI Compatible microSD Cards Only
	Flash	16MB External Quad SPI
	RAM	8MB Internal Octal SPI
	I2C	2 x Channels
	GPIO	20 x GPIO
	SPI	2 x Channels
	Codeless Programming	Yes
	IDE	4D Workshop4 IDE
	Serial Speed	5000000 baud
	Serial Interfaces	1 x dedicated and 3 x configurable
	8-bit Parallel Transfer	Yes

Table 1.1: GEN4-ESP32-35 specifications

As seen in the Table 1.1, the display has a Serial Interface, namely the UART mentioned above, through which the ESP32 microcontroller will send the data to be shown on the LCD and the LCD will send the acknowledge signal back to the ESP32. It can also be seen that the GEN4-ESP32-35 has a SD Card Slot, through which the interface, and by interface I mean all the graphical objects displayed on the screen, can be stored and loaded at boot time. The GEN4-ESP32-35 module knows when the data on the SD card has changed, and it will reload the update process at boot time in order to reprogram the LCD with the new interface.

1.2 Microcontroller ESP32 WROOM 32 description

The processing unit of the dashboard electronic module is the ESP32 WROOM 32 microcontroller. It has 2 cores, each running at 240 MHz, and it is based on the Xtensa LX6 architecture on 32bit. The ESP32 has 520 KB of RAM and 448 KB of ROM, and 4 MB of flash memory. The microcontroller also supports eFuse memory, which can be written into just one, then it is read only. This kind of memory is used for storing the MAC address of the microcontroller as well as the keys used for flash encryption and decryption. The 4MB of flash memory is connected to the chip via an SPI bus. In order to program the ESP32, the chip must be put in boot mode by pressing the boot button and then the reset button. GPIO 0 is the boot pin, which is connected to both a button and the programming circuit. The following table explains how the state of GPIO 0 and the other strapping pins are used to determine the boot mode of the ESP32 [6].

Voltage of Internal LDO (VDD_SDIO)					
Pin	Default	3.3 V		1.8 V	
MTDI	Pull-down	0		1	
Booting Mode					
Pin	Default	SPI Boot		Download Boot	
GPIO0	Pull-up	1		0	
GPIO2	Pull-down	Don't-care		0	
Enabling/Disabling Debugging Log Print over U0TXD During Booting					
Pin	Default	U0TXD Active		U0TXD Silent	
MTDO	Pull-up	1		0	
Timing of SDIO Slave					
Pin	Default	FE Sampling FE Output	FE Sampling RE Output	RE Sampling FE Output	RE Sampling RE Output
MTDO	Pull-up	0	0	1	1
GPIO5	Pull-up	0	1	0	1

Figure 1.2: ESP32 strapping pins explained

In this project, the SoC that includes the ESP32 microcontroller, the flash memory and the oscillator is used in the WROOM 32 package. The ESP32 has a lot of peripherals and therefore it is a good choice for this project.

It includes an I2C intellectual property (IP) that is used to communicate with the accelerometer. The I2C IP supports both master and slave modes, but for this project, it will be configured as a master. The fast mode of the I2C bus is used to communicate with MPU6050, which is the accelerometer used in this project.

Regarding the CAN communication, it is done via the CAN controller IP that is included in the ESP32 microcontroller. The name of the IP is TWAI from two wire automotive interface, because CAN is a trademark of Bosch, and Espressif could use in the code the same name as the trademark. The TWAI (CAN) IP supports both 11 bit and 29 bit identifiers, and it can be configured to work in multiple modes: normal, listen only and loopback. For this project, the normal mode is used in order to communicate with the CAN transceiver and on the CAN bus. Since ESP32 does not include a built-in CAN transceiver, the transceiver is an external one that is described in the next subchapter.

1.3 CAN transceiver SN65HVD230DR description

The CAN transceiver that was selected for this project is the SN65HVD230DR from Texas Instruments. It is a high speed CAN transceiver recognized for its reliability and efficiency in the automotive industry and industrial applications. The transceiver is built to withstand signaling within a high noise environment, as per the CAN standard. CAN is a differential signaling protocol, which means that it is more robust to additive noise and also allows for errors to be detected when the transmission occurs.

Some key characteristics of the SN65HVD230DR are the following. It allows for a maximum of 1 Mbps data rate on the CAN bus, which is more than enough for the FS car bus. It is supplied with 3.3V, and it has a low current consumption of just 370 uA. The CAN bus operates, usually, with differential +/- 2.5V signals, however, this transceiver allows for +/- 36V signals on the bus in case it is needed. On the microcontroller side of the transceiver, it allows for 3.3V logic levels, which is in sync with the ESP32 microcontroller that is used in this project, which also operates at such logic levels [7].

The CAN network is not only used in the automotive industry, but also in the industrial applications where the noise is high and the reliability of the data is crucial. The same can be said for some medical applications where the data must be transmitted in a precise manner and must also be error free.

The purpose of this transceiver is to allow the ESP32 microcontroller to communicate on the CAN bus. The ESP32 does not include a CAN transceiver in its package, therefore it cannot read and write differential signals from the bus directly. The transceiver is able to transform the differential signals into TTL signals that can be interpreted by the ESP. However, the ESP includes a CAN controller IP, which is used in this project to interpret the TTL signals received from the transceiver and also to send such TTL to it that will be further converted into differential signals. In this way, the transceiver converts the TTL CAN RX and CAN TX signals from the ESP32 into the differential CAN H and CAN L signals. A detailed description of the transceivers specifications can be found in the Table 1.2.

_	Specification	Value
	Data rate	1 Mbps
	Supply voltage	3.3V
	Current consumption	370 uA
	Operating Temperature (Max)	+125 °C
	Mounting style	SMD
	Package case	SOIC-8
	Output type	Digital
	Logic level CAN TX, CAN RX	3.3V

Table 1.2: SN65HVD230DR CAN transceiver specifications

1.4 Display GEN4-ESP32-35 4D Systems description

I also conducted a research about the brightness of the LCD in order to see weather the brightness of 320 cd/m2 is enough for the pilot to see the display in direct sunlight. This is an imporant feature, because the pilot also wears a helmet, which has sunlight glasses like wind protection, and this also reduces further the visibility of the LCD.

There is a an ISO 3664:2009 standard that regulates the viewing conditions for the color proofing of originals and reproductions in the graphic arts industry. It says that 80-120 cd/m2 in a controlled environment should be enough for a display. By a controlled environment they mean a dimmed room with minimal light from outside [8]. So, this can be taken as a minimum amount of brightness for an indoor display. For the minimum amount of light for a outdoor display, there are vendors ad displays that mention a minimum of 1000 cd/m2 needed in order to have a visible screen in a bright sunny day [9]. Another aspect worth considering is the brightness of the high-end daily driver phones. The S21 5G has an 856 cd/m2 peak brightness when showing white spots, whereas the iPhone 13 has a peak brightness of 1200 cd/m2.

To strengthen the argument, I compared completely arbitrary 5 average priced phones that are ranging from 1500 – 2000 RON on emag.ro marketplace. The first one is Motorola Edge 40Neo which has a peak brightness of 1300 nits. The second one is the Xiaomi 13 Lite having a peak brightness of 1000 nits. The third one is the Honor Magic5 Lite having a HBM brightness of 800 nits. The fourth one is the Samsung Galaxy A54 which has a 1000 nits HBM brightness and the last one is the Xiaomi Poco X5 Pro having a HBM brightness of 900 nits. Taking all the points above into consideration there yields an average of: 956 cd/m2 peak brightness. Therefore, the following Figure 1.4 can be made for a better visualization of the data taken from the phone's specification sheets.

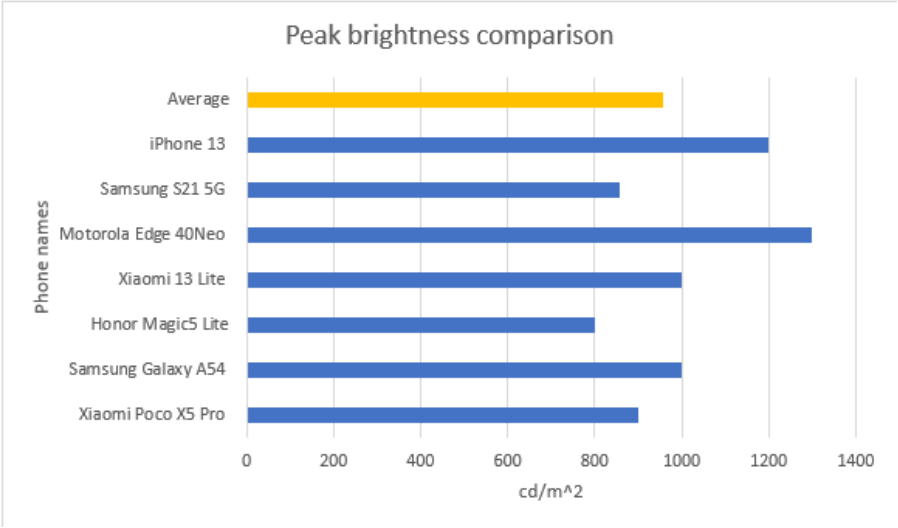


Figure 1.3: Peak brigthness of the mobile phones

In order to understand better the market choices that we have, I compared the brightness of the displays of choice from 4D Systems with the maximum brightness of the phones as seen in the Figure 1.4.

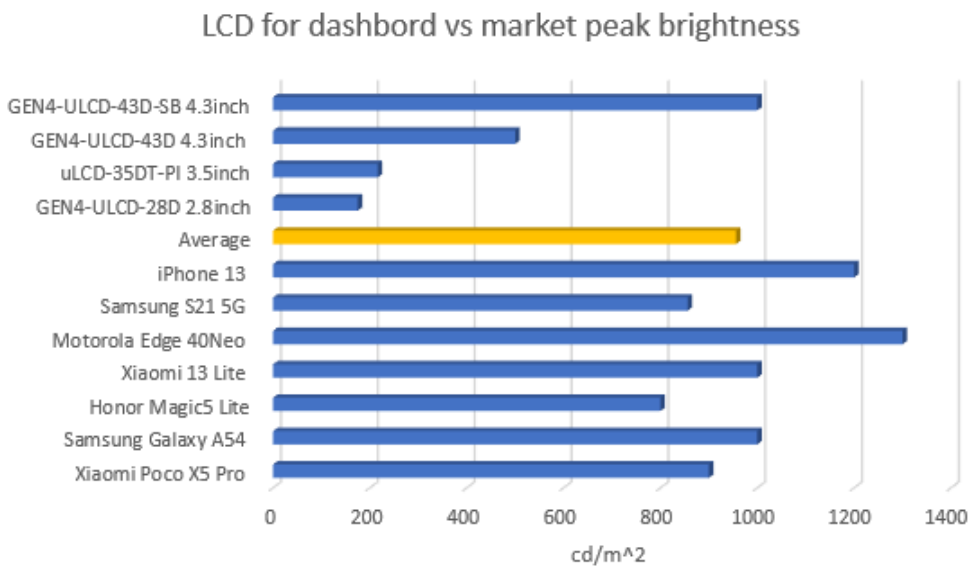


Figure 1.4: Peak brigthness of the LCD from 4D Systems and mobile phones

An issue arises, the price of the displays, however, are not comparable, the cheapest being around 300 RON the most expensive being around 600-700 RON. The comparison above is only meant to see the difference in brightness of the 4 displays of choice. As you can see, the GEN4-ESP32-35 has a brightness of 320 cd/m2, was a choice of compromise between the brightness and the dimension constraints, the dimension one being a more important decision factor. On the other side, the low brightness of the LCD will affect the effectiveness of it, but it will be doubled by a LED bar which will show the most important aspect of the car, the RPM, which is very bright and will compensate for the low brightness of the LCD.

1.5 LED bar NeoPixel Stick description

Another hardware aspect of the dashboard is the redundancy of the LCD. From the point of view of the reader, this would seem like a nonsense, since the LCD lifespan is around 35 years, as stated above, however, the software controlling the LCD is not open source. I refer to this by the software incorporated in the GEN4-ESP32-35 module, which reads the data sent by the ESP32 microcontroller and displays it on the dashboard. What if it fails mid run? There are zero guarantees from 4D Systems that their software is bug free, unit tested, and that it respects any automotive standard. Therefore, a LED bar was chosen to be integrated in the dashboard such that to double the most important information that should be displayed to the pilot, the RPM value. The LED bar is a software free solution, from the point of view of the bar itself. It uses a one wire communication protocol, which respects the NRZ standard. On that wire, data is sent by ESP32 microcontroller, which is then read by digital latches. In this way, the ESP32 microcontroller can effectively control any number of LEDs that are connected in a serial way to the one wire bus, because the 1 LED will read the first 3 bytes in the data stream, then act like a mirror and sent the others to the next LED. These LEDs are named WS2812B and all the above information is taken from their datasheet [10].

_	Specification	Value
	Power supply voltage VDD (V)	+3.5V - +5.3V
	Input voltage (V)	-0.5V - VDD+0.5V
	Operation junction temperature	-25℃ - +80 ℃
	Storage temperature range	-40℃ - +105 ℃
	Input current	1 uA
	Input capacity	1 pF
	Red luminous intensity	390-420 mcd
	Green luminous intensity	660-720 mcd
	Blue luminous intensity	180-200 mcd
	Data transfer time TH+TL	1.25us±600ns

Table 1.3: WS2812B LED specifications from LED Bar

As seen in the Table 1.3, the WS2812B LED has an input voltage of -0.5V to VDD+0.5V, which is a good feature, because the ESP32 microcontroller is operating with 3.3V logic levels which is in the range of -0.5V to VDD+0.5V. Also, the LED bar is already implemented on a PCB, which boosts the implementation of this project in terms of less time invested. Another good aspect from the hardware point of view is the fact that the LED bar can be moved in any position on the dashboard, since it can be connected via wires to the main PCB of the dashboard, which contains the ESP32 microcontroller. The wires will be connected through vias and soldered to the main PCB and the PCB of the LED bar respectively. This kind of connection is not easily removable, but, the overall design of the dashboard is to be removable as a whole, this including the LCD, the main PCB of the dashboard, the LED bar and the buttons. Also, in order to connect it to the FS car, the dashboard will have a single 12 pin connector that will make the connection between it and the wire harness of the car.

1.6 Connector automotive TE DT15-12PA-B016 description

The connector choice was tricky, since the rulebook does not contain any kind of regulations regarding this, so at first it seems that any kind of connector can be a choice for this. Narrowing down the spectrum, I looked into automotive industry approved connectors, since they are designed to withstand the harsh environment of the car, which includes vibrations, high temperatures, and humidity. In the case of the dashboard itself, I would say that high temperatures are not a problem, but humidity will sure be. The FS car is an open cockpit car, which means that the dashboard will be in direct contact with the environment, including rain and dust. Therefore, the connector must be waterproof in order to avoid problems of accumulating water in the connection pins. It seems that TE electronics has a good solution for this, their automotive connectors being waterproof and having a good resilience to vibrations.

However, the TE connectors are not cheap, and the budget of the project is limited, therefore one I conducted and analysis of the available solution in the market in order to determine of the solution that seems to me to be the most feasible is also the objective feasible one. By objective, I mean a measurable quantity based on the criteria that the connector must be from a reputable vendor, must be waterproof, easy to connect and disconnect, and must be able to withstand vibrations. The following Table 1.4 resumes the analysis of the connectors available on the market.

_	Specification	TE DT15-12PA-B016	Automotive 2.54 mm pins	Molex 36783-1208
	Ingress protection	IP68 dust and water protection	No rating, pins are exposed	No rating, pins are sealed
	Material	Plastic housing and bronze pins	Plastic and bronze pins	Plastic housing and bronze pins
	Product type	Automotive Connectors	Headers and Wire Housings	Automotive Connectors
	Storage temperature range	-40°C - +105 °C	-55°C - +105 °C	-40°C - +105 °C
	Mechanical Attachment	Screwed in the PCB, soldered pins	Soldered pins	Soldered pins
	Operating Temperature (Max)	+105 °C	+105 °C	+105 °C

Table 1.4: Comparison between different connectors on the market

As seen in the Table 1.4, from the point of view of vibration redundancy, the TE DT15-12PA-B016 is the best of the three, because it is screwed in the PCB, which means that it does not rely solely on the soldered pins, which can break in case of vibrations. Also, from the point of view of ingress protection, TE DT15-12PA-B016 is the best, is IP68 rated, making it the best choice for the dashboard of the FS car. So, from an objective point of view, the TE DT15-12PA-B016 is the best choice for the connector of the connectors presented above, not only from a subjective point of view.

1.7 Accelerometer MPU6050 description

From the point of view of the accelerometer, the choice was made based also on experience of the team with programming it. The accelerometer is a key component in the telemetry of the FS car, since it determines the acceleration of it on all three orthogonal axes, X, Y and Z. The MPU6050 accelerometer has a 16 bit resolution on each axis, with a programmable maximum reading of 2g, 4g, 8g or 16g [11]. The dashboard accelerometer is a redundancy solution for the accelerometer inside other telemetry module of the car, the Vehicle Dynamics Unit (VDU). The accelerometer communicates with the ESP32 microcontroller via I2C protocol, which is a two wire communication protocol, easy to implement, and easy to program. It supports fast I2C standard having a clock of 400 kHz, which is achievable by the ESP32 microcontroller, which also supports fast I2C. A downside of this choice is the fact that the MPU6050 is not automotive grade, and it is also at the end of life, which means that it is not meant to be implemented in new designs.

However, the specifications of MPU6050 makes it a great choice aligned with the expectations that we have from it. It allows for a precise measurement of the acceleration of the car, which can be read easily via I2C protocol. It also includes advanced processing techniques such as filtering which allows for a better signal to noise ratio. Also, the ESP32 will not do any signal processing which is a good thing considering that it is a general purpose microcontroller, not a digital signal processing one. The documentation of MPU6050 is also very good, because it contains all the information needed for the programmer to communicate with it. It details every register of the accelerometer, and also the way that the data is formatted inside it.

The accelerometer has the following characteristics detailed in the Table 1.5.

_	Specification	Value
	Sensors	3 axis MEMS accelerometer and a gyroscope
	Measurement range accelerometer	+/- 2g; +/- 4g; +/- 8g; +/- 16g
	Measurement range gyroscope [degrees/s]	+/- 250; +/- 1000; +/- 2000
	Filtering	Includes a Digital Motion Processor (DMP)
	Operating Temperature (Max)	+105 ℃
	Mounting style	SMD
	Package case	QFN-24
	Output type	Digital
	Supply voltage (MAX)	3.46V
	Supply voltage (MIN)	2.375V

Table 1.5: MPU6050 accelerometer specifications

1.8 Buttons panel mount and emergency description

In order to have a complete dashboard unit, the buttons needed for the pilot to interact with this module are needed. There are several buttons needed for the dashboard module, namely the: change page button, emergency button, launch control button and engine start button. In the dashboard will be integrated just the change page button, the emergency button and the engine start one. The launch control, and the fan control button are integrated in the steering wheel.

For the change page button, a simple push button was chosen, which is also panel mount. The behavior of it is the ON-OFF, meaning that when the button is pressed, the circuit is closed, but when it is released, the circuit opens. It is a non retain button. It will act in a low power circuit, which has 3.3V maximum voltage and a current of 10 mA. The button is connected to the ESP32 microcontroller via a GPIO that will monitor if the button is pressed, then it will change the page on the LCD display. The button connects the GPIO to the ground.

The engine start button is also a push button, and non retain. It is also a panel mount button, which means that it can be mounted on the dashboard panel. The electrical behavior of the button is the same as the change page button, however it is not connected to any GPIO on the ESP32 microcontroller. It operates in a low voltage circuit, which has a maximum of 12V and a maximum current of 100mA when the button is pressed. This button is connected to the engine start relay, that is a high current relay.

The emergency button is a push button that is panel mount and is a retain button. In this case the behavior of it is (ON)-(OFF), meaning that when the button is pressed, the circuit is open, but when the button is not pressed, the circuit is closed. This button operates in a low voltage circuit, which has a maximum of 12V and a maximum current of 500mA when the button is not pressed

1.9 PCB Design and EDA tool description

The schematic of this project and the PCB design was made in KiCAD 7.0.6. KiCAD is an open source tool that allows for the design of the schematic and the PCB in the same software. The schematic is the first step in the design process, where the components are placed on the design sheet and connected to one another via wires. Compared to OrCAD, KiCAD is free and open source, therefore it is accessible to anyone and can be used without any license. This allows for the design of the PCB to be directly linked with the schematic, that meaning if someone tries to add a new component to the schematic, it will be automatically linked in the

PCB design tool. This great feature was missing in the OrCAD 16.6 software that was given by the university to the students.

Another feature of KiCAD is the community support. There are many footprints, schematic symbols and 3D models available just by downloading the tool from the official website. Besides them, there are also many other footprints, virtually for any component that the user can think of, available on the Mouser website. One can search directly for the component that he wishes on the website, check the datasheet, then, if the component fits the requirements of the project, one can download a package that contains the footprint of the component as well as the schematic symbol and the 3D model of it. All of them can be directly imported in KiCAD, without the need of any other tools and hassle.

1.10 PCB integration circuit of the ESP32 microcontroller SoC

In order to maximize the space on the PCB, the ESP32 microcontroller's development board was reimplemented on the PCB such that it has the shape that is needed for this project. For this, the ESP32 SoC that includes the Wi-Fi antenna, the Flash memory and ESP32 core and the oscillator needed for the core to run, is considered as a standalone component. For it to be programmed and to run, one would need to further design the programming circuit and the power supply circuit. The ESP32 SoC is supplied with 3.3V, therefore an LDO regulator is used to drop the 5V supply voltage from the USB connector or the Buck converter to 3.3V.

The programming circuit of the ESP32 SoC is needed because the ESP32 SoC does not have a USB port to be programmed directly. It has a USB to UART converter that is used to transform the differential USB signal into TTL signal that can be interpreted by the ESP32 itself. An interesting approach of this design is the fact that Espressif is using the RS232 extra signals, besides TX and RX, namely DTR and RTS to reset the ESP32 and to put it in programming mode via the bootstrap pins, which can be referred into the datasheet of the microcontroller [6]. This feature allows the programmer to just send the code via USB, and the ESP32 will automatically reset and enter flash mode. The flashing tool provided by Espressif ensures that DTR and RTS signals are sent to the ESP32 at the right moment in time in order for the chip to enter flash mode.

The schematic of the ESP32 microcontroller integration on the PCB can be seen in the Figure 1.5.

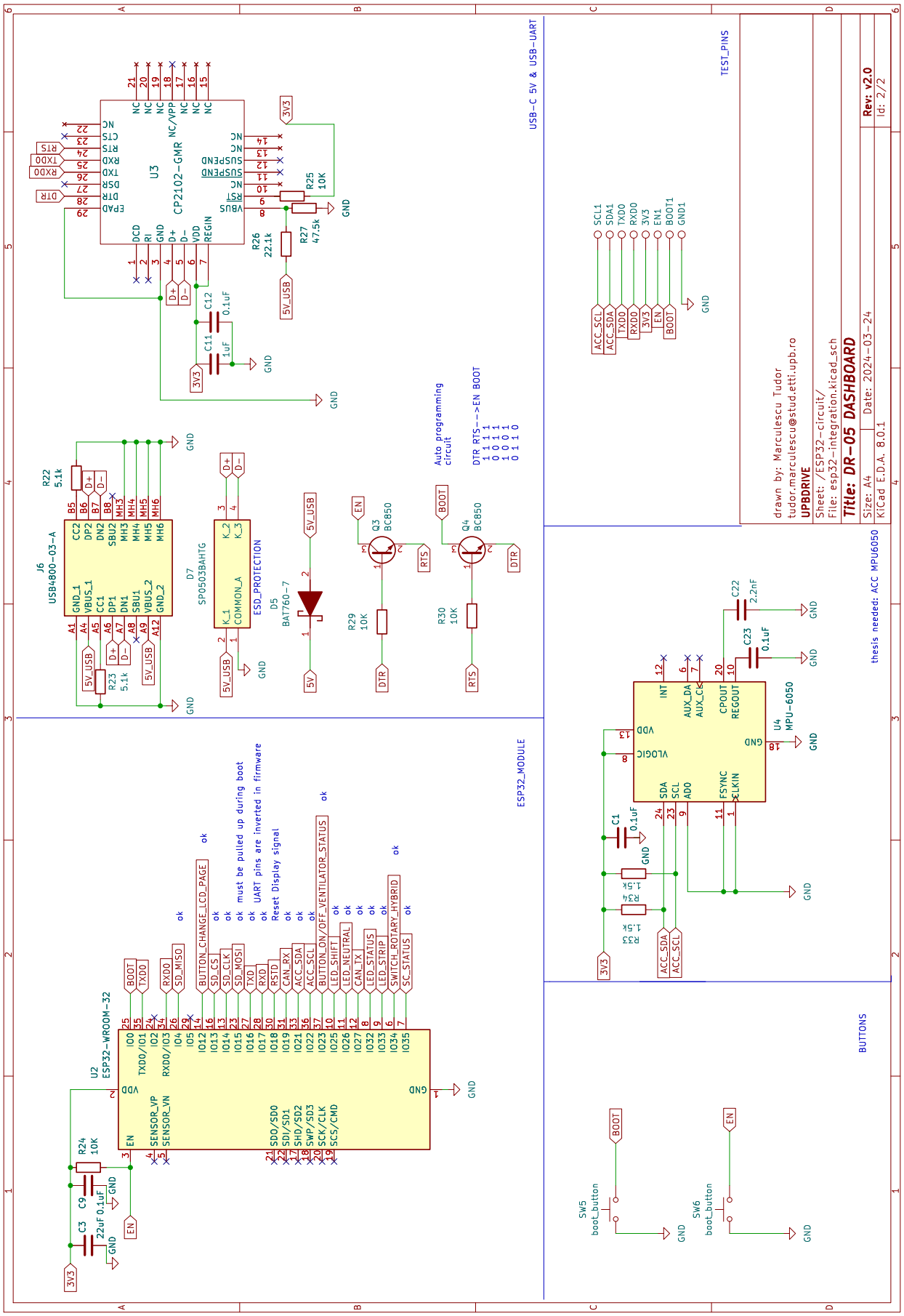


Figure 1.5: ESP32 schematic

Coming to the ESP32 module, 2 decoupling capacitors are used on the power supply line, one for high frequency noise and one for low frequency noise. The capacitors are placed as close as possible to the ESP32 module, such that to have the best effect of decoupling. A 10k pull-up resistor is used on the EN pin, which ensures that the chip stays in the enabled state when the enable signal is not connected to the ground. This EN pin is used to reset the ESP32 microcontroller, and also to put it in programming mode. Also, the pins TXD0 and RXD0 are used sending and receveing the TTL signal from the USB to UART converter, namely the CP2102-GMR chip.

The GPIO 4, 13, 14, 15 are used to communicate with the SD card slot that. The protocol that is used in this case is Serial Peripheral Interface (SPI) this is why 4 pins are needed. The SPI protocol is a synchronous serial communication protocol that is used to communicate with the devices that are in close proximity with the microcontroller. It has a master-slave architecture, where the ESP32 is the master and the SD card is the slave. The data lines are called MISO and MOSI, the clock line is called CLK and the chip select line is called CS. CS is needed to signal the slave device that the master wants to communicate with it. So, if the master wants to communicate with 5 slaves on the same bus, it will need 5 CS lines, one for each slave. This is the main difference between the SPI and I2C protocol, where the I2C protocol has only one address line that is used to address the slave device, so one master can communicate with multiple slaves on the address line SDA, namely up to 127 slaves.

GPIO 12 is a digital input pin that is used to read the state of the button that is used to change the page on the LCD display of the dashboard. The button is connected to the PCB via through hole technology, hovewer, it will not be directly connected to the PCB itself, it will have cables that extend from the PCB to the button. This offers flexibility of placement in case of the buttons.

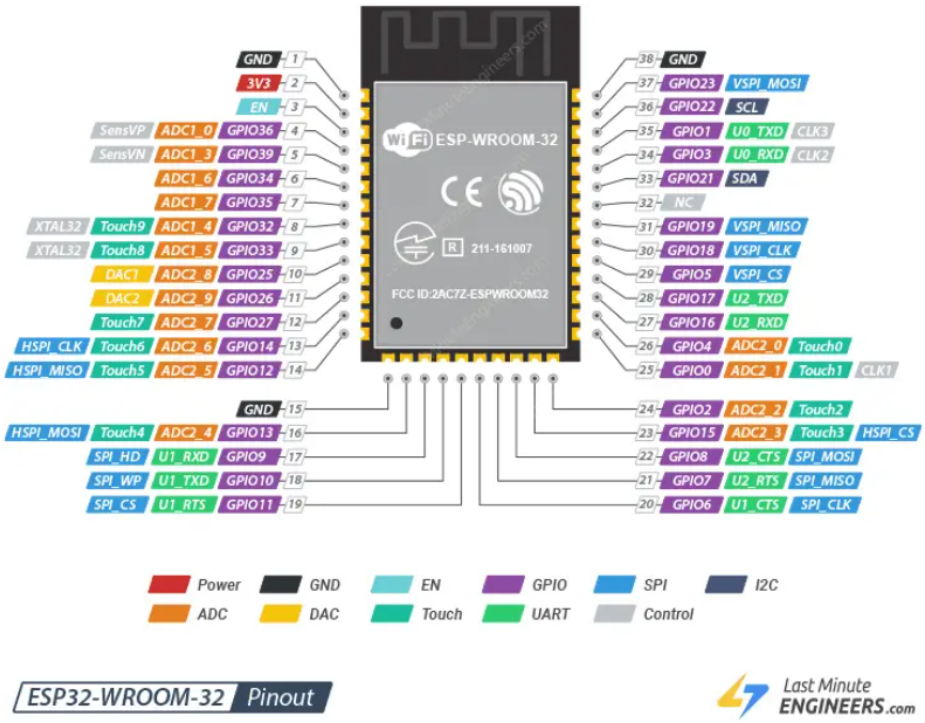


Figure 1.6: ESP32 pinout

GPIO 17, 18 and 16 are used to communicate with the LCD of the dashboard. GPIO 16 and 17 are the TX and RX pins from the UART 2 IP integrated in the ESP32 microcontroller.

The TX pin is used to send the data to be displayed on the LCD, and the RX pin is used to receive the acknowledge signal from the LCD. The acknowledge signal is used to determine if the LCD is working properly or if it is in a freeze state. GPIO 18 is a standard digital output pin that is used to reset the LCD in case of a software bug.

GPIO 19 and GPIO 27 are used to communicate on the CAN bus via the CAN transceiver. The first one is CAN RX and the second one is CAN TX. On the CAN RX lines the packets from the bus are received and via the TX line packets are sent on the bus. The CAN bus is a differential signaling protocol, however CAN RX and CAN TX are TTL signals, the differential CAN signal being obtained on the CAN transceiver side.

GPIO 21 and GPIO 22 are used to communicate with the accelerometer via the I2C protocol. GPIO 21 is the SDA line and GPIO 22 is the SCL line. They are connected to the I2C IP integrated on the ESP32 microcontroller, the pinout of the chip being detailed in the Figure 1.6.

GPIO 23 is a digital input pin that is used to read the state of the button that is controlling the radiator fan of the car. The button is connected to the PCB via through hole technology, hovewer, it will not be directly connected to the PCB itself, as in the case of the change page button.

GPIO 25 is a digital output pin that is used to control that is used to control the shift LEDs. Besides the shift bar, there are also some shift LEDs that are used to show the pilot that it is the proper time to shift the gear.

GPIO 33 is a digital output pin that is used to control the LED bar of the dashboard.nm, Via this pin, the data is sent to the LED bar, based on the protocol that is described in the datasheet of WS2812B LED. The LED bar is used to display the RPM of the car.

GPIO 34 is an analog input pin that is used to read the state of the rotary switch that is used to change the way the hybrid system works. The rotary switch has 11 positions through which different voltage dividers are used to determine the position of the rotary switch. The voltage is between 0V and 3.3V and is read by the ADC2 peripheral of the ESP32 microcontroller.

GPIO 35 is a digital input pin that is used to read the state of the safety circuit of the car, if it is closed or open. The safety circuit is used to determine if the car is in a safe state to be started. If the safety circuit is not closed, the engine can not start since the spark plugs and the fuel pump are not powered.

The programming circuit of the ESP32 microcontroller is detailed in the Figure 1.5. It has a USB to UART converter that is compatible with the RS232 standard. DTR and RTS signals from the standard are used in order to reset the ESP32 microcontroller and to put it in programming mode. Two bipolar transistors are used to convert the signal logic into the logic that is required by the Espressif. The following table of truth describes the functionality of the programming circuit.

_	DTR	RTS	EN	BOOT
	1	1	1	1
	0	0	1	1
	1	0	0	1
	0	1	1	0

Table 1.6: Programming circuit truth table

By using this kind of circuit, the programmer can directly flash the binary code into the microcontroller without the need to press any button on the board. This is especially usefull when doing OTA updates.

Coming back to the integration circuit, all the components were placed on the top layer of the PCB, such that to have an easier soldering process. The SoC is placed on the top left corner of the PCB in order for the antenna to not be placed directly on the PCB, which would have a negative impact on the signal strength. The CP2102-GMR chip is placed as near as possible to the USB-C connector in order to have the shortest possible differential traces between the USB connector and the chip. In terms of power supply, the LDO regulator is placed near the SoC, that is also because the harness connector takes most of the space on the right side of the PCB. Figure 1.7 shows the 3D view of the integration circuit.

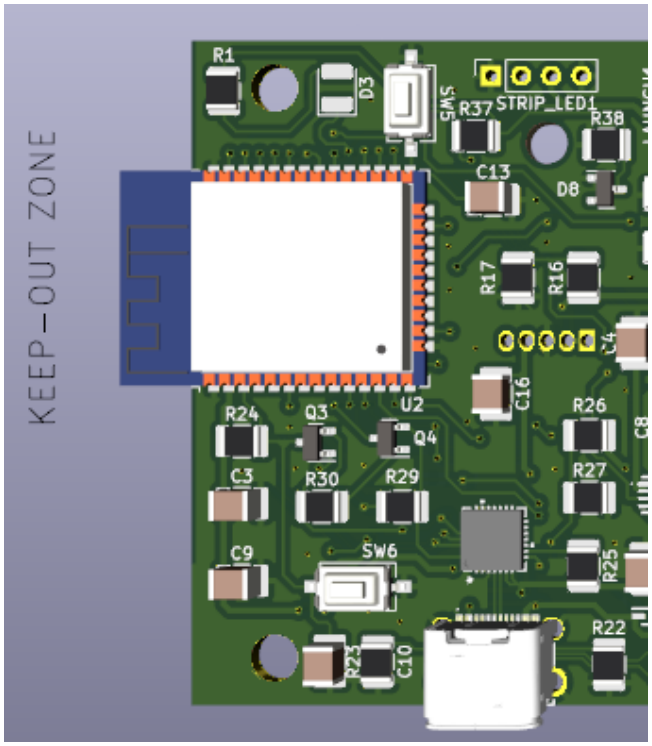


Figure 1.7: ESP32 SoC physical implementation on the PCB

Buttons for the EN pin and BOOT pin are also placed on the PCB, in order to have the possibility to manually reset the microcontroller. They are usefull in the debugging process, when the ESP32 might not respond to the commands sent via the USB to UART converter. Also a heartbeat LED is placed in the "D3" position in order to check if the ESP32 is starting the code that is flashed on it.

1.11 PCB implementation of the CAN transceiver

The design of this part of the PCB was implemented by following the reference design of the datasheet of the transceiver. The interesting part is the fact that the transceiver has a selectable slope speed for the CAN bus, which can be selected via the RS pin. In the case of this project, the RS pin is connected to the ground, which means that the slope of the CAN bus is abrupt, and the data is transmitted at a high speed, ranging from 500 Kbps to 1 Mbps. The schematic of the CAN transceiver can be seen in Figure 1.8.

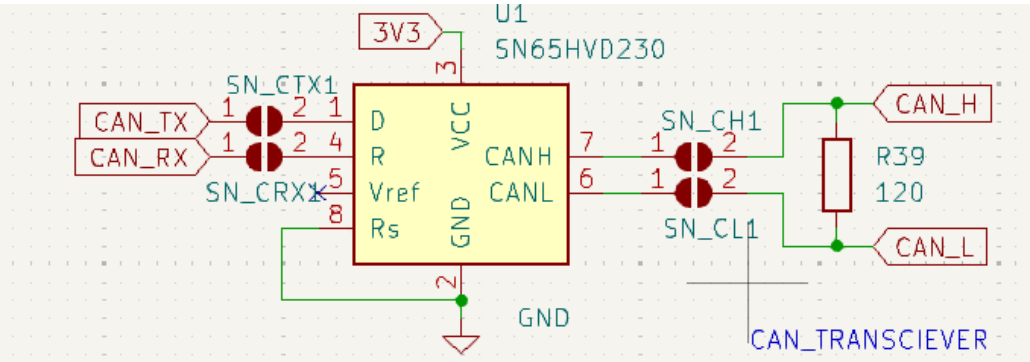


Figure 1.8: SN65HVD230DR CAN transceiver schematic

A 120 Ohm resistor is connected between the CAN H and CAN L pins, which is used to terminate the CAN bus. This resistor is called the termination resistor and its purpose is to eliminate the reflections of the packets on the bus. This phenomenon can occur if the data rate of the bus is high and the packets are not read correctly by the receiver or are not sent correctly. The termination resistor in the case of the CAN bus needs to be implemented only on the first and last node of the bus, such that the measured impedance is 60 Ohms.

A constraint of the PCB design is the fact that the CAN High and CAN Low copper traces need to be of the same length, because CAN is a differential signaling protocol. This was obtained by using the length matching tool in KiCAD, which allows for the copper traces to be equalized in length.

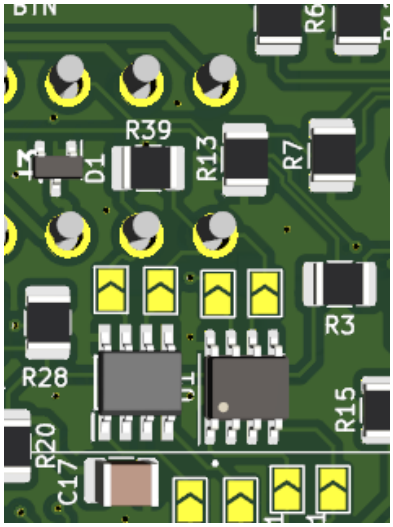


Figure 1.9: CAN transceiver physical implementation on the PCB

In Figure 1.9 is depicted the physical implementation of the CAN transceiver on the PCB. It is also placed on the top layer, near the CAN HIGH and CAN LOW pins in the harness connector. This was done in order to have differential traces as close in length as possible, since CAN HIGH and CAN LOW are differential signal lines. There are also implemented pads in order to select from the SN65HVD230DR transceiver or the ATA6563 transceiver from Microchip. For the purpose of this project, the SN65HVD230DR was connected to the ESP32 microcontroller

1.12 PCB implementation of the accelerometer

The accelerometer is a key component of the dashboard, since it is used to measure the acceleration of the car on all three axes. It is connected to the ESP32 microcontroller via I2C protocol. The schematic of the accelerometer can be seen in the Figure 1.10.

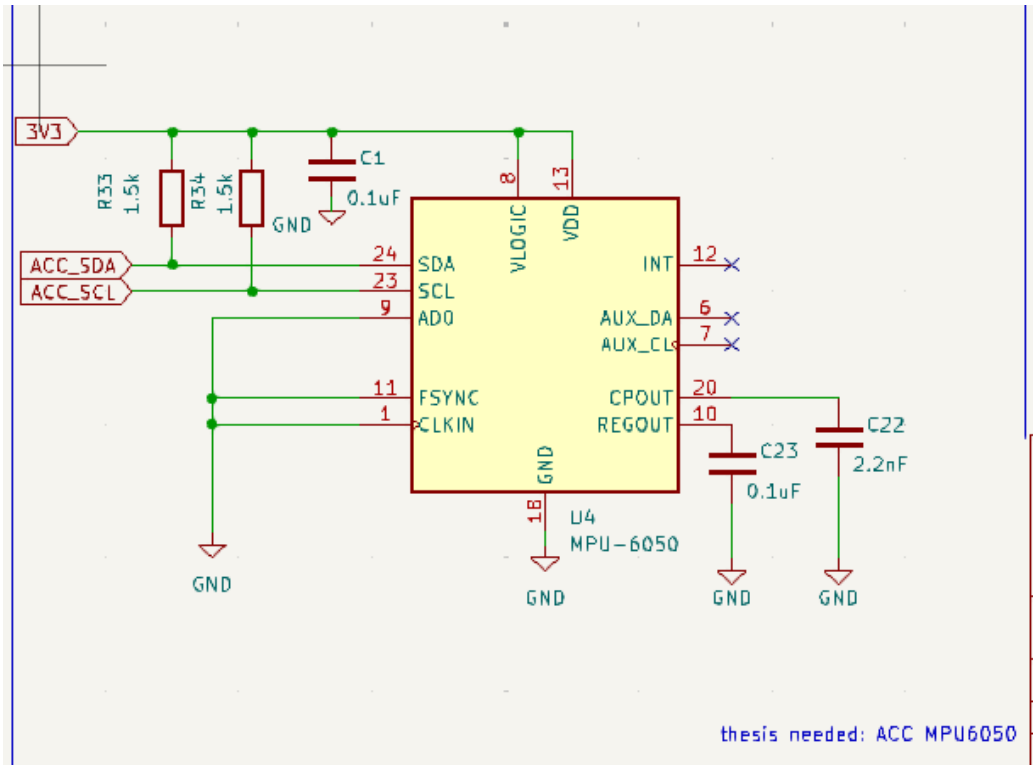


Figure 1.10: Accelerometer schematic

This schematic is the reference design proposed in the datasheet by the manufacturer of the accelerometer. There are also designs that can be used, but this one is specifically utilized in the context of I2C communication. The accelerometer as said in the previous section will be connected with the ESP32 microcontroller via the I2C bus. The power supply is 3.3V and the logic levels are also 3.3V, which makes it compatible with ESP32.

Two pull-up resistors R33 and R34 are used to pull the SDA and SCL lines to the power supply voltage. The ESP32 could ensure the pull-up, however, it has a weak pull-up, which could be problematic in the case when the copper traces SDA and SCL are too long. The pull-up resistors are used to ensure that the signal is not distorted by the noise.

The PCB design of the accelerometer can be seen in the Figure 1.11. All the components are placed on the top layer of the PCB, and are centered around the accelerometer package itself. The accelerometer is also centered on the PCB, such that to have the best reading of the lateral movement as well as the pivotal movement.

In order to test the accelerometer, the I2C communication lines do have test pads on the bottom copper layer on the PCB. They can be seen in the Figure 1.12.

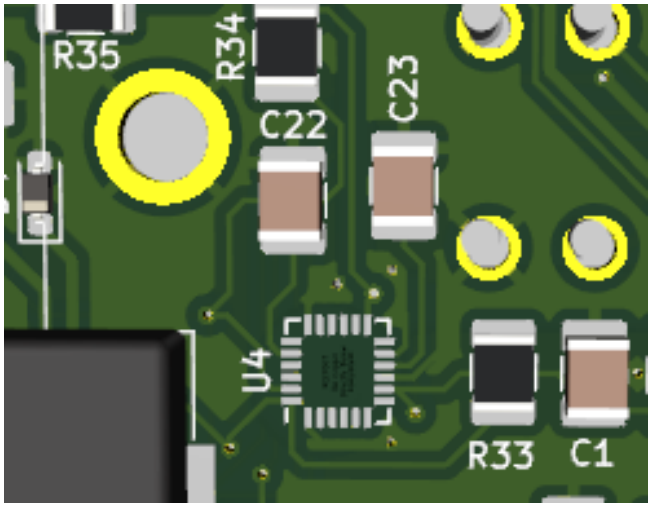


Figure 1.11: Accelerometer physical implementation on the PCB

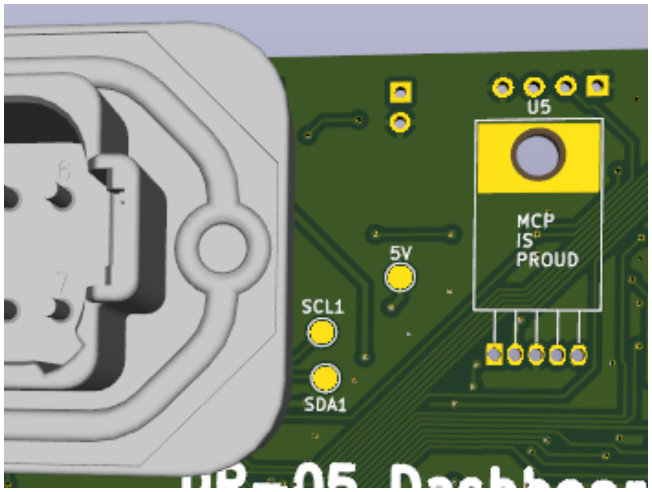


Figure 1.12: Accelerometer test pads for I2C communication

1.13 PCB implementation of the SD card slot

In order to store the data that is read from the CAN bus lines, the ESP32 microcontroller, needs to have an external memory. The SD card is a good choice for this, since it is cheap and has a good storage capacity. In the ESP-IDF examples, there is a driver for the SD card, in which, it states how the SD card should be connected to the microcontroller. The SPI protocol lines are connected to GPIO 4, 13, 14 and 15, which are the MISO, MOSI, CLK and CS lines.

The schematic of the SD card slot can be seen in the Figure 1.13. In order to understand how the SD card works, one needs can go to the official website of the SD card association, where the specifications of the SD card are detailed. [12]. The connector, however, does not have an explicit schematic of how the pins are mapped to the SPI lines, therefore one would need to reffer to the documentation that links the SDIO lines to the SPI lines. The symbol of the connector depicts only the SDIO lines. DAT0 is the MISO line, DAT3 the CS line, MOSI is the CMD line and the CLK line si the CLK line in the case of SPI. As stated by the ESP-IDF documentation, the CS, MOSI and MISO lines are connected via pull-up resistors to the 3.3V power supply. The C17 and C18 capacitors are used to decouple the power supply of the SD card.

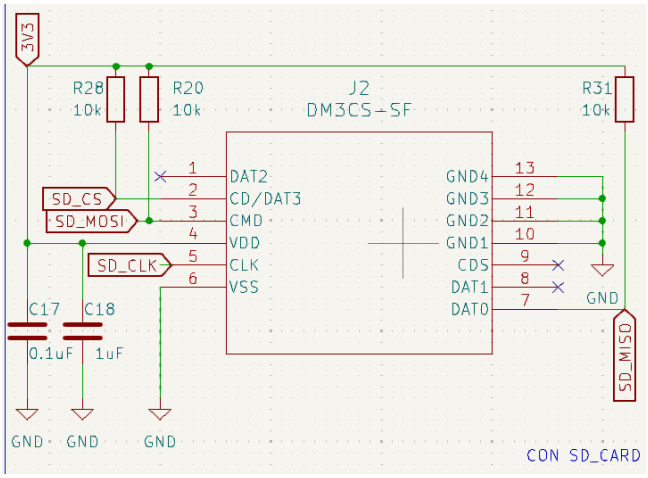


Figure 1.13: Accelerometer test pads for I2C communication

The physical implementation of the SD card slot can be referred in the Figure 1.14. This slot is also placed on the top layer of the PCB, near the LCD connector. It is a hinged type slot, that resists vibrations and shocks. The SD card is inserted in the slot and then it is closed by sliding the hinged part. The capacitors as well as the resistors are placed in the upper part of the figure, near the accelerometer.

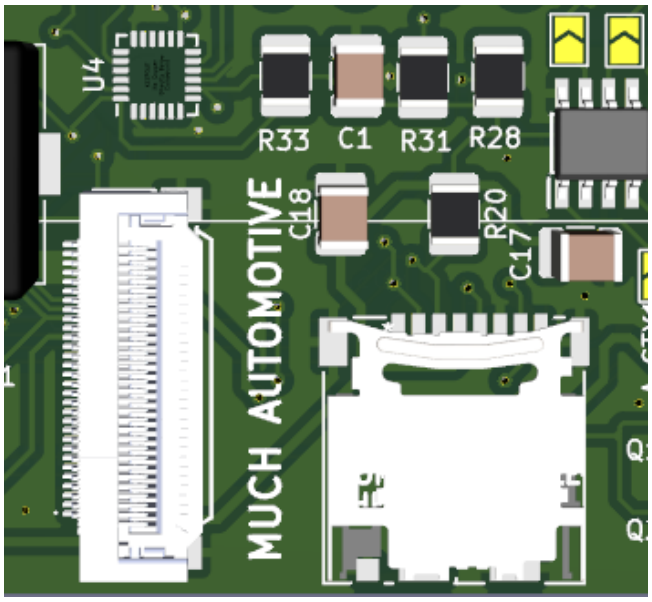


Figure 1.14: SD card slot physical implementation on the PCB

1.14 PCB implementation of the LCD connector

Coming to the PCB implementation of the LCD connector, it was a tricky part of the project, because of the small pins of the connector, it is very hard to solder them to the PCB. Now, it is worth noting that the PCB is a 2 layer one, most of the components being implemented on the top layer of the PCB. This is the case of the LCD connector, which is also implemented on the top layer. The decision was made based on the cable that will connect the LCD with the PCB [13], which is a 30 pin ribbon that could not be bent in a 90 degrees angle, because it would break. Therefore, it was needed for the connector to be oriented in the same direction as the one on the LCD module and to be as close as possible to it in order to avoid any kind of bending.

The schematic of the LCD connector is detailed in the Figure 1.15. It includes the UART communication lines, the power supply lines and the ground pins, as well as a reset pin.

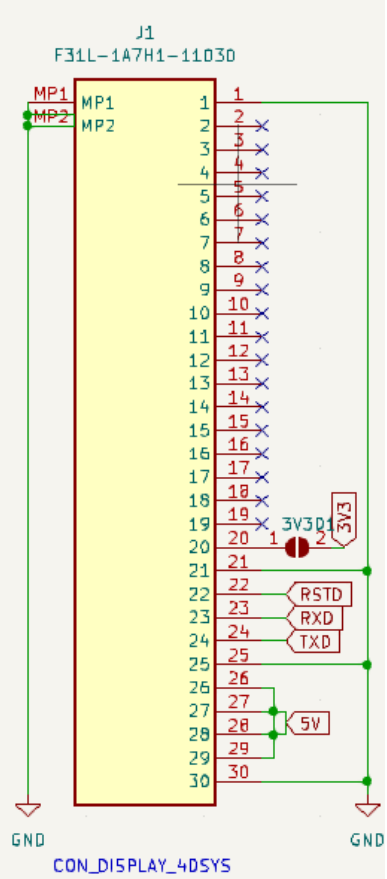


Figure 1.15: LCD connector schematic

The pinout of the connector is detailed on the 4DSystems website. As said before, the ESP32 microcontroller will communicate with the LCD via the TXD and RXD pins, which are the UART communication pins. The ESP32 will send the data to be displayed on the LCD via the TXD pin, and the LCD will send the acknowledge signal back to the ESP32 via the RXD pin. The power supply pins are the 5V pins and the ground pins. The reset pin is used to reset the LCD in case of a software bug, which is a good feature to have, since the software is not open source and the team cannot modify it in case of a bug. It is also worth noting that the LCD can exhibit a freeze state if the ESP32 is not powered at the same time as the LCD, this is why the reset pin is needed.

Also, the display has a built in LDO source that can be used in case that the LDO used for powering the ESP32, integrated on the PCB fails. This is why another 3.3V pin is shown in the figure, that being the 3.3V output of the LCD LDO. It can be either connected or disconnected from the board by having a solder bridge on the PCB.

Another aspect of this connector is the fact that the UART communication lines for the ESP32 cannot be the same used to program the microcontroller. The ESP32 chip has 2 UART IPs, one for programming the chip and for communication and another one just for communication. The progamming UART is UART 1 and the one used to communicate with the LCD is UART 2. Therefore TXD and RXD pins will go to the UART 2 pins on the ESP32 microcontroller, in order to avoid conflicts between the programming and the communication with the LCD. During the prototyping phase, I noticed that the ESP32 could not be flashed with another program if it was connected to the LCD via UART 1. In the Figure 1.16 is exhibited the physical implementation on the PCB of the 30 pin connector.

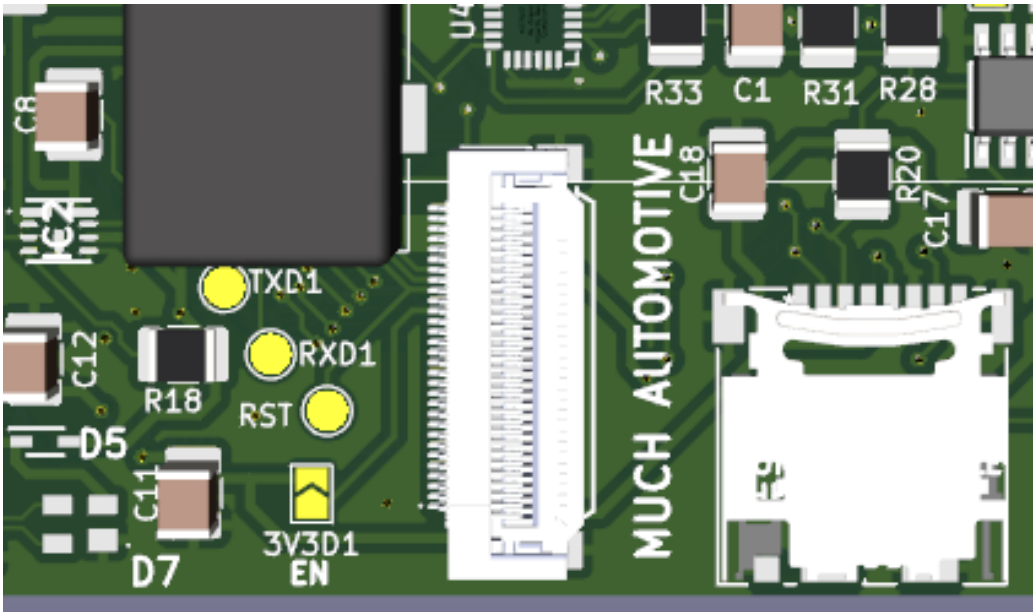


Figure 1.16: LCD connector on the PCB

1.15 PCB implementation of the automotive connector

The connector used is a 12 pin automotive one from TE electronics. The PCB symbol of it was obtained from the Mouser website and then imported in KiCAD EDA. The schematic of the connector is detailed in the Figure 1.17. The pins of the connector contain the CAN H and CAN L pins that are used to communicate with the car's CAN bus. On the bus will be the ECU which is sending the data to the dashboard. The power supply pins are also included in the connector, the 12V and the ground pins. A protection diode was connected in parallel with the 12V pin in order to protect the circuit from the over voltage spikes.

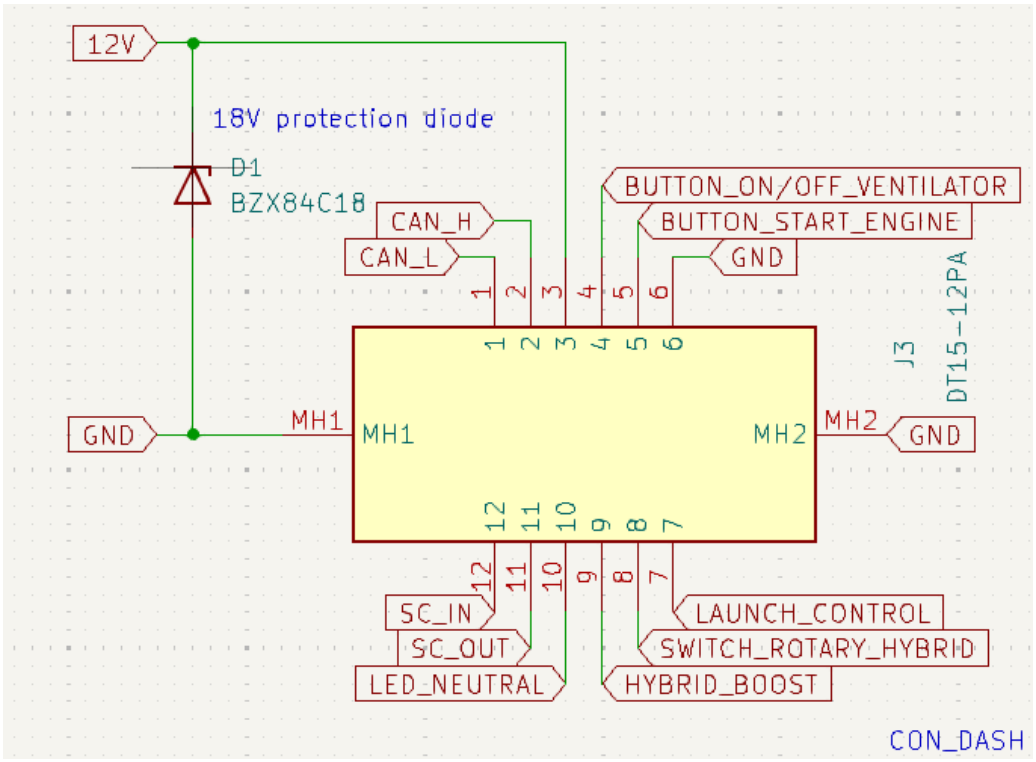


Figure 1.17: FS car connector schematic

Besides them, the connector includes a button to switch on and off the car's radiator fan, which is a feature that was needed to be implemented in order to avoid overheating while the car is stationary. The button is not connected to any of the ESP32 pins.

The start engine pin is also included in the connector, which is used to start the engine of the car. The pin is also not connected to any of the ESP32 pins. It is just connected to a button that will be pressed by the pilot in order to start the engine.

Launch control pin is an ON/OFF pin that is used to enable or disable the launch control of the Formula Student car. The pin is not connected to any of the ESP32 pins. It just has a pull-up resistor connected to it, such that it is adapted to the design imposed by the ECU reference manual.

The switch rotary hybrid is pin is an additional control pin that is used by the dashboard PCB to send the position of the rotary switch to the Hybrid module of the Formula Student car. The rotary switch is a 12 position switch that is used to set select between 11 different voltage dividers. So the hybrid selector will have 11 voltage levels that can be selected from.

The hybrid boost pin is an ON/OFF pin that is used to enable or disable the electric boost of the Formula Student car. The pin is not connected to any of the ESP32 pins.

The LED Neutral pin is used as a ground pin for the Neutral LED that is implemented on the dashboard. The LED is used to show the pilot that the car is in neutral gear. The grounding effect comes from coupling the gear fork with the engine as the car is in neutral, the engine being also grounded electrically.

Also, the safety circuit in and safety circuit (SC IN and SC OUT) pins are used for the cars safety system that allows for ignition cutoff in case of emergency.

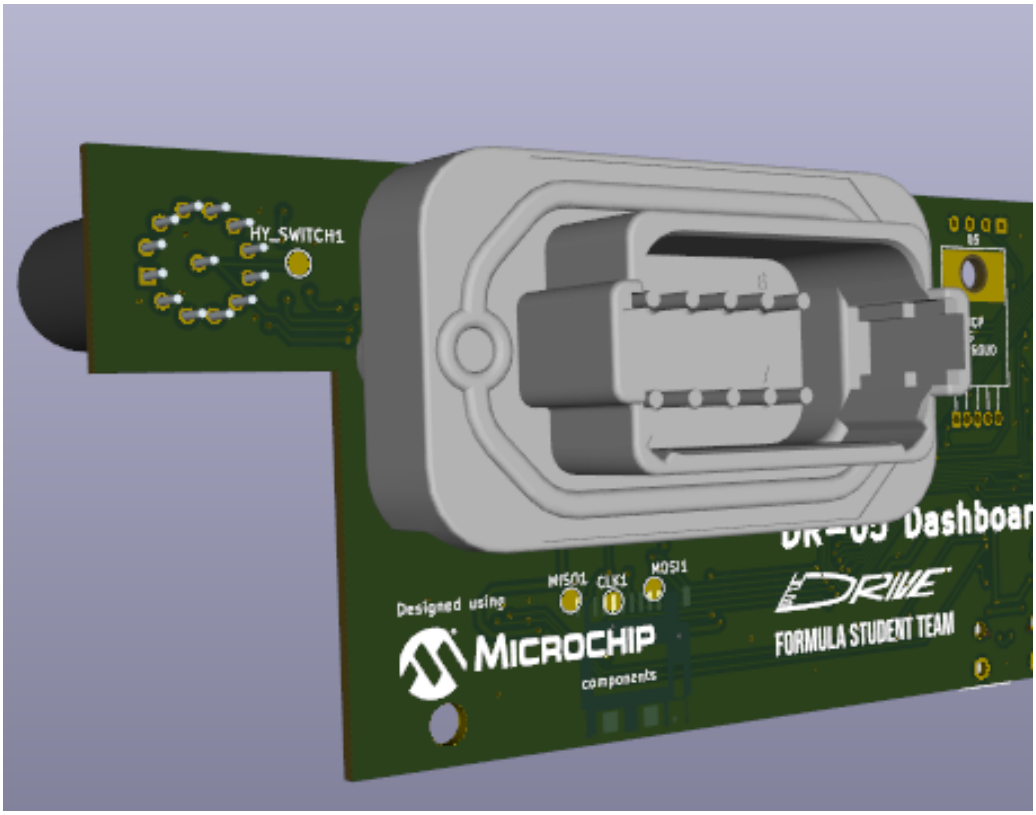


Figure 1.18: FS car connector physical implementation on the PCB

Chapter 2
Software component

2.1 Software tools - VS Code editor and ESP-IDF Framework

For this project I have used the Visual Studio Code editor to write the C code of the ESP32 microcontroller as well as the C/C++ code for Raspberry Pi Pico microcontroller. ESP32 is used in the dashboard module, whereas the Pi Pico is used in the testing module that is connected via the CAN bus to the Dashboard module in order to showcase the functionality of it.

Visual Studio Code is a text editor that integrates many functionalities. By default, it has no more features than Notepad, however it has the ability to integrate extensions. These extensions can be of many kinds, but the ones that I have used relate to the C/C++ software component of this project. The C/C++ extension was used extensively because it highlights the C/C++ code as seen in Figure 2.1. The keywords highlighted in a violet color, macros are blue, the variables are cyan, and function names are yellow. In this way, the color coding allows the developer to understand the code faster and develop faster.

```
57  ////////////////////////////////////////////////////////////////////////////
58  ////                          LOCAL FUNCTIONS                           ////
59  ////////////////////////////////////////////////////////////////////////////
60
61  esp_err_t dashboard_init()
62  {
63      esp_err_t esp_response          = ESP_FAIL;
64      twai_mode_t can_mode            = TWAI_MODE_NORMAL;
65
66      #if (CAN_SPEED == 250)
67      twai_timing_config_t can_speed  = TWAI_TIMING_CONFIG_250KBITS();
68      #elif (CAN_SPEED == 500)
69      twai_timing_config_t can_speed  = {.clk_src = TWAI_CLK_SRC_APB, .quanta_res
70      #elif (CAN_SPEED == 800)
71      twai_timing_config_t can_speed  = TWAI_TIMING_CONFIG_800KBITS();
72      #elif (CAN_SPEED == 1000)
73      twai_timing_config_t can_speed  = TWAI_TIMING_CONFIG_1MBITS();
74      #endif /* CAN_SPEED */
75
76      twai_filter_config_t packet_filter_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();
77
78      /* setup can bus */
79      esp_response = can_setup(can_mode, &can_speed, &packet_filter_config);
80      if (esp_response != ESP_OK)
81      {
82          return esp_response;
83      }
```

Figure 2.1: Example of highlighting in Visual Studio Code

The functionality of C/C++ extension does not stop here, though, it allows for references in code. This means that the developer can use a shortcut CTRL + Left Click on any macro and then the editor will switch to the file in which the macro was defined, or if it was defined in the same file, it will scroll to the part of the file where the macro was defined. It behaves the

same in the case of function definitions and function declaration. The developer can choose to switch between definition and declaration to easily search for the function bodies in order to understand what the function does.

Another extension that was used extensively for this project is the ESP-IDF one. This one allows for easier download and integration of the ESP sdk into the project. At first, it prompts the developer with a setup section, in which one can choose to opt for a particular version of the ESP-IDF. For this project I have used ESP-IDF v5.1 for the whole C code that is on the ESP32 microcontroller. This is important because there are changes in the libraries that I have used in other versions of it. For example, in the case of the I2C communication, there are many changes that have occurred to the driver. The Programming Guide of the release/v5.1 was used to understand the APIs of the functions that were used in the project [14]

The ESP-IDF extension allows for the developer to create a new project, to build it and to flash it on the ESP32 microcontroller directly from the VS Code editor. This is a powerful feature, since all the development can be done in the same environment, making VS code an IDE type of software, and not just a text editor.

ESP-IDF is a framework with the help of which the developer can write the code in C faster for the ESP32 microcontroller. It includes many libraries that make the hardware abstract to the developer. A good example is the fact that the APIs of the ESP-IDF are not using any register level programming, but are using functions that are easy to understand and use. ESP-IDF also includes many examples that can be used as a framework for the developer to write the code. ESP-IDF is hosted on GitHub, and it is open source, having an Apache-2.0 license. Also examples of the ESP-IDF code that enables the peripherals are hosted on GitHub, and be accessed by the developer in order to understand how the peripherals are enabled and used in the code. [15]

The IDF was used in order to program all the features in this project, this meaning the analog to digital converter (ADC), the controlled area network (CAN) communication, the GPIO control and the interruptions generated by the GPIO pins, the I2C communication, the UART communication, the SPI communication and the Wi-Fi communication via the HTTP server. All the features have an initialization function in which all the parameters are set in order to have the desired functionality. IDF has a RTOS kernel integrated into it, namely the FreeRTOS kernel. This is a real time operating system that allows for the developer to write the code in a more structured way. It also allows the programmer to enable multiprocessing and multi threading in the code, which makes it easier to synchronize certain aspects and parts of the code. A clear example in this project is the fact that the function that is responsible with sending data to the display is running separately than the one that is responsible for reading the CAN bus. Ideally the CAN bus reading happens before the data is sent to the display. However, this can be excepted since the functions are executed very fast, therefore for the user it seems that they are running in parallel.

The RTOS kernel can be used to also stop race conditions from happening, since it integrates data structures that act as a mutex. They are called Semaphores and are available in the IDF. In this project a Semaphore was used to check if the data that is sent to the display is also used by the CAN bus reading function. The problem is that if the data is updated and read at the same time, it may create inconsistencies, which will seem to the user as flickering of the display, or worse, it could create a false trigger of the error messages shown on the display, that are meant to protect the pilot from a failing engine. Therefore, the Semaphore is used to lock the data when it is used by the function that sends it to the display, then it is unlocked and locked by the function that executes CAN read. This way, the data is consistent.

Since multi threading is also a feature of the IDF, the project makes use of it. ESP32 has 2 processing cores, therefore the functions that are responsible with the main features of the code are split between them in order to have a faster processing time and a more responsive

dashboard.

2.2 Software tools - 4DSytems Workshop 4 IDE

2.3 Dashboard software file structure

In the Figure 2.2 is shown the file structure of the ESP-IDF project that was used in order to program the ESP32 microcontroller embedded in the dashboard electronic module. One can see that the Firmware directory contain other 5 directories: .vscode, 4dsystems-project, components, main, misc and a .gitignore file, the main CMakeLists.txt of the project, a README.md file and a sdkconfig file. The purpose of the components' directory is to split the source files of the project into directories with the name of the peripherals that are used for that part of the source code. For example in the adc folder one will find code that makes use mostly of the ADC peripheral of the ESP32 microcontroller. The same can be said about the can folder, in which reside the source files that make use of the CAN peripheral and so on. In this way, the programmer can easily understand the scope of the source files that reside in the folder and then easily find the source code that needs to be changed or debugged.

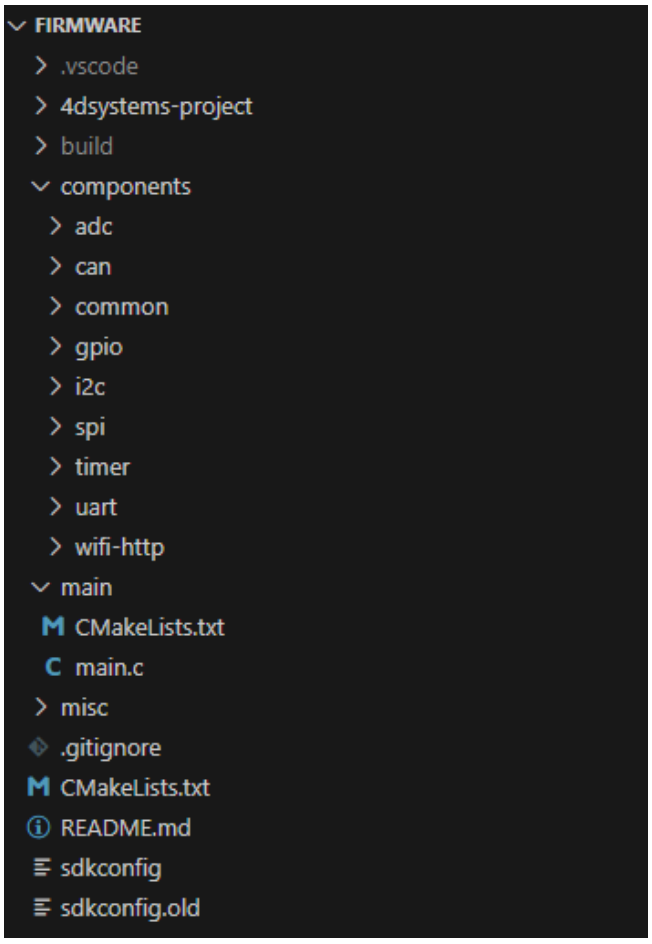


Figure 2.2: File structure of the dashboard software project

The challenge of this way of splitting the project is to understand which components of the project, for example the LCD, makes use of which ESP32 peripheral. For this, I have written Table 2.1 that explains exactly which peripheral is used for which part of the project.

_	ESP32 peripheral	Project feature
	Analog to digital converter (ADC)	Rotary selector that choses between 11 voltage levels in order to define the state for the hybrid module
	Controlled Area Network (CAN)	CAN bus communication via the SN65HVD230DR transciever with the ECU of the FS car
	General Purpose Input Output (GPIO)	Button read, for example the button used to change the page of the LCD
	Inter-Integrated Circuit (I2C)	Read the data from the accelerometer included in the dashboard module
	Serial Peripheral Interface (SPI)	Read and write data on the SD card module included in the dashboard module
	Universal Asynchronous Receiver Transmitter (UART)	Send data to the LCD display and read the acknowledgment or error signal
	Wi-Fi and HTTP	Download the data stored in the SD card via the ESP32 Wi-Fi access point

Table 2.1: ESP32 peripherals used for the features of the project

Coming back to the ESP-IDF project, it has a more complex structure. In the CMakeLists.txt, from the root folder, are declared the other components of the project, "adc", "can", "common", by using the relative path from the root.

In the main folder resides the main.c source file, in which is declared and defined the main function of the code. Besides it, there is another CMakeLists.txt file in which are declared the required components that need to be compiled before the main.c file in order for it to successfully compile. This folder is required as per the IDF framework project structure requirements. When trying to build the project, the IDF framework will look for the main folder of the project. If it does not find it, it will throw an error.

The "components" folder is also a required folder by the IDF framework, and it usually contains other directories in which reside the source files of the functions used in the main.c file. The structure of any subdirectory of the "components" directory is similar to the main directory. Let's take the example of the "adc" folder. In it resides a source file with the source code that makes use of the adc peripheral, a CMakeLists.txt file that declares the source file and it's prerequisites and an "inc" folder in which resides the header files that is used in the project to include the declarations of the functions which are defined in the source file.

The "4dsystems-project" directory is used to store the project of the interface that is shown on the LCD module. The project is developed in the 4D Systems Workshop 4 IDE, which is a software that allows for the development of the interface by dragging and dropping certain parts of the interface on a mock LCD screen. The project is then exported as a .4dct file, which is a binary file that contains the interface. This file is then copied to an SD card module that is included in the LCD module. After a powen on reset, the LCD module will read the file from the SD card and flash it in the memory, and then show the new interface on the screen.

The "misc" directory is used to store the data related to the README.md file, such as screenshots, that can be referenced in the README. In this way, the project repository is more organized and the developer can easily find the data that is searching for.

The ".vscode" directory is used to store the settings of the Visual Studio Code editor. This is useful in the case that the developer wants to share the project with other developers, such that they can have the same settings as the original developer. The settings are stored in a JSON file, which is a text file that contains the settings of the editor.

2.4 RTOS kernel and multi-threading

ESP-IDF is a framework developed and maintained by the Espressif company. It is a software tool that was heavily used to develop the software that runs on the ESP32 chip, which is integrated in the dashboard module. The framework is open source and is adapted to the hardware of ESP32 microcontroller which is also developed by Espressif. The ESP-IDF is based on the FreeRTOS kernel, which is a real time operating system kernel that integrated many features that are useful for the applications that need to run in real time [16]. The RTOS concepts are the following: the code is split into functions that are executed in parallel. This functions are called tasks. In order to synchronize tasks, one can use semaphores. A semaphore is a data structure that is used to lock a resource when it is used by a task, and then unlock it when the task is done with it.

The multi-threading concept comes from the fact that tasks can be run on different cores of the microcontroller. The ESP32 microcontroller has 2 cores, therefore the developer can split the tasks between the cores in order to have a faster processing time. However, by doing so, one can run into issues such as race conditions and deadlocks. This will be explained in the following paragraphs. Coming back to tasks and multi-threading, this project has multiple functions, each one of them being responsible for a certain feature of the code. These functions are the tasks of the RTOS kernel. Each task has a while loop inside it that runs indifinetly, but, after each run, it is paused for a certain amount of time. One can already identify a problem here, if the while loop does not get stopped, only one task will run indifinetly. However, there is a certain feature of the RTOS kernel that will stop this from happening, namely preemption. In ESP-IDF flavor of FreeRTOS, the preemption is enabled, that meaning that if the task exceeds a certain time threshold, namely the CPU quanta, the task will be paused and another higher priority task will run for a time quanta. This is called preemptive scheduling, and it is depicted in Figure 2.3.

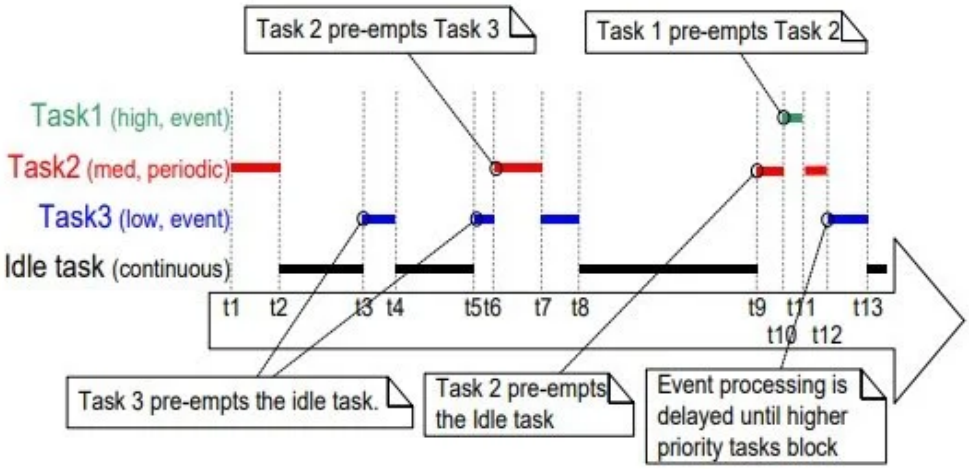


Figure 2.3: Preemptive scheduling

From the figure, it's hard to determine which is the time quanta allocation, but it is a good representation of the fact that the higher priority tasks will preempt the lower priority tasks in order to run. This is why the kernel is called "real time", because it can guarantee that the tasks will run in a certain amount of time, given that they have high priority. This project makes of this feature in order to have real time CAN bus communication as well as real time data communication with the LCD.

Because these two tasks are of high priority, they are run on different cores, however, they

can share data via the shared SRAM memory between the cores. The block diagram of the ESP32 microcontroller can be seen in the Figure 2.4, and it depicts the shared RAM and SRAM memory of the 2 Xtensa LX6 micorprocessors. This is a good architectural feature of the microcontroller, because the data can be shared in a faster manner between the cores than in the case of cores having separate SRAM and ROM memory.

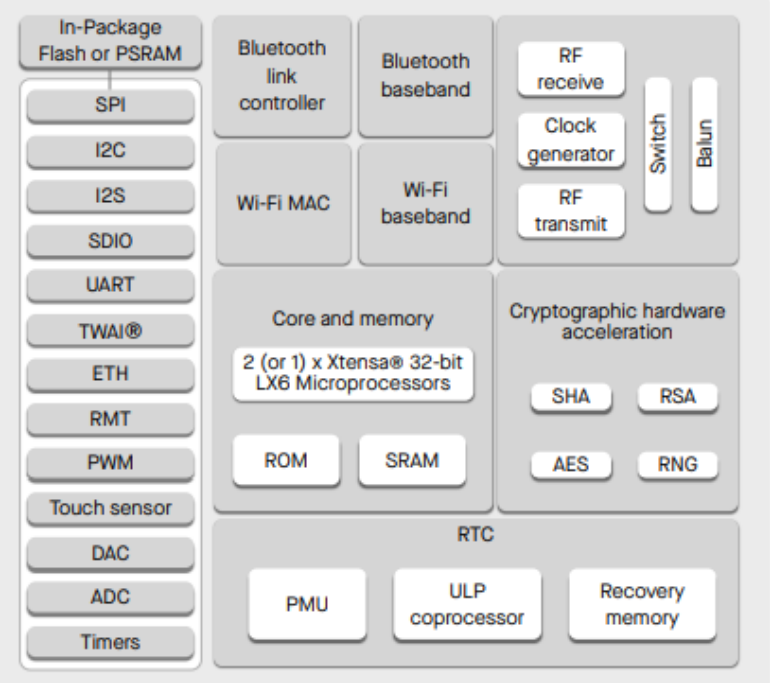


Figure 2.4: ESP32 core block diagram

In the main function of the code, the tasks are created and they are automatically started by the RTOS kernel. The tasks are created with the xTaskCreatePinnedToCore function, which takes as arguments the name of the task, the function that is the task, the parameters that are passed to the task, the stack size of the task, the priority of the task and the handle of the task. In the case of this project, the tasks have no handle, because they are not stopped or paused in any moment of time.

2.5 Common header macros and data structure shared between cores

In order to better understand the way that the cores share data between them, it is important to see the data structure used to store the data that is read from the CAN bus. This data structure is of type display_data_t and is defined in the common.h header that is included in all the source files of the project. This header is intended to be a common way of handling the way the code behaves. In it are defined macros that are used to define certain settings of the code such as:

- the shift threshold of the RPM LED bar by #define SHIFT_THRESHOLD
- the number used for the neutral gear by #define NEUTRAL_GEAR
- the LCD refresh rate by #define LCD_REFRESH_RATE
- the LCD start up screen delay by #define LCD_STARTUP_SCREEN_DELAY
- the LCD communication baudrate by #define LCD_BAUDRATE
- the hybrid selector polling rate by #define HYBRID_SELECTOR_POLLING_RATE

- the accelerometer polling rate by #define MPU6050_POLLING_RATE
- the accelerometer scale by #define MPU6050_ACCEL_SCALE
- the CAN bus speed by #define CAN_SPEED
- the SDCARD_LOGGING_RATE by #define SDCARD_LOGGING_RATE
- the AP name by #define ESP_WIFI_SSID
- the AP password by #define ESP_WIFI_PASSWORD

Besides them, there are pins definition in order to give meaning to the pins numbers that are used in the code. For example, GPIO_NUM_4 is redefined as GPIO_SDCARD_MISO in order be clear about the purpose of the pin. These redefinitons are based on the ESP32 pinout as well as the schematic of the dashboard module that is depicted in the Figure 1.5. In the following code snippet is depicted the data structure that is shared between the cores of the ESP32 microcontroller, and , in which, the data that is read from the CAN bus is stored and then sent to the LCD module.

```
typedef struct display_data_t {

 /* Keeps track of the current_page shown on the display */
 uint8_t current_page;


 /*************************************************************
  *                    MAIN DISPLAY PAGE                      *
  *************************************************************/
 /* Expected rpm is between 0 - 12.000 (uint16_t max 65.536) */
 uint16_t rpm;

 /* Current gear is between 0 - 5 (uint8_t max 255) */
 uint8_t current_gear;

 /* Coolant temperature is between 0 - 100 (uint8_t max 255) */
 uint8_t coolant_temperature;

 /* Radiator Fan state is either true for ON or false for OFF */
 uint8_t fan_state;

 /* Oil temperature value is between 0 - 150 (uint8_t max 255) */
 uint8_t oil_temperature;


 /*************************************************************
  *                SECOND DISPLAY PAGE for debug              *
  *************************************************************/
 /* CAN bus status is 0x00 for ON or 0xFF for OFF */
 uint8_t can_status;

 /* Hybrid system status is 0x00 for OK or 0xFF for FAULT */
 uint8_t hybrid_status;

 /* Safety circuit status is 0x00 for OK or 0xFF for FAULT */
 uint8_t safety_circuit_status;

 /* Oil pressure value is between 0 - 150 (uint8_t max 255) */
 uint8_t oil_pressure;

 /* Raw bps value from the sensor ToDo */
 uint8_t brake_pressure_raw;

 /* Throttle Position Sensor value is between 0 - 100 (uint8_t max 255) */
```

```
45  uint8_t tps;
46
47  /* Oil temperature value is between 0 - 150 (uint8_t max 255)
48
49     @note: the field oil_temperature is scaled before being sent to MAIN_PAGE
50            this field will contain the unscaled version of oil_temperature from
51            the CAN message
52  */
53  uint8_t oil_temperature_2;
54
55  /* Hybrid status value HYBRID_STATE_1 - HYBRID_STATE_11 or HYBRID_STATE_ERROR
      */
56  uint8_t hybrid_selector_value;
57
58  /* Battery voltage is between 0 - 240 (uint8_t max 255) */
59  uint8_t battery_voltage;
60
61
62  /*********************************************************
63  *                  THIRD DISPLAY PAGE for debug          *
64  *********************************************************/
65  /* Lambda value between 0 - 155 (uint8_t max 255) */
66  uint8_t lambda;
67
68  /* Manifold Absolute Pressure MAP value 0 - 45 (uint8_t max 255) */
69  uint8_t map;
70
71  /* Fuel pressure value is between 0 - 45 (uint8_t max 255) */
72  uint8_t fuel_pressure;
73
74  /* EGT values between 0 - 450 (uint8_t max 255) */
75  uint16_t egt[4];
76
77  /* Voltage on the PMU input rail 0 - 14 (uint8_t max 255) */
78  uint8_t input_voltage_pmu;
79
80  /* Input current from alternator 0 - 10 (uint8_t max 255) */
81  uint8_t input_current_altr_pmu;
82
83
84  /*********************************************************
85   *              FOURTH DISPLAY PAGE for debug            *
86   *********************************************************/
87  /* Tyre pressure values for each wheel between 0 - 30 bar (uint8_t max 255)*/
88  uint8_t tyre_pressure[4];
89
90
91  /*********************************************************
92   *                     NOT IN A PAGE                     *
93   *********************************************************/
94
95  /* Raw accelerometer value on X, Y and Z, note that the max value is
96     adjusted considering the MPU6050_ACCEL_SCALE macro. The formula is
97     ((accelerometer[i] / 2^15) * MPU6050_ACCEL_SCALE). It is divided to
98     2^15 since the values are integer ones on 16 bit, so the max value of
99     int16_t is 2^15 */
100 int16_t accelerometer[MPU6050_AXIS_NUM];
101
102 /* Intake Air Temperature (IAT) is between 0 - 100 (uint8_t max 255) */
103 uint8_t iat;
```

```
104
105  /* This is a byte dedicated to error flags.
106  Bits in error flags:
107  [0]: coolant_temperature higher than 100 degrees
108  [1]: oil_temperature higher than 150 degrees
109  [2]: low oil pressure error
110  [3]: hybrid_status unknown
111  [4]: safety_circuit_status unknown
112  [5]: can_status unknown
113  [6]: tps application is higher than 100 %
114  */
115  uint8_t error_flags;
116 } display_data_t ;
```

2.6 CAN bus read task

This task from the code is responsible for reading the data from the CAN bus as well as to send data from the dashboard to other electronic modules through the CAN bus of the FS car. The CAN bus is a differential signaling protocol that is used in the automotive industry in order to communicate between the electronic modules of the car. The way it behaves was defined in the CAN specification from Bosch [17].

It makes use of the TWAI peripheral of the ESP32 microcontroller. A particularly funny story about this peripheral is that it was not named CAN, but is CAN. The reason for this is that the CAN bus is a trademark of Bosch, and the Espressif company could not use the name in the code. Therefore, the name of the peripheral is TWAI, which stands for Two Wire Automotive Interface. In the case of each task, the variables are declared in initialized with 0 or FAIL values in order to prevent the task from running with garbage data. This is a good practice for safety in C programming. A problem of this TWAI IP is that it's default timing settings do not match the official CAN bus timing settings that were cited in the spectification sheet from Bosch, therefore a calculator was needed to determine the actual bit time needed to be configured for the ESP32 to behave correctly on the CAN bus [18].

In order to understand what the task does, it's important to see the following code snippet. The snippet is taken from the task that reads can packets and it depicts the functionality of the task, namely the reading of the CAN bus packets and then the processing of the packets into a data structure that is used to store it and then to send it to the LCD via another task.

```
1
2   /* Receive a message from the CAN bus */
3   esp_response = twai_receive(&message, CAN_POLLING_RATE);
4
5   /* If the message was succesfully receiver, then process it */
6   if (esp_response == ESP_OK)
7   {
8     ...
9
10    if( xSemaphoreTake( xSemaphore_display_data, ( TickType_t ) 10 ) == pdTRUE )
11    {
12        /* Update display_data structure based on the CAN packet id */
13        switch(message.identifier)
14        {
15            case CAN_PACKET_ECU_1:
16            {
17                display_data->rpm                = (uint16_t)message.data[0] *
    100U;
18                display_data->current_gear       = message.data[1];
```

```
                display_data->tps                  = message.data[2];
                display_data->oil_pressure         = message.data[3];
                display_data->coolant_temperature  = message.data[4];
                display_data->fuel_pressure        = message.data[5];
                display_data->lambda               = message.data[6];
                display_data->iat                  = message.data[7];
                break;
            }

            ...

            default:
            {
              break;
            }

        }

        xSemaphoreGive( xSemaphore_display_data );

        ...

        else if (esp_response == ESP_ERR_TIMEOUT)
        {
            /* No packet was received, increment the fault counter */
            counter_no_packet_recieved = counter_no_packet_recieved + 1;

            #ifdef ENABLE_CAN_READ_DEBUG
            printf("Timeout occurred, no message received\n");
            #endif /* ENABLE_CAN_READ_DEBUG */
        }

       ...

        /* If more than 200 packets were missed, then signal CAN bus FAULT */
        if (counter_no_packet_recieved == 200U )
        {
            display_data->can_status    = false;
            counter_no_packet_recieved = 200U;
        }

        vTaskDelay(CAN_POLLING_RATE);
      }
  }
...
```

In order to avoid race conditions on the data structure that is used to store the display data, a semaphore is used. The semaphore is taken before the data is updated and then it is given after the data is stored in the structure. The semaphore is also a binary one, meaning that when it is taken by a task, it cannot be taken by any other task until it is given back by the task that took it first. So in any part of the code in which this data structure is used, the semaphore is checked if it can be taken.

Another feature of the CAN bus communication task, is that it has a mechanism that is used to detect if the bus is faulty. This is done by incrementing a counter each time a packet is not received, and then checking if the counter is equal to a certain value. If it is, then the CAN bus is considered faulty and the data structure is updated with the fault status. This is a safety feature that is used for debugging purposes.

The vTaskDelay function used at the end of the task is used to put the task in a blocked

state for a certain amount of time. The amount of time is given in ticks, and it is calculated based on the RTOS kernel tick rate. The macro CAN_POLLING_RATE is used to define the amount of ticks that the can read task is blocked. That macro is defined in the common.h file, in a part of code that is used to define certain constants in order to have a way of changing them easily and also to change the way in which the code behaves.

2.7 CAN bus write task

In order to send packets on the CAN bus, there is another task that is used in for this process. The following code snippet is taken from it.

```
...
if( xSemaphoreTake( xSemaphore_display_data, ( TickType_t ) 10 ) == pdTRUE )
        {
            /* Send the data to the CAN bus */
            message.identifier       = CAN_PACKET_DASH;
            message.data_length_code = 1;
            message.data[0]          = display_data->hybrid_selector_value;
            message.rtr              = 0;
            message.flags            = 0;
            xSemaphoreGive( xSemaphore_display_data );
        }

        esp_response = twai_transmit(&message, CAN_TRANSMIT_RATE);
        if(esp_response != ESP_OK)
        {
            #ifdef ENABLE_CAN_SEND_DEBUG
            printf("Failed to send message with id 0x384\n");
            #endif /* ENABLE_CAN_SEND_DEBUG */
        }
        else
        {
            #ifdef ENABLE_CAN_SEND_DEBUG
            printf("Message with id 0x384 was sent\n");
            #endif /* ENABLE_CAN_SEND_DEBUG */
        }

        vTaskDelay(CAN_TRANSMIT_RATE);
    ...
```

One can see that a similar function from the twai library was used to send the data to the CAN bus. At first the message structure is filled with the data that is stored in the common display_data structure. One can see that the semaphore is checked before accessing it in order to avoid garbage data to be sent on the bus.

After that, the semaphore can be given back, and the message is sent on the CAN bus. If debug mode is enabled, a message is printed in the console in order to show the status of the message transmission.

In the end, the vTaskDelay function is used to block the task for a CAN_TRANSMIT_RATE amount of time. This macro is defined in the common.h file, which is used as an aggregator of the constants that are used in the code.

2.8 SD card write task

This task makes use of the SPI peripheral of the ESP32 microcontroller. It is run on the same core as the LCD update task, namely CORE 1 of the ESP32 microcontroller. The peripheral

is first initialized in the "sdcard_setup" function, in which the SD card is also mounted to the file system. The SD card is used to store the data that is read from the CAN bus, as well as the data that is read from the accelerometer in order to have long term data storage. The telemetry data is stored on it in a file called "LOG.txt" in the root directory. The same file is used by the Wi-Fi module in order to send the data to the user via HTTP request.

The SD SPI Host Driver from ESP-IDF is used in order to access the SD card as a SD SPI device. This is important to mention since the SD card can also be accessed with the SDIO driver, which has 4 data lines, intstead of just 2 in the case of SPI. It may seem that the SDIO driver will have a higher throughput, but in the reality, it is not the case, since it depends the most on the SD card speed rating. The SPI driver is used because it is easier to implement.

The "sdspi_device_config_t" structure is used to configure the SPI peripheral. SDSPI_DEVICE_CONFIG_DEFAULT macro is used to set the default values for it. After the SD SPI driver is running, the card is mounted to the file system by the function "esp_vfs_fat_sdspi_mount" having the mount point defined by the macro MOUNT_POINT in the common.h file. Now the devleper can use the standard C file functions to read and write data to the SD card since it is mounted in the Virtual File System (VFS) abstraction layer.

After each POR the SD card is checked if it can be mounted. If the mount is successful, the characteristics of the SD card are printed in the LOG.txt file. Afterwards, data regarding the RPM and the current gear is written to the SD card in the LOG.txt. "snprintf" function is used to format the data that is written to the file in order to have a clear view of the data that is stored, meaning that the data is written in a human-readable format.

This part of the code is highly susceptible to be changed in the future, since the human readable format is not the best for data storage. A better approach would be to store the data in binary format, since it is more compact and it can be read faster. The data can be stored in a structure and then written to the file in binary format. This will be a future improvement.

The code responsible for writing to the SD card is depicted in the following code snippet taken from the sdcard_write task. Note that the semaphore is used in order to check if the common data structure is used by another task, since if it would have, the data written to the SD card could be garbage and this needs to be avoided.

```
1  ...
2  esp_err_t esp_response = ESP_FAIL;
3
4  FILE *log_file = fopen(path, "a+");
5  if (log_file == NULL)
6  {
7      ESP_LOGE(TAG, "Failed to open file for writing");
8
9      general_status.sdcard_logging = false;
10     esp_response = ESP_FAIL;
11     return esp_response;
12 }
13
14 ESP_ERROR_CHECK(sdcard_write_log_point(log_file, COOLANT_LOG_POINT,
     display_data->coolant_temperature));
15 ESP_ERROR_CHECK(sdcard_write_log_point(log_file, OIL_PRESSURE_LOG_POINT,
     display_data->oil_pressure));
16 ESP_ERROR_CHECK(sdcard_write_log_point(log_file, BATTERY_LOG_POINT,
     display_data->battery_voltage));
17 ESP_ERROR_CHECK(sdcard_write_log_point(log_file, RPM_LOG_POINT, display_data->
     rpm));
18 ESP_ERROR_CHECK(sdcard_write_log_point(log_file, TPS_LOG_POINT, display_data->
     tps));
19 ESP_ERROR_CHECK(sdcard_write_log_point(log_file, ACCEL_X_LOG_POINT,
     display_data->accelerometer[0]));
```

```
20    ESP_ERROR_CHECK(sdcard_write_log_point(log_file , ACCEL_Y_LOG_POINT,
        display_data->accelerometer[1]));
21    ESP_ERROR_CHECK(sdcard_write_log_point(log_file , ACCEL_Z_LOG_POINT,
        display_data->accelerometer[2]));
22
23    fclose(log_file);
24    ESP_LOGI(TAG, "File written");
25
26    general_status.sdcard_logging = true;
27    return ESP_OK;
28    ...
```

The general_status structure has a field that is used to check if the SD card logging is enabled or not. This is used in the web interface where a data point shown in the table is used to show the status of the SD card logging. It can be seen in the Figure I.1. This field of the structure is sent to the web interface via the HTTP server, whic responds to the GET request of the user with a particular URI that will redirect the server to the handler function which responds with all the data that is needed to be shown in the web interface.

2.9 LCD interface update task

This function is called "lcd_update" and it is used to send the data that is stored in the "display_data" structure to the LCD module. The LCD module is a 4D Systems module that parses the data that is sent to it via the UART communication protocol. 4DSystems does not define a clear way of sending data to the LCD, however, they offer an Arduino library for this purpose, it is called VisiGenie. This library is not directly compatible with the ESP32 microcontroller, since it needs the Arduino abstraction layer in order to run. A first implementation of this task was to include all the Arduino abstraction code in the ESP-IDF project in order for the VisiGenie library to succesfully compile. It worked but it was not the ideal solution.

The ESP core has 2 UART peripherals, UART1 and UART2, the first one being used also for programming the chip. Unfortunately, the VisiGenie library uses the first UART peripheral in order to send data to the LCD and it was not trivial to change it in to UART1 in the library. Therefore, a new solution was found, to rewrite the part of the VisiGenie library that sends data to the LCD by using ESP-IDF functions. The following function was taken from the VisiGenie library and it was reversed engineered in ESP-IDF.

```
1  uint16_t Genie::WriteObject (uint16_t object, uint16_t index, uint16_t data) {
2      uint16_t msb, lsb ;
3      uint8_t checksum ;
4      lsb = lowByte(data);
5      msb = highByte(data);
6      Error = ERROR_NONE;
7      deviceSerial->write(GENIE_WRITE_OBJ) ;
8      checksum  = GENIE_WRITE_OBJ ;
9      deviceSerial->write(object) ;
10     checksum ^= object ;
11     deviceSerial->write(index) ;
12     checksum ^= index ;
13     deviceSerial->write(msb) ;
14     checksum ^= msb;
15     deviceSerial->write(lsb) ;
16     checksum ^= lsb;
17     deviceSerial->write(checksum) ;
18   PushLinkState(GENIE_LINK_WFAN);
19   return FALSE;
```

```
}
```

The actual implementation of the function in the ESP-IDF project is the following.

```
static esp_err_t lcd_write_object (uint8_t object, uint8_t index, uint16_t data)
{
  uint8_t msb, lsb;
  uint8_t checksum, write_signal;
  write_signal = GENIE_WRITE_OBJ;

  lsb = lowByte(data);
  msb = highByte(data);

  uart_write_bytes(UART_NUM_2, &write_signal, 1);
  checksum  = GENIE_WRITE_OBJ ;
  uart_write_bytes(UART_NUM_2, &object, 1);
  checksum ^= object ;
  uart_write_bytes(UART_NUM_2, &index, 1);
  checksum ^= index ;
  uart_write_bytes(UART_NUM_2, &msb, 1);
  checksum ^= msb;
  uart_write_bytes(UART_NUM_2, &lsb, 1);
  checksum ^= lsb;
  uart_write_bytes(UART_NUM_2, &checksum, 1);


  uart_read_bytes(UART_NUM_2, &checksum, 1, 1000 / portTICK_PERIOD_MS);

  return 0;
}
```

So, the lcd_write_object function is used in order to send data to the LCD module. It takes as arguments the object id of the interface object, which is the type of the interface object that needs to be updated, the index of the object, which is makes the distinction between objects of the same type and the data that needs to be sent to the object. The data is sent in 2 bytes, the most significant byte and the least significant byte. The checksum is calculated by XOR-ing the bytes that are sent to the LCD module. The flow of the data is first the write signal, namely GENIE_WRITE_OBJ, then the object type, then the index of the object, then the most significant byte of the data, then the least significant byte of the data and then the checksum.

After all the above are sent, an acknoledge signal is sent by the LCD module. Based on it, it can be determined if the display is in an error state and the interface is not updating, or that the interface was updated succesfully with the data that was sent. Do note that the UART2 peripheral needs to be initialized before using, and this is done in the lcd_setup function. It is also important to check that the baudrate of the LCD module is matched with the baudrate of the ESP32 microcontroller. This is done by setting the macro LCD_BAUDRATE in the common.h file to the same value as the one set in the 4D Systems Workshop 4 IDE project settings.

Another aspect of the "lcd_update" task is that it needs to have a mechanism through which the page displayed on the LCD is changed. The display page is called a FORM by the 4D Systems Workshop 4 IDE. In order to change the current FORM displayed on the LCD, the macro GENIE_OBJ_FORM is set as an argument in the lcd_write_object function, with the index being the number of the FORM in the 4D Systems Workshop 4 IDE project. In the code, a switch case is used to change between the FORMS that are displayed on the LCD. In the display_data structure, there is a field called current_page that is used to store the current page that is displayed on the LCD, since the LCD does not have a way of sending the current

page to the ESP32 microcontroller. The switch between pages happens only if the user presses on a button, the change page button, and that can be problematic if the user presses too fast. In order to avoid this, a timer is used to block the change page button functionality for 250ms before it can be pressed again. This makes the user experience better, since the user needs to see a certain amount of delay between changing pages that is ideally constant, such that the user can get used to it.

A welcome FORM is displayed at the power up of the LCD. This is done by changing the current page to the welcome page, in the lcd_update task startup sequence, then the task is blocked for the amount of time given by the macro LCD_STARTUP_SCREEN_DELAY, defined in the common.h file.

A problem that was encountered in the development of the lcd_update task was the fact that the 4D Systems Workshop 4 IDE was not able to display correctly numbers higher than 100 in the progress bar object. If the number 12000 was set as the highest value of the progress bar, it could not have more than 1 color segment, even tough it could have 3 segments. This is a limitation, since in the case of the RPM, one would be interested to have a green segment for the normal RPM range, a yellow one for the high RPM range and a red one for the critical RPM range, which would be also ideal for shifting. The solution was to implement another function, "lcd_data_check_sanity" that is used to check the data being sent to the LCD and also to rescale it if it is outside the normal range (0 - 100).

Another problem is that one would want to expand a certain range of values on a progress bar. For example, in the case of water temperature, the ideal range would be between 85 - 95 degrees Celsius. Keep in mind that the progress bar range is between 0 - 100, so the range of 85 - 95 would be very small. In order to expand the following code snippet was used.

```
1  ...
2    /* This is the blue area of the progress bar, engine is cold  */
3    else if (LCD_CHECK_IF_COOLANT_TEMP_LOWER_85(sanitized_display_data->
     coolant_temperature))
4    {
5        sanitized_display_data->coolant_temperature = (uint8_t)((
     sanitized_display_data->coolant_temperature / 85.0) * 40.0);
6    }
7    /* This is the green area of the progress bar, considered to be optimal */
8    else if (LCD_CHECK_IF_COOLANT_TEMP_BTWEEN_85_95(sanitized_display_data->
     coolant_temperature))
9    {
10       sanitized_display_data->coolant_temperature = 40 + (uint8_t)((
     sanitized_display_data->coolant_temperature - 85.0) / 10.0 * 30.0);
11   }
12   /* This is the red area of the progress bar, considered dangerous */
13   else if (LCD_CHECK_IF_COOLANT_TEMP_HIGHER_95(sanitized_display_data->
     coolant_temperature))
14   {
15       sanitized_display_data->coolant_temperature = 70 + (uint8_t)((
     sanitized_display_data->coolant_temperature - 95) / 5.0 * 30.0);
16   }
17 ...
```

2.10 Accelerometer read task

The accelerometer read task is used to read the data directly from the registers of the MPU6050 accelerometer. It is based on the I2C communication protocol, which is can be understood as a concept from the following resource, that was used in the development of the Firmware [19]. In I2C, the master device sends a start condition on the bus, then it sends the

address of the slave device, then it sends the register address from which it wants to read the data. In the case of MPU6050, it automatically increments the register address after reading the data from it.

First, in order to understand this concept, one must understand the way the data is stored into the MPU6050 registers [20]. The accelerometer has a resolution of 16 bits on each axis, X, Y and Z, therefore the data is stored in 2 registers for each axis in the 2's complement format. In this way, one would need to read 2 registers in order to obtain the data from one axis. This is where the mechanism of autoincrementing the register comes into place. Let's assume that the ESP32 sends the address of the first register of the X axis to the MPU6050. The MPU6050 will send the data of all the 6 registers, starting from the X axis, then the Y axis and then the Z axis.

In order to better understand the way the data is read from the MPU6050, the following code snippet is taken from the accelerometer read task.

```
...
/* MPU6050 has 8 bit registers, but the acceleration resolution is on
16bit for each axis */
uint8_t read_stream[MPU6050_AXIS_NUM * 2] = {0};
uint8_t transmit_stream                   = MPU6050_ACCEL_XOUT_H;
while(true)
{
  /* Start reading acceleration registers from register 0x3B for 6 bytes */
  esp_response = i2c_master_write_read_device( i2c_master_port,
                                               MPU6050_I2C_ADDRESS,
                                               &transmit_stream,
                                               1,
                                               &read_stream[0],
                                               STREAM_SIZE(read_stream),
                                               MPU6050_POLLING_RATE);
  if (esp_response != ESP_OK)
  {
    ESP_LOGE("accelerometer_read", "Failed to read data");
  }

  /* Catch MPU6050 error or entering sleep mode
  Typically it manifests in reading 0 on all accelerometer axis */
  if ((read_stream[0] == 0U) && (read_stream[1] == 0U)
      && (read_stream[2] == 0U) && (read_stream[3] == 0U))
  {
      accelerometer_reset();
      vTaskDelay(MPU6050_POLLING_RATE);
  }

  /* Check if the data structure is used by other tasks */
  if ( xSemaphoreTake(xSemaphore_display_data, portMAX_DELAY) == pdTRUE)
  {
      /* Combine the High and Low bytes from read_stream into a single
variable  */
      for (uint_fast8_t i = 0; i < MPU6050_AXIS_NUM; i++)
      {
          *(display_data->accelerometer + i) = (read_stream[i * 2] << 8 |
read_stream[(i * 2) + 1]);
      }
      xSemaphoreGive(xSemaphore_display_data);
  }
  ...
  vTaskDelay(MPU6050_POLLING_RATE);
}
```

```
43    ...
```

As seen in the code snippet, the data is read using a polling mechanism. This is not a problem since it is read on a lower frequency than the rate at which the accelerometer registers are updated. One can see that an array, namely "read_stream" is used to store the data that is read from the accelerometer and the type of the array is equal in size with the register size of the accelerometer, namely 8 bits. This is because in the i2c_master_write_read_device function, that includes the steps of the I2C communication protocol, each byte is read separately and then written in the array without any processing. In order to combine the 2 bytes of data into one, a for loop is used to shift the first byte by 8 bits and then to add the second byte to it, because the first byte is the 8 higher bits of the 16 bit data, and the second byte is the 8 lower bits of the 16 bit data.

Because one can not be sure that the accelerometer is working as expected, an error catching mechanism is used. This mechanism is based on the fact that the accelerometer will send 0 data if it is in sleep mode or if it is not working properly. In this case, the accelerometer is reset and then the task is blocked for a certain amount of time. Empirically, this was tested and it was found that the accelerometer then sends the correct data, not 0s.

Another important aspect of the code is the semaphore that locks the common data structure "display_data" if it is used in another task. This is a safety feature to avoid garbage data to be written in the data structure. A downside of this approach is the fact that it does not guarantee that the data is written in the structure each time it is read from the accelerometer.

The reset function of this peripheral was particularly interesting to develop, because many of the code examples on the internet were not working as expected. Almost all the reset sequences forgot to also reset the signal path of the accelerometer inside the MPU6050 peripheral. This caused a glitch in the reset sequence which would make the MPU6050 sends 0s on all the axes, even though it was reset, then enabled and configured correctly. After many hours of debugging, the solution was found in the datasheet of the MPU6050, in which it was advised that the signal path of the accelerometer must also be reset in order to have a correct reset sequence.

The polling rate of the accelerometer is defined in the common.h file, in order to have a way of changing it easily. The same goes for the other constants that are used in the code. This is a good practice in C programming, because it allows for the developer to change the way the code behaves without changing the code itself. It was chosen based on the fact that the signals that were defined by the team to be read from the accelerometer are not changing very fast. By using the Niquest theorem, the sampling rate was chosen to be 100 Hz, which is a good rate for the signals that are under 50 Hz.

2.11 Wi-Fi module and HTTP server

In this subchapter I will describe the way the Wi-Fi module is implemented in the code and how the HTTP server is used to send the data to the user. This part of code is highly asynchronous, therefore there is no need to run a task for this. An interesting aspect is that when the Wi-Fi module is enabled, one can not use ADC 1 pins, since that ADC is used by the antenna of the ESP32 SoC. This is a taken into consideration in the hardware design of the dashboard module. In order to learn more about the Wi-Fi protocol one can refer to the following resource [21].

It is also worth to mention that the Wi-Fi module is using a lot of power when enabled and causes the ESP32 chip to heat up. This is why a heat sink is used on the SoC and the ground thermal relief pads under the SoC that take all the heat in the ground plane of the PCB.

The Wi-Fi module is initialized in the "wifi_setup" function, in which the Wi-Fi module is enabled and the Access Point (AP) is started. The AP name and password are defined in the common.h file, by the macros ESP_WIFI_SSID and ESP_WIFI_PASSWORD. It is worth to mention that the ESP-IDF libraries include a DHCP server, so when a device connects to the AP, it will automatically get an IP address in the same network as the AP.

As usual, an initialization function is used to set the Wi-Fi module in a known state, named "wifi_setup" in which a structure of type "wifi_config_t" is used in order to set the configuration parameters such as:

- the service set identifier (SSID), which is the name of the AP
- the password of the AP, the maximum number of clients that can connect to the AP
- the channel of the AP
- the authentication mode of the AP, which is set to WPA2_PSK

After the Wi-Fi module is started, the HTTP server is started in the "http_server_start" function. The HTTP server [22] is used to send the data that is stored on the SD card to the user. A simple mechanism is used in order to this. The HTTP server is configured as an asynchronous type of software [23], in which the user can send a GET request to access a certain URI and the server will respond with the "LOG.txt" file that contains the telemetry data. The HTTP library has a built-in function that is used to register a URI and the HTTP method that is used to access it to a callback function. In order to showcase this mechanism, the following code snippet is shown below.

```
/* Start page URI */
    httpd_uri_t index_html_uri = {
        .uri       = "/",
        .method    = HTTP_GET,
        .handler   = index_html_get_handler,
        .user_ctx  = NULL
    };
    httpd_register_uri_handler(http_server, &index_html_uri);

    /* Log download URI */
    httpd_uri_t log_download_uri = {
        .uri       = "/sdcard/log.txt",
        .method    = HTTP_GET,
        .handler   = download_get_handler,
        .user_ctx  = &server_data
    };
    httpd_register_uri_handler(http_server, &log_download_uri);

    /* Log live data URI */
    httpd_uri_t live_data_uri = {
        .uri       = "/data",
        .method    = HTTP_GET,
        .handler   = data_get_handler,
        .user_ctx  = &general_status
    };
    httpd_register_uri_handler(http_server, &live_data_uri);

    /* Image URI */
    httpd_uri_t image_uri = {
        .uri       = "/UPBDRIVE_Logo_Horizontal.jpg",
        .method    = HTTP_GET,
        .handler   = logo_get_handler,
        .user_ctx  = &general_status
    };
    httpd_register_uri_handler(http_server, &image_uri);
```

One can see that the start page is registered to the URI "/", which is the start page of the HTTP server. When accessing the AP IP address, the user will be redirected to the start page. The start page is a simple HTML page that is stored in the "index.html". An interesting aspect of this approach is that the HTML page is embedded in the elf file that is flashed on the ESP32 microcontroller, so it lives in the flash memory of the ESP32. This approach was chosen since it is not dependent on the SD card, which is an external peripheral that could fail. In order to do this, "EMBED_TXTFILES" command is used in the CMakeLists.txt file that resides in the "wifi-http" component of the project. In this way, the .elf will contain the "index.html" and the UPBDRIVE logo jpg file that is shown in the beginning of the html file. "EMBED_TXTFILES" is a custom command from the ESP-IDF framework that is used to embed text files in the .elf file.

The download URI is used to send the "LOG.txt" file that is stored on the SD card to the user. The index page has a button that redirects the user to the download URI, which will trigger the "downloag_get_handler" function. In that function the "LOG.txt" file is opened, and the data is read from it. A do while loop is used to read the data from the file in chunks and then to sent it to the user via the "httpd_resp_send_chunk" function. When the "chunksize" the looping stops and the transfer is finished.

```
...
fd = fopen(dashboard_log, "r");
if (NULL == fd)
{
    httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR, "Failed to read existing file");

    return ESP_FAIL;
}

httpd_resp_set_type(req, "text/plain");
do {
    /* Read file in chunks into the scratch buffer */
    chunksize = fread(chunk, 1, SCRATCH_BUFSIZE, fd);

    if (chunksize > 0) {
        /* Send the buffer contents as HTTP response chunk */
        if (httpd_resp_send_chunk(req, chunk, chunksize) != ESP_OK) {
            fclose(fd);
            ESP_LOGE(TAG, "File sending failed!");
            /* Abort sending file */
            httpd_resp_sendstr_chunk(req, NULL);
            /* Respond with 500 Internal Server Error */
            httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR, "Failed to send file");
            return ESP_FAIL;
        }
    }

    /* Keep looping till the whole file is sent */
} while (chunksize != 0);
...
```

2.12 Buttons interruptions and debouncing

In order to check the state of the buttons, the GPIO pins are used. ESP-IDF framework has a library that is used to configure the GPIO pins ar either input, output or both and to set the

interrupt type that is used to trigger a handler when the state of the pin changes. The function "buttons_setup" is used to initialize the buttons. As in the case of the other peripherals, a structure is used to set the configuration parameters of the buttons, namely the "gpio_config_t" structure. The interrupt routine is triggered based on the edge of the signal that is selected in the configuration structure, which can be either rising edge, falling edge or both. The handler function is used to debounce the signal that is read from the button, since the button has a mechanical part that can cause the signal to bounce. The following code snippet is taken from the change page button handler function.

```
1   ...
2   void IRAM_ATTR button_change_page_isr(void)
3 {
4     if ((gpio_get_level(GPIO_BUTTON_LCD_PAGE) == 0) && (change_page_ready == 1))
5     {
6         general_status.signal_change_page = 0xFF;
7     }
8 }
9   ...
```

As one can see, a flag is used to check if the button is ready to be pressed again. This is a particular aspect of the button handler that introduces a 250ms delay between button presses, such that the user can not press the button too fast and the page change is done in a constant time. This ISR is activated on the falling edge of the signal, therefore a debounce mechanism that was implemented was to check if the signal is actually low, at GND level. The processing time between the button press and the page change ensures the debouncing of the signal which is not plausible for 10 ns to be at GND level. The "signal_change_page" flag is 0xFF in order to signal the "lcd_update" task that the page change button was pressed. That function will then change the page displayed on the LCD and set the flag back to 0x00.

An important aspect of the ISR handlers of the buttons is the IRAM_ATTR attribute. This attribute is used to make a function run from the internal RAM of the ESP32 instead of being stored in the flash memory. This will make the interrupt routine run faster, since the internal RAM is faster than the flash memory. This is a core feature of the ISR handler, to be run as fast as possible.

Chapter 3
Testing the dashboard module

3.1 PCB design of the testing module

In this chapter, the testing of the dashboard electronic module is presented and how the testing module was developed. This test module was created in order to showcase the functionality of the dashboard when it is not connected to the FS car. It emulates the ECU of the FS car and it connects to the dashboard module via the CAN bus. The methodology of testing is black-box [24], since the behaviour is evaluated based on the input and the output of the dashboard module.

When speaking about the architecture of the testing module, it is important to mention that it is based on the Raspberry Pi Pico microcontroller. The Raspberry Pi Pico is a microcontroller that is based on the RP2040 chip, which is a dual-core ARM Cortex-M0+ microcontroller [25]. It is a versatile microcontroller that can be used in a wide range of applications, and it is also very cheap. The emulator module uses three potentiometers to emulate the RPM, the coolant temperature and the oil temperature. These potentiometers are connected to the power supply of the Raspberry Pi Pico, namely 3.3V and to the ground. The middle pins are connected to the ADC pins of the Raspberry Pi Pico, which are used to read the analog values of the potentiometers. In order to showcase the functionality of error FORMS, acting like a check engine light in the case of the common car, two buttons are used to emulate them. The first button will trigger a low battery error, the second button will trigger a low oil pressure. The gear shift button is also implemented in the testing module in order to showcase the functionality of the gear number on the main FORM of the dashboard interface.

In order to send the data to the dashboard module, there are two ways of doing it. The first way is to use only a CAN transceiver module that is connected to the Raspberry Pi Pico pins. Raspberry Pico does not have an integrated CAN peripheral, therefore it cannot send data on the CAN bus, but there is a library that makes use of the PIO peripheral. PIO is an acronym for Programmable Input/Output, which is a peripheral that is used to implement custom ways of how a GPIO pin behaves. This library is called "can2040" and it is used to send and receive data on the CAN bus by using just the SN65HVD230DR transceiver. The second way is to use a CAN transceiver module that is connected to a CAN controller module, namely the MCP2515 module, whose library was integrated from an Arduino ported one [26] which is connected to the Raspberry Pi Pico via the SPI peripheral. By using the MCP2515 module, the Raspberry Pi Pico can send and receive data on the CAN bus without the need of emulating the CAN peripheral on the PIO peripheral, since the data is sent via the SPI peripheral to the MCP2515 module, which then makes sure to correctly send the data on the CAN bus. Based on the bus speed, certain values are set in the registers of MCP2515 [27] in order to have a correct reading of the data on the CAN bus. A problem that was encountered is the fact that the CAN 2.0 protocol standard suggests a sampling point of 82.5% of the bit. This is a problem since the MCP2515 module has a sampling point of around 60%, and this can cause the data to be read incorrectly. In order to solve this, the ESP32 microcontroller was recalibrated to have a sampling point that would be compatible with the MCP2515 module.

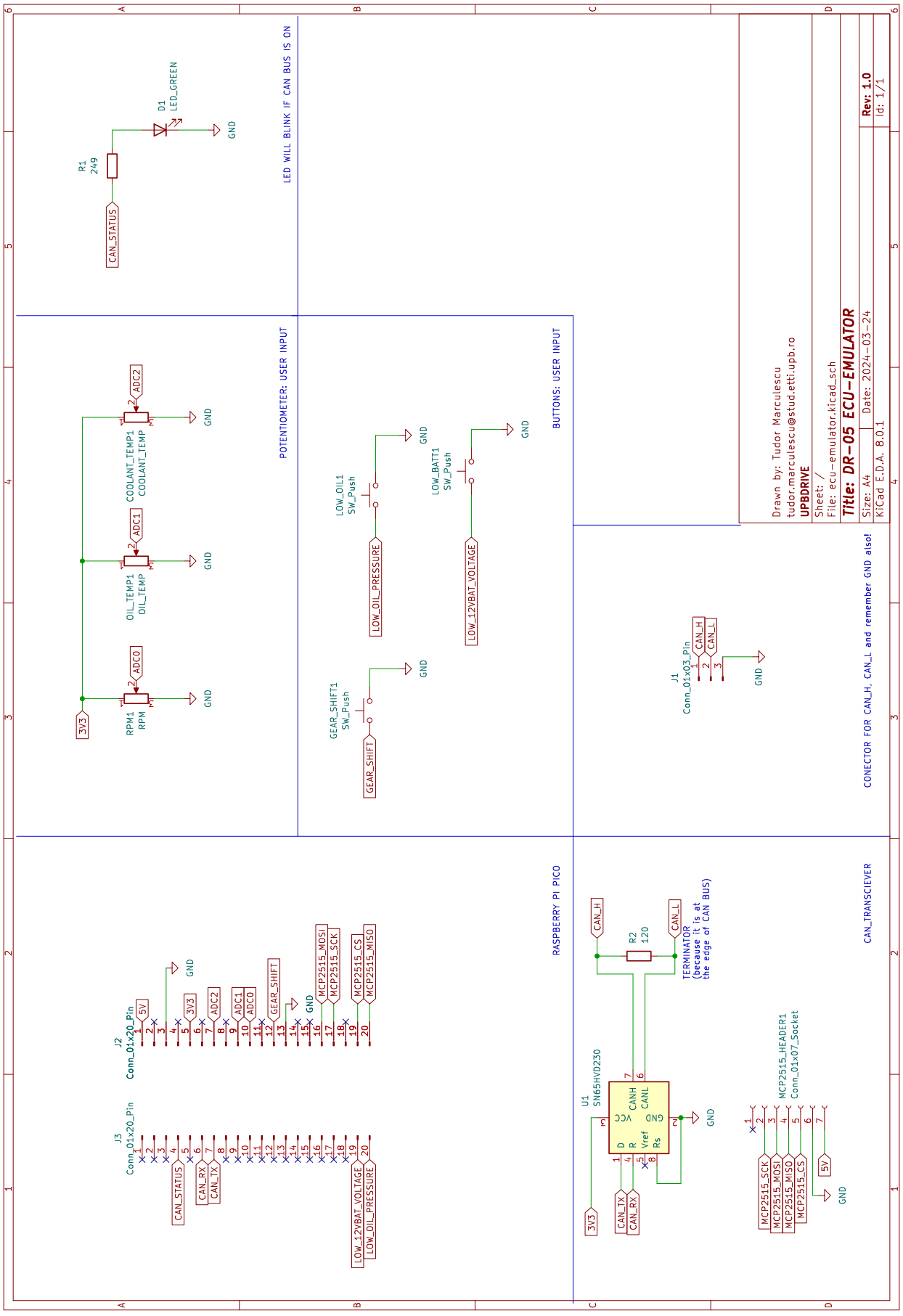


Figure 3.1: ECU Emulator schematic

As in the case of the dashboard module, the pinout of the Raspberry Pi Pico is presented in the following paragraphs. It is important to understand the pins that are used in the testing module are compatible with the desired functionality. A difference that can be seen between ESP32 and Raspberry Pi Pico is the fact that the Raspberry Pi Pico has a more versatile GPIO pinout, meaning that the pins can take more roles than the ESP32 pins. This is a good thing since it allows the developer to use the pins in a more efficient manner and it offers more flexibility.

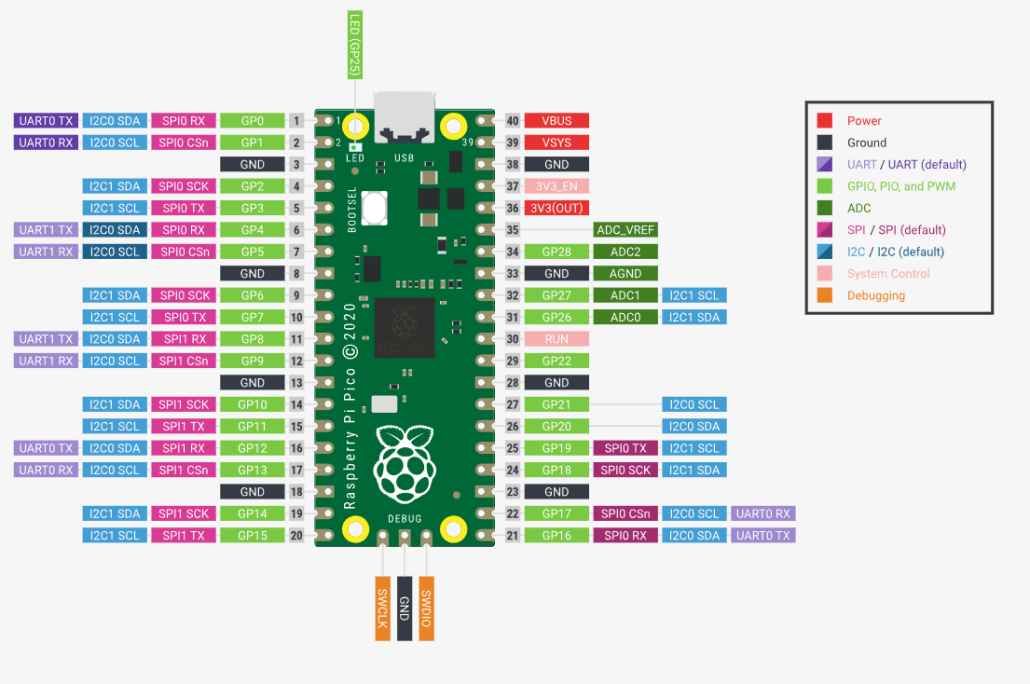


Figure 3.2: Raspberry Pi Pico pinout

GPIO 6 and 7 are used in order to send data to the CAN transceiver SN65HVD230DR, if the developer chooses to use the "can2040" library to send data on the CAN bus.

GPIO 14 and 15 are used in order to check the state of the buttons used for the error FORMS. They are checked using the pooling mechanism. It was an easier to implement mechanism and it was chosen because the buttons are not pressed very often, so the pooling mechanism is not a problem in this case.

Pin 40 and pin 36 are used as power supplies. Pin 40 is directly connected to the 5V power supply of the USB port connected to the computer, and, it is used to power up the MCP2515 module. Pin 36 is connected to the 3.3V power supply of the Raspberry Pi Pico and it is used to power up the SN65HVD230DR transceiver as well as the potentiometers and the buttons, since the GPIOs of the Raspberry Pi Pico operate at voltages between GND and 3.3V.

GPIO 28, 27 and 26 are used to read the analog values of the potentiometers that are used to emulate the RPM, the coolant temperature and the oil temperature. This pins are connected to the ADC peripheral of the Raspberry Pi Pico which has a 3.3V reference voltage and a 12 bit resolution.

GPIO 19, 18, 17 and 16 are used to communicate with the MCP2515 module via SPI, As seen in the Figure 3.2, they are connected to the SPI0 peripheral of the Raspberry Pi Pico and each one of them has a designated role in the SPI communication protocol: GPIO 19 is the SPI0_TX, GPIO 18 is SPI0_SCK, GPIO 17 is SPI0_CS and GPIO 16 is SPI0_RX. As a note, SPI0_TX is refering to the MOSI pin, while the SPI0_RX is refering to the MISO pin.

3.2 Software description of the testing module

The software of the testing module is also based on an SDK, namely the one that is provided by the Raspberry Pi Foundation, called pico-sdk. The pico-sdk is a C SDK that is used to develop software in a more efficient way on the Pico, since as in the case of the ESP-IDF, it integrates a hardware abstraction layer that makes the life of a developer easier. It is hosted on GitHub and it is open source, having a BSD-3-Clause license. [28]

As in the case of the dashboard module, the software of the testing module has a similar project structure which can be seen in the Figure 3.3. Here, only a main project CMakeLists.txt file is used to declare all the source files and the folder that include headers, to the project and to be compiled by the ARM toolchain. The main source file is the "main.c" file, in which resides the main function of the program. In this project there is no RTOS used, the programming paradigm is bare metal [29].

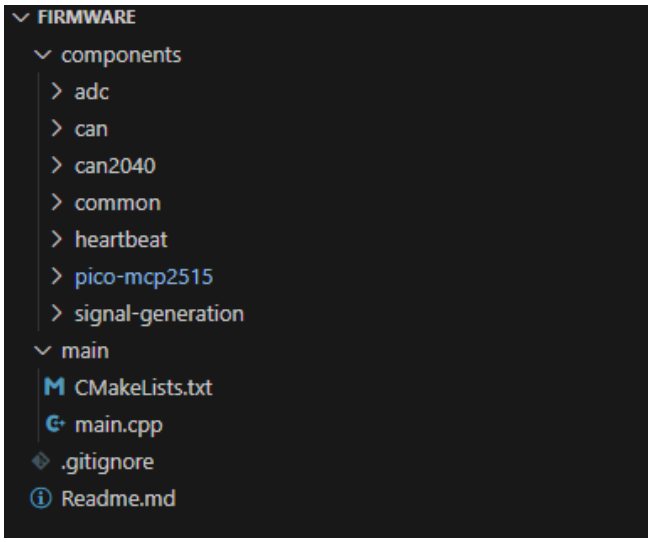


Figure 3.3: Testing Module software file structure

Remarkable are the "can2040" and "pico-mcp2515" libraries that are included in the components folder. The "can2040" library is used to send and receive data on the CAN bus by using the PIO peripheral of the Raspberry Pi Pico. The "pico-mcp2515" library is used to send and receive data on the CAN bus by using the SPI peripheral and the MCP2515 module that is connected to the PCB of the testing module.

In order to describe the way the software of the testing module works, the "send_can" function is presented in the following code snippet. This function is periodically called in the main loop of the program in order to send the acquired data from the potentiometers and the buttons via the CAN bus. The code of the function changes based on the SUPPORT macros that are defined in the common.h file from the common component. If SUPPORT_CAN2040 is defined, then this function will used the "can2040" library in order to send data via CAN, while if SUPPORT_MCP2515 is defined, then the "pico-mcp2515" library will be used to send data via CAN.

An interesting aspect is the fact that the common display_data_t structure is defined also in the code of the testing module, such that to have uniformity between the dashboard module and the testing module data types. This is a good practice since it allows the developer to easily define which data to be generated in which field of the display_data_t structure.

```
void send_can(display_data_t *data)
{
    //
  ////////////////////////////////////////////////////////////////////////
    ////                          DEFINE CAN PACKETS
  ///
    //
  ////////////////////////////////////////////////////////////////////////
    #ifdef SUPPORT_CAN2040
    struct can2040_msg outbound1;
    outbound1.id = CAN_PACKET_ECU_1;
    outbound1.dlc = 8;

    struct can2040_msg outbound2;
    outbound2.id = CAN_PACKET_ECU_2;
    outbound2.dlc = 8;
    #endif /* SUPPORT_CAN2040 */

    #ifdef SUPPORT_MCP2515
    struct can_frame outbound1;
    outbound1.can_id = CAN_PACKET_ECU_1;
    outbound1.can_dlc = 8;

    struct can_frame outbound2;
    outbound2.can_id = CAN_PACKET_ECU_2;
    outbound2.can_dlc = 8;
    #endif /* SUPPORT_MCP2515 */

    //
  ////////////////////////////////////////////////////////////////////////
    ////                          INITIALIZE CAN PACKET DATA
  ///
    //
  ////////////////////////////////////////////////////////////////////////
    /* RPM */
    outbound1.data[0] = data->rpm / 100;

    /* CurrGear */
    outbound1.data[1] = data->current_gear;

    /* TPS*/
    outbound1.data[2] = data->tps;

    /* Oil pressure */
    outbound1.data[3] = data->oil_pressure;

    /* Water temperature */
    outbound1.data[4] = data->coolant_temperature;

   ...

    /* MAP */
    outbound2.data[5] = data->map;

    /* Brake pressure (raw) */
    outbound2.data[6] = data->brake_pressure_raw;

    /* Oil temperature */
    outbound2.data[7] = data->oil_temperature;

```

```
55        //
    ////////////////////////////////////////////////////////////////////////
56        ////                          SEND CAN PACKETS
    ///
57        //
    ////////////////////////////////////////////////////////////////////////
58        #ifdef SUPPORT_CAN2040
59        can_transmit(&outbound1);
60
61        /* Wait for outbound1 packet to be sent */
62        sleep_ms(20);
63
64        can_transmit(&outbound2);
65        #endif /* SUPPORT_CAN2040 */
66
67        #ifdef SUPPORT_MCP2515
68        if(can0.sendMessage(&outbound1) == MCP2515::ERROR_OK)
69        {
70            printf("Message sent: %d\n", outbound1.can_id);
71        }
72        else
73        {
74            printf("Error sending message\n");
75        }
76
77        /* Wait for outbound1 packet to be sent */
78        sleep_ms(20);
79
80        if(can0.sendMessage(&outbound2) == MCP2515::ERROR_OK)
81        {
82            printf("Message sent: %d\n", outbound1.can_id);
83        }
84        else
85        {
86            printf("Error sending message\n");
87        }
88
89        /* Wait for outbound2 packet to be sent */
90        sleep_ms(10);
91
92        if(can0.readMessage(&outbound1) == MCP2515::ERROR_OK)
93        {
94            printf("New frame from ID: %10x\n", outbound1.can_id);
95        }
96        #endif /* SUPPORT_MCP2515 */
97    }
```

One can also see that the same macros CAN_PACKET_ECU_1 and CAN_PACKET_ECU_2 are used to define the CAN packets ids, as in the case of the dashboard software. So, as an overall concept, the CAN bus acts as a copying mechanism of the display_data_t structure between the testing module and the dashboard module in a periodic manner.

Coming back to the aquisition of the data from the potentiometers and the buttons, the code snippet below showcases the function that is used to select the ADC input on the ADC peripheral of the Raspberry Pi Pico and to read the data from the selected GPIO pin. After reading from all the ADC inputs, the values are normalized using the formula:

$$value = \frac{value \times MAX}{ADC\_RANGE},$$

where ADC_RANGE is the maximum value that can be read from the ADC peripheral, which

is 4095 in the case of the Pico and MAX is the maximum value of the range that needs to be emulated, so in the case of RPM is 12000.

```
//initialise buffers for adc
uint16_t rpm_value     = 0UL;
uint16_t oil_temp      = 0UL;
uint16_t coolant_temp  = 0UL;

//read the rpm potentiometer
adc_select_input(ADC_INPUT_RPM);
rpm_value = adc_read();

//read the oil_temp potentiometer
adc_select_input(ADC_INPUT_OIL_TEMP);
oil_temp = adc_read();

//read the coolant_temp potentiometer
adc_select_input(ADC_INPUT_COOLANT_TEMP);
coolant_temp = adc_read();

/* Set the ADC data in the display_data structure, after normalizing it  */
display_data->rpm                = (uint16_t)((rpm_value * 12000) / ADC_RANGE);
display_data->oil_temperature    = (uint8_t)((oil_temp * 150) / ADC_RANGE);
display_data->coolant_temperature = (uint8_t)((coolant_temp * 100) / ADC_RANGE);

/* If the gear shift button is pressed, increase the gear value */
if ((gpio_get(GPIO_NUM_GEAR_SHIFT) == 0U) && (button_state_gearshift == 0U))
{
    button_state_gearshift = 0xFF;

    display_data->current_gear++;

    if (display_data->current_gear >= 5U)
    {
        display_data->current_gear = 0U;
    }
}
/* This else branch is meant to protect the incrementing from happening
unless the button is released */
else if ((gpio_get(GPIO_NUM_GEAR_SHIFT) != 0U) && (button_state_gearshift == 0xFF))
{
    button_state_gearshift = 0U;
}


/* If the low battery button is pressed, set the battery_voltage to 11V */
if ((gpio_get(GPIO_NUM_LOW_BATTERY) == 0U) && (button_state_lowbattery == 0U))
{
    button_state_lowbattery = 0xFF;

    display_data->battery_voltage = 110U;
}
/* This else branch is meant to protect the incrementing from happening
unless the button is released */
else if ((gpio_get(GPIO_NUM_LOW_BATTERY) != 0U) && (button_state_lowbattery == 0xFF))
{
    button_state_lowbattery = 0x00;
```

```
55          display_data->battery_voltage = 123U;
56      }
57
58
59      /* If the low oil pressure is pressed, set the oil_pressure to 0U */
60      if ((gpio_get(GPIO_NUM_LOW_OIL_PRESSUIRE) == 0U) && (
          button_state_lowoilpressure == 0U))
61      {
62          button_state_lowoilpressure = 0xFF;
63
64          display_data->oil_pressure = 0U;
65      }
66      /* This else branch is meant to protect the incrementing from happening
67      unless the button is released */
68      else if ((gpio_get(GPIO_NUM_LOW_OIL_PRESSUIRE) != 0U) && (
          button_state_lowoilpressure == 0xFF))
69      {
70          button_state_lowoilpressure = 0x00;
71          display_data->oil_pressure = 100U;
72      }
```

A control mechanism over the gear value is implemented in this function in order to avoid the gear value to be higher than 4, since the FS car has only 4 gears. If the value of the gear reaches 5, then it is set back to 0. The buttons are read by using the polling method. An additional variable is checked to see if the button was already pressed. This is done in order to debounce the signal that is read from the button [30].

In the case of the low battery button, it is used to trigger a lower than expected value on the battery voltage field of the display_data structure. The same happens with the low oil pressure button, which is used to trigger a 0 value on the oil pressure field of the display_data structure. This buttons are used to showcase the functionality of the error FORMS that are displayed on the dashboard module, in case such error values are received from the ECU module of the FS car.

3.3 Testing methodology

In order to test the dashboard module, the testing module is connected to the dashboard module via the CAN bus. There are 3 wires that are used to connect the two devices together and they are the CAN HIGH, the CAN LOW and the GND. In the Table 3.1 on the right is shown the test pad from the dashboard module that needs to be connected to the test pad from the right, which corresponds to the testing module.

_	Dashboard Module Pin	Test Module Pin
	CAN HIGH	CAN HIGH
	CAN LOW	CAN LOW
	GND	GND

Table 3.1: Connection methodology

Then the two modules can be powered up. After they successfully initialize, the dashboard module will display the main FORM that shows the RPM, the coolant temperature, the current gear and the oil temperature. Table 3.2 shows the software parts that were tested.

_	Software part that was tested	Testing method	Did it pass?
	"can_read" and "can_send" functions	1. Check on both the testing module and on the dashboard module that CAN packets are successfully recieved 2. Check that the data in the packet is the same on the receiver and the transmitter side	Yes
	"lcd_update" function	1. Check that the LCD interface is updating correctly based on the data that is sent to it via CAN bus 2. Check the response of the LCD via UART	Yes
	"sdcard_write" function	1. Check that the file "LOG.TXT" is succesfully accessed by the function and updated with the RPM and GEAR data	Yes
	Wi-Fi AP and HTTP server	1. Connect to the AP of the dashboard module 2. Navigate to 192.168.4.1 web page 3. Download the LOG.txt file and check it's contents 4. Compare with the LOG.txt file on the sd card	Yes
	Buttons interruptions and debouncing	1. Press the change page button on the dashboard module and check that the FORMS are changing correctly 2. Check that the FAN button is read correctly by the dashboard module 3. Check that the Safety Button is read by the dashboard module	Yes
	"accelerometer_read" function	1. Check that the accelerometer data is read correctly by the dashboard module by printing the data to serial monitor	Yes
	"shift_strip_led_update" function	1. Check that the shift strip updates correctly with the increment of the RPM, by monitoring the way the LEDs light up	Yes

Table 3.2: Software testing methodology

_	Hardware part that was tested	Testing method	Did it pass?
	Buck converter 12V to 5V	1. Power up the dashboard module via the 12V and GND pin in the harness connector with 12V 2. Check that the output of the Buck converter is in the range of the 5V - 5.5V	Yes
	LDO regulator 5V to 3.3V	1. Power up the module via the USB connector 2. Check that the output of the LDO regulator is in the range of the 3V - 3.6V	Yes
	Safety Circuit Button	1. Press the safety button 2. Check that the circuit is open with a multimeter Connected to the safety circuit pins of the harness connector 3. Release the safety button 4. Check that the circuit is closed with a multimeter Connected to the safety circuit pins of the harness connector	Yes

Table 3.3: Hardware testing methodology

Keep in mind that all the values shown on the main FORM are emulated by potentiometers or buttons on the testing module. The data is then generated by the testing module and sent to the dashboard via the CAN bus, where it is read, processed, displayed and stored on the SD card. So, after running this test, the following firmware parts are tested.

The testing procedure was not done in a controlled environment since it is not a critical system, but it was done in a normal office environment. The testing was done by the author of this paper. It was not trivial to check all the aspects of the firmware since the testing module is limited in how it can emulate the data, and it certainly does not represent all the real world scenarios that the dashboard module can encounter. These tests are not exhaustive and represent just the software component of the dashboard module.

The hardware component was tested by checking the connections between different parts of the dashboard's PCB, for example to check all the pins from the ESP32 SoC that are connected to the LCD connector, the connections with the SD card module and the connections with the CAN transceiver module. All the other pins can be checked via the software, if the hardware responds correctly to the software comands. The accelerometer was tested in this way, by checking if the data that is read from it and printed to the serial output of ESP32 is correct. The power rails of the PCB were also checked in order to see if the power supply voltage is under the specified values. The Low Dropout Regulator (LDO) which can be powered up directly from the USB connector. The LDO is used to convert the 5V supply from the USB or the Buck converter into a 3.3V supply. Test pads were used to check the voltage on the LDO output and it was around 3.3V which was a succes.

Chapter 4
Conclusions

The purpose of this project is to build an electronic dashboard module that has multiple features and that will be used in a Formula Student car. The main features of the dashboard module are: interface reconfigurability, data logging, CAN bus communication, Wi-Fi communication. Other premises of the project were the fact that it needs to offer redundancy to the power supplies, that meaning, it is directly powered from the 12V car battery or from the USB port. The dashboard module is built around the ESP32 microcontroller, which is a versatile microcontroller that has a lot of peripherals that can be used in a wide range of applications.

The software part of the project was developed in the ESP-IDF framework, which offers access to the ESP32 peripherals and libraries that are included in the framework. The functionalities proposed in the project were implemented in the software, and the testing was done in order to check if the software is working as expected. The overall software architecture is the main contribution of this project, including the CAN bus communication, data logging and reading from the SD card, serial control of the LCD interface via UART, I2C communication with the accelerometer, Wi-Fi communication via HTTP server. All these components are developed in C programming language and are running on the ESP32 microcontroller.

The hardware part of the project was developed in the KiCAD software, where the contribution consisted in developing the schematic and the PCB layout of the dashboard as well as the testing module. The PCB layout includes the power supply circuitry which consists of a Buck converter and a Low Dropout Regulator (LDO). Also, in the schematic was included the CAN transceiver module, the SD card module control circuit and the accelerometer MPU6050 integration circuit. For the testing module, the schematic and PCB layout includes the Raspberry Pi Pico microcontroller, the CAN transceiver, the potentiometers and the buttons that were used to emulate the ECU of the FS car.

All the above hardware and software components were successfully developed and tested. The testing was not done in a controlled environment, but in a normal one. The tests defined in the testing methodology were also successful, therefore it can be said that the project was a success. The dashboard module is ready to be used in the FS car, and it will be included in this season's car. Also, the diploma thesis attained the proposed objectives and the results are in line with the initial thesis requirements that were also the requirements of the project.

In conclusion, this diploma thesis was a success, and it was a great experience for learning the architecture and the development framework for the ESP32 microcontroller, as well as the KiCAD software for the PCB design. The project was a great opportunity to learn about the CAN bus communication protocol, the Wi-Fi communication protocol and the I2C communication protocol, as well as advanced C programming techniques.

Bibliography

[1] Enoch Jamie, McDonald Leanne, Jones Lee, Jones Pete, and Crabb David. Evaluating whether sight is the most valued sense. (10.3389), 2019.

[2] *Genesis*. biblegateway.com, 2020. Accessed: (2024 06 23).

[3] Pong. https://en.wikipedia.org/wiki/Pong. Accessed: (2024 06 23).

[4] Liquid-crystal-display. https://en.wikipedia.org/wiki/Liquid-crystal_display. Accessed: (2024 06 23).

[5] Catelani Marcantonio, Ciani Lorenzo, and Signorini Lorenzo. Tft-lcd for avionics applications: development, characterization and reliability analysis. *IEEE Instrumentation and Measurement Technology*, (10.1109), 2010. Accessed: (2024 06 23).

[6] Espressif Systems. Esp32-wroom-32 datasheet. https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf. Accessed: (2024 06 23).

[7] Microchip Technology. Mcp2515 datasheet. https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf. Accessed: (2024 06 23).

[8] DPreview.com. 300 cd/m2 monitor brightness too low? https://www.dpreview.com/forums/thread/4432712, 2019. Accessed: (2024 06 23).

[9] Faytech.com. The outdoor usage of screens and what to think of when installing a touchscreen outdoors. https://www.faytech.com/blog/the-outdoor-usage-of-screens-what-to-think-of-when-installing-a-touchscreen-outdoor Accessed: (2024 06 23).

[10] Worldsemi. Ws2812b datasheet. https://pdf1.alldatasheet.com/datasheet-pdf/view/1179113/WORLDSEMI/WS2812B.html. Accessed: (2024 06 23).

[11] TDK InvenSense. Mpu6050 datasheet with schematic. https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf, 2013. Accessed: (2024 06 23).

[12] SD Association. Sd standard overview. https://www.sdcard.org/developers/sd-standard-overview/. Accessed: (2024 06 23).

[13] 4D Systems. Lcd display specification sheet. https://resources.4dsystems.com.au/datasheets/esp32/gen4-esp32/#hardware-overview, 2022. Accessed: (2024 06 23).

[14] Espressif Systems. Esp32-idf programming guide. https://docs.espressif.com/projects/esp-idf/en/release-v5.1/esp32/index.html. Accessed: (2024 06 23).

[15] Espressif Systems. Esp-idf examples. https://github.com/espressif/esp-idf/tree/release/v5.1/examples.

[16] FreeRTOS Association. The freertos reference manual. https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf, 2024. Accessed: (2024 06 23).

[17] Bosch. Controller area network (can) specification. http://esd.cs.ucr.edu/webres/can20.pdf. Accessed: (2024 06 23).

[18] Can bit time calculation. http://www.bittiming.can-wiki.info/, 2016. Accessed: (2024 06 23).

[19] Scott Campbell. Basics of the i2c communication protocol. https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/. Accessed: (2024 06 23).

[20] TDK InvenSense. Mpu6050 register map. https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf, 2013. Accessed: (2024 06 23).

[21] Wi-Fi Association. Discover wi-fi. https://www.wi-fi.org/discover-wi-fi, 2023. Accessed: (2024 06 23).

[22] IETF. Hypertext transfer protocol – http/1.1. https://datatracker.ietf.org/doc/html/rfc2616, 1999. Accessed: (2024 06 23).

[23] IEEE group. Osi model. https://datatracker.ietf.org/doc/html/rfc1142, 1990. Accessed: (2024 06 23).

[24] Geeks for Geeks. Black box testing – software engineering. https://www.geeksforgeeks.org/software-engineering-black-box-testing/, 2024. Accessed: (2024 06 23).

[25] Raspberry Pi Foundation. Raspberry pi pico and pico w. https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html, 2012. Accessed: (2024 06 23).

[26] Adamczyk Piotr. Raspberry pi pico mcp2515 can interface library. https://github.com/adamczykpiotr/pico-mcp2515, 2022. Accessed: (2024 06 23).

[27] Microchip Technology. Mcp2515 datasheet. https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf. Accessed: (2024 06 23).

[28] Raspberry Pi Foundation. Pico sdk. https://github.com/raspberrypi/pico-sdk. Accessed: (2024 06 23).

[29] Bare machine. Bare machine. https://en.wikipedia.org/wiki/Bare_machine, 2021. Accessed: (2024 06 23).

[30] Jens Christoffersen. Switch bounce and how to deal with it. https://www.allaboutcircuits.com/technical-articles/switch-bounce-how-to-deal-with-it/, 2015. Accessed: (2024 06 23).

Annex A
Dashboard source code

main/main.c

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

/* Include tasks */
#include <lcd_task.h>
#include <led_task.h>
#include <heartbeat_task.h>
#include <can_task.h>
#include <accelerometer_task.h>
#include <hybrid_selector_task.h>
#include <sdcard_task.h>
#include <buttons_task.h>
#include <wifi_task.h>
#include <timer_task.h>

#include <common.h>
#include <stdio.h>
#include <driver/gpio.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                                ////
////////////////////////////////////////////////////////////////////////////////

/* Global semaphore for display_data struct */
SemaphoreHandle_t xSemaphore_display_data;

display_data_t display_data =
{
    /* Initialize the structure members with 0 */
    0
};

status_firmware_t general_status =
{
    /* Initialize the structure members with 0 */
    0
};

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL FUNCTIONS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL MACROS                                    ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL CONSTANTS                                 ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL FUNCTIONS                                 ////
////////////////////////////////////////////////////////////////////////////////

esp_err_t dashboard_init()
{
    esp_err_t esp_response                  = ESP_FAIL;
    twai_mode_t can_mode                    = TWAI_MODE_NORMAL;
```

```
63      #if (CAN_SPEED == 250)
64      twai_timing_config_t can_speed            = TWAI_TIMING_CONFIG_250KBITS();
65      #elif (CAN_SPEED == 500)
66      twai_timing_config_t can_speed            = {.clk_src = TWAI_CLK_SRC_APB, .
        quanta_resolution_hz = 8000000, .brp = 10, .tseg_1 = 13, .tseg_2 = 2, .sjw = 1, .
        triple_sampling = false};
67      #elif (CAN_SPEED == 800)
68      twai_timing_config_t can_speed            = TWAI_TIMING_CONFIG_800KBITS();
69      #elif (CAN_SPEED == 1000)
70      twai_timing_config_t can_speed            = TWAI_TIMING_CONFIG_1MBITS();
71      #endif /* CAN_SPEED */
72      twai_filter_config_t packet_filter_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();
73
74      general_status.display_data               = &display_data;
75      general_status.time_second                = 99U;
76      general_status.time_minute                = 99U;
77      general_status.time_hour                  = 99U;
78      general_status.sdcard_logging             = false;
79
80      /* setup the lcd display */
81      ESP_ERROR_CHECK(lcd_setup());
82
83      /* setup can bus */
84      ESP_ERROR_CHECK(can_setup(can_mode, &can_speed, &packet_filter_config));
85
86      /* setup buttons */
87      ESP_ERROR_CHECK(buttons_setup());
88
89      /* setup led status */
90      ESP_ERROR_CHECK(heartbeat_setup());
91
92      /* setup the Semaphore for display data */
93      xSemaphore_display_data                = xSemaphoreCreateBinary();
94      general_status.xSemaphore_display_data = xSemaphore_display_data;
95
96      /* Because for some reason it is initialized to 0 instead of 1*/
97      xSemaphoreGive(xSemaphore_display_data);
98
99      /* Init RTC data in general_status with 99 which is an imposible value */
100
101     /* setup the shift and neutral led */
102     ESP_ERROR_CHECK(shift_neutral_led_setup());
103
104     /* setup shift strip */
105     ESP_ERROR_CHECK(shift_strip_led_setup());
106
107     /* setup the accelerometer module */
108     ESP_ERROR_CHECK(accelerometer_setup());
109
110     /* setup the hybrid selector */
111     ESP_ERROR_CHECK(hybrid_selector_setup());
112
113     /* setup the sdcard module */
114     ESP_ERROR_CHECK(sdcard_setup());
115
116     /* setup wifi */
117     wifi_setup();
118
119     /* setup http server */
120     ESP_ERROR_CHECK(http_server_setup());
121
122     /* ToDo override time check */
123     general_status.time_second = 0;
124     general_status.time_minute = 30;
125     general_status.time_hour   = 12;
126
127     /* setup timer for data logging - every second increments the time value
128        stored in general_status struct */
129     ESP_ERROR_CHECK(timer_setup());
130
131     esp_response = ESP_OK;
132     return esp_response;
133 }
134
135
136 void app_main(void)
```

```
{
    esp_err_t esp_response = ESP_FAIL;
    esp_response = dashboard_init();

    if (esp_response != ESP_OK)
    {
        while(true)
        {
            printf("Dashboard failed to initialize\n");
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            /* Infinite loop caused by dash perpiherals
                init failure */
        }
    }

    /* CORE 0 tasks*/
    xTaskCreatePinnedToCore(heartbeat,            "heartbeat",            1024, NULL,
                        0, NULL, 0);
    xTaskCreatePinnedToCore(can_read,             "can_read",             2048, &
    general_status,                   3, NULL, 0);
    xTaskCreatePinnedToCore(can_send,             "can_send",             2048, &
    general_status,                   3, NULL, 0);
    xTaskCreatePinnedToCore(accelerometer_read,   "accelerometer_read",   2048, &
    general_status,                   1, NULL, 0);
    xTaskCreatePinnedToCore(hybrid_selector_read, "hybrid_selector_read", 2048, &
    general_status,                   2, NULL, 0);

    /* CORE 1 tasks */
    xTaskCreatePinnedToCore(neutral_led_update,       "shift_neutral_led_update",   1024, &
    general_status,                   0, NULL, 1);
    xTaskCreatePinnedToCore(shift_strip_led_update,   "shift_strip_led_update",     4096, &
    general_status,                   1, NULL, 1);
    xTaskCreatePinnedToCore(lcd_update,               "lcd_update",                 4096, &
    general_status,                   3, NULL, 1);
    xTaskCreatePinnedToCore(sdcard_write,             "sdcard_write",               4096, &
    general_status,                   2, NULL, 1);

    /* DEBUG tasks */
    #ifdef ENABLE_DEBUG_BUTTONS
    xTaskCreatePinnedToCore(buttons_read,         "buttons_task",         1024, NULL,
                        0, NULL, 0);
    #endif /* ENABLE_DEBUG_BUTTONS */

}
```

components/adc/inc/hybrid_selector_task.h

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#ifndef HYBRID_SELECTOR_TASK_H
#define HYBRID_SELECTOR_TASK_H

#include <common.h>
#include <esp_adc/adc_oneshot.h>

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL MACROS                                   ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL CONSTANTS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        FUNCTION PROTOYPES                              ////
////////////////////////////////////////////////////////////////////////////////

/*
    * @brief This function setups the gpio used for the hybrid selector as output
    *        and the ADC_UNIT_1 in oneshot mode with CHANNEL 6 attenuation 11db
```

```
    *
    * @note ADC_UNIT_2 cannot be used because it is used by Wi-Fi
    *
    * @param void
    *
    * @return esp_err_t
*/
esp_err_t hybrid_selector_setup(void);


/*
    * @brief This task will read the hybrid selector
    *
    * @param[in] pvParameters IN
    *
    * @return void
*/
void hybrid_selector_read(void *pvParameters);

#endif /* HYBRID_SELECTOR_TASK_H */
```

components/adc/inc/hybrid_selector_task.c

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#include <hybrid_selector_task.h>

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL FUNCTIONS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL VARIABLES                                 ////
////////////////////////////////////////////////////////////////////////////////

static adc_oneshot_unit_handle_t adc1_handle;

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL MACROS                                    ////
////////////////////////////////////////////////////////////////////////////////

/* Error state */
#define HYBRID_STATE_ERROR (0)

/* Voltage value between 0.4V - 0.6V */
#define HYBRID_STATE_1 (1)

/* Voltage value between 0.7V - 0.9V */
#define HYBRID_STATE_2 (2)

/* Voltage value between 1.0V - 1.2V */
#define HYBRID_STATE_3 (3)

/* Voltage value between 1.3V - 1.5V */
#define HYBRID_STATE_4 (4)

/* Voltage value between 1.6V - 1.8V */
#define HYBRID_STATE_5 (5)

/* Voltage value between 1.9V - 2.1V */
#define HYBRID_STATE_6 (6)

/* Voltage value between 2.1V - 2.3V */
#define HYBRID_STATE_7 (7)

/* Volatage value between 2.4V - 2.6V */
#define HYBRID_STATE_8 (8)

/* Voltage value between 2.7V - 2.9V */
#define HYBRID_STATE_9 (9)

/* Voltage value between 2.9V - 3.1V */
#define HYBRID_STATE_10 (10)

```

```
55 /* Voltage value between 3.2V - 3.3V */
56 #define HYBRID_STATE_11 (11)
57
58 /* Numeric digital outputs of the 12bit adc conversion for different voltages*/
59 #define RAW_VOLTAGE_0V4 (496)
60 #define RAW_VOLTAGE_0V6 (744)
61 #define RAW_VOLTAGE_0V7 (868)
62 #define RAW_VOLTAGE_0V9 (1117)
63 #define RAW_VOLTAGE_1V0 (1241)
64 #define RAW_VOLTAGE_1V2 (1489)
65 #define RAW_VOLTAGE_1V3 (1613)
66 #define RAW_VOLTAGE_1V5 (1861)
67 #define RAW_VOLTAGE_1V6 (1985)
68 #define RAW_VOLTAGE_1V8 (2234)
69 #define RAW_VOLTAGE_1V9 (2358)
70 #define RAW_VOLTAGE_2V1 (2606)
71 #define RAW_VOLTAGE_2V3 (2854)
72 #define RAW_VOLTAGE_2V4 (2978)
73 #define RAW_VOLTAGE_2V6 (3227)
74 #define RAW_VOLTAGE_2V7 (3351)
75 #define RAW_VOLTAGE_2V9 (3599)
76 #define RAW_VOLTAGE_3V1 (3847)
77 #define RAW_VOLTAGE_3V2 (3971)
78
79 ////////////////////////////////////////////////////////////////////////////
80 ////                          LOCAL CONSTANTS                           ////
81 ////////////////////////////////////////////////////////////////////////////
82
83 esp_err_t hybrid_selector_setup(void)
84 {
85     esp_err_t esp_response                  = ESP_FAIL;
86
87     adc_oneshot_unit_init_cfg_t init_adc1_config =
88     {
89         .unit_id = ADC_UNIT_1
90     };
91
92     adc_oneshot_chan_cfg_t channel_adc1_config = {
93         .atten    = ADC_ATTEN_DB_11,
94         .bitwidth = ADC_BITWIDTH_12
95     };
96
97     esp_response = adc_oneshot_new_unit(&init_adc1_config, &adc1_handle);
98     if (esp_response != ESP_OK)
99     {
100         printf("Failed to configure the ADC1 unit\n");
101         return esp_response;
102     }
103
104     esp_response = adc_oneshot_config_channel(adc1_handle, ADC_CHANNEL_6, &channel_adc1_config
    );
105     if (esp_response != ESP_OK)
106     {
107         printf("Failed to configure the ADC1 channel\n");
108         return esp_response;
109     }
110
111     esp_response = ESP_OK;
112     return esp_response;
113 }
114
115 void hybrid_selector_read(void *pvParameters)
116 {
117     status_firmware_t *general_status         = (status_firmware_t*)pvParameters;
118     display_data_t    *display_data           = general_status->display_data;
119     SemaphoreHandle_t xSemaphore_display_data = general_status->xSemaphore_display_data;
120     int       adc_buffer                      = 0;
121     uint8_t   i                               = 0U;
122     uint8_t   hybrid_selector_value           = HYBRID_STATE_ERROR;
123     uint32_t  hybrid_selector_raw             = 0UL;
124
125     while(true)
126     {
127         ESP_ERROR_CHECK(adc_oneshot_read(adc1_handle, ADC_CHANNEL_6, &adc_buffer)) ;
128
129         /* Add the value to the hybrid selector raw buffer */
```

```
130         hybrid_selector_raw = hybrid_selector_raw + adc_buffer;
131
132         /* Check if the iterator reached the maximum or overflowed it */
133         if( i >= (HYBRID_SELECTOR_MA_FILTER_SAMPLES) )
134         {
135             /* Compute the moving average result of the raw buffer */
136             hybrid_selector_raw = hybrid_selector_raw / (HYBRID_SELECTOR_MA_FILTER_SAMPLES+1);
137
138             /* Update the state of the of the hybrid selector */
139             if (hybrid_selector_raw >= RAW_VOLTAGE_0V4 && hybrid_selector_raw <=
    RAW_VOLTAGE_0V6)
140             {
141                 hybrid_selector_value = HYBRID_STATE_1;
142             }
143             else if (hybrid_selector_raw >= RAW_VOLTAGE_0V7 && hybrid_selector_raw <=
    RAW_VOLTAGE_0V9)
144             {
145                 hybrid_selector_value = HYBRID_STATE_2;
146             }
147             else if (hybrid_selector_raw >= RAW_VOLTAGE_1V0 && hybrid_selector_raw <=
    RAW_VOLTAGE_1V2)
148             {
149                 hybrid_selector_value = HYBRID_STATE_3;
150             }
151             else if (hybrid_selector_raw >= RAW_VOLTAGE_1V3 && hybrid_selector_raw <=
    RAW_VOLTAGE_1V5)
152             {
153                 hybrid_selector_value = HYBRID_STATE_4;
154             }
155             else if (hybrid_selector_raw >= RAW_VOLTAGE_1V6 && hybrid_selector_raw <=
    RAW_VOLTAGE_1V8)
156             {
157                 hybrid_selector_value = HYBRID_STATE_5;
158             }
159             else if (hybrid_selector_raw >= RAW_VOLTAGE_1V9 && hybrid_selector_raw <=
    RAW_VOLTAGE_2V1)
160             {
161                 hybrid_selector_value = HYBRID_STATE_6;
162             }
163             else if (hybrid_selector_raw >= RAW_VOLTAGE_2V1 && hybrid_selector_raw <=
    RAW_VOLTAGE_2V3)
164             {
165                 hybrid_selector_value = HYBRID_STATE_7;
166             }
167             else if (hybrid_selector_raw >= RAW_VOLTAGE_2V4 && hybrid_selector_raw <=
    RAW_VOLTAGE_2V6)
168             {
169                 hybrid_selector_value = HYBRID_STATE_8;
170             }
171             else if (hybrid_selector_raw >= RAW_VOLTAGE_2V7 && hybrid_selector_raw <=
    RAW_VOLTAGE_2V9)
172             {
173                 hybrid_selector_value = HYBRID_STATE_9;
174             }
175             else if (hybrid_selector_raw >= RAW_VOLTAGE_2V9 && hybrid_selector_raw <=
    RAW_VOLTAGE_3V1)
176             {
177                 hybrid_selector_value = HYBRID_STATE_10;
178             }
179             else if (hybrid_selector_raw >= RAW_VOLTAGE_3V2)
180             {
181                 hybrid_selector_value = HYBRID_STATE_11;
182             }
183             else
184             {
185                 hybrid_selector_value = HYBRID_STATE_ERROR;
186             }
187
188             /* Reset hybrid buffer and iterator */
189             hybrid_selector_raw = 0UL;
190             i                   = 0U;
191
192             if (xSemaphoreTake(xSemaphore_display_data, (TickType_t)10) == pdTRUE)
193             {
194                 display_data->hybrid_selector_value = hybrid_selector_value;
195                 xSemaphoreGive(xSemaphore_display_data);
```

```
            }

            #ifdef ENABLE_DEBUG_HYBRID_SELECTOR
            printf("hybrid_selector_value: %d, iterator: %d\n", hybrid_selector_value, i);
            #endif /* ENABLE_DEBUG_HYBRID_SELECTOR */
        }

        /* Increment the iterator */
        i++;

        /* put the task in blocking state */
        vTaskDelay(HYBRID_SELECTOR_POLLING_RATE);
    }
}
```

components/can/inc/can_task.h

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#ifndef CAN_TASK_H
#define CAN_TASK_H

#include <common.h>
#include <driver/gpio.h>
#include <driver/twai.h>
#include <string.h>

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL MACROS                                   ////
////////////////////////////////////////////////////////////////////////////////

#define CAN_POLLING_RATE      (20/portTICK_PERIOD_MS)

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL CONSTANTS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        FUNCTION PROTOYPES                              ////
////////////////////////////////////////////////////////////////////////////////

/*
    * @brief This function will setup the CAN bus
    *
    * @param[in] can_mode    IN configures if the CAN driver sends ack signals upon receiving a message or not
    *
    * @note: if the can_mode is set to TWAI_MODE_LISTEN_ONLY and there are not any other nodes on the bus, then the
    *           sender node will softlock itself in retrying to send their message
    *
    * @param[in] can_speed   IN sets the bitrate of the can bus
    * @param[in] packet_filter_config IN sets to filter certain messages
    *
    * @return esp_err_t
*/
esp_err_t can_setup(twai_mode_t can_mode, twai_timing_config_t *can_speed, twai_filter_config_t *packet_filter_config);

/*
    * @brief This task will read packets from the CAN bus
    *
    * @param[in] pvParameters IN
    *
    * @return void
*/
void can_read(void *pvParameters);


/*
```

```
58      * @brief This task will send packets on the CAN bus
59      *
60      * @param[in] pvParameters IN
61      *
62      * @return void
63  */
64  void can_send(void* pvParameters);
65
66
67
68  #endif /* CAN_TASK_H */
```

components/can/inc/can_task.c

```
1   /*****************************************************************************/
2   /* UPBDRIVE Dashboard Firmware                                               */
3   /* All rights reserverd 2023                                                 */
4   /*****************************************************************************/
5
6   #include <can_task.h>
7
8   ////////////////////////////////////////////////////////////////////////////////
9   ////                        GLOBAL VARIABLES                                ////
10  ////////////////////////////////////////////////////////////////////////////////
11
12  ////////////////////////////////////////////////////////////////////////////////
13  ////                        GLOBAL FUNCTIONS                                ////
14  ////////////////////////////////////////////////////////////////////////////////
15
16  ////////////////////////////////////////////////////////////////////////////////
17  ////                        LOCAL MACROS                                    ////
18  ////////////////////////////////////////////////////////////////////////////////
19
20  /*
21      * @brief data structure:
22      * data[0]: RPM / 100
23      * data[1]: CurrGear
24      * data[2]: TPS
25      * data[3]: Oil pressure
26      * data[4]: Water temperature
27      * data[5]: Fuel pressure
28      * data[6]: Lamda
29      * data[7]: IAT
30  */
31  #define CAN_PACKET_ECU_1 (0x3E8)
32
33  /*
34      * @brief data structure:
35      * data[0]: EGT1 temp / 100
36      * data[1]: EGT2 temp / 100
37      * data[2]: EGT3 temp / 100
38      * data[3]: EGT4 temp / 100
39      * data[4]: Vehicle speed
40      * data[5]: Manifold Air Pressure (MAP)
41      * data[6]: BPS raw value
42      * data[7]: Oil temperature
43  */
44  #define CAN_PACKET_ECU_2 (0x3E9)
45
46  /*
47      * @brief data structure:
48      * data[0]: LV battery voltage
49      * data[1]: Input voltage
50      * data[2]: Input current battery
51      * data[3]: Input current alternator
52  */
53  #define CAN_PACKET_PMU_1 (0x3ED)
54
55  /*
56      * @brief data structure:
57      * data[4]: Tyre pressure Left-Front
58    * data[5]: Tyre pressure Right-Front
59    * data[6]: Tyre pressure Left-Rear
60    * data[7]: Tyre pressure Right-Rear
61  */
62  #define CAN_PACKET_VDU (0x3EC)
63
```

```
64 /*
65     * @brief data structure:
66     * data[0]: Selector value
67 */
68 #define CAN_PACKET_DASH (0x384)
69
70 /*
71     * @brief data structure:
72     * data[0]: HOUR value
73     * data[1]: MINUTE value
74     * data[2]: SECOND value
75 */
76 #define CAN_PACKET_DAT (0x10)
77
78 /*
79     * @brief:
80     * Macro used to determine if the can packet received can be used to update
81     * the display_data structure
82 */
83 #define CAN_PACKET_FOR_DISPLAY_DATA(id) ((id == CAN_PACKET_ECU_1) || (id == CAN_PACKET_ECU_2)
       || (id == CAN_PACKET_PMU_1) || (id == CAN_PACKET_VDU))
84 ////////////////////////////////////////////////////////////////////////////////
85 ////                         LOCAL CONSTANTS                                ////
86 ////////////////////////////////////////////////////////////////////////////////
87
88 esp_err_t can_setup(twai_mode_t can_mode, twai_timing_config_t *can_speed,
       twai_filter_config_t *packet_filter_config)
89 {
90     twai_general_config_t g_config = TWAI_GENERAL_CONFIG_DEFAULT(GPIO_CAN_TX, GPIO_CAN_RX,
       can_mode);
91     twai_timing_config_t  t_config;
92     twai_filter_config_t  f_config;
93     esp_err_t esp_response = ESP_FAIL;
94
95     /* Copy the configuration parameters locally */
96     memcpy(&t_config, can_speed,            sizeof(twai_timing_config_t));
97     memcpy(&f_config, packet_filter_config, sizeof(twai_filter_config_t));
98
99     /* Install the can driver */
100    ESP_ERROR_CHECK(twai_driver_install(&g_config, &t_config, &f_config));
101
102    /* Start CAN driver */
103    ESP_ERROR_CHECK(twai_start());
104
105    /* If everything went ok, exit with ESP_OK response */
106    esp_response = ESP_OK;
107    return esp_response;
108 }
109
110
111 void can_read(void *pvParameters)
112 {
113     status_firmware_t *general_status         = (status_firmware_t*)pvParameters;
114     display_data_t *display_data              = general_status->display_data;
115     SemaphoreHandle_t xSemaphore_display_data = general_status->xSemaphore_display_data;
116     twai_message_t message                    = {0};
117     esp_err_t esp_response                    = ESP_FAIL;
118     uint8_t counter_no_packet_recieved        = 0;
119
120     while(true)
121     {
122         #ifdef ENABLE_CAN_READ_DEBUG
123         printf("I'm in the can read task\n");
124         #endif /* ENABLE_CAN_READ_DEBUG */
125
126         esp_response = twai_receive(&message, CAN_POLLING_RATE);
127
128         switch (esp_response)
129         {
130             case ESP_OK:
131             {
132                 #ifndef ENABLE_CAN_READ_DEBUG
133                 if (CAN_PACKET_FOR_DISPLAY_DATA(message.identifier))
134                 {
135                     if( xSemaphoreTake( xSemaphore_display_data, ( TickType_t ) 10 ) == pdTRUE
       )
```

```
136                     {
137                         /* Update display_data structure based on the CAN packet id */
138                         switch(message.identifier)
139                         {
140                             case CAN_PACKET_ECU_1:
141                             {
142                                 display_data->rpm                 = (uint16_t)message.data[0]
    * 100U;
143                                 display_data->current_gear        = message.data[1];
144                                 display_data->tps                 = message.data[2];
145                                 display_data->oil_pressure        = message.data[3];
146                                 display_data->coolant_temperature = message.data[4];
147                                 display_data->fuel_pressure       = message.data[5];
148                                 display_data->lambda              = message.data[6];
149                                 display_data->iat                 = message.data[7];
150                                 break;
151                             }
152
153                             case CAN_PACKET_ECU_2:
154                             {
155                                 display_data->egt[0]              = (uint16_t)((float)(message
    .data[0] & 0xF0U) + ((message.data[0] & 0x0FU) * 0.0625)) * 100U;
156                                 display_data->egt[1]              = (uint16_t)((float)(message
    .data[1] & 0xF0U) + ((message.data[1] & 0x0FU) * 0.0625)) * 100U;
157                                 display_data->egt[2]              = (uint16_t)((float)(message
    .data[2] & 0xF0U) + ((message.data[2] & 0x0FU) * 0.0625)) * 100U;
158                                 display_data->egt[3]              = (uint16_t)((float)(message
    .data[3] & 0xF0U) + ((message.data[3] & 0x0FU) * 0.0625)) * 100U;
159                                 display_data->map                 = message.data[5];
160                                 display_data->brake_pressure_raw  = message.data[6];
161                                 display_data->oil_temperature     = message.data[7];
162                                 break;
163                             }
164
165                             case CAN_PACKET_PMU_1:
166                             {
167                                 display_data->battery_voltage         = message.data[0];
168                                 display_data->input_voltage_pmu       = message.data[1];
169                                 display_data->input_current_altr_pmu  = message.data[3];
170                                 break;
171                             }
172
173                             case CAN_PACKET_VDU:
174                             {
175                                 display_data->tyre_pressure[0] = message.data[4];
176                                 display_data->tyre_pressure[1] = message.data[5];
177                                 display_data->tyre_pressure[2] = message.data[6];
178                                 display_data->tyre_pressure[3] = message.data[7];
179                                 break;
180                             }
181
182                             case CAN_PACKET_DAT:
183                             {
184                                 general_status->time_hour = message.data[0];
185                                 general_status->time_minute = message.data[1];
186                                 general_status->time_second = message.data[2];
187                                 break;
188                             }
189
190                             default:
191                                 break;
192                         }
193
194                         xSemaphoreGive( xSemaphore_display_data );
195
196                         /* If a packet was received, then the CAN bus is available */
197                         display_data->can_status   = true;
198                         /* Reset the fault counter */
199                         counter_no_packet_recieved = 0U;
200                     }
201                 }
202                 else if(CAN_PACKET_DAT == message.identifier)
203                 {
204                     general_status->time_hour   = message.data[0];
205                     general_status->time_minute = message.data[1];
206                     general_status->time_second = message.data[2];
```

```
                }

                #else
                printf("Message received\n");
                printf("ID is %ld\n", message.identifier);
                if (!(message.rtr))
                {
                    for (int i = 0; i < message.data_length_code; i++)
                    {
                        printf("Data byte %d = %d\n", i, message.data[i]);
                    }
                }
                #endif /* ENABLE_CAN_READ_DEBUG */

                break;
            }

            case ESP_ERR_TIMEOUT:
            {
                /* No packet was received, increment the fault counter */
                counter_no_packet_recieved = counter_no_packet_recieved + 1;

                #ifdef ENABLE_CAN_READ_DEBUG
                printf("Timeout occurred, no message received\n");
                #endif /* ENABLE_CAN_READ_DEBUG */

                break;
            }

            default:
            {
                /* Other error has occured, increment the fault counter */
                counter_no_packet_recieved = counter_no_packet_recieved + 1;

                #ifdef ENABLE_CAN_READ_DEBUG
                printf("Failed to receive message\n");
                continue;
                #endif /* ENABLE_CAN_READ_DEBUG */

                break;
            }
        }

        /* If more than 200 packets were missed, then signal CAN bus FAULT */
        if (counter_no_packet_recieved == 200U )
        {
            display_data->can_status   = false;
            counter_no_packet_recieved = 200U;
        }

        vTaskDelay(CAN_POLLING_RATE);
    }
}

void can_send(void *pvParameters)
{
    status_firmware_t *general_status         = (status_firmware_t*)pvParameters;
    display_data_t *display_data              = general_status->display_data;
    SemaphoreHandle_t xSemaphore_display_data = general_status->xSemaphore_display_data;
    twai_message_t message                    = {0};

    while(true)
    {
        if( xSemaphoreTake( xSemaphore_display_data, ( TickType_t ) 10 ) == pdTRUE )
        {
            /* Send the data to the CAN bus */
            message.identifier       = CAN_PACKET_DASH;
            message.data_length_code = 1;
            message.data[0]          = display_data->hybrid_selector_value;
            message.rtr              = 0;
            message.flags            = 0;
            xSemaphoreGive( xSemaphore_display_data );
        }

        if(ESP_ERR_TIMEOUT == twai_transmit(&message, CAN_TRANSMIT_RATE))
        {
```

```
            ESP_LOGE("CAN_SEND", "Failed to transmit CAN message\n");
        };

        vTaskDelay(CAN_TRANSMIT_RATE);
    }
}
```

components/common/inc/common.h

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/
#ifndef COMMON_TYPES_H
#define COMMON_TYPES_H

#include <stdio.h>
#include <string.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/semphr.h>
#include <freertos/timers.h>
#include <driver/gpio.h>
#include <esp_log.h>

////////////////////////////////////////////////////////////////////////////////
////                                DEBUG                                   ////
////////////////////////////////////////////////////////////////////////////////
/* By uncommenting one macro, debug console output will be enabled for the selected
   functionality */
//#define ENABLE_DEBUG_ACCELEROMETER

//#define ENABLE_DEBUG_BUTTONS
/* Button polling rate, update at 60hz just for debug */
#define BUTTON_POLLING_RATE  (17/portTICK_PERIOD_MS)

//#define ENABLE_CAN_READ_DEBUG
//#define ENABLE_CAN_SEND_DEBUG
//#define ENABLE_DEBUG_DISPLAY
//#define ENABLE_DEBUG_HEARTBEAT
//#define ENABLE_DEBUG_HYBRID_SELECTOR
//#define ENABLE_DEBUG_SDCARD

////////////////////////////////////////////////////////////////////////////////
////                            CAR SETTINGS                                ////
////////////////////////////////////////////////////////////////////////////////

/*  @brief: Set the redline of the car
    @note:  possible values are 0 - 12000
*/
#define SHIFT_THRESHOLD      (10000)

/* @brief: Set the gear in which the car is considered to be in neutral
   @note: possible values: 0 - 4 */
#define NEUTRAL_GEAR        (0)

/*
   @brief: Set the coolant overheat message threshold
   @note: possible values are 0 - 255
*/
#define COOLANT_OVERHEAT_THRESHOLD (100)

/*
   @brief: Set the oil pressure message threshold
   @note: possible values are 0 - 255
*/
#define LOW_OIL_PRESSURE_THRESHOLD (50)

/*
   @brief: Set the low 12 battery message threshold
   @note: possible values are 0 - 255
*/
#define LOW_12V_BATTERY_THRESHOLD (12)

/* @brief: Set the display data transmission rate,
   @note: possible values are:
    60 HZ: 17/portTICK_PERIOD_MS
```

```
69      30 HZ: 34/portTICK_PERIOD_MS
70  */
71  #define LCD_REFRESH_RATE     (17/portTICK_PERIOD_MS)
72
73  /* @brief: Set the start-up screen delay in ms
74     @note: possible values are:
75     0   seconds:     0/portTICK_PERIOD_MS
76     2.5 seconds:     2500/portTICK_PERIOD_MS
77     5   seconds:     5000/portTICK_PERIOD_MS
78     7.5 seconds:     7500/portTICK_PERIOD_MS
79     10  seconds:     10000/portTICK_PERIOD_MS
80  */
81  #define LCD_START_UP_SCREEN_DELAY (5000/portTICK_PERIOD_MS)
82
83  /* @brief: Set the baudrate of the lcd screen in bps, keep in mind that the
84             baudrate must be the same with the one set in the 4d systems workshop
85             project
86     @note: possible values are:
87     9600 bps:     9600
88     115200 bps:   115200
89  */
90  #define LCD_BAUDRATE         (115200)
91
92  /* @brief: Hybrid selector polling rate, note that this polling rate is divided
93             by the number of samples of the moving average filter in order to obtain
94             the actual value renewal rate
95     @note: possible values are:
96     100 HZ:  10/portTICK_PERIOD_MS
97     60  HZ:  17/portTICK_PERIOD_MS
98     30  HZ:  34/portTICK_PERIOD_MS
99  */
100 #define HYBRID_SELECTOR_POLLING_RATE (10/portTICK_PERIOD_MS)
101
102 /* No of samples of the moving average filter */
103 #define HYBRID_SELECTOR_MA_FILTER_SAMPLES (50)
104
105 /* @brief: Accelerometer polling rate
106    @note: possible values are:
107    100 HZ:  10/portTICK_PERIOD_MS
108    60  HZ:  17/portTICK_PERIOD_MS
109    30  HZ:  34/portTICK_PERIOD_MS
110 */
111 #define MPU6050_POLLING_RATE         (10/portTICK_PERIOD_MS)
112
113 /* @brief: Set the maximum value in G for the accelerometer reading
114    @note: possible value are:
115           2,
116           4,
117           8 or
118           16
119 */
120 #define MPU6050_ACCEL_SCALE          (2)
121 /* DO NOT CHANGE MPU6050_AXIS_NUM */
122 #define MPU6050_AXIS_NUM             (3)
123
124 /* @brief: Set the blink period for the SHIFT LEDs on the dashboard
125    @note: possible value are:
126    1  HZ:   1000/portTICK_PERIOD_MS
127    5  HZ:   200/portTICK_PERIOD_MS
128    10 HZ:   100/portTICK_PERIOD_MS
129 */
130 #define NEUTRAL_LED_RATE        (1000/portTICK_PERIOD_MS)
131
132 /* @brief: Set the refresh rate for the SHIFT STRIP on the dashboard
133    @note: possible value are:
134    10  HZ:   100/portTICK_PERIOD_MS
135    15  HZ:   67/portTICK_PERIOD_MS
136    20  Hz:   50/portTICK_PERIOD_MS
137    30  HZ:   34/portTICK_PERIOD_MS
138 */
139 #define SHIFTLED_STRIP_RATE   (100/portTICK_PERIOD_MS)
140
141 /* @brief: Set the can bus speed in kpbs
142    @note: possible value are:
143           250,
144           500,
```

```
145           800 or
146           1000
147 */
148 #define CAN_SPEED           (500)
149
150 /* @brief: Set the can packet transmit rate of the dashboard
151    @note: possible value are:
152    100 HZ:  10/portTICK_PERIOD_MS
153    60  HZ:  17/portTICK_PERIOD_MS
154    30  HZ:  34/portTICK_PERIOD_MS
155 */
156 #define CAN_TRANSMIT_RATE     (500/portTICK_PERIOD_MS)
157
158 /* @brief: Set the log frequency on the sdcard of the display_data_t struct
159    @note: possible value are:
160    60  HZ:  17/portTICK_PERIOD_MS
161    30  HZ:  34/portTICK_PERIOD_MS
162    15  HZ:  58/portTICK_PERIOD_MS
163    1   HZ:  1000/portTICK_PERIOD_MS
164 */
165 #define SDCARD_LOGGING_RATE   (1000/portTICK_PERIOD_MS)
166
167 /* @brief: Set the SSID of the wifi network from dashboard's AP
168    @note: possible value are:
169    "dashboard_dr-05"
170 */
171 #define ESP_WIFI_SSID "dashboard_dr-05"
172
173 /* @brief: Set the password of the wifi network from dashboard's AP
174    @note: possible value are:
175    "FlatOut5"
176 */
177 #define ESP_WIFI_PASS "FlatOut5"
178
179 ////////////////////////////////////////////////////////////////////////////
180 ////                          GPIO PINS USAGE                           ////
181 ////////////////////////////////////////////////////////////////////////////
182
183 /* Define the pins used in the firmware */
184 #define GPIO_LED_STATUS         GPIO_NUM_32
185 #define GPIO_LED_SHIFT          GPIO_NUM_25
186 #define GPIO_LED_NEUTRAL        GPIO_NUM_26
187 #define GPIO_LED_STRIP          GPIO_NUM_33
188
189 #define GPIO_LCD_RX             GPIO_NUM_16
190 #define GPIO_LCD_TX             GPIO_NUM_17
191 #define GPIO_LCD_RST            GPIO_NUM_18 /* ToDo pull up in code */
192
193 #define GPIO_SDCARD_MISO        GPIO_NUM_4
194 #define GPIO_SDCARD_MOSI        GPIO_NUM_15
195 #define GPIO_SDCARD_CLK         GPIO_NUM_14
196 #define GPIO_SDCARD_CS          GPIO_NUM_13
197
198 #define GPIO_ACCEL_SDA          GPIO_NUM_21
199 #define GPIO_ACCEL_SCL          GPIO_NUM_22
200
201 #define GPIO_CAN_RX             GPIO_NUM_19
202 #define GPIO_CAN_TX             GPIO_NUM_27
203
204 #define GPIO_BUTTON_RAD_FAN     GPIO_NUM_23
205 #define GPIO_BUTTON_LCD_PAGE    GPIO_NUM_12
206 #define GPIO_BUTTON_SAFETY_CIRC GPIO_NUM_35
207
208 /* GPIO pin for the hybrid selector */
209 #if 0 // do not compile this code, only for documentation purposes
210 #define GPIO_HYBRID_SELECTOR  IS  ADC_CHANNEL_6  ON GPIO_NUM_34
211 #endif
212
213 ////////////////////////////////////////////////////////////////////////////
214 ////                         GLOBAL STRUCTURE TYPES                     ////
215 ////////////////////////////////////////////////////////////////////////////
216 /*
217     * @brief: This structure contains the data that is displayed on the lcd screen
218     *         as well as the data that is logged on the sdcard.
219     * @note: The data is real-time, meaning that all of it, besides the accelerometer
220     *        and current_page fields, the rest of the data is updated at the
```

```
221     *        CAN_POLLING_RATE frequency. The accelerometer data is updated at
222     *        the MPU6050_POLLING_RATE frequency.
223     *
224     *        This approach has data loss if the frequency of the SDCARD logging rate is
225     *        lower than CAN_POLLING_RATE or MPU6050_POLLING_RATE.
226     *
227 */
228 typedef struct display_data_t {
229
230     /* Keeps track of the current_page shown on the display */
231     uint8_t current_page;
232
233
234     /*************************************************************
235      *                    MAIN DISPLAY PAGE                      *
236      *************************************************************/
237     /* Expected rpm is between 0 - 12.000 (uint16_t max 65.536) */
238     uint16_t rpm;
239
240     /* Current gear is between 0 - 5 (uint8_t max 255) */
241     uint8_t current_gear;
242
243     /* Coolant temperature is between 0 - 100 (uint8_t max 255) */
244     uint8_t coolant_temperature;
245
246     /* Radiator Fan state is either true for ON or false for OFF */
247     uint8_t fan_state;
248
249     /* Oil temperature value is between 0 - 150 (uint8_t max 255) */
250     uint8_t oil_temperature;
251
252
253     /*************************************************************
254      *              SECOND DISPLAY PAGE for debug                *
255      *************************************************************/
256     /* CAN bus status is true for ON or false for FAULT */
257     uint8_t can_status;
258
259     /* Hybrid system status is true for OK or false for FAULT */
260     uint8_t hybrid_status;
261
262     /* Safety circuit status is true for OK or false for FAULT */
263     uint8_t safety_circuit_status;
264
265     /* Oil pressure value is between 0 - 150 (uint8_t max 255) */
266     uint8_t oil_pressure;
267
268     /* Raw bps value from the sensor ToDo */
269     uint8_t brake_pressure_raw;
270
271     /* Throttle Position Sensor value is between 0 - 100 (uint8_t max 255) */
272     uint8_t tps;
273
274     /* Oil temperature value is between 0 - 150 (uint8_t max 255)
275
276        @note: the field oil_temperature is scaled before being sent to MAIN_PAGE
277               this field will contain the unscaled version of oil_temperature from
278               the CAN message
279     */
280     uint8_t oil_temperature_2;
281
282     /* Hybrid status value HYBRID_STATE_1 - HYBRID_STATE_11 or HYBRID_STATE_ERROR */
283     uint8_t hybrid_selector_value;
284
285     /* Battery voltage is between 0 - 240 (uint8_t max 255) */
286     uint8_t battery_voltage;
287
288
289     /*************************************************************
290     *                 THIRD DISPLAY PAGE for debug                *
291     *************************************************************/
292     /* Lambda value between 0 - 155 (uint8_t max 255) */
293     uint8_t lambda;
294
295     /* Manifold Absolute Pressure MAP value 0 - 45 (uint8_t max 255) */
296     uint8_t map;
```

```
297
298     /* Fuel pressure value is between 0 - 45 (uint8_t max 255) */
299     uint8_t fuel_pressure;
300
301     /* EGT values between 0 - 450 (uint8_t max 255) */
302     uint16_t egt[4];
303
304     /* Voltage on the PMU input rail 0 - 14 (uint8_t max 255) */
305     uint8_t input_voltage_pmu;
306
307     /* Input current from alternator 0 - 10 (uint8_t max 255) */
308     uint8_t input_current_altr_pmu;
309
310
311     /*************************************************************
312      *             FOURTH DISPLAY PAGE for debug                 *
313      *************************************************************/
314     /* Tyre pressure values for each wheel between 0 - 30 bar (uint8_t max 255)*/
315     uint8_t tyre_pressure[4];
316
317
318     /*************************************************************
319      *                     NOT IN A PAGE                         *
320      *************************************************************/
321
322     /* Raw accelerometer value on X, Y and Z, note that the max value is
323        adjusted considering the MPU6050_ACCEL_SCALE macro. The formula is
324        ((accelerometer[i] / 2^15) * MPU6050_ACCEL_SCALE). It is divided to
325        2^15 since the values are integer ones on 16 bit, so the max value of
326        int16_t is 2^15 */
327     int16_t accelerometer[MPU6050_AXIS_NUM];
328
329     /* Accelerometer data computed in g's */
330     float accelerometer_g[MPU6050_AXIS_NUM];
331
332     /* Intake Air Temperature (IAT) is between 0 - 100 (uint8_t max 255) */
333     uint8_t iat;
334
335 } display_data_t ;
336
337 /*
338      * @brief This structure contains pointers to other relevant data structures
339      *        that are used in the firmware as well as to the Semaphores that
340      *        are used to handle concurrent access to them.
341      *
342      * @param uint8_t display_page_num: number of the display page
343      *
344 */
345 typedef struct status_firmware_t {
346
347     /* pointer to the structure containing all the data to be
348        send to the display and logged on the sdcard */
349     display_data_t *display_data;
350
351     /* gloabl semaphore handle for the display data struct that will handle
352        concurrent access requests from the:
353        - can_send   task
354        - lcd_update task
355        - can_read   task
356        - sdcard_write task
357        - neutral_led_update task
358        - shift_strip_led_update task
359        - hybrid_selector_read task
360        - acceleromter_read task */
361     SemaphoreHandle_t xSemaphore_display_data;
362
363     /* flag that signals a change_page request. It is handled in lcd_update task */
364     uint8_t signal_change_page;
365
366     /* Store the time from the RTC of the car */
367     /* time in seconds */
368     uint8_t time_second;
369
370     /* time in minutes */
371     uint8_t time_minute;
372
```

```
373     /* time in hours */
374     uint8_t time_hour;
375 
376     /* sdcard_logging state is either true for OK or false for ERROR */
377     uint8_t sdcard_logging;
378 
379 } status_firmware_t ;
380 
381 #endif /* COMMON_TYPES_H */
```

components/gpio/button/inc/button_task.h

```
1  /*****************************************************************************/
2  /* UPBDRIVE Dashboard Firmware                                               */
3  /* All rights reserverd 2023                                                 */
4  /*****************************************************************************/
5  
6  #ifndef BUTTONS_TASK_H
7  #define BUTTONS_TASK_H
8  
9  #include <common.h>
10 
11 
12 ////////////////////////////////////////////////////////////////////////////
13 ////                        GLOBAL VARIABLES                            ////
14 ////////////////////////////////////////////////////////////////////////////
15 
16 ////////////////////////////////////////////////////////////////////////////
17 ////                        GLOBAL MACROS                               ////
18 ////////////////////////////////////////////////////////////////////////////
19 
20 ////////////////////////////////////////////////////////////////////////////
21 ////                        GLOBAL CONSTANTS                            ////
22 ////////////////////////////////////////////////////////////////////////////
23 
24 ////////////////////////////////////////////////////////////////////////////
25 ////                        FUNCTION PROTOYPES                          ////
26 ////////////////////////////////////////////////////////////////////////////
27 
28 /*
29     * @brief This function will setup the LCD change page button and ventilator ON/OFF button
30     *
31     * @param void
32     *
33     * @return esp_response
34 
35 */
36 esp_err_t buttons_setup(void);
37 
38 /*
39     * @brief This task will poll the LCD change page button and ventilator ON/OFF button at
       BUTTON_POLLING_RATE refresh
40     *        rate
41     *
42     * @param[in] pvParameters IN
43     *
44     * @return void
45 */
46 void buttons_read(void *pvParameters);
47 
48 
49 #endif /* BUTTONS_TASK_H */
```

components/gpio/button/button_task.c

```
1  /*****************************************************************************/
2  /* UPBDRIVE Dashboard Firmware                                               */
3  /* All rights reserverd 2023                                                 */
4  /*****************************************************************************/
5  
6  #include <buttons_task.h>
7  
8  ////////////////////////////////////////////////////////////////////////////
9  ////                        GLOBAL VARIABLES                            ////
10 ////////////////////////////////////////////////////////////////////////////
11 
12 extern status_firmware_t general_status;
```

```
13 extern display_data_t    display_data;
14 extern uint8_t change_page_ready;
15
16 ////////////////////////////////////////////////////////////////////////////
17 ////                          GLOBAL FUNCTIONS                          ////
18 ////////////////////////////////////////////////////////////////////////////
19
20 ////////////////////////////////////////////////////////////////////////////
21 ////                          LOCAL MACROS                              ////
22 ////////////////////////////////////////////////////////////////////////////
23
24 ////////////////////////////////////////////////////////////////////////////
25 ////                          LOCAL CONSTANTS                           ////
26 ////////////////////////////////////////////////////////////////////////////
27
28 ////////////////////////////////////////////////////////////////////////////
29 ////                          FUNCTION DEFINITIONS                      ////
30 ////////////////////////////////////////////////////////////////////////////
31
32 void IRAM_ATTR button_change_page_isr(void)
33 {
34     /* Software debounce consisting in:
35        - taking into consideration bouncing effect of ns order
36        - gpio level should be 0 if the button is pressed
37        - bouncing effect happens at button release, mitigated with gpio level
38        - should change page at a defined speed, does not care if button is pressed faster */
39     if ((gpio_get_level(GPIO_BUTTON_LCD_PAGE) == 0) && (change_page_ready == 1))
40     {
41         general_status.signal_change_page = 0xFF;
42     }
43 }
44
45 void IRAM_ATTR button_rad_fan_isr(void)
46 {
47     /* Recheck the status of radiator button if it is ON or OFF */
48     if(gpio_get_level(GPIO_BUTTON_RAD_FAN) == 1)
49     {
50         display_data.fan_state = true;
51     }
52     else
53     {
54         display_data.fan_state = false;
55     }
56 }
57
58 void IRAM_ATTR button_safety_circ_isr(void)
59 {
60     /* Recheck the status of safety circ button if it is ON or OFF */
61     if(gpio_get_level(GPIO_BUTTON_SAFETY_CIRC) == 1)
62     {
63         display_data.safety_circuit_status = true;
64     }
65     else
66     {
67         display_data.safety_circuit_status = false;
68     }
69 }
70
71 esp_err_t buttons_setup(void)
72 {
73     esp_err_t esp_response = ESP_FAIL;
74
75     /* When the button is pressed, gpio pin is connected with 2.71V, therefore,
76        enable PULLDOWN when for the button not pressed state */
77     gpio_config_t gpio_button_rad_fan =
78     {
79         .pin_bit_mask = (1ULL << GPIO_BUTTON_RAD_FAN),
80         .mode         = GPIO_MODE_INPUT,
81         .pull_up_en   = GPIO_PULLUP_DISABLE,
82         .pull_down_en = GPIO_PULLDOWN_ENABLE,
83         .intr_type    = GPIO_INTR_ANYEDGE
84     };
85
86     /* When the button is pressed, gpio pin is connected with GND, therefore,
87        enable PULLUP when for the button not pressed state */
88     gpio_config_t gpio_button_lcd_page =
```

```
89      {
90          .pin_bit_mask = (1ULL << GPIO_BUTTON_LCD_PAGE),
91          .mode         = GPIO_MODE_INPUT,
92          .pull_up_en   = GPIO_PULLUP_ENABLE,
93          .pull_down_en = GPIO_PULLDOWN_DISABLE,
94          .intr_type    = GPIO_INTR_NEGEDGE
95      };
96
97      /* When the button is pressed, gpio pin is connected with 2.71V */
98      gpio_config_t gpio_button_safety_circuit =
99      {
100         .pin_bit_mask = (1ULL << GPIO_BUTTON_SAFETY_CIRC),
101         .mode         = GPIO_MODE_INPUT,
102         .pull_up_en   = GPIO_PULLUP_DISABLE,
103         .pull_down_en = GPIO_PULLDOWN_DISABLE,
104         .intr_type    = GPIO_INTR_ANYEDGE
105     };
106
107     ESP_ERROR_CHECK(gpio_config(&gpio_button_rad_fan));
108
109     ESP_ERROR_CHECK(gpio_config(&gpio_button_lcd_page));
110
111     ESP_ERROR_CHECK(gpio_config(&gpio_button_safety_circuit));
112
113     ESP_ERROR_CHECK(gpio_install_isr_service(ESP_INTR_FLAG_IRAM));
114
115     ESP_ERROR_CHECK(gpio_isr_handler_add(GPIO_BUTTON_LCD_PAGE, (gpio_isr_t)
        button_change_page_isr, NULL));
116
117     ESP_ERROR_CHECK(gpio_isr_handler_add(GPIO_BUTTON_RAD_FAN, (gpio_isr_t)button_rad_fan_isr,
        NULL));
118
119     ESP_ERROR_CHECK(gpio_isr_handler_add(GPIO_BUTTON_SAFETY_CIRC, (gpio_isr_t)
        button_safety_circ_isr, NULL));
120
121     /* Avoid unknown boot-up state for SAFETY CIRCUIT */
122     if(gpio_get_level(GPIO_BUTTON_SAFETY_CIRC) == 1)
123     {
124         display_data.safety_circuit_status = true;
125     }
126     else
127     {
128         display_data.safety_circuit_status = false;
129     }
130
131     esp_response = ESP_OK;
132     return esp_response;
133 }
134
135 #ifdef ENABLE_DEBUG_BUTTONS
136 void buttons_read(void *pvParameters)
137 {
138     while(true)
139     {
140
141         /* Debug print for button change page */
142         if(general_status.signal_change_page == 0xFF)
143         {
144             #ifdef ENABLE_DEBUG_DISPLAY
145             general_status.signal_change_page = 0x00;
146             #endif /* ENABLE_DEBUG_DISPLAY */
147
148             printf("Button change page!\n");
149         }
150
151         /* Debug print for fan state button */
152         if(display_data.fan_state == true)
153         {
154             printf("Fan is ON!\n");
155         }
156
157         vTaskDelay(1000 / portTICK_PERIOD_MS);
158     }
159 }
160 #endif /* ENABLE_DEBUG_BUTTONS */
```

components/gpio/led/inc/heartbeat_task.h

```c
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/
#ifndef HEARTBEAT_TASK_H
#define HEARTBEAT_TASK_H

#include <common.h>
#include <driver/gpio.h>
#include <stdio.h>

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL MACROS                                   ////
////////////////////////////////////////////////////////////////////////////////

#define HEARTBEAT_LED_RATE  (1000/portTICK_PERIOD_MS)

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL CONSTANTS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        FUNCTION PROTOYPES                              ////
////////////////////////////////////////////////////////////////////////////////

/*
    * @brief This function will setup the gpio used for heartbeat led as an output
    *
    * @param void
    *
    * @return esp_err_t
*/
esp_err_t heartbeat_setup(void);

/*
    * @brief This task will update the heartbeat 1HZ refresh rate
    *
    * @param[in] pvParameters IN
    *
    * @return void
*/
void heartbeat(void *pvParameters);

#endif /* HEARTBEAT_TASK_H */
```

components/gpio/led/heartbeat_task.c

```c
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#include <heartbeat_task.h>

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL FUNCTIONS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL MACROS                                    ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL CONSTANTS                                 ////
////////////////////////////////////////////////////////////////////////////////


```

```
esp_err_t heartbeat_setup(void)
{
    esp_err_t esp_response = ESP_FAIL;
    gpio_config_t gpio_led_status =
    {
        .pin_bit_mask = (1ULL << GPIO_LED_STATUS),
        .mode         = GPIO_MODE_OUTPUT,
        .pull_up_en   = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_ENABLE,
        .intr_type    = GPIO_INTR_DISABLE
    };

    ESP_ERROR_CHECK(gpio_config(&gpio_led_status));

    esp_response = ESP_OK;
    return esp_response;
}


void heartbeat(void *pvParameters)
{
    uint8_t heartbeat_led_state = 0;

    while(true)
    {
        ESP_ERROR_CHECK(gpio_set_level(GPIO_LED_STATUS, heartbeat_led_state));

        /* toggle the level of the heartbeat */
        heartbeat_led_state = !heartbeat_led_state;

        /* delay the loop for debugging purposes */
        vTaskDelay(HEARTBEAT_LED_RATE);
    }
}
```

components/gpio/led/inc/led_strip_encoder.h

```
/*
 * SPDX-FileCopyrightText: 2021-2022 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#pragma once

#include <stdint.h>
#include <driver/rmt_encoder.h>

#ifdef __cplusplus
extern "C" {
#endif

#define SHIFTLED_THRESHOLD  (SHIFT_THRESHOLD)

/**
 * @brief Type of led strip encoder configuration
 */
typedef struct {
    uint32_t resolution; /*!< Encoder resolution, in Hz */
} led_strip_encoder_config_t;

/**
 * @brief Create RMT encoder for encoding LED strip pixels into RMT symbols
 *
 * @param[in] config Encoder configuration
 * @param[out] ret_encoder Returned encoder handle
 * @return
 *      - ESP_ERR_INVALID_ARG for any invalid arguments
 *      - ESP_ERR_NO_MEM out of memory when creating led strip encoder
 *      - ESP_OK if creating encoder successfully
 */
esp_err_t rmt_new_led_strip_encoder(const led_strip_encoder_config_t *config,
    rmt_encoder_handle_t *ret_encoder);

#ifdef __cplusplus
}
#endif
```

components/gpio/led/led_strip_encoder.c

```c
/*
 * SPDX-FileCopyrightText: 2021-2022 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <esp_check.h>
#include <led_strip_encoder.h>

static const char *TAG = "led_encoder";
typedef struct {
    rmt_encoder_t base;
    rmt_encoder_t *bytes_encoder;
    rmt_encoder_t *copy_encoder;
    int state;
    rmt_symbol_word_t reset_code;
} rmt_led_strip_encoder_t;

rmt_led_strip_encoder_t led_encoder = {0};

static size_t rmt_encode_led_strip(rmt_encoder_t *encoder, rmt_channel_handle_t channel, const void *primary_data, size_t data_size, rmt_encode_state_t *ret_state)
{
    rmt_led_strip_encoder_t *led_encoder = __containerof(encoder, rmt_led_strip_encoder_t, base);
    rmt_encoder_handle_t bytes_encoder = led_encoder->bytes_encoder;
    rmt_encoder_handle_t copy_encoder = led_encoder->copy_encoder;
    rmt_encode_state_t session_state = RMT_ENCODING_RESET;
    rmt_encode_state_t state = RMT_ENCODING_RESET;
    size_t encoded_symbols = 0;
    switch (led_encoder->state) {
    case 0: // send RGB data
        encoded_symbols += bytes_encoder->encode(bytes_encoder, channel, primary_data, data_size, &session_state);
        if (session_state & RMT_ENCODING_COMPLETE) {
            led_encoder->state = 1; // switch to next state when current encoding session finished
        }
        if (session_state & RMT_ENCODING_MEM_FULL) {
            state |= RMT_ENCODING_MEM_FULL;
            goto out; // yield if there's no free space for encoding artifacts
        }
    // fall-through
    case 1: // send reset code
        encoded_symbols += copy_encoder->encode(copy_encoder, channel, &led_encoder->reset_code,
                                                sizeof(led_encoder->reset_code), &session_state);
        if (session_state & RMT_ENCODING_COMPLETE) {
            led_encoder->state = RMT_ENCODING_RESET; // back to the initial encoding session
            state |= RMT_ENCODING_COMPLETE;
        }
        if (session_state & RMT_ENCODING_MEM_FULL) {
            state |= RMT_ENCODING_MEM_FULL;
            goto out; // yield if there's no free space for encoding artifacts
        }
    }
out:
    *ret_state = state;
    return encoded_symbols;
}

static esp_err_t rmt_del_led_strip_encoder(rmt_encoder_t *encoder)
{
    rmt_led_strip_encoder_t *led_encoder = __containerof(encoder, rmt_led_strip_encoder_t, base);
    rmt_del_encoder(led_encoder->bytes_encoder);
    rmt_del_encoder(led_encoder->copy_encoder);
    free(led_encoder);
    return ESP_OK;
}

static esp_err_t rmt_led_strip_encoder_reset(rmt_encoder_t *encoder)
{
```

```
    rmt_led_strip_encoder_t *led_encoder = __containerof(encoder, rmt_led_strip_encoder_t, base);
    rmt_encoder_reset(led_encoder->bytes_encoder);
    rmt_encoder_reset(led_encoder->copy_encoder);
    led_encoder->state = RMT_ENCODING_RESET;
    return ESP_OK;
}

esp_err_t rmt_new_led_strip_encoder(const led_strip_encoder_config_t *config, rmt_encoder_handle_t *ret_encoder)
{
    esp_err_t ret = ESP_OK;

    ESP_GOTO_ON_FALSE(config && ret_encoder, ESP_ERR_INVALID_ARG, err, TAG, "invalid argument");
    led_encoder.base.encode = rmt_encode_led_strip;
    led_encoder.base.del = rmt_del_led_strip_encoder;
    led_encoder.base.reset = rmt_led_strip_encoder_reset;
    // different led strip might have its own timing requirements, following parameter is for WS2812
    rmt_bytes_encoder_config_t bytes_encoder_config = {
        .bit0 = {
            .level0 = 1,
            .duration0 = 0.3 * config->resolution / 1000000, // T0H=0.3us
            .level1 = 0,
            .duration1 = 0.9 * config->resolution / 1000000, // T0L=0.9us
        },
        .bit1 = {
            .level0 = 1,
            .duration0 = 0.9 * config->resolution / 1000000, // T1H=0.9us
            .level1 = 0,
            .duration1 = 0.3 * config->resolution / 1000000, // T1L=0.3us
        },
        .flags.msb_first = 1 // WS2812 transfer bit order: G7...G0R7...R0B7...B0
    };
    ESP_GOTO_ON_ERROR(rmt_new_bytes_encoder(&bytes_encoder_config, &led_encoder.bytes_encoder), err, TAG, "create bytes encoder failed");
    rmt_copy_encoder_config_t copy_encoder_config = {};
    ESP_GOTO_ON_ERROR(rmt_new_copy_encoder(&copy_encoder_config, &led_encoder.copy_encoder), err, TAG, "create copy encoder failed");

    uint32_t reset_ticks = config->resolution / 1000000 * 50 / 2; // reset code duration defaults to 50us
    led_encoder.reset_code = (rmt_symbol_word_t) {
        .level0 = 0,
        .duration0 = reset_ticks,
        .level1 = 0,
        .duration1 = reset_ticks,
    };
    *ret_encoder = &led_encoder.base;
    return ESP_OK;
err:
    return ret;
}
```

components/gpio/led/inc/led_task.h

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/
#ifndef SHIFTLIGHT_TASK_H
#define SHIFTLIGHT_TASK_H

#include <common.h>
#include <driver/gpio.h>
#include <esp_log.h>
#include <driver/rmt_tx.h>
#include <led_strip_encoder.h>

////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                            ////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
////                        GLOBAL MACROS                               ////
////////////////////////////////////////////////////////////////////////////
```

```
21
22 #define SHIFTLED_THRESHOLD  (SHIFT_THRESHOLD)
23
24 ////////////////////////////////////////////////////////////////////////////
25 ////                        GLOBAL CONSTANTS                            ////
26 ////////////////////////////////////////////////////////////////////////////
27
28 ////////////////////////////////////////////////////////////////////////////
29 ////                        FUNCTION PROTOYPES                          ////
30 ////////////////////////////////////////////////////////////////////////////
31
32 /*
33     * @brief This function will setup the shift led and neutral led GPIOs as output
34     *
35     * @param void
36     *
37     * @return esp_err_t
38 */
39 esp_err_t shift_neutral_led_setup(void);
40
41 /*
42     * @brief This function will setup the adafruit shift led strip GPIO as output
43     *
44     * @param void
45     *
46     * @return esp_err_t
47 */
48 esp_err_t shift_strip_led_setup(void);
49
50 /*
51     * @brief This task will update the the neutral led based on the gear @4Hz
52     *
53     * @param[in] pvParameters
54     *
55     * @return esp_err_t
56 */
57 void neutral_led_update(void *pvParameters);
58
59 /*
60     * @brief This task will update the shift strip at 30HZ refresh rate
61     *
62     * @param pvParameters IN
63     *
64     * @return void
65 */
66 void shift_strip_led_update(void *pvParameters);
67
68
69 #endif /* BUTTON_TASK_H */
```

components/gpio/led/led_task.c

```
1 /*****************************************************************************/
2 /* UPBDRIVE Dashboard Firmware                                               */
3 /* All rights reserverd 2023                                                 */
4 /*****************************************************************************/
5
6 #include <led_task.h>
7
8 ////////////////////////////////////////////////////////////////////////////
9 ////                        GLOBAL VARIABLES                            ////
10 ////////////////////////////////////////////////////////////////////////////
11
12 ////////////////////////////////////////////////////////////////////////////
13 ////                        GLOBAL FUNCTIONS                            ////
14 ////////////////////////////////////////////////////////////////////////////
15
16 ////////////////////////////////////////////////////////////////////////////
17 ////                        LOCAL MACROS                                ////
18 ////////////////////////////////////////////////////////////////////////////
19
20 #define RMT_LED_STRIP_RESOLUTION_HZ (10000000) // 10MHz resolution, 1 tick = 0.1us (led strip
       needs a high resolution)
21
22 #define STRIP_LED_NUMBER                (8)
23 #define EXAMPLE_CHASE_SPEED_MS          (0)
24
```

```
/*****************************************************************************
 *                            LED STRIP RPM CASE                             *
 * @note: The maximum RPM of the vehicle is 12000 RPM, the LED strip has 8 LEDs*
 *        therefore an LED might indicate a value <= 12000 / 8 = 1500 RPM     *
 *****************************************************************************/

#define LOWER_THAN_1500             (0)
#define BETWEEN_1500_3000           (1)
#define BETWEEN_3000_4500           (2)
#define BETWEEN_4500_6000           (3)
#define BETWEEN_6000_7500           (4)
#define BETWEEN_7500_9000           (5)
#define BETWEEN_9000_10500          (6)
#define BETWEEN_10500_12000         (7)
#define IS_12000                    (8)

////////////////////////////////////////////////////////////////////////////
////                        LOCAL CONSTANTS                             ////
////////////////////////////////////////////////////////////////////////////
static rmt_channel_handle_t led_chan          = NULL;
static  rmt_tx_channel_config_t tx_chan_config =
{
        .clk_src            = RMT_CLK_SRC_DEFAULT, // select source clock
        .gpio_num           = GPIO_LED_STRIP,
        .mem_block_symbols  = 128,                 // increase the block size can make the LED
     less flickering
        .resolution_hz      = RMT_LED_STRIP_RESOLUTION_HZ,
        .trans_queue_depth  = 4, // set the number of transactions that can be pending in the
    background
};
static rmt_encoder_handle_t led_encoder        = NULL;
static rmt_transmit_config_t tx_config = {
        .loop_count = 0, // no transfer loop
};


esp_err_t shift_neutral_led_setup(void)
{
    esp_err_t esp_response = ESP_FAIL;

    /* Configuration of the neutral led gpio */
    gpio_config_t gpio_neutral =
    {
        .pin_bit_mask = (1ULL << GPIO_LED_NEUTRAL),
        .mode         = GPIO_MODE_INPUT,
        .pull_up_en   = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type    = GPIO_INTR_DISABLE
    };

    /* Configure the gpio based on the struct */
    ESP_ERROR_CHECK(gpio_config(&gpio_neutral));

    esp_response = ESP_OK;
    return esp_response;
}

esp_err_t shift_strip_led_setup(void)
{
    esp_err_t esp_response = ESP_FAIL;

    ESP_ERROR_CHECK(rmt_new_tx_channel(&tx_chan_config, &led_chan));

    led_strip_encoder_config_t encoder_config =
    {
        .resolution = RMT_LED_STRIP_RESOLUTION_HZ,
    };

    ESP_ERROR_CHECK(rmt_new_led_strip_encoder(&encoder_config, &led_encoder));

    ESP_ERROR_CHECK(rmt_enable(led_chan));

    esp_response = ESP_OK;
    return esp_response;
}

```

```
99  void neutral_led_update(void *pvParameters)
100 {
101     status_firmware_t *general_status           = (status_firmware_t*) pvParameters;
102     display_data_t  *display_data               = general_status->display_data;
103     SemaphoreHandle_t xSemaphore_display_data   = general_status->xSemaphore_display_data;
104
105     while(true)
106     {
107         if (gpio_get_level(GPIO_LED_NEUTRAL) == 0)
108         {
109             if( xSemaphoreTake( xSemaphore_display_data, ( TickType_t ) 10 ) == pdTRUE )
110             {
111                 display_data->current_gear = NEUTRAL_GEAR;
112                 xSemaphoreGive( xSemaphore_display_data );
113             }
114         }
115
116         vTaskDelay(NEUTRAL_LED_RATE);
117     }
118 }
119
120 void shift_strip_led_update(void *pvParameters)
121 {
122     status_firmware_t *general_status           = (status_firmware_t*) pvParameters;
123     display_data_t  *display_data               = general_status->display_data;
124     SemaphoreHandle_t xSemaphore_display_data   = general_status->xSemaphore_display_data;
125
126     /* By using this local stack variables, one can give faster the semaphore
127        for the global struct display_data */
128     uint16_t rpm                                = 0U;
129     uint8_t rpm_case                            = 0U;
130
131     /* There are 3 bytes needed for each LED:
132        first    byte for GREEN
133        second   byte for BLUE
134        third    byte for RED */
135     uint8_t led_strip_pixels[STRIP_LED_NUMBER * 3]  = {0U};
136
137     while(true)
138     {
139         /* Set the data about the LEDs to OFF (0U) */
140         memset(led_strip_pixels, 0U, sizeof(led_strip_pixels));
141
142         if( xSemaphoreTake( xSemaphore_display_data, ( TickType_t ) 10 ) == pdTRUE )
143         {
144             /* Find the number of leds on the strip to be lit based on the current rpm */
145             rpm = display_data->rpm;
146             xSemaphoreGive( xSemaphore_display_data );
147
148             /* Sanitize the rpm, keep it capped to 12000 RPM */
149             if (rpm > 12000U)
150             {
151                 rpm = 12000U;
152             }
153
154             rpm_case = rpm / 1500;
155
156             switch(rpm_case)
157             {
158                 case LOWER_THAN_1500:
159                 {
160                     /*  There are 3 values corresponding to the first LED
161                         led_strip_pixels[0] encoding GREEN
162                         led_strip_pixels[1] encoding BLUE
163                         led_strip_pixels[2] encoding RED
164
165                         then the values
166                         led_strip_pixels[3]
167                         led_strip_pixels[4]
168                         led_strip_pixels[5] correspond to the second LED and so on
169                     */
170                     /* Set the first LED in the STRIP to be GREEN */
171                     led_strip_pixels[0] = 255;
172                     break;
173                 }
174                 case BETWEEN_1500_3000:
```

```
175                 {
176                     /* Set the first LED on the STRIP to be GREEN */
177                     led_strip_pixels[0] = 255;
178                     break;
179                 }
180                 case BETWEEN_3000_4500:
181                 {
182                     /* Set the first LED on the STRIP to be GREEN */
183                     led_strip_pixels[0] = 255;
184                     /* Set the second LED on the STRIP to be GREEN */
185                     led_strip_pixels[3] = 255;
186                     break;
187                 }
188                 case BETWEEN_4500_6000:
189                 {
190                     /* Set the first LED on the STRIP to be GREEN */
191                     led_strip_pixels[0] = 255;
192                     /* Set the second LED on the STRIP to be GREEN */
193                     led_strip_pixels[3] = 255;
194                     /* Set the third LED on the STRIP to be GREEN */
195                     led_strip_pixels[6] = 255;
196                     break;
197                 }
198                 case BETWEEN_6000_7500:
199                 {
200                     /* Set the first LED on the STRIP to be GREEN */
201                     led_strip_pixels[0] = 255;
202                     /* Set the second LED on the STRIP to be GREEN */
203                     led_strip_pixels[3] = 255;
204                     /* Set the third LED on the STRIP to be GREEN */
205                     led_strip_pixels[6] = 255;
206                     /* Set the fourth LED on the STRIP to be BLUE */
207                     led_strip_pixels[11] = 255;
208                     break;
209                 }
210                 case BETWEEN_7500_9000:
211                 {
212                     /* Set the first LED on the STRIP to be GREEN */
213                     led_strip_pixels[0] = 255;
214                     /* Set the second LED on the STRIP to be GREEN */
215                     led_strip_pixels[3] = 255;
216                     /* Set the third LED on the STRIP to be GREEN */
217                     led_strip_pixels[6] = 255;
218                     /* Set the fourth LED on the STRIP to be BLUE */
219                     led_strip_pixels[11] = 255;
220                     /* Set the fifth LED on the STRIP to be BLUE */
221                     led_strip_pixels[14] = 255;
222                     break;
223                 }
224                 case BETWEEN_9000_10500:
225                 {
226                     /* Set the first LED on the STRIP to be GREEN */
227                     led_strip_pixels[0] = 255;
228                     /* Set the second LED on the STRIP to be GREEN */
229                     led_strip_pixels[3] = 255;
230                     /* Set the third LED on the STRIP to be GREEN */
231                     led_strip_pixels[6] = 255;
232                     /* Set the fourth LED on the STRIP to be BLUE */
233                     led_strip_pixels[11] = 255;
234                     /* Set the fifth LED on the STRIP to be BLUE */
235                     led_strip_pixels[14] = 255;
236                     /* Set the sixth LED on the STRIP to be BLUE */
237                     led_strip_pixels[17] = 255;
238 
239                     break;
240                 }
241                 case BETWEEN_10500_12000:
242                 {
243                     /* Set the first LED on the STRIP to be GREEN */
244                     led_strip_pixels[0] = 255;
245                     /* Set the second LED on the STRIP to be GREEN */
246                     led_strip_pixels[3] = 255;
247                     /* Set the third LED on the STRIP to be GREEN */
248                     led_strip_pixels[6] = 255;
249                     /* Set the fourth LED on the STRIP to be BLUE */
250                     led_strip_pixels[11] = 255;
```

```
                    /* Set the fifth LED on the STRIP to be BLUE */
                    led_strip_pixels[14] = 255;
                    /* Set the sixth LED on the STRIP to be BLUE */
                    led_strip_pixels[17] = 255;
                    /* Set the seventh LED on the STRIP to be RED */
                    led_strip_pixels[19] = 255;
                    break;
                }

                case IS_12000:
                {
                    /* Set the first LED on the STRIP to be GREEN */
                    led_strip_pixels[0] = 255;
                    /* Set the second LED on the STRIP to be GREEN */
                    led_strip_pixels[3] = 255;
                    /* Set the third LED on the STRIP to be GREEN */
                    led_strip_pixels[6] = 255;
                    /* Set the fourth LED on the STRIP to be BLUE */
                    led_strip_pixels[11] = 255;
                    /* Set the fifth LED on the STRIP to be BLUE */
                    led_strip_pixels[14] = 255;
                    /* Set the sixth LED on the STRIP to be BLUE */
                    led_strip_pixels[17] = 255;
                    /* Set the seventh LED on the STRIP to be RED */
                    led_strip_pixels[19] = 255;
                    /* Set the eighth LED on the STRIP to be RED */
                    led_strip_pixels[22] = 255;
                }
                default:
                {
                    break;
                }
            }

            /* Send the values to the led strip */
            ESP_ERROR_CHECK(rmt_transmit(led_chan, led_encoder, led_strip_pixels, sizeof(
    led_strip_pixels), &tx_config));
            ESP_ERROR_CHECK(rmt_tx_wait_all_done(led_chan, portMAX_DELAY));

            /* Blink the LED strip if the RPM is in the good shift range */
            if (rpm_case >= BETWEEN_10500_12000 && rpm_case <= IS_12000)
            {
                /* Wait before turning off the strip */
                vTaskDelay(SHIFTLED_STRIP_RATE);

                /* Set the led strip to OFF */
                memset(led_strip_pixels, 0U, sizeof(led_strip_pixels));
                ESP_ERROR_CHECK(rmt_transmit(led_chan, led_encoder, led_strip_pixels, sizeof(
    led_strip_pixels), &tx_config));
                ESP_ERROR_CHECK(rmt_tx_wait_all_done(led_chan, portMAX_DELAY));
            }
        }

        /* Update the SHIFT STRIP at SHIFTLED_STRIP_RATE set in common_types */
        vTaskDelay(SHIFTLED_STRIP_RATE);
    }
}
```

components/i2c/accelerometer-mpu6050/inc/accelerometer_task.h

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#ifndef ACCELEROMETER_TASK_H
#define ACCELEROMETER_TASK_H

#include <common.h>
#include <driver/i2c.h>

////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                            ////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
////                        GLOBAL MACROS                               ////
```

```
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL CONSTANTS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        FUNCTION PROTOYPES                              ////
////////////////////////////////////////////////////////////////////////////////

/*
    * @brief This function will setup the I2C bus for communication with MPU6050
    *
    * @param void
    *
    * @return void
*/
esp_err_t accelerometer_setup(void);

/*
    * @brief This task will read the accelerometer data
    *
    * @param[in] pvParameters IN
    *
    * @return void
*/
void accelerometer_read(void *pvParameters);

#endif /* ACCELEROMETER_TASK_H */
```

components/i2c/accelerometer-mpu6050/accelerometer_task.c

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#include <accelerometer_task.h>

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL VARIABLES                                 ////
////////////////////////////////////////////////////////////////////////////////

static i2c_port_t i2c_master_port = I2C_NUM_0;

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL FUNCTIONS                                 ////
////////////////////////////////////////////////////////////////////////////////

static esp_err_t accelerometer_reset(void);

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL MACROS                                    ////
////////////////////////////////////////////////////////////////////////////////

/* i2c BUS speed */
#define I2C_FREQ_400khz             (400 * 1000)

/* MPU6050 i2c address */
#define MPU6050_I2C_ADDRESS         (0x68)

/* MPU6050 register addresses */
/* Bits in MPU6050_PWR_MGMT_1
   [7]: DEVICE_RESET, when set to 1 resets all internal registers to default value
   [6]: SLEEP, when set to 1 MPU6050 enters into sleep mode
   [5]: CYCLE, when set to 1 MPU6050 cycles between sleep mode and waking up to
        take a single sample of data from active sensors
   [4]: - not used
   [3]: TEMP_DIS, when set to 1 disables the temperature sensor
   [2]: CLKSEL, specifies the clock source of the device, when set to 0 the internal
                oscillator of 8 MHz is used
   [1]: CLKSEL
   [0]: CLKSEL
*/
#define MPU6050_PWR_MGMT_1          (0x6B)
/* Bits in MPU6050_SIGNAL_PATH_RESET
   [7]: - not used
```

```
   [6]: - not used
   [5]: - not used
   [4]: - not used
   [3]: - not used
   [2]: GYRO_RESET, when set to 1 resets the gyro sens analog and digital parts
   [1]: ACCEL_RESET, when set to 1 resets the accel sens analog and digital parts
   [0]: TEMP_RESET, when set to 1 resets the temp sens analog and digital parts
*/
#define MPU6050_SIGNAL_PATH_RESET   (0x68)
/* Bits in MPU6050_ACCEL_CONFIG
   [7]: XA_ST, when set to 1 the self-test of the accel on X axis is enabled
   [6]: YA_ST, when set to 1 the self-test of the accel on Y axis is enabled
   [5]: ZA_ST, when set to 1 the self-test of the accel on Z axis is enabled
   [4]: AFS_SEL, when set to 00 full scale range is +/- 2g, 01 +/- 4g, 10- +/- 8g
   [3]: AFS_SEL, 11 +/- 16g
   [2]: - not used
   [1]: - not used
   [0]: - not used
*/
#define MPU6050_ACCEL_CONFIG        (0x1C)

/* Bits in MPU6050_ACCEL_XOUT_H
   MSB of X axis acceleration reading */
#define MPU6050_ACCEL_XOUT_H        (0x3B)

/* Accelerometer scale selection based on the MPU6050_ACCEL_SCALE macro
   defined in common/common.h header file */
#if (MPU6050_ACCEL_SCALE == 2)

#define MPU6050_ACCEL_SET_RANGE     (0b00000000)

/* RAW_TO_G macro divides the raw acceleration to 32768 since it is a int16 value
   which has a -32768 up to 32767 range */
#define RAW_TO_G(raw_acceleration)  ((raw_acceleration / 32768.0) * 2)

#elif (MPU6050_ACCEL_SCALE == 4)

#define MPU6050_ACCEL_SET_RANGE     (0b00001000)
#define RAW_TO_G(raw_acceleration)  ((raw_acceleration / 32768.0) * 4)

#elif (MPU6050_ACCEL_SCALE == 8)

#define MPU6050_ACCEL_SET_RANGE     (0b00010000)
#define RAW_TO_G(raw_acceleration)  ((raw_acceleration / 32768.0) * 8)

#elif (MPU6050_ACCEL_SCALE == 16)

#define MPU6050_ACCEL_SET_RANGE     (0b00011000)
#define RAW_TO_G(raw_acceleration)  ((raw_acceleration / 32768.0) * 16)

#endif /* MPU6050_ACCEL_RANGE */

/* MPU6050 trigger device reset register value*/
#define MPU6050_DEVICE_RESET        (0x80)
#define MPU6050_DEVICE_ENABLED      (0x00)

/* MPU6050 trigger gyro / accel / temp reset*/
#define MPU6050_GYRO_RESET          (0b00000100)
#define MPU6050_ACCEL_RESET         (0b00000010)
#define MPU6050_TEMP_RESET          (0b00000001)

/* Macro that returns the size of the stream to be sent */
#define STREAM_SIZE(transmit_stream) (sizeof(transmit_stream)/sizeof((transmit_stream)[0]))

////////////////////////////////////////////////////////////////////////////
////                        LOCAL CONSTANTS                             ////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
////                       FUNCTION DEFINITIONS                         ////
////////////////////////////////////////////////////////////////////////////

esp_err_t accelerometer_setup(void)
{
    esp_err_t esp_response             = ESP_FAIL;
    i2c_config_t configuration_options = {0};
```

```
122
123     /* Setup ESP as master on the i2c bus */
124     configuration_options.mode = I2C_MODE_MASTER;
125
126     /* GPIO_NUM_21 is the only SDA pin on ESP32_WROOM_32 */
127     configuration_options.sda_io_num    = GPIO_NUM_21;
128     configuration_options.sda_pullup_en = GPIO_PULLUP_ENABLE;
129
130     /* GPIO_NUM_22 is the only SCL pin on ESP32_WROOM_32 */
131     configuration_options.scl_io_num    = GPIO_NUM_22;
132     configuration_options.scl_pullup_en = GPIO_PULLUP_ENABLE;
133
134     /* Setup the bus speed to 400 kHz, I2C fast mode */
135     configuration_options.master.clk_speed = I2C_FREQ_400khz;
136
137     /* No clk flags are used */
138     configuration_options.clk_flags = 0;
139
140     /* Configure the bus and install the driver */
141     ESP_ERROR_CHECK(i2c_param_config(i2c_master_port, &configuration_options));
142
143     ESP_ERROR_CHECK(i2c_driver_install(i2c_master_port, configuration_options.mode, 0, 0, 0));
144
145     ESP_ERROR_CHECK(accelerometer_reset());
146
147     esp_response = ESP_OK;
148     return esp_response;
149 }
150
151 static esp_err_t accelerometer_reset(void)
152 {
153     esp_err_t esp_response     = ESP_FAIL;
154     uint8_t transmit_stream[2] = {MPU6050_PWR_MGMT_1, MPU6050_DEVICE_RESET};
155
156     /**************************************************************************
157      *                          MPU6050 reset sequence                        *
158      *  1. Reset the mpu6050 peripheral                                       *
159      *  2. Reset the signal path of acc, gyro and temp sensor in mpu6050      *
160      *  3. Reset the mpu6050 peripheral again                                 *
161     **************************************************************************/
162
163     /* Trigger a reset by setting PWR_MGMT_1 bit7 DEVICE_RESET bit  */
164     ESP_ERROR_CHECK(i2c_master_write_to_device(i2c_master_port, MPU6050_I2C_ADDRESS, &
        transmit_stream[0], STREAM_SIZE(transmit_stream), pdMS_TO_TICKS(100)));
165
166     /* Wait 100ms in order for the issued reset to complete */
167     vTaskDelay(100 / portTICK_PERIOD_MS);
168
169     transmit_stream[0] = MPU6050_SIGNAL_PATH_RESET;
170     transmit_stream[1] = MPU6050_ACCEL_RESET | MPU6050_GYRO_RESET | MPU6050_TEMP_RESET;
171
172     /* Trigger a reset of accelerometer, gyro and temp sensor on MPU6050 */
173     ESP_ERROR_CHECK(i2c_master_write_to_device(i2c_master_port, MPU6050_I2C_ADDRESS, &
        transmit_stream[0], STREAM_SIZE(transmit_stream), pdMS_TO_TICKS(100)));
174
175     /* Wait 100ms in order for the issued reset to complete */
176     vTaskDelay(100 / portTICK_PERIOD_MS);
177
178     transmit_stream[0] = MPU6050_PWR_MGMT_1;
179     transmit_stream[1] = MPU6050_DEVICE_ENABLED;
180
181     /* Finally, reset the PWR_MGMT_1 bit7 DEVICE_RESET bit 7 (it should automatically
182        clear to 0, but this write forces it to 0) */
183     ESP_ERROR_CHECK(i2c_master_write_to_device(i2c_master_port, MPU6050_I2C_ADDRESS, &
        transmit_stream[0], STREAM_SIZE(transmit_stream), pdMS_TO_TICKS(100)));
184
185     /* Wait 100ms in order for the issued reset to complete */
186     vTaskDelay(100 / portTICK_PERIOD_MS);
187
188     /**************************************************************************
189      *                        MPU6050 configuration sequence                  *
190      * 1. Set the accelerometer range in the ACCEL_CONFIG register of mpu6050  *
191     **************************************************************************/
192
193     transmit_stream[0] = MPU6050_ACCEL_CONFIG;
194
```

```
195     /* The MPU6050_ACCEL_SET_RANGE macro is defined based on the settings from
196        common/common.h header file */
197     transmit_stream[1] = MPU6050_ACCEL_SET_RANGE;
198 
199     ESP_ERROR_CHECK(i2c_master_write_to_device(i2c_master_port, MPU6050_I2C_ADDRESS, &
        transmit_stream[0], STREAM_SIZE(transmit_stream), pdMS_TO_TICKS(100)));
200 
201     /* Wait 100ms in order for the configuratio to complete  */
202     vTaskDelay(100 / portTICK_PERIOD_MS);
203 
204     esp_response = ESP_OK;
205     return esp_response;
206 }
207 
208 void accelerometer_read(void *pvParameters)
209 {
210     status_firmware_t *general_status         = (status_firmware_t*)pvParameters;
211     display_data_t *display_data              = general_status->display_data;
212     SemaphoreHandle_t xSemaphore_display_data = general_status->xSemaphore_display_data;
213 
214     /* MPU6050 has 8 bit registers, but the acceleration resolution is on
215         16bit for each axis */
216     uint8_t read_stream[MPU6050_AXIS_NUM * 2] = {0};
217     uint8_t transmit_stream                   = MPU6050_ACCEL_XOUT_H;
218     while(true)
219     {
220         /* Start reading acceleration registers from register 0x3B for 6 bytes */
221         ESP_ERROR_CHECK(i2c_master_write_read_device( i2c_master_port,
222                                                       MPU6050_I2C_ADDRESS,
223                                                       &transmit_stream,
224                                                       1,
225                                                       &read_stream[0],
226                                                       STREAM_SIZE(read_stream),
227                                                       MPU6050_POLLING_RATE));
228 
229         /* Catch MPU6050 error or entering sleep mode
230         Typically it manifests in reading 0 on all accelerometer axis */
231         if ((read_stream[0] == 0U) && (read_stream[1] == 0U)
232             && (read_stream[2] == 0U) && (read_stream[3] == 0U))
233         {
234             accelerometer_reset();
235             vTaskDelay(MPU6050_POLLING_RATE);
236         }
237 
238         /* Check if the data structure is used by other tasks */
239         if ( xSemaphoreTake(xSemaphore_display_data, portMAX_DELAY) == pdTRUE)
240         {
241             /* Combine the High and Low bytes from read_stream into a single variable  */
242             for (uint_fast8_t i = 0; i < MPU6050_AXIS_NUM; i++)
243             {
244                 *(display_data->accelerometer + i)   = (read_stream[i * 2] << 8 | read_stream
    [(i * 2) + 1]);
245                 *(display_data->accelerometer_g + i) = (float)RAW_TO_G((*(display_data->
    accelerometer + i)));
246             }
247             xSemaphoreGive(xSemaphore_display_data);
248         }
249 
250         #ifdef ENABLE_DEBUG_ACCELEROMETER
251         /* Transform the value in G's into a m/s^2 one */
252         printf("X: %f m/s^2, Y: %f  m/s^2, Z: %f  m/s^2\n", (RAW_TO_G(*(display_data->
    accelerometer)) * 9.81), (RAW_TO_G(*(display_data->accelerometer+1)) * 9.81), (RAW_TO_G(*(
    display_data->accelerometer+2)) * 9.81));
253         #endif /* ENABLE_DEBUG_ACCELEROMETER */
254 
255         vTaskDelay(MPU6050_POLLING_RATE);
256     }
257 }
```

components/spi/sdcard/inc/sdcard_task.h

```
1 /*****************************************************************************/
2 /* UPBDRIVE Dashboard Firmware                                               */
3 /* All rights reserverd 2023                                                 */
4 /*****************************************************************************/
5 
6 #ifndef SDCARD_TASK_H
```

```
7  #define SDCARD_TASK_H
8
9  #include <common.h>
10 #include <string.h>
11 #include <esp_vfs_fat.h>
12 #include <sdmmc_cmd.h>
13
14 #include <sys/unistd.h>
15 #include <sys/stat.h>
16
17 ////////////////////////////////////////////////////////////////////////////
18 ////                        GLOBAL VARIABLES                            ////
19 ////////////////////////////////////////////////////////////////////////////
20
21 ////////////////////////////////////////////////////////////////////////////
22 ////                        GLOBAL MACROS                               ////
23 ////////////////////////////////////////////////////////////////////////////
24
25 /* Max char size */
26 #define MAX_CHAR_SIZE   (1024)
27
28 /* Mount point of the sdcard */
29 #define MOUNT_POINT "/sdcard"
30
31 ////////////////////////////////////////////////////////////////////////////
32 ////                        GLOBAL CONSTANTS                            ////
33 ////////////////////////////////////////////////////////////////////////////
34
35 ////////////////////////////////////////////////////////////////////////////
36 ////                        FUNCTION PROTOYPES                          ////
37 ////////////////////////////////////////////////////////////////////////////
38
39 /*
40     * @brief This function will setup the sdcard on SPI bus
41     *
42     * @param void
43     *
44     * @return esp_err_t
45 */
46 esp_err_t sdcard_setup(void);
47
48 /*
49     * @brief This task will write a file to the sdcard and log the display data at 1 Hz
50     *
51     * @param[in] pvParameters
52     *
53     * @return void
54 */
55 void sdcard_write(void *pvParameters);
56
57 #endif /* SDCARD_TASK_H */
```

components/spi/sdcard/sdcard_task.c

```
1  /*****************************************************************************/
2  /* UPBDRIVE Dashboard Firmware                                               */
3  /* All rights reserverd 2023                                                 */
4  /*****************************************************************************/
5
6  #include <sdcard_task.h>
7
8  ////////////////////////////////////////////////////////////////////////////
9  ////                        LOCAL VARIABLES                             ////
10 ////////////////////////////////////////////////////////////////////////////
11
12 /*
13     Local structure used to store temporarly the data from the global
14     structure display_data
15
16     @note:  this is done in order to have the semaphore taken for
17             as little as possible in order to not block other tasks
18 */
19 static display_data_t log_data = {0};
20
21 extern status_firmware_t general_status;
22
23 ////////////////////////////////////////////////////////////////////////////
```

```
////                        LOCAL MACROS                                    ////
//////////////////////////////////////////////////////////////////////////////////

#define COOLANT_LOG_POINT       (0)
#define OIL_PRESSURE_LOG_POINT  (1)
#define BATTERY_LOG_POINT       (2)
#define RPM_LOG_POINT           (3)
#define TPS_LOG_POINT           (4)
#define ACCEL_X_LOG_POINT       (5)
#define ACCEL_Y_LOG_POINT       (6)
#define ACCEL_Z_LOG_POINT       (7)

//////////////////////////////////////////////////////////////////////////////////
////                        LOCAL CONSTANTS                                 ////
//////////////////////////////////////////////////////////////////////////////////

static const char *TAG = "example";
static const char *dashboard_log = MOUNT_POINT"/log.txt";
static const char *log_name[]  = {  "COOLANT",
                                    "OIL PRESSURE",
                                    "BATTERY",
                                    "RPM",
                                    "TPS",
                                    "ACCEL_X",
                                    "ACCEL_Y",
                                    "ACCEL_Z"
                                 };

//////////////////////////////////////////////////////////////////////////////////
////                        LOCAL FUNCTIONS                                 ////
//////////////////////////////////////////////////////////////////////////////////

static esp_err_t sdcard_write_log_point(FILE* log_file, uint8_t log_point, uint16_t log_value)
{
    esp_err_t esp_response    = ESP_FAIL;
    int16_t printed           = 0UL;

    printed = fprintf(log_file, "%d:%d:%d %s: %d\n",
                        general_status.time_hour,
                        general_status.time_minute,
                        general_status.time_second,
                        log_name[log_point],
                        log_value
                    );
    if (printed < 0)
    {
        ESP_LOGE(TAG, "Failed to write to file");

        /* Also signal an error on the webpage */
        general_status.sdcard_logging = false;
        return esp_response;
    }

    esp_response = ESP_OK;
    return esp_response;
}

static esp_err_t sdcard_write_file(const char *path, display_data_t* display_data)
{
    esp_err_t esp_response = ESP_FAIL;

    FILE *log_file = fopen(path, "a+");
    if (log_file == NULL)
    {
        ESP_LOGE(TAG, "Failed to open file for writing");

        general_status.sdcard_logging = false;
        esp_response = ESP_FAIL;
        return esp_response;
    }

    ESP_ERROR_CHECK(sdcard_write_log_point(log_file, COOLANT_LOG_POINT, display_data->
    coolant_temperature));
    ESP_ERROR_CHECK(sdcard_write_log_point(log_file, OIL_PRESSURE_LOG_POINT, display_data->
    oil_pressure));
```

```
97      ESP_ERROR_CHECK(sdcard_write_log_point(log_file, BATTERY_LOG_POINT, display_data->
        battery_voltage));
98      ESP_ERROR_CHECK(sdcard_write_log_point(log_file, RPM_LOG_POINT, display_data->rpm));
99      ESP_ERROR_CHECK(sdcard_write_log_point(log_file, TPS_LOG_POINT, display_data->tps));
100     ESP_ERROR_CHECK(sdcard_write_log_point(log_file, ACCEL_X_LOG_POINT, display_data->
        accelerometer[0]));
101     ESP_ERROR_CHECK(sdcard_write_log_point(log_file, ACCEL_Y_LOG_POINT, display_data->
        accelerometer[1]));
102     ESP_ERROR_CHECK(sdcard_write_log_point(log_file, ACCEL_Z_LOG_POINT, display_data->
        accelerometer[2]));
103
104     fclose(log_file);
105     ESP_LOGI(TAG, "File written");
106
107     general_status.sdcard_logging = true;
108     return ESP_OK;
109 }
110
111 esp_err_t sdcard_setup(void)
112 {
113     esp_err_t esp_response                          = ESP_FAIL;
114     esp_vfs_fat_sdmmc_mount_config_t mount_config = {0};
115     sdmmc_card_t *card                              = NULL;
116     sdmmc_host_t host                               = SDSPI_HOST_DEFAULT();
117     const char mount_point[]                        = MOUNT_POINT;
118     spi_bus_config_t bus_cfg                        = {0};
119     sdspi_device_config_t slot_config               = SDSPI_DEVICE_CONFIG_DEFAULT();
120
121     mount_config.format_if_mount_failed             = false;
122     mount_config.max_files                          = 5;
123     mount_config.allocation_unit_size               = 16 * 1024;
124
125     bus_cfg.intr_flags                              = 0;
126     bus_cfg.isr_cpu_id                              = INTR_CPU_ID_AUTO;
127     bus_cfg.flags                                   = SPICOMMON_BUSFLAG_MASTER |
        SPICOMMON_BUSFLAG_GPIO_PINS;
128     bus_cfg.mosi_io_num                             = GPIO_SDCARD_MOSI;
129     bus_cfg.miso_io_num                             = GPIO_SDCARD_MISO;
130     bus_cfg.sclk_io_num                             = GPIO_SDCARD_CLK;
131     bus_cfg.quadwp_io_num                           = -1;
132     bus_cfg.quadhd_io_num                           = -1;
133     bus_cfg.max_transfer_sz                         = 4000;
134
135     ESP_ERROR_CHECK(spi_bus_initialize((spi_host_device_t)host.slot, &bus_cfg,
        SDSPI_DEFAULT_DMA));
136
137     slot_config.gpio_cs = GPIO_SDCARD_CS;
138     slot_config.host_id = (spi_host_device_t)host.slot;
139     ESP_ERROR_CHECK(esp_vfs_fat_sdspi_mount(mount_point, &host, &slot_config, &mount_config, &
        card));
140
141     esp_response = ESP_OK;
142     return esp_response;
143 }
144
145 void sdcard_write(void *pvParameters)
146 {
147     status_firmware_t   *general_status         = (status_firmware_t*)pvParameters;
148     display_data_t      *display_data           = general_status->display_data;
149     SemaphoreHandle_t   xSemaphore_display_data = general_status->xSemaphore_display_data;
150
151     while(true)
152     {
153         /* Check if the RTC data was received on the CAN bus */
154         while( general_status->time_hour == 99U && general_status->time_minute == 99U &&
        general_status->time_second == 99U)
155         {
156             /* Stay in this loop */
157         }
158
159         if (xSemaphoreTake(xSemaphore_display_data, portMAX_DELAY) == pdTRUE)
160         {
161             /* Make a local copy of the global data structure display_data */
162             memcpy(&log_data, display_data, sizeof(display_data_t));
163             xSemaphoreGive(xSemaphore_display_data);
164         }
```

```
165
166         ESP_ERROR_CHECK(sdcard_write_file(dashboard_log, &log_data));
167
168         vTaskDelay(SDCARD_LOGGING_RATE);
169     }
170 }
```

components/uart/lcd-4dsystems/inc/lcd_task.h

```
1  /*****************************************************************************/
2  /* UPBDRIVE Dashboard Firmware                                               */
3  /* All rights reserverd 2023                                                 */
4  /*****************************************************************************/
5  #ifndef LCD_TASK_H
6  #define LCD_TASK_H
7
8  #include <common.h>
9  #include <driver/uart.h>
10
11 ////////////////////////////////////////////////////////////////////////////////
12 ////                        GLOBAL VARIABLES                                ////
13 ////////////////////////////////////////////////////////////////////////////////
14
15 ////////////////////////////////////////////////////////////////////////////////
16 ////                        GLOBAL MACROS                                   ////
17 ////////////////////////////////////////////////////////////////////////////////
18
19 #define LCD_START_UP_PAGE           (0x00)
20 #define LCD_SECONDARY_PAGE          (0x01)
21 #define LCD_THIRD_PAGE              (0x02)
22 #define LCD_MAIN_PAGE               (0x03)
23 #define LCD_FOURTH_PAGE             (0x04)
24 #define LCD_LOW_12V_BATTERY_PAGE    (0x05)
25 #define LCD_LOW_OIL_PRESSURE_PAGE   (0x06)
26 #define LCD_OVERHEAT_PAGE           (0x07)
27
28 ////////////////////////////////////////////////////////////////////////////////
29 ////                        GLOBAL CONSTANTS                                ////
30 ////////////////////////////////////////////////////////////////////////////////
31
32 ////////////////////////////////////////////////////////////////////////////////
33 ////                        FUNCTION PROTOYPES                              ////
34 ////////////////////////////////////////////////////////////////////////////////
35
36 /*
37     * @brief This function will setup the lcd display from 4DSystems
38     *
39     * @param[in] void
40     *
41     * @return esp_err_t
42 */
43 esp_err_t lcd_setup(void);
44
45 /*
46     * @brief This task will update the display objects @ LCD_REFRESH_RATE frequency
47     *
48     * @param[in] pvParameters
49     *
50     * @return void
51 */
52 void lcd_update(void *pvParameters);
53
54 #endif /* LCD_TASK_H */
```

components/uart/lcd-4dsystems/lcd_task.c

```
1  /*****************************************************************************/
2  /* UPBDRIVE Dashboard Firmware                                               */
3  /* All rights reserverd 2023                                                 */
4  /*****************************************************************************/
5
6  #include <lcd_task.h>
7
8  ////////////////////////////////////////////////////////////////////////////////
9  ////                        GLOBAL VARIABLES                                ////
10 ////////////////////////////////////////////////////////////////////////////////
11
```

```
12 /* Timer handle for changing page at 250ms rate */
13 static TimerHandle_t xTimer_change_page;
14
15 /* Uart queue handle used in lcd_setup function */
16 static QueueHandle_t uart_queue;
17
18 /* Variable to avoid change page faster than 250ms */
19 uint8_t change_page_ready = true;
20
21 /*
22     @brief: This struct holds the sanitized data to be displayed on the LCD
23             display, it is a local struct in which it is copied the data from
24             the global display_data struct
25 */
26 static display_data_t sanitized_display_data = {0};
27
28 static uint8_t lcd_transmission_error = 0;
29 static uint8_t error_page_displayed   = false;
30
31 ////////////////////////////////////////////////////////////////////////////
32 ////                        GLOBAL FUNCTIONS                            ////
33 ////////////////////////////////////////////////////////////////////////////
34
35 ////////////////////////////////////////////////////////////////////////////
36 ////                        LOCAL MACROS                                ////
37 ////////////////////////////////////////////////////////////////////////////
38
39 /* Value returned by the LCD if the transmission of data was successful */
40 #define LCD_TRANSMISSION_OK     (6)
41
42 /* Errors defined in the firmware to change the page accordingly to them */
43 #define LCD_DISPLAY_NO_ERROR            (0)
44 #define LCD_DISPLAY_OVERHEAT            (10)
45 #define LCD_DISPLAY_LOW_OIL_PRESSURE    (20)
46 #define LCD_DISPLAY_LOW_12V_BATTERY     (30)
47
48 #define GENIE_READ_OBJ          0
49 #define GENIE_WRITE_OBJ         1
50 #define GENIE_WRITE_STR         2
51 #define GENIE_WRITE_STRU        3
52 #define GENIE_WRITE_CONTRAST    4
53 #define GENIE_REPORT_OBJ        5
54 #define GENIE_REPORT_EVENT      7
55 #define GENIEM_WRITE_BYTES      8
56 #define GENIEM_WRITE_DBYTES     9
57 #define GENIEM_REPORT_BYTES     10
58 #define GENIEM_REPORT_DBYTES    11
59 #define GENIE_WRITE_INH_LABEL   12
60
61 #define GENIE_OBJ_DIPSW                 0
62 #define GENIE_OBJ_KNOB                  1
63 #define GENIE_OBJ_ROCKERSW              2
64 #define GENIE_OBJ_ROTARYSW              3
65 #define GENIE_OBJ_SLIDER                4
66 #define GENIE_OBJ_TRACKBAR              5
67 #define GENIE_OBJ_WINBUTTON             6
68 #define GENIE_OBJ_ANGULAR_METER         7
69 #define GENIE_OBJ_COOL_GAUGE            8
70 #define GENIE_OBJ_CUSTOM_DIGITS         9
71 #define GENIE_OBJ_FORM                  10
72 #define GENIE_OBJ_GAUGE                 11
73 #define GENIE_OBJ_IMAGE                 12
74 #define GENIE_OBJ_KEYBOARD              13
75 #define GENIE_OBJ_LED                   14
76 #define GENIE_OBJ_LED_DIGITS            15
77 #define GENIE_OBJ_METER                 16
78 #define GENIE_OBJ_STRINGS               17
79 #define GENIE_OBJ_THERMOMETER           18
80 #define GENIE_OBJ_USER_LED              19
81 #define GENIE_OBJ_VIDEO                 20
82 #define GENIE_OBJ_STATIC_TEXT           21
83 #define GENIE_OBJ_SOUND                 22
84 #define GENIE_OBJ_TIMER                 23
85 #define GENIE_OBJ_SPECTRUM              24
86 #define GENIE_OBJ_SCOPE                 25
87 #define GENIE_OBJ_TANK                  26
```

```
88 #define GENIE_OBJ_USERIMAGES            27
89 #define GENIE_OBJ_PINOUTPUT             28
90 #define GENIE_OBJ_PININPUT              29
91 #define GENIE_OBJ_4DBUTTON              30
92 #define GENIE_OBJ_ANIBUTTON             31
93 #define GENIE_OBJ_COLORPICKER           32
94 #define GENIE_OBJ_USERBUTTON            33
95 // reserved for magic functions         34
96 #define GENIE_OBJ_SMARTGAUGE            35
97 #define GENIE_OBJ_SMARTSLIDER           36
98 #define GENIE_OBJ_SMARTKNOB             37
99 // Not advisable to use the below 3, use the above 3 instead.
100 #define GENIE_OBJ_ISMARTGAUGE           35 // Retained for backwards compatibility, however
        Users should use SMARTGAUGE instead of ISMARTGAUGE
101 #define GENIE_OBJ_ISMARTSLIDER          36 // Retained for backwards compatibility, however
        Users should use SMARTSLIDER instead of ISMARTSLIDER
102 #define GENIE_OBJ_ISMARTKNOB            37 // Retained for backwards compatibility, however
        Users should use SMARTKNOB instead of ISMARTKNOB
103 // Comment end
104 #define GENIE_OBJ_ILED_DIGITS_H         38
105 #define GENIE_OBJ_IANGULAR_METER        39
106 #define GENIE_OBJ_IGAUGE                40
107 #define GENIE_OBJ_ILABELB               41
108 #define GENIE_OBJ_IUSER_GAUGE           42
109 #define GENIE_OBJ_IMEDIA_GAUGE          43
110 #define GENIE_OBJ_IMEDIA_THERMOMETER    44
111 #define GENIE_OBJ_ILED                  45
112 #define GENIE_OBJ_IMEDIA_LED            46
113 #define GENIE_OBJ_ILED_DIGITS_L         47
114 #define GENIE_OBJ_ILED_DIGITS           47
115 #define GENIE_OBJ_INEEDLE               48
116 #define GENIE_OBJ_IRULER                49
117 #define GENIE_OBJ_ILED_DIGIT            50
118 #define GENIE_OBJ_IBUTTOND              51
119 #define GENIE_OBJ_IBUTTONE              52
120 #define GENIE_OBJ_IMEDIA_BUTTON         53
121 #define GENIE_OBJ_ITOGGLE_INPUT         54
122 #define GENIE_OBJ_IDIAL                 55
123 #define GENIE_OBJ_IMEDIA_ROTARY         56
124 #define GENIE_OBJ_IROTARY_INPUT         57
125 #define GENIE_OBJ_ISWITCH               58
126 #define GENIE_OBJ_ISWITCHB              59
127 #define GENIE_OBJ_ISLIDERE              60
128 #define GENIE_OBJ_IMEDIA_SLIDER         61
129 #define GENIE_OBJ_ISLIDERH              62
130 #define GENIE_OBJ_ISLIDERG              63
131 #define GENIE_OBJ_ISLIDERF              64
132 #define GENIE_OBJ_ISLIDERD              65
133 #define GENIE_OBJ_ISLIDERC              66
134 #define GENIE_OBJ_ILINEAR_INPUT         67
135
136 #define lowByte(w) ((uint8_t) ((w) & 0xff))
137 #define highByte(w) ((uint8_t) ((w) >> 8))
138
139 #define LCD_CHECK_IF_COOLANT_TEMP_LOWER_85(coolant_temperature)     (coolant_temperature <=
        85)
140 #define LCD_CHECK_IF_COOLANT_TEMP_BTWEEN_85_95(coolant_temperature) ((coolant_temperature >
        85) && (coolant_temperature <= 95))
141 #define LCD_CHECK_IF_COOLANT_TEMP_HIGHER_95(coolant_temperature)    (coolant_temperature > 95)
142
143 #define LCD_CHECK_IF_OIL_TEMP_LOWER_100(oil_temperature)            (oil_temperature <= 100)
144 #define LCD_CHECK_IF_OIL_TEMP_BWTEEN_100_130(oil_temperature)       ((oil_temperature > 100)
        && (oil_temperature <= 130))
145 #define LCD_CHECK_IF_OIL_TEMP_HIGHER_130(oil_temperature)           (oil_temperature > 130)
146
147 /* Error flag register bits mask*/
148 #define LCD_LOG_COOLANT_ERROR               (0b00000001)
149 #define LCD_LOG_OIL_TEMP_ERROR              (0b00000010)
150 #define LCD_LOG_OIL_PRESSURE_ERROR          (0b00000100)
151 #define LCD_LOG_HYBRID_STATUS_ERROR         (0b00001000)
152 #define LCD_LOG_SAFETY_STATUS_ERROR         (0b00010000)
153 #define LCD_LOG_CAN_STATUS_ERROR            (0b00100000)
154 #define LCD_LOG_TPS_ERROR                   (0b01000000)
155
156 ////////////////////////////////////////////////////////////////////////////
157 ////                        LOCAL CONSTANTS                             ////
```

```
158 ////////////////////////////////////////////////////////////////////////////////
159
160 ////////////////////////////////////////////////////////////////////////////////
161 ////                          LOCAL FUNCTIONS                               ////
162 ////////////////////////////////////////////////////////////////////////////////
163 /*
164     @brief: This function is used to reset the LCD display in case of
165             transmission error
166
167     @note: The reset pin is hardcoded to GPIO_LCD_RST
168 */
169 static void lcd_reset();
170
171 /*
172     @brief: This function is used to change the page of the LCD display
173             based on the signal_change_page flag and on the current_page variable
174
175     @note: The signal_change_page variable is set in the button ISR and current_page
176            is a local variable of the lcd_update task
177 */
178 static esp_err_t change_page_handler(uint8_t* signal_change_page, uint8_t* current_page,
    uint8_t* error_type);
179
180 /*  @brief: This function sanitizes the data in the display_data struct before
181            sending it to the LCD display
182
183     @note: If a value higher then expected is sent to the LCD display, it will
184            crash and it will need a power cycle to recover, this function mitigates
185            that
186
187     @param[IN]: display_data_t* display_data pointer to the global struct that holds
188                 the data from the CAN bus and the ACCELEROMETER
189
190     @param[IN]: SemaphoreHandle_t xSemaphore_display_data semaphore to protect the
191                 display_data struct from being accessed by multiple tasks at the same time
192
193     @param[OUT]: esp_err_t esp_response
194 */
195 static esp_err_t lcd_data_check_sanity(display_data_t* display_data, display_data_t*
    sanitized_display_data, SemaphoreHandle_t xSemaphore_display_data, uint8_t* error_type);
196
197 /*
198     @brief: This function is used to write data to the LCD 4DSystems display
199     via the UART port,
200
201     @note: The uart port is hardcoded to UART_NUM_2 because the code is
202     written for ESP32 WROOM 32 mcu. Change it accourdingly if you are not
203     using the same mcu.
204 */
205 static void lcd_write_object (uint8_t object, uint8_t index, uint16_t data);
206
207 /*
208     @brief: This is a callback function to set the variable change_page_ready as
209             true after 250ms
210
211     @note: When you press a button, it will have a bouncing effect, this timer
212     mitigates that at button press, the debounce at button release is mitigated in
213     the buttton ISR
214 */
215 void change_page_timer_callback();
216
217
218 void lcd_update(void *pvParameters)
219 {
220     status_firmware_t *general_status         = (status_firmware_t*)pvParameters;
221     display_data_t    *display_data           = general_status->display_data;
222     SemaphoreHandle_t xSemaphore_display_data = general_status->xSemaphore_display_data;
223     uint8_t current_page                      = LCD_MAIN_PAGE;
224     uint8_t error_type                        = LCD_DISPLAY_NO_ERROR;
225
226     /*************************************************************************
227      *                      ENTRANCE PAGE LOGIC                              *
228     *************************************************************************/
229     /* Wait for the screen to start */
230     vTaskDelay(LCD_START_UP_SCREEN_DELAY);
231
```

```
232     /* Switch back to the LCD_MAIN_PAGE */
233     lcd_write_object(GENIE_OBJ_FORM, 0x03, 0);
234 
235     #ifdef ENABLE_DEBUG_DISPLAY
236     /* initialize the object for display control */
237     uint8_t value = 0;
238     #endif /* ENABLE_DEBUG_DISPLAY */
239 
240     while(true)
241     {
242         #ifndef ENABLE_DEBUG_DISPLAY
243         ESP_ERROR_CHECK(lcd_data_check_sanity(display_data, &sanitized_display_data,
    xSemaphore_display_data, &error_type));
244 
245         ESP_ERROR_CHECK(change_page_handler(&general_status->signal_change_page, &current_page
    , &error_type));
246 
247         switch(current_page)
248         {
249 
250             /*************************************************************
251              *                      MAIN PAGE UPDATE LOGIC                *
252             *************************************************************/
253             case LCD_MAIN_PAGE:
254             {
255                 /* Set the RPM digits */
256                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 0, sanitized_display_data.rpm);
257 
258                 /* Set the RPM bar */
259                 lcd_write_object(GENIE_OBJ_IGAUGE,     0, sanitized_display_data.rpm/100);
260 
261                 /* Set the GEAR digit */
262                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 1, sanitized_display_data.current_gear)
    ;
263 
264                 /* Set the COOLANT bar */
265                 lcd_write_object(GENIE_OBJ_IGAUGE,     1, sanitized_display_data.
    coolant_temperature);
266 
267                 /* Set the FAN led status */
268                 lcd_write_object(GENIE_OBJ_IMEDIA_LED, 0, sanitized_display_data.fan_state);
269 
270                 /* Set the OIL temperature bar */
271                 lcd_write_object(GENIE_OBJ_IGAUGE,     2, sanitized_display_data.
    oil_temperature);
272 
273                 break;
274             }
275 
276             /*************************************************************
277              *                    SECONDARY PAGE UPDATE LOGIC             *
278             *************************************************************/
279             case LCD_SECONDARY_PAGE:
280             {
281                 /* Set the CAN led status */
282                 lcd_write_object(GENIE_OBJ_IMEDIA_LED, 1, sanitized_display_data.can_status);
283 
284                 /* Set the HYBRID system led status */
285                 lcd_write_object(GENIE_OBJ_IMEDIA_LED, 2, sanitized_display_data.hybrid_status
    );
286 
287                 /* Set the SAFETY circuit led status */
288                 lcd_write_object(GENIE_OBJ_IMEDIA_LED, 3, sanitized_display_data.
    safety_circuit_status);
289 
290                 /* Set the OIL PRESSURE digits */
291                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 5, sanitized_display_data.oil_pressure)
    ;
292 
293                 /* Set the BRAKE PRESSURE digits */
294                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 6, sanitized_display_data.
    brake_pressure_raw);
295 
296                 /* Set the TPS % digits */
297                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 7, sanitized_display_data.tps);
298 
```

```
299                 /* Set the OIL TEMPERATURE digits */
300                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 8, sanitized_display_data.
    oil_temperature_2);
301
302                 /* Set the MAP digits */
303                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 9, sanitized_display_data.
    hybrid_selector_value);
304
305                 /* Set the LV BATT VOLTAGE digits */
306                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 10, sanitized_display_data.
    battery_voltage);
307                 break;
308             }
309
310             /*********************************************************
311              *                     THIRD PAGE UPDATE LOGIC           *
312             *********************************************************/
313             case LCD_THIRD_PAGE:
314             {
315                 /* Set the LAMBDA digits */
316                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 2, sanitized_display_data.lambda);
317
318                 /* Set the MASS AIRFLOW PREESURE digits */
319                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 3, sanitized_display_data.map);
320
321                 /* Set the FUEL PRESSURE digits */
322                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 4, sanitized_display_data.fuel_pressure
    );
323
324                 /* Set the EGT #1 digits */
325                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 11, sanitized_display_data.egt[0]);
326
327                 /* Set the EGT #2 digits */
328                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 12, sanitized_display_data.egt[1]);
329
330                 /* Set the EGT #3 digits */
331                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 13, sanitized_display_data.egt[2]);
332
333                 /* Set the EGT #4 digits */
334                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 14, sanitized_display_data.egt[3]);
335
336                 /* Set the IN CURRENT ALTERN digits */
337                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 16, sanitized_display_data.
    input_current_altr_pmu);
338
339                 /* Set the IN VOLTAGE PMU digits */
340                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 15, sanitized_display_data.
    input_voltage_pmu);
341                 break;
342             }
343
344             /*********************************************************
345              *                     FOURTH PAGE UPDATE LOGIC          *
346             *********************************************************/
347             case LCD_FOURTH_PAGE:
348             {
349                 /* Set the TYRE LEFT FRONT digits */
350                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 17, sanitized_display_data.
    tyre_pressure[0]);
351
352                 /* Set the TYRE LEFT BACK digits */
353                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 18, sanitized_display_data.
    tyre_pressure[1]);
354
355                 /* Set the TYRE RIGHT FRONT digits */
356                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 20, sanitized_display_data.
    tyre_pressure[2]);
357
358                 /* Set the TYRE RIGHT BACK digits */
359                 lcd_write_object(GENIE_OBJ_LED_DIGITS, 19, sanitized_display_data.
    tyre_pressure[3]);
360                 break;
361             }
362
363             /* Enter here only if current_page is corrupted, force it to MAIN */
364             default:
```

```
365             {
366                 current_page = LCD_MAIN_PAGE;
367                 break;
368             }
369         }
370 
371         /* This code is used for debug purposes, it will increment the gear
372             as well as the rpm */
373         #else
374         printf("lcd update\n");
375         lcd_write_object(GENIE_OBJ_LED_DIGITS, 0, value);
376         lcd_write_object(GENIE_OBJ_LED_DIGITS, 1, value);
377         lcd_write_object(GENIE_OBJ_IGAUGE,     0, value);
378         value += 1;
379         #endif /* ENABLE_DEBUG_DISPLAY */
380 
381         /* Put the task in blocking state */
382         vTaskDelay(LCD_REFRESH_RATE);
383     }
384 }
385 
386 /*
387     @brief: This function is used to change the page of the LCD display
388             based on the signal_change_page flag and on the current_page variable
389 
390     @note: The signal_change_page variable is set in the button ISR and current_page
391            is a local variable of the lcd_update task
392 */
393 static esp_err_t change_page_handler(uint8_t* signal_change_page, uint8_t* current_page,
    uint8_t* error_type)
394 {
395     esp_err_t esp_response       = ESP_FAIL;
396 
397     /* Check that no error occured, display common pages */
398     if (*error_type == LCD_DISPLAY_NO_ERROR)
399     {
400         /* Check if the change page button was pressed */
401         if ((0xFF == *signal_change_page) && (true == change_page_ready))
402         {
403             /* Set this variable to false to avoid multiple changes */
404             change_page_ready = false;
405             xTimerStart(xTimer_change_page, 0);
406 
407             /* Check what is the current page (MAIN, SECONDARY, THIRD) */
408             switch (*current_page)
409             {
410                 /* If the page is MAIN switch to SECONDARY */
411                 case LCD_MAIN_PAGE:
412                 {
413                     *current_page = LCD_SECONDARY_PAGE;
414                     lcd_write_object(GENIE_OBJ_FORM, LCD_SECONDARY_PAGE, 0);
415                     break;
416                 }
417 
418                 /* If the page is SECONDARY switch to THIRD */
419                 case LCD_SECONDARY_PAGE:
420                 {
421                     *current_page = LCD_THIRD_PAGE;
422                     lcd_write_object(GENIE_OBJ_FORM, LCD_THIRD_PAGE, 0);
423                     break;
424                 }
425 
426                 /* If the page is THIRD switch to FOURTH */
427                 case LCD_THIRD_PAGE:
428                 {
429                     *current_page = LCD_FOURTH_PAGE;
430                     lcd_write_object(GENIE_OBJ_FORM, LCD_FOURTH_PAGE, 0);
431                     break;
432                 }
433 
434                 /* If the page is FOURTH switch to MAIN */
435                 case LCD_FOURTH_PAGE:
436                 {
437                     *current_page = LCD_MAIN_PAGE;
438                     lcd_write_object(GENIE_OBJ_FORM, LCD_MAIN_PAGE, 0);
439                     break;
```

```
                }

                /* In case of unknown current page state, or start-up
                    switch to MAIN */
                default:
                {
                    *current_page = LCD_MAIN_PAGE;
                    lcd_write_object(GENIE_OBJ_FORM, LCD_MAIN_PAGE, 0);
                    break;
                }
            }

            /* Set the signal to 0x00, to mark that the page change was processed */
            *signal_change_page = 0x00;
        }
        else if (error_page_displayed == true)
        {
                *current_page = LCD_MAIN_PAGE;
                lcd_write_object(GENIE_OBJ_FORM, LCD_MAIN_PAGE, 0);

                error_page_displayed = false;
        }
    }
    else
    {
        if (error_page_displayed == false)
        {
            switch(*error_type)
            {
                case LCD_DISPLAY_OVERHEAT:
                {
                    *current_page = LCD_OVERHEAT_PAGE;
                    lcd_write_object(GENIE_OBJ_FORM, LCD_OVERHEAT_PAGE, 0);
                    break;
                }

                case LCD_DISPLAY_LOW_OIL_PRESSURE:
                {
                    *current_page = LCD_LOW_OIL_PRESSURE_PAGE;
                    lcd_write_object(GENIE_OBJ_FORM, LCD_LOW_OIL_PRESSURE_PAGE, 0);
                    break;
                }

                case LCD_DISPLAY_LOW_12V_BATTERY:
                {
                    *current_page = LCD_LOW_12V_BATTERY_PAGE;
                    lcd_write_object(GENIE_OBJ_FORM, LCD_LOW_12V_BATTERY_PAGE, 0);
                    break;
                }

                default:
                {
                    *current_page = LCD_MAIN_PAGE;
                    lcd_write_object(GENIE_OBJ_FORM, LCD_MAIN_PAGE, 0);
                    break;
                }
            }

            error_page_displayed = true;
        }
    }

    esp_response = ESP_OK;
    return esp_response;
}

/*
    @brief: This function is used to write data to the LCD 4DSystems display
    via the UART port,

    @note: The uart port is hardcoded to UART_NUM_2 because the code is
    written for ESP32 WROOM 32 mcu. Change it accourdingly if you are not
    using the same mcu.
*/
static void lcd_write_object (uint8_t object, uint8_t index, uint16_t data)
{
```

```
516     uint8_t msb, lsb;
517     uint8_t checksum, write_signal;
518     write_signal = GENIE_WRITE_OBJ;
519 
520     lsb = lowByte(data);
521     msb = highByte(data);
522 
523     uart_write_bytes(UART_NUM_2, &write_signal, 1);
524     checksum  = GENIE_WRITE_OBJ ;
525     uart_write_bytes(UART_NUM_2, &object, 1);
526     checksum ^= object ;
527     uart_write_bytes(UART_NUM_2, &index, 1);
528     checksum ^= index ;
529     uart_write_bytes(UART_NUM_2, &msb, 1);
530     checksum ^= msb;
531     uart_write_bytes(UART_NUM_2, &lsb, 1);
532     checksum ^= lsb;
533     uart_write_bytes(UART_NUM_2, &checksum, 1);
534 
535     /* Read response from the LCD */
536     uart_read_bytes(UART_NUM_2, &checksum, 1, 1000 / portTICK_PERIOD_MS);
537 
538     /* Check the reponse, if it is not LCD_TRANSMISSION_OK, issue a reset */
539     if (checksum != LCD_TRANSMISSION_OK)
540     {
541         lcd_transmission_error++;
542         if (lcd_transmission_error > 20)
543         {
544             lcd_reset();
545             lcd_transmission_error = 0;
546         }
547     }
548 }
549 
550 /*  @brief: This function sanitizes the data in the display_data struct before
551            sending it to the LCD display
552 
553     @note: If a value higher then expected is sent to the LCD display, it will
554            crash and it will need a power cycle to recover, this function mitigates
555            that
556 
557     @param[IN]: display_data_t* display_data pointer to the global struct that holds
558                 the data from the CAN bus and the ACCELEROMETER
559 
560     @param[IN]: SemaphoreHandle_t xSemaphore_display_data semaphore to protect the
561                 display_data struct from being accessed by multiple tasks at the same time
562 
563     @param[OUT]: esp_err_t esp_response
564 */
565 static esp_err_t lcd_data_check_sanity(display_data_t* display_data, display_data_t*
        sanitized_display_data, SemaphoreHandle_t xSemaphore_display_data, uint8_t* error_type)
566 {
567     esp_err_t esp_response = ESP_FAIL;
568 
569     /* Wait for the semaphore to be available, this is a blocking point in the code */
570     if(xSemaphoreTake( xSemaphore_display_data, portMAX_DELAY ) == pdTRUE )
571     {
572         /* Make a local copy of the display_data struct in the sanitized_display_data */
573         memcpy(sanitized_display_data, display_data, sizeof(display_data_t));
574 
575         xSemaphoreGive( xSemaphore_display_data );
576 
577         *error_type = LCD_DISPLAY_NO_ERROR;
578 
579         if (sanitized_display_data->coolant_temperature > COOLANT_OVERHEAT_THRESHOLD)
580         {
581             *error_type = LCD_DISPLAY_OVERHEAT;
582         }
583 
584         /* Check if the COOLANT is higher than 100, if it is, set it to 100
585 
586             @note: That the coolant temperature is represented with a progress bar
587             on the LCD display, which is not symmetrical therefore the coolant temperature
588             needs to be readjusted before being sent to the display
589 
590             0 - 85 degrees celsius are represented between 0 - 40 on the progress bar
```

```
591             0 being 0 and 85 being 40
592
593             85 - 95 degrees celsius are represented between 40 - 70 on the progress
594             bar
595
596             95 - 100 degrees celsius are represented between 70 - 100 on the progress
597             bar
598         */
599         if (sanitized_display_data->coolant_temperature > 100)
600         {
601             sanitized_display_data->coolant_temperature = 100;
602
603             /* This is used in change_page_handler function to switch
604             to the LCD_PAGE_OVERHEAT form */
605             *error_type = LCD_DISPLAY_OVERHEAT;
606         }
607
608         /* This is the blue area of the progress bar, engine is cold  */
609         else if (LCD_CHECK_IF_COOLANT_TEMP_LOWER_85(sanitized_display_data->
    coolant_temperature))
610         {
611             sanitized_display_data->coolant_temperature = (uint8_t)((sanitized_display_data->
    coolant_temperature / 85.0) * 40.0);
612         }
613         /* This is the green area of the progress bar, considered to be optimal */
614         else if (LCD_CHECK_IF_COOLANT_TEMP_BTWEEN_85_95(sanitized_display_data->
    coolant_temperature))
615         {
616             sanitized_display_data->coolant_temperature = 40 + (uint8_t)((
    sanitized_display_data->coolant_temperature - 85.0) / 10.0 * 30.0);
617         }
618         /* This is the red area of the progress bar, considered dangerous */
619         else if (LCD_CHECK_IF_COOLANT_TEMP_HIGHER_95(sanitized_display_data->
    coolant_temperature))
620         {
621             sanitized_display_data->coolant_temperature = 70 + (uint8_t)((
    sanitized_display_data->coolant_temperature - 95) / 5.0 * 30.0);
622         }
623
624         /* Check if oil temparature is higher than 100, if it is, set it to 100
625
626             @note: That the oil temperature is represented with a progress bar
627             on the LCD display, which is not symmetrical therefore the oil temperature
628             needs to be readjusted before being sent to the display
629
630             0 - 100 degress celsius are represented between 0 - 50 on the progress bar
631
632             100 - 130 degrees celsius are represented between 50 - 80 on the progress
633             bar
634
635             130 - 150 degrees celsius are represented between 80 - 100 on the progress
636         */
637         /* Copy the unscaled version of oil_temperature in oil_temperature_2 */
638         sanitized_display_data->oil_temperature_2 = sanitized_display_data->oil_temperature;
639
640         if (sanitized_display_data->oil_temperature > 150)
641         {
642             sanitized_display_data->oil_temperature = 150;
643         }
644
645         /* This is the blue area on the display, engine is cold */
646         if (LCD_CHECK_IF_OIL_TEMP_LOWER_100(sanitized_display_data->oil_temperature))
647         {
648             sanitized_display_data->oil_temperature = sanitized_display_data->oil_temperature
    / 2;
649         }
650         /* This is the green area on the display, it is considered optimal */
651         else if (LCD_CHECK_IF_OIL_TEMP_BWTEEN_100_130(sanitized_display_data->oil_temperature)
    )
652         {
653             sanitized_display_data->oil_temperature = 50 + (sanitized_display_data->
    oil_temperature - 100);
654         }
655         /* This is the red area on the display, it is considered dangerous */
656         else if (LCD_CHECK_IF_OIL_TEMP_HIGHER_130(sanitized_display_data->oil_temperature))
657         {
```

```
658             sanitized_display_data->oil_temperature = 80 + (sanitized_display_data->
    oil_temperature - 130);
659         }
660 
661         /* Check if the fan_state is either true or false, set it to false to signal error  */
662         if ((true != sanitized_display_data->fan_state) && (false != sanitized_display_data->
    fan_state))
663         {
664             sanitized_display_data->fan_state = false;
665         }
666 
667         /* Check if the can_state is either true or false, set error flag */
668         if ((true != sanitized_display_data->can_status) && (false != sanitized_display_data->
    can_status))
669         {
670             sanitized_display_data->can_status = false;
671         }
672 
673         /* Check if the hybrid_status is either true or false, set error flag */
674         if ((true != sanitized_display_data->hybrid_status) && (false != 
    sanitized_display_data->hybrid_status))
675         {
676             /* Set it to false */
677             sanitized_display_data->hybrid_status = false;
678         }
679 
680         /* Check if the safety_circuit_status is either true or false */
681         if ((true != sanitized_display_data->safety_circuit_status) && (false != 
    sanitized_display_data->safety_circuit_status))
682         {
683             /* Set it to false */
684             sanitized_display_data->safety_circuit_status = false;
685         }
686 
687         if (sanitized_display_data->oil_pressure < LOW_OIL_PRESSURE_THRESHOLD)
688         {
689             *error_type = LCD_DISPLAY_LOW_OIL_PRESSURE;
690         }
691 
692         if (sanitized_display_data->battery_voltage < LOW_12V_BATTERY_THRESHOLD)
693         {
694             *error_type = LCD_DISPLAY_LOW_12V_BATTERY;
695         }
696     }
697 
698     esp_response = ESP_OK;
699     return esp_response;
700 }
701 
702 /*
703     @brief: This function is used to reset the LCD display in case of
704             transmission error
705 
706     @note: The reset pin is hardcoded to GPIO_LCD_RST
707 */
708 static void lcd_reset()
709 {
710     /* If gpio is set to GND, the display is reset */
711     gpio_set_level(GPIO_LCD_RST, 0);
712 
713     /* Set level back to 3V3, release the reset */
714     gpio_set_level(GPIO_LCD_RST, 1);
715 
716     return;
717 }
718 
719 esp_err_t lcd_setup(void)
720 {
721     esp_err_t esp_response = ESP_FAIL;
722 
723     /*********************************************************************
724      *                      UART SETUP - for LCD transmission                    *
725     *********************************************************************/
726     uart_config_t uart_config = {
727                                     /* Set the LCD baudarate according to common.h headfer */
728                                     .baud_rate = LCD_BAUDRATE,
```

```
                                    /* The lcd communicates on 8 bit data without parity */
                                    .data_bits = UART_DATA_8_BITS,
                                    .parity = UART_PARITY_DISABLE,
                                    .stop_bits = UART_STOP_BITS_1,
                                    /* There is no flow control needed to communicate with
                                       the lcd */
                                    .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
                                    .rx_flow_ctrl_thresh = 122,
                                };

    /* Setup the configuration parameters on UART_2, since UART_1 is used
       to flash the ESP32 microcontroller */
    ESP_ERROR_CHECK(uart_param_config(UART_NUM_2, &uart_config));

    /* Setup UART2 buffered IO with event queue */
    const int uart_buffer_size = (1024 * 2);

    /* Install UART2 driver using an event queue here */
    ESP_ERROR_CHECK(uart_driver_install(UART_NUM_2, uart_buffer_size, \
                                        uart_buffer_size, 10, &uart_queue, 0));

    /* Configure the GPIO_LCD_TX and GPIO_LCD_RX as UART2 pins */
    ESP_ERROR_CHECK(uart_set_pin(UART_NUM_2, GPIO_LCD_TX, GPIO_LCD_RX, -1, -1));

    /*************************************************************************
     *       RESET PIN SETUP - for LCD reset in case of transmission error   *
    *************************************************************************/
    gpio_config_t gpio_lcd_rst =
    {
        .pin_bit_mask = (1ULL << GPIO_LCD_RST),
        .mode         = GPIO_MODE_OUTPUT,
        .pull_up_en   = GPIO_PULLUP_ENABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type    = GPIO_INTR_DISABLE
    };

    ESP_ERROR_CHECK(gpio_config(&gpio_lcd_rst));

    /* Issue a reset here, otherwise screen goes white */
    lcd_reset();

    /*************************************************************************
     *        CREATE TIMER (250ms between each change page request)          *
    *************************************************************************/
    /* Create a software timer for change_page_ready variable, wait 250ms before being able to
     change the page again */
    xTimer_change_page = xTimerCreate("change_page", (250/portTICK_PERIOD_MS), pdFALSE, (void
    *)0, change_page_timer_callback);

    esp_response = ESP_OK;
    return esp_response;
}

void change_page_timer_callback()
{
    change_page_ready = true;
}
```

components/wifi-http/inc/wifi_task.h

```
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#ifndef WIFI_TASK_H
#define WIFI_TASK_H

#include <common.h>
#include <file_server.h>

#include <esp_mac.h>
#include <esp_wifi.h>
#include <esp_event.h>
#include <nvs_flash.h>
#include <esp_http_server.h>
#include <esp_random.h>
```

```c
#include <esp_tls_crypto.h>
#include <esp_log.h>
#include <esp_netif.h>
#include <esp_err.h>


////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                            ////
////////////////////////////////////////////////////////////////////////////

typedef struct {
    char    *username;
    char    *password;
} basic_auth_info_t;

////////////////////////////////////////////////////////////////////////////
////                        GLOBAL MACROS                               ////
////////////////////////////////////////////////////////////////////////////

#define ESP_WIFI_CHANNEL 5
#define ESP_MAXIMUM_RETRY 10

/* The event group allows multiple bits for each event, but we only care about two events:
 * - we are connected to the AP with an IP
 * - we failed to connect after the maximum amount of retries */
#define WIFI_CONNECTED_BIT BIT0
#define WIFI_FAIL_BIT      BIT1

#define EXAMPLE_HTTP_QUERY_KEY_MAX_LEN  (64)
#define HTTPD_401      "401 UNAUTHORIZED"           /*!< HTTP Response 401 */

/* Mount point of the sdcard */
#define MOUNT_POINT "/sdcard"

////////////////////////////////////////////////////////////////////////////
////                        GLOBAL CONSTANTS                            ////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
////                        FUNCTION PROTOYPES                          ////
////////////////////////////////////////////////////////////////////////////

/*
    * @brief This function will setup the wifi access point to ESP32
    *
    * @param void
    *
    * @return void
*/
void wifi_setup(void);

/*
    * @brief This function will setup the http server on the ESP32 for
    *        file serving the log file on the SDCARD
    *
    * @param void
    *
    * @return esp_err_t
*/
esp_err_t http_server_setup(void);


#endif /* WIFI_TASK_H */
```

components/wifi-http/wifi_task.c

```c
/*****************************************************************************/
/* UPBDRIVE Dashboard Firmware                                               */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#include <wifi_task.h>

////////////////////////////////////////////////////////////////////////////
////                        LOCAL VARIABLES                             ////
////////////////////////////////////////////////////////////////////////////

```

```
12 ////////////////////////////////////////////////////////////////////////////
13 ////                          GLOBAL FUNCTIONS                            ////
14 ////////////////////////////////////////////////////////////////////////////
15
16 ////////////////////////////////////////////////////////////////////////////
17 ////                          LOCAL MACROS                                ////
18 ////////////////////////////////////////////////////////////////////////////
19
20 ////////////////////////////////////////////////////////////////////////////
21 ////                          LOCAL CONSTANTS                             ////
22 ////////////////////////////////////////////////////////////////////////////
23
24 ////////////////////////////////////////////////////////////////////////////
25 ////                          FUNCTION DEFINITIONS                        ////
26 ////////////////////////////////////////////////////////////////////////////
27
28 static void wifi_event_handler(void* arg, esp_event_base_t event_base,
29                                int32_t event_id, void* event_data)
30 {
31     if (event_id == WIFI_EVENT_AP_STACONNECTED)
32     {
33         wifi_event_ap_staconnected_t* event = (wifi_event_ap_staconnected_t*) event_data;
34         ESP_LOGI("wifi station", "station "MACSTR" join, AID=%d",
35                  MAC2STR(event->mac), event->aid);
36     }
37     else if (event_id == WIFI_EVENT_AP_STADISCONNECTED)
38     {
39         wifi_event_ap_stadisconnected_t* event = (wifi_event_ap_stadisconnected_t*) event_data
    ;
40         ESP_LOGI("wifi station", "station "MACSTR" leave, AID=%d",
41                  MAC2STR(event->mac), event->aid);
42     }
43 }
44
45 void wifi_setup(void)
46 {
47     ESP_ERROR_CHECK(nvs_flash_init());
48     ESP_ERROR_CHECK(esp_netif_init());
49     ESP_ERROR_CHECK(esp_event_loop_create_default());
50     esp_netif_create_default_wifi_ap();
51
52     wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
53     ESP_ERROR_CHECK(esp_wifi_init(&cfg));
54
55     ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
56                                                         ESP_EVENT_ANY_ID,
57                                                         &wifi_event_handler,
58                                                         NULL,
59                                                         NULL));
60
61     wifi_config_t wifi_config = {
62         .ap = {
63             .ssid     = ESP_WIFI_SSID,
64             .password = ESP_WIFI_PASS,
65             .ssid_len = strlen(ESP_WIFI_SSID),
66             .channel  = ESP_WIFI_CHANNEL,
67             .authmode = WIFI_AUTH_WPA2_PSK,
68             .max_connection = 2,
69             .pmf_cfg = {
70                     .required = true,
71             },
72         },
73     };
74
75     if (strlen(ESP_WIFI_PASS) == 0) {
76         wifi_config.ap.authmode = WIFI_AUTH_OPEN;
77     }
78
79     ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_AP));
80     ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_AP, &wifi_config));
81     ESP_ERROR_CHECK(esp_wifi_start());
82
83     ESP_LOGI("wifi station", "wifi_init_softap finished. SSID:%s password:%s channel:%d",
84              ESP_WIFI_SSID, ESP_WIFI_PASS, ESP_WIFI_CHANNEL);
85 }
86
```

```
87
88 esp_err_t http_server_setup(void)
89 {
90     esp_err_t esp_response = ESP_FAIL;
91
92     /* Start the file server */
93     ESP_ERROR_CHECK(start_file_server(MOUNT_POINT));
94     ESP_LOGI("wifi station", "File server started");
95
96     esp_response = ESP_OK;
97     return esp_response;
98 }
```

components/wifi-http/inc/file_server.h

```
1
2 /* HTTP File Server Example, common declarations
3
4    This example code is in the Public Domain (or CC0 licensed, at your option.)
5
6    Unless required by applicable law or agreed to in writing, this
7    software is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
8    CONDITIONS OF ANY KIND, either express or implied.
9 */
10
11 #pragma once
12
13 #include "sdkconfig.h"
14 #include "esp_err.h"
15
16 #ifdef __cplusplus
17 extern "C" {
18 #endif
19
20
21 /* Mount point of the sdcard */
22 #define MOUNT_POINT "/sdcard"
23
24 esp_err_t start_file_server(const char *base_path);
25
26 #ifdef __cplusplus
27 }
28 #endif
```

components/wifi-http/file_server.c

```
1 /*****************************************************************************/
2 /* UPBDRIVE Dashboard Firmware                                               */
3 /* All rights reserverd 2023                                                 */
4 /*****************************************************************************/
5
6 #include <common.h>
7
8 #include <file_server.h>
9 #include <esp_http_server.h>
10 #include <esp_vfs.h>
11 #include <image.h>
12
13 ////////////////////////////////////////////////////////////////////////////
14 ////                        GLOBAL VARIABLES                            ////
15 ////////////////////////////////////////////////////////////////////////////
16
17 extern status_firmware_t general_status;
18
19 ////////////////////////////////////////////////////////////////////////////
20 ////                        LOCAL MACROS                                ////
21 ////////////////////////////////////////////////////////////////////////////
22
23 /* Scratch buffer size */
24 #define SCRATCH_BUFSIZE     (8192)
25
26 #define FILE_PATH_MAX       (150)
27
28 ////////////////////////////////////////////////////////////////////////////
29 ////                        LOCAL CONSTANTS                             ////
30 ////////////////////////////////////////////////////////////////////////////
31
```

```
32 static const char *TAG = "file_server";
33
34 /* The index.html file is embedded into the binary file uploaded in the ESP32
35    flash memory */
36 extern const char index_start[] asm ("_binary_index_html_start");
37
38 /* The path to dashboard log */
39 static const char *dashboard_log = MOUNT_POINT"/log.txt";
40
41 ////////////////////////////////////////////////////////////////////////////
42 ////                        LOCAL VARIABLES                             ////
43 ////////////////////////////////////////////////////////////////////////////
44
45 /*
46     Allocate the file server data struct containing:
47     - the base path of the files on the server, in this case MOUNT_POINT
48     - a scratchpad for file transfer
49 */
50 static struct file_server_data {
51     /* Base path of file storage */
52     char base_path[ESP_VFS_PATH_MAX + 1];
53
54     /* Scratch buffer for temporary storage during file transfer */
55     char scratch[SCRATCH_BUFSIZE];
56 } server_data;
57
58 static char http_response[200] = {0};
59
60 ////////////////////////////////////////////////////////////////////////////
61 ////                        LOCAL FUNCTIONS                             ////
62 ////////////////////////////////////////////////////////////////////////////
63
64 /* Handler to respond with index.html page */
65 static esp_err_t index_html_get_handler(httpd_req_t *req)
66 {
67     return httpd_resp_send(req, index_start, HTTPD_RESP_USE_STRLEN);
68 }
69
70 static esp_err_t logo_get_handler(httpd_req_t *req)
71 {
72     httpd_resp_set_type(req, "image/jpeg");
73     return httpd_resp_send(req, image, image_len);
74 }
75
76 /* Handler to download a the log file from sdcard */
77 static esp_err_t download_get_handler(httpd_req_t *req)
78 {
79     FILE *fd = NULL;
80     /* Retrieve the pointer to scratch buffer for temporary storage */
81     char *chunk = ((struct file_server_data *)req->user_ctx)->scratch;
82     size_t chunksize;
83
84     fd = fopen(dashboard_log, "r");
85     if (NULL == fd)
86     {
87         httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR, "Failed to read existing
    file");
88
89         return ESP_FAIL;
90     }
91
92     httpd_resp_set_type(req, "text/plain");
93     do {
94         /* Read file in chunks into the scratch buffer */
95         chunksize = fread(chunk, 1, SCRATCH_BUFSIZE, fd);
96
97         if (chunksize > 0) {
98             /* Send the buffer contents as HTTP response chunk */
99             if (httpd_resp_send_chunk(req, chunk, chunksize) != ESP_OK) {
100                 fclose(fd);
101                 ESP_LOGE(TAG, "File sending failed!");
102                 /* Abort sending file */
103                 httpd_resp_sendstr_chunk(req, NULL);
104                 /* Respond with 500 Internal Server Error */
105                 httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR, "Failed to send file
    ");
```

```
106                 return ESP_FAIL;
107             }
108         }
109 
110         /* Keep looping till the whole file is sent */
111     } while (chunksize != 0);
112 
113     /* Close file after sending complete */
114     fclose(fd);
115     ESP_LOGI(TAG, "File sending complete");
116 
117     httpd_resp_send_chunk(req, NULL, 0);
118     return ESP_OK;
119 }
120 
121 static esp_err_t data_get_handler(httpd_req_t *req)
122 {
123     status_firmware_t* general_status = (status_firmware_t*)req->user_ctx;
124     display_data_t* display_data = general_status->display_data;
125 
126     if (xSemaphoreTake (general_status->xSemaphore_display_data, ( TickType_t ) 10) == pdTRUE)
127     {
128         sprintf(http_response, "%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %f %f %f %d %d
     %d %d %d",
129                 display_data->rpm,
130                 display_data->coolant_temperature,
131                 display_data->oil_temperature,
132                 display_data->battery_voltage,
133                 display_data->can_status,
134                 display_data->hybrid_status,
135                 display_data->safety_circuit_status,
136                 display_data->fan_state,
137                 display_data->tps,
138                 display_data->brake_pressure_raw,
139                 display_data->map,
140                 display_data->lambda,
141                 display_data->tyre_pressure[0],
142                 display_data->tyre_pressure[1],
143                 display_data->tyre_pressure[2],
144                 display_data->tyre_pressure[3],
145                 display_data->accelerometer_g[0],
146                 display_data->accelerometer_g[1],
147                 display_data->accelerometer_g[2],
148                 general_status->time_hour,
149                 general_status->time_minute,
150                 general_status->time_second,
151                 general_status->sdcard_logging,
152                 display_data->hybrid_selector_value
153                 );
154 
155         xSemaphoreGive (general_status->xSemaphore_display_data);
156     }
157 
158     return httpd_resp_send(req, http_response, HTTPD_RESP_USE_STRLEN);
159 }
160 
161 /* Function to start the file server */
162 esp_err_t start_file_server(const char *base_path)
163 {
164     httpd_handle_t http_server = NULL;
165     httpd_config_t config = HTTPD_DEFAULT_CONFIG();
166 
167     /* Use the URI wildcard matching function in order to
168      * allow the same handler to respond to multiple different
169      * target URIs which match the wildcard scheme */
170     config.uri_match_fn = httpd_uri_match_wildcard;
171 
172     ESP_LOGI(TAG, "Starting HTTP Server on port: '%d'", config.server_port);
173     if (httpd_start(&http_server, &config) != ESP_OK) {
174         ESP_LOGE(TAG, "Failed to start file server!");
175         return ESP_FAIL;
176     }
177 
178     /* Start page URI */
179     httpd_uri_t index_html_uri = {
180         .uri       = "/",
```

```
        .method    = HTTP_GET,
        .handler   = index_html_get_handler,
        .user_ctx  = NULL
    };
    httpd_register_uri_handler(http_server, &index_html_uri);

    /* Log download URI */
    httpd_uri_t log_download_uri = {
        .uri       = "/sdcard/log.txt",
        .method    = HTTP_GET,
        .handler   = download_get_handler,
        .user_ctx  = &server_data
    };
    httpd_register_uri_handler(http_server, &log_download_uri);

    /* Log live data URI */
    httpd_uri_t live_data_uri = {
        .uri       = "/data",
        .method    = HTTP_GET,
        .handler   = data_get_handler,
        .user_ctx  = &general_status
    };
    httpd_register_uri_handler(http_server, &live_data_uri);

    /* Image URI */
    httpd_uri_t image_uri = {
        .uri       = "/UPBDRIVE_Logo_Horizontal.jpg",
        .method    = HTTP_GET,
        .handler   = logo_get_handler,
        .user_ctx  = &general_status
    };
    httpd_register_uri_handler(http_server, &image_uri);

    return ESP_OK;
}
```

components/wifi-http/html/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title color: white>UPBDrive Formula Student Team - File Download</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background-color: #241e1e;
      margin: 20px;
    }

    h2 {
      color: white;
    }

    .logo {
      width: 60%;  /* Responsive width */
      max-width: 700px;  /* Maximum width */
      align-items: center;
      height: auto;
    }

    .content {
    display: flex;
    flex-direction: column;
    align-items: center;
    }

    title {
      color: white;
    }
    .container {
      text-align: center;
    }
    .download-btn {
      display: inline-block;
      padding: 10px 20px;
```

```
      font-size: 16px;
      background-color: #ff9100;
      color: black;
      text-decoration: none;
      border-radius: 5px;
      border: none;
      cursor: pointer;
    }
    .download-btn:hover {
      background-color: #ffffff;
    }

    .table-container {
    border: 1px solid #cccccc;
    border-radius: 8px;
    overflow: hidden;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    }

    table {
        width: 100%;
        border-collapse: collapse;
        text-align: left;
        background-color: #fff;
    }

    th, td {
        padding: 12px 15px;
        border-bottom: 1px solid #cccccc;
    }

    th {
        background-color: #ff9100;
    }

    tbody tr:hover {
        background-color: #f9f9f9;
}
  </style>
</head>
<body>
  <div class="content">
    <img class="logo" src="/UPBDRIVE_Logo_Horizontal.jpg" alt="Embedded Image">
  </div>
  <div class="container">
    <h2>DR-05 Log file</h2>
    <a class="download-btn" href="/sdcard/log.txt" download="log.txt">Download File</a>
  </div>
  <br>
  <br>
  <div class="table-container">
    <div class="container">
        <table>
            <th colspan="8">Live Data</th>
            <tr>
                <th>RPM</th>
                <th>COOLANT TEMP</th>
                <th>OIL PRESSURE</th>
                <th>LV BATTERY</th>
            </tr>
            <tr>
                <td id="rpm">-</td>
                <td id="coolant_temperature">-</td>
                <td id="oil_pressure">-</td>
                <td id="lv_battery">-</td>
            </tr>
            <tr>
              <th>CAN status</th>
              <th>HYBRID status</th>
              <th>SAFETY circuit status</th>
              <th>FAN status</th>
            </tr>
            <tr>
              <td id="can_status">-</td>
              <td id="hybrid_status">-</td>
              <td id="safety_circuit_status">-</td>
```

```
            <td id="fan_status">-</td>
          </tr>
          <tr>
            <th>TPS %</th>
            <th>BRAKE APPL %</th>
            <th>MAP</th>
            <th>LAMBDA</th>
          </tr>
          <tr>
            <td id="tps">-</td>
            <td id="brake_appl">-</td>
            <td id="map">-</td>
            <td id="lambda">-</td>
          </tr>
          <tr>
            <th>TYRE LF</th>
            <th>TYRE RF</th>
            <th>TYRE LR</th>
            <th>TYRE RR</th>
          </tr>
          <tr>
            <td id="tyre_lf">-</td>
            <td id="tyre_rf">-</td>
            <td id="tyre_lr">-</td>
            <td id="tyre_rr">-</td>
          </tr>
          <tr>
            <th>ACCEL_X</th>
            <th>ACCEL_Y</th>
            <th>ACCEL_Z</th>
            <th>TIME   </th>
          </tr>
          <tr>
            <td id="accel_x">-</td>
            <td id="accel_y">-</td>
            <td id="accel_z">-</td>
            <td id="time">-</td>
          </tr>
          <tr>
            <th>SDCARD_LOGGING</th>
            <th>HYBRID SELECTOR</th>
            <th></th>
            <th></th>
          </tr>
          <tr>
            <td id="sdcard_logging">-</td>
            <td id="hybrid_selector">-</td>
          </tr>

        </table>
    </div>
</div>

<script>

  // function to obtain data from ESP32 backend
  function getData()
  {
      let xhr = new XMLHttpRequest();

      // /data URI returns a string with data
      xhr.open("GET", "/data", false);
      xhr.onload = function()
      {
          let str = xhr.response;

          // the values are in "number number number number", where number is a
          // an integer value, therefore it can be split by the space character
          let values = str.split(" ").map(Number);

          // update the values in the table
          document.getElementById("rpm").innerText = values[0];
          document.getElementById("coolant_temperature").innerText = values[1];
          document.getElementById("oil_pressure").innerText = values[2];
          document.getElementById("lv_battery").innerText = values[3] / 10;
          document.getElementById("tps").innerText = values[8];
```

```
192           document.getElementById("brake_appl").innerText = values[9];
193           document.getElementById("map").innerText = values[10];
194           document.getElementById("lambda").innerText = values[11];
195           document.getElementById("tyre_lf").innerText = values[12];
196           document.getElementById("tyre_rf").innerText = values[13];
197           document.getElementById("tyre_lr").innerText = values[14];
198           document.getElementById("tyre_rr").innerText = values[15];
199           document.getElementById("accel_x").innerText = values[16];
200           document.getElementById("accel_y").innerText = values[17];
201           document.getElementById("accel_z").innerText = values[18];
202           document.getElementById("time").innerText = values[19] + ":" + values[20] + ":" +
      values[21];
203           document.getElementById("hybrid_selector").innerText = values[23];
204
205           if (values[4] == 1)
206           {
207             document.getElementById("can_status").innerText = "OK";
208           }
209           else
210           {
211             document.getElementById("can_status").innerText = "ERROR";
212           }
213
214           if (values[5] == 1)
215           {
216             document.getElementById("hybrid_status").innerText = "OK";
217           }
218           else
219           {
220             document.getElementById("hybrid_status").innerText = "ERROR";
221           }
222
223           if (values[6] == 1)
224           {
225             document.getElementById("safety_circuit_status").innerText = "OK";
226           }
227           else
228           {
229             document.getElementById("safety_circuit_status").innerText = "ERROR";
230           }
231
232           if (values[7] == 0)
233           {
234             document.getElementById("fan_status").innerText = "OFF";
235           }
236           else
237           {
238             document.getElementById("fan_status").innerText = "ON";
239           }
240           if (values[22] == 0)
241           {
242             document.getElementById("sdcard_logging").innerText = "ERROR";
243           }
244           else
245           {
246             document.getElementById("sdcard_logging").innerText = "OK";
247           }
248       }
249
250       xhr.send();
251       setTimeout(getData, 500);
252   }
253
254   getData();
255 </script>
256
257 </body>
258 </html>
```

Annex B
Test module source code

main/main.cpp

```
/*****************************************************************************/
/* UPBDRIVE ECU-Emulator Firmware                                            */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

/* Include components */
#include "adc.h"
#include "can.h"
#include "heartbeat.h"
#include "sawtooth-signal.h"

#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/adc.h"
#include <string.h>

/* Global data structure for display data */
static display_data_t display_data = {0};

int main() {

    /* This 2 values need to be init to nominal
       values in order to avoid dashboard showing error */
    display_data.battery_voltage     = 123U;
    display_data.oil_pressure        = 100U;

    /* This values are emulating the DAT system
        that will send a CAN packet with the time */
    display_data.time_hour = 12U;
    display_data.time_minute = 30U;
    display_data.time_second = 45U;

    /* sawtooth signal variable */
    uint8_t sawtooth = 0U;

    /* set heartbeat led state to OFF */
    uint8_t heartbeat_state = 0U;

    /* initialize usb uart communication */
    stdio_init_all();

    /* initialize adc and shift button */
    init_adc_and_buttons();

    /* initialize the can bus */
    #ifdef SUPPORT_CAN2040
    canbus_setup();
    #endif /* SUPPORT_CAN2040 */

    #ifdef SUPPORT_MCP2515
    mcp2515_setup();
    #endif /* SUPPORT_MCP2515 */

    /* initialize heartbeat led */
    init_heartbeat(&heartbeat_state);

    while(true) {
        /* fill display_data structure */
        read_adc_and_button(&display_data);
        fill_with_sawtooth(&display_data);

        /* send the display_data struct fields via CAN */
```

```
        send_can(&display_data);

        /* toggle the heartbeat LED */
        toggle_heartbeat(&heartbeat_state);

        #ifdef ENABLE_DEBUG_PRINTF
        //send via USB UART the data to be sent to CAN bus
        printf("rpm          = %d\n", display_data.rpm);
        printf("oil_temp     = %d\n", display_data.oil_temperature);
        printf("coolant_temp = %d\n", display_data.coolant_temperature);
        printf("gear         = %d\n", display_data.current_gear);
        printf("battery      = %d\n", display_data.battery_voltage);
        printf("oil_pressure = %d\n", display_data.oil_pressure);
        sleep_ms(250);
        #endif /* ENABLE_DEBUG_PRINTF */
    }

    return 0;
}
```

components/adc/inc/adc.h

```
/*****************************************************************************/
/* UPBDRIVE ECU-Emulator Firmware                                            */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#ifndef ADC_H
#define ADC_H

#include "common.h"
#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL MACROS                                   ////
////////////////////////////////////////////////////////////////////////////////

#define ADC_VREF 3.3
#define ADC_RANGE 4096

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL CONSTANTS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        FUNCTION PROTOTYPES                             ////
////////////////////////////////////////////////////////////////////////////////

//reads TPS, OIL, BPS then returns data in %
void read_adc_and_button(struct display_data_t* display_data);

/*
    * @brief Initialize the ADC
*/
void init_adc_and_buttons();


#endif /* ADC_H */
```

components/adc/adc.cpp

```
/*****************************************************************************/
/* UPBDRIVE ECU-Emulator Firmware                                            */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#include "adc.h"

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                                ////
////////////////////////////////////////////////////////////////////////////////
```

```
11
12 static uint8_t button_state_gearshift      = 0U;
13 static uint8_t button_state_lowbattery     = 0U;
14 static uint8_t button_state_lowoilpressure = 0U;
15
16 ////////////////////////////////////////////////////////////////////////////
17 ////                        GLOBAL FUNCTIONS                            ////
18 ////////////////////////////////////////////////////////////////////////////
19
20 ////////////////////////////////////////////////////////////////////////////
21 ////                        LOCAL MACROS                                ////
22 ////////////////////////////////////////////////////////////////////////////
23
24 /* The ADC inputs are connected to 3 potentiometers
25     - ADC0 input emulates the RPM value
26     - ADC1 input emulates the OIL_TEMPERATURE value
27     - ADC2 input emulates the COOLANT_TEMPERATURE value */
28 #define ADC_INPUT_RPM               (0)
29 #define ADC_INPUT_OIL_TEMP          (1)
30 #define ADC_INPUT_COOLANT_TEMP      (2)
31
32 ////////////////////////////////////////////////////////////////////////////
33 ////                        LOCAL CONSTANTS                             ////
34 ////////////////////////////////////////////////////////////////////////////
35
36 void init_adc_and_buttons()
37 {
38     /* initialise ADC */
39     adc_init();
40     adc_gpio_init(GPIO_NUM_POTENTIOMETER);
41     adc_gpio_init(GPIO_NUM_OIL_TEMP);
42     adc_gpio_init(GPIO_NUM_COOLANT_TEMP);
43
44     /* initialize button gear shift */
45     gpio_init(GPIO_NUM_GEAR_SHIFT);
46     gpio_set_dir(GPIO_NUM_GEAR_SHIFT, GPIO_IN);
47     gpio_set_pulls(GPIO_NUM_GEAR_SHIFT, true, false);
48
49     /* initialize button low battery */
50     gpio_init(GPIO_NUM_LOW_BATTERY);
51     gpio_set_dir(GPIO_NUM_LOW_BATTERY, GPIO_IN);
52     gpio_set_pulls(GPIO_NUM_LOW_BATTERY, true, false);
53
54     /* initialize button low oil pressure */
55     gpio_init(GPIO_NUM_LOW_OIL_PRESSUIRE);
56     gpio_set_dir(GPIO_NUM_LOW_OIL_PRESSUIRE, GPIO_IN);
57     gpio_set_pulls(GPIO_NUM_LOW_OIL_PRESSUIRE, true, false);
58 }
59
60 void read_adc_and_button(struct display_data_t* display_data)
61 {
62     //initialise buffers for adc
63     uint16_t rpm_value     = 0UL;
64     uint16_t oil_temp      = 0UL;
65     uint16_t coolant_temp  = 0UL;
66
67     //read the rpm potentiometer
68     adc_select_input(ADC_INPUT_RPM);
69     rpm_value = adc_read();
70
71     //read the oil_temp potentiometer
72     adc_select_input(ADC_INPUT_OIL_TEMP);
73     oil_temp = adc_read();
74
75     //read the coolant_temp potentiometer
76     adc_select_input(ADC_INPUT_COOLANT_TEMP);
77     coolant_temp = adc_read();
78
79     /* Set the ADC data in the display_data structure, after normalizing it  */
80     display_data->rpm                = (uint16_t)((rpm_value * 13000) / ADC_RANGE);
81     display_data->oil_temperature    = (uint8_t)((oil_temp * 170) / ADC_RANGE);
82     display_data->coolant_temperature = (uint8_t)((coolant_temp * 120) / ADC_RANGE);
83
84     /* If the gear shift button is pressed, increase the gear value */
85     if ((gpio_get(GPIO_NUM_GEAR_SHIFT) == 0U) && (button_state_gearshift == 0U))
86     {
```

```
        button_state_gearshift = 0xFF;

        display_data->current_gear++;

        if (display_data->current_gear >= 5U)
        {
            display_data->current_gear = 0U;
        }
    }
    /* This else branch is meant to protect the incrementing from happening
    unless the button is released */
    else if ((gpio_get(GPIO_NUM_GEAR_SHIFT) != 0U) && (button_state_gearshift == 0xFF))
    {
        button_state_gearshift = 0U;
    }


    /* If the low battery button is pressed, set the battery_voltage to 11V */
    if ((gpio_get(GPIO_NUM_LOW_BATTERY) == 0U) && (button_state_lowbattery == 0U))
    {
        button_state_lowbattery = 0xFF;

        display_data->battery_voltage = 110U;
    }
    /* This else branch is meant to protect the incrementing from happening
    unless the button is released */
    else if ((gpio_get(GPIO_NUM_LOW_BATTERY) != 0U) && (button_state_lowbattery == 0xFF))
    {
        button_state_lowbattery = 0x00;
        display_data->battery_voltage = 123U;
    }


    /* If the low oil pressure is pressed, set the oil_pressure to 0U */
    if ((gpio_get(GPIO_NUM_LOW_OIL_PRESSUIRE) == 0U) && (button_state_lowoilpressure == 0U))
    {
        button_state_lowoilpressure = 0xFF;

        display_data->oil_pressure = 0U;
    }
    /* This else branch is meant to protect the incrementing from happening
    unless the button is released */
    else if ((gpio_get(GPIO_NUM_LOW_OIL_PRESSUIRE) != 0U) && (button_state_lowoilpressure == 0
    xFF))
    {
        button_state_lowoilpressure = 0x00;
        display_data->oil_pressure = 100U;
    }
}
```

components/can/inc/can.h

```
/*****************************************************************************/
/* UPBDRIVE ECU-Emulator Firmware                                            */
/* All rights reserverd 2023                                                 */
/*****************************************************************************/

#ifndef CAN_H
#define CAN_H

#include "common.h"
#include "can2040.h"
#include "RP2040.h"
#include "pico/stdlib.h"
#include "pico/binary_info.h"

#include "mcp2515.h"

////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                            ////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
////                        GLOBAL MACROS                               ////
////////////////////////////////////////////////////////////////////////////

/*
```

```
26      * @brief data structure:
27      * data[0]: RPM / 100
28      * data[1]: CurrGear
29      * data[2]: TPS
30      * data[3]: Oil pressure
31      * data[4]: Water temperature
32      * data[5]: Fuel pressure
33      * data[6]: Lamda
34      * data[7]: IAT
35  */
36  #define CAN_PACKET_ECU_1 (0x3E8)
37
38  /*
39      * @brief data structure:
40      * data[0]: EGT1 temp / 100
41      * data[1]: EGT2 temp / 100
42      * data[2]: EGT3 temp / 100
43      * data[3]: EGT4 temp / 100
44      * data[4]: Vehicle speed
45      * data[5]: Manifold Air Pressure (MAP)
46      * data[6]: BPS raw value
47      * data[7]: Oil temperature
48  */
49  #define CAN_PACKET_ECU_2 (0x3E9)
50
51  /*
52      * @brief data structure:
53      * data[0]: LV battery voltage
54      * data[1]: Input voltage
55      * data[2]: Input current battery
56      * data[3]: Input current alternator
57  */
58  #define CAN_PACKET_PMU_1 (0x3ED)
59
60  /*
61      * @brief data structure:
62      * data[4]: Tyre pressure Left-Front
63    * data[5]: Tyre pressure Right-Front
64    * data[6]: Tyre pressure Left-Rear
65    * data[7]: Tyre pressure Right-Rear
66  */
67  #define CAN_PACKET_VDU (0x3EC)
68
69  /*
70      * @brief data structure:
71      * data[0]: Selector value
72  */
73  #define CAN_PACKET_DASH (0x384)
74
75  /*
76      * @brief data structure:
77      * data[0]: HOUR value
78      * data[1]: MINUTE value
79      * data[2]: SECOND value
80  */
81  #define CAN_PACKET_DAT (0x10)
82
83  ////////////////////////////////////////////////////////////////////////////
84  ////                        GLOBAL CONSTANTS                            ////
85  ////////////////////////////////////////////////////////////////////////////
86
87  ////////////////////////////////////////////////////////////////////////////
88  ////                        FUNCTION PROTOTYPES                         ////
89  ////////////////////////////////////////////////////////////////////////////
90
91  //generate and send CAN package
92  void send_can(display_data_t *data);
93
94  //function to setup the CAN bus
95  void canbus_setup();
96  void mcp2515_setup();
97
98  #endif /* CAN_H */
```

components/can/can.cpp

```
1  /*****************************************************************************/
```

```c
/* UPBDRIVE ECU-Emulator Firmware                                              */
/* All rights reserverd 2023                                                   */
/*****************************************************************************/

#include "can.h"
#include "string.h"

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL VARIABLES                                ////
////////////////////////////////////////////////////////////////////////////////

#ifdef SUPPORT_CAN2040
/*
    Structure that is used in order to send / receive can packets via the CAN2040
    library
*/
static struct can2040 cbus;
#endif /* SUPPORT_CAN2040 */

#ifdef SUPPORT_MCP2515
/* Object that is used to send can packets via the MCP2515 dev-board
    - The pins are defined as per the schematic of ECU-Emulator
    - SPI0_SPEED is set to 10MHz
    - The MCP2515 is connected to SPI0, because GPIO_CS_MCP2515, GPIO_MOSI_MCP2515,
    GPIO_MISO_MCP2515, GPIO_SCK_MCP2515 are connected to the SPI0 ip of the Raspberry
    Pi Pico
*/
static MCP2515 can0(
        spi0,
        GPIO_CS_MCP2515,
        GPIO_MOSI_MCP2515,
        GPIO_MISO_MCP2515,
        GPIO_SCK_MCP2515,
        SPI0_SPEED
    );
#endif /* SUPPORT_MCP2515 */
////////////////////////////////////////////////////////////////////////////////
////                        LOCAL VARIABLES                                 ////
////////////////////////////////////////////////////////////////////////////////

static uint8_t counter_dat_packets = 0U; // Counter for the number of data packets sent

////////////////////////////////////////////////////////////////////////////////
////                        GLOBAL FUNCTIONS                                ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL MACROS                                    ////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////                        LOCAL CONSTANTS                                 ////
////////////////////////////////////////////////////////////////////////////////

#ifdef SUPPORT_CAN2040
//function to read data from the CAN bus
static void can2040_cb(struct can2040 *cd, uint32_t notify, struct can2040_msg *msg);

//function to handle interrupts from the PIO
static void PIOx_IRQHandler();

//function to send CAN package
static void can_transmit(struct can2040_msg *outbound);

//function to read CAN packages
static void can2040_cb(struct can2040 *cd, uint32_t notify, struct can2040_msg *msg)
{
    struct can2040_msg message;
    memcpy(&message, msg, sizeof(message));

    if ((message.id != CAN_PACKET_ECU_1) && (message.id != CAN_PACKET_ECU_2)) \
    {
        printf("CAN message with id %x\n", message.id);
        printf("Data[0] %d\n", message.data[0]);
        printf("\n");
    }
```

```
78
79 }
80
81 static void PIOx_IRQHandler()
82 {
83     can2040_pio_irq_handler(&cbus);
84 }
85
86 void canbus_setup()
87 {
88     uint32_t pio_num = 0;
89     uint32_t sys_clock = 125000000, bitrate = 500e3;
90     uint32_t gpio_rx = 4, gpio_tx = 5;
91
92     // Setup canbus
93     can2040_setup(&cbus, pio_num);
94     can2040_callback_config(&cbus, can2040_cb);
95
96     // Enable irqs
97     irq_set_exclusive_handler(PIO0_IRQ_0_IRQn, PIOx_IRQHandler);
98     NVIC_SetPriority(PIO0_IRQ_0_IRQn, 1);
99     NVIC_EnableIRQ(PIO0_IRQ_0_IRQn);
100
101    // Start canbus
102    can2040_start(&cbus, sys_clock, bitrate, gpio_rx, gpio_tx);
103 }
104
105 void can_transmit(struct can2040_msg *outbound)
106 {
107     if(can2040_check_transmit(&cbus)) {
108         if(can2040_transmit(&cbus, outbound) == 0) {
109             //printf("Packet %d was transmitted \n", outbound->id);
110         }
111     }
112     else {
113         printf("No space \n");
114     }
115 }
116 #endif /* SUPPORT_CAN2040 */
117
118 #ifdef SUPPORT_MCP2515
119 void mcp2515_setup()
120 {
121     can0.reset();
122     can0.setBitrate(CAN_500KBPS, MCP_8MHZ);
123     can0.setNormalMode();
124 }
125 #endif /* SUPPORT_MCP2515 */
126
127
128 void send_can(display_data_t *data)
129 {
130     ////////////////////////////////////////////////////////////////////////
131     ////                        DEFINE CAN PACKETS                      ///
132     ////////////////////////////////////////////////////////////////////////
133     #ifdef SUPPORT_CAN2040
134     struct can2040_msg outbound1;
135     outbound1.id = CAN_PACKET_ECU_1;
136     outbound1.dlc = 8;
137
138     struct can2040_msg outbound2;
139     outbound2.id = CAN_PACKET_ECU_2;
140     outbound2.dlc = 8;
141     #endif /* SUPPORT_CAN2040 */
142
143     #ifdef SUPPORT_MCP2515
144     struct can_frame outbound1;
145     outbound1.can_id = CAN_PACKET_ECU_1;
146     outbound1.can_dlc = 8;
147
148     struct can_frame outbound2;
149     outbound2.can_id = CAN_PACKET_ECU_2;
150     outbound2.can_dlc = 8;
151
152     struct can_frame outbound3;
153     outbound3.can_id  = CAN_PACKET_VDU;
```

```
154     outbound3.can_dlc = 8;
155
156     struct can_frame outbound4;
157     outbound4.can_id  = CAN_PACKET_PMU_1;
158     outbound4.can_dlc = 8;
159
160     struct can_frame outbound5;
161     outbound5.can_id  = CAN_PACKET_DAT;
162     outbound5.can_dlc = 8;
163     #endif /* SUPPORT_MCP2515 */
164
165     ////////////////////////////////////////////////////////////////////////////
166     ////                        INITIALIZE CAN PACKET DATA                  ///
167     ////////////////////////////////////////////////////////////////////////////
168     /* RPM */
169     outbound1.data[0] = data->rpm / 100;
170
171     /* CurrGear */
172     outbound1.data[1] = data->current_gear;
173
174     /* TPS*/
175     outbound1.data[2] = data->tps;
176
177     /* Oil pressure */
178     outbound1.data[3] = data->oil_pressure;
179
180     /* Water temperature */
181     outbound1.data[4] = data->coolant_temperature;
182
183     /* Fuel rail pressure */
184     outbound1.data[5] = data->fuel_pressure;
185
186     /* Lambda */
187     outbound1.data[6] = data->lambda;
188
189     /* IAT */
190     /* Not shown on display, use rpm as placeholder */
191     outbound1.data[7] = data->rpm / 100;
192
193     /* EGT 1 */
194     outbound2.data[0] = data->egt[0];
195
196     /* EGT 2 */
197     outbound2.data[1] = data->egt[1];
198
199     /* EGT 3 */
200     outbound2.data[2] = data->egt[2];
201
202     /* EGT 4 */
203     outbound2.data[3] = data->egt[3];
204
205     /* Veh speed */
206     /* Not shown on display, use rpm as placeholder */
207     outbound2.data[4] = data->rpm / 100;
208
209     /* MAP */
210     outbound2.data[5] = data->map;
211
212     /* Brake pressure (raw) */
213     outbound2.data[6] = data->brake_pressure_raw;
214
215     /* Oil temperature */
216     outbound2.data[7] = data->oil_temperature;
217
218     /* Tyre pressure from VDU */
219     outbound3.data[4] = data->rpm/100;
220     outbound3.data[5] = data->rpm/100;
221     outbound3.data[6] = data->rpm/100;
222     outbound3.data[7] = data->rpm/100;
223
224     /* LV battery voltage */
225     outbound4.data[0] = data->battery_voltage;
226     outbound4.data[1] = data->rpm/100;
227     outbound4.data[2] = data->rpm/100;
228
229     /* RTC data from DAT */
```

```
230     outbound5.data[0] = data->time_hour;
231     outbound5.data[1] = data->time_minute;
232     outbound5.data[2] = data->time_second;
233 
234     ////////////////////////////////////////////////////////////////////////
235     ////                        SEND CAN PACKETS                        ///
236     ////////////////////////////////////////////////////////////////////////
237     #ifdef SUPPORT_CAN2040
238     can_transmit(&outbound1);
239 
240     /* Wait for outbound1 packet to be sent */
241     sleep_ms(20);
242 
243     can_transmit(&outbound2);
244     #endif /* SUPPORT_CAN2040 */
245 
246     #ifdef SUPPORT_MCP2515
247     if(can0.sendMessage(&outbound1) == MCP2515::ERROR_OK)
248     {
249         printf("Message sent: %d\n", outbound1.can_id);
250     }
251     else
252     {
253         printf("Error sending message\n");
254     }
255 
256     /* Wait for outbound1 packet to be sent */
257     sleep_ms(5);
258 
259     if(can0.sendMessage(&outbound2) == MCP2515::ERROR_OK)
260     {
261         printf("Message sent: %d\n", outbound2.can_id);
262     }
263     else
264     {
265         printf("Error sending message\n");
266     }
267 
268     /* Wait for outbound2 packet to be sent */
269     sleep_ms(5);
270 
271     if(can0.sendMessage(&outbound3) == MCP2515::ERROR_OK)
272     {
273         printf("Message sent: %d\n", outbound3.can_id);
274     }
275     else
276     {
277         printf("Error sending message\n");
278     }
279 
280     /* Wait for outbound3 packet to be sent */
281     sleep_ms(5);
282 
283     if(can0.sendMessage(&outbound4) == MCP2515::ERROR_OK)
284     {
285         printf("Message sent: %d\n", outbound4.can_id);
286     }
287     else
288     {
289         printf("Error sending message\n");
290     }
291 
292     /* Wait for outbound4 packet to be sent */
293     sleep_ms(5);
294     if(can0.sendMessage(&outbound5) == MCP2515::ERROR_OK)
295     {
296         printf("Message sent: %d\n", outbound5.can_id);
297     }
298     else
299     {
300         printf("Error sending message\n");
301     }
302 
303     if (counter_dat_packets < 10)
304     {
305         sleep_ms(5);
```

```
306         if(can0.sendMessage(&outbound5) == MCP2515::ERROR_OK)
307         {
308             printf("Message sent: %d\n", outbound5.can_id);
309         }
310         else
311         {
312             printf("Error sending message\n");
313         }
314         counter_dat_packets++;
315     }
316
317     /* Wait for outbound5 packet to be sent */
318     sleep_ms(20);
319
320     if(can0.readMessage(&outbound1) == MCP2515::ERROR_OK)
321     {
322         printf("New frame from ID: %10x\n", outbound1.can_id);
323     }
324     #endif /* SUPPORT_MCP2515 */
325 }
```

components/common/inc/common.h

```
1  /*********************************************************************************/
2  /* UPBDRIVE ECU-Emulator Firmware                                                */
3  /* All rights reserverd 2023                                                     */
4  /*********************************************************************************/
5
6  #ifndef COMMON_H
7  #define COMMON_H
8
9  #include <stdio.h>
10
11 ////////////////////////////////////////////////////////////////////////////////
12 ////                        ECU EMULATOR SETTINGS                           ////
13 ////////////////////////////////////////////////////////////////////////////////
14 //#define ENABLE_DEBUG_PRINTF
15
16 /* Select the library to be used in order to send / receive can packets */
17 //#define SUPPORT_CAN2040
18 #define SUPPORT_MCP2515
19
20 /* SPI0 bus speed, do not modify! */
21 #define SPI0_SPEED          (10000000)
22
23 ////////////////////////////////////////////////////////////////////////////////
24 ////                        PINS SETTINGS                                   ////
25 ////////////////////////////////////////////////////////////////////////////////
26
27 #define GPIO_CS_MCP2515          (17)
28 #define GPIO_MOSI_MCP2515        (19)
29 #define GPIO_MISO_MCP2515        (16)
30 #define GPIO_SCK_MCP2515         (18)
31
32 #define GPIO_NUM_POTENTIOMETER      (26)
33 #define GPIO_NUM_OIL_TEMP           (27)
34 #define GPIO_NUM_COOLANT_TEMP       (28)
35 #define GPIO_NUM_GEAR_SHIFT         (22)
36 #define GPIO_NUM_LOW_BATTERY        (14)
37 #define GPIO_NUM_LOW_OIL_PRESSUIRE  (15)
38 #define GPIO_NUM_HEARTBEAT          PICO_DEFAULT_LED_PIN
39
40 ////////////////////////////////////////////////////////////////////////////////
41 ////                          GLOBAL MACROS                                 ////
42 ////////////////////////////////////////////////////////////////////////////////
43
44 /* Get the first 8 bits in uint16_t */
45 #define GET_MSB(x) (uint8_t)((x >> 8) & 0xFF)
46
47 /* Get the last 8 bits in uint16_t */
48 #define GET_LSB(x) (uint8_t)(x & 0xFF)
49
50 ////////////////////////////////////////////////////////////////////////////////
51 ////                          GLOBAL TYPEDEFS                               ////
52 ////////////////////////////////////////////////////////////////////////////////
53 /* Structure that is used in dashboard firmware to save the data */
54 typedef struct display_data_t {
```

```
55
56      /*************************************************************
57       *                  MAIN DISPLAY PAGE                        *
58       *************************************************************/
59      /* Expected rpm is between 0 - 12.000 (uint16_t max 65.536) */
60      uint16_t rpm;
61
62      /* Current gear is between 0 - 5 (uint8_t max 255) */
63      uint8_t current_gear;
64
65      /* Coolant temperature is between 0 - 100 (uint8_t max 255) */
66      uint8_t coolant_temperature;
67
68      /* Radiator Fan state is either true for ON or false for OFF */
69      uint8_t fan_state;
70
71      /* Oil temperature value is between 0 - 150 (uint8_t max 255) */
72      uint8_t oil_temperature;
73
74
75      /*************************************************************
76       *              SECOND DISPLAY PAGE for debug                *
77       *************************************************************/
78      /* CAN bus status is 0x00 for ON or 0xFF for OFF */
79      uint8_t can_status;
80
81      /* Hybrid system status is 0x00 for OK or 0xFF for FAULT */
82      uint8_t hybrid_status;
83
84      /* Safety circuit status is 0x00 for OK or 0xFF for FAULT */
85      uint8_t safety_circuit_status;
86
87      /* Oil pressure value is between 0 - 150 (uint8_t max 255) */
88      uint8_t oil_pressure;
89
90      /* Raw bps value from the sensor ToDo */
91      uint8_t brake_pressure_raw;
92
93      /* Throttle Position Sensor value is between 0 - 100 (uint8_t max 255) */
94      uint8_t tps;
95
96      /* Oil temperature value is between 0 - 150 (uint8_t max 255)
97
98         @note: the field oil_temperature is scaled before being sent to MAIN_PAGE
99                this field will contain the unscaled version of oil_temperature from
100               the CAN message
101     */
102     uint8_t oil_temperature_2;
103
104     /* Hybrid status value HYBRID_STATE_1 - HYBRID_STATE_11 or HYBRID_STATE_ERROR */
105     uint8_t hybrid_selector_value;
106
107     /* Battery voltage is between 0 - 240 (uint8_t max 255) */
108     uint8_t battery_voltage;
109
110
111     /*************************************************************
112     *                 THIRD DISPLAY PAGE for debug                *
113     *************************************************************/
114     /* Lambda value between 0 - 155 (uint8_t max 255) */
115     uint8_t lambda;
116
117     /* Manifold Absolute Pressure MAP value 0 - 45 (uint8_t max 255) */
118     uint8_t map;
119
120     /* Fuel pressure value is between 0 - 45 (uint8_t max 255) */
121     uint8_t fuel_pressure;
122
123     /* EGT values between 0 - 450 (uint8_t max 255) */
124     uint16_t egt[4];
125
126     /* Voltage on the PMU input rail 0 - 14 (uint8_t max 255) */
127     uint8_t input_voltage_pmu;
128
129     /* Input current from alternator 0 - 10 (uint8_t max 255) */
130     uint8_t input_current_altr_pmu;
```

```
131
132
133     /*********************************************************
134      *              FOURTH DISPLAY PAGE for debug               *
135      *********************************************************/
136     /* Tyre pressure values for each wheel between 0 - 30 bar (uint8_t max 255)*/
137     uint8_t tyre_pressure[4];
138
139
140     /*********************************************************
141      *            EXTRA - EMULATION PURPOSES                 *
142      *********************************************************/
143     /* Store the time from the RTC of the car */
144     /* time in seconds */
145     uint8_t time_second;
146
147     /* time in minutes */
148     uint8_t time_minute;
149
150     /* time in hours */
151     uint8_t time_hour;
152
153 } display_data_t ;
154
155
156 #endif /* COMMON_H */
```

components/heartbeat/inc/heartbeat.h

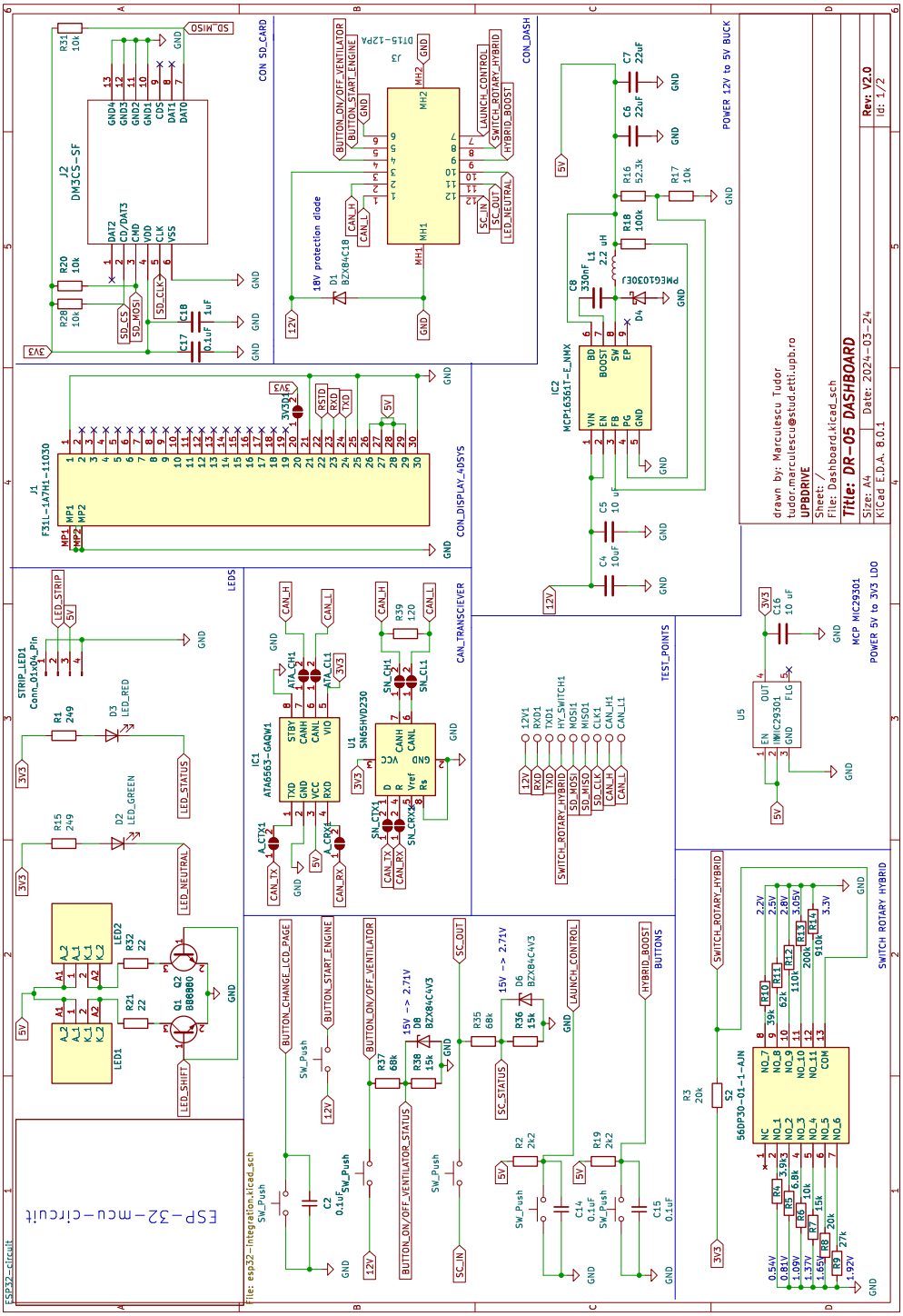
```
1  /*****************************************************************************/
2  /* UPBDRIVE ECU-Emulator Firmware                                            */
3  /* All rights reserverd 2023                                                 */
4  /*****************************************************************************/
5
6  #ifndef HEARTBEAT_H
7  #define HEARTBEAT_H
8
9  #include "common.h"
10 #include "pico/stdlib.h"
11 #include "hardware/gpio.h"
12 #include "hardware/adc.h"
13
14 ////////////////////////////////////////////////////////////////////////////
15 ////                        GLOBAL VARIABLES                            ////
16 ////////////////////////////////////////////////////////////////////////////
17
18 ////////////////////////////////////////////////////////////////////////////
19 ////                        GLOBAL MACROS                               ////
20 ////////////////////////////////////////////////////////////////////////////
21
22 ////////////////////////////////////////////////////////////////////////////
23 ////                        GLOBAL CONSTANTS                            ////
24 ////////////////////////////////////////////////////////////////////////////
25
26 ////////////////////////////////////////////////////////////////////////////
27 ////                        FUNCTION PROTOTYPES                         ////
28 ////////////////////////////////////////////////////////////////////////////
29
30 void init_heartbeat(uint8_t* heartbeat_state);
31
32 void toggle_heartbeat(uint8_t *heartbeat_state);
33
34
35 #endif /* HEARTBEAT_H */
```

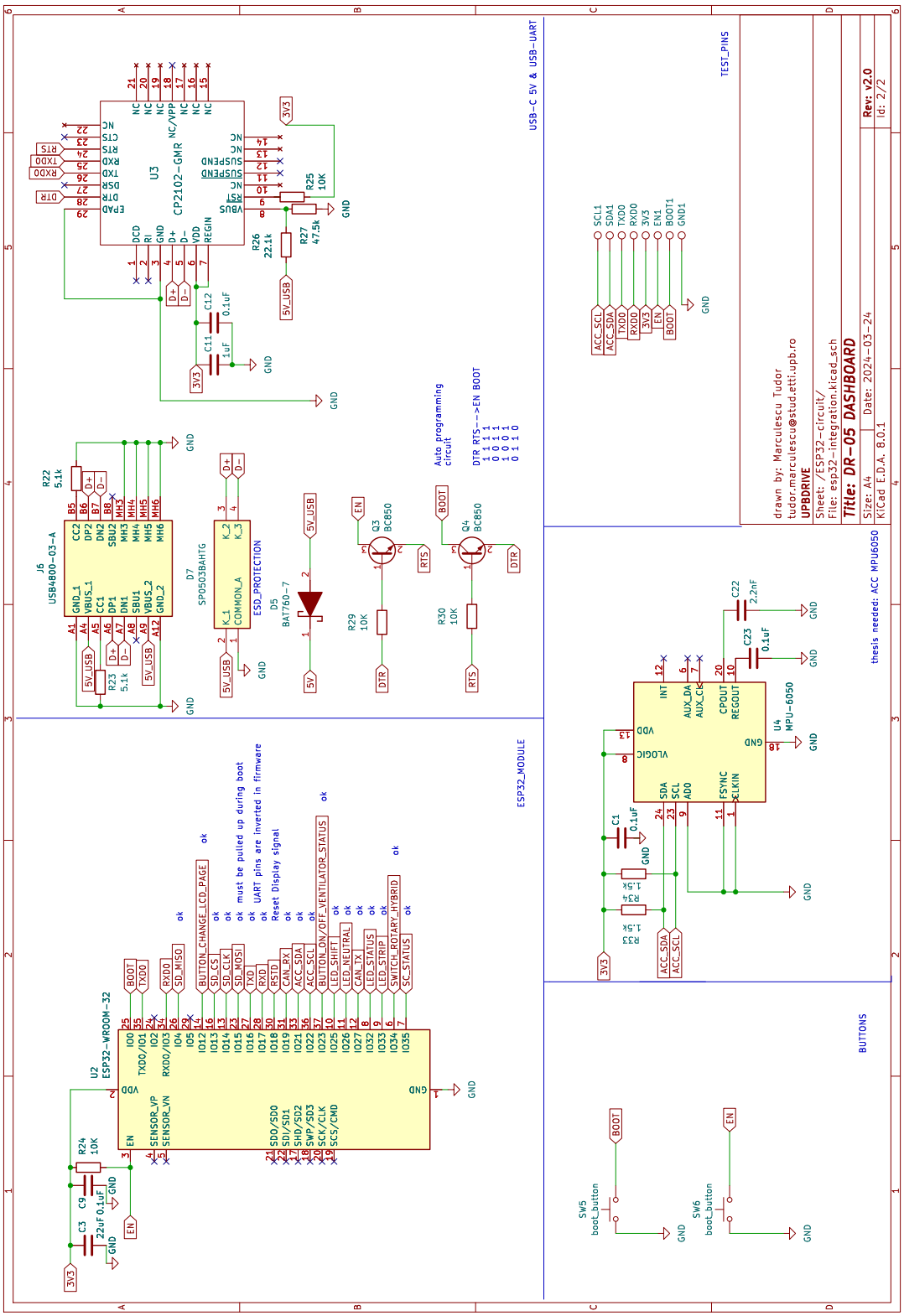
components/heartbeat/heartbeat.cpp

```
1  /*****************************************************************************/
2  /* UPBDRIVE ECU-Emulator Firmware                                            */
3  /* All rights reserverd 2023                                                 */
4  /*****************************************************************************/
5
6  #include "heartbeat.h"
7
8  ////////////////////////////////////////////////////////////////////////////
9  ////                        GLOBAL VARIABLES                            ////
10 ////////////////////////////////////////////////////////////////////////////
```

```
11
12 ////////////////////////////////////////////////////////////////////////////
13 ////                        GLOBAL FUNCTIONS                            ////
14 ////////////////////////////////////////////////////////////////////////////
15
16 ////////////////////////////////////////////////////////////////////////////
17 ////                        LOCAL MACROS                                ////
18 ////////////////////////////////////////////////////////////////////////////
19
20 ////////////////////////////////////////////////////////////////////////////
21 ////                        LOCAL CONSTANTS                             ////
22 ////////////////////////////////////////////////////////////////////////////
23
24 void init_heartbeat(uint8_t* heartbeat_state)
25 {
26     /* initialize the heartbeat led pin */
27     gpio_init(GPIO_NUM_HEARTBEAT);
28     gpio_set_dir(GPIO_NUM_HEARTBEAT, GPIO_OUT);
29
30     /* set the initial state of the heartbeat_state variable */
31     gpio_put(GPIO_NUM_HEARTBEAT, heartbeat_state);
32 }
33
34 void toggle_heartbeat(uint8_t *heartbeat_state)
35 {
36     /* invert the heartbeat_state value such that the LED
37        will have an ON-OFF blink */
38     *heartbeat_state = !(*heartbeat_state);
39
40     /* change the state of the LED based on the heartbeat_state value */
41     gpio_put(GPIO_NUM_HEARTBEAT, *heartbeat_state);
42 }
```

Annex C
Dashboard electronic module schematic




USB-C 5V & USB-UART
ESP32_MODULE
BUTTONS
TEST_PINS
U2
ESP32-WROOM-32
U3
CP2102-GMR
U4
MPU-6050
J6
USB4800-03-A
D7
SP0503BAHTG
D5
BAT760-7
Q3
BC850
Q4
BC850
Auto programming circuit
ESD_PROTECTION
drawn by: Marculescu Tudor
tudor.marculescu@stud.etti.upb.ro
UPBDRIVE
Sheet: /ESP32-circuit/
File: esp32-integration.kicad_sch
Title: DR-05 DASHBOARD
Size: A4
Date: 2024-03-24
Rev: v2.0
KiCad E.D.A. 8.0.1
Id: 2/2
thesis needed: ACC MPU6050

Annex D
Dashboard electronic module PCB layout

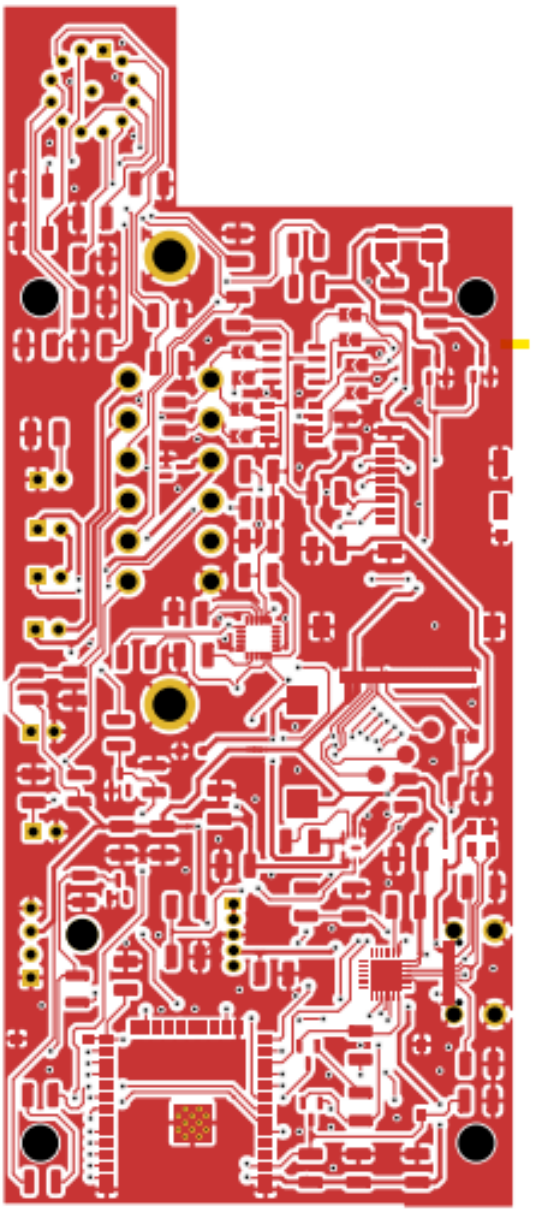


Figure D.1: Front copper layer

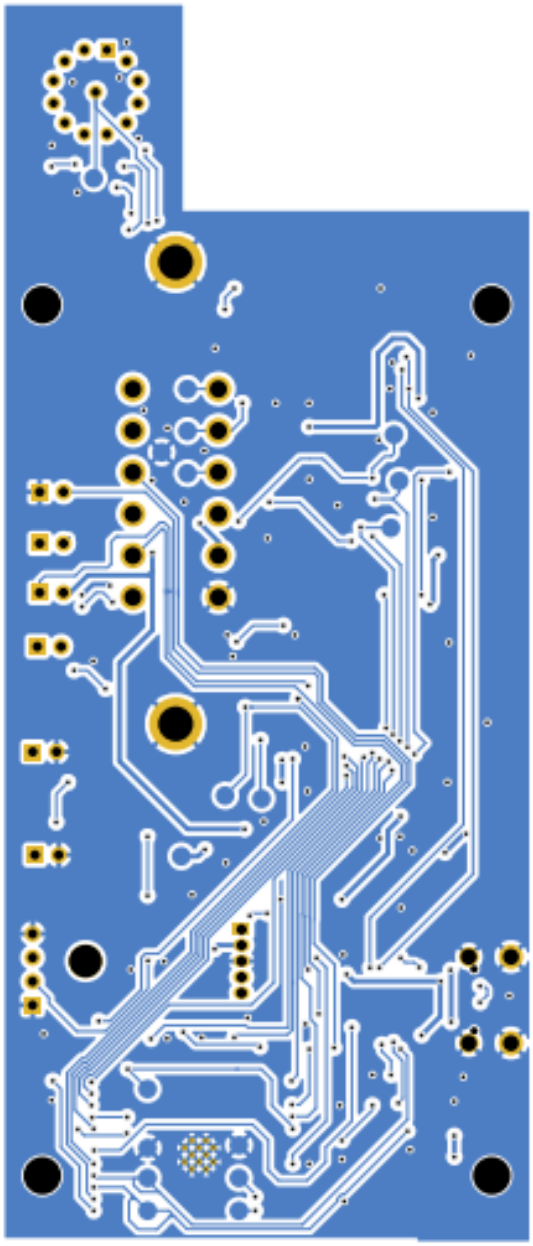


Figure D.2: Back copper layer

Annex E
Dashboard electronic module 3D view

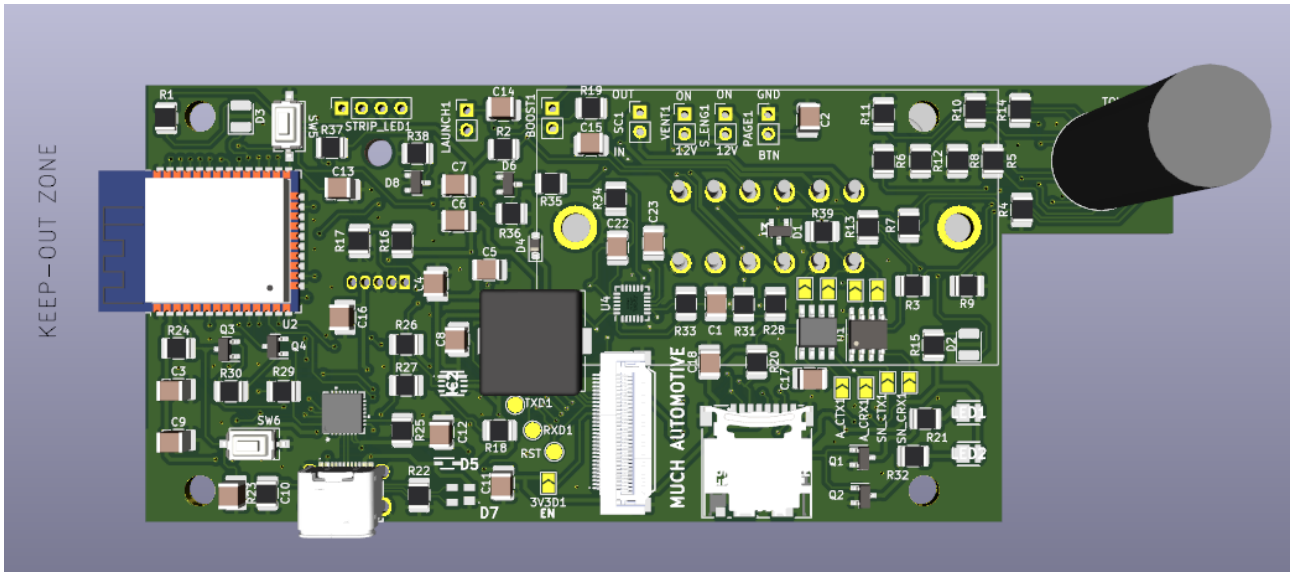


Figure E.1: Front view

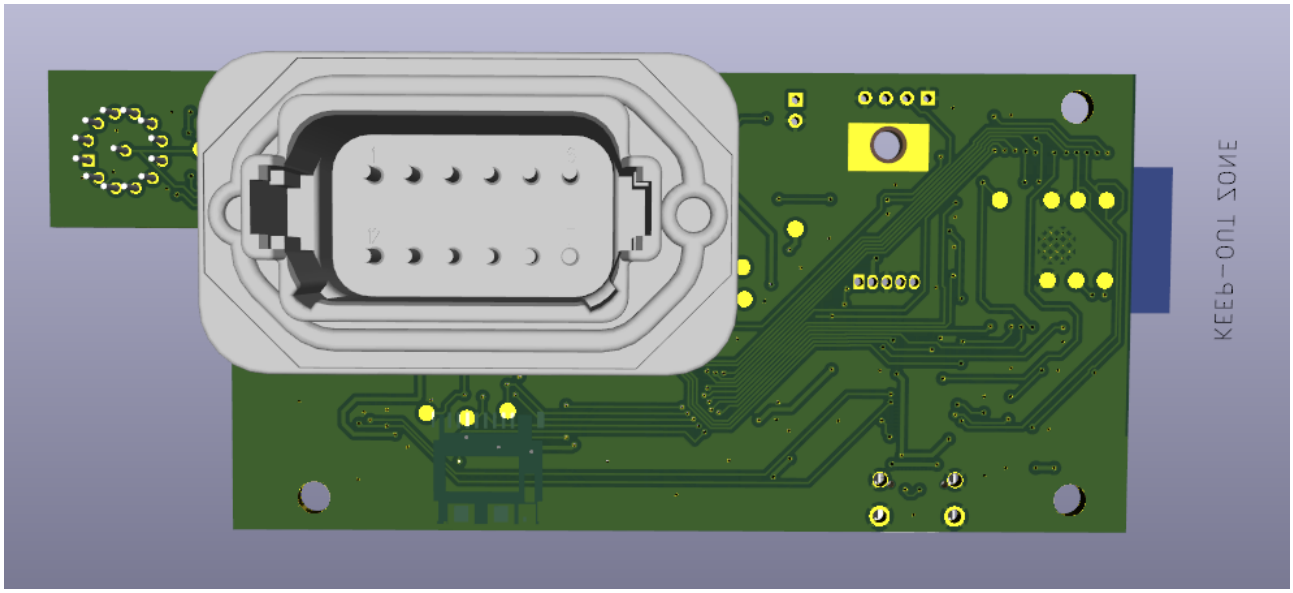
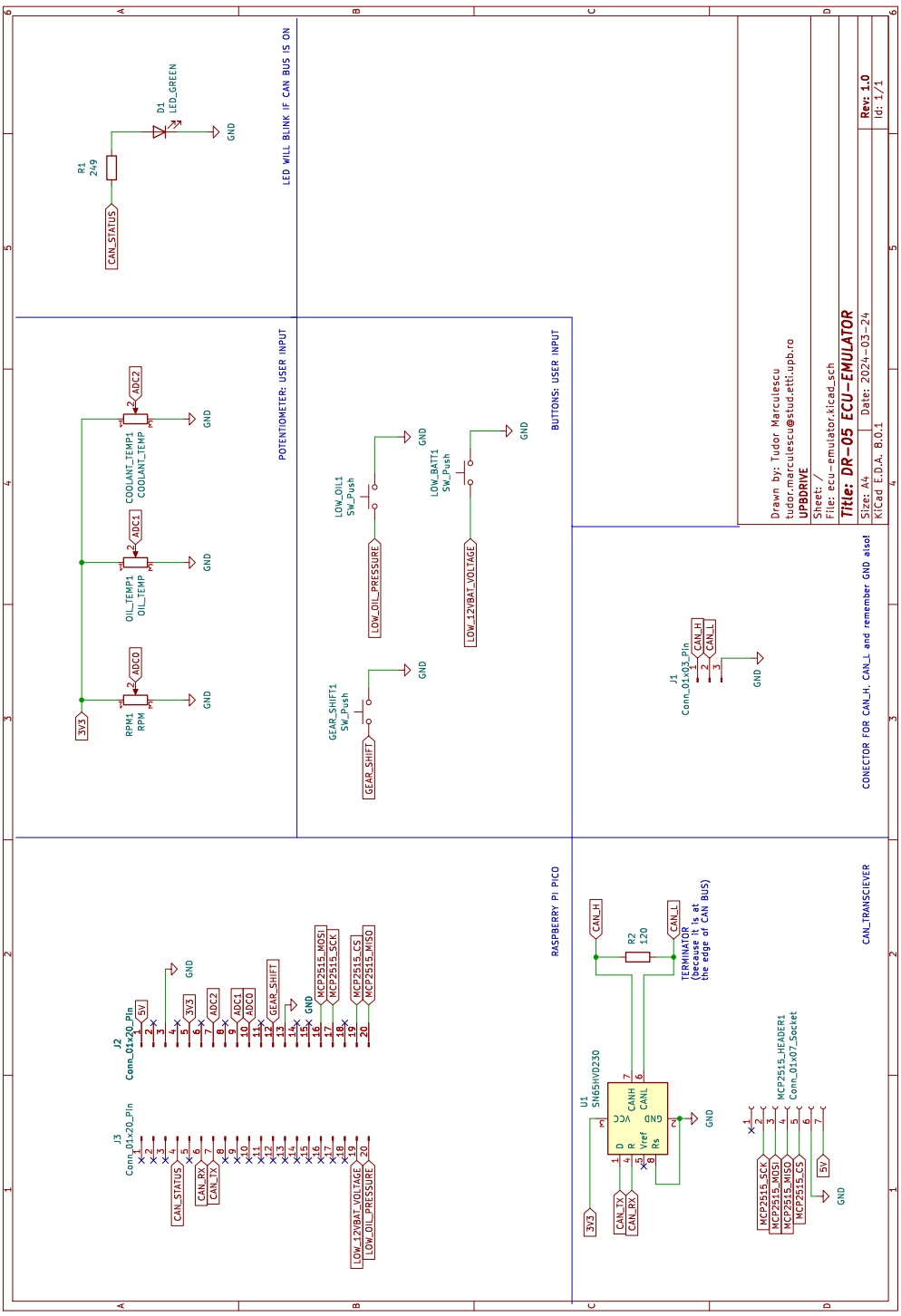


Figure E.2: Back view

Annex F
Testing module schematic



Annex G
Testing module PCB layout

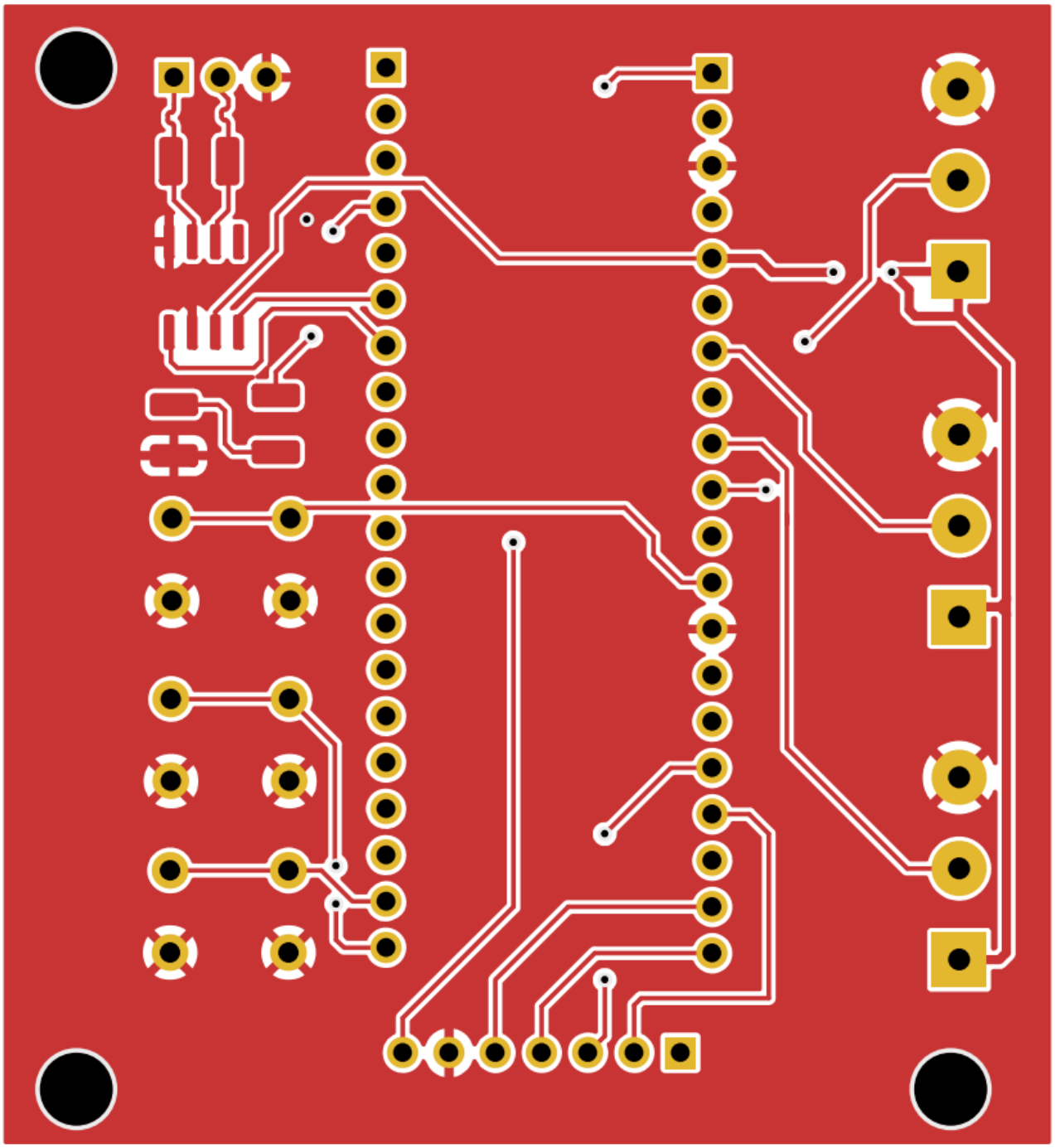


Figure G.1: Front copper layer

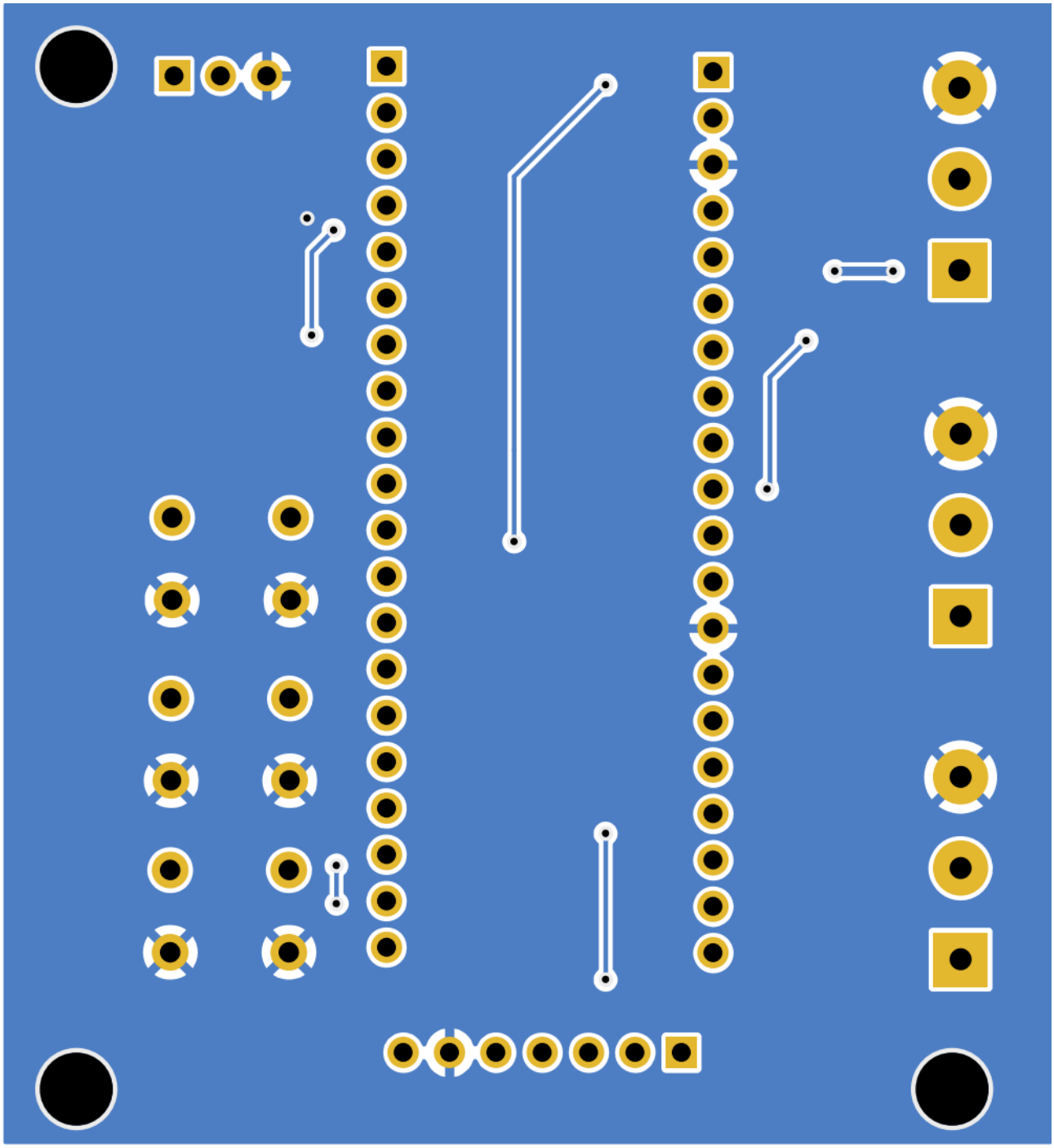


Figure G.2: Back copper layer

Annex H
Testing module 3D view

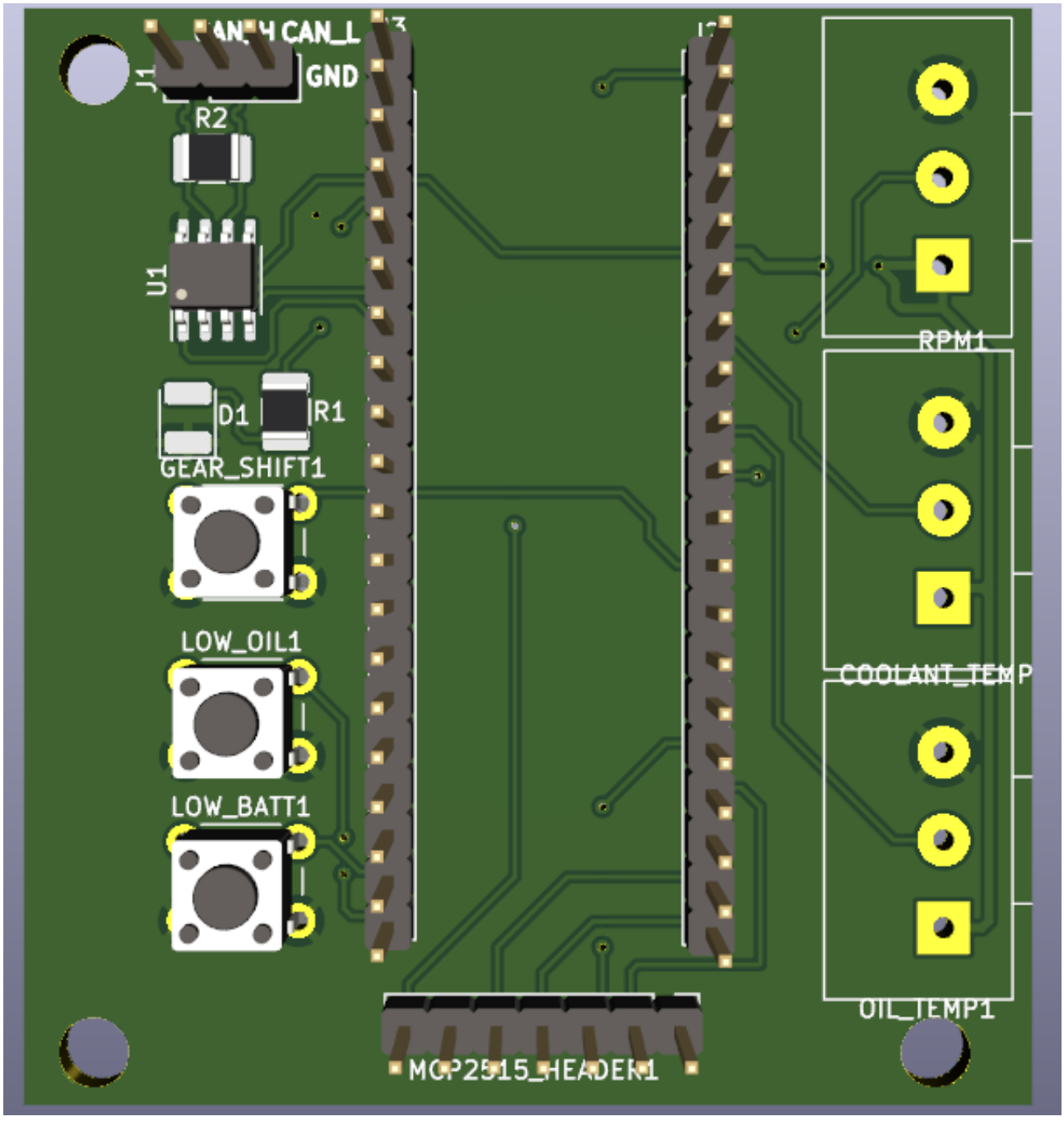


Figure H.1: Front view

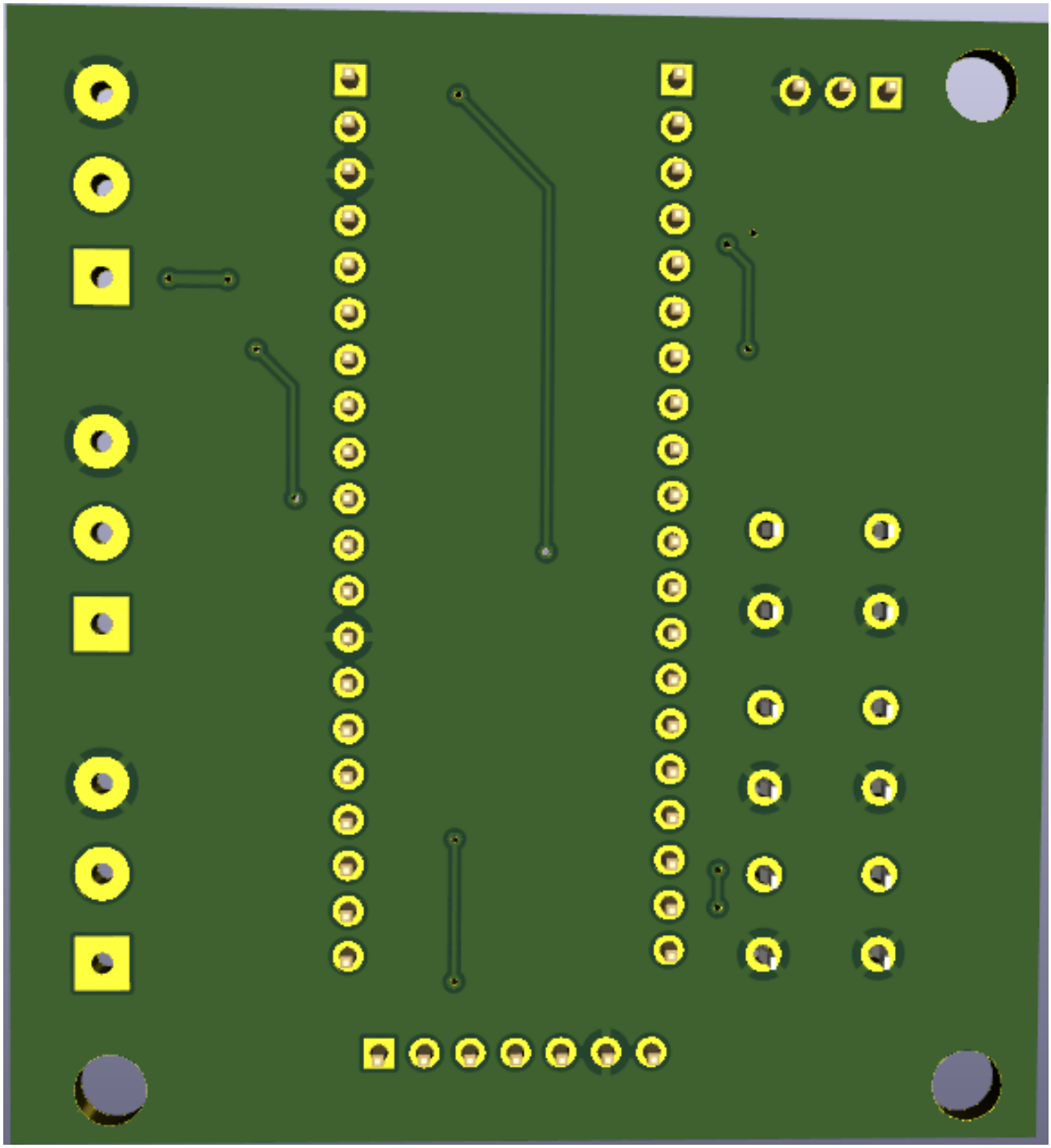


Figure H.2: Back view

Annex I
WEB interface of the dashboard module

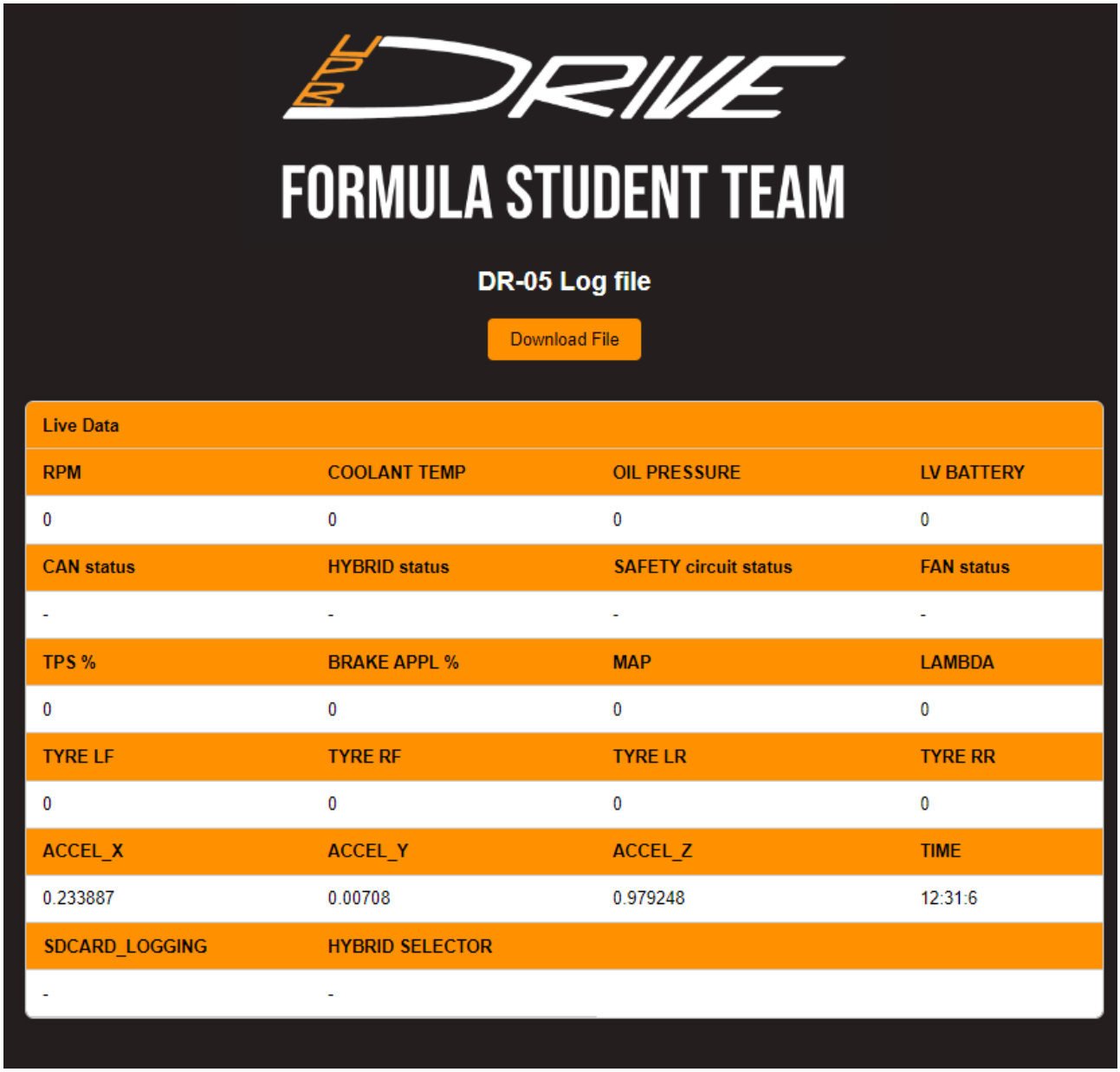


Figure I.1: Web interface hosted on the ESP32 microcontroller