**CarMonitor - Sistem de monitorizare de la distanță al parametrilor autovehiculelor**

**Candidat: Paul-Șerban Horvath**

**Coordonator științific: Conf. Dr.Ing. Lucian Prodan**

Sesiunea: Iunie 2024

Contents

[Rezumat 4](#_Toc167623310)

[1. INTRODUCERE 5](#_Toc167623311)

[1.1 Aspecte generale 5](#_Toc167623312)

[2. SOTA - State of the art, studiu de piață 6](#_Toc167623313)

[2.1 ESP-32 7](#_Toc167623314)

[3. ARHITECTURA 8](#_Toc167623315)

[3.1 ESP-32 8](#_Toc167623316)

[3.2 Diagrama de decizii generica 9](#_Toc167623317)

[3.3 Arhitectura Bluetooth 11](#_Toc167623318)

[3.3.1 Diagrama de decizii 13](#_Toc167623319)

[3.4 Arhitectura Web Server-ului 13](#_Toc167623320)

[3.4.1 Diagrama de decizii 14](#_Toc167623321)

[3.5 Arhitectura sistemului 15](#_Toc167623322)

[3.5.1 Diagrama 16](#_Toc167623323)

[3.6 SPIFFS 17](#_Toc167623324)

[4. IMPLEMENTARE DETALIATĂ 18](#_Toc167623325)

[4.1 Implementare Bluetooth 18](#_Toc167623326)

[4.1.1 Diagrama de secvență dinamică 20](#_Toc167623327)

[4.1.2 Interogarea unui parametru (rpm) prin API-ul ELMduino 22](#_Toc167623328)

[4.1.3 Diagrama de secvență pentru CarMonitor - Se adauga si detalii de returnari, etc. 25](#_Toc167623329)

[4.2 Implementare WebAsyncServer 27](#_Toc167623330)

[4.2.1 Diagrama de secvență dinamică 28](#_Toc167623331)

[4.2.2 Diagrama de secvență pentru WebServer.begin() 30](#_Toc167623332)

[4.2.3 Diagrama de secvență pentru handleSensorData() 32](#_Toc167623333)

[4.3 Interfața cu utilizatorul 34](#_Toc167623334)

[4.3.1 Aspecte generale 34](#_Toc167623335)

[4.3.2 Fișierul HTML (HyperText Markup Language) 34](#_Toc167623336)

[4.3.3 Fișierul CSS (Cascading Style Sheets) 38](#_Toc167623337)

[4.3.4 Fișierul JS (JavaScript) 40](#_Toc167623338)

[4.4 Interfața vizuală 45](#_Toc167623339)

[5. DEZVOLTARE SI TESTARE 46](#_Toc167623340)

[5.1 Testare și configurare 46](#_Toc167623341)

[5.2 Planul de afacere(vezi GreenSpot) 46](#_Toc167623342)

[6. Concluzie 46](#_Toc167623343)

[6.1 Obiective îndeplinite 46](#_Toc167623344)

[6.2 Impedimente 47](#_Toc167623345)

# Rezumat

Lucrarea de față are ca scop utilizarea unui sistem alcătuit dintr-o placă dedicată proiectelor de tip IoT (Internet of Things), și anume ESP-32, placă furnizată de către compania Espressif, folosind framework-ul Espressif IoT Development Framework (esp-idf). Aceasta are rolul de a fi un nod care se ocupă de preluarea datelor în timp real de la un autoturism, aflat în staționare sau în mers, prin intermediul unui convertor OBD (On board diagnostics) versiunea II, dar și de găzduirea unui server web, cu scopul de a afișa parametrii acestui vehiculului în timp real. Placa ESP-32 este programată într-o manieră în care să poată trimite cereri de citire al parametrilor (turația motorului, viteza instantă, consum, temperatura uleiului, consumul instant de carburant, temperatura lichidului de răcire și presiunea din galeria de admisie) către un convertor conectat la portul OBD-II al vehiculului.

Convertorul, sau cititorul OBD-II, este furnizat de compania VGate, și este echipat cu un chip de tip ELM327, care permite transferul datelor primite de la ECU (Electronic Control Unit) al autoturismului prin portul OBD-II către ESP-32 prin Bluetooth. A fost ales acest dispozitiv datorită versatilității și compatibilității crescute cu alte dispozitive. Acest dispozitiv trimite cereri primite de la placa ESP-32 către ECU prin coduri standard numite PID (Parameter ID).

Mai departe, după request-ul primit, ECU va returna valoarea dorită către cititorul OBD-II (dotat cu ELM327), care la rândul lui va trimite valoarea către ESP-32. Acesta va afișa valoarea pe un server web, cu scopul de a afișa parametrii autoturismului într-o maniera user-friendly.

Placa ESP-32 este programată astfel încât să folosească funcții dintr-un API open-source, numit ELMduino, disponibil pe plăcile de tip ESP-32 Development board. Acest API are rolul de a forma și identifica PID-urile corespunzătoare, și de a le trimite prin Bluetooth către cititorul OBD-II. Pe de alta parte, este necesar ca placa sa fie conectată la o rețea pentru a putea găzdui cu succes serverul cu scopul afișării datelor. Această placă a fost aleasă datorită capabilităților WiFi si Bluetooth pe care le prezintă.

# INTRODUCERE

## **Aspecte generale**

Într-o lume în care informația înseamnă putere, fiecare dintre noi dorim să facem alegeri cât mai bune, bazate pe informațiile care ni se prezintă. Autovehiculele, pe de altă parte, cu toate că sunt folosite intens de fiecare dintre noi, pe întreg Pământul, sunt construite într-o manieră în care sa furnizeze doar strictul necesar de informații pe care o persoană obișnuită le poate procesa. Astfel, pentru o persoană cunoscătoare, fie că pentru ea este un hobby sau este ocupația acesteia, simplele date furnizate de către un autovehicul în bord pot să nu fie îndeajuns.

Istoria diagnozei mașinilor datează de la începutul secolului XX, când autovehiculele încă erau relativ simple, iar sistemele de monitorizare electronice abia debutau. Depanarea problemelor se făcea manual, unde mecanicii se bazau pe setul de cunoștințe proprii, necesare pentru interpretarea diferitelor sunete, vibrații, al fumului etc. De aceea, s-a hotărât standardizarea acestora de către Societatea Inginerilor Automotive (Society of Automotive Engineers - SAE) în vederea existenței unui protocol robust și bine stabilit de depanare al erorilor mașinilor.

În 1968, Volkswagen AG introduce primul computer de board, acesta fiind analog. În anul 1980, General Motors introduce pe modelele Cadillac Eldorado și Seville, sistemul de diagnoză unde erorile (Diagnostic Trouble Codes) sunt afișate în ecranul de climatizare când vehiculul se afla în modul de depanare, iar în 1981 GM introduce sistemul “Computer Command Control” capabil de a comunica cu ECU pentru a iniția o cerere de diagnoză, după cum este descris în referința [1].

În 1988 Societatea Inginerilor Automotive (SAE) recomanda un conector și un set de erori standard și astfel se introduce standardul OBDI care este necesar tuturor mașinilor vândute în California din acest an, așa cum este menționat în [1].

În anul 1994, CARB (California Air Resource Board) încep demersurile pentru o nouă versiune de OBD, numit OBDII. Acesta include recomandările din OBDI făcute de SAE, iar în 1996 devine un standard obligatoriu pentru mașinile vândute în Statele Unite ale Americii, preluat din [1].

În 2001, Uniunea Europeana adopta standardul OBDII, numit EOBD (European OBD) care întâi este necesar mașinilor cu combustibil pe benzina, iar în 2004 EOBD se introduce și pe mașinile cu combustibil pe motorină, preluat din [1].

Un PID este un șir de bytes, unde fiecare byte are o însemnătate, unde mai departe acest șir este folosit pentru a face cereri către ECU al unui vehicul. De exemplu, pentru citirea vitezei actuale ale vehiculului vom trimite 010D. Citirea oricărui parametru se face prin byte-ul 01 (frecvent utilizat în această lucrare), urmat de parametrul dorit (în acest caz 0D). Pentru turația motorului avem PID-ul 0C, pentru consumul de combustibil avem 5E, pentru temperatura uleiului între 01 si 1F, depinzând de producătorul vehiculului, pentru temperatura lichidului de răcire avem 05 iar pentru presiunea galeriei de admisie se va referi prin valoarea 0B.

Standardul SAE J1979 defineste o varietate de PID-uri, iar standardul SAE J1962 presupune ca aceste PID-uri să fie accesate printr-un conector universal (OBD), așa cum este descris în [1].

Conform aceleiași referințe, un avantaj principal al acestui sistem este că sistemele OBD oferă proprietarului vehiculului sau tehnicianului de reparații acces la starea diferitelor subsisteme ale vehiculului. Cantitatea de informații de diagnostic disponibile prin OBD a variat considerabil de la introducerea sa în anii 1980, odată cu primele versiunile ale computerelor de bord ale vehiculelor. Primele versiuni de OBD doar aprindeau un bec de avertizare (Malfunction indicator light - MIL) sau „lumină de idiot” (“idiot light”) dacă era detectată o problemă, dar nu ofereau nicio informație despre natura problemei. Implementările moderne de OBD folosesc un port de comunicații digitalizat standardizat pentru a furniza date în timp real, în plus față de o serie standardizată de coduri de probleme de diagnostic, sau DTC-uri, care permit unei persoane să identifice și să remedieze rapid defecțiunile din vehicul.

În cadrul SAE s-au definit două tipuri de conectoare OBD-II, mai exact cel de tip A și cel de tip B. Acestea sunt asemănătoare, unde firele și pinii sunt așezați exact la fel, dar diferența constă în suportul din mijloc. În cazul OBD-II tip A, suportul dintre port-urile OBD este dintr-o singură bucată, iar în cazul OBD-II tip B, suportul este divizat în două bucăți. Acest fapt permite identificarea celor două tipuri, deoarece OBD-II tip A este specific autovehiculelor rutiere de uz comun, iar OBD-II tip B este specific autoutilitarelor sau vehiculelor de mare tonaj, specifice industriei. Diferențele dintre cele două tipuri se pot observa mai jos:

A group of colored squares

Description automatically generated

OBD-II tip A – vedere frontala [1]

A group of colored squares

Description automatically generated

OBD-II tip B – vedere frontală [1]

În [1] se menționează că specificația OBD-II prevede o interfață hardware standardizată: conectorul J1962 mamă cu 16 pini (2x8), unde tipul A este utilizat pentru vehiculele de 12 volți și tipul B pentru vehiculele de 24 volți. Spre deosebire de conectorul OBD-I, care se găsea uneori sub capota vehiculului, conectorul OBD-II trebuie să fie la 2 picioare (0,61m) de volan (cu excepția cazului în care producătorul solicită o derogare, în caz în care este încă undeva la îndemâna șoferului). De aceea, conectorul J1962 specific are următoarea configurație a pinilor:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | **La discreția producătorului**  Audi/VW: comunicare status contact  Mercedes: linie K pentru statusul climatizării | 9 | **La discreția producătorului**  BMW: comunicare RPM  Mercedes: Linie K – diagnoză ABS |
| 2 | **Linia pozitivă a magistralei** | 10 | **Linia negativă a magistralei** |
| 3 | **La discreția producătorului**  Ethernet Tx+ (Diagnoză prin IP) | 11 | **La discreția producătorului**  Activarea ETH (Diagnoză prin IP)  Mercedes: Linie K pentru cutia de viteze |
| 4 | **Împământarea șasiului** | 12 | **La discreția producătorului**  Activarea ETH (Diagnoză prin IP)  Mercedes: Linie K pentru modulele active (ex. radio) |
| 5 | **Împământarea semnalului** | 13 | **La discreția producătorului**  Activarea ETH (Diagnoză prin IP)  Mercedes: Linie K pentru sistemele de siguranță |
| 6 | **CAN HIGH**  (ISO 15765-4 și SAE J2284) | 14 | **CAN LOW**  (ISO 15765-4 și SAE J2284) |
| 7 | **Linia K**  (ISO 9141-2 și ISO 14230-4) | 15 | **Linia L**  (ISO 9141-2 și ISO 14230-4) |
| 8 | **La discreția producătorului**  Activarea ETH (Diagnoză prin IP)  BMW: a doua linie K pentru Infotainment)  Mercedes: Aprindere | 16 | **Tensiunea bateriei**  +12V pentru tip A  +24V pentru tip B |

Tabel preluat din [1]

OBD-II este un standard introdus în Statele Unite ale Americii pe piața auto din aceasta, iar echivalentul ei în Europa este EOBD (European on-board diagnostics) și se aplică tuturor vehiculelor de pasageri de categoria M1 (cu cel mult 8 locuri pentru pasageri și o Greutate Brută a Vehiculului de 2500kg sau mai puțin) înmatriculată pentru prima dată în statele membre UE de la 1 ianuarie 2001 pentru mașinile cu motor pe benzină (pe benzină) și de la 1 ianuarie 2004 pentru mașinile cu motor diesel, așa cum este descris în [1]. De asemenea, în aceeași referința se descrie că pentru autoturismele cu greutatea nominală a vehiculului mai mare de 2500kg și pentru vehiculele utilitare ușoare, regulamentul se aplică de la 1 ianuarie 2002, pentru modelele pe benzină, și 1 ianuarie 2007, pentru modelele diesel.

Implementarea tehnică a EOBD este în principiu aceeași cu OBD-II, în sensul în care se folosește același conector de legătura utilizat la diagnoză prin SAE J1962 și protocoalele de semnal aferente.

# SOTA - State of the art, studiu de piață

Scopul acestui subcapitol este de a prezenta diferitele tehnologii care există în prezent pe partea de monitorizare al autovehiculelor și de a avea o vedere de ansamblu asupra altor dispozitive care se asociază sistemului „Car Monitor”.

În prezent, există numeroase metode și tehnologii de a monitoriza parametrii unui vehicul, toate având scopuri ușor diferite unele față de altele. Pe de-o parte există produsele comerciale care au ca grup țintă mecanicii profesioniști din service-urile autorizate. Aceștia în mod uzual folosesc produse și tehnologii mai scumpe (și deseori mai inaccesibile publicului larg) care permit conectarea la portul OBDII în mod direct prin fir. Scopul acestora este să identifice toate erorile mașinii care au apărut de-a lungul timpului și să ajute mecanicul să ia o decizie informată ținând cont de acestea. Aceste erori se pot șterge din memoria calculatorului de bord, prin resetarea stării regiștrilor care stochează erorile respective. Aici, cele mai folosite sunt dispozitivele produse de Autel, SnapOn Diagnostic Tool sau XTOOL.

Pe de altă parte, pe nișa clienților de uz general, există produse care sunt mai accesibile din punct de vedere al prețului, dar precizia datelor returnate (date care se modifică în timp real) este mai slabă. Conectivitatea wireless apare la această categorie deoarece elimină nevoia de un hardware fizic care să asigure transmisia datelor. De aceea, datele sunt transmise OTA (over the air) prin intermediul Bluetooth către un receptor, în care în cele mai multe din cazuri este un smartphone capabil sa afișeze într-o aplicație instalata. În cazurile care seamănă cu proiecte DIY (Do It Yourself), există dispozitive care au încorporat un display LCD pe care se face afișajul precum și senzori plasați în zonele de interes ale mașinii.

În continuare se prezintă un tabel în care se realizează o comparație între sistemele de monitorizare profesionale, sistemele destinate publicului larg care se folosesc de o aplicație creată special în acest sens și sistemele de monitorizare de tip DYI.

Tabel 1: Comparație între metodele actuale de monitorizare

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Cost | Dificultate utilizare | Scalabilitate | Fiabilitatea datelor | Functionalități |
| Scanner OBDII de bază | Scăzut | Ușor (prin aplicație) | Mică | Mică | Diagnoză de bază |
| Scanner OBDII avansat | Mare | Greu | Mică | Mare | Diagnoză avansată |
| Sistem DYI | Mediu | Mediu | Mică | Medie | Monitorizarea depinde de senzorii plasați de utilizator |
| Car Monitor | Scăzut | Mediu | Mare | Medie | Monitorizarea parametrilor în timp real (disponibili în API) + diagnoză de bază al DTC |

## ESP-32

# ARHITECTURA

## ESP-32

ESP-32 este o familie de plăci low-cost, de putere redusă, dezvoltat de către firma Espressif Systems, companie din China bazată în Shanghai. Acesta vine ca un succesor al microcontroller-ului ESP-8266, care introduce modulul Bluetooth, un modul WiFi mai puternic, frecvența de operare mai mare, mai multă memorie, etc.

A diagram of a computer chip

Description automatically generated

*Imagine preluată din [2]*

PCB-ul folosit care are System on Chip (SoC) de tip ESP-32 este unul ESP32-WROOM-32, care are 4 MiB memorie flash, alimentat cu 3.3V curent continuu (DC) și este bazat pe chip-ul ESP32-D0WDQ6. Acesta are o frecventa a procesorului de 240 MHz, 34 pini de uz general, iar dimensiunile sunt de 6mm x 6mm, precum este menționat în [3]. ESP32 folosește framework-ul NodeMCU, care este un framework high-level des întâlnit în proiectele de tip IoT. NodeMCU are avantajul faptului că este mai ușor de folosit, dar prezintă performanțe mai reduse. Totuși, are o comunitate largă și există numeroase librării care fac lucrul cu senzori să fie cât mai facil. Deci acest framework este în principal folosit pentru proiectele de dificultate medie.

De asemenea, pentru operațiile low-level și pentru proiectele de dificultate mare, este disponibil framerwork-ul ESP IoT Development Framework, sau ESP-IDF. Acesta este programabil în C/C++, suită de limbaje care oferă performanțe mari și care permit accesul la componente hardware low-level. ESP-IDF suportă configurarea componentelor precum cele WiFi, Bluetooth, GPIO, SPI, CAN, ADC (12 biți) etc., după cum este prezentat în [3]. De aceea, în lucrarea de față, s-a utilizat o combinație intre cele două framework-uri.

Această placă introduce, după cum s-a menționat anterior, atât un modul Bluetooth, cât și un modul WiFi, iar pentru programarea ei s-a folosit Arduino Framework, prin extensia PlatformIO disponibilă în Visual Studio Code. Aceste module folosesc aceeași antenă, iar aplicațiile care necesită utilizarea lor trebuie să gestioneze, prin software sau hardware, accesul la antenă. În cazul de față, s-a ales o gestionare software, prezentă în ESP-IDF, datorită simplității și versatilității pe care această soluție o prezintă.

## Diagrama de decizii generica

Întreg codul proiectului este bazat pe un macro care decide use case-ul pe care il va urma codul. Directiva #if (SERVER\_TESTING == true) decide dacă se va testa doar interfața și se vor afișa valori fictive, sau dacă se dorește use case-ul normal de execuție care include activarea WiFi, Bluetooth, iar valorile vor fi cele reale returnate de cititorul OBD montat în vehicul.

A diagram of a car monitor

Description automatically generated

Diagrama [1]: Diagrama de decizii generică a codului.

A diagram of a software system

Description automatically generated

Diagrama [2]: Diagrama de decizii pentru funcția setup()

A diagram of a computer program

Description automatically generated

Diagrama [3]: Diagrama de decizii generică pentru funcția loop()

## Arhitectura Bluetooth

Modulul Bluetooth prezent pe placa ESP-32 are versiunea 4.2 BR/EDR, ceea ce înseamnă ca poate comunica cu dispozitive precum tastaturi, mouse-uri, telefoane, inclusiv cititoare OBDII cu chip ELM327. Pentru a-l folosi, va trebui inclus fișierul “Bluetooth.h”.

Există două standarde Bluetooth disponibile pe ESP-32: Bluetooth Classic și Bluetooth LE. Bluetooth Classic permite ca placa sa poată juca simultan atât rolul de Bluetooth client cât și de server, iar ca și profile suportă SPP (Serial Port Protocol), folosit pentru comunicarea serială prin Bluetooth. În general acest standard este folosit de aplicațiile care au un trafic mare, sau continuu de date schimbate între client și server, ceea ce îl face potrivit pentru acest proiect, după cum se menționează în [4].

Pentru Bluetooth LE (Low Energy), așa cum este descris în [4], ESP-32 suportă servicii și profile precum GATT (Generic Attribute Profile), folosit atunci când conexiunea este deja realizată pentru a manevra și schimba datele, și GAP (Generic Access Profile) folosit pentru a seta parametrii de conectare între dispozitive. Standardul LE pentru Bluetooth este folosit în situațiile în care consumul de energie este prioritar, iar datele transmise între dispozitive sunt puține.

În cazul sistemului “Car Monitor”, protocolul Bluetooth folosit este cel Classic, deoarece există un flux constant de date care trebuie procesate, în urma comunicării plăcii ESP-32 cu cititorul OBDII. Placa trimite prin Bluetooth PID-uri în urma cărora OBD-ul răspunde cu datele cerute. De asemenea, la începutul comunicării se activează și se setează parametrii de comunicare, trimiși sub forma de bytes.

Manipularea protocolului Bluetooth se face sub o directivă care activează sau dezactivează codul respectiv, în funcție de use case-ul dorit. La liniile unde se întâlnește directiva #if (SERVER\_TESTING == true) se împarte codul în use case-ul în care se dorește doar testarea interfeței, iar datele afișate sunt imitări ale valorilor reale, sau use case-ul în care se dorește firul normal al execuției. Macro-ul SERVER\_TESTING poate fi activat sau dezactivat prin true sau false în fisierul src/config/config.h.

Activarea modulului Bluetooth are la bază afirmația că macro-ul SERVER\_TESTING este pus pe *false*. În acest caz, are loc includerea header-ului Bluetooth.h, care cuprinde toate funcționalitățile de care este nevoie pentru a lucra cu acest modul.

După aceasta, pentru a se putea folosi de metodele puse la dispoziție de acest header, trebuie declarat un obiect de tipul *BluetoothSerial.* Prin intermediul acestuia se vor face operațiile prin Bluetooth, de care se va folosi și API-ul ELMduino. Se realizează activarea Bluetooth, și urmează o secvență de două clauze de tip if care testează dacă conexiunea la cititorul OBD și activarea lui au fost realizate cu succes. Dacă la ambele clauze există un răspuns afirmativ, activarea Bluetooth a fost realizată cu succes. Dacă la oricare clauze răspunsul este unul negativ, activarea Bluetooth a eșuat. Acest comportament este vizibil în diagrama de decizii de la 2.2.1.

### Diagrama de decizii

A diagram of a bluetooth software

Description automatically generated

## Arhitectura Web Server-ului

Modulul integrat WiFi folosește portul 80 (HTTP) și suportă protocoalele IEEE 802.11 b/g/n, unde viteza de transmisie poate ajunge la 150Mbps, iar lățimea de bandă este de 2.4GHz, fiind cea mai utilizata lățime de bandă pentru WiFi la nivel global. Capabilitățile WiFi ale ESP-32 sunt puse la dispoziție de către ESP-IDF, care asigura diferite API-uri pentru lucrul cu acest modul. De asemenea, puterea de transmitere este de pana la +20 dBm, care poate fi ajustată în funcție de legislația în vigoare.

Așa cum se prezinta în [4], modulul WiFi are trei moduri de operare:

* Station mode: ESP-32 se conectează la un router WiFi ca și client.
* SoftAP mode: ESP-32 devine însuși un access point la care alte dispozitive se pot conecta.
* Promiscuous mode: Mod prin care ESP-32 poate asculta pachetele WiFi, mod folositor în cazul în care se dorește un sistem de analiza al rețelei din care acesta face parte.

În cod, modulul WiFi nu depinde de un macro, acesta fiind activat mereu, iar prin intermediul lui, web serverul este activ la fiecare rulare. Includerea metodelor WiFi se fac prin fișierul <WebServer.h>. De asemenea, în realizarea sistemului, placa a fost conectată la un hotspot mobil al unui smartphone. De aceea, încă de pe acum se poate anticipa o metodă de scalare tehnică al acestui proiect, prin adăugarea unui modul SIM care să găzduiască singur acest web server, și nu prin localhost cum se întâmplă momentan.

Web server-ul este componenta proiectului care se activează și rulează indiferent dacă SERVER\_TESTING este pus pe *true* sau pe *false.* Aceasta reprezintă interfața utilizatorului cu sistemul din spate, supranumit în cod SERVER.

Motivul pentru care serverul nu depinde de macro este reprezentat de faptul că sistemul este gândit sa aibă două use case-uri, unul de testare al interfeței și unul normal. În ambele use case-uri este nevoie de activarea serverului, dar doar în use case-ul de testare nu se activează, spre deosebire, modulul Bluetooth.

### Diagrama de decizii

A diagram of a computer network

Description automatically generated

Acest API este disponibil open-source pe GitHub [6] iar rolul lui este de a îmbunătăți comunicarea pe web server. În spre deosebire de metodele clasice, unde serverul procesează datele într-un mod secvențial, AsyncWebServer reușește să preia cererile și sa le trateze în mod asincron. Din acest lucru rezultă faptul că nu se va bloca firul normal de execuție al programului principal, nu vor apărea întârzieri rezultate din procesarea secvențială a datelor, astfel îmbunătățind receptivitatea la modificările din procesele din spate.

În testare s-a încercat rularea web serverului cu el activat dar și dezactivat, și se observa o îmbunătățire clară atunci când este introdus în fișierul de build disponibil în PlatformIO. Din punct de vedere al scalabilității, sistemul face uz de toate resursele plăcii într-un mod cât mai eficient. De aceea, această dependința este foarte importantă și este potrivită pentru sistemele IoT, inclusiv pentru sistemul “Car Monitor”.

## Arhitectura sistemului

Având descrise cele două module care compun sistemul, se poate realiza arhitectura întregului sistem prin imbricarea celor două. Ordinea în care cele două se rulează este întâi configurarea serverului, apoi configurarea Bluetooth. Ulterior, în funcție de macro, se selectează use case-ul dorit.

După ce aceste două module au fost inițializate, poate începe rularea efectivă a programului. Programul rulează funcția loop() după ce funcția setup() s-a realizat cu succes. Deoarece codul este bazat pe framework-ul Arduino, funcția loop() este cea în care codul efectiv se prezintă, iar aceasta este rulată ciclic, de la terminarea funcției setup() până când placă intalneste semnalul de reset (RST) sau alimentarea este oprită.

Așa cum a fost cazul și la funcția setup(), funcția loop depinde de macro-ul SERVER\_TESTING. Dacă acesta este pus pe true, se va rula un cod care exemplifică funcționalitatea interfeței utilizatorului cu sistemul. În acest caz, parametrii afișați vor avea valori care se vor schimba ciclic, dar vor reprezenta valori și evoluție realiste.

În situația în care macro-ul este setat pe false, codul va trece prin diferite stări, pentru a interoga OBD-ul pentru diferiți parametrii. Acesta este implementat astfel încât stările posibile sunt: SPEED (unde se va interoga viteza in kph ai vehicului), ENG\_RPM (turația), VOLTAGE (tensiunea bateriei), THROTTLE (cât % este apăsată pedala de acceleratie) , ENG\_COOLANT (temperature in Celsius al lichidului de răcire), LOAD (cât % este solicitat motorul), FUEL\_LEVEL (cât % mai este combustibil in rezervor), OIL\_TEMP (temperature uleiului in Celsius).

### Diagrama

A screenshot of a diagram

Description automatically generated

A screenshot of a computer

Description automatically generated

## SPIFFS

SPIFFS (Serial Peripheral Interface Flash File System) este un sistem al fișierelor special conceput pentru sistemele embedded care utilizează memoria non-volatilă. Acesta este construit în așa măsură încât să ocupe memorie RAM cât mai puțină.

Sistemul SPIFFS este pus la dispoziție de către ESP-IDF, iar acesta poate deschide, citi, scrie, închide și șterge fișiere, suportă ierarhii de directoare, dar subdirectoarele nu sunt permise, decât dacă se utilizează ‘/’ în componența numelor fișierelor, așa cum se menționează în documentația furnizată de Espressif [5].

Acest sistem a fost folosit în proiect pentru a îmbunătăți interfața pe care o întâlnește utilizatorul atunci când folosește “Car Monitor”. Cu ajutorul al SPIFFS, fișiere statice de tip Javascript, CSS și HTML pot fi introduse și rulate cu succes în web serverul găzduit de către ESP-32. Folosind acest sistem, managementul fișierelor este mai simplu, iar afișarea este cu mult optimizata pentru aplicația de tip IoT care necesită rulare în timp real.

# IMPLEMENTARE DETALIATĂ

Pentru gestionarea versiunilor proiectului s-a utilizat GitHub, printre cele mai importante platforme de versionare. Prin natura lor, sistemele sunt organizate în branch-uri (ramuri) care pornesc dintr-un master branch (ramura principală). În cadrul proiectului, s-au utilizat 3 clase de branch-uri:

- main: branch-ul folosit pentru versiunea finală a proiectului.

- P1 (Phase 1): branch folosit pentru lucrul, configurarea și construirea web serverului, unde informația este afișată în timp real.

- P2 (Phase 2): branch reprezentat de capacitatea plăcii de a se asocia cu succes cu dispozitivul OBD-II prin modulul Bluetooth și de a citi și stoca temporar datele rezultate în urma comunicării cu cititorul OBD.

De asemenea, branch-urile P1 și P2 sunt urmate de indicativul -Bx unde x este un număr începând de la 1, care semnifica branch-ul efectiv din acel Phase. De exemplu, pentru P1-B2 avem Phase 1, Branch 2. Rolul împărțirii branch-urilor de tip Phase este de a organiza în task-uri atomice realizarea proiectului de licență. Branch-urile de tip Phase sunt combinate (merged) cu branch-ul master, atunci când caracteristicile dorite sunt complet implementate și sunt gata de a fi livrate.

## Implementare Bluetooth

Pentru a folosi metodele specifice librăriei Bluetooth, întâi se va crea un obiect de tipul BluetoothSerial. În următorul pas, pentru activarea modulului se va folosi metoda .begin("ArduHUD", true) unde primul parametru reprezintă numele pe care îl va purta placa ESP-32, iar al doilea parametru este o valoare booleană care indică dacă placă este sau nu master în comunicarea Bluetooth.

A computer screen shot of a black screen

Description automatically generatedSpre exemplu, în imaginea următoare se observă în terminalul oferit de PlatformIO procesul de conectare și stabilire al parametrilor de comunicare:

Este important de menționat faptul că mesajele din terminal sunt activate din funcția de inițializare al cititorului OBD. Aceste printuri se pot activa sau dezactiva prin al doilea parametru al funcției begin() din librăria ELMduino, care este reprezentat de o variabilă booleană. Dacă acesta este pe true, se vor printa pașii la care a ajuns funcția de citire (oricare ar fi ea – rpm(), kph() etc.), în caz contrar, dacă variabila este pusă pe false în apelarea funcției, aceste printuri nu se vor afișa.:

A computer error message on a black background

Description automatically generated

La pașii 4 și 5 se observă că are loc conexiunea către cititorul OBD în două faze. În prima fază, ESP-32 se conectează la OBD, care prin construcția lui, are două protocoale de comunicare prin Bluetooth, pentru sistemele de operare iOS sau Android. Protocolul Android este preferat deoarece nu necesită ca ESP-32 să fie certificat, unde cel de iOS are nevoie de certificare MFi (Made for iPhone/iPod/iPad). La pasul 6 se confirma faptul că conexiunea a avut loc cu succes iar cererile se pot emite către OBD.

În captura de ecran de mai sus se observă ca se printează în terminal și faptul că se trimit și se primesc variabile de tip char care au rolul de a inițializa cititorul OBD.

Funcțiile din librăria BluetoothSerial.h folosite în cod sunt cele uzuale pentru pornirea modulului și conectarea plăcii ESP-32 la dispozitivul OBD, care așa cum s-a menționat, va fi cel pentru Android OS (deci dispozitivul operează sub numele „Android-Vlink”). Aceste funcții sunt:

* bool BluetoothSerial::begin(String localName, bool isMaster)

folosit pentru a crea un Bluetooth event și pornire modul.

* bool BluetoothSerial::connect(String remoteName)

folosit pentru a se conecta la un dispozitiv menționat ca parametru.

Librăria ELMduino folosește funcțiile de trimitere/recepție a datelor care sunt disponibile în librăria BluetoothSerial.h după ce se apelează funcții interne de formare a PID-ului pentru interogarea OBD-ului. Cele mai folosite funcții folosite din această librărie în ELMduino sunt:

* int BluetoothSerial::available(void)

folosit pentru a returna numărul de mesaje disponibile în coadă (funcție facută efectiv de uxQueueMessagesWaiting(\_spp\_rx\_queue);)

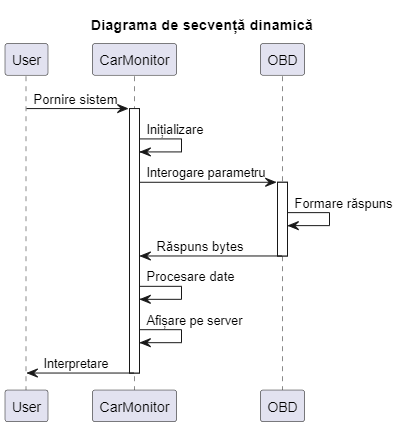
* int BluetoothSerial::read()

folosit pentru a citi din buffer valoarea primită.

Pentru a observa unde se folosesc funcțiile de Bluetooth, pe lângă cele uzuale folosite pentru pornirea modulului plăcii, trebuie realizată o analiză a API-ului ELMduino. Atât funcția available() cât și cea read() sunt chemate în interiorul funcției get\_response() care aparține clasei ELM327. Aceasta la rândul ei este chemată de ELM327::processPID care este funcția de bază pentru orice parametru disponibil în librărie.

### Diagrama de secvență dinamică

În figura următoare este descrisa diagrama de secvență dinamică sau generică, care explică modul de funcționare al sistemului. Procesul începe prin pornirea sistemului, acțiune inițiată de către utilizator. Sistemul apoi se initializeaza, atat din punct de vedere al Bluetooth pentru conexiunea cu OBD, cât și al WiFi pentru rularea serverului. În faza urmatoare, sistemul interoghează ECU al masinii cu parametrul necesar prin intermediul OBD. Se formează răspunsul sub forma de sir de bytes este returnat catre sistemul de pe placa ESP-32. Atât interogarea parametrului necesar cât și procesarea datelor pentru a fi disponibile într-o manieră lizibilă sunt operate de funcțiile din librăria ELMduino. După ce procesarea datelor s-a încheiat cu succes, resursele sunt cedate procesului de rulare a serverului pentru a minimiza întârzierile.



În continuare se vor prezenta atât diagrame de secvență cât și detalii de implementare al sistemului, mai exact pe partea de funcționare al API-ului ELMduino, serverului WebAsyncServer, și modul de conectare al acestora în proiectul CarMonitor.

### Interogarea unui parametru (rpm) prin API-ul ELMduino

A screenshot of a computer program

Description automatically generated

ELMduino este un API folosit în acest proiect, disponibil de asemenea pe GitHub [7], prin intermediul căruia placa trimite PID-uri către cititorul OBD. De asemenea, tot prin intermediul lui, se tratează răspunsul primit sub forma de 8 bytes, într-o maniera care se poate trimite către web server.

După cum se menționează în [7], libraria este non-blocking. Acest lucru semnifică faptul că execuția firului principal nu este oprită, ci permite sa fie executată în paralel. Un dezavantaj pe care îl prezintă este faptul că într-un fir de execuție nu se pot trimite mai multe request-uri de PID deodată, ci trebuie realizată o implementare al unei mașini cu stări finite, ceea ce s-a realizat în codul proiectului. Astfel, la fiecare rulare al funcției loop(), codul va avea un use-case diferit de cel anterior, și va permite trimiterea unui PID altul decât cel anterior.

API-ul poate trimite o listă variată de request-uri PID, dintre care folosim kph(), rpm(), fuelRate(), oilTemp(). În plus, avem alte funcții precum engineLoad(), care returnează tipul float, și reprezintă procentul solicitării motorului, throttle() care returnează un float reprezentând procentul apăsării pedalei de accelerație, float manifoldPressure() care reprezintă presiunea în kPa al admisiei de aer. De asemena, prin functia void resetDTC(), se sterg erorile prezente în ECU. Această functie este de tip blocking, deoarece standardul SAE necesită existența unui mesaj de confirmare al intenției de resetare al erorilor din ECU.

În diagrama anterioară se observă fluxul de date și controlul dintre diferitele componente ale sistemului, pe partea de interogare și cedare de informații al sistemului CarMonitor, din partea ECU al mașinii. Actorii principali în acest segment al sistemului sunt: utilizatorul (care interacționează cu sistemul prin server sau pagina web), sistemul propriu-zis CarMonitor care se ocupă cu logica de manevrare în timp real al serverului și al API-ului; API-ul ELMduino care pune la dispoziție funcțiile de interogare și procesare al informațiilor, cititorul OBD dotat cu chipul ELM327 capabil sa transmită prin Bluetooth date, și calculatorul central al vehiculului, ECU, de la care se preiau informațiile dorite.

În cazul parametrului rpm (rotațiile pe minut ale motorului), sistemul incepe prin a seta parametrii de service și PID pe 0x01 respectiv 0x02, care sunt echivalenți cu comanda de citire rpm. Apoi, asemenea logicii CarMonitor, API-ul ELMduino este implementat sub forma de state machine, cu cinci cazuri posibile: SEND\_COMMAND, WAITING\_RESP, RESPONSE\_RECEIVED, DECODED\_OK, ERROR.

Dacă starea API-ului este de SEND\_COMMAND, are loc formarea PID-ului și interogarea ECU prin cititorul OBD, care acționează ca un mijlocitor dintre cele două module. După trimiterea request-ului, starea următoare a API-ului va fi WAITING\_RESP.

ECU răspunde atât cu un mesaj cât și cu un status, care, prin intermediul cititorului OBD, sunt puse într-un registru de Rx pe placa ESP-32. Dacă răspunsul cititorului este ELM\_SUCCESS, buffer-ul de primire este parcurs iar valoarea din acesta este prelucrata conform formulelor prezentate în [1], mai exact în tabelul pentru serviciul 0x01. După acestea, valoarea interogată, în acest caz fiind cea de rpm, este returnată către CarMonitor de către ELMduino. În caz contrar, dacă răspunsul primit de la OBD este eroare sau time-out, starea următoare va fi SEND\_COMMAND, pentru a asigura interogarea următorului parametru, chiar dacă cel actual a eșuat.

Diagrama prezentată, cu toate că exemplifică cazul în care parametrul interogat este rpm, se aseamănă în mare măsura cu ceilalți parametrii disponibili. În cazul altui parametru, diferența principala constă în faptul că se va schimba doar setarea PID-ului. În cazul în care parametrul dorit a fi citit este viteza actuală, PID-ul va fi 0x0D, pentru tensiunea bateriei există o funcție specială care detectează acest parametru deoarece nu este implementat un cod special. Mai exact, se trimite comanda “AT RV” pentru a returna tensiunea detectată la cititorul OBD. Deci, acesta nu este un request PID adresat către ECU, ci un request către OBD-ul în sine.

Pentru poziția pedalei de accelerație se folosește 0x69. Acesta va returna o valoare care reprezintă curentul în Amperi. Valoarea finală este obținută prin calcularea cu formula , care va returna procentul de apăsare al pedalei. Pentru temperatura lichidului de răcire se folosește PID-ul 0x05, iar valoarea reală va rezulta din formula .

Pentru procentul de sarcină al motorului, se va folosi PID-ul 0x04, iar formula va fi asemănătoare cu cea de la poziția pedalei de accelerație, mai exact . În cazul nivelului de combustibil se va folosi PID-ul 0x2F, cu aceeași formulă. Pentru temperatura uleiului se va folosi PID-ul 0x5C care este trecut prin formula

### Diagrama de secvență pentru CarMonitor - Se adauga si detalii de returnari, etc.

A screen shot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

## Implementare WebAsyncServer

Așa cum s-a menționat în subcapitolul anterior, rolul conexiunii dintre ESP-32 și cititorul OBD dotat cu ELM327 este de a aduce datele cerute de la mașină, care se află într-o stare neprelucrată. Acestea se prelucrează în interiorul API-ului ELMduino, și devin disponibile pentru a fi manipulate în continuare. Următoarea fază a sistemului este reprezentată de luarea acestor date prelucrate și afișarea lor într-o interfață online, cu scopul de a monitoriza parametrii mașinii de la distanță.

A black screen with white text

Description automatically generatedÎn cazul sistemului “CarMonitor”, s-a folosit protocolul de securitate WPA3-SAE, activat de flag-ul DCONFIG\_ESP\_WIFI\_ENABLE\_WPA3\_SAE, deoarece in momentul testării, placa se conecta prin hotspot la un smartphone cu sistemul de operare iOS, și orice client care se conectează la acest tip de dispozitiv trebuie să aibă acest protocol de securitate activat. De asemenea, așa cum s-a menționat anterior, în momentul build-ului este necesară activarea flag-ului DCONFIG\_SW\_COEXIST\_ENABLE pentru a asigura coexistența taskurilor de WiFi și Bluetooth. Motivul acestui flag este reprezentat de faptul că atât modulul WiFi cât și cel Bluetooth folosesc aceeași antenă, cu frecvența de 2.4 GHz. În cazul în care acest flag nu era specificat, modulul WiFi ar fi controlat în totalitate accesul la antenă, fără a preda controlul către modulul Bluetooth. Aceste două flag-uri s-au activat în fișierul plarformio.ini la secțiunea build\_flags.

Tot în fișierul platformio.ini, la secțiunea lib\_deps, se specifică dependințele pentru proiectul PlatformIO. Acesta, pe lângă dependința ELMduino, va descărca și ESP Async WebServer, care are scopul de a crea un web server HTTP asincron.

Un web server asincron este un server numit non-blocking, însemnând faptul că acesta poate administra mai multe conexiuni în același timp fără a aștepta terminarea altui task, care ar întârzia semnificativ performanțele sistemului. De asemenea, utilizarea unui web server asincron are ca scop principal în această lucrare de a îmbunătăți performanțele plăcii ESP-32, dar și de a utiliza resursele disponibile într-un mod cât mai optim.

Un alt motiv pentru care s-a decis utilizarea acestei dependințe este îl reprezintă potențialul de scalabilitate al sistemului, în sensul în care dacă se va dori ca serverul să prezinte datele a mai multor autovehicule care se află sub monitorizare, acesta va putea suporta conexiunea cu mai multe dispozitive ESP-32 client, conectate la altul care ar avea rol de server gazdă. Serverul asincron poate prelucra request-uri de la multiple dispozitive fără a crea un thread separat pentru fiecare client.

### Diagrama de secvență dinamică

A diagram of a computer program

Description automatically generated

În momentul activării sistemului „CarMonitor”, acesta la rândul lui activează serverul de tip HTTP în constructorul din clasa WebServerHandler, clasă creată pentru a conține metodele folosite în crearea serverului, construirea datelor de tip json, montarea fișierelor statice și construirea vectorilor pentru graficele vitezei și turației pe minut.

La pornirea serverului, automat începe activarea interfeței, iar un local IP este printat în serial monitor-ul pus la dispoziție de PlatformIO, pentru ca orice client din acea rețea să se poată conecta și să monitorizeze în timp real parametrii vehiculului.

Acesta apoi începe procesul de conectare și comunicare cu cititorul OBD prin API-ul ELMduino, iar după ce primele date sunt disponibile, sunt afișate în interfață prin funcția updateHistory() și metoda handleClient().

Pe măsură ce codul avansează în ciclul loop(), datele continuă să fie afișate la un interval de o secundă în interfață pentru a asigura integritatea datelor și pentru a acorda timp să se formeze atât răspunsul din partea ELMduino, cât și al obiectului json care conține datele respective. Aceste date ulterior vor fi trimise către un addEventListener din fișierul script.js, care face parte din setul de fișiere statice montate în memoria ESP-32.

Toate metodele și funcțiile descrise în acest capitol fac parte din fișierele webhandler/webhandler.cpp și webhandler/webhandler.h, fișiere care se ocupă de tot ce ține de serverul asincron găzduit de ESP-32.

### Diagrama de secvență pentru WebServer.begin()

A screenshot of a computer program

Description automatically generated

În contextul fișierului webhandler.cpp și al webserver-ului în general, se utilizează protocolul SPIFFS (SPI Flash File System), destinat dispozitivelor cu memorie de tip NOR, pentru a stoca fișierele folosite în construirea serverului. În momentul în care se apelează server.begin(), SPIFFS se inițializează la rândul sau prin SPIFFS.begin(true). Atunci, sistemul de fișiere se pregătește de utilizare, iar dacă montarea a eșuat (dacă sistemul de fișiere este corupt), are loc reformatarea, în sensul în care se vor șterge datele existente în memoria alocată, și se va cea un nou sistem de fișiere statice, gol. După această reformatare, se va reîncerca montarea fișierelor statice. Este important de menționat faptul că aceste mecanism se va derula în modul descris doar dacă parametrul formatOnFail al metodei .begin() este pus pe true. Altfel, dacă parametrul formatOnFail se mapează la false, metoda .begin() va returna o eroare și nu se va mai încerca rezolvarea prin formatare a erorii întâlnite.

Prin montarea sistemului de fișiere se înțelege aducerea acestora de către sistemul de operare într-o zonă de memorie disponibilă pentru utilizator sub formă de arbore de directoare. De asemenea, montarea se referă și la pregătirea acestora pentru a fi citite (indicat de parametrul al doilea din apelarea metodei .open() al obiectului SPIFFS). Acest aspect include și setările aferente creării structurilor de date și al meta datelor din memoria flash.

File file = SPIFFS.open(path, "r");

Un alt aspect important de menționat îl reprezintă apelarea metodelor .on() ale obiectului server. Rolul acestor metode este de a seta rutele către fișierele folosite în construcția web serverului, dar și de a seta ce se va întâmpla cand un request se face la fișierul respectiv folosind metoda HTTP. În cazul proiectului, s-a utilizat HTTP\_GET deoarece aduce informații de la server și nu se dorește modificarea lor, iar pentru a trata request-ul GET primit la calea respectivă, se va apela metoda handleSensorData() pentru a forma string-ul de date. Handle-urile sunt reprezentate de funcții lambda pentru a asigura un cod compact, care apelează metodele handleFile() în cazul celor trei fișiere ale interfeței, cât și metoda handleSensorData() discutată anterior.

### Diagrama de secvență pentru handleSensorData()

A screenshot of a computer

Description automatically generated

Această metodă este responsabilă pentru tratarea request-urilor HTTP GET trimise către calea /data al serverului. Atunci când un request este trimis, metoda handleSensorData() va fi apelată. Metoda începe prin a crea un obiect JSON de tip String. Acesta va conține atât datele actuale preluate prin API-ul ELMduino cât și datele istorice folosite în graficele pentru viteză și turațiile pe minut ale motorului. Obiectul JSON este creat și inițializat cu caracterul ‘{’ care indică începutul șirului de date.

În următoarea fază, se adaugă la șir valorile care se vor folosi în interfață, la care se aplică funcția String() pentru a converti valoarea respectivă cu scopul de a putea fi introdusă în șirul obiectului JSON. Fiecare valoare convertită este urmată de separatorul ‘,’ pentru a delimita datele între ele. Valorile trimise sunt atribuite la ID-urile "historySize", "speedCurrent", "rpmCurrent", "voltageCurrent", "coolantTempCurrent", "oilTempCurrent", "loadCurrent", "throttleCurrent", și "fuelCurrent". Aceste ID-uri sunt mapate la variabilele din cod HISTORY\_SIZE, speedVariable, rpmVariable, voltageVariable, coolantTempVariable, oilTempVariable, loadVariable, throttleVariable, și fuelLevelVariable. Aceste variabile sunt cele care stochează valorile primite de la ECU prin intermediul ELMduino în logica sistemului CarMonitor.

După ce aceste date sunt atașate obiectului JSON, urmează adăugarea valorilor istorice folosite de grafice. Actualizarea lor se face o dată pe secundă pentru a asigura integritatea și corectitudinea datelor primite și transmise. Aceste valori sunt stocate în structuri de date de tip vector tipizați (template) pentru a suporta atât valori întregi cât și în virgulă flotantă. Aceste date sunt adăugate în vectori prin funcția join(), care este supraîncărcată, cu scopul de a construi un String pe baza valorilor din vector, care mai departe să fie adăugat în obiectul JSON. Funcția join() are doi parametri: vectorul (array-ul) care conține datele anterioare și un separator sau delimitator reprezentat de caracterul ‘,’. Aceste șiruri sunt incluse în paranteze închise [ ] pentru a forma un array JSON.

În final, obiectul JSON este închis prin caracterul ‘}’. Metoda server.send() este apoi apelată pentru a trimite un răspuns către client. Statusul trimis este setat la 200 (OK), se specifică tipul conținutului “application/json” și structura de date trimisă este chiar jsonData construită anterior. Această secțiune trimite datele preluate de la ECU într-o formă care poate fi citită și folosită facil.

## Interfața cu utilizatorul

### Aspecte generale

Așa cum s-a menționat anterior, fișierele de tip HTML, CSS și JavaScript sunt stocate în memoria plăcii ESP-32 folosind sistemul de fișiere SPIFFS, care are un strat de abstractizare între maparea la nivel de adrese și utilizarea efectivă al acestor fișiere. De aceea, programatorul manevrează aceste fișiere prin intermediul API-urilor puse la dispoziție de Espressif. Acest fapt permite programatorului să nu fie nevoit să cunoască detalii precum adresa la care sunt montate fișierele. În general nivelurile de abstractizare puse la dispoziție programatorului de către compania Espressif sunt implementate în mod robust, ceea ce facilitează programarea pe această placă de dezvoltare.

Aceste fișiere folosite în construirea interfeței definesc structura, stilul și comportamentul pe care interfața UI web o va avea în momentul în care utilizatorul o folosește. În continuare se detaliază fiecare fișier care compune această interfață.

### Fișierul HTML (HyperText Markup Language)

HTML (HyperText Markup Language) este un limbaj de programare de tip markup care definește structura informației pe care un utilizator dorește să o expună. Așa cum se detaliază în [8], HTML este compus dintr-o serie de elemente folosite pentru a formata informația. Cu aceste elemente, utilizatorul poate să încapsuleze (enclosing) sau să înfășoare (wrap) datele pentru a modifica le aspectul vizual. De aceea, orice element (tag) din fișierul HTML este deschis și închis folosind caracterele <[cuvânt cheie]> respectiv <[cuvânt cheie]/>.

La începutul oricărui fișier de tip HTML este necesar să se introducă comanda <!DOCTYPE html> . Rolul acestuia este de a preveni browser-ul să folosească moduri de redare diferite față de cele standard, sau să îl forțeze să folosească cele mai relevante specificații în defavoarea altora c are pot compromite compatibilitatea dintre versiuni sau specificații. Acest lucru este menționat și detaliat în amănunt la sursa [9].

HTML permite introducerea de link-uri, imagini, titluri de tip antet (header) pe mai multe nivele, formatarea scrisului în tip RTF (Rich Text Format) ca fiind italic, îngroșat, subliniat, etc. Aceasta permite o oarecare personalizare a afișării dar se observă aspectul rigid în acest sens.

Așa cum s-a menționat anterior, fișierul HTML este scheletul interfeței, de aceea, index.html conține elementele care pun în interfață informațiile de care are nevoie utilizatorul, datele mașinii afișate în timp real organizate în tabele, grafice specifice vitezei, rotaților pe minut ale motorului, dar și butonul de “Freeze Readings”.

Fișierul începe prin a face disponibil browser-ului versiunea de HTML în care pagina este scrisă. Aceasta este prima linie al oricărui document HTML5.

<!DOCTYPE html>

Urmează elementul <html> </html> care reprezintă elementul de bază al oricărui document HTML. În acest tag se încapsulează toate celelalte tag-uri care formează pagina.

<html>

</html>

Următorul element este <head>. Acest element conține meta-informațiile legate de documentul HTML, precum link-uri către fișiere, stiluri sau script-uri externe. Acesta nu este vizibil utilizatorului, face parte din setul de informații care este atribuit lui index.html.

În cazul fișierului sistemului, există două sub-elemente în cadrul <head>. Primul este un link intern către fișierul CSS care conține stilurile folosite, iar al doilea este un element script care corespunde unei librării JavaScript folosită pentru a declara comportamentul graficelor care se utilizează în interfață. Aceste două elemente sunt vizibile mai jos.

<head>

  <link rel="stylesheet" type="text/css" href="style.css">

  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

</head>

Elementul care conține tot content-ul care este afișat pe pagină este în tag-ul <body>. Această secțiune începe prin a declara elementele de tip header Tag-ul <h1> este la cel mai înalt nivel și este folosit pentru a scrie titlul sistemului, acela de “CarMonitor” . Al doilea heading, <h2>, este folosit pentru a afișa autorul proiectului iar instrucțiunile de utilizare sau contextul sunt scrise sub tag-ul <p>.

De asemenea, sub tag-ul <button> se găsește o funcționalitate cu care utilizatorul poate interacționa. Acesta definește un buton care are diferite proprietăți și comportamentul este descris în fișierul JavaScript. În starea în care datele circulă constant pe interfață, textul butonului este “Freeze Reading”. Dacă utilizatorul dorește să oprească datele din a circula și să le analizeze direct, acesta poate apăsa butonul. În acest moment, datele din tabele și grafice sunt oprite iar textul butonului este “Unfreeze Reading”. Odată ce utilizatorul consideră că dorește să continue cu urmărirea parametrilor, acesta poate apăsa butonul din nou, iar datele vor fi afișate în timp real atât în tabele cât și în grafice. Funcția responsabilă de acest comportament al butonului este denumită toggleDataFetch(), funcție disponibilă în script.js. De asemenea, se observă că butonului i se atribuie un id = ”toggleButton”. Prin intermediul acestuia, fișierul JavaScript se poate referi la buton și să îl manevreze în funcție de dorința programatorului. Secvența de cod care declară aceste elemente este descrisă mai jos.

  <h2>by Paul Horvath</h2>

  <h1>Car Monitor</h1>

  <p>In order to successfully observe the car parameters, you should first ensure that the OBD reader is paired to the ESP-32 board. Then, ensure that the ESP-32 is connected to the internet using the config file.</p>

  <p>Real-Time Measurements:</p>

  <button id="toggleButton" onclick="toggleDataFetch()">Freeze Reading</button>

În continuarea tag-ului <body> se definește zona în care datele vor fi afișate în timp real la un interval de o secundă. Această zonă este implementă sub forma unui tabel. Zona este împărțită în două tabele și două grafice. Primul tabel este afișat și conține linii și coloane pentru primii patru parametrii pe care sistemul îi citește. Este important de menționat faptul că acești parametri pot fi configurați sau aleși în a fi expuși în funcție de ce se dorește a se urmări. Această configurare nu este una fixă, se poate modifica. Primul tabel indică parametrii următori: viteza (pe linia “Speed”), turațiile motorului (pe linia “RPM”), tensiunea bateriei (pe linia “Battery Voltage”) și temperatura lichidului de răcire al motorului (pe linia “Coolant Temperature”). Acestea, ca în cazul butonului, au legate id-uri pentru fișierul JavaScript.

  <table>

    <tr>

      <th>Speed</th>

      <td id="speed"></td>

    </tr>

    <tr>

      <th>RPM</th>

      <td id="rpm"></td>

    </tr>

    <tr>

      <th>Battery Voltage</th>

      <td id="voltage"></td>

    </tr>

    <tr>

      <th>Coolant Temperature</th>

      <td id="cool"></td>

    </tr>

  </table>

Următoarele elemente sunt două grafice, pentru viteză respectiv turațiile motorului. Acestea sunt clase de tip “charts-container”. Primul container este un container pentru toate graficele care urmează. În interiorul lui se definesc două grafice cu clasa “chart-container”. Ambele tipuri de containere pot fi stilizate în fișierul CSS. Acestea conțin un element de tip <canvas> cu id-uri specifice pentru a fi identificabile de către script.js în DOM (Document Object Model).

<div class="charts-container">

    <div class="chart-container">

      <canvas id="rpmChart"></canvas>

    </div>

    <div class="chart-container">

      <canvas id="speedChart"></canvas>

    </div>

  </div>

Al doilea tabel este definit după aceste grafice, tabel care continuă cu afișarea celorlalți parametri. În configurația prezentată în această lucrare următorii parametri sunt: temperatura uleiului (pe linia “Oil Temperature”), nivelul de solicitare al motorului (pe linia “Engine Load”), poziția pedalei de accelerație (pe linia “Throttle Position”) și nivelul combustibilului în rezervor (pe linia “Fuel Level”). De asemenea, și aceste date au id-uri specifice.

  <table>

    <tr>

        <th>Oil Temperature</th>

        <td id="oil"></td>

      </tr>

    <tr>

        <th>Engine Load</th>

        <td id="load"></td>

      </tr>

      <tr>

        <th>Throttle Position</th>

        <td id="throttle"></td>

      </tr>

      <tr>

        <th>Fuel Level</th>

        <td id="fuel"></td>

      </tr>

</table>

În ultima parte a fișierului există un tag <script> care are rolul de a crea o referință către un fișier JavaScript, în acest caz script.js. În atributul “src” se specifică acest fișier. În momentul în care browser-ul întâlnește acest tag, el trimite un request la URL-ul specificat, descarcă fișierul JS, și execută codul din acel fișier. Un aspect important este plasarea acestui tag, deoarece dacă este plasat la începutul fișierului, afișarea interfeței poate fi întârziata deoarece browser-ul construiește DOM la început. Dacă este plasat la final, înainte de a închide tag-ul <body> atunci script-ul va fi executat după ce DOM este construit, fapt care poate îmbunătăți semnificativ performantele web serverului.

<script src="script.js"></script>

</body>

</html>

### Fișierul CSS (Cascading Style Sheets)

Conform [10], CSS nu este un limbaj de programare, nici un limbaj markup, ci este un “style sheet language”. Prin CSS se stilizează elementele HTML. Fișierul HTML are legat stilul scris în CSS prin linia documentată anterior unde se menționează acest lucru, în tag-ul <head>. De asemenea, așa cum se menționează în aceeași referință, un stil este alcătuit dintr-un selector, și declarare, care la rândul ei este împărțita în proprietate și valoare.



Imagine preluată din [10]

În cazul de față, există o singură regulă pentru tag-ul <p> din fișierul HTML, aceea ca fontul să aibă culoarea roșie. În acest mod se realizează orice regulă pe care o dorește programatorul, și care va fi implementată de către browser. În cazul unui selector, multiple proprietăți pot fi adăugate împreuna cu valorile lor, pentru a forma un set de reguli.

Un aspect important la care programatorul trebuie să fie atent este stilul de „cascadare” al seturilor de reguli pentru același selector. În cazul în care avem două sau mai multe stiluri pentru același selector, ultimul set de reguli va fi luat în considerare, din cauza modului „cascadă” în care fișierul CSS este citit. Dacă totuși se dorește ca un set de reguli să aibă prioritate, se poate adăuga cuvântul cheie !important pentru a oferi această proprietate selectorului respectiv.

Un beneficiu pe care îl aduce utilizarea CSS în stilizarea paginii web este reprezentat de consistența cu care se pot prezenta interfețele de-a lungul sistemului. Această conferă un aspect uniform și conferă utilizatorului o îmbunătățire considerabilă a experienței pe care acesta o are în momentul în care utilizează platforma. Acest beneficiu are importanță și în momentul în care se dorește scalarea sistemului, în sensul în care codul nu este duplicat, iar același fișier CSS poate fi folosit sau actualizat în funcție de noul nivel la care a ajuns sistemul.

Fișierul CSS în cadrul sistemului reprezintă o componenta crucială care asigură interfața atrăgătoare din punct de vedere vizual. Deci, în implementarea fișierului CSS s-a avut în vedere aspectul dorit al paginii, care să se prezinte ca unul plăcut și atractiv pentru utilizator. În plus, acesta este construit cu scopul de a avea o lizibilitate ridicată, utilizabilitate și practicabilitate adecvată unui utilizator care dorește să urmărească parametrii unei mașini.

Una dintre caracteristicile cheie pe care această componentă le are este tehnica de design receptivă, în sensul în care prin natura lui, fișierul este construit să funcționeze pe o varietate mai largă de dispozitive. Acest lucru este implementat de regula @media (max-width: 600px), care aplică stiluri diferite în momentul în care lățimea este 600 pixeli sau mai puțin.

Acest fișier CSS este, de asemenea, proiectat având în vedere scalabilitate. Utilizează selectoare de clasă și selectoare de elemente, permițând adăugarea ușoară de elemente și componente noi, fără a fi nevoie de modificări CSS extinse. Utilizarea unităților relative (cum ar fi procentele pentru lățimi) și layout-urilor flexibile (cum ar fi Flexbox) asigură că designul se poate adapta la diferite dimensiuni și cantități de conținut. În continuare se vor detalia selectoarele cele mai importante în construirea interfeței.

Un selector foarte important este cel care se referă la elementul <body> din fișierul HTML. Acesta are regula de a specifica fontul “Segoe UI”, dacă acesta nu este disponibil, va încerca fontul „Tahoma”, „Geneva”, „Verdana”, sau orice alt font de tip sans-serif. Aici, de asemenea, se setează culoarea de fundal, culoarea textului, elementele aliniate la centru, setează proprietatea flex, pentru a permite utilizarea layout-ului Flexbox pentru moștenitori direcți ai elementului body.

body {

    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;

    background-color: #cac8ee; /\* Background color \*/

    color: #495057; /\* Text color \*/

    margin: 0;

    padding: 20px;

    display: flex;

    flex-direction: column;

    align-items: center;

    justify-content: start;

}

Alte selectoare importante sunt reprezentate de cele pentru grafice și cel pentru butonul de “Freeze/Unfreeze Reading”. În cazul containerelor pentru grafice s-a urmărit un design simplu, ușor de a fi citit și interpretat de către utilizator. În cazul butonului, s-a urmărit să fie ușor de anticipat comportamentul sistemului în urma apăsării acestuia. De asemenea, există selectoare pentru tabel, heading-uri și paragrafe, cu reguli care urmăresc aspectul general cât mai user-friendly, pentru a fi cât mai ușor de folosit pentru oricine.

Pe scurt, style.css joacă un rol vital în a face aplicația web atractivă, ușor de utilizat și adaptabilă la diferite dispozitive și dimensiuni de ecran. Este o componentă crucială în crearea design-ului web captivant și receptiv.

### Fișierul JS (JavaScript)

Așa cum se descrie în [11], JavaScript este un limbaj de programare foarte folosit în website-uri, webserver-e, care aduce introduce interactivitatea pe aceste platforme. Acesta a fost inventat de Brendan Eich în cadrul companiei Netscape (predecesorul Mozilla). Scopul final al acestuia era crearea unui limbaj de programare către mase, pentru a ajuta non-programatorii să creeze web site-uri dinamice și interactive, așa cum se menționează în [12]. De asemenea, tot în aceeași sursă se descrie și că „JavaScript este o marcă comercială a Oracle Corporation din Statele Unite. Marca comercială a fost eliberată inițial către Sun Microsystems la 6 mai 1997 (companie cu care Netscape a colaborat pentru crearea acestui limbaj) și a fost transferată către Oracle când au achiziționat Sun în 2009”.

Revenind la datele furnizate de [11], se poate identifica faptul că JavaScript este compact dar foarte flexibil, prin scrierea tool-urilor bazate pe acesta, care permit accesul la funcționalități precum API-uri construite în web browser-e, pentru a crea dinamic fișiere HTML și stiluri CSS; API-uri de la third-parties care permit programatorilor să adopte dar și construirea framework-urilor și librăriilor care se pot aplica HTML-urilor pentru a accelera crearea site-urilor și aplicatiilor.

O caracteristică importantă al JS este reprezentată de prezența interfeței DOM (Document Object Model). Aceasta este o structură de documente bazată pe un arbore, care permite programatorilor să manipuleze conținutul, structura și stilul acestora. În cadrul acestei interfețe, fiecare element, atribut sau text este reprezentat de un nod, iar mai multe noduri sunt organizate într-un arbore. În continuare se prezintă un exemplu de cum este reprezentat un fișier HTML în DOM:

Document

└── <html>

├── <head>

│ └── <title>

└── <body>

├── <h1>

└── <p>

Așa cum s-a discutat în introducere, JavaScript este un limbaj care conferă serverului web un caracter dinamic și interactiv, eliminând natura statică al acestuia. Cu acest principiu în vedere a fost construit și fișierul script.js. Acest element este crucial deoarece caracterul dinamic adus de JS este indispensabil experienței utilizatorului cu sistemul “CarMonitor”. Cu ajutorul lui, elementele care reprezintă parametrii mașinii se pot modifica în timp real în interfața web, pe măsură ce sunt interogate, primite, procesate și prelucrate pentru a fi utilizabile și relevante către utilizator.

În acest sens se vor detalia componentele care alcătuiesc fișierul script.js și care sunt necesare pentru a oferi utilizatorului informații în timp real despre autovehiculului pe care dorește să îl monitorizeze.

Începând cu graficele, se declară obiectele care stochează datele și opțiunile acestora. În cazul datelor, acestea sunt împărțite în două, pentru fiecare grafic: cel de viteză instantă speedChartData și cel de turația motorului, rpmChartData. În ambele cazuri avem doi vectori declarați: unul numit labels care va fi populat dinamic, și reprezintă valoarea pe care un element o are cand este transpus în grafic. Această valoare corespunde și este egală cu valoarea instantanee al parametrului respectiv citit. Al doilea vector prezent în obiectul de date este datasets, care este alcătuit din: label (care reprezintă numele tabelului), data (vector care va fi populat cu date istorice), fill (boolean care marchează dacă se va colora interiorul graficului), borderColor (care specifică culoarea pe care o va avea graficul) și tension (care reprezintă cât va avea linia aspectul de curbă la fiecare tranziție). Pentru opțiunile graficelor avem variabila chartOptions care specifică faptul că graficele sunt de tip linie, încep pe axa Y de la valoarea zero iar raportul de aspect al diagramelor nu este menținut, permițându-le să se redimensioneze dinamic.

let speedChartData = {

    labels: [], // This will be populated dynamically

    datasets: [{

        label: 'Speed',

        data: [], // This will be populated with historical data

        fill: false,

        borderColor: 'rgb(75, 192, 192)',

        tension: 0.1

    }]

};

let rpmChartData = {

    labels: [], // This will be populated dynamically

    datasets: [{

        label: 'RPM',

        data: [], // This will be populated with historical data

        fill: false,

        borderColor: 'rgb(192, 75, 75)',

        tension: 0.1

    }]

};

let chartOptions = {

    type: 'line',

    options: {

        scales: {

            y: {

                beginAtZero: true

            }

        },

        maintainAspectRatio: false

    }

};

Pentru crearea acestor grafice avem liniile de cod următoare care apelează constructorul Chart cu elementele adecvate.

let speedChart = new Chart(

    document.getElementById('speedChart'),

    Object.assign({ data: speedChartData }, chartOptions)

);

// Create the RPM chart

let rpmChart = new Chart(

    document.getElementById('rpmChart'),

    Object.assign({ data: rpmChartData }, chartOptions)

);

În continuare se declară o funcție special creată pentru butonul de “Freeze/Unfreeze Reading”. Aceasta este toggleDataFetch() și scopul ei este să manipuleze variabila isFetching pentru a elibera sau seta intervalul la care se aduc informațiile prelucrate de “CarMonitor” pentru a fi afișate pe interfață. Această funcție apelează funcția fetchSensorData() când starea butonului corespunde cu aceasta.

let isFetching = false;

let fetchInterval;

// Function to start or stop fetching data based on button state

function toggleDataFetch() {

    isFetching = !isFetching; // Toggle the fetching state

    const button = document.getElementById('toggleButton');

    if (isFetching) {

        // Ensure no multiple intervals are set

        if (fetchInterval) clearInterval(fetchInterval);

        fetchInterval = setInterval(fetchSensorData, 1000); // Start fetching data every second

        button.textContent = "Freeze Reading"; // Update button text

    } else {

        clearInterval(fetchInterval); // Stop the interval

        button.textContent = "Unfreeze Reading"; // Update button text

    }

}

Următoarea funcție documentată este fetchSensorData() care este responsabilă cu actualizarea datelor furnizate de ECU al autovehiculului și prelucrate prin API-ul ELMduino. Aceasta pentru început utilizează funcția de preluare (fetch) pentru a face un request GET către endpoint-ul /data documentat anterior la server în clasa WebServerHandler. Funcția fetch apoi returnează un așa-numit “Promise” (obiect returnat care marchează dacă o operație asincronă a fost terminată cu succes sau nu) care reprezintă răspunsul request-ului aflat în primul bloc then. Apoi funcția response.json() este chemată pentru a procesa datele JSON primite de la webhandler. În al doilea bloc de tip then, datele analizate sunt extrase din formatul JSON și sunt folosite pentru a actualiza conținutul text al diferitelor elemente din UI. Funcția updateCharts() este apelată cu structura de date extrase și preluate ca parametru. Această funcție actualizează datele din diagrame și le reîmprospătează. Dacă apare vreo eroare în timpul acestui proces al funcției fetchSensorData(), aceasta este prinsă în blocul catch și afișată în consolă.

function fetchSensorData() {

    fetch('/data')

        .then(response => response.json())

        .then(data => {

            document.getElementById('speed').textContent = data.speedCurrent + ' km/h';

            document.getElementById('rpm').textContent = data.rpmCurrent + ' RPM';

            document.getElementById('voltage').textContent = data.voltageCurrent + ' V';

            document.getElementById('cool').textContent = data.coolantTempCurrent + ' C';

            document.getElementById('oil').textContent = data.oilTempCurrent + ' C';

            document.getElementById('load').textContent = data.loadCurrent + ' %';

            document.getElementById('throttle').textContent = data.throttleCurrent + ' %';

            document.getElementById('fuel').textContent = data.fuelCurrent + ' %';

            // Update charts

            updateCharts(data);

        })

        .catch(error => console.error('Unable to get sensor data:', error));

}

Următorul element important de prezentat este metoda .addEventListener care. Acest event handler este setat să se activeze atunci când DOMContentLoaded este activat. El se activează când documentul HTML inițial a fost complet analizat și încărcat, fără a aștepta după stilurile din fișierul CSS sau alte date. Acest lucru îmbunătățește semnificativ performanța serverului și viteza cu care datele sunt aduse către UI. Event handler-ul simulează apăsarea butonului “Unfreeze Reading” pentru a porni aducerea datelor fără a mai fi nevoie de input-ul user-ului.

document.addEventListener('DOMContentLoaded', function() {

    // Initially stop data fetching until the button is pressed

    document.getElementById('toggleButton').click(); // Simulate button click to set initial state

});

Pe lângă aceste funcții descrise mai există și funcția updateCharts(data) care este apelată de fetchSensorData() și rolul ei este de a actualiza datele din grafice în funcție de data.historySize care sunt stocate sub formă de array-uri. Graficele operează sub formă de LIFO (Last In First Out) în sensul în care datele cele mai recente sunt adăugate la începutul graficului, iar datele cele mai vechi sunt șterse din istoric.

## Interfața vizuală

A screenshot of a computer monitor

Description automatically generated

A screenshot of a computer

Description automatically generated

# DEZVOLTARE SI TESTARE

## Testare și configurare

În testarea acestui proiect s-a utilizat o mașină marca BMW Seria 1 E87, anul 2009, cu un motor de 2.0 litri benzină, ce dezvoltă 122 cai putere și un cuplu de 180 Newton metru, cutie de viteză manuală. De asemenea, în programarea plăcii, s-a folosit limbajul de programare C++ (care include metode speciale pentru , Arduino SDK, iar pentru interfață s-a utilizat Visual Studio Code cu extensia PlatformIO care asigură script-uri pentru flash-uirea plăcii, atât a codului cât și a fișierelor folosite pentru web server.

De asemenea, s-au utilizat diferite dependințe și API-uri care pun la dispoziție funcționalitatea întregului sistem. Cele mai importante dintre acestea fiind ESP WebAsyncServer, SPIFFS și ELMduino, care vor fi prezentate în cele ce urmează.

[Mentionez cât ocupa sistemul, cât consuma la run-time, mapare, adrese, etc.]

## Planul de afacere(vezi GreenSpot)

Analiza SWOT Componente, cost Previziuni financiare ipotetice

# CONCLUZIE

## Obiective îndeplinite

De aceea, sistemul prezentat în această lucrare, care poartă numele „Car Monitor”, se explică singur. Acest sistem se conectează la autovehicul, reușește să preia în timp real și afișează într-o interfață prietenoasă cu utilizatorul, parametrii mașinii, cu scopul de a monitoriza și de a lua decizii calculate, pe baza datelor reale primite de la aceasta.

Acest sistem poate fi folosit, pe de-o parte, de către persoane fizice, care doresc o monitorizare mai amănunțită asupra propriei mașini, în vederea depanării problemelor, defectelor, sau doar pentru simpla evaluare a stării acesteia. În cazul în care se observa un comportament ciudat al unui parametru al mașinii, se pot lua decizii care să prevină o daună mai mare, economisindu-se bani și resurse. De aceea, utilizând acest sistem, se poate anticipa ce componentă și când în viitor va trebui schimbată.

Pe de altă parte, acest sistem poate fi folosit și de către persoane juridice, mai exact firme, care au în dotare o flotă de mașini în vederea închirierii lor. Pentru a se asigura că mașinile sunt tratate corespunzător și nu sunt abuzate, acestea pot fi dotate cu sistemul „Car Monitor” care poate înregistra o gamă mai variată de parametrii în care a fost utilizată aceasta. În cazul în care se constată că vehiculul a fost abuzat, (de exemplu turația motorului au fost nejustificat de mare pentru un timp îndelungat), se poate percepe o taxă suplimentară suportată de client, care are ca scop achitarea reparațiilor ulterioare aduse mașinii, ca rezultat al utilizării necorespunzătoare.

În plus, acest sistem poate fi folosit chiar și de către service-urile auto, mai exact de către mecanici și ingineri auto, pentru a decide dacă mașina funcționează în parametrii optimi sau nu. De exemplu, în cazul unei inspecții tehnice periodice, necesară pentru orice vehicul care utilizează drumurile publice, sistemul „Car Monitor” poate afișa utilizând ECU al mașinii dacă noxele emise sunt în parametrii normali sau nu. În acest caz, sistemul „Car Monitor” ar trebui sa fie o unealtă de confirmare și inspectorul să nu se bazeze exclusiv pe acesta, deoarece calculatorul mașinii poate returna valori false ale noxelor emise (comportament care poate fi accidental sau intenționat). În cazul în care ECU returnează valori false, exista posibilitatea de a se efectua o investigație pentru a vedea ce alte modificări au fost aduse mașinii, și daca acestea sunt conforme cu legislație în vigoare.

Important de menționat este faptul că parametrii afișați de către sistemul „Car Monitor” pot fi personalizați în funcție de fiecare caz.

Aspectul diferențiator al proiectului propus îl reprezintă afișarea datelor pe un web server dedicat și nu într-o interfață care este disponibilă doar pe dispozitivul conectat la cititor. Acest lucru restricționează monitorizarea și forțează utilizatorul să se afle în proximitatea mașinii, indiferent daca se folosește transmisia fizica (prin cablu) sau wireless (prin Bluetooth). Sistemul „Car Monitor” poate fi scalat ușor astfel încât monitorizarea să se facă de la distanță, iar web serverul să fie găzduit remote. Astfel, operatorul care monitorizează mașina (sau flota de mașini) poate observa parametrii de la sediul său.

## Impedimente

Bibliografie

|  |  |
| --- | --- |
| [1] | „Wikipedia OBDII & OBDII PIDS,” 9 March 2024. [Interactiv]. Available: https://en.wikipedia.org/wiki/OBD-II\_PIDs. |
| [2] | „ES32 vs ESP8266,” [Interactiv]. Available: https://makeradvisor.com/esp32-vs-esp8266/. |
| [3] | „Wikipedia ESP32,” 29 April 2024. [Interactiv]. Available: https://en.wikipedia.org/wiki/ESP32. |
| [4] | N. Kolban, Kolban's Book on ESP32, 2017. |
| [5] | „SPIFFS,” Espressif, [Interactiv]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/spiffs.html. |
| [6] | „ESPAsyncWebServer,” [Interactiv]. Available: https://github.com/me-no-dev/ESPAsyncWebServer. |
| [7] | „GitHub ELMduino,” [Interactiv]. Available: https://github.com/PowerBroker2/ELMduino/. |
| [8] | „HTML Basics,” Mozilla MDN, [Interactiv]. Available: https://developer.mozilla.org/en-US/docs/Learn/Getting\_started\_with\_the\_web/HTML\_basics. |
| [9] | „Doctype,” Mozila MDN, [Interactiv]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Doctype. |
| [10] | PowerBroker2, „Github,” [Interactiv]. Available: https://github.com/PowerBroker2/ELMduino/tree/2c242bcf28a05ebc46c48f650b16611452b04695/examples. |
| [11] | „CarScanner,” 2023. [Interactiv]. Available: https://www.carscanner.info/choosing-obdii-adapter/. |
| [12] | D. Manstetten, F. Beruscha, H.-J. Bieg, F. Kobiela, A. Korthauer, W. Krautter și C. Marberger, „The Evolution of Driver Monitoring Systems: A Shortened Story on Past, Current and Future Approaches How Cars Acquire Knowledge About the Driver's State,” *ResearchGate,* pp. 54-60, October 2020. |
| [13] | Dashy, „The Evolution of Vehicle Diagnostics: From Manual Troubleshooting to Smart Devices,” 8 July 2023. [Interactiv]. Available: https://medium.com/@DashyAuto/the-evolution-of-vehicle-diagnostics-from-manual-troubleshooting-to-smart-devices-69a57e18a0a2. |
| [14] | „On board diagnostics,” 17 March 2024. [Interactiv]. Available: https://en.wikipedia.org/wiki/On-board\_diagnostics#History. |
| [15] | S. Spânulescu, „ESP32 programming for the Internet of Things,” 2020. |
| [16] | „CSS Basics,” Mozilla MDN, [Interactiv]. Available: https://developer.mozilla.org/en-US/docs/Learn/Getting\_started\_with\_the\_web/CSS\_basics. |
| [17] | „JavaScript Basics,” Mozilla MDN, [Interactiv]. Available: https://developer.mozilla.org/en-US/docs/Learn/Getting\_started\_with\_the\_web/JavaScript\_basics. |