Calculatoare și Tehnologia Informației
2023 - 2024

Universitatea Politehnica din Timișoara

CarMonitor - Sistem de monitorizare de la distanță al parametrilor autovehiculelor

Candidat: Paul-Șerban Horvath

Coordonator științific: Conf. Dr. Ing. Lucian Prodan

Sesiunea: Iunie 2024

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 alți parametri) 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 manieră 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 și Bluetooth pe care le prezintă.

Cuprins

1. INTRODUCERE

1.1 Scurt istoric OBD

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 să 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 adoptă standardul OBD-II, 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].

1.2 Descriere OBD-II și PID

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 0x010D. Citirea oricărui parametru se face prin byte-ul 0x01 (frecvent utilizat în această lucrare), urmat de parametrul dorit (în acest caz 0x0D). Pentru turația motorului avem PID-ul 0x0C, pentru consumul de

combustibil avem 0x5E, pentru temperatura uleiului între 0x01 si 0x1F, depinzând de producătorul vehiculului, pentru temperatura lichidului de răcire avem 0x05, pentru poziția pedalei de accelerație 0x11 iar pentru tensiunea furnizată de baterie 0x9A.

Standardul SAE J1979 definește 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:

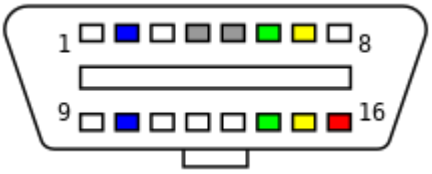


Fig. 1.2.1OBD-II tip A – vedere frontală [1]

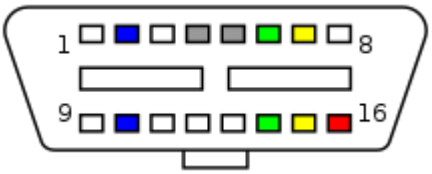


Fig. 1.2.2 OBD-II tip B – vedere frontală [1]

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

Fig. 1.2.3 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.

2. STATE OF THE ART

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 prezentat în această lucrare.

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 OBD-II al vehiculului 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 (modificabile 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 să afișeze acești parametri într-o interfață al unei aplicații. 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.

De asemenea, pe piața aceasta se găsesc soluții care prezintă o abordare diferită față de cele prezentate anterior. Aceste soluții integrează capabilitățile de interogare, preluare și prelucrare al parametrilor mașinii, dar elementul în plus este că acestea sunt afișate pe servere, pentru ca utilizatorul să le monitorizeze, să ia decizii informate, să verifice starea vehiculului sau orice altă operație, toate acestea de la distanță. Asemenea celor prezentate anterior, și în această nișă secundară se observă împărțirea ofertelor, produselor și tehnologiilor oferite pe piață către diferite grupuri țintă. Există produse oferite de companii, inclusiv de către producători de autovehicule, care pun la dispoziție o aplicație mobilă pentru utilizatorii fizici permițându-le monitorizarea și chiar controlul de la distanță al mașinii proprii. Acest sistem de monitorizare și control este cel mai des întâlnit la mașinile electrice, care au parte de popularitate aflată în plină creștere în acest moment. De cele mai multe ori, pentru a avea acces la o astfel de aplicație, proprietarul mașinii electrice trebuie să își creeze un cont de utilizator, iar apoi mașina se asociază cu acest cont și va transmite date către un server central. La aceste date are acces proprietarul și poate să verifice starea bateriei, fluxul de încărcare, și alte informații utile.

2.1 Bluelink

Un exemplu de comerciant care oferă un astfel de sistem este Hyundai. Producătorul sud-coreean oferă aplicația Bluelink disponibilă sistemele de operare Android și iOS care este compatibilă cu vehiculele oferite de ei. Indiferent de modelul ales, fie că motorul are o arhitectură cu combustie internă, mild-hybrid, PHEV (Plug-In Hybrid Electric Vehicle) sau EV (Electric Vehicle), aplicația oferă funcționalități folositoare utilizatorilor. Prin intermediul acesteia, proprietarul poate să deschidă sau să închidă mașina, să fie notificat de declanșarea alarmei sau să urmărească datele și statusul mașinii. Modelele electrice cum ar fi cele din gama IONIQ sau KONA beneficiază de funcționalități extra în aplicația Bluelink față de cele PHEV sau IC (Internal Combustion – combustie internă). Proprietarul mașinii electrice este capabil să urmărească nivelul de încărcare al bateriei, să seteze un orar al încărcării, pentru ca mașina să fie pregătită de drum anticipat, sau chiar să seteze temperatura din habitaclu, să pornească încălzirea sau ventilația din scaune și chiar dezghețarea parbrizului sau a lunetei pe timp de iarnă. De asemenea, utilizatorul poate să vadă în timp real pe hartă care este distanța maximă pe care vehiculul o poate parcurge cu nivelul respectiv de încărcare al bateriei. Produsul pe care Hyundai îl oferă are ca interfață o aplicație și nu un webserver, dar ideea de monitorizare de la distanță al parametrilor mașinii (inclusiv controlul anumitor funcționalități) este prezentă, fapt care merită prezentat. Aceste funcționalități dar și altele sunt descrise la referința [2].

Fig. 2.1.1. Imagine cu prezentarea Bluelink preluată din [2]

În schimb, în această nișă secundară se vizează și comerțul de tip B2B (Business To Business) în sensul în care companiile care pun la dispoziție sisteme de monitorizare al

poziției prin GPS, parametrilor mașinii, stilul de condus sau chiar gradul de atenție al șoferului, au ca și clienți firmele de transport sau curierat, care își desfășoară activitatea cu ajutorul unei flote de vehicule. În acest caz, persoana care monitorizează parametrii vehiculului, a multiple vehicule din flotă sau chiar a întregii flote, este o persoană aflată la sediul firmei al cărei rol este acesta. Adăugarea acestei funcționalități într-un sistem de flote de vehicule poate însemna implementarea unui program de mentenanță predictibil care va permite vehiculelor să opereze în parametrii optimi, dar și economisirea de bani și resurse alocate.

2.2 Azuga

Un exemplu în acest sens este produsul oferit de compania Americană Azuga, care pune la dispoziția firmelor de transport un sistem de monitorizare de la distanță al flotei de vehicule. Acest produs furnizează informații detaliate despre performanța vehiculului, locația și comportamentul șoferului în timp ce se folosește localizarea GPS în timp real al vehiculului de transport.

Prin intermediul unui dispozitiv care se conectează la portul OBD al vehiculului, acest sistem trimite date precum condiția mecanică, nevoile de întreținere și raportul de utilizare către un server central, iar managerii flotei pot primi alerte imediate despre orice probleme potențiale, așa cum se menționează în [3]. În aceeași referință se detaliază și că sistemul urmărește obiceiurile și stilul de condus ale șoferului, cum ar fi viteza, frânările bruște repetate sau fără motiv întemeiat. Acest sistem de punctare se numește “Azuga Driver Scoring System”și are ca rezultat oferirea recompenselor către șoferii care prezintă un punctaj bun.

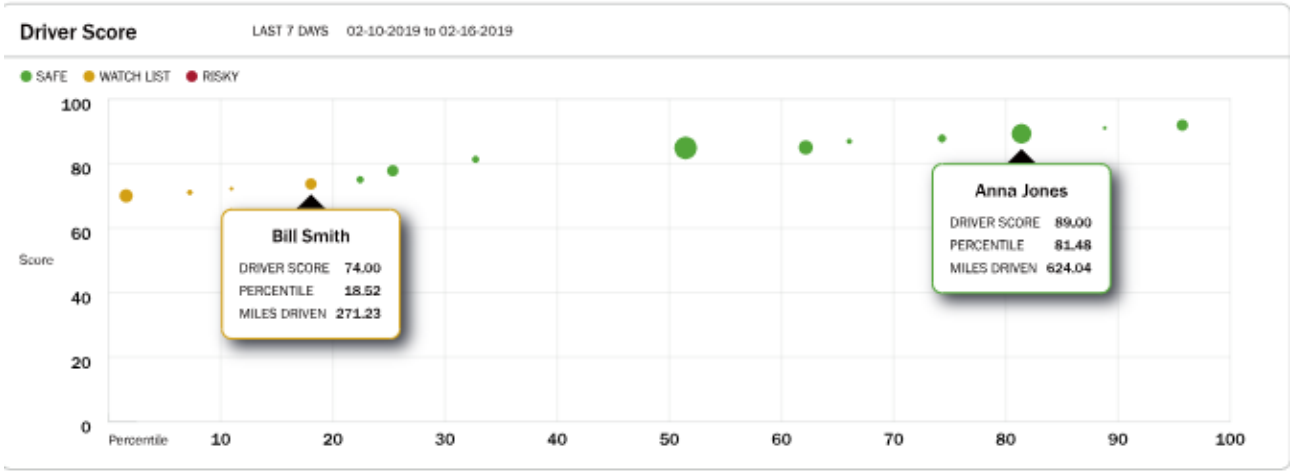


Fig. 2.2.1. Grafic de prezentare al scorului șoferilor, preluat din [3]

Un alt aspect introdus de Azuga este managementul prin cloud al flotei, ceea ce permite managerilor să gestioneze flota de vehicule de oriunde. Dispozitivul de urmărire a vehiculului se conectează la unitatea computerului vehiculului, la sateliții GPS și la Internet. Informațiile captate de pe vehicul și locația GPS sunt trimise într-o formă criptată

către serverele Azuga. Folosind un browser web sau o aplicație mobilă cu o conexiune securizată la Internet, persoanele autorizate (de exemplu managerul flotei) pot accesa aceste date într-un tablou de bord care conține live poziția pe hartă, diagrame și rapoarte care fac posibilă monitorizarea și gestionare vehiculului în cauză. De asemenea, este posibilă citirea datelor despre starea de sănătate a vehiculului, precum și comportamentul specific ale șoferului, cum ar fi stilul de condus, viteza, timpul de mers al motorului la relanti (idle, virajele, frânarea sau accelerația.

Fig.2.2.2. Ilustrare pagina monitorizare, preluată din [3]

2.3 OVMS (Open Vehicle Monitoring System)

OVMS (Open Vehicle Monitoring System) este un sistem de monitorizare al mașinilor care poate fi utilizat atât pentru o singură mașină, dar și pentru o flotă de autovehicule. Asemenea sistemului furnizat de Azuga, la nivel de hardware este necesar un dispozitiv care se conectează la portul OBD-II al mașinii de dimensiuni reduse. Initial a fost dezvoltat pentru vehiculele electrice, în special Tesla Roadster, dar și-a extins compatibilitatea pentru a include o gamă largă de vehicule. Sistemul este construit în jurul unei arhitecturi modulare, permițând utilizatorilor să-și personalizeze și să-și extindă funcționalitatea în funcție de nevoile lor specifice. Acest sistem are capabilități de transmitere al datelor OTA (Over The Air) printr-un modem 2G/3G/4G și este certifica FCC și CE.

OVMS este format din componente hardware și software. Modulul hardware este instalat în vehicul și se conectează la computerul principal al mașinii și la portul de diagnosticare la bord (OBD) al vehiculului. Acest modul este responsabil pentru colectarea datelor despre parametri precum starea bateriei, temperatura, viteza și locația GPS. Datele sunt apoi transmise prin rețele celulare sau WiFi către serverul OVMS, unde pot fi accesate printr-un portal web sau o aplicație mobilă. Componenta software include infrastructura server-side, precum și aplicațiile client pentru diferite platforme.

Sistemul OVMS este open source, disponibil pe GitHub [4] iar acest fapt este un element care încurajează dezvoltarea și inovația conduse de comunitate. Utilizatorii și dezvoltatorii de software pot contribui la proiect adăugând noi funcții, îmbunătățind funcționalitățile existente și asigurând compatibilitatea cu modelele noi de vehicule. Această abordare colaborativă a condus la un sistem robust și complex, care continuă să evolueze. În plus, modelul open-source permite o mai mare transparență și securitate, deoarece baza de cod poate fi revizuită și auditată de oricine. Planurile PCB-ului (Printed Circuit Board) și schema electronică pot fi consultate și studiate în repository-ul GitHub [4].

În acest sens, prin lucrarea de față îmi propun documentarea construirii unui sistem asemănător cu proiectele prezentate anterior, proiect care este intitulat “CarMonitor”, și care are ca scop monitorizarea fără fir a parametrilor mașinii, în primă fază în proximitatea vehiculului iar apoi disponibilizarea acesteia în manieră remote, indiferent de locația mașinii.

3. ARHITECTURA SISTEMULUI CARMONITOR

3.1 Componente necesare

Datorită specificațiilor unui sistem de monitorizare al parametrilor mașinilor, trebuie ales un PCB dotat cu un microcontroller care are atât capabilități de comunicare prin Bluetooth, dar și capabilități WiFi care să permită susținerea unui server web. De aceea, cea mai potrivită componentă s-a considerat că este un Espressif 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.

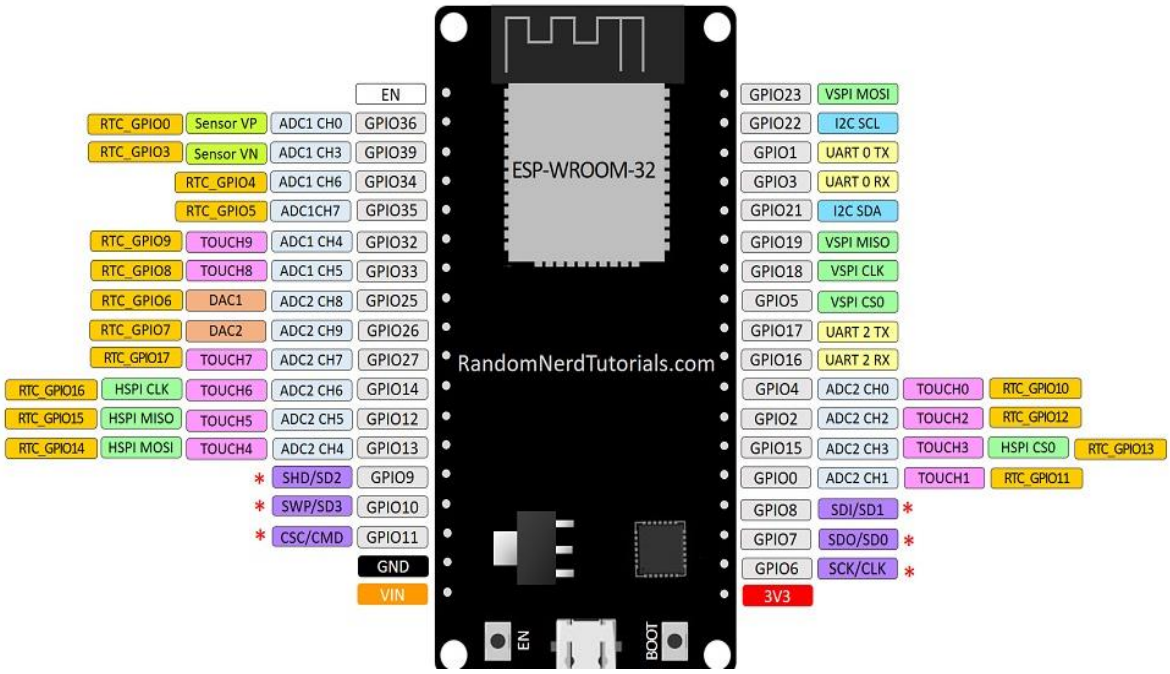


Fig. 3.1.1. ESP-32. Imagine preluată din [5]

PCB-ul folosit care are System on Chip (SoC) de tip ESP-32 este unul ESP32-WROOM-32, care are 2 MiB memorie flash, rulează sistemul de operare FreeRTOS, 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 [6]. 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 și are performanțe mai satisfăcătoare. 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 [6]. 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 (pusă la dispoziție de FreeRTOS), datorită simplității și versatilității pe care această soluție o prezintă.

Pentru capabilitățile de comunicare prin Bluetooth cu ECU al mașinii este nevoie de o componentă robustă care să acționeze ca un interlocutor între ECU și ESP-32. Este nevoie de un cititor OBD-II, dotat cu un chip ELM327, capabil de a recepționa și transmite date în mod bidirecțional. În acest sens s-a ales cititorul OBD furnizat de compania VGate, mai exact iCar Pro V2. Acest cititor are calități importante care asigură versatilitatea, integritatea datelor, și o compatibilitate largă cu alte produse ceea ce îl face alegerea potrivită pentru un astfel de sistem.

Fig.3.1.2. VGate iCar Pro V2

3.2 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.

Fig. 3.2.1. Car Monitor

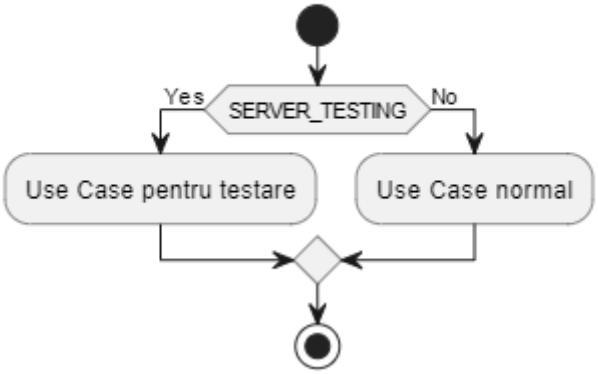


Fig. 3.2.2. Funcția setup()

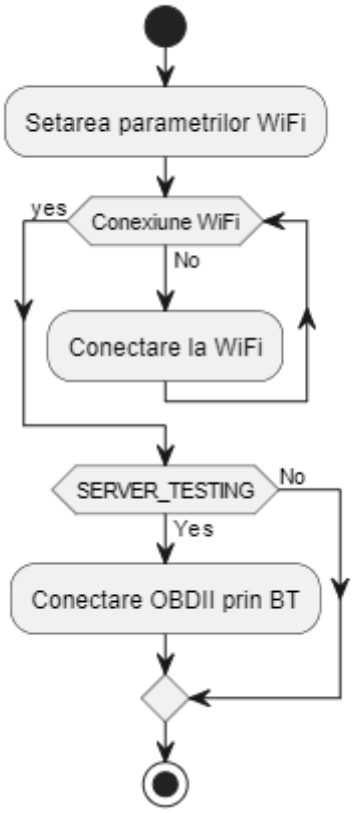
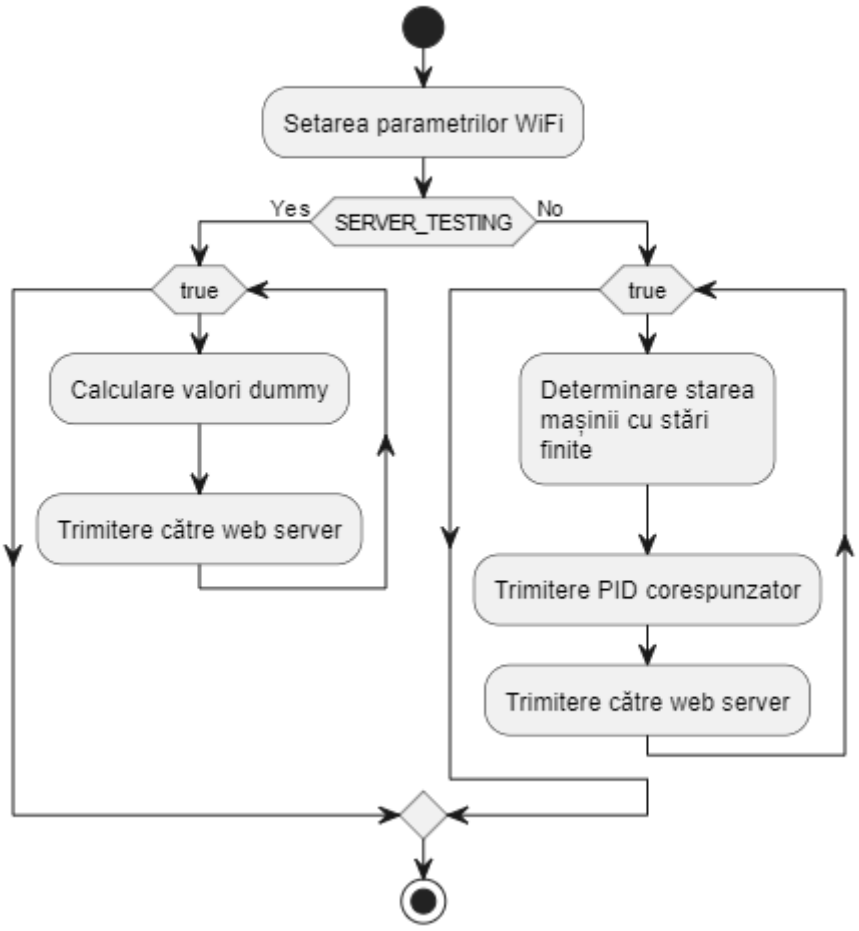


Fig. 3.2.3. Funcția loop()



3.3 Arhitectura modulului 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 [7].

Pentru Bluetooth LE (Low Energy), așa cum este descris în [7], 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.

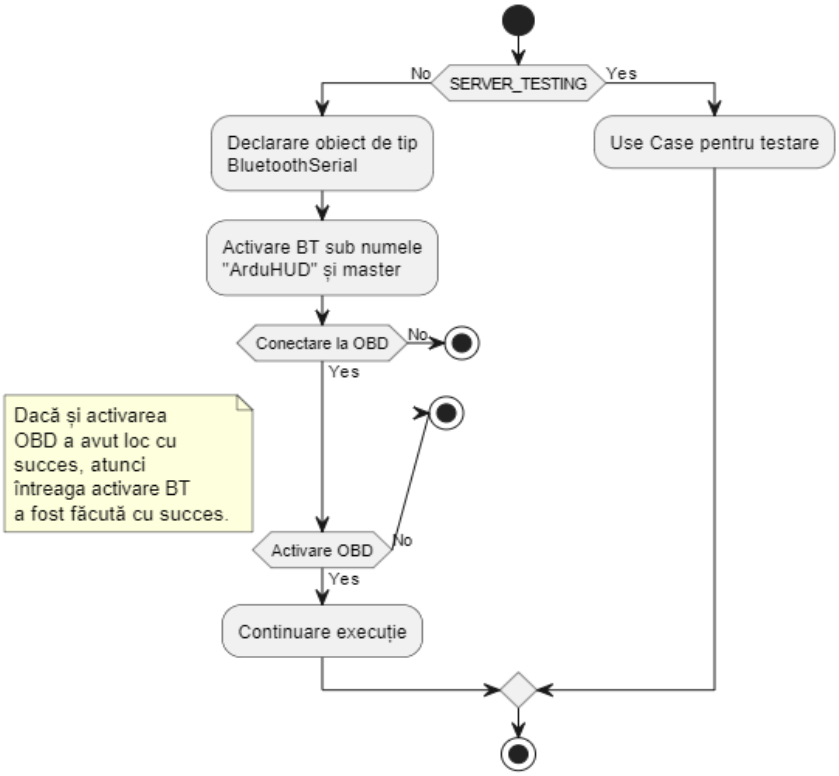
În cazul sistemului “CarMonitor”, protocolul Bluetooth folosit este cel Classic, deoarece există un flux constant de date care trebuie procesate, în urma comunicării plăcii ESP-32 cu cititorul OBDII. Placa trimite prin Bluetooth PID-uri în urma cărora OBD-ul răspunde cu datele cerute. De asemenea, la 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 3.3.1.

Fig. 3.3.1. Activare Bluetooth



3.4 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 asigură 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 prezintă în [7], 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.

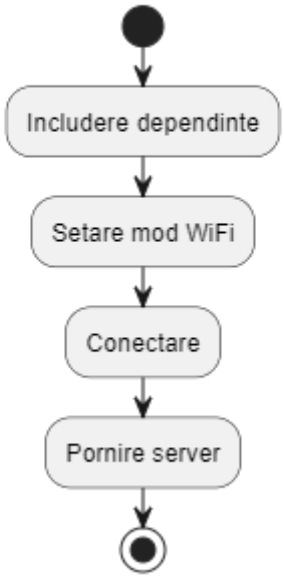
În cod, modulul WiFi nu depinde de un macro, acesta fiind activat mereu, iar prin intermediul lui, web serverul este activ la fiecare rulare. Includerea metodelor WiFi se fac prin fișierul `<WebServer.h>`. De asemenea, în realizarea sistemului, placa a fost conectată la un hotspot mobil al unui smartphone. 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.

Fig. 3.4.1. Activare Web Server



Acest API este disponibil open-source pe GitHub [8] 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 observă o îmbunătățire clară atunci când este introdus în fișierul de build disponibil în `PlatformIO`. Motivul din spatele acesteia este în strânsă legătură cu punctul de vedere al scalabilității, deoarece sistemul face uz de toate resursele plăcii într-un mod cât mai eficient, iar timpii de afișare sunt îmbunătățiți. De aceea, această dependința este foarte importantă și este potrivită pentru sistemele IoT, inclusiv pentru sistemul “CarMonitor”.

3.5 Arhitectura sistemului

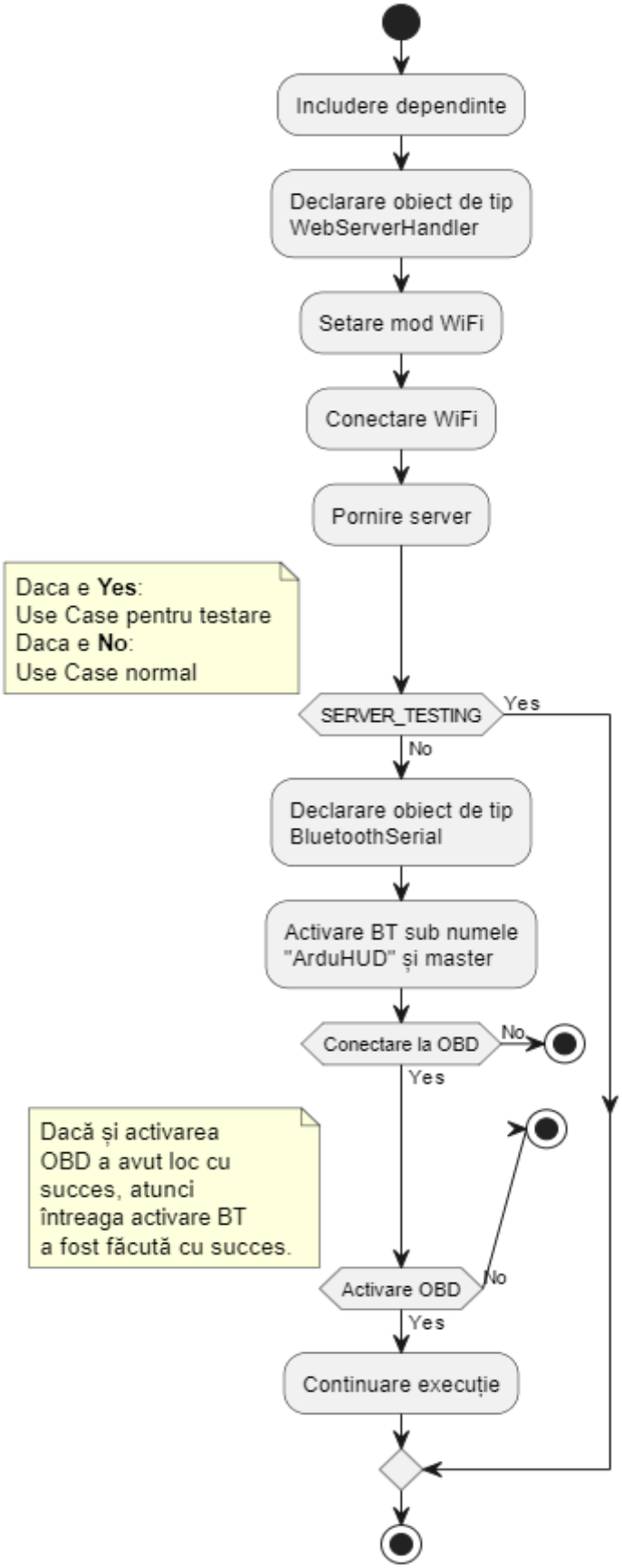
Având descrise cele două module care compun sistemul, se poate realiza arhitectura întregului sistem prin imbricarea celor două. Ordinea în care cele două se rulează este întâi configurarea serverului, apoi configurarea Bluetooth. Ulterior, în funcție de macro, se selectează use case-ul dorit.

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 întâlnește 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 în km/h), `ENG_RPM` (turația), `VOLTAGE` (tensiunea bateriei), `THROTTLE` (cât % este apăsată pedala de accelerație) , `ENG_COOLANT` (temperatura în Celsius al lichidului de răcire), `LOAD` (cât % este solicitat motorul), `FUEL_LEVEL` (cât % mai este combustibil în rezervor), `OIL_TEMP` (temperatura uleiului din blocul motor în Celsius).

Fig. 3.5.1. Arhitectura sistemului



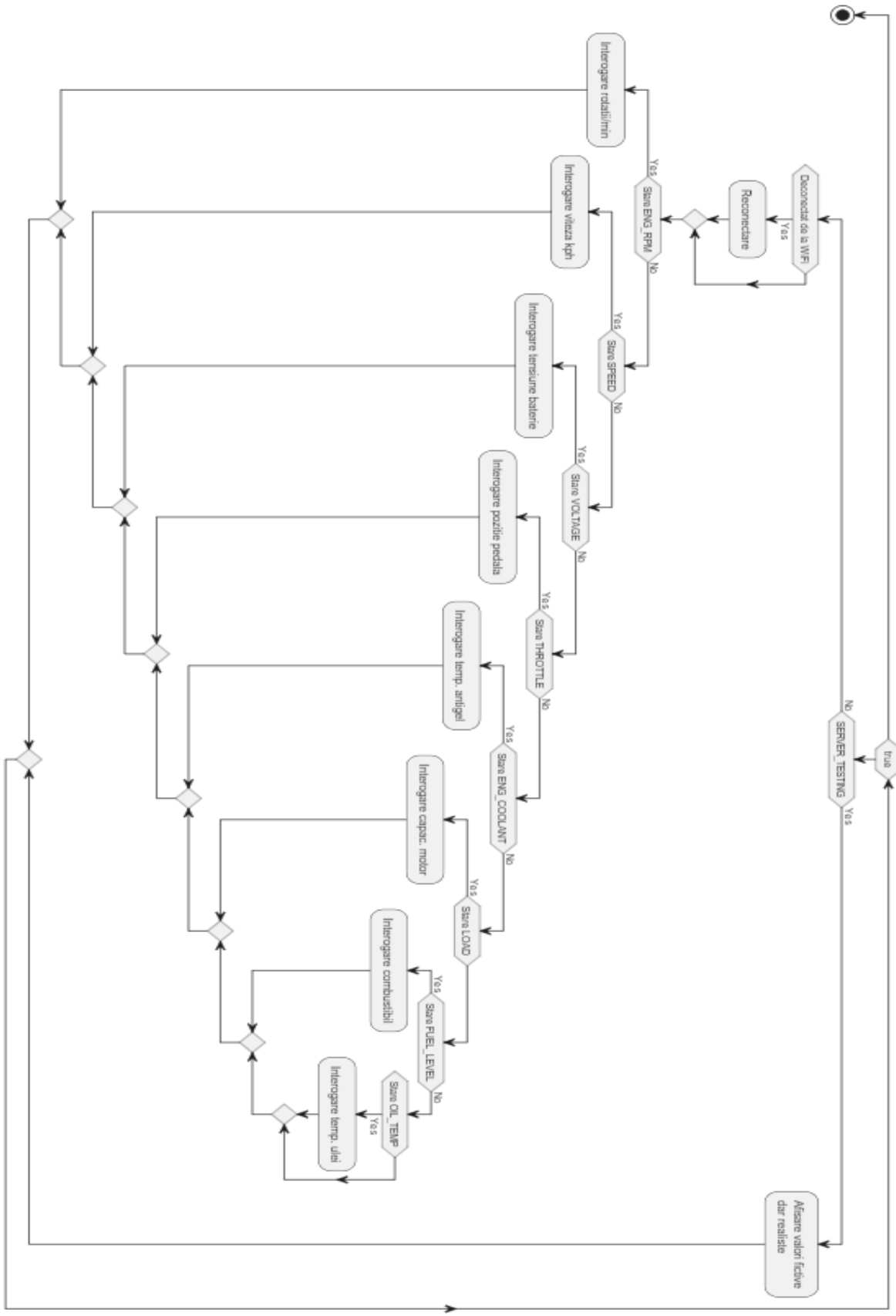


Fig. 3.5.2. Arhitectura sistemului

3.6 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 [9].

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 în web serverul găzduit de către ESP-32. Folosind acest sistem, managementul fișierelor este mai simplu, iar afișarea este cu mult optimizată pentru aplicația de tip IoT care necesită rulare în timp real. Actualizarea fișierelor statice se poate realiza mult mai rapid, iar rescrierea întregului sistem nu este necesară, fapt care ajută la economisirea timpului și al resurselor.

4. MODUL DE IMPLEMENTARE

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, dar și furnizarea de statistici asupra proiectului.

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 semnifică 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.

4.1 Implementarea modulului 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.

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

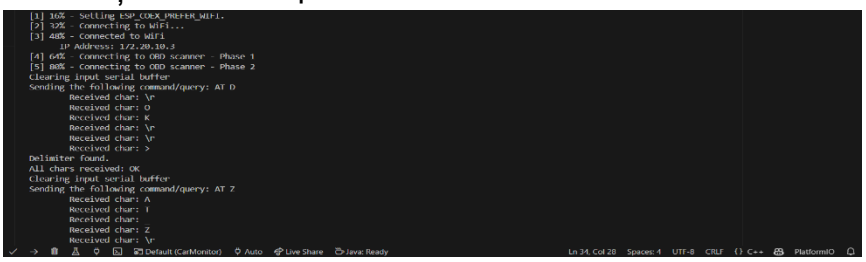


Fig 4.1.1 Output terminal

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:

```
DEBUG_PORT.println("[5] 80% - Connecting to OBD scanner - Phase 2");
if (!ELM327Reader.begin(ELM_PORT, true, 2000)) // 2nd param: bool for debug prints
{
    DEBUG_PORT.println("[5] 80% - ERROR - Couldn't connect to OBD scanner - Phase 2");
    while (1)
        ;
}
```

Fig. 4.1.2 Secvență de cod.

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)
```

Fig. 4.1.3 Secvență de cod

care folosit pentru a crea un Bluetooth event și pornire modul.

```
bool BluetoothSerial::connect(String remoteName)
```

Fig. 4.1.4 Secvență de cod

care 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)
```

- *Fig. 4.1.5 Secvență de cod*

folosit pentru a returna numărul de mesaje disponibile în coadă (funcție facută efectiv de uxQueueMessagesWaiting(_spp_rx_queue);)

```
int BluetoothSerial::read()
```

- *Fig. 4.1.6 Secvență de cod*

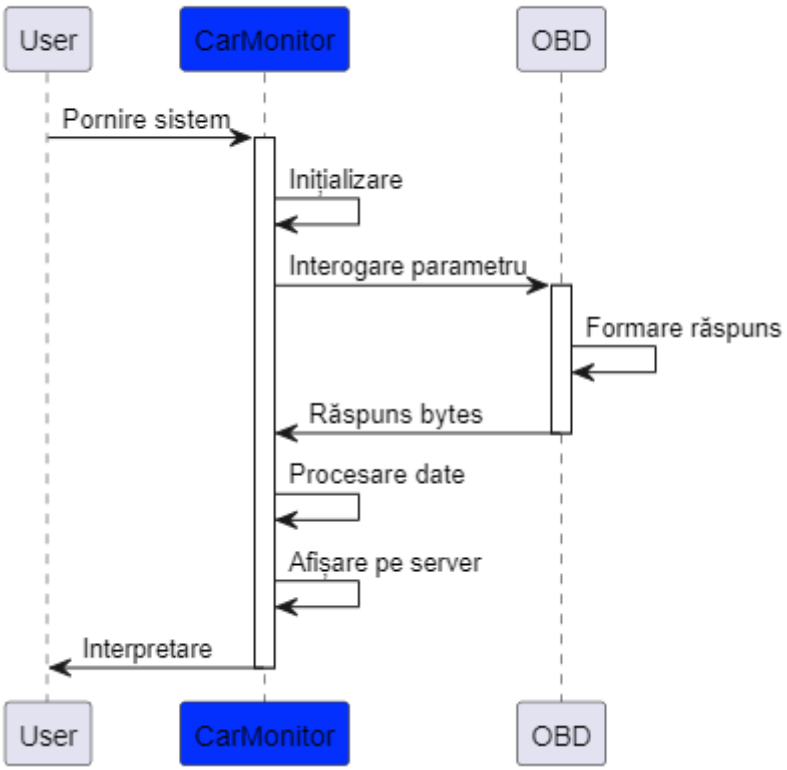
folosit pentru a citi din buffer valoarea primită.

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.

4.1.1 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.

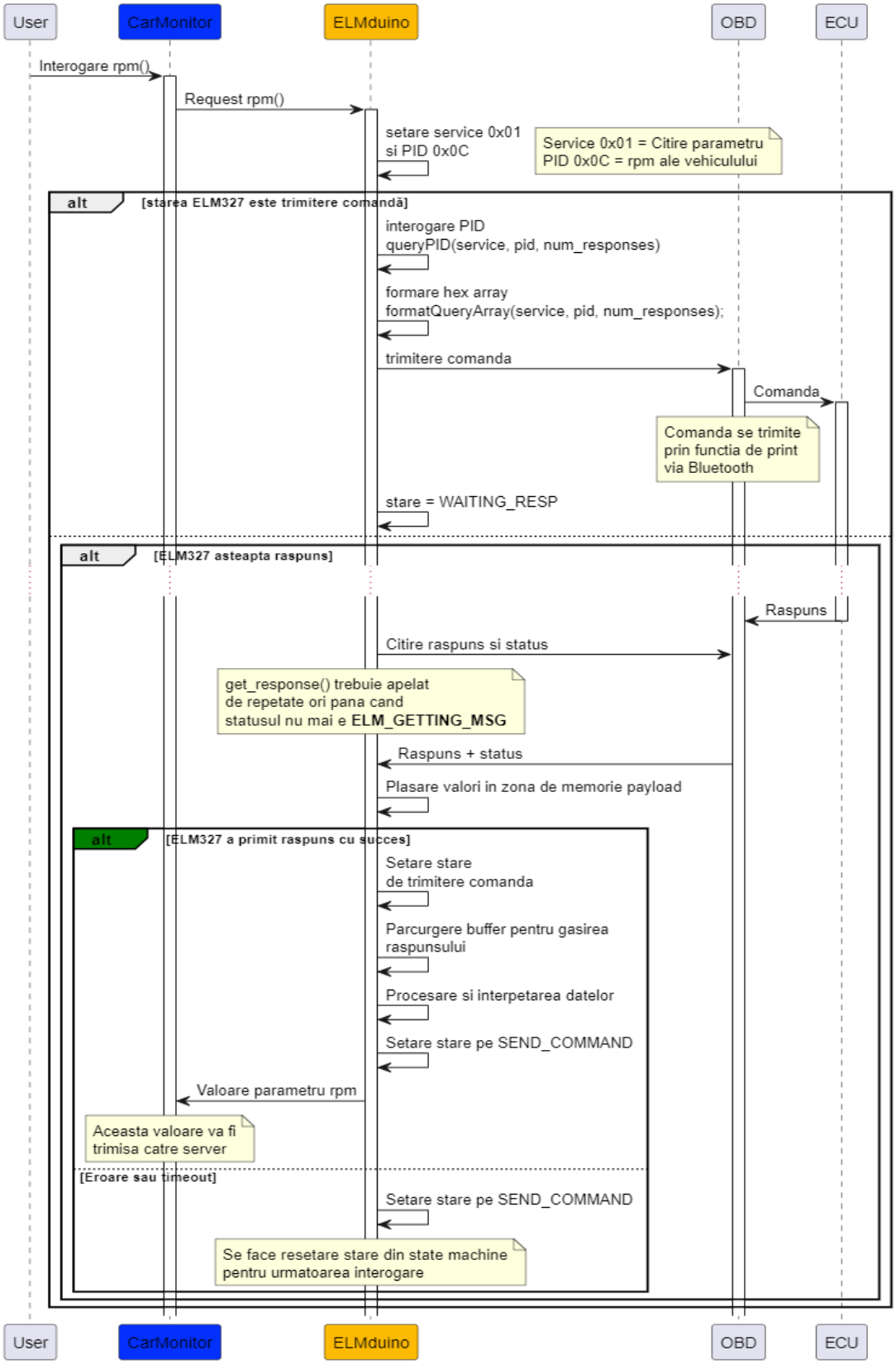
Fig. 4.1.1.1. Diagrama de secvență dinamică



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.

4.1.2 Interogarea parametrilor prin API-ul ELMduino

Fig. 4.1.2.1. Diagrama de secvență pentru interogarea rpm



ELMduino este un API folosit în acest proiect, disponibil de asemenea pe GitHub [10], 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 [10], librăria 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 trimis în starea anterioară.

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, și multe altele. De asemenea, prin funcția `void resetDTC()`, se șterg erorile prezente în ECU. Această funcție 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 începe 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 $\frac{100}{255}A$, 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 $A - 40$.

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 $\frac{100}{255}A$. 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 $A - 40$.

4.1.3 Diagrama de secvență pentru CarMonitor

Fig. 4.1.3.1. Diagrama de secvență pentru CarMonitor

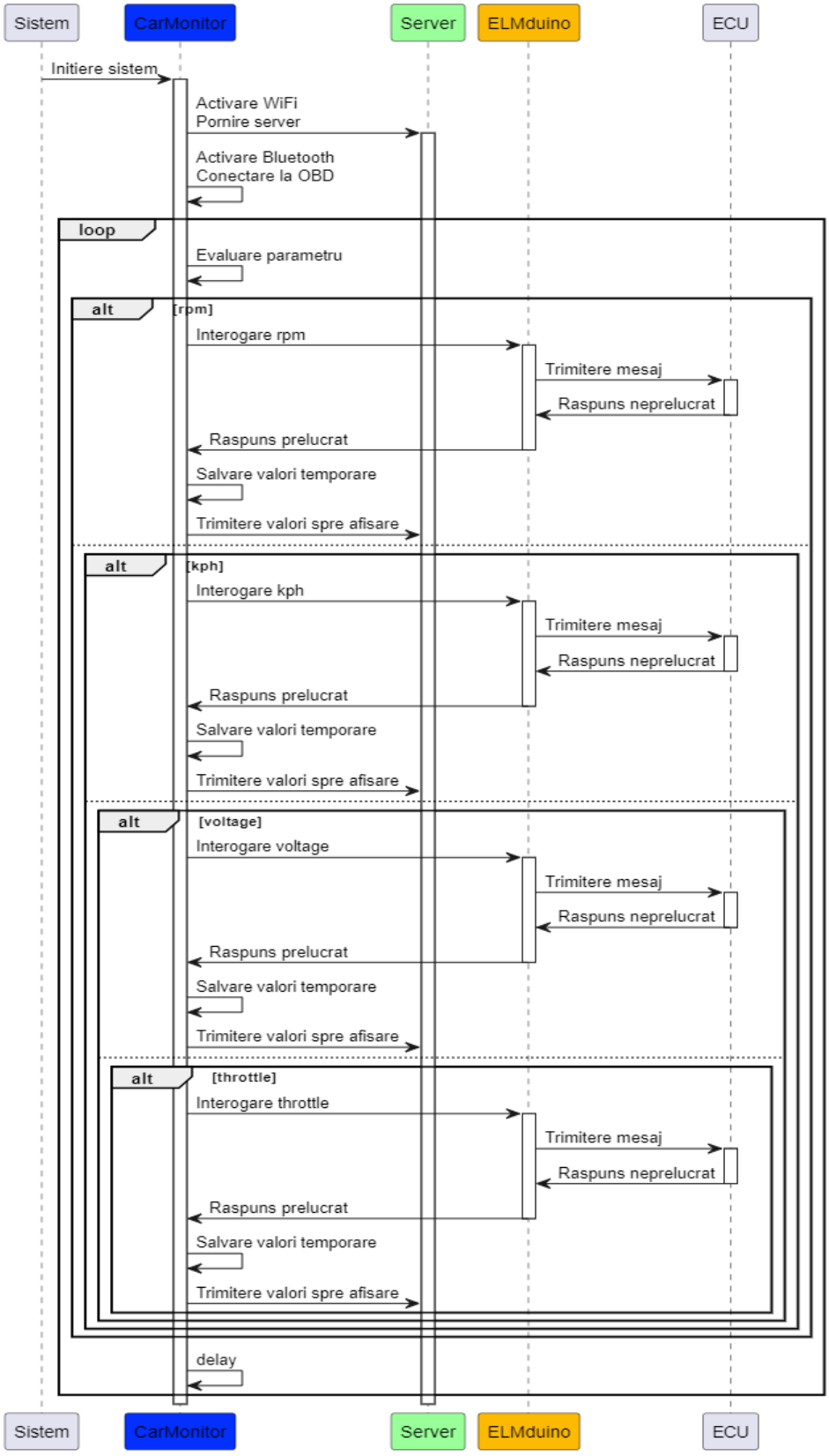
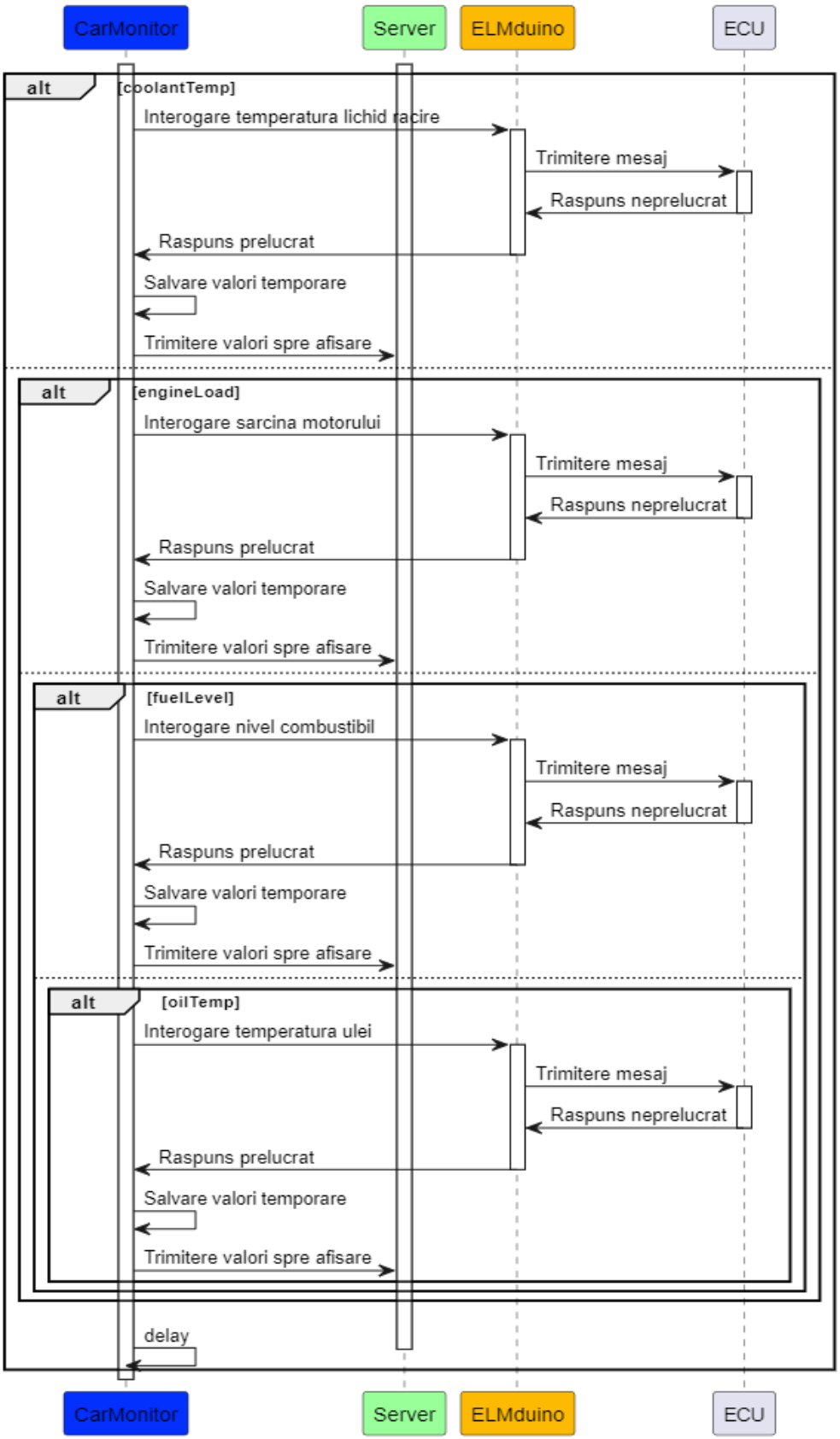


Fig. 4.1.3.2. Diagrama de secvență pentru CarMonitor



4.2 Implementarea serverului de monitorizare

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.

În cazul sistemului "CarMonitor", s-a folosit protocolul de securitate `WPA3-SAE`, activat de flag-ul `DCONFIG_ESP_WIFI_ENABLE_WPA3_SAE`, deoarece în 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 `platformio.ini` la secțiunea `build_flags`.

```
17  build_flags =
18    -DCONFIG_ESP_WIFI_ENABLE_WPA3_SAE ;for ios connection to hotspot
19    -DCONFIG_SW_COEXIST_ENABLE ;for coexistence between wifi and bluetooth
```

Fig. 4.2.1 Secvență de cod

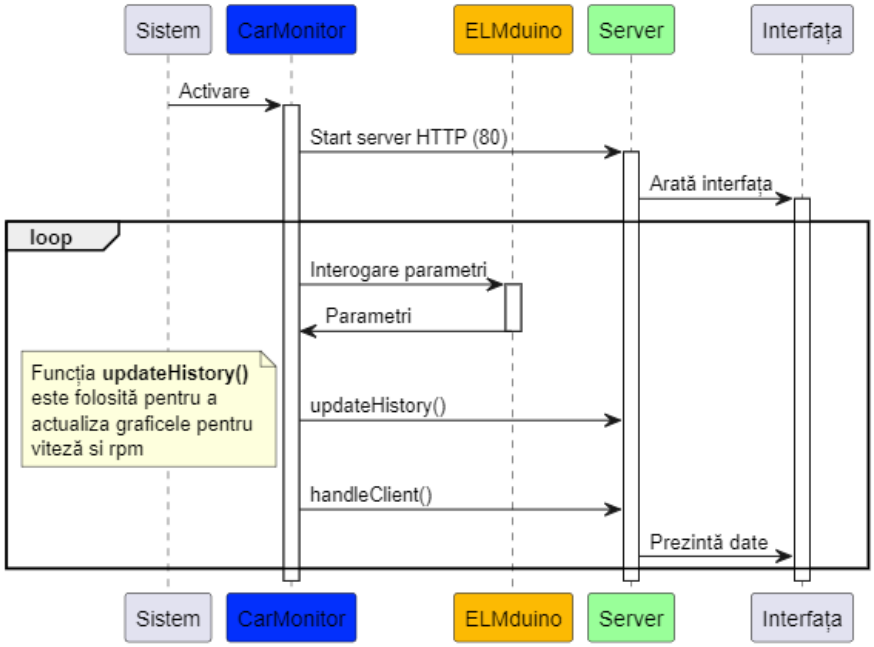
Tot în fișierul `platformio.ini`, la secțiunea `lib_deps`, se specifică dependințele pentru proiectul PlatformIO. Acesta, pe lângă dependința ELMduino, va descărca și `ESP Async WebServer`, care are scopul de a crea un web server HTTP asincron.

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 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.

4.2.1 Diagrama de secvență dinamică

Fig. 4.2.1.1. Diagrama de dinamică pentru WebAsyncServer



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.

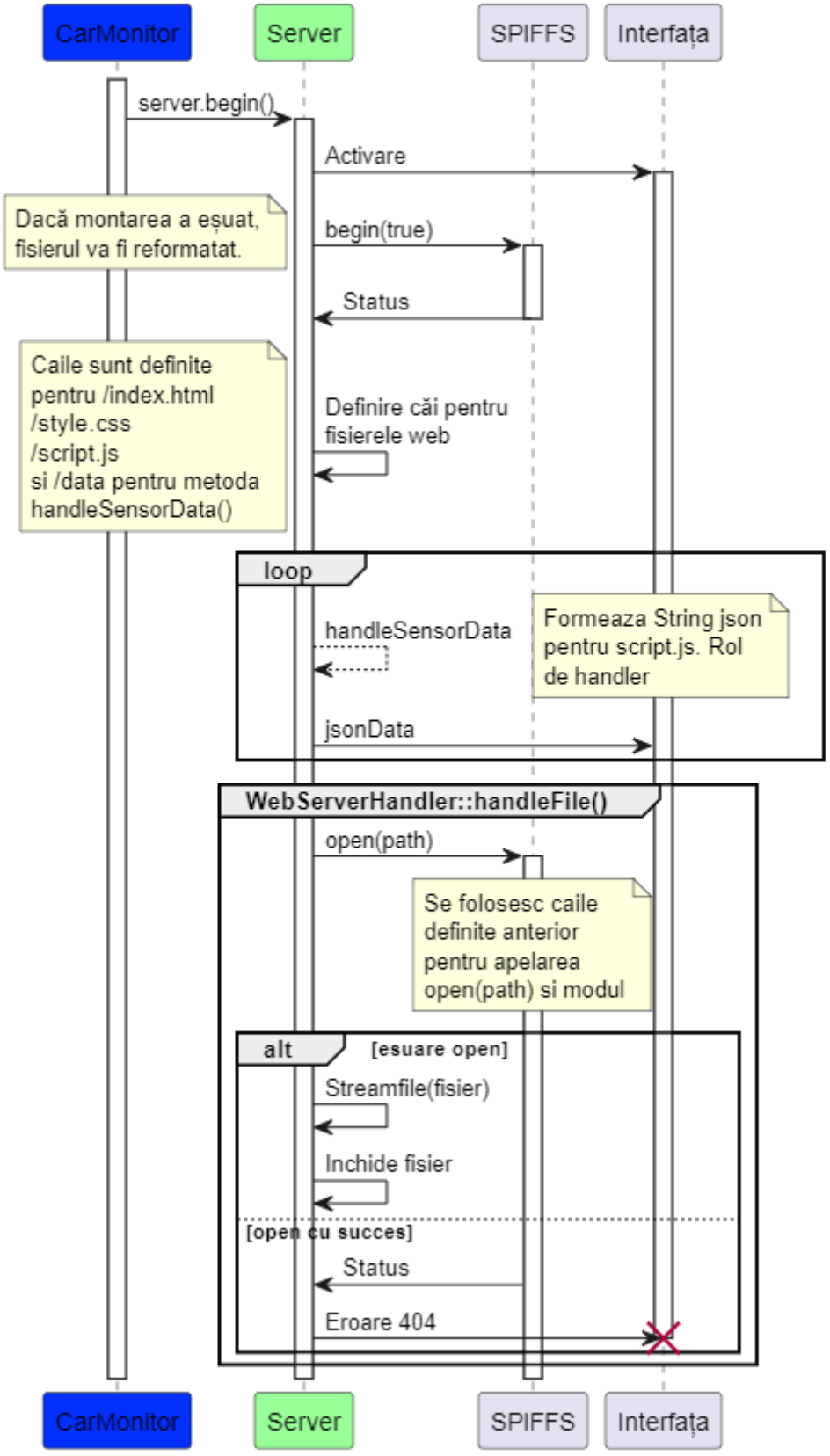
Acesta apoi începe procesul de conectare și comunicare cu cititorul OBD prin API-ul ELMduino, iar după ce primele date sunt disponibile, sunt afișate în interfață prin funcția `updateHistory()` și metoda `handleClient()`.

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.

4.2.2 Diagrama de secvență pentru WebServer.begin()

Fig. 4.2.2.1. Diagrama de secvență pentru metoda begin()



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.

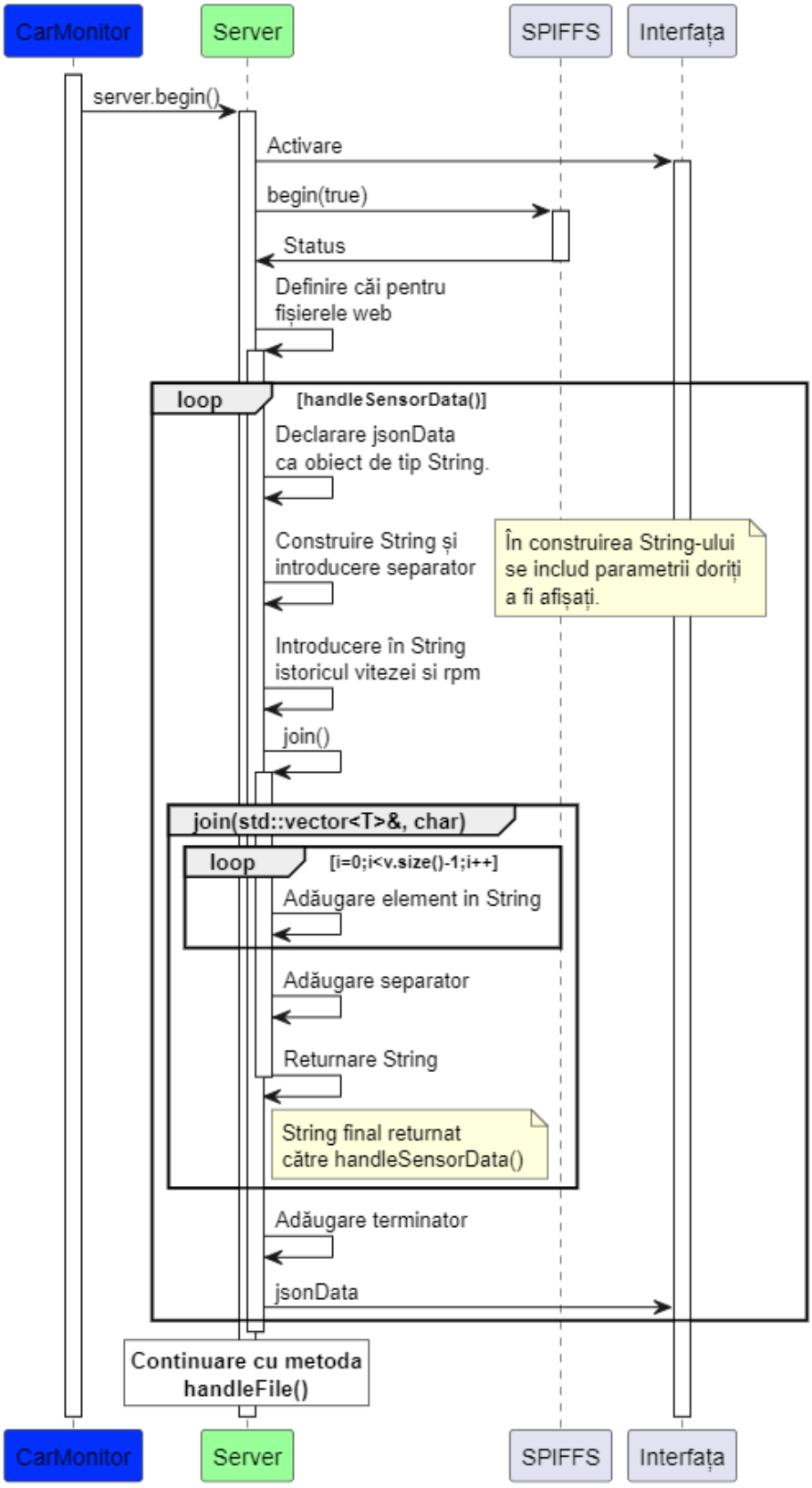
```
File file = SPIFFS.open(path, "r");
```

Fig. 4.2.2.2 Secvență de cod

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.

4.2.3 Diagrama de secvență pentru handleSensorData()

Fig. 4.2.3.1 Diagrama de secvență pentru metoda handleSensorData()



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.

4.3 Interfața cu utilizatorul

4.3.1 Introducere

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.

4.3.2 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 [11], 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 [12].

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>
```

Fig. 4.3.2.1 Secvență de cod

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>
```

Fig. 4.3.2.2 Secvență de cod

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>
```

Fig. 4.3.2.3 Secvență de cod

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>
```

Fig. 4.3.2.4 Secvență de cod

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>
```

Fig. 4.3.2.5 Secvență de cod

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>
```

Fig. 4.3.2.6 Secvență de cod

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>
```

Fig. 4.3.2.7 Secvență de cod

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>
```

Fig. 4.3.2.8 Secvență de cod

4.3.3 Fișierul CSS (Cascading Style Sheets)

Conform [13], 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.

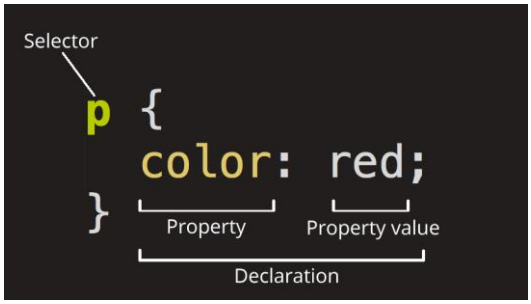


Fig. 4.3.3.1 Descriere selector [13].

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`.

```css
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;
}
```

Fig. 4.3.3.2 Secvență de cod

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.

4.3.4 Fișierul JS (JavaScript)

Așa cum se descrie în [14], 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 [15]. 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 [14], 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 terți (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:

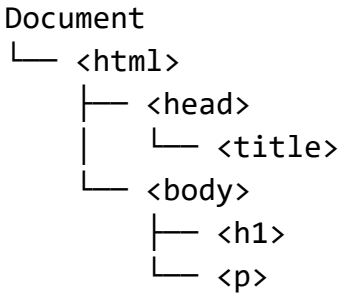


Fig. 4.3.4.1 Exemplu fișier reprezentat în DOM.

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
    }
};
```

Fig. 4.3.4.2. Secvență de cod.

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)
);
```

Fig. 4.3.4.3. Secvență de cod.

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
    }
}
```

Fig. 4.3.4.4. Secvență de cod.

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));
}
```

Fig. 4.3.4.5. Secvență de cod

.

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
});
```

Fig. 4.3.4.6. Secvență de cod

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.

4.4 Interfața vizuală

În continuare se prezintă screenshot-uri cu valorile afișate pe web server în timpul rulării efective al codului atât în timp ce se execută afișările valorilor fictive (care au rol de a testa interfața), cât și al autovehiculul în timp ce acesta rulează. Valorile de test sunt prezentate:

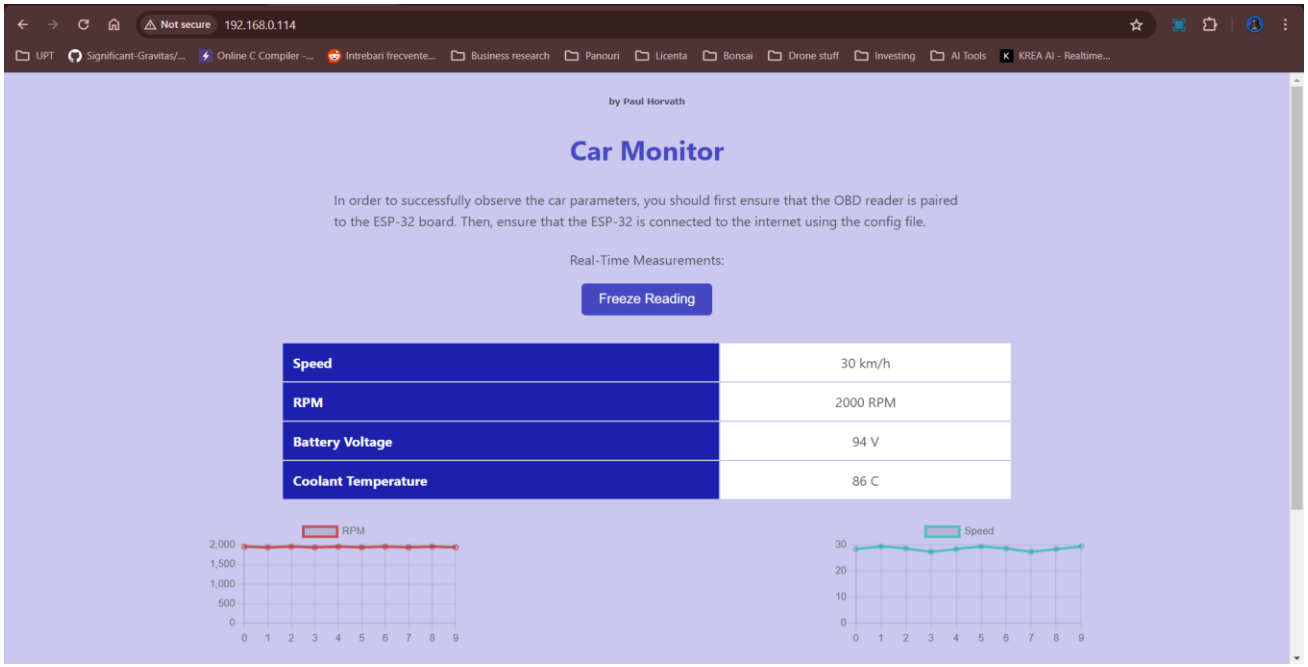


Fig. 4.4.1 Interfață vizuală – valori de test

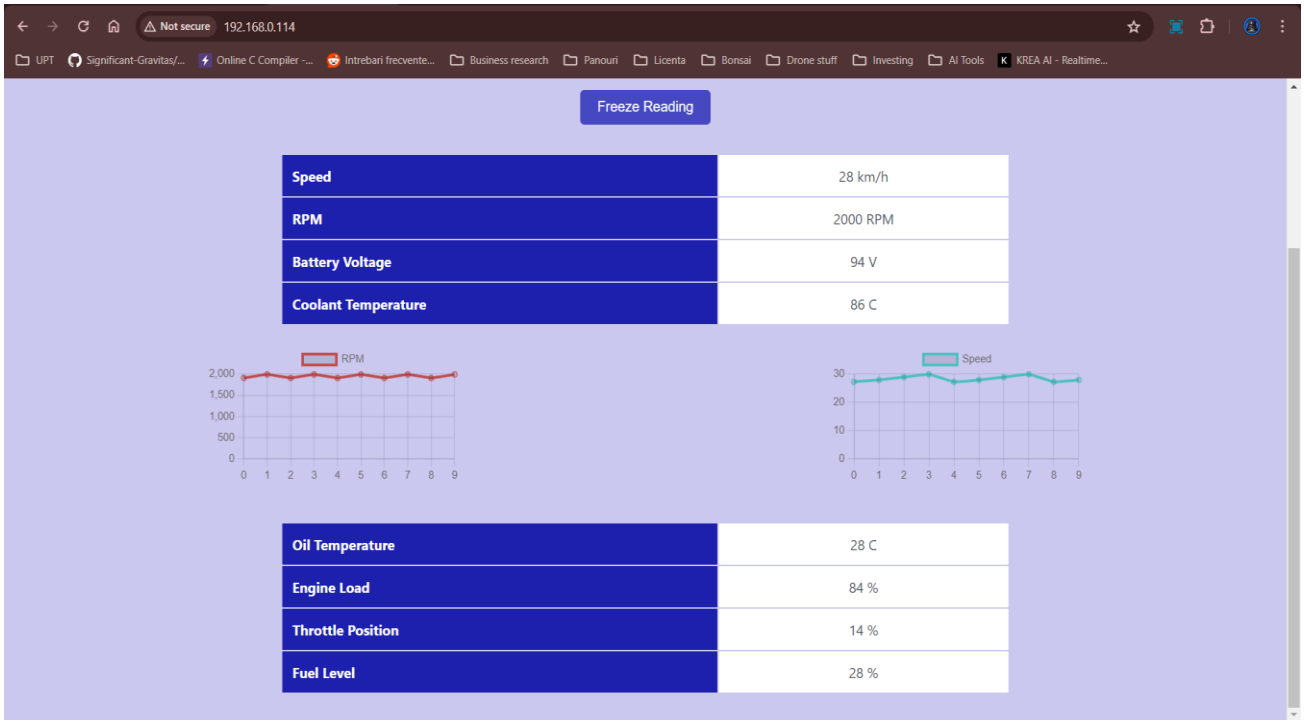


Fig. 4.4.2 Interfață vizuală – valori de test

Fig. 4.4.3 Interfață vizuală – valori reale afișate în timpul rulării

4.5 Testare și configurare

În testarea acestui proiect s-a folosit placă de dezvoltare ESP32-WROOM-32 care are, așa cum s-a menționat în introducere, 2MB Flash și 320KB memoria Random Access Memory (RAM). De asemenea, 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

Sistemul întreg este construit folosind script-urile disponibile în PlatformIO, mai exact “Build” folosit pentru compilare, “Upload” care este folosit la urcarea codului pe placa și “Serial Monitor” folosit pentru a urmări output-ul în consolă al sistemului care rulează pe ESP-32. Folosind sistemul integrat în PlatformIO de analizare a sistemului, acesta în forma lui finală, inclusiv mecanismele introduse pentru excluderea valorilor eronate, ajunge să ocupe 79.5% din memoria flash, și 17.7% din memoria RAM. Aceste calcule sunt furnizate de script-ul de Build incorporat în PlatformIO.

```
RAM:   [==        ]  17.7% (used 57992 bytes from 327680 bytes)
Flash: [========  ]  79.5% (used 1563341 bytes from 1966080 bytes)
```

De asemenea, PlatformIO pune la dispoziție funcția de Inspect care oferă informații despre întreg proiectul, inclusiv posibile adăugări de padding, sau overhead (definit ca „orice combinație de timp de calcul în exces sau indirect, memorie, lățime de bandă sau alte resurse care sunt necesare pentru a îndeplini o anumită sarcină” [16]). Această funcție este ilustrată în interfața următoare:

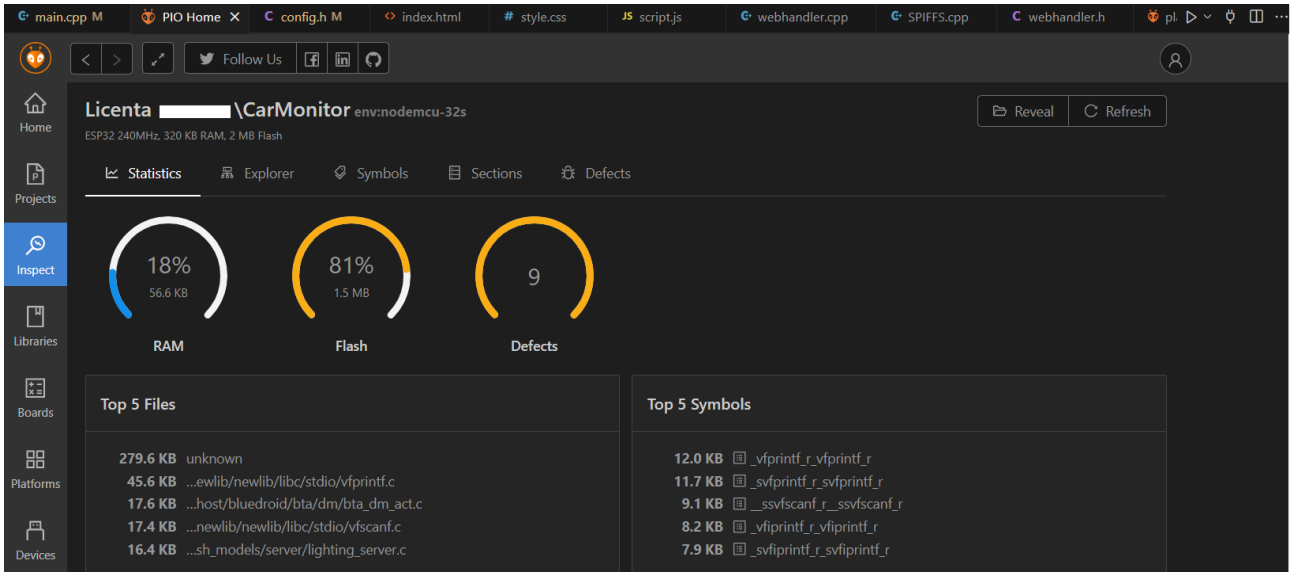


Fig. 4.5.1 Performanțele sistemului CarMonitor furnizate de PlatformIO.

Se observă că dacă se folosește această funcționalitate, sistemul ne spune că memoria RAM ocupată este de 18% (56.6KB) și 81% (1.5MB) memorie Fash ocupată. În plus, se observă semnalarea a 9 defecte, dar cele mai importante dintre acestea fac parte din API-ul ELMduino cu care acest proiect a avut doar o relație de chemare a funcțiilor, nu și de modificare al lor.

În cazul configurării sistemului, există un fișier special în cadrul proiectului care permite alegerea diferitelor use-case-uri. Acesta este un fișier de tip header și se numește `config.h`. În cadrul lui, utilizatorul care configurează sistemul poate alege valoarea lui `SERVER_TESTING`, macro care a fost explicat extensiv în capitolul de implementare detaliată. Acesta, prin punerea lui pe `true` permite testarea web serverului prin completarea câmpurilor cu valori fictive, iar dacă este pus pe `false`, se va adăuga logica efectivă din spatele proiectului “CarMonitor”, unde toate valorile prezentate sunt cele reale și citite de la adaptorul OBD-II via Bluetooth și prelucrate de ELMduino.

Un alt aspect important realizat prin intermediul configurării fișierului config.h il reprezintă alegerea valorilor pentru WIFI_SSID și WIFI_PASS, macro-uri folosite în metoda WiFi.begin() pentru a permite plăcii să se conecteze la rețea.

```
    WiFi.begin(WIFI_SSID, WIFI_PASS);
```

Fig. 4.5.2 Secvență de cod

Tot în legătură cu conexiunea la rețea, s-a implementat codul care permite unui utilizator să își dea seama doar prin mod vizual dacă placa s-a conectat la rețea. Acesta este un mod simplu dar eficace de a urmări progresul funcționalității de bază al plăcii, mai exact chiar prima etapă din logica sistemului “CarMonitor”.

În testarea întregului sistem s-a observat faptul că parametrii returnați de ECU și prelucrați de ELMduino aveau uneori infiltrări de valori false. De exemplu turația motorului mașinii nu era consecventă cu realitatea, uneori valorile erau imposibil de mici, alteori erau conforme cu realitatea, sau chiar erau prea mari, toate acestea întâmplând-se într-un

interval de câteva secunde. De aceea, s-a implementat pentru fiecare parametru interogat un micro-sistem care decide dacă valoarea actuală citită este posibilă, dar în special dacă este reală. S-au introdus verificări adiționale și un mecanism care reține ultima valoare care s-a decis că este reală. Dacă valoarea nou introdusă este prea departe de cea anterioară, aceasta nu este luată în calcul, și se păstrează cea citită înainte. Micro-algoritmul care evaluează din punct de vedere al corectitudinii parametrilor ia în considerare evoluția pe care o poate avea o mașină într-o secundă. În acest sens, spre exemplu, o accelerare de la 10km/h la 180km/h nu este realizabilă într-o secundă, indiferent de mașina monitorizată, cel puțin la momentul scrierii acestei documentații. Fiecare parametru are micro-algoritm-ul propriu care intenționează să urmeze un grafic posibil și continuu de valori reale, fără schimbări bruște de valori, și fără diferențe prea mari între ele. În continuare se exemplifică acest mecanism în cazul parametrului `SPEED`:

```
if(speedVariable <= 220){
            if (ELM327Reader.nb_rx_state == ELM_SUCCESS)
            {
                DEBUG_PORT.print("kph: ");
                DEBUG_PORT.println(speedVariable);
                rpmVariable < 1000 ? speedTemp = 0 : speedTemp = speedVariable;
                int temp = speedTemp - speedVariable;
                abs(temp) > 15 ? speedVariable = speedTemp : speedTemp =
speedVariable;
                delay(100);
                updateHistory(); // Update the history buffers
                server.handleClient();
            }
            else if (ELM327Reader.nb_rx_state != ELM_GETTING_MSG)
            {
                ELM327Reader.printError();
            }
        }
```

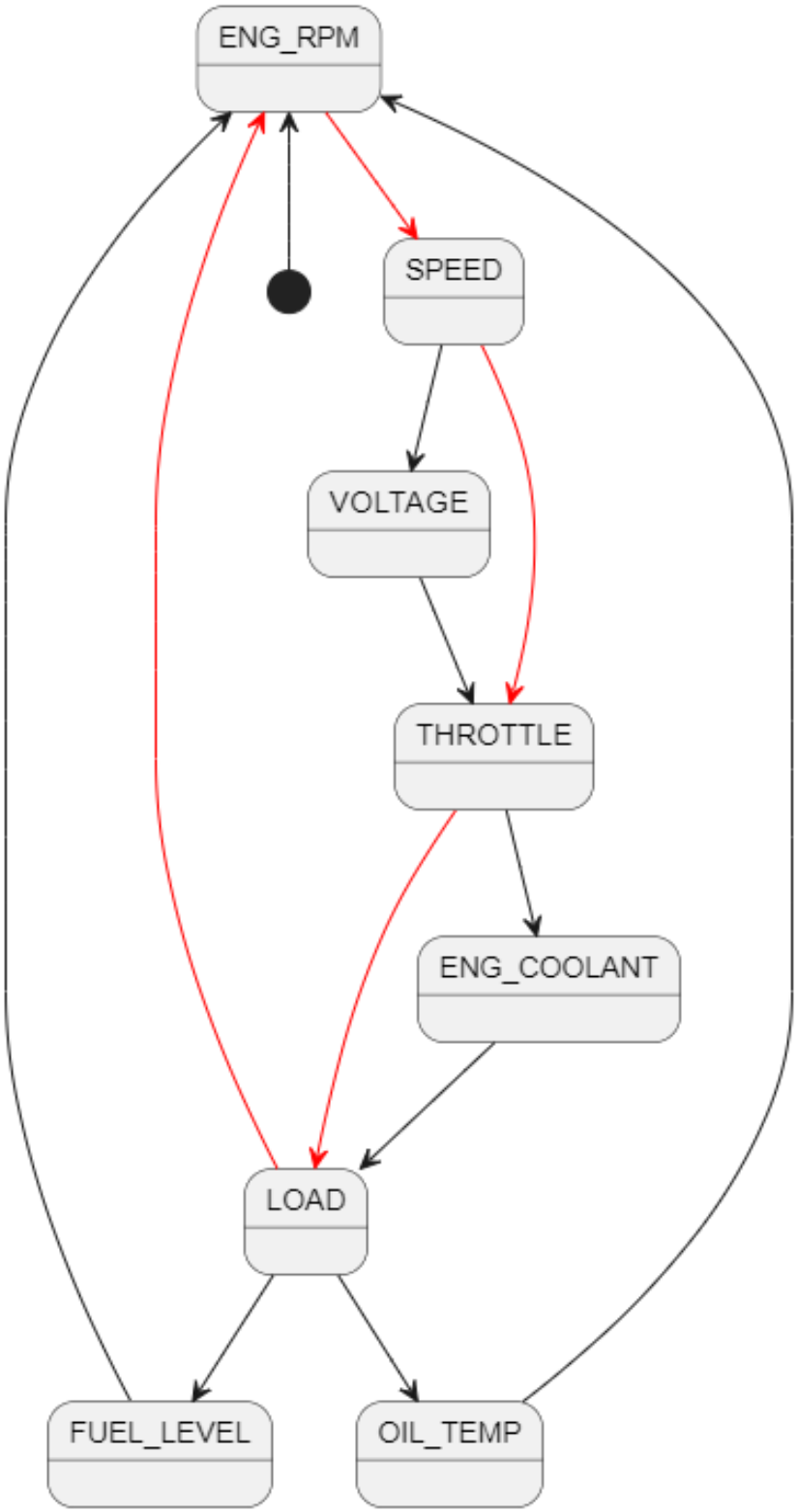
Fig. 4.5.3 Secvență de cod

Un aspect important care a fost implementat îl reprezintă ordinea și frecvența adoptată pentru afișarea parametrilor. Parametrii `ENG_RPM`, `SPEED`, `THROTTLE` și `LOAD` sunt aleși pentru a afișa și interoga mai des ca restul parametrilor deoarece acestea se modifică constant, iar monitorizarea lor în timp real se face la un interval mai mic decât `VOLTAGE`, `ENG_COOLANT`, `FUEL_LEVEL` sau `OIL_TEMP`. În acest sens se prezintă diagrama 5.1.2 care ilustrează modul care stările sunt parcurse pentru a asigura integritatea datelor în funcție de nevoia de interogare al acestora.

Linia roșie reprezintă cazul cel mai des parcurs de către sistem. Aceasta parcurge `ENG_RPM`, `SPEED`, `THROTTLE`, `LOAD` în cele mai multe dintre situații, iar linia neagră este un drum mai lung parcurs de către sistem, odată ce un anumit timp a trecut de când sistemul este pus în funcționare. Stările `VOLTAGE`, `ENG_COOLANT`, `FUEL_LEVEL`, `OIL_TEMP` sunt

parcurse la intervale de timp diferite, mai exact la 2 minute, 1 minut, 5 minute și respectiv 30 secunde, în funcție de importanța lor.

Fig. 4.5.4 Diagrama starilor în procesul de testare al sistemului



5. PREMIZE ANTREPRENORIALE

În acest subcapitol se va descrie în manieră redusă un plan de afaceri ipotetic, care ar putea fi implementat dacă se dorește comercializarea acestui sistem, bineînțeles rafinat, testat, și modelat pentru a satisface nevoile reale ale posibililor clienți.

5.1 Plan Financiar

Obiectivul principal al CarMonitor este de a oferi un sistem cuprinzător de monitorizare a vehiculelor care să permită firmelor care închiriază mașini dar și autoutilitare, să urmărească în mod continuu starea vehiculelor lor. Acest sistem oferă date în timp real despre starea, performanța și utilizarea vehiculelor, iar CarMonitor își propune să ajute companiile de închiriere să minimizeze timpul petrecut la revizii și în service-uri, să optimizeze programele de întreținere și să sporească în general fiabilitatea flotei lor. Acest sistem nu numai că ar putea reduce costurile operaționale, dar îmbunătățește și satisfacția clienților, în timp ce se asigură că vehiculele sunt bine întreținute, bine manevrate, și reduce șansele de apariție al reparațiilor neașteptate.

CarMonitor este conceput special pentru a răspunde nevoilor companiilor de închiriere de mașini. Aceste companii se bazează în mare măsură pe pregătirea operațională și fiabilitatea vehiculelor lor. Utilizând sistemul nostru de monitorizare, firmele de închiriere pot obține informații în timp real asupra stării de sănătate și a stării flotei lor. Aceasta include monitorizarea parametrilor cheie, cum ar fi performanța motorului, tensiunea bateriei, temperatura lichidului de răcire și alți parametri critici. Cu aceste informații, companiile de închiriere pot efectua întreținere proactivă, pot detecta probleme potențiale înainte ca acestea să devină costisitoare și se pot asigura că vehiculele sunt întotdeauna în stare optimă pentru clienții lor.

Pentru a lansa și a opera cu succes CarMonitor, este esențial să existe în spatele sistemului și al afacerii o echipă minimă de angajați dedicați, care să cunoască bine produsul, nevoile clienților, cererile pieței și cele mai noi tehnologii în acest domeniu. În acest sens, pentru o afacere care se află la început, putem defini următoarele roluri cheie care trebuie acoperite.

1 sau 2 Software Developers - Responsabili pentru dezvoltarea și întreținerea firmware-ului pentru ESP32 și asigurarea interfeței cu cititorul OBD dar totodată și de întreținerea paginii web. Acest rol necesită experiență în programarea embedded și cunoștințe de bază în programarea web.

1 Tester – Responsabil pentru testarea și integrarea sistemului cu cititorul OBD și integrarea acestuia în mașina dorită.

1 Persoană pentru Customer Support – Se ocupă cu promovarea CarMonitor dar și de asistență acordată primilor clienți. Pentru această poziție, dacă există posibilitatea se pot adăuga și responsabilități de marketing.

1 Project Manager - Gestionează întregul proiect, coordonează între diferite echipe și se asigură că etapele de dezvoltare și obiectivele de afaceri sunt îndeplinite, important pentru menținerea termenelor și pentru supravegherea ciclului de viață al dezvoltării produsului.

Nr. crt.	Denumirea postului	Experiența solicitată	Studii	Număr ore alocate/luna	Salariul net
1	Software Developer x 2	Minim 6 luni în domeniu	Cunoștințe tehnice în programare, atât embedded cât și web	80 ore/luna repartizate inegal / 4 ore/zi	3222RON brut
2	Software Tester	Minim 6 luni în domeniu	Cunoștințe în testarea proiectelor software	80 ore/luna repartizate inegal / 4 ore/zi	3000 RON brut
3	Customer Support	Minim 6 luni în domeniu	Cunoștințe în departamentul de HR (și posibil Marketing)	80 ore/luna repartizate inegal / 4 ore/zi	2900 RON brut
4	Project Manager	Minim 6 luni în domeniu	Cunoștințe în managementul proiectelor software	80 ore/luna repartizate inegal / 4 ore/zi	3400 RON brut

Fig. 5.1.1. Tabel cu configurația minimă de angajați

În cazul planului de marketing, ar fi nevoie de o strategie care să fie implementată in mediile cele mai frecventate de utilizatori. Această strategie va folosi atât canalele media, cât și cele tradiționale pentru a ajunge la publicul țintă, mai exact firmele de încheiere de mașini și proprietarii individuali de vehicule. Prezența online ar trebui să fie cea mai importantă, iar aceasta constă în construirea unui site web profesionist și crearea conturilor de business pe rețelele de socializare cele mai importante (cum ar fi Facebook, Instagram, TikTok), prin intermediul cărora se prezintă caracteristicile, beneficiile și capacitățile sistemului de monitorizare în timp real. Eforturile de marketing digital vor include optimizarea motoarelor de căutare pentru a genera trafic organic, campanii de publicitate cu pay-per-click (PPC) pentru a capta interesul imediat și marketing prin e-mail direcționat pentru a implica clienții potențiali. În plus, se poate produce conținut informativ, cum ar fi bloguri, videoclipuri și studii de caz, pentru a educa publicul cu privire la avantajele sistemului propus.

În cazul abordării tradiționale, se pot participa la târguri comerciale din industrie și conferințe auto pentru a demonstra produsul și rețeaua cu potențiali clienți de afaceri. De asemenea, oferirea de teste sau demonstrații gratuite companiilor de închiriere de mașini poate ajuta la asigurarea contractelor inițiale și la construirea încrederii în soluția propusă. Studiile de caz de la cei care adoptă din timp sistemul, sau cei care sunt în procesul de testare beta vor fi evidențiate pentru a construi credibilitatea și pentru a prezenta beneficiile sistemului CarMonitor.

Pentru producția și implementarea cu succes a “CarMonitor”, este importantă stabilirea unor relații de încredere și rentabile cu furnizorii pieselor de care este nevoie pentru construirea sistemului. Componentele hardware principale sunt microcontrolerul ESP-32 și cititorul OBD-II. Ne vom procura microcontrolerele ESP-32 de la furnizori de electronice reputați, cum ar fi Sigmanortec, Digi-Key sau Adafruit, care sunt cunoscuți

pentru componentele de calitate și pentru seriozitatea cu clienții. Acești furnizori oferă plăci de dezvoltare ESP-32 la prețuri competitive. Astfel, se menține eficiența costurilor, aspect important de ținut sub control mai ales în cazul unui start-up ipotetic aflat la început de drum.

Pentru cititorul OBD-II, așa cum este folosit și în testarea reală al acestui sistem, prezentată în capitolul 5.1, se poate utiliza VGate iCar Pro V2, la un preț de aproximativ 200 RON. VGate este renumit pentru instrumentele sale de diagnosticare auto de înaltă calitate, iar iCar Pro V2 se încadrează ca fiind potrivit pentru aplicația propusă spre comercializare, datorită performanței sale robuste și compatibilității cu o gamă largă de vehicule. Se poate stabili un acord de furnizare directă cu VGate sau distribuitori autorizați pentru a asigura o aprovizionare constantă și fiabilă a acestor cititoare OBD-II. Asigurând furnizori de încredere pentru aceste componente critice, se poate asigura producția constantă și calitatea înaltă a sistemelor noastre CarMonitor, iar angajamentul de a livra clienților noștri un produs de încredere este respectat.

5.2 Analiza SWOT

Analiza SWOT reprezintă o metodă strategică de evaluarea al punctelor forte (Strengths), punctelor slabe (Weaknesses), oportunităților (Oportunities) și amenințărilor (Threats) legate de o afacere sau un proiect. Aceasta este folositoare pentru identificarea situației interne și externe ale pieței, iar strategiile pot fi construite astfel încât să eficientizez creșterea și îmbunătățirea rezultatelor în timp ce aspectele negative sunt atenuate.

“CarMonitor” posedă câteva puncte forte care îl poziționează favorabil pe piața de monitorizare a vehiculelor. În primul rând, sistemul folosește tehnologia avansată, cum ar fi microcontrolerul ESP-32 și cititorul VGate iCar Pro V2 OBD-II, asigurând performanță și fiabilitate ridicate. Capacitățile de monitorizare în timp real și analiza completă a datelor oferite de sistemul nostru oferă o valoare semnificativă atât firmelor de închiriere de mașini, cât și proprietarilor individuali de vehicule. În plus, interfața web ușor de utilizat și capacitatea de a monitoriza de la distanță starea vehiculului îmbunătățesc experiența generală a utilizatorului, aspect care este unul diferențiator față de o posibilă concurență. Accentul pe componentele de calitate și dezvoltarea robustă de software subliniază, de asemenea, angajamentul de a oferi un produs valoros.

În ciuda punctelor sale forte, “CarMonitor” poate avea de suferit cu anumite puncte slabe care trebuie abordate. O potențială slăbiciune este dependența de componente hardware specifice, care ar putea duce la vulnerabilități ale lanțului de aprovizionare. Orice întrerupere în aprovizionarea acestor componente ar putea afecta programul de producție și disponibilitatea produselor. În plus, în calitate de nou intrat pe piață, construirea recunoașterii mărcii și a încrederii clienților poate dura timp și necesită eforturi semnificative de marketing. Costurile inițiale pentru aceste eforturi de marketing dar și cele pentru dezvoltare ar putea fi mari, reprezentând o provocare pentru managementul bugetului în stadiile de început ale afacerii ipotetice.

În situația oportunităților, piața se prezintă deschisă pentru CarMonitor. Adoptarea tot mai mare a tehnologiilor IoT în sectorul automotive și cererea tot mai mare de sisteme de monitorizare a vehiculelor în timp real creează un mediu favorabil pentru produsul propus. Firmele de închiriere de mașini care caută în permanență modalități de a îmbunătăți gestionarea flotei și de a reduce costurile de întreținere, le fac candidații de principali pentru soluția aceasta. Extinderea ofertelor de produse pentru a include funcții suplimentare, cum ar fi alertele de întreținere predictivă, introducerea unui sistem de punctaj al șoferului sau integrarea cu alte software-uri de gestionare a flotei, poate atrage, de asemenea, mai mulți clienți. În plus, creșterea orașelor inteligente și a vehiculelor conectate deschide noi căi de colaborare și extindere, ceea ce poate duce la parteneriate strategice și noi fluxuri de venituri.

"CarMonitor" se confruntă, de asemenea, cu amenințări externe care i-ar putea afecta progresul. Competiția pe piața de monitorizare a vehiculelor este intensă, cu noi inovări continue și lupta pentru cota de piață. Progresele tehnologice rapide ar putea duce la uzură dacă nu se ține pasul cu tendințele și inovațiile care au loc. În plus, crizele economice sau fluctuațiile condițiilor de piață ar putea afecta puterea de cumpărare a clienților noștri țintă, în special a firmelor de închiriere de mașini. Schimbările de reglementare din industria auto, cum ar fi noile standarde pentru diagnosticarea vehiculelor și emisiile, ar putea necesita, de asemenea, modificări ale produsului nostru, ceea ce ar putea crește costurile și timpul de dezvoltare.

5.3 Costul și prețul ipotetic

Când se estimează prețul posibil pentru sistemul "CarMonitor", este esențial să se ia în considerare atât costurile hardware, cât și factorii suplimentari, cum ar fi dezvoltarea, marketingul și asistența. Componentele de bază ale sistemului nostru includ microcontrolerul ESP32 și cititorul VGate iCar Pro V2 OBD-II, la prețuri de aproximativ 50 RON cu TVA, respectiv 200 RON cu TVA. Aceste componente aduc costul total hardware la 250 RON pe unitate cu TVA. În plus, trebuie luate în considerare costurile pentru dezvoltarea de software, integrarea sistemului și întreținerea continuă, estimate la 200 RON în plus pe unitate, dar și o posibilă garanție și livrare, estimate la 50 RON cu TVA fiecare. Aceasta aduce costul total de producție la aproximativ 450 RON pe unitate cu TVA. Pentru a asigura o marjă de profit rezonabilă și pentru a acoperi alte cheltuieli de afaceri precum marketing și suport pentru clienți, se poate stabili un preț de bază ipotetic de 1000 RON cu TVA pe unitate în cazul achiziționării de către persoane fizice, și prețul ipotetic de 2500 RON cu TVA pe unitate dacă clientul reprezintă o firmă de închiriat mașini. Este important de menționat că acest preț ipotetic se poate modifica dacă clientul dorește o modificare avansată al parametrilor monitorizați pentru flota proprie.

Diferența de preț este se justifică deoarece în cazul sistemului destinat persoanelor juridice, sistemul poate fi configurat și personalizat în funcție de ce parametrii sunt doriți a fi monitorizați. Numărul de parametrii nu este limitat, iar acest lucru necesită o investiție mai mare din partea start-up-ului din punct de vedere al lucrului, timpului și al resurselor.

Mai mult, în cazul unui sistem de acest tip, cel mai probabil se va dori monitorizarea a mai multor vehicule în același timp, toate ilustrate în aceeași interfață, caracteristică care momentan nu există în lucrarea reală “CarMonitor”, dar care poate fi introdusă la cererea clientului ipotetic.

În continuare se va calcula costul marginal ipotetic al unei firme, care se referă la costul suplimentar suportat pentru producerea încă o unitate dintr-un produs. Se calculează luând modificarea costului total care apare atunci când cantitatea produsă se modifică cu o unitate.

Cost variabil	Cost fix
ESP-32: 50 RON cu TVA / unitate	**Dezvoltare sistem de bază**: 1200 RON /an
VGate iCar Pro V2 OBD-II: 200 RON cu TVA / unitate	**Salarii**: 150.264 RON / an
Integrarea Software: 100 RON cu TVA / unitate	**Marketing**: 3000 RON (estimativ)
Testare: 50 RON cu TVA / unitate	**Închiriere birou și utilități**: 12.000 RON (estimativ)
Livrare, garanție: 50 RON cu TVA / unitate	**Inventar inițial (unelte, echipament, licențe ...)**: 200 RON
Cost variabil total: 450 RON cu TVA / unitate	**Cost fix total: 166.664 RON**
Cost total: 391.664 RON cu TVA (considerat cu 500 unități pe an)	

Fig. 5.3.1. Tabel Cost total

Dacă se consideră că în anul întâi de operare se vor vinde în total 500 de unități de monitorizare dintre care 25% vor fi pentru persoane fizice și 75% pentru persoane juridice, iar în al doilea an o creștere de 25% datorită majorării cererilor din partea clienților, se poate deduce costul marginal descris mai jos.

$$C_{mg}=\frac{\Delta CT}{\Delta Q}=\frac{CT2-CT1}{Nr.sisteme\ an\ 2-Nr.sisteme\ an\ 1}=\frac{489{,}580-391.664}{625-500}=\frac{97{,}916}{125}=783.3\ RON\ cu\ TVA$$

Prin monitorizarea costului marginal, start-up-ul care se ocupă de comercializarea “CarMonitor” poate lua decizii informate privind creșterea producției și ajustarea strategiilor de preț pentru a rămâne competitiv și profitabil.

5.4 Proiecții financiare – cheltuieli și venituri (1 an)

Luna	Salarii (RON)	Piese	Office Utilități	Licențe	Marketing	Dezvoltare	Cost total lunar
1	12.522,00	616,67	1000	16,67	250,00	100,00	14.505,34
2	12.522,00	812,98	1000	16,67	250,00	100,00	14.701,65
3	12.522,00	1.071,37	1000	16,67	250,00	100,00	14.959,04
4	12.522,00	1.410,83	1000	16,67	250,00	100,00	15.301,17
5	12.522,00	1.857,56	1000	16,67	250,00	100,00	15.746,89
6	12.522,00	2.447,10	1000	16,67	250,00	100,00	16.336,44
7	12.522,00	3.226,16	1000	16,67	250,00	100,00	17.134,84
8	12.522,00	4.254,46	1000	16,67	250,00	100,00	18.223,80
9	12.522,00	5.606,33	1000	16,67	250,00	100,00	19.695,67
10	12.522,00	7.380,08	1000	16,67	250,00	100,00	21.789,42
11	12.522,00	9.711,90	1000	16,67	250,00	100,00	23.617,23
12	12.522,00	12.779,58	1000	16,67	250,00	100,00	26.668,92

Fig. 5.4.1 Tabel cheltuieli ipotetice

Tabelul acesta este o reprezentare ipotetică al desfășurării cheltuielilor pe parcursul al unui an pentru un start-up care comercializează sistemul “CarMonitor”. Costul fix include salariile, biroul inclusiv utilitățile, licențe, marketing (dar care poate fi estimat și ca un cost variabil) și costul de dezvoltare .Costul variabil este reprezentat de coloana pieselor, care estimează o cerere exponențială din momentul intrării pe piață.

Luna	Unități vândute	Venit	Cost lunar	Profit (RON)
1	2	5.000	14.505,34	-9.505,34
2	3	7.500	14.701,65	-6.201,65
3	5	12.500	14.959,04	-3.459,04
4	6	15.000	15.301,17	-3.001,17
5	9	22.500	15.746,89	6.753,11
6	13	32.500	16.336,44	16.163,56
7	19	47.500	17.134,84	30.365,16
8	27	67.500	18.223,80	49.276,20
9	38	95.000	19.695,67	75.304,33
10	53	132.500	21.789,42	110.710,58
11	74	185.000	23.617,23	161.382,77
12	94	235.000	26.668,92	208.331,08

Fig. 5.4.2 Tabel venituri ipotetice

Pentru estimarea profitului s-a luat în considerare o creștere exponențială a vânzării, mai exact un număr de 500 de sisteme pe an vândute. Se observă că în primele patru luni profitul este negativ, din cauza costului lunar mare și al venitului mic dar situația

se îmbunătățește pe măsură ce numărul de sisteme crește. Această creștere ar putea fi un rezultat al campaniilor de marketing care au rezultate satisfăcătoare.

6. CONCLUZIE

6.1 Obiective îndeplinite

Sistemul prezentat în această lucrare, care poartă numele „CarMonitor”, 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 observă un comportament neașteptat al unui parametru al mașinii, se pot lua decizii care să prevină o posibilă daună mai mare, economisindu-se bani și resurse. De aceea, utilizând acest sistem, se poate anticipa ce componentă și când va trebui schimbată în viitor.

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 puse la dispoziție sunt tratate corespunzător și nu sunt abuzate, acestea pot fi dotate cu sistemul „CarMonitor” 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 acestei utilizări 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 „CarMonitor” 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, există posibilitatea de a se efectua o investigație pentru a stabili ce alte modificări au fost aduse mașinii, și daca acestea sunt conforme cu legislația în vigoare.

Important de menționat este faptul că parametrii afișați de către sistemul „Car Monitor” pot fi personalizați în funcție de fiecare caz. Dacă un client, fie că acesta este o persoană fizică sau juridică dorește să monitorizeze alți parametrii decât cei cu care sistemul vine în mod predefinit, acesta poate să aleagă din lista de parametrii suportați, pusă la dispoziție de ELMduino. În cazul în care parametrul dorit nu se află printre acestea, pentru a afișa parametrul dorit se poate realiza adăugarea funcțiilor necesare și compunerea PID-ului respectiv (dacă este acceptat de ECU al mașinii).

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. Sistemul „CarMonitor” 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.

Unul dintre obiectivele cheie care au fost îndeplinite în acest proiect este crearea unei interfețe pentru afișarea parametrilor mașinii care este construită cu vizibilitatea și utilizabilitatea acesteia dintr-o locație de la distanță, sporind productivitatea și gestionarea vehiculului respectiv. Capacitatea de a accesa date critice legate de parametrii principali ai vehiculului oferă utilizatorilor sau proprietarilor, informații importante legate de starea acestuia, în timp ce întreținerea proactivă este facilitată și riscul defecțiunilor neașteptare este redus semnificativ

O altă realizare semnificativă este implementarea cu succes a trimiterii și primirii PID-urilor (ID-urilor parametrilor) către și de la ECU (Unitatea de control al motorului) folosind API-ul ELMduino disponibil pe GitHub [17]. Această capacitate este crucială pentru diagnosticarea promptă și monitorizarea performanței, deoarece permite sistemului să comunice direct cu computerul de bord al vehiculului. Prin folosirea acestui API, s-a asigurat compatibilitatea cu o gamă largă de vehicule dar și prelucrarea datelor brute primite de la computerul de bord. Utilizarea acestuia extinde astfel aplicabilitatea sistemului “CarMonitor” pe o gamă mai largă de vehicule, și oferă informații detaliate de diagnosticare în timp real esențiale pentru menținerea performanței optime.

În cadrul proiectului s-a realizat, de asemenea, coexistența între capabilitățile WiFi și Bluetooth, o etapă tehnică importantă. Această conectivitate dual asigură că sistemul poate gestiona eficient transmisia de date de la computerul de bord al vehiculului, până la webserver-ul dedicat. Prin integrarea atât a WiFi, cât și a Bluetooth, “CarMonitor” oferă flexibilitate și fiabilitate în diferite scenarii iar prin asigurarea unui flux de date continuu, fără întreruperi, experiența utilizatorului este satisfăcută.

Gestionarea diferiților parametri la scară ierarhică în timpul testării a fost un alt obiectiv important realizat în acest proiect. Prin stabilirea unei logici structurate a managementului ratei de interogare al parametrilor, s-a asigurat că aceștia nu folosesc mai multe resurse decât ar fi necesar. Acest cadru ierarhic de testare a permis prioritizarea funcțiilor critice și identificarea problemelor legate de fluxul datelor care au apărut în timpul testării.

6.2 Impedimente

Construirea acestui proiect, ca orice alt proiect de amploare, nu a fost realizat fără apariția impedimentelor. Primul impediment semnificativ întâlnit în timpul proiectului a fost utilizarea inițială a unui cititor OBD-II defect. Acest dispozitiv defect a împiedicat fazele inițiale ale dezvoltării lucrării de licențe, prin faptul că nu era descoperibil față de placa ESP-32. Pentru a rezolva această problemă, am contactat proprietarul repository-ului ELMduino de pe GitHub prin secțiunea Issues, având în minte posibilitatea ca API-ul să fie defect. Îndrumarea lui a fost de ajutor în depanare, iar în cele din urmă, am decis să cumpăr un cititor OBD-II de calitate superioară de la VGate. Acest nou dispozitiv, VGate

iCar Pro V2, a funcționat excepțional de bine, conectându-se la ESP-32 prin protocolul destinat dispozitivelor cu sistemul de operare Android. Acesta oferă date precise și consistente către placă, ceea ce a fost critic pentru succesul proiectului.

O altă provocare a apărut când am încercat să conectez microcontrolerul ESP-32 la hotspot-ul iOS. ESP-32 nu se putea conecta la rețeaua oferită de telefonul mobil. După cercetări ample și depanare, am constatat că modificarea fișierului de configurare platformio.ini pentru a include semnale specifice pentru securitatea WPA3 a rezolvat problema. Această ajustare a permis ESP-32 să se conecteze fără probleme la hotspot-ul iOS și a permis testarea webserver-ului.

Integrarea ambelor funcționalități WiFi (pentru webserver) și Bluetooth (pentru comunicarea cu ECU) pe ESP-32 a prezentat un alt impediment, deoarece ambele module folosesc aceeași antenă. Această utilizare dublă a cauzat interferențe și probleme de conectivitate. În faza de început al sistemului, webserver-ul se inițializa, dar comunicarea prin Bluetooth către computerul de bord era compromisă, deoarece nu exista o logică de acordare a controlului antenei. Pentru a depăși acest lucru, s-a folosit caracteristica de coexistență disponibilă în bibliotecile încorporate ale ESP-32 puse la dispoziție de sistemul de operare FreeRTOS. Această caracteristică a permis ESP-32 să gestioneze eficient operațiile pe WiFi și Bluetooth.

În timpul fazei de testare, am întâmpinat probleme cu rata de refresh a diferiților parametri ai vehiculului. Datele se actualizau prea lent, ceea ce făcea sistemul neutilizabil. Pentru a rezolva acest lucru, un sistem de management pentru parametrii și triere al valorilor false recepționate din cauza interferențelor a fost implementat. Această ierarhie a prioritizat datele cele mai critice, descrise în capitolul dedicat testării. Implementarea acestor abordări a rezolvat problema ratei de refresh și al valorilor false recepționate.

6.3 Implementări viitoare

Unul dintre aspectele care ar putea fi îmbunătățite în viitor este modul de găzduire a serverului web. În prezent, proiectul a fost dezvoltat utilizând un server web local (localhost), ceea ce înseamnă că trebuie să fie pe aceeași rețea cu ESP-32 pentru a fi accesibil. Totuși, găzduirea acestuia prin AWS (Amazon Web Services) ar putea face posibilă disponibilitatea online a serverului web, fără a fi necesară conectarea la aceeași rețea. Aceasta ar permite monitorizarea vehiculului de la distanță și ar reprezenta prima îmbunătățire care ar trebui făcută, deoarece prin intermediul acesteia se oferă utilizatorilor acces la datele vehiculului din orice locație cu acces la Internet.

O altă implementare viitoare ar putea include posibilitatea ca mai multe vehicule (fiecare dotate cu ESP-32 și un OBD-II) să trimită parametrii către același server web. Există posibilitatea, de asemenea, ca în acest caz plăcile ESP-32 să trebuiască să fie dotate cu module SIM, cu acces la Internet prin cartelă, pentru a trimite către un server master. Aceasta ar acorda proiectului actual o soluție scalabilă pentru monitorizarea flotelor de vehicule. În forma sa actuală, acest proiect servește drept proof of concept, demonstrând că este posibilă colectarea și transmiterea datelor vehiculului în timp real.

Extinderea capacității de a gestiona date de la mai multe dispozitive simultan ar crește semnificativ utilitatea și aplicabilitatea sistemului pentru utilizatori comerciali, cum ar fi companiile de închirieri auto sau firmele de logistică.

Fiind open source, API-ul ELMduino poate fi extins pentru a gestiona mai mulți parametri, dacă programatorul decide să facă acest lucru și dacă vehiculul suportă PID-ul respectiv. În prezent, ELMduino API gestionează un set limitat de parametri OBD-II, dar există potențial pentru a adăuga suport pentru parametri suplimentari. Aceasta ar necesita contribuții din partea comunității de dezvoltatori, care ar putea adăuga noi funcționalități și optimizări API-ului. Extinderea API-ului ar permite monitorizarea unor seturi mai largi de date, pentru a oferi utilizatorilor informații mai detaliate despre starea și performanța vehiculului.

7. BIBLIOGRAFIE

[1] - „Basic Information OBDII & OBDII PIDS," US EPA. Disponibil pe: https://www.epa.gov/obd/basic.htm.

[2] - „Hyundai Bluelink," Hyundai. Disponibil pe: https://www.hyundai.com/eu/driving-hyundai/owning-a-hyundai/bluelink-connectivity/bluelink-app.html#tab-remote-climate.

[3] - „Azuga," Azuga. Disponibil pe: https://www.azuga.com/blog/what-is-remote-vehicle-monitoring.

[4] - „OVMS," OVMS. Disponibil pe: https://github.com/openvehicles/Open-Vehicle-Monitoring-System-3?tab=readme-ov-file.

[5] - „ES32 vs ESP8266. Disponibil pe: https://makeradvisor.com/esp32-vs-esp8266/.

[6] - „Datasheet ESP32," Espressif Systems, 29 April 2024. Disponibil pe: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[7] - N. Kolban, Kolban's Book on ESP32, 2017.

[8] - „ESPAsyncWebServer" . Disponibil pe: https://github.com/me-no-dev/ESPAsyncWebServer

[9] - „SPIFFS" Espressif. Disponibil pe: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/spiffs.html.

[10] - „GitHub ELMduino". Disponibil pe: https://github.com/PowerBroker2/ELMduino/.

[11] - „HTML Basics," Mozilla MDN. Disponibil pe: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics.

[12] - „Doctype," Mozila MDN. Disponibil pe: https://developer.mozilla.org/en-US/docs/Glossary/Doctype.

[13] - „CSS Basics," Mozilla MDN. Disponibil pe: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics.

[14] - „JavaScript Basics," Mozilla MDN. Disponibil pe: https://developer.mozilla.org/en-

US/docs/Learn/Getting_started_with_the_web/JavaScript_basics.

[15] - „JavaScript,” Wikipedia. Disponibil pe: https://en.wikipedia.org/wiki/JavaScript.

[16] - „Overhead,” Wikipedia. Disponibil pe: https://en.wikipedia.org/wiki/Overhead_(computing)#:~:text=In%20computer%20science%2C%20overhead%20is,to%20perform%20a%20specific%20task..

[17] - PowerBroker2, „Github”, Cod. Disponibil pe: https://github.com/PowerBroker2/ELMduino/tree/2c242bcf28a05ebc46c48f650b16611452b04695/examples.

Calculatoare și Tehnologia Informației
Sesiunea Iunie 2024
Horvath Paul Șerban
CarMonitor

Universitatea Politehnica din Timișoara

DECLARAŢIE DE AUTENTICITATE A LUCRĂRII DE FINALIZARE A STUDIILOR*

Subsemnatul HORVATH PAUL ȘERBAN,
legitimat cu CI seria ZH nr. 501139,
CNP 5010711055138
autorul lucrării SISTEM DE MONITORIZARE DE LA DISTANȚĂ AL PARAMETRILOR AUTOVEHICULELOR - CARMONITOR
elaborată în vederea susţinerii examenului de finalizare a studiilor de LICENȚĂ organizat de către Facultatea AUTOMATICĂ ȘI CALCULATOARE din cadrul Universităţii Politehnica Timişoara, sesiunea IUNIE a anului universitar 2023 - 2024, coordonator CONF. DR. ING. PRODAN LUCIAN, luând în considerare art. 34 din *Regulamentul privind organizarea și desfășurarea examenelor de licență/diplomă și disertație*, aprobat prin HS nr. 109/14.05.2020 și cunoscând faptul că în cazul constatării ulterioare a unor declaraţii false, voi suporta sancțiunea administrativă prevăzută de art. 146 din Legea nr. 1/2011 – legea educației naționale și anume anularea diplomei de studii, declar pe proprie răspundere, că:

- această lucrare este rezultatul propriei activități intelectuale;
- lucrarea nu conține texte, date sau elemente de grafică din alte lucrări sau din alte surse fără ca acestea să nu fie citate, inclusiv situația în care sursa o reprezintă o altă lucrare/alte lucrări ale subsemnatului;
- sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor;
- această lucrare nu a mai fost prezentată în fața unei alte comisii de examen/prezentată public/publicată de licență/diplomă/disertație;
- În elaborarea lucrării ~~am utilizat~~ instrumente specifice inteligenței artificiale (IA) și anume _____ (denumirea) _____ (sursa), pe care le-am citat în conținutul lucrării/<u>nu am utilizat</u> instrumente specifice inteligenței artificiale (IA)[1].

Declar că sunt de acord ca lucrarea să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului său într-o bază de date în acest scop.

Timişoara,

Data
21.06.2024

Semnătura

*Declaraţia se completează de student, se semnează olograf de acesta și se inserează în lucrarea de finalizare a studiilor, la sfârșitul lucrării, ca parte integrantă.
[1] Se va păstra una dintre variante: 1 - s-a utilizat IA și se menționează sursa 2 – nu s-a utilizat IA