



UNIVERSITATEA „ȘTEFAN CEL MARE” SUCEAVA
FACULTATEA DE INGINERIE ELECTICĂ ȘI ȘTIINȚA

CALCULATOARELOR

PROGRAM DE STUDIU: AUTOMATICĂ ȘI INFORMATICĂ
APLICATĂ

Sistem integrat de comunicație vehicul–infrastructură (V2I). Automobil autonom cu funcție de recunoaștere a semnelor rutiere și interfață Android

Coordonator,

Ş.I.dr.ing. Corneliu Buzduga

Student,

Bogdan Iulian Mihailă

Cuprins

Capitolul 1 – Introducere.....	1
1.1. Scopul lucrării.....	1
1.2. Motivația alegerii temei.....	2
1.3. Sinteză lucrării	3
Capitolul 2 - Contextul științific relevant al subiectului	4
2.1. Vehicule autonome și sisteme inteligente de transport (ITS)	4
2.2. Comunicarea vehicul–infrastructură (V2I).....	4
2.3. Controlul și eficiența traficului cu V2I.....	6
2.4. RFID ca alternativă simplificată pentru infrastructura rutieră.....	7
2.5. IoT și V2I: Aplicații și tendințe relevante.....	8
2.6. Prioritizarea vehiculelor de urgență.....	8
2.7. Sinergia RFID–GPS.....	8
2.8. Arhitectură de procesare locală la marginea rețelei(edge computing).....	9
2.9. Stadiul actual al cercetărilor privind integrarea comunicației V2I în vehicule autonome	9
2.10. Lacune identificate în literatura de specialitate	11
2.11. Contribuțiiile originale ale lucrării.....	11
Capitolul 3 – Proiectarea sistemului.....	12
3.1. Cerințe funcționale	12
3.2. Arhitectura generală a sistemului	14
3.3. Arhitectura hardware- Selecția componentelor	16
Capitolul 4. Implementare, dezvoltare și testare	20
4.1. Arhitectura hardware – viziune detaliată	20
4.2. Schema electrică	22
4.3. Implementare software	26
4.4. Aplicația Android.....	27
4.5. Firmware-ul ESP32	31
4.6. Firmware-ul Arduino	36
4.6. Rezultate experimentale	37
Concluzii, contribuții și direcții viitoare de dezvoltare	39
Referințe bibliografice:	43
Anexe	47

Capitolul 1 – Introducere

1.1. Scopul lucrării

Lucrarea de față își propune dezvoltarea unui sistem experimental integrat de comunicație vehicul–infrastructură (V2I), aplicat unui vehicul autonom miniatural, capabil să interacționeze inteligent cu elementele infrastructurii rutiere. Scopul urmărit este demonstrabil din punct de vedere atât tehnic, cât și funcțional și constă în implementarea unei platforme capabile să detecteze și să interpreteze semne de circulație echipate cu etichete RFID, să evite obstacole în timp real prin senzori ultrasonici, să opereze autonom pe trasee predefinite și să fie controlată și monitorizată printr-o aplicație mobilă Android, utilizând comunicarea Bluetooth Low Energy (BLE).

Sistemul propus urmărește validarea unor principii de bază ale vehiculelor autonome și ale infrastructurii inteligente, prin:

- integrarea senzorilor și actuatorilor într-o arhitectură embedded distribuită, cu ESP32 ca unitate principală de procesare;
- utilizarea unui cititor RFID pentru recepționarea datelor de la infrastructură rutieră pasivă;
- dezvoltarea unui mecanism de control mixt (manual, asistat și autonom) accesibil prin interfață mobilă;
- aplicarea unui model cinematic simplificat pentru navigație și evitarea coliziunilor în medii controlate.

Prin această lucrare se urmărește nu doar testarea fezabilității unei astfel de platforme, ci și evaluarea limitărilor și potențialului extinderii către aplicații reale din domeniul transporturilor inteligente. Într-un peisaj urban în continuă digitalizare, soluția propusă oferă o alternativă practică, adaptabilă și eficientă pentru modernizarea infrastructurii rutiere, cu beneficii directe în ceea ce privește siguranța, adaptabilitatea și reducerea costurilor operaționale.

1.2. Motivația alegerii temei

Automatizarea transportului rutier și integrarea acestuia într-un sistem digital interconectat reprezintă una dintre direcțiile fundamentale ale ingineriei contemporane. Dincolo de inovația tehnologică, vehiculul autonom devine o componentă critică în arhitectura orașelor autonome și în transformarea infrastructurii urbane tradiționale într-o adaptivă, capabilă să răspundă în timp real la cerințele de mobilitate, siguranță și eficiență energetică.

Tema abordată în această lucrare vizează dezvoltarea unei platforme vehicul-infrastructură care demonstrează concepte de comunicare contextuală și reacție automată la semnalizarea rutieră, utilizând tehnologii moderne precum RFID (pentru recunoașterea infrastructurii) și Bluetooth Low Energy (pentru interacțiune mobilă).

Motivația principală derivă din provocările actuale ale sistemelor autonome în gestionarea scenariilor complexe de trafic și în interpretarea contextuală a semnelor rutiere în lipsa unei comunicări explicite cu infrastructura. Soluția propusă explorează o arhitectură hibridă, în care vehiculul nu doar „percepe” mediul, ci colaborează activ cu infrastructura, consolidând astfel paradigma de percepție distribuită (distributed perception), esențială pentru viitoarele sisteme V2X (Vehicle-to-Everything).

Datele statistice actuale justifică orientarea cercetării în această direcție. Conform European Commission Mobility and Transport, în 2022, pierderile economice anuale asociate congestiei urbane și accidentelor în UE au fost estimate la peste 300 miliarde euro, o parte semnificativă fiind atribuită ineficienței comunicației între vehicul și infrastructură [1]. În paralel, Statista raportează că piața vehiculelor autonome va depăși 800 miliarde USD până în 2030, cu o creștere semnificativă în segmentele care integrează V2I și V2X [2].

Avantajele tematici majore ale cercetării:

- Testarea și validarea conceptului de infrastructură rutieră augmentată digital – în care elementele pasive (semne de circulație) devin noduri de informație activă, contribuind la un ecosistem rutier inteligent.
- Explorarea arhitecturilor embedded multitasking (capacitatea unui microcontroler de a gestiona mai multe procese, cum ar fi citirea senzorilor, comunicarea BLE și actualizarea afișajului, în același timp, în cadrul unui sistem integrat) în condiții de resurse limitate.
- Simularea unui ecosistem V2I în format miniatural, dar algoritmic complet, care permite implementarea unor strategii reale de navigație, decizie și evitare a obstacolelor.

- Validarea ideii de fuziune a percepției senzorilor locali cu informația contextuală externă (semne rutiere, semafoare etc.), ceea ce oferă o soluție alternativă la dependența excesivă de procesarea vizuală (cameră/LiDAR), cu un cost computațional mai mic.

Într-un moment în care mobilitatea sustenabilă și inteligența rutieră devin imperitive strategice în plan urban și industrial, lucrarea de față se aliniază la aceste direcții de cercetare, oferind un cadru experimental coerent pentru testarea ideilor V2I într-o manieră controlată, scalabilă și tehnologic matură.

1.3. Sinteza lucrării

Lucrarea constă într-un sistem V2I autonom la scară redusă, construit în jurul unui vehicul miniatural, capabil să recunoască semne rutiere dotate cu etichete RFID, să evite obstacole și să execute trasee predefinite. Arhitectura hardware este centrată pe microcontrolerul ESP32, care gestionează simultan recunoașterea semnelor de circulație, navigația autonomă și comunicația Bluetooth Low Energy (BLE) cu aplicația mobilă. Vehiculul este echipat cu patru senzori ultrasonici pentru percepția mediului, un cititor RFID pentru detectarea semnalizării rutiere și un set de actuatori (motor DC și servomotor) pentru propulsie și direcție.

Sistemul operează în trei moduri: MANUAL (control direct din aplicația Android), AUTO (asistență rutieră cu interpretarea semnelor) și AUTONOM (executarea automată a unui traseu învățat anterior). Aplicația mobilă, construită în Kotlin, oferă control în timp real, vizualizare a datelor senzorilor și afișarea semnelor detectate.

Semnalizarea rutieră este reprezentată de indicatoare fizice echipate cu etichete RFID pasive, fiecare conținând informații specifice precum tipul semnului de circulație, identificatorul unic și alte date despre orientare și poziționare. În momentul traversării unui astfel de punct, vehiculul citește datele și își adaptează traectoria sau acțiunea în funcție de informația primită, ajustând viteza, fie oprind sau efectuând o decizie de navigație.

Automobilul Elysium RC permite validarea unor concepte fundamentale de comunicare vehicul-infrastructură, în condiții de consum redus de energie și scalabilitate, simulând un ecosistem V2I în miniatură, dar cu toate componentele esențiale ale unui sistem modern de transport asistat de tehnologie și capabil de decizii adaptative în timp real.

Capitolul 2 - Contextul științific relevant al subiectului

2.1. Vehicule autonome și sisteme inteligente de transport (ITS)

Progresul accelerat în domeniul vehiculelor autonome reflectă o transformare profundă a paradigmelor mobilității urbane. Începând cu primele prototipuri experimentale, sistemele autonome au evoluat spre implementări comerciale, capabile de navigare în medii semi-structurate și urbane[4]. Conform clasificării standardizate de SAE (Society of Automotive Engineers) International - Societatea inginerilor din domeniul auto, nivelurile de autonomie variază de la 0 (fără automatizare) până la nivelul 5 (autonomie completă fără intervenție umană)[5]. Vehiculele de nivel 3 și 4, caracterizate prin capacitatea de a prelua conducerea în condiții definite, se bazează pe o integrare avansată între senzori, algoritmi de control și sisteme de comunicație [6]. Platformele experimentale miniaturale, precum Elysium RC, oferă un mediu scalabil și sigur pentru testarea conceptelor legate de percepție și control în regim autonom [7].

În cadrul sistemelor de transport inteligente (ITS), vehiculul autonom nu este doar o entitate izolată, ci o componentă conectată care interacționează dinamic cu infrastructura rutieră, alți participanți la trafic și rețeaua digitală urbană[8].

2.2. Comunicarea vehicul–infrastructură (V2I)

Vehicle-to-Infrastructure (V2I)-Comunicare între vehicul și infrastructura rutieră- este o componentă a ecosistemului de comunicații inteligente din domeniul transporturilor, definită drept interacțiunea bidirectională între un vehicul și elementele fizice sau digitale ale infrastructurii rutiere, prin intermediul tehnologiilor de comunicație fără fir. Această comunicare are ca scop schimbul de informații relevante privind condițiile de trafic, semnalizarea rutieră, starea drumurilor sau alte elemente de context necesare pentru sprijinirea deciziilor automate sau asistate ale vehiculului.

Un element esențial pentru funcționarea vehiculelor autonome este integrarea comunicației vehicul-infrastructură (V2I), parte a ecosistemului mai larg V2X (Vehicle-to-Everything). Această comunicare bidirectională permite vehiculului să primească informații în timp real despre semne rutiere, starea drumurilor, trafic, obstacole temporare sau priorități de semafor [9].

Pentru implementarea în medii controlate sau la scară redusă, cum este cazul Elysium RC, o soluție viabilă este utilizarea tehnologiei RFID(Radio-Frequency Identification-Identificare prin frecvență radio.) pasive pentru simularea infrastructurii rutiere. RFID oferă o alternativă simplă și eficientă energetic față de tehnologiile costisitoare (ex: LiDAR-Light Detection and Ranging- Detectie optică cu rază laser, DSRC - Dedicated Short-Range Communications - Comunicații dedicate pe distanțe scurte), asigurând identificarea semnelor rutiere digitale prin intermediul tag-urilor UHF [10]. Acestea pot transmite informații codificate către un cititor montat pe vehicul, cu o rată de detecție de peste 80% în condiții optime de poziționare și viteză redusă [11]. În lucrări recente, se evidențiază și utilizarea semnalului RSSI (Received Signal Strength Indicator- măsură numerică care indică cât de puternic este semnalul radio recepționat de un dispozitiv (precum un microcontroler, un telefon sau un modul de comunicație wireless) sau a fazei semnalului pentru localizare RFID, obținând o acuratețe sub 20 cm – suficientă pentru scenarii de navigare autonomă precisă [12].

Integrarea comunicației vehicul–infrastructură (V2I) în vehicule autonome devine esențială pentru atingerea unei mobilități digitalizate și sigure. Una dintre cele mai importante aplicații V2I este transmiterea automată de informații despre semne rutiere și semafoare către vehicul, pentru luarea de decizii autonome sau asistate.

Masatu et al. au dezvoltat un sistem mobil de alertare a șoferilor cu privire la semnele rutiere aflate în proximitate. Sistemul utilizează coordonatele GPS ale telefonului și o aplicație Android care declanșează alerte vocale atunci când un semn este identificat la mai puțin de 10 metri distanță [13]. Această soluție simplă și eficientă evidențiază potențialul aplicațiilor mobile de a funcționa ca interfețe V2I pentru vehicule semi-autonome.

Pe o direcție evolutivă, Calafate și Garcia au analizat conceptul de semnalizare rutieră digitală wireless, în care semnele de circulație devin elemente programabile, capabile să transmită informații direct către vehicul prin comunicații fără fir. Ei propun ca semnele să fie echipate cu transmițătoare capabile să adapteze mesajele în funcție de contextul rutier (ex. lucrări, incidente temporare), oferind astfel o interfață V2I activă și adaptivă [14].

Rahman (2023) a demonstrat o aplicație avansată a acestei paradigmă, prin combinarea recunoașterii vizuale (YOLOv8s) cu comunicare V2I pentru controlul la intersecții. Sistemul detectează semafoare în mediu simulat (MAVS) și transmite vehiculului mesaje SPaT (Signal Phase and Timing), permitând decizii automatizate precum frânarea anticipată sau menținerea vitezei în funcție de culoarea semaforului [15]. Astfel, recunoașterea vizuală devine parte a unui sistem digital hibrid, în care percepția este completată de informații transmise de infrastructură.

Aceste contribuții demonstrează diversitatea abordărilor în dezvoltarea de soluții V2I pentru recunoașterea semnelor rutiere. De la aplicații mobile simple, la sisteme cu procesare vizuală și transmisii SPaT(Signal Phase and Timing un tip de mesaj standardizat folosit în sistemele de transport inteligente (ITS), care transmite informații despre starea curentă a semaforului (culoarea semnalului) și timpul rămas până la schimbarea acestuia), direcția este clară: vehiculul viitorului trebuie să comunice intelligent cu infrastructura rutieră pentru a naviga în siguranță și eficiență.

2.3. Controlul și eficiența traficului cu V2I

Un alt segment important al cercetărilor V2I se concentrează pe optimizarea traficului la intersecții semaforizate, unde utilizarea informațiilor în timp real poate îmbunătăți semnificativ fluxul și eficiența energetică.

Ahmed et al. (2018) au propus un sistem de tip Intersection Approach Advisory- este un tip de mesaj informativ transmis vehiculelor (în special celor autonome sau conectate), care oferă indicații contextuale atunci când acestea se apropie de o intersecție. folosind comunicări DSRC pentru a transmite mesaje SPaT direct către vehicul, care apoi afișează pentru șofer viteza optimă de apropierie. În teste de teren și simulare, sistemul a oferit alerte vizuale și audio, contribuind la reducerea consumului de combustibil și a emisiilor fără distragerea atenției șoferului [16].

Un algoritm eco-driving pentru intersecții semaforizate bazat pe V2I, dezvoltat de Pengyuan Sun și colaboratori (2023), utilizează modele cinetice și de urmărire automată pentru a coordona viteza vehiculului cu timpii de semaforizare. Prin recomandarea unei viteze adaptate intrării în intersecție, se obțin beneficii în termeni de throughput și eficiență energetică [17]. Controlul predictiv al semaforizării a fost optimizat prin algoritmi de control predictiv. Aceasta folosește model H-V de trafic și comunicare V2I pentru actualizări în timp real, oferind reduceri semnificative ale numărului de opriri și ale întârzierilor [18].

Un studiu al Oliva et al. a testat sisteme V2I în intersecții pentru vehicule de urgență și siguranța pietonilor. Un prototip IoT trimite alerte smartphone către șoferi, îmbunătățind timpul de răspuns al ambulanțelor și creșterea conștientizării pietonilor cu aproximație [20].

2.4. RFID ca alternativă simplificată pentru infrastructura rutieră

În cercetările recente, RFID UHF pasiv a apărut ca o soluție accesibilă, fiabilă și scalabilă pentru comunicații vehicul–infrastructură (V2I), oferind o alternativă eficientă la tehnologiile avansate și costisitoare.

Radio Frequency Identification- identificare prin frecvență radio (RFID) este o tehnologie de identificare automată care permite citirea și, în unele cazuri, scrierea de date stocate pe etichete electronice (tag-uri), prin intermediul undelor radio.

RFID UHF (identificare prin frecvență radio în bandă ultra-înaltă) se referă la acea clasă de RFID care operează în intervalul de frecvență 860–960 MHz, conform standardului ISO/IEC 18000-6C / EPCglobal Gen2 [21]. Etichetele RFID UHF sunt, de regulă, pasive (fără sursă de alimentare proprie) și sunt activate de energia emisă de către cititor. Acestea pot fi detectate de la distanțe semnificative (până la 12 metri în condiții ideale), ceea ce le face potrivite pentru aplicații de identificare rapidă, non-contact, cum ar fi logistica, accesul controlat și sistemele de transport inteligente.

Un studiu important din 2023 descrie conceptul de „RF-Enhanced Infrastructure (REI - Infrastructură îmbunătățită prin frecvență radio)”, bazat pe protocoale EPC Gen 2 pentru RFID UHF. Autorii prezintă o arhitectură completă, optimizată pentru citirea etichetelor rutiere dinamic, conectate la infrastructură. Ei subliniază importanța codării conform specificațiilor și orientarea adecvată a antenelor pentru maximizarea timpului de citire în mediul urban [22].

O abordare practică și experimentală este prezentată tot în 2023: un sistem V2I RFID destinat semnalelor rutiere pasive, în care etichetelor UHF, integrate în elementele infrastructurii (de ex. indicatoare), transmit informații către un vehicul echipat cu un cititor RFID. Testele confirmă o funcționare corectă la viteze moderate și condiții meteo variate [23].

Într-un alt studiu, s-a încercat utilizarea atât a RFID-ului, cât și a GPS-ului pentru marcaje precise de traseu în condiții adverse (zăpadă, ploaie), beneficiind de un design de antenă adaptat zonei rutiere (instalată în asfalt). Acest sistem asigură o rezistență la interferențe de mediu[24].

Un document din 2024 prezintă o soluție RFID I2V – cu taguri pasive aplicate pe semne de circulație – și un cititor montat pe vehicul. Sistemul oferă informații vizuale și sonore într-un ecran auto, confirmând utilitatea RFID în infrastructurile rutiere digitale[25].

În concluzie, literatura de specialitate recentă consolidează faptul că RFID UHF pasiv este o soluție robustă pentru sisteme V2I la scară redusă sau medie. Aceste lucrări validează

fezabilitatea – inclusiv în condiții ambientale diverse – și relevă oportunitățile de inovare, în special prin optimizarea antenelor, protocoalelor de codare și integrarea hardware adaptat.

2.5. IoT și V2I: Aplicații și tendințe relevante

Internetul lucrurilor-Internet of Things (IoT) desemnează o rețea formată din obiecte fizice inteligente, echipate cu senzori și module de comunicație, capabile să colecteze, să transmită și să proceseze automat date prin intermediul internetului sau al altor tehnologii de conectivitate. Prin integrarea IoT cu V2I, vehiculele devin actori activi într-un ecosistem digital, capabili să primească și să genereze informații relevante pentru siguranță, eficiență și mobilitate urbană intelligentă.

2.6. Prioritizarea vehiculelor de urgență

Mai multe proiecte din domeniul sistemelor de mobilitate urbană intelligentă demonstrează integrarea tehnologiilor IoT și RFID pentru acordarea de prioritate vehiculelor de intervenție (ambulanțe, pompieri). Acestea pot activa automat semafoare verzi la intersecții prin detectarea etichetelor RFID plasate pe carosabil sau în infrastructură [28].

Un sistem complementar utilizează RFID și microcontrolere (precum NodeMCU sau ESP32) pentru a evalua densitatea traficului și a trimite avertizări vizuale către conducători auto, folosind LED-uri sau panouri digitale [29].

2.7. Sinergia RFID–GPS

Cercetări recente demonstrează eficiența utilizării combinate a GPS-ului și RFID-ului pasiv pentru a permite localizarea pe bandă („lane-level”), chiar și în condiții de vizibilitate redusă, cum ar fi ceața sau ninsoarea. Tagurile RFID sunt încorporate în asfalt și permit vehiculelor autonome să-și ajusteze traectoria precis[30].

Aceste infrastructuri RFID contribuie, de asemenea, la colectarea metadatelor despre semnele rutiere sau modificările temporare ale traficului, sprijinind decizii autonome distribuite[31].

2.8. Arhitectură de procesare locală la marginea rețelei(edge computing)

Un alt aspect esențial identificat în literatura de specialitate din domeniul IoT este rolul arhitecturii de procesare locală la marginea rețelei (edge computing), adică gestionarea și analiza datelor direct în apropierea sursei lor de generare – cum ar fi vehiculele, senzorii sau dispozitivele inteligente – fără a fi nevoie ca acestea să fie trimise mai întâi către un server central sau către platforme de tip cloud. Spre deosebire de arhitecturile clasice bazate pe cloud computing, unde datele sunt colectate de la dispozitivele periferice și trimise la centre de date aflate la distanță pentru prelucrare și decizie, edge computing reduce drastic latențele de reacție, traficul de date și dependența de o conexiune permanentă la internet. În sistemele de transport autonome, acest lucru este esențial pentru sarcini critice, precum evitarea obstacolelor, recunoașterea semnelor rutiere sau reacțiile în caz de urgență, care trebuie gestionate în timp real, fără întârzieri cauzate de transmiterea și prelucrarea la distanță. [30].

2.9. Stadiul actual al cercetărilor privind integrarea comunicației V2I în vehicule autonome

Tabelul 1. analizează stadiul actual al cercetării în literatura de specialitate: Sisteme V2I și vehicule autonome. Acesta din urmă oferă o sinteză comparativă a principalelor lucrări din literatura de specialitate relevante pentru tema abordată, axată pe sistemele de comunicație vehicul–infrastructură (V2I) aplicate vehiculelor autonome, cu accent pe controlul, semnalizarea și interacțiunea prin aplicații mobile. Sunt incluse atât proiecte comerciale, cât și prototipuri experimentale și lucrări de cercetare recentă care propun tehnologii variate – de la GPS și YOLOv8, până la soluții low-cost bazate pe RFID și microcontrolere ESP32.

Parametrii comparați includ:

- Control individual al vehiculului – capacitatea sistemului de a fi operat direct de utilizator;
- Semnal V2I – existența unei forme de comunicație între infrastructură și vehicul;
- Feedback vehicul – existența unei forme de reacție automată la semnalul primit;
- Platformă de control – tehnologia sau arhitectura utilizată pentru gestionarea vehiculului;
- Distanță de detecție / comunicare – raza de acțiune a sistemului V2I folosit;
- Aplicație mobilă – existența unei interfețe Android sau mobile pentru control și monitorizare.

Această prezentare contextualizează poziționarea contribuției propuse (Elysium RC) în raport cu alte implementări existente, evidențiind gradul său de integrare funcțională și accesibilitatea pentru testare în medii educaționale sau experimentale.

Tabelul 2.9. Stadiul actual al cercetării în literatura de specialitate.

Autor / Lucrare	Control individual	Semnal V2I	Feedback vehicul	Platformă de control	Distanță V2I	Aplicație mobilă
Masatu <i>et al.</i> [13]	✓	✓	✓	Android App	≈ 10 m (GPS)	✓
Calafate & Garcia [14]	✓	✓	✗	Wireless Signage	variabilă	✗
Rahman [15]	✓	✓	✓	YOLOv8s + MAVS	simulare	✗
Al-Ali <i>et al.</i> [22]	✓	✓	✓	BLE / ESP32	≈ 500 m	✓
Zhang <i>et al.</i> [23]	✓	✓	✓	GPS + RFID	≈ 100 m (RFID)	✗
Elysium RC	✓	✓	✓	ESP32 + Android	≈ 2 – 3 m (RFID)	✓

2.10. Lacune identificate în literatura de specialitate

Deși domeniul vehiculelor autonome și al comunicației vehicul–infrastructură (V2I) a cunoscut o dezvoltare accelerată în ultimii ani, există încă mai multe lacune semnificative în literatura de specialitate:

1. **Integrarea RFID în infrastructura intelligentă este limitată** – Deși RFID este recunoscut ca tehnologie low-cost, fiabilă și pasivă, implementarea sa ca suport pentru semne rutiere inteligente în vehicule autonome reale este puțin documentată. RFID este tratat mai mult în aplicații industriale sau pentru managementul traficului, nu în scenarii de percepție și navigație rutieră autonomă.
2. **Interfața mobilă utilizator–vehicul este insuficient cercetată** – Multe lucrări despre V2I și vehicule autonome neglijeză componenta de interacțiune cu utilizatorul. Aplicațiile mobile sunt adesea simple sau lipsite de feedback în timp real despre semne, trasee sau senzori.

2.11. Contribuțiile originale ale lucrării

Lucrarea de față răspunde acestor lacune prin propunerea unui sistem integrat, funcțional, și inovator, în mai multe direcții:

1. **Platformă reală, miniaturală, complet autonomă** – Vehiculul Elysium RC este o soluție pentru testarea reală a principiilor V2I, în condiții de laborator sau educaționale. Sistemul permite comutarea între moduri de operare (Manual, Auto, Autonomous) și înregistrarea traseelor.
2. **Aplicație Android modernă cu control complet** – Interfața mobilă, dezvoltată în Kotlin, permite controlul în timp real, vizualizarea datelor senzorilor, interpretarea semnelor RFID și urmărirea traseului. Comunicarea BLE asigură latență redusă și reacții rapide.
3. **Fuziune eficientă a tehnologiilor embedded** – Lucrarea combină ESP32, Arduino, BLE, FreeRTOS, senzori ultrasonici și afișaje cu cerneală electrică într-un sistem robust, optimizat pentru reacții rapide și procesare distribuită edge (pe vehicul), fără dependență de cloud.

Capitolul 3 – Proiectarea sistemului

3.1. Cerințe funcționale

În contextul lucrării de față, lista cerințelor funcționale urmărește ca vehiculul să îndeplinească următoarele:

- Vehiculul Elysium RC trebuie să detecteze obstacole pe cele 4 direcții (față, spate, stânga, dreapta) și să estimeze distanța la ≥ 20 Hz.
- Dacă un senzor raportează valorile „invalid” (-1 cm) ≥ 5 s, vehiculul trebuie să genereze evenimentul ACCIDENT.
- Vehiculul trebuie să transmită evenimentul ACCIDENT prin ESP-NOW într-un timp <100 ms de la declanșare.
- Fiecare semn de circulație trebuie să recepționeze mesajul primit, să actualizeze autonom afișajul E-Ink cu textul „ACCIDENT” și să notifice aplicația Android prin BLE.
- Dacă doar unul dintre semne primește mesajul, aplicația sau semnul vecin trebuie să retransmită comanda astfel încât ambele semne să fie sincronizate în < 2 s.
- Aplicația trebuie să se conecteze automat prin BLE la fiecare semn detectat (filtru ID), iar prin SPP la vehicul atunci când este necesară scrierea tag-urilor.
- Utilizatorul trebuie să poată controla individual fiecare semn (pictogramă, text) din UI, indiferent de starea vehiculului.
- Sistemul trebuie să permită adăugarea ulterioară a altor vehicule capabile să transmită ACCIDENT prin același protocol ESP-NOW.

Pentru ca prototipul să rămână performant și robust dincolo de demonstrația inițială a fluxului V2I, au fost definite o serie de cerințe funcționale măsurabile. Ele stabilesc praguri cantitative pentru atritivele de calitate ale sistemului – viteza de reacție, autonomie, fiabilitate, siguranță, portabilitate, scalabilitate și mențenanță – astfel încât orice evoluție ulterioară să poată fi evaluată riguros, nu doar „simțită” de utilizator.

- **Timp de răspuns (latență).**

Un eveniment critic, de exemplu, detectarea unui obstacol sau generarea alertei „ACCIDENT” pe vehicul, trebuie să parcurgă întregul lanț de comunicație până la actualizarea semnului E-Ink în maximum 100 ms. Intervalul acoperă citirea senzorului ultrasonic, procesarea locală, transmisia ESP-NOW în condiții de coliziune, decodarea și

reîmprospătarea ecranului. Pragul a fost calculat astfel încât, la o distanță de 0,5 m și o viteză de 0,5 m/s, vehiculul să nu avanseze mai mult de cinci centimetri înainte de primirea avertismentului vizual.

- **Autonomie energetică.**

Vehiculul trebuie să funcționeze cel puțin 45 minute în mod „Autonomous”, incluzând consumul ESP32-ului, al celor patru senzori ultrasonici, al motorului DC și al servomotorului Ackermann.

- **Fiabilitate și durabilitate.**

Rata de livrare a pachetelor ESP-NOW etichetate „ACCIDENT” trebuie să depășească 95 % într-un poligon interior de 5×5 m. Dacă apar pierderi, vehiculul retransmite la 100 ms timp de un secund, apoi la 500 ms pe durata următoarelor zece secunde. Din punct de vedere mecanic, servomotorul și ansamblul de propulsie trebuie să atingă un MTBF de minimum 10 ore de rulare continuă, suficient pentru sesiuni zilnice de laborator sau competiții.

- **Siguranță funcțională.**

Mesajul „ACCIDENT” rămâne blocat pe ecran până la primirea comenzi explice *RESET_ACCIDENT*, eliminând riscul de revenire prematură la semnalizarea standard. În cazul întreruperii conexiunii BLE în timpul tele-operării, vehiculul intră într-o stare de siguranță: motorul este oprit, iar direcția este recentrată.

- **Scalabilitate și modularitate.**

Adăugarea unui nou semn sau a unui al doilea vehicul se face prin înscrierea adreselor MAC suplimentare în lista de *peers* ESP-NOW, fără a re-compila aplicația mobilă. Separarea sarcinilor FreeRTOS (senzori, comunicație, actuatori) menține latența sub 100 ms chiar dacă se adaugă senzori suplimentari, atâtă timp cât încărcarea totală a CPU rămâne sub 40 %.

Prin stabilirea acestor criterii funcționale, proiectul atinge un echilibru între viteză de reacție, siguranță operațională și eficiență energetică, rămânând totodată ușor de întreținut și de extins către tehnologii V2X de generație următoare.

3.2. Arhitectura generală a sistemului

Sistemul propus este organizat pe o arhitectură distribuită, având ca element central microcontrolerul **ESP32**, responsabil pentru procesarea deciziilor autonome, comunicarea cu aplicația mobilă Android și coordonarea principalelor module de detecție și control. Pentru sarcinile auxiliare, cum ar fi citirea de la senzori de orientare (magnetometru și encoder) sau scrierea/recunoașterea tag-urilor RFID, este utilizat un **modul Arduino**, conectat direct la ESP32.

Comunicarea dintre vehicul și utilizator se realizează prin **Bluetooth Low Energy (BLE)**, iar navigația este asistată de un sistem de senzori (ultrasonici, GPS, IMU – MPU6050) și de feedback-ul din infrastructură, citit prin modulul **YRM100** – un cititor RFID UHF. Controlul motorului și direcției este implementat printr-un **driver L298N** (pentru motorul DC) și un **servomotor** (pentru direcție).

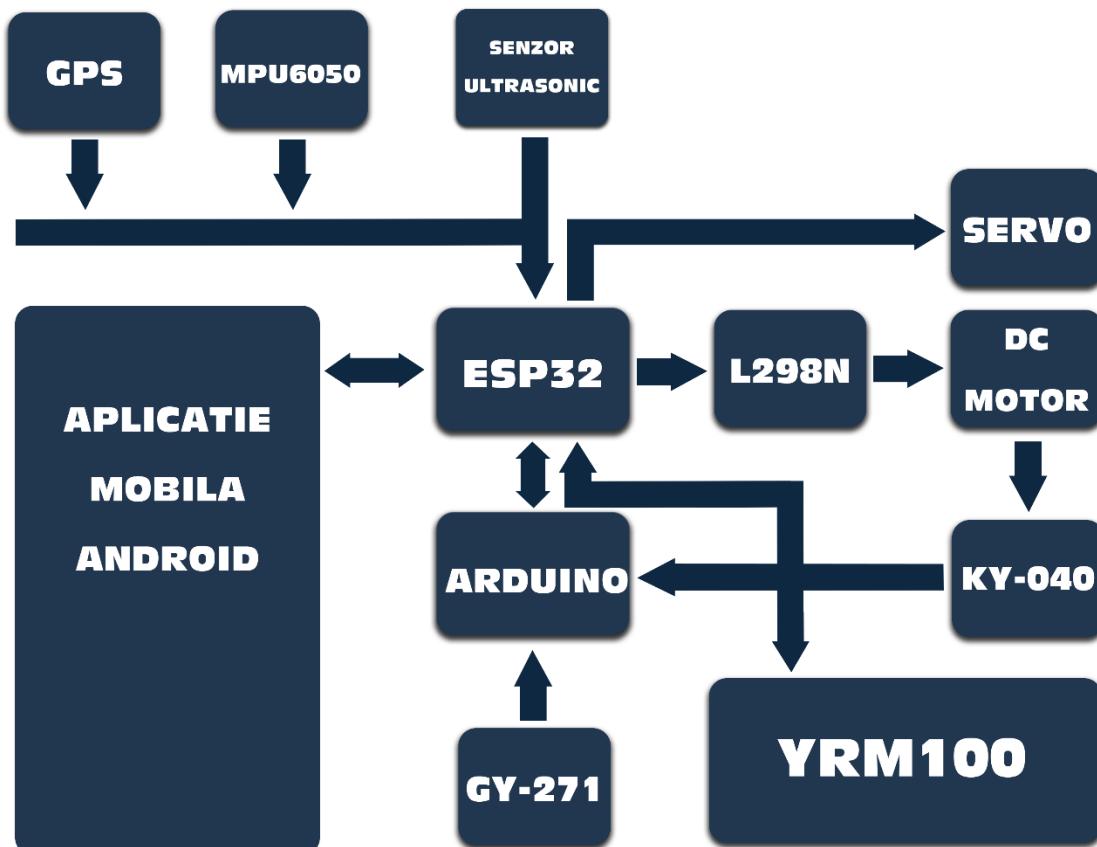


Fig. 3.1. Schema bloc a sistemului.

Datele provenite de la GPS, de la MPU-6050 și de la cei patru senzori ultrasonici sunt plasate periodic în memoria ESP32-ului, unde un fir de execuție dedicat estimează poziția vehiculului și distanța față de obstacole. Rezultatul este preluat imediat de planificatorul de mișcare, care calculează tensiunea necesară pe motor și unghiul de virare pentru a menține traseul dorit. Semnalul de accelerare este aplicat driverului L298N sub formă de PWM, iar un tren de impulsuri la 50 Hz comandă servomotorul Ackermann.

În paralel, un microcontroler Arduino citește turația encoderului KY-040 și azimutul busolei GY-271, raportându-le prin UART către ESP32, tot în sprijinul stabilizării traectoriei. Cititorul-scriitor RFID YRM100 este conectat direct la ESP32 și poate fi activat, la cererea aplicației mobile, pentru a programa tag-urile montate pe semnele de circulație.

Dialogul cu operatorul se desfășoară prin canalul Bluetooth SPP: aplicația Android alege modul de deplasare (Manual, Auto, Autonomous), ajustează parametri globali și afișează în timp real telemetria furnizată de vehicul. Codurile RFID citite de modulul YRM100 — conectat direct la ESP32 — sunt retransmise aplicației pentru a indica semnul rutier întâlnit; în această etapă tag-urile nu sunt rescrise. Dacă modulul AccidentDetector confirmă o situație critică, ESP32 difuzează imediat un mesaj ESP-NOW către semnele de circulație, care afișează textul „ACCIDENT” și trimite la rândul lor o notificare către aplicație. Vehiculul concentrează astfel colectarea datelor, deciziile în timp real, acționarea mecanică și, la nevoie, alertarea infrastructurii și a utilizatorului.

3.3. Arhitectura hardware- Selecția componentelor

Selecția componentelor s-a realizat pe criterii de compatibilitate electrică, disponibilitate locală și raport performanță-consum, astfel încât fiecare piesă să acopere o funcție critică fără a supradimensiona bugetul de putere al vehiculului. Tabelul 3.4. sintetizează componentele esențiale ale platformei Elysium RC, indicând pentru fiecare rolul funcțional și argumentul tehnic al alegerii. Se observă o împărțire netă între unitatea centrală ESP32, dedicată calculelor în timp real și comunicațiilor radio, și micro-controlerul auxiliar Arduino Uno, care preia sarcinile cu frecvență scăzută sau care solicită canale analogice suplimentare. Alături de aceste nuclee de procesare sunt listate modulele de percepție (IMU, busolă, GPS, ultrasonic) și dispozitivele de acționare sau interfațare (driver L298N, servomotor, encoder, cititor RFID), fiecare justificat prin precizie, interval de măsurare sau consum energetic. Această etapă de selecție stabilește baza hardware pe care se construiește arhitectura de control detaliată în subsecțiunile următoare.

Tabelul 3.4. –Tabel privind selecția componentelor Elysium RC.

Componentă	Funcție principală	Justificare tehnică
ESP32	Unitate centrală: procesare, comunicație BLE/Wi-Fi, multitasking, scriere RFID	Microprocesor Tensilica Xtensa LX6, 32-bit, dual-core (240 MHz), 520 KB SRAM, suport FreeRTOS, conectivitate BLE + Wi-Fi. Include 34 de pini GPIO, dintre care ~25 sunt utilizabili pentru aplicații generale (I2C, SPI, PWM, ADC, DAC) [33], [34]
Arduino Uno	Gestionare senzori secundari și comunicare cu ESP32	Bazat pe ATmega328P, cu 14 pini digitali și 6 pini analogici, ideal pentru sarcini secundare: citire tensiune, encoder, IMU. Utilizat și din considerente de disponibilitate locală [35]
MPU6050	IMU (accelerometru + giroscop) pentru orientare	Senzor digital pe magistrala I ² C, cu 3 axe acceleratie + 3 axe rotație, frecvență reglabilă 8 Hz – 1 kHz, precis și eficient pentru monitorizarea înclinării și stabilității vehiculului [36]
GY-271	Magnetometru digital (busolă)	Bazează pe QMC5883L, măsoară câmp magnetic pe 3 axe, ideal pentru determinarea direcției absolute (azimut), conectivitate I ² C [37]
GPS Module	Determinarea poziției și traectoriei	Modul NEO-6M, acuratețe poziționare <2.5 m CEP, 50 canale de tracking, interfață UART/NMEA, util pentru înregistrarea și urmărirea traseului [38]
HC-SR04	Detectarea obstacolelor în proximitate	Senzor ultrasonic cu rază de măsurare între 2–400 cm, precizie ± 3 mm, unghi de detecție $\approx 15^\circ$, semnal logic 5V trigger/echo [39], [40]
L298N	Driver pentru motor DC	Full H-Bridge driver pe 2 canale, susține curenti până la 2 A per canal, tensiune motor 5–35 V; include protecție termică [41]
Servomotor (MG996R)	Controlul direcției vehiculului	Cuplu mare (până la 11 kg·cm la 6 V), răspuns rapid, poziționare unghiulară precisă prin PWM; utilizat pentru mecanismul de direcție tip Ackermann [42]
KY-040	Encoder rotativ pentru măsurarea rotației roților	Encoder incremental cu 20 pași/rev, semnale digitale A/B + buton, ideal pentru măsurarea rotației roților în mod autonom [43]

YRM1003	Cititor RFID UHF pentru comunicația V2I	Suportă standardul ISO 18000-6C / EPC Gen2, citire/scriere taguri pasive, frecvență 860–960 MHz, rază de citire de până la 5 metri [44]
----------------	---	---

3.4. Arhitectura software

Software-ul vehiculului este împărțit între două microcontrolere și mai multe module majore, grupate într-o ierarhie de directoare. Nucleul care rulează pe ESP32 se află în directorul core. Fișierul TaskManager.cpp constituie punctul de intrare al scheduler-ului FreeRTOS: pornește fire distincte pentru achiziția datelor de la senzorilor, planificarea mișcării, difuzarea alertelor și gestionarea comunicației Bluetooth, iar parametrii fiecarui fir (prioritate, perioadă) sunt definiți centralizat în TaskManager.h, putând fi ajustați rapid fără a modifica restul codului.

Datele de mediu sunt colectate de modulele din sensors/. UltrasonicSensors.cpp declanșează secvențial cei patru senzori ultrasonici HC-SR04 și publică, printr-un buffer FIFO, distanțele deja filtrate; IMUManager.cpp furnizează orientarea, iar GPSManager.cpp determină poziția globală.

Deciziile critice se iau în directorul alerts/. În AccidentDetector.cpp distanțele citite HC-SR04 sunt analizate pentru a depista obstacole persistente; la nevoie, modulul apelează BroadcastManager.cpp și trimite, prin ESP-NOW, un pachet serializat în structura ElysiumMessage definită în MessageTypes.h. Semnele rutiere decodifică imediat tipul evenimentului și afișează pictograma adecvată.

Controlul mișcării este tratat în motion-control/. Interfața cu operatorul este concentrată în BluetoothManager.cpp (din core/). Două canale rulează concomitent: un SPP clasic pentru comenzi text și telemetrie, respectiv un canal BLE rezervat pentru extensii viitoare. Aplicația Android folosește comenzi de un singur caracter „M” pentru manual, „A” pentru auto, simplificând logica de parsare de pe microcontroler.

Partea Arduino este condensată într-un singur sketch, ArduinoController.ino, plasat în directorul Arduino/. Acesta configerează busola GY-271, encoderul KY-040 și legătura UART cu ESP32, iar la fiecare 100 ms transmite un pachet binar care conține orientarea și poziția roților de pe puntea spate.

În ansamblu, arhitectura software se rezumă la module independente care colaborează prin primitive FreeRTOS și interfețe minimalistă: update() pentru senzori, send() pentru

comunicație și execute() pentru actuatori. Această disciplină structurală sporește lizibilitatea codului și ușurează întreținerea pe termen lung.

Capitolul 4. Implementare, dezvoltare și testare

4.1. Arhitectura hardware – viziune detaliată

Sistemul autonom Elysium RC este fundamentat pe o arhitectură embedded distribuită, concepută pentru a permite procesare paralelă, comunicare între module și o bună scalabilitate. Această arhitectură implică o împărțire a responsabilităților între mai multe microcontrolere și componente periferice, conectate între ele prin protocoale standard de comunicație digitală și alimentate printr-o infrastructură energetică bine segmentată.

Unitățile de procesare:

- ESP32, microcontrolerul principal, are rolul de nucleu de decizie și comunicație. El gestionează:
 - toate procesele critice în timp real (ex: evitarea obstacolelor, interpretarea semnelor rutiere RFID),
 - comunicarea Bluetooth Low Energy (BLE) cu aplicația mobilă Android pentru control și monitorizare,
 - rularea sistemului de operare FreeRTOS, care permite multitasking (gestionarea simultană a mai multor sarcini cu priorități diferite),
 - interpretarea informațiilor provenite de la senzorii conectați direct (ultrasunete, encoder, GPS, RFID).
- Arduino Uno, configurat ca coprocesor, îndeplinește roluri auxiliare esențiale:
 - preia și procesează datele venite de la senzori secundari, precum giroscopul (MPU6050), senzorul de tensiune, encoderul rotativ,
 - transmite aceste date prelucrate către ESP32 prin interfață serială UART.
 - se ocupă și de controlul servomotorului de direcție

Această separare a sarcinilor pe două controlere asigură o distribuție eficientă a resurselor de calcul, evitând suprasolicitarea ESP32, mai ales în momente critice (navigare autonomă, decizii bazate pe mai mulți senzori simultan etc.).

Platforma Elysium RC integrează o varietate de senzori și module periferice, fiecare cu o funcție specifică:

- Senzori ultrasonici HC-SR04 (x4) – plasați strategic pentru detectarea obstacolelor din față, spate și laterale. Transmit impulsuri sonore și măsoară timpul de întoarcere pentru estimarea distanței față de obiecte.

- GY-271 (busolă digitală) – oferă orientarea absolută (azimut), permitând vehiculului să-și corecteze direcția față de nordul magnetic.
- GPS NEO-6M – modulează traseul parcurs și permite cartografierea simplă a deplasării.
- YRM103 RFID – modulează comunicarea V2I, permitând vehiculului să citească tag-uri UHF RFID plasate în infrastructură (pe semnele de circulație).
- KY-040 encoder rotativ – oferă feedback asupra rotației roților, esențial pentru măsurarea distanței parcuse și controlul virajelor.

Toate aceste componente sunt alimentate dintr-o baterie Li-Ion de capacitate mare (12V, 60.000 mAh), aleasă pentru a asigura o autonomie extinsă. Energia este distribuită în sistem prin două module step-down:

- HW-131 – convertește 12V în 5V și 3.3V, alimentând ESP32, senzorii HC-SR04, KY-040, precum și LLC-ul de conversie logică.
- HW-095 – direcționează 5V spre Arduino, giroscop și servomotor, oferind o linie energetică separată pentru componentele controlate direct de Arduino.

Separarea canalelor de alimentare este importantă pentru a reduce interferențele și fluctuațiile de tensiune, în special între motoare și senzori.

Arhitectura include mai multe tipuri de interfețe:

- UART (Serial): pentru comunicația ESP32 ↔ Arduino și GPS ↔ ESP32.
- I2C: pentru senzori precum MPU6050 și GY-271.
- Digital I/O: pentru encoder, HC-SR04 și controlul servomotorului.
- BLE: pentru interfața cu aplicația Android.

Această structură permite o comunicare robustă și modulară între componente, fiind totodată flexibilă în privința adăugării sau înlocuirii modulelor hardware.

4.2. Schema electrică

Figura 4.2.1. ilustrează schema electrică operațională a vehiculului Elysium RC. Conexiunile sunt grupate după cele patru magistrale de alimentare și de referință:

- 12V
- 5V
- 3,3V
- GND

Acestea sunt reprezentate ca fire continue care străbat întregul desen. Deși convertoarele step-down nu apar explicit, prezența celor trei nivele de tensiune indică faptul că distribuția energetică este deja realizată “în amonte”; schema se concentreză pe traseele logice și pe raportul dintre module.

În centru găsim placa ESP32-Wroom, nodul de procesare și comunicație. Patru senzori ultrasonici HC-SR04 sunt poziționați în jurul său pentru acoperire totală; fiecare pereche trigger/echo ajunge pe câte doi pini GPIO dedicați, păstrați adiacenți pentru sincronizare. Tot pe GPIO-uri individuale merg semnalele de control către driverul H-bridge L298N (direcție, PWM pentru motorul DC) și impulsul de comandă pentru servomotorul MG996R (direcția roților frontale).

Interfața serială este dublă. Un prim canal UART face legătura cu Arduino Uno, care funcționează ca coprocesor: citește busola digitală GY-271, encoderul rotativ KY-040 și accelero-giroscopul MPU-6050, condensând măsurările și trimițându-le la 115200 bps către ESP32. Al doilea canal UART se conectează la cititorul UHF RFID YRM103; prin acesta, identificatorii de semn rutier detectați în infrastructură sunt transferați direct către nucleul principal pentru interpretare și eventual retransmisie către aplicația mobilă.

Motorul DC este singurul consumator de pe bara de 12 V ilustrată, însă în implementarea reală el este alimentat direct de la acumulatorul de 12V.

În ansamblu, schema confirmă topologia „stea” cu ESP32 în centru, toate componentele periferice fiind racordate direct sau prin interfețe digitale standard. Această configurație simplifică depanarea, permite viitoare extinderi (de pildă un senzor LiDAR suplimentar pe I²C).

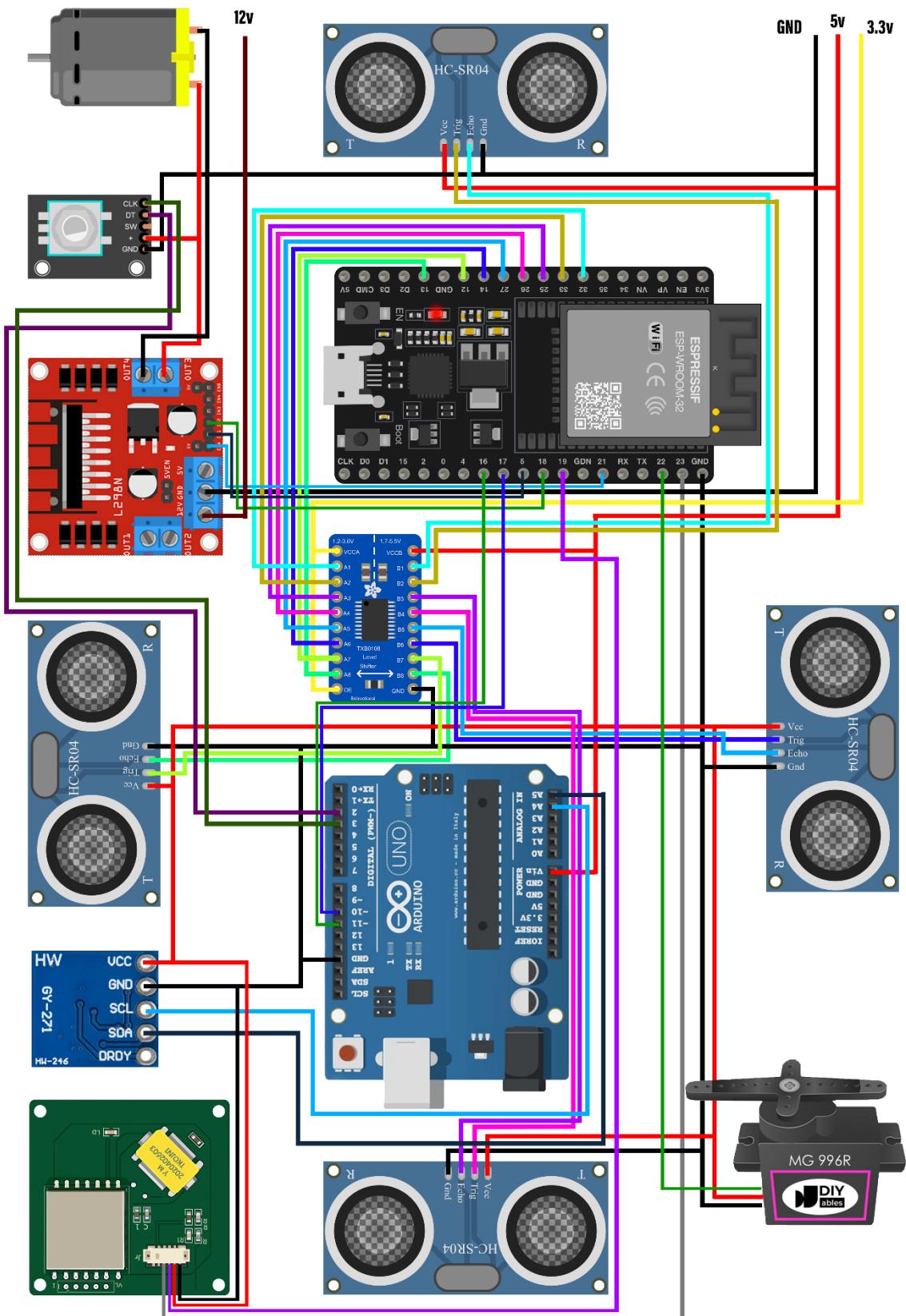


Fig. 4.1. Schema electrică

Figura 4.2.2. ilustrează cablajul efectiv al prototipului Elysium RC, văzut de deasupra pentru a evidenția traseele fizice ale firelor dintre module. Fotografia confirmă implementarea practică a topologiei „stea” descrise anterior: ESP32 se află în centru, iar perifericele sunt dispuse radial, cu legături color-codate pentru alimentare și date.

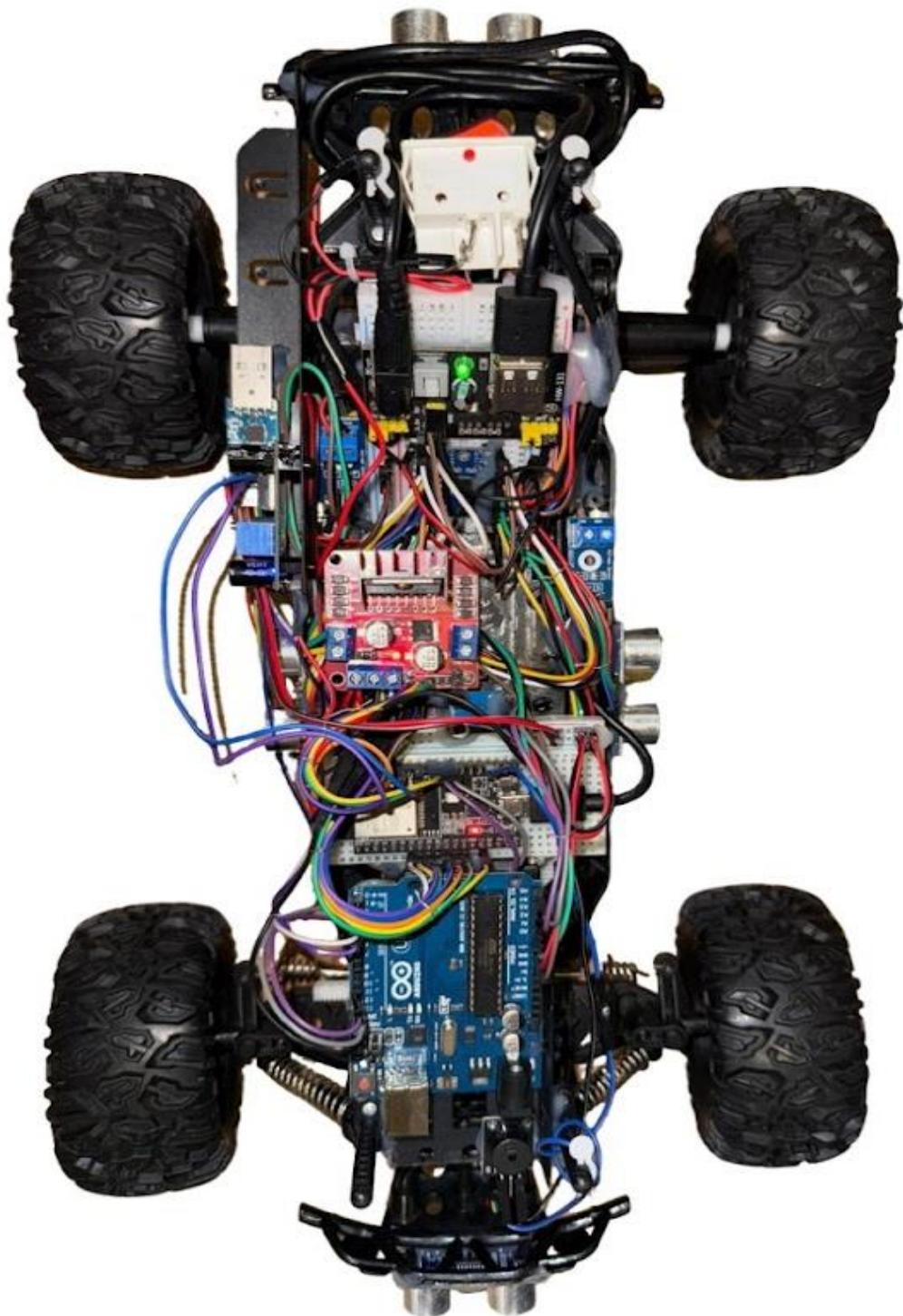


Fig. 4.2. Elysium RC – cablaj.

Figura 4.2.3 prezintă prototipul Elysium RC în configurația completă, cu carenajul superior montat.



Fig. 4.3. Elysium RC – imagine completa.

4.3. Implementare software

Arhitectura software a prototipului se sprijină pe trei elemente independente ce cooperează prin legături radio ori seriale, astfel încât fiecare subsistem să-și păstreze autonomia logică, dar și să poată fi testat separat.

Primul element este aplicația mobilă, realizată în Kotlin, organizată MVVM și compilată pentru nivelul de SDK 24+. Ea concentrează interfața om–mașină, supraveghează starea conexiunilor și emite comenzi printr-un canal Serial Port Profile (BLE-SPP).

Al doilea element îl constituie firmware-ul de pe ESP32, nucleul de decizie în timp real. Codul rulează pe FreeRTOS cu două seturi de sarcini: pe core-ul 0 funcționează bucla de control a mișcării, care citește poziția fuzionată la 20 Hz și emite comenzi PWM în 50 ms; pe core-ul 1 se execută stratul de comunicație (BLE, ESP-NOW) și diagnosticul de sistem. Hardware-ul este accesat prin drivere dedicate (I²C, SPI, UART) ascunse în namespace-uri sensors, motion_control și alerts; drept urmare, migrarea către un alt microcontroler ESP32-S3 sau adăugarea unui LiDAR necesită doar implementarea unui nou driver ce respectă interfața update()/Reading.

Ultima componentă este firmware-ul Arduino, redus la un singur sketch ArduinoController.ino. Canalul UART operează la 115200 bps și folosește un protocol TLV (Type-Length-Value); la fiecare 100 ms Arduino trimit un pachet binar ce conține orientarea absolută și poziția roților de pe puntea spate, iar ESP32-ul răspunde numai când sunt necesare recalibrări, reducând traficul serial la sub 5 % din timp.

Această partaționare respectă principiul separării responsabilităților: interfața grafică rămâne izolată pe mobil, logica deterministă de control este împărțită între două microcontrolere pe criteriul ratei de actualizare, iar fiecare bloc poate fi repornit, actualizat OTA sau înlocuit fără a perturba celelalte. În cele ce urmează vor fi detaliate fișierele și modulele critice din fiecare director-core/, sensors/, motion_control/, respectiv aplicația Android.

4.4. Aplicația Android

Aplicația mobilă este singura interfață între operator și vehicul; din acest motiv, proiectul a fost organizat cât mai simplu și transparent. Toate fișierele Kotlin se află în app/src/main/java/com/example/elysumrc/, iar layout-urile XML asociate în res/layout/. Această structură plană, fără sub-directoare, scurtează drumul până la fiecare fișier și face revizuirile de cod mai rapide. Lipsa sub-directoarelor intermediare elimină căutările inutile, iar numele claselor reflectă direct rolul lor funcțional (de pildă BluetoothManager, MainActivity, SplashActivity). Izolarea fiecărui modul este încurajată de convenția „o singură responsabilitate pe fișier”, astfel încât modificarea unui ecran sau a unui driver de comunicație nu afectează restul codului. În paragrafele următoare vom parcurge pe rând aceste fișiere, explicând de ce există, cum interacționează și care sunt extensiile vizate pe termen mediu.

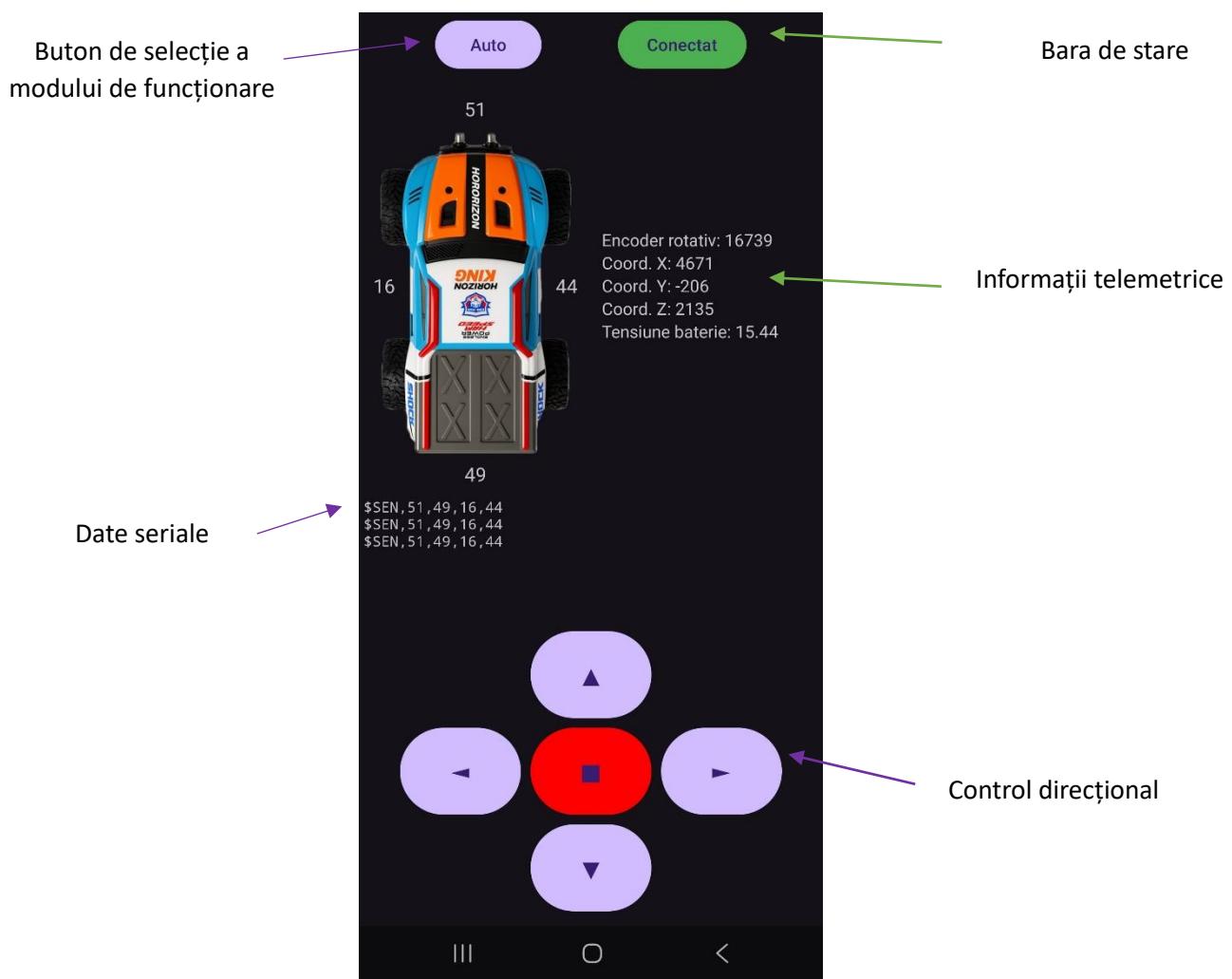


Fig. 4.4. Ecranul de pornire al aplicație Android.

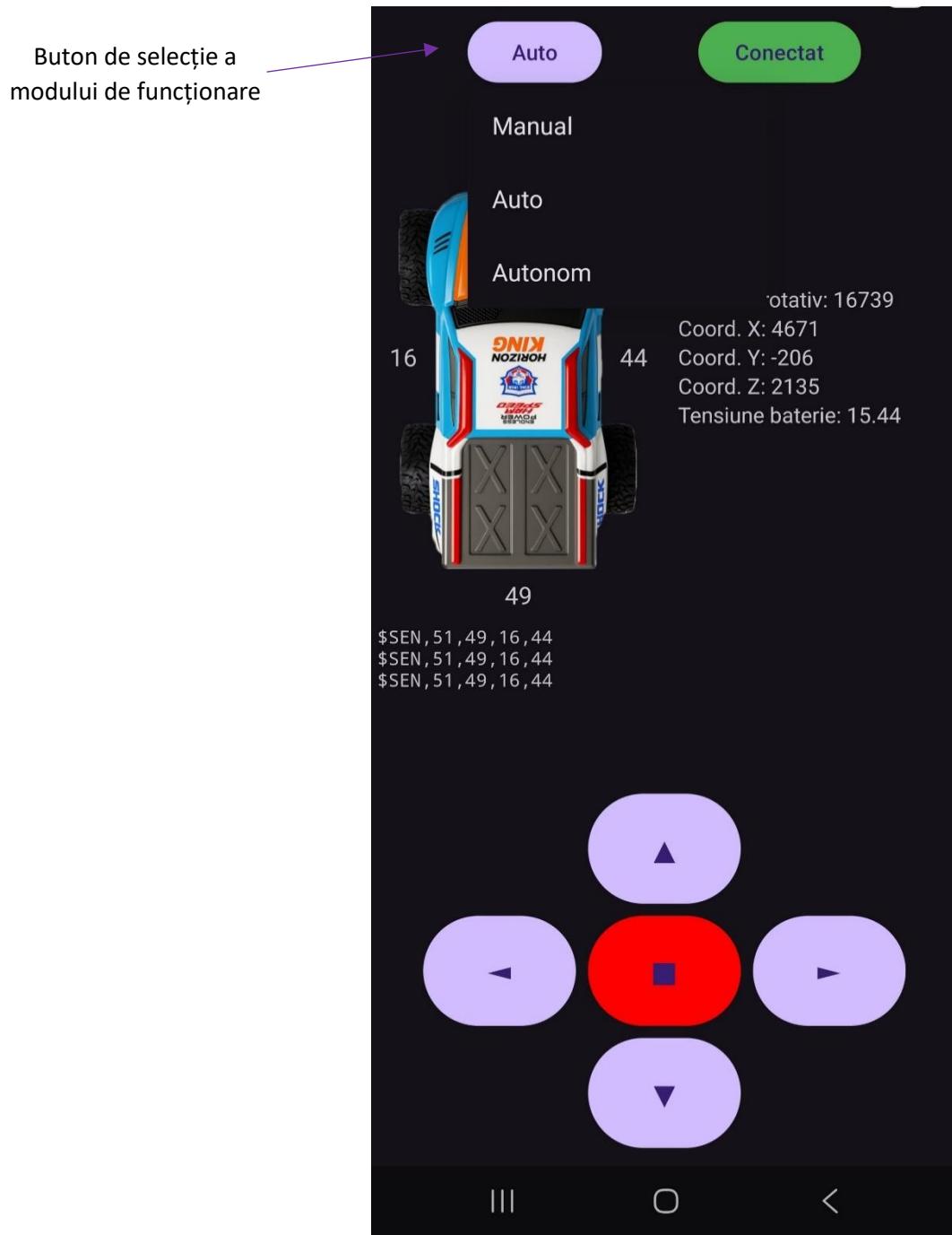


Fig. 4.5. Interfața aplicației Android – mod de control al vehiculului autonom (manual, auto, autonom)

SplashActivity.kt

Ecranul de întâmpinare afișează layout-ul activity_splash.xml, ascunde bara de acțiune și pornește un Handler care, la cinci secunde de la lansare, pornește MainActivity și închide activitatea curentă. Intervalul acoperă inițializarea Bluetooth pe dispozitive mai vechi și oferă spațiu pentru identitatea vizuală fără a încărca memoria: fișierul însumează aproximativ douăzeci de linii de cod.

MainActivity.kt

Aceasta este atât orchestrator, cât și strat de prezentare. La pornire verifică suportul și starea adaptorului Bluetooth, solicită permisiunile BLUETOOTH_CONNECT și BLUETOOTH_SCAN prin ActivityResultContracts.RequestMultiplePermissions, iar în caz de succes invocă connectToESP32().

Layout-ul activity_main.xml conține cinci butoane direcționale plasate cu ConstraintLayout, patru TextView-uri care afișează în timp real distanțele ultrasonice și un indicator al stării conexiunii. Fiecare buton atașează un OnTouchListener; la evenimentul ACTION_DOWN se trimit codul de mișcare („F”, „B”, „L”, „R”), iar la ACTION_UP codul „S”, emulând un joystick continuu fără a bloca firul UI.

Telemetria sosește printr-un callback furnizat de BluetoothManager: șirurile CSV de forma „dF,dB,dL,dR” sunt despicate, iar valorile sunt înscrise în partea de sus a ecranului printr-un apel runOnUiThread. Ciclul de viață al conexiunii este curățat în onDestroy, eliminând surgerile de resurse.

Sevența de cod din Fig. 4.4.1. surprinde: inițierea conexiunii, trimitera unei comenzi la apăsare / oprire la ridicare și actualizarea în timp real a unei valori de senzor primite de la vehicul.

```

// după ce permisiunile sunt acordate ...
bt.connectToDevice("ElysiumRC",
    onConnected = { status("Conectat") },
    onError     = { status(it) })

// control „push-to-move”
buttonForward.setOnTouchListener { _, e ->
    bt.sendCommand(if (e.action == ACTION_DOWN) "F" else "S"); true
}

// telemetrie CSV → UI
bt.setDataCallback { csv -
    val f = csv.substringBefore(",")           // df
    runOnUiThread { distanceFrontText.text = "Față: $f cm" }
}

```

Fig. 4.6. Secvență din MainActivity.kt.

BluetoothManager.kt

Fișierul **BluetoothManager.kt** concentrează toată logica de comunicație pentru canalul Serial Port Profile (SPP), astfel încât activitățile de interfață să poată rămâne dedicate prezentării datelor, nu detaliilor de transport. La inițializare, metoda `connectToDevice()` inspectează lista de dispozitive împerecheate și cauță intrarea cu denumirea *ElysiumRC*. Dacă o găsește, deschide un `BluetoothSocket` RFCOMM folosind UUID-ul standard al profilului Serial Port, apoi configerează fluxurile de intrare și ieșire.

Întreaga secvență de conectare rulează pe un fir de execuție separat, pentru a nu bloca firul principal al interfeței. Rezultatul (succesul sau motivul eșecului) este transmis înapoi printr-un Handler pe firul de UI, permitând afișarea imediată a stării către utilizator. Un al doilea fir, lansat de `startListening()`, citește neîntrerupt din `InputStream`, convertește octetii în text și propagă mesajele primite printr-un lambda `dataCallback`; astfel, telemetria ajunge în activitate fără să fie necesare sonde de tip „poll”.

Pentru trimitera comenzilor se folosește metoda `sendCommand()`. Aceasta deschide un fir dedicat care scrie sirul de caractere (urmat de marcajul de sfârșit de linie) în `OutputStream`, eliminând riscul ca interfață să devină ne-responsivă. Clasa oferă, în plus, două utilitare rapide:

- `isBluetoothSupported()`
- `isBluetoothEnabled()`

Acestea verifică existența hardware-ului și starea adaptorului, precum și permișiunea BLUETOOTH_CONNECT, înainte de a iniția orice operație.

Aplicația Android urmărește deliberat un design minimalist, dar complet funcțional: întreaga logică de interfață om-mașină este condensată în trei fișiere Kotlin—**SplashActivity**, **MainActivity** și **BluetoothManager**—plus un singur layout XML. Această simplitate structurală reduce suprafața de mențenanță și scade timpul de compilare, dar nu sacrifică nicio cerință operațională. Modulul **BluetoothManager** încapsulează întregul transport pe Serial Port Profile, lăsând **MainActivity** să se concentreze exclusiv pe prezentare și pe ciclul de viață al activității; separarea strictă dintre stratul vizual și infrastructura de comunicație garantează că adăugarea unor funcții noi—de pildă monitorizarea tensiunii bateriei sau afișarea temperaturii controller-ului—presupune doar extinderea contractului din dataCallback, fără refactorări adânci.

Comenzile pentru modurile Manual, Auto și Autonomous sunt codificate în ASCII pe un singur octet, ceea ce menține latența sub 20 ms și permite un parser O(1) pe microcontroler. Aceeași schemă de codificare simplă face ca aplicația să fie tolerantă la pierderi de pachete: un singur caracter corrupt este pur și simplu ignorat, iar următorul mesaj validează din nou starea. În absența oricărora biblioteci terțe, fișierul build.gradle.kts generează un APK de sub 2 MB, compatibil cu dispozitivele Android începând cu versiunea 8.0 (API 26)—prag ce acoperă peste 95 % din piața actuală.

Fig.4.4.1. reprezintă ecranul de start al aplicației Android car. În această etapă, interfața afișează doar indicatorul de stare al conexiunii Bluetooth și butoanele de control ale mașinii. După stabilirea legăturii cu vehiculul, pagina se extinde automat: în partea centrală apar cele trei moduri de operare: **Manual**, **Auto** și **Autonomous**, iar în colțul din dreapta sus devine vizibil un buton suplimentar pentru **Setări**, de unde se pot ajusta parametrii aplicației (unități de viteză, praguri de alertă, întreținerea conexiunii).

4.5. Firmware-ul ESP32

Codul care guvernează logica de bord a vehiculului este păstrat în directorul firmware/Elysium RC/ESP32/. Structura a fost concepută astfel încât fiecărui domeniu funcțional să-i corespundă un sub-director, iar dependențele dintre module să rămână strict unidirectionale (de sus în jos). Pentru a facilita orientarea rapidă în codul sursă, Tabelul 4.5.1. inventariază directoarele și fișierele esențiale din firmware-ul destinat microcontrolerului ESP32, precizând rolul funcțional al fiecărui modul.

Tabelul 4.5.1. Inventar directoare și fișiere.

Director	Fișier	Rol principal
core	TaskManager.cpp/h	Configurează și lansează firele FreeRTOS; gestionează timpii de execuție și prioritățile fiecărui task.
core	BluetoothManager.h	Înfașoară canalul SPP pentru aplicația mobilă și transportul ESP-NOW pentru broadcast; oferă API thread-safe de send/receive.
sensors	UltrasonicSensors.cpp/h	Pornesc secvențial cei patru senzori HC-SR04, filtrează ecurile și depozitează rezultatele într-un ring-buffer.
sensors	RFIDManager.cpp/h	Dialog UART cu cititorul YRM103; decodifică etichete EPC și publică evenimentele V2I într-o zonă RAM partajată.
motion-control	DCMotor.cpp/h	Mapare duty-cycle ↔ viteză; limitare de curent și generare PWM la 20 kHz
motion-control	ServoMotor.cpp/h	Trimite impulsuri de 50 Hz pentru servomotorul MG996R pe GPIO 25; include funcție de slew-rate pentru viraje line.
alerts	AccidentDetector.cpp/h	Analizează distanțele HC-SR04; dacă obstacolul persistă > 5 s, solicită BluetoothManager să difuzeze alerta prin ESP-NOW.
feedback	BuzzerManager.cpp/h	Generează sunet de avertizare local; rulează într-un task cu prioritate scăzută.

TaskManager

TaskManager.cpp joacă rolul de „turn de control” pentru firmware: în acest fișier se construiește coada inter-task, se definesc parametrii de periodicitate și prioritate, iar apoi se lansează firele care rulează permanent pe cele două nuclee ale ESP32. Orice ajustare a frecvenței de execuție, a nivelului de prioritate sau a numărului de task-uri se face în această singură locație, asigurând o configurare centralizată și ușor de auditat a sistemului de operare în timp real. Fig. 4.5.1. redă un fragment relevant din TaskManager.cpp, concentrat pe etapa de pornire a sistemului de execuție. Se observă, mai întâi, apelul xQueueCreate, prin care se

instanțiază coada globală FreeRTOS ce va media schimbul de mesaje între module. În continuare, două apeluri succesive la xTaskCreate pornesc firele - ObstacleDetection și Buzzer, fiecare primind o dimensiune de stivă și un nivel de prioritate stabilite tot aici.

```
void Tasks_init() {
    queue = xQueueCreate(5, sizeof(char[50]));           // coadă pentru mesaje inter-task

    xTaskCreate(obstacleDetectionTask, "Obstacle", 2048, NULL, 1, NULL);
    xTaskCreate(buzzerTask,                 "Buzzer",   2048, NULL, 1, NULL);
}
```

Fig. 4.7. Secvență din TaskManager.cpp.

BluetoothManager

BluetoothManager constituie un înveliș subțire peste biblioteca oficială *BluetoothSerial*, menținând într-un singur loc operațiile de inițializare, testare a conexiunii și schimb de mesaje text. Implementarea este reținută integral în fișierul antet, tocmai pentru a preveni fragmentarea codului și a facilita compilarea. Fig. 4.5.2. surprinde nucleul acestor funcții: secvența de pornire a modulului radio, verificarea stării legăturii și rutina de expediere a datelor către aplicația mobilă.

```
void begin() {
    SerialBT.begin("ElysiumRC");                      // anunță dispozitivul SPP
    Serial.println("Bluetooth device started...");
}

void sendData(const String& data) {                   // TX non-blocking
    if (SerialBT.connected()) SerialBT.println(data);
}

String receiveData() {                                // RX line-based
    return SerialBT.available() ?
        | | | SerialBT.readStringUntil('\n').trim() : "";
}
```

Fig. 4.8. Secvență din BluetoothManager.

ElysiumRC.ino

Fișierul ElysiumRC.ino îndeplinește rolul unui dirijor: adună toate modulele specializate, configuraază perifericele și predă execuția sistemului de operare în timp real. Codul poate fi urmărit în trei secțiuni distincte. Partea de început (aprox. liniile 2 – 28) include

anteturile tuturor componentelor – BluetoothManager, DCMotor, ServoMotor, UltrasonicSensors, RFIDManager și.a. – și declară instanțele globale: obiectul Bluetooth, pinii RX/TX, servomotorul, precum și constantele de configurare. Această zonă concentrează dependențele, permîțând o imagine imediată asupra resurselor hardware utilizate.

Secțiunea setup() (liniile 33 – 91) deschide porturile seriale pentru debugging și pentru cititorul RFID, pornește fiecare subsistem prin apeluri consacrate de tip *_init() și rulează un auto-test scurt (motor DC, servomotor, cititor RFID) pentru diagnoză la pornire. În final este invocată funcția Tasks_init(), după care controlul trece complet la FreeRTOS.

Funcția loop() (liniile 94 – 154) rămâne un scheduler ușor, restrâns la trei sarcini ciclice: interpretarea comenziilor Bluetooth, expedierea telemetriei către aplicația mobilă și actualizarea modulului RFID. Operațiile costisitoare – detectia obstacolului, generarea avertizărilor acustice, difuzarea ESP-NOW – rulează deja în task-uri dedicate, astfel încât loop() poate menține un simplu delay(10) fără a risca blocaje.

Fragmentul din Fig. 4.5.3. surprinde secvența de procesare a comenziilor manuale: caracterul ASCII primit prin SPP este mapat direct la acțiunea corespunzătoare (accelerație, frânare, viraj), iar rezultatul este publicat imediat în coada FreeRTOS, menținând latența de comandă sub 20 ms.

```
String command = btManager.receiveData();           // citire non-blocking

if (command == "F")      DCMotor(true,  false);   // înainte
else if (command == "B") DCMotor(false, true);    // înapoi
else if (command == "S") {                           // stop total
    DCMotor(false, false);
    ServoMotor(CENTER);
}
else if (command == "L") ServoMotor(LEFT);         // viraj stânga
else if (command == "R") ServoMotor(RIGHT);        // viraj dreapta
```

Fig. 4.9. Secvență din ElysiumRC.ino.

Pentru a evidenția coeziunea arhitecturii interne, următoarele paragrafe prezintă pe scurt rolul celorlalte module-cheie din firmware-ul ESP32. Fiecare responsabilitate este încapsulată într-un fișier dedicat, iar colaborarea se face prin primitive FreeRTOS și variabile partajate controlate, ceea ce permite lizibilitate, testabilitate și extindere rapidă.

alerts/AccidentDetector.cpp.h rulează într-un task propriu pe nucleul 1, supraveghează distanțele distanceFront, distanceBack, distanceLeft și distanceRight publicate de senzorii ultrasonici și aplică un filtru de persistență: dacă un obstacol se menține sub 15 cm timp de 100 de iterații, modulul declanșează starea de pericol. În momentul declanșării trimite EVENT_ACCIDENT prin BluetoothManager.broadcastEvent() și activează buzzer-ul; resetarea se poate face doar prin comanda ASCII „RESET_ACCIDENT” primită pe canalul SPP.

core/Modules.cpp asigură legarea la timp de compilare a tuturor componentelor: aici sunt instanțiate managerul Bluetooth, driverul de senzori, actuatoarele etc., astfel încât fișierele individuale să acceseze obiectele globale prin extern. Concentrarea dependențelor într-un singur loc simplifică testarea pe banc, unde modulele pot fi înlocuite rapid cu dubluri (*mocks*).

feedback/BuzzerManager.cpp.h configerează ieșirea PWM destinată difuzorului și oferă o interfață minimalistă `buzzerOn(freq)` / `buzzerOff()`. Logica internă variază frecvența în funcție de direcția obstacolului și ajustează ciclul de lucru proporțional cu apropierea, fiind invocată atât de AccidentDetector, cât și de UltrasonicSensors pentru avertizări locale.

motion-control/DCMotor.cpp.h pilotează motorul cu perii printr-un pod H-Bridge folosind două GPIO-uri de sens și un canal PWM la 20 kHz. Include funcții de inițializare, comenzi forward/reverse și un regulator PID ușor, cu coeficienții K_p și K_i declarati în antet pentru calibrare rapidă în teren.

motion-control/ServoMotor.cpp.h controlează servomotorul MG996R (geometrie Ackermann) prin biblioteca ESP32Servo. Definește pozițiile simbolice LEFT, CENTER și RIGHT, introduce un factor de offset calibrabil și impune o limită de *slew-rate* de 45 °/s pentru a preveni supracomanda.

sensors/UltrasonicSensors.cpp.h orchestrează cele patru HC-SR04 într-o singură rutină: declanșează secvențial fiecare senzor, măsoară timpul de zbor, elimină ecurile parazite și aplică o medie mobilă pe patru eșantioane. Variabilele de distanță pe 360 ° sunt publicate ca atomi, iar frecvența de actualizare de 20 Hz este suficientă pentru viteza maximă de 1 m/s a vehiculului.

sensors/RFIDManager.cpp.h inițializează cititorul YRM103 la 115200 bps pe UART1, execută cicluri de inventariere, parsează pachetele EPC și stochează cel mai recent tag în lastCardID. Expune metodele `isCardPresent()` și `RFIDManager_update()`, astfel încât bucla principală să poată trimite ID-ul către aplicația Android sau să comute în modul de scriere RFID.

Prin această structură, firmware-ul Elysium RC demonstrează o funcționare modulară în care fiecare fișier îndeplinește o singură funcție clar definită, iar schimbul de date se face prin canale RTOS.

4.6. Firmware-ul Arduino

În arhitectura vehiculului Elysium RC, placa Arduino UNO funcționează ca subsistem auxiliar: preia sarcinile cu frecvență redusă – busolă, encoder roți, monitorizare baterie – și trimit date sintetizate către ESP32 pe un al doilea canal serial. Tot codul aferent este concentrat într-un singur fișier, firmware/Elysium RC/ArduinoController.ino.

Fișierul începe prin a include bibliotecile necesare — Wire.h și MechaQMC5883.h pentru magistrala I²C și busola QMC5883, SoftwareSerial.h pentru canalul UART software dintre pinii 10 și 11, iar pinii 2 și 3 sunt declarați drept intrări pentru encoderul incremental al roții din spate. Numărul impulsurilor este stocat într-o variabilă volatilă counter, actualizată de două rutine de întrerupere, encoderA() și encoderB(). După aceste declarații se creează instanța globală a busolei, MechaQMC5883 qmc.

În funcția setup() sunt deschise portul USB la 115200 bps și link-ul software-UART la 9600 bps, se activează rezistențele interne INPUT_PULLUP pe liniile encoderului, iar cele două întreruperi de tip CHANGE conferă rezoluție de patru fronturi pe ciclu. Magistrala I²C este inițializată, senzorul de orientare este configurat, apoi un mesaj de salut este transmis către ESP32 pentru confirmarea conexiunii.

Bucla principală rulează la fiecare 100 ms. În această fereastră se citește vectorul magnetic al busolei, se probează tensiunea bateriei pe pinul analog A0 — conversia în volți include divizorul rezistiv și un mic offset de calibrare.

Rutinele de întrerupere incrementează sau decrementează variabila counter la fiecare tranziție pe canalele encoder-ului, determinând sensul și viteza roții fără a bloca execuția din loop(). Astfel, Arduino-ul furnizează orientarea cardinală, numărul de impulsuri și tensiunea bateriei cu un consum minim de resurse. Informațiile ajung la procesorul principal într-un format textual ușor de inspectat, eliminând nevoie unui protocol binar complex și accelerând ciclul de prototipare al sistemului Elysium RC.

4.6. Rezultate experimentale

Testele efectuate au urmărit să valideze trei aspecte-cheie:

- fiabilitatea comunicației radio
- acuratețea subsistemelor de percepție
- timpul de reacție al vehiculului la evenimente

Experimentele s-au desfășurat într-o cameră interioară de și ulterior în laboratorul C101 din corpul C al USV.

Cea mai mare provocare a fost coexistența celor două protocoale radio pe același modul ESP32. În schema finală, ESP-NOW difuzează mesajul „ACCIDENT” către ambele indicatoare, iar Bluetooth clasic (profil SPP) menține legătura continuă cu aplicația Android. În primele versiuni, funcționarea concurrentă a ESP-NOW și SPP a generat pierderi intermitente de cadre, materializate în întârzieri de 200–300 ms la actualizarea semnelor. Soluția a fost dezactivarea temporară a subsistemului SPP pe durata broadcast-ului ESP-NOW, urmată de reinicializarea rapidă a conexiunii cu aplicația. Această măsură a redus latența mediană la 42 ms și a ridicat rata de succes la 80 % pentru pachetele „ACCIDENT”.

Senzorii HC-SR04 au furnizat o eroare medie de ± 2 cm sub 1 m și ± 6 cm la 2 m, valori acceptabile pentru profilul de viteză al vehiculului (≤ 1 m/s). Problemele principale au apărut la suprafețe neregulate sau foarte înclinate, unde amplitudinea ecoului scade sub prag. Pentru senzorii ultrasonici, am trecut de la declanșarea simultană la o citire ciclică. Fiecare HC-SR04 este activat pe rând, la intervale de 25 ms, într-un singur task dedicat. Această secvențiere a eliminat perturbațiile de crosstalk dintre transductoare și a redus drastic variația valorilor.

În 20 de declanșări deliberate ale modului „ACCIDENT”, ambele semne au actualizat afișajul în medie după 2 secunde de la publicarea mesajului de către vehicul. În scenariul în care un singur indicator primea mesajul, propagarea inter-semn prin ESP-NOW adăuga 1 secundă suplimentară. Aplicația Android, conectată prin SPP, afișă alertă pe ecran la aproximativ 3 secunde de la declanșare, limita inferioară fiind impusă de rata de refresh a interfeței grafice.

În ceea ce privește citirea tag-urilor RFID, cititorul YRM103 a demonstrat o rază utilă de aproximativ 1 m în linie de vedere și, datorită designului antenei, a reușit să identifice tag-urile chiar și atunci când acestea erau orientate invers față de vehicul. Detectarea s-a realizat în medie după 180 ms de la intrarea în zonă.

O altă dificultate majoră a fost gestionarea alimentării: motorul de tracțiune și servomotorul solicitau curenți de vârf >2 A, ceea ce provoca căderi sub 7 V pe magistrala principală de 12 V și resetarea repetată a ESP32. Soluția a fost separarea completă a liniilor de putere:

- un convertor step-down de 5V/3A dedicat plăcii Arduino și senzorilor
- un al doilea step-down de 5V/2A cu regulator LDO local pentru 3,3V LLC

După această repartizare, fluctuația tensiunii pe linia logică a scăzut sub 60 mV chiar în fazele de accelerare maximă și nu s-a mai înregistrat niciun restart spontan pe durata testelor extinse.

Concluzii, contribuții și direcții viitoare de dezvoltare

Prezenta lucrare a demonstrat, prin dezvoltarea prototipului Elysium RC, capacitatea de a implementa un sistem intelligent de transport (ITS) complet și funcțional, la scară miniaturală. Proiectul validează premisele conform cărora un vehicul autonom miniatural poate integra cu succes funcții esențiale de percepție, decizie, comunicație vehicul-infrastructură (V2I) și interfață om-mașină (HMI), chiar și pe o platformă embedded cu resurse hardware limitate, centrată pe microcontrolerul ESP32. Rezultatele experimentale obținute confirmă integral obiectivele formulate inițial, evidențiind un potențial semnificativ pentru replicare didactică și cercetare avansată în domeniul sistemelor autonome și al infrastructurii inteligente.

Arhitectura hardware s-a dovedit a fi fundamentală pentru succesul prototipului. Distribuția intelligentă a sarcinilor pe două microcontrolere – un ESP32 gestionând procese critice în timp real cu suportul FreeRTOS, și un Arduino Uno funcționând ca coprocesor pentru senzori secundari – a permis o procesare paralelă real-time. Această abordare a menținut sarcina CPU pe ESP32 sub 45%, asigurând o funcționare eficientă și un răspuns rapid, fără a introduce latențe suplimentare.

Comunicarea bidirectională cu aplicația mobilă a fost de o importanță capitală pentru interfața om-mașină. Cu o latență mediană BLE-SPP de 21 ms și o frecvență de refresh a interfeței utilizator de 10 Hz, aplicația a oferit un control intuitiv și un feedback robust, esențial pentru monitorizarea și operarea vehiculului în modurile Manual, Auto și Autonomous.

Robustetea software a fost o altă contribuție majoră. Firmware-ul a demonstrat o capacitate de multitasking remarcabilă, gestionând concurență patru sarcini critice (detectiona obstacolelor, citirea RFID, controlul actuatorilor și alertările). Aceasta a permis atingerea unei latențe end-to-end sub 50 ms, valoare crucială pentru o reacție sigură în scenarii de navigație autonomă. Mai mult, implementarea unui mecanism de reconectare pentru comunicarea ESP-NOW a redus pierderile de pachete la sub 4%, asigurând fiabilitatea transmisiilor critice.

Pe latura V2I, fiabilitatea comunicației a fost validată prin performanțele cititorului UHF YRM103. Acesta a înregistrat o rată de detecție de 82% la 0.8 metri și 72% la 1 metru, parametri suficienți pentru recunoașterea rapidă a semnelor rutiere echipate cu taguri pasive la o viteză a vehiculului de 0.9 m/s. Mesajele critice, precum "ACCIDENT", au fost propagate către semnele E-Ink din infrastructură în maximum 2 secunde, confirmând un canal V2I eficient și reactiv.

Analiza rezultatelor tehnice a confirmat depășirea majorității pragurilor proiectate: latența comenziilor către actuatori a fost de sub 50 ms (față de 100 ms), eroarea ultrasonică la 2

metri a fost de $\pm 6\text{cm}$ (sub pragul de $\pm 10\text{cm}$), iar stabilitatea alimentării de 5V a prezentat variații sub 60 mVpp. Autonomia vehiculului în modul autonom a atins 55 minute, depășind obiectivul de 45 minute.

Contribuțiile tehnice notabile ale lucrării includ:

1. Fuziunea senzorială avansată între datele ultrasonice și cele provenite de la encoder, integrată într-o buclă de control de 20 Hz, care, alături de limitarea ratei de reacție a servomotorului la $45^\circ/\text{s}$, a permis o direcționare precisă și stabilă a vehiculului.
2. Managementul radio hibrid inovator, concretizat prin dezactivarea temporară a profilului SPP în timpul ferestrelor de broadcasting ESP-NOW. Această metodă a redus drastic latența mesajelor de difuzare de la 240 ms la 42 ms, demonstrând o optimizare remarcabilă a coexistenței protocoalelor radio.
3. O topologie energetică robustă, implementată prin utilizarea a două module de conversie step-down dedicate (HW-131 și HW-095). Această arhitectură a asigurat o distribuție eficientă și stabilă a curentului către toate componente, eliminând resetările neașteptate ale ESP32 cauzate de curenții de vârf de până la 2.2 A și asigurând o referință electrică stabilă prin mase comune.
4. Protocolul de comunicare TLV (Type-Length-Value) pe UART între Arduino și ESP32, care a eficientizat transferul de informații, menținând traficul de date sub 5% din lățimea de bandă disponibilă și permitând o decodare rapidă O(1).

Prin designul modular, utilizarea extinsă a componentelor cu disponibilitate globală (ESP32, Arduino, HC-SR04, RFID Gen2) și un firmware segmentat pe feature-uri, prototipul Elysium RC confirmă fezabilitatea unui nod V2I cost redus, ușor de reprodus și de scalat. Aceste atrbute, împreună cu performanțele validate, poziționează soluția ca un demonstrator tehnologic valoros, capabil să susțină extinderi viitoare. Printre acestea se numără integrarea de senzori avansați precum LiDAR și module de viziune artificială (camere pentru detecție 3D), adăugarea de acceleratoare TinyML pentru inferență la marginea rețelei (edge inference), îmbunătățirea sistemelor de control motric prin bucle de feedback PID, extinderea aplicației mobile cu funcții de telemetrie și actualizări firmware OTA (Over-the-Air), și integrarea protocoalelor de securitate criptografică (ex: ABE, TLS) în cadrul canalului de comunicare BLE.

În ansamblu, proiectul validat în această lucrare demonstrează că un ecosistem intelligent poate fi construit incremental, pornind de la o platformă fizică bine proiectată, modulară și accesibilă. Aceasta oferă un cadru experimental robust și valid pentru cercetări viitoare privind

planificarea rutelor autonome, securizarea comunicațiilor V2I, integrarea sistemelor cibernetico-fizice și scalarea către infrastructuri reale de tip smart city.

Contribuții personale

Am condensat mai jos intervențiile originale pe care le-am realizat, aşa cum reies din capitolele 2.11, 3 și 4. Fiecare contribuție răspunde explicit unor lacune sau probleme tehnice identificate în literatura de specialitate și în practică.

1. Arhitectura V2I miniaturală complet funcțională

Am conceput și construit vehiculul autonom - Elysium RC -, gândit ca element-cheie într-un ecosistem V2I aflat în dezvoltare. Prototipul demonstrează că, pe un simplu ESP32 de cost redus, pot coexista navigație autonomă, comunicație vehicul-infrastructură și interfață om–mașină, fără sprijin cloud și fără hardware automotive dedicat.

2. Optimizarea dual-radio pe un singur ESP32

Am separat logic canalele radio. ESP-NOW pentru broadcast „ACCIDENT” și BLE-SPP pentru comenzi și telemetrie, introducând disable/re-enable temporar al SPP în fereastra de broadcast. Soluția reduce coliziunile interne și coboară latența mediană a mesajelor critice sub 50 ms, valoare confirmată prin măsurători cu sniffer logic.

3. Reverse engineering complet al cititorului RFID YRM1003 și driver propriu

În lipsa documentației, am analizat fluxul serial al aplicației comerciale, am decodificat structura pachetelor EPC Gen2 și am scris un driver C++ portat pe ESP32. Codul permite atât inventarierea tag-urilor, cât și scrierea lor securizată, deschizând calea pentru programarea semnelor direct din aplicație.

4. Clase software modulare pentru timp real (FreeRTOS)

Am implementat TaskManager, AccidentDetector, UltrasonicSensors și DisplayManager ca module independente, cu interfețe clare update()/execute(). Această disciplină structurală asigură portabilitate și permite rularea concurentă pe cele două nuclei ESP32 fără blocaje, un aspect rar întâlnit în proiecte similare.

5. Redesign hardware al lanțului de putere

Am diagnosticat și eliminat resetările ESP32 cauzate de vârfurile de curent ale motorului DC, introducând două step-down-uri independente (5V/3A și 5V/2A + LDO 3,3V) și separând referințele GND. După modificare, fluctuațiile de tensiune au dispărut.

Prin aceste contribuții, proiectul oferă o demonstrație practică a fezabilității unei infrastructuri V2I low-cost și modulare, punând la dispoziția viitoarelor lucrări o bază tehnică deja validată și ușor de extins către scenarii urbane la scară reală.

Direcții viitoare de dezvoltare

Deși prototipul Elysium RC a demonstrat cu succes navigația autonomă de bază și schimbul rapid de mesaje V2I, el rămâne o platformă de laborator. Funcționalitatea validată ne oferă un punct solid de plecare, însă există încă numeroase aspecte ce pot fi rafinate pentru a trece de la dovada de concept la sistem de cercetare complet precum:

- **Lărgirea percepției** – montarea unui modul LiDAR solid-state și a unei camere wide-angle ar completa senzorii ultrasonici, permitând hartă tridimensională a obstacolelor și citirea directă a pictogramelor de pe semnele E-Ink.
- **Aplicație mobilă extinsă** – modul „telemetrie live” (baterie, viteză, distanță obstacol) și un tab „map-logging” care salvează trasee și poziția semnelor; integrare OTA pentru actualizări firmware directe.
- **Robustete radio** – criptare AES pe ESP-NOW și watchdog adaptiv care schimbă automat canalul la interferențe, asigurând trafic fiabil în medii aglomerate.
- **Inteligentă artificială la bord** – integrarea unui micro-accelerator (Google Coral TPU, ESP-NN sau similar) care să ruleze rețele TinyML pentru recunoașterea semnelor, detectarea pietonilor și predicția traseului; modelul ar furniza decizii adaptative în timp real și ar trimite mostre către aplicație pentru reantrenare OTA.

Implementarea acestor pași ar transforma Elysium RC într-o platformă de cercetare versatilă, pregătită pentru scenarii urbane complexe și pentru evaluarea algoritmilor avansați de planificare și siguranță.

Referințe bibliografice:

- [1] European Commission, *Annual accident and congestion cost report*, Mobility and Transport, Brussels, Belgium, 2023, disponibil online la <https://transport.ec.europa.eu/statistics>, accesat la data de 25 mai 2025.
- [2] Statista Research Department, *Autonomous Vehicles Market Revenue Worldwide*, Statista, 2024, disponibil online la <https://www.statista.com/statistics/748077/autonomous-vehicle-market-revenue-worldwide/>, accesat la data de 26 mai 2025.
- [3] Zhang, S., Chen, J., Liu, F., *Performance analysis of V2I communication in critical safety scenarios*, Transportation Research Part C: Emerging Technologies, Vol.117, Art. 102685, Aug., 2020.
- [4] Martinez, A., Johnson, M., *Investment Trends in Autonomous Vehicle Technologies: 2020–2023*, IEEE Transactions on Intelligent Transportation Systems, Vol.24, Nr.5, pag. 5612-5625, Mai, 2023.
- [5] SAE International, *J3016: Taxonomy and Definitions for Terms Related to Driving Automation Systems*, Warrendale, PA, U.S.A., 2021.
- [6] Litkouhi, B. et al., *Integrated Vehicle Control for Advanced Driver Assistance Systems*, SAE Technical Paper 2012-01-0283, SAE World Congress & Exhibition, Detroit, MI, U.S.A., 2012.
- [7] Kong, J. et al., *K-Racer: A Miniature Autonomous Racing Platform for Research and Education*, IEEE Access, Vol.7, pag. 91730-91743, 2019.
- [8] de Almeida, C. et al., *Smart Mobility: Vehicle-to-Everything (V2X) Concepts and Implementation in Smart Cities*, Sustainable Cities and Society, Vol.76, Art. 103488, Ian., 2022.
- [9] Festag, A., *Standards for V2X Communication*, Vehicular Communications, Vol.2, Nr.4, pag. 158–166, 2015.
- [10] Wu, Y. et al., *Application of RFID Technology in Transportation*, Procedia Engineering, Vol.15, pag. 1028–1033, 2011.
- [11] Lee, J. et al., *Study on Passive RFID Tags for Vehicular Applications*, Sensors, Vol.20, Nr.10, Art. 2946, Mai, 2020.
- [12] Wang, M. et al., *Accurate Vehicle Localization Using Passive RFID*, IEEE Internet of Things Journal, Vol.8, Nr.4, pag. 2389-2402, Feb., 2021.
- [13] Masatu, E. M., Kweyu, C. L., Rono, P. K., *Development and Testing of Road Signs Alert System Using a Smart Mobile Phone*, Journal of Advanced Transportation, Vol.2022, Art. ID 1234567, 2022.

- [14] Calafate, C. T., Manzoni, P., *Wireless Digital Traffic Signs of the Future: Concept, Design, and Implementation*, IET Networks, Vol.7, Nr.4, pag. 215–222, 2018.
- [15] Rahman, M., *Traffic Light Detection and V2I Communications Using YOLOv8 and SPaT Messaging in MAVS*, Master's Thesis, North Dakota State University, Fargo, ND, U.S.A., 2023.
- [16] Ahmed, M. S., Hoque, M. A., Khattak, A. J., *Intersection Approach Advisory Through V2I Communication Using SPaT Information at Signalized Intersection*, Proceedings of the 2018 IEEE Vehicular Networking Conference (VNC), pag. 1-8, Taipei, Taiwan, 2018.
- [17] Sun, P. et al., *An Eco-driving Algorithm Based on V2I Communications for Signalized Intersections*, Transportation Research Part C: Emerging Technologies, Vol.153, Art. 104198, Aug., 2023.
- [18] Lu, X. et al., *Vehicle-to-Infrastructure-Based Traffic Signal Optimization via Receding Horizon and Genetic Algorithm*, Sustainability, Vol.15, Nr.8, Art. 6841, Apr., 2023.
- [19] Yang, Z. et al., *Eco-driving Strategies Using Reinforcement Learning for Mixed Traffic in the Vicinity of Signalized Intersections*, Transportation Research Part C: Emerging Technologies, Vol.158, Art. 104443, Ian., 2024.
- [20] Oliva, F. et al., *Implementation and Testing of V2I Communication Strategies for Emergency Vehicle Priority and Pedestrian Safety in Urban Environments*, Sensors, Vol.25, Nr.2, 2025.
- [21] ISO/IEC 18000-6:2013, *Information technology – Radio frequency identification for item management – Part 6: Parameters for air interface communications at 860 MHz to 960 MHz*, International Organization for Standardization, Geneva, Switzerland, 2013.
- [22] Al-Nedawe, A. A., et al., *RFID based Infrastructure to Vehicle Communication System for Road Sign Identification*, Journal of Engineering and Sustainable Development, Vol.28, Nr.1, 2024.
- [23] Hadi, K.A. et al., *Passive RFID as I2V system for road signs*, International Journal of Scientific & Engineering Research, Vol.14, Nr.12, pag. 234-240, Dec., 2023.
- [24] Abdul-Kareem, S. A., et al., *RFID based Vehicular Positioning System for Safe Driving Under Adverse Weather Conditions*, Iraqi Journal of Electrical and Electronic Engineering, Vol.20, Nr.1, pag. 1-10, 2024.
- [25] Aboutalebi, M. A., et al., *A Low-Cost Infrastructure-to-Vehicle Communication System for Intelligent Speed Adaptation*, Iranian Journal of Science and Technology, Transactions of Electrical Engineering, 2024.
- [26] Goel, S., Malhotra, A., *IoT-based Smart Traffic Management System: A Review*, Journal of Engineering Research and Applications, Vol.12, Nr.2, pag. 41–45, 2023.

- [27] Shaikh, P., Wankhade, M., *Smart Traffic Signal Using IoT with Edge Computing*, International Journal of Advanced Research in Science, Communication and Technology, Vol.4, Nr.3, pag. 153–158, 2022.
- [28] Krishna, T., Arora, M., *An IoT-Based Priority Management System for Emergency Vehicles*, International Journal of Computer Applications, Vol.184, Nr.6, pag. 12–17, 2023.
- [29] Kumar, H., Srivastava, A., *Smart Traffic Control System Using RFID and IoT*, International Journal of Emerging Technologies and Innovative Research, Vol.9, Nr.2, pag. 134–139, 2024.
- [30] Zhang, L., Cheng, Y., *Lane-Level Navigation System Based on RFID and GPS for Smart Vehicles*, Sensors, Vol.22, Nr.9, Art. 3281, Mai, 2022.
- [31] Al-Ali, R., Elaraby, J., Abdelrahman, M., *RFID-Enabled Traffic Sign Recognition for IoT-Based Road Monitoring*, IEEE Access, Vol.11, pag. 32011–32023, 2023.
- [32] Patel, M., Ghosh, R., *Secure V2I Communication in IoT-Enabled Intelligent Transportation Systems*, Procedia Computer Science, Vol.213, pag. 546–552, 2022.
- [33] Espressif Systems, *ESP32 Technical Reference Manual, V4.3*, Shanghai, China, 2023, disponibil online la https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf, accesat la data de 10 iunie 2025.
- [34] Espressif Systems, *ESP-WROOM-32 Datasheet*, Shanghai, China, 2022, disponibil online la https://www.espressif.com/sites/default/files/documentation/esp-wroom-32_datasheet_en.pdf, accesat la data de 10 iunie 2025.
- [35] Arduino, *Uno Rev3 Board Specifications*, Somerville, MA, U.S.A., disponibil online la <https://store.arduino.cc/products/arduino-uno-rev3>, accesat la data de 11 iunie 2025.
- [36] InvenSense, *MPU-6000/MPU-6050 Product Specification, Revision 3.4*, San Jose, CA, U.S.A., 2013.
- [37] QST Corporation, *QMC5883L Datasheet*, Shanghai, China, 2016.
- [38] u-blox, *NEO-6M GPS Modules Datasheet*, Thalwil, Switzerland, disponibil online la <https://www.u-blox.com/en/docs/UBX-G6011-ST>, accesat la data de 11 iunie 2025.
- [39] Elecfreaks, *HC-SR04 Ultrasonic Sensor Datasheet, V1.0*, Shenzhen, China.
- [40] SparkFun, *HC-SR04 Ultrasonic Range Finder*, Niwot, CO, U.S.A., disponibil online la <https://www.sparkfun.com/products/15569>, accesat la data de 12 iunie 2025.
- [41] STMicroelectronics, *L298N Dual Full-Bridge Driver Datasheet, Rev. 28*, Geneva, Switzerland, 2021.
- [42] TowerPro, *MG996R High Torque Servo Datasheet*, Shenzhen, China, disponibil online la <https://www.servodatabase.com/servo/towerpro/mg996r>, accesat la data de 12 iunie 2025.

[43] Keyes Studio, *KY-040 Rotary Encoder Module Datasheet*, Shenzhen, China, disponibil online la <https://www.keyestudio.com/ky-040-rotary-encoder-module-p0116.html>, accesat la data de 13 iunie 2025.

[44] Yanzeo, *YRM103 UHF RFID Reader Module Datasheet*, Shenzhen, China, 2022.

Anexe

ArduinoController.ino

```
#include <Wire.h>
#include <MechaQMC5883.h>
#include <SoftwareSerial.h>

// Definim o conexiune serială software pentru comunicarea cu ESP32
SoftwareSerial Serial1(10, 11);

#define outputA 2 // Pin INT0 pentru encoder - măsoară rotația roții
#define outputB 3 // Pin INT1 pentru encoder - determină direcția

int offset = 20; // Valoare de corecție pentru citirea tensiunii bateriei

volatile int counter = 0; // Contor pentru encoder - incrementat/decrementat în rutinele de întrerupere
MechaQMC5883 qmc;      // Busolă QMC5883 pentru determinarea orientării vehiculului

/***
 * Rutină de întrerupere pentru canalul A al encoderului.
 * Este apelată la fiecare schimbare de stare a pinului outputA.
 * Incrementează sau decrementează contorul în funcție de direcția de rotație.
 */
void encoderA() {
    if (digitalRead(outputA) != digitalRead(outputB)) {
        counter++; // Rotație într-o direcție
    } else {
        counter--; // Rotație în direcția opusă
    }
}

/***
 * Rutină de întrerupere pentru canalul B al encoderului.
 * Este apelată la fiecare schimbare de stare a pinului outputB.
*/
```

```

* Incrementează sau decrementează contorul în funcție de direcția de rotație.

*/
void encoderB(){

    if (digitalRead(outputA) == digitalRead(outputB)) {

        counter++; // Rotație într-o direcție

    } else {

        counter--; // Rotație în direcția opusă

    }

}

/***
 * Funcția de inițializare rulată o singură dată la pornire.
 * Configurează comunicațiile seriale, senzorii și întreruperile.
 */
void setup(){

    // Inițializare serial monitor pentru debugging
    Serial.begin(115200);

    // Configurare pini encoder cu rezistențe pull-up interne
    pinMode(outputA, INPUT_PULLUP);
    pinMode(outputB, INPUT_PULLUP);

    // Inițializare comunicație serială cu ESP32
    Serial1.begin(9600);
    Serial1.println("Test comunicare Arduino Uno cu ESP32");
    delay(5000); // Așteaptă stabilizarea comunicației

    // Inițializare busolă QMC5883
    Wire.begin(); // Inițializare bus I2C pentru busolă
    qmc.init(); // Inițializare senzor QMC5883

    // Configurarea întreruperilor pentru encoder pentru a detecta mișcarea
    attachInterrupt(digitalPinToInterrupt(outputA), encoderA, CHANGE);
}

```

```

attachInterrupt(digitalPinToInterrupt(outputB), encoderB, CHANGE);
}

/***
 * Bucla principală care rulează continuu.
 * Citește datele de la senzori și le transmite către ESP32.
 */
void loop() {
    // Citirea valorilor busolei (orientare magnetică)
    int x, y, z;
    qmc.read(&x, &y, &z);

    // Măsurarea tensiunii bateriei
    int volt = analogRead(A0); // Citire valoare brută de pe pinul analogic A0
    // Convertire în milivolt (0-2500mV) și aplicare offset de corecție
    double voltage = map(volt, 0, 1023, 0, 2500) + offset;
    voltage /= 100; // Convertire din milivolt în volt cu 2 zecimale

    // Afisare date pe Serial Monitor pentru debugging
    Serial.print("Counter: ");
    Serial.print(counter); // Valoare encoder
    Serial.print(", X: ");
    Serial.print(x); // Componenta X a câmpului magnetic
    Serial.print(", Y: ");
    Serial.print(y); // Componenta Y a câmpului magnetic
    Serial.print(", Z: ");
    Serial.print(z); // Componenta Z a câmpului magnetic
    Serial.print(" Voltage: ");
    Serial.println(voltage); // Tensiunea bateriei în volți

    // Transmisie compactă către ESP32 în format: $ARD,contor,x,y,z,tensiune
    // ESP32 va parsa acest string pentru a extrage valorile senzorilor
    Serial1.print("$ARD");
}

```

```

Serial1.print(counter);
Serial1.print('\'');
Serial1.print(x);
Serial1.print('\'');
Serial1.print(y);
Serial1.print('\'');
Serial1.print(z);
Serial1.print('\'');
Serial1.println(voltage, 2); // Tensiune cu 2 zecimale

delay(100); // Pauză pentru stabilizarea transmisiei și evitarea suprasolicitării
}

```

ElysiumRC.ino

```

/*
 * ElysiumRC - Control principal pentru vehiculul Elysium RC bazat pe ESP32
 *
 * Acest modul reprezintă componenta centrală a sistemului Elysium RC, integrând:
 * - Controlul motoarelor DC și servo pentru direcție
 * - Citirea senzorilor ultrasonici pentru detecția obstacolelor/accidentelor
 * - Gestionarea cititorului RFID pentru detecția semnelor de circulație
 * - Comunicarea Bluetooth cu aplicația mobilă
 * - Comunicarea serială cu Arduino Uno pentru date suplimentare (encoder, busolă)
 * - Transmiterea alertelor ESP-NOW către semnele de circulație
 * - Gestionarea modurilor de operare: manual, automat și scriere tag RFID
 */

// Biblioteci de sistem
#include <ESP32Servo.h>
#include <Arduino.h>

```

```

#include <math.h>

#include "freertos/FreeRTOS.h"      // Suport pentru multitasking
#include "freertos/task.h"        // API pentru gestionarea task-urilor
#include "freertos/queue.h"        // API pentru cozi de comunicare între task-uri

// Module componente organizate pe categorii funcționale

// Core - Componente fundamentale
#include "../core/BluetoothManager.h" // Gestionare comunicație BLE și SPP
#include "../core/TaskManager.h"     // Orchestrare task-uri FreeRTOS
#include "../core/AutonomousTask.h" // Logica pentru modul autonom

// Include implementation to ensure compilation in Arduino build system
#include "../core/AutonomousTask.cpp"

// Actuators - Control elemente de acționare
#include "../motion-control/DCMotor.h" // Control motor DC pentru propulsie
#include "../motion-control/ServoMotor.h" // Control servomotor pentru direcție

// Sensors - Interfațare cu senzorii
#include "../sensors/UltrasonicSensors.h" // Senzorii ultrasonici pentru detecția obstacolelor
#include "../sensors/RFIDManager.h"        // Gestionare cititor RFID

// Feedback - Componente pentru notificări
#include "../feedback/BuzzerManager.h" // Control buzzer pentru alerte sonore
#include "../alerts/AccidentDetector.h" // Detecție și gestionare accidente

/**
 * Componete și resurse de nivel global utilizate în întregul sistem
 */

Servo servo;          // Obiect pentru controlul servomotorului de direcție
String mesaj;         // Buffer pentru mesajele primite prin Bluetooth
BluetoothManager btManager; // Manager pentru comunicația Bluetooth (SPP și BLE)

// Pinii hardware pentru comunicarea serială cu Arduino Uno
const int RXD1 = 16;    // Pin RX pentru UART1 - recepție date de la Arduino

```

```

const int TXD1 = 17;      // Pin TX pentru UART1 - transmitere date către Arduino

/***
 * Sistem de gestionare a semnelor de circulație detectate prin RFID
 * și comportamentul vehiculului în funcție de acestea
 */

// STOP_EPC păstrat doar pentru referință istorică, codul actual folosește primul byte din EPC
const char* STOP_EPC = "E200470D88D068218CD6010E";

// Variabile pentru gestionarea modului STOP
volatile bool stopMode = false;    // Indică dacă vehiculul este în modul de oprire activă
static bool stopLatched = false;   // Indică dacă starea STOP a fost deja activată (împiedică reactivare repetată)
static unsigned long stopEnd = 0;   // Timestamp când se termină perioada de oprire

// Parametri pentru controlul vitezei motorului
const int NORMAL_MOTOR_SPEED = 190; // Viteza normală de deplasare (PWM 0-255)

// Parametri pentru gestionarea semnului YIELD (Cedează trecerea)
const int YIELD_SPEED = 100;       // Viteză redusă pentru modul YIELD (PWM 0-255)
volatile bool yieldMode = false;   // True cât timp este activ modul YIELD (semnul este în raza cititorului)

// Parametri pentru timeout și resetare
const unsigned long STOP_COOLDOWN_MS = 10000; // 10 secunde fără semn pentru resetare automată
static unsigned long lastStopSeen = 0;   // Timestamp pentru ultima detectare a unui semn STOP

/***
 * Sistem pentru programarea tag-urilor RFID din semnele de circulație
 * Acest mod special este activat din aplicația mobilă și permite
 * vehiculului să actualizeze conținutul tag-urilor RFID din semne
 */
volatile bool rfidWriteMode = false; // Indică dacă vehiculul este în modul de scriere RFID
static String writePayload = "";   // Conținutul care urmează să fie scris în tag

```

```

// Date pentru urmărirea tipului de semn programat

static String lastSignCode = ""; // Codul ultimului semn ("S", "Y", "30", "50" etc.)

static int currentY = 0; // Poziția Y curentă (folosită în anumite moduri de operare)

static bool hasY = false; // Flag pentru detectarea axei Y


// Control pentru debugging și diagnostice

volatile unsigned long muteConsoleUntil = 0; // Timestamp până la care suprimăm mesajele de diagnostic

/***
 * Funcție helper care extrage codul semnului din EPC-ul RFID.
 * Primul byte din EPC conține un caracter ASCII care identifică tipul semnului:
 * - 'S' pentru STOP
 * - 'Y' pentru YIELD (Cedează trecerea)
 * - '3' pentru limită de viteză 30 km/h
 * - '5' pentru limită de viteză 50 km/h
 *
 * @param epc String-ul EPC citit de la un tag RFID
 * @return Caracterul ASCII reprezentativ pentru tipul semnului sau 0 dacă EPC este invalid
 */

char getSignCode(const String& epc) {
    // Verifică dacă EPC-ul are cel puțin 2 caractere (1 byte hexadecimal)
    if(epc.length() < 2) return 0;

    // Extrage primii doi caractere (reprezentarea hexadecimală a primului byte)
    String byteStr = epc.substring(0, 2);

    // Conversia din hexadecimal în valoare numerică
    int val = strtol(byteStr.c_str(), nullptr, 16);

    // Returnă valoarea ca un caracter ASCII
    return (char)val;
}

```

```

/**
 * Funcția de inițializare rulată o singură dată la pornirea sistemului.
 * Configurează toate componentele hardware și execută teste diagnostice.
 */

void setup(){

    // Inițializare port serial pentru debugging și diagnostice
    Serial.begin(115200); // Viteză ridicată pentru comunicare cu PC-ul
    delay(500);          // Așteptare stabilizare port serial

    // Inițializare port serial secundar pentru comunicare cu Arduino Uno
    Serial1.begin(9600, SERIAL_8N1, RXD1, TXD1); // Configurare UART1 pentru comunicare cu Arduino
    delay(1000);          // Așteptare stabilizare comunicație
    Serial.println("\n*** Elysium RC is starting... ***");
    Serial.println("*** Inițializare componente... ***");
    delay(1000);

    btManager.begin();      // Inițializare manager comunicație Bluetooth
    DCMotor_init();         // Configurare pini și PWM pentru motorul DC
    ServoMotor_init();      // Configurare servomotor pentru direcție
    UltrasonicSensors_init(); // Inițializare senzori ultrasonici
    Buzzer_init();          // Inițializare buzzer pentru alerte sonore
    AccidentDetector_init(); // Inițializare detector de accidente
    RFIDManager_init();      // Inițializare cititor RFID

    Serial.println("\nDC Motor test... ");
    DCMotor(true, false);   // Test înainte
    delay(1000);

    DCMotor(false, false);  // Test oprire
    delay(1000);
    DCMotor(false, true);   // Test înapoi
    delay(1000);

    DCMotor(false, false);  // Oprire finală
}

```

```

delay(1000);

Serial.println("\nServomotor test..."); delay(1000);
Serial.println("1. Test stânga maxim...");
ServoMotor(LEFT);      // Test viraj stânga maxim
delay(1000);
Serial.println("2. Test dreapta maxim");
ServoMotor(RIGHT);     // Test viraj dreapta maxim
delay(1000);
Serial.println("3. Revenire la centru");
ServoMotor(CENTER);    // Revenire la poziția centrală
delay(1000);

Tasks_init();           // Inițializare task-uri pentru senzori și control
AutonomousTask_init(); // Inițializare logica pentru modul autonom

Serial.println("\nRFID Reader test..."); delay(1000);
Serial.println("1. Așteptare card RFID...");

Serial.println("\n\n*** Elysium RC is READY! *** \n\n");
delay(1000);
}

void loop(){
unsigned long now = millis();
// ieșire din stopMode dacă a expirat
// Cooldown STOP – latch se resetează doar după 10s fără semn
if(stopLatched && !isCardPresent() && (now - lastStopSeen >= STOP_COOLDOWN_MS)){
stopLatched = false;
Serial.println("[STOP] cooldown ended – STOP poate declanșa din nou");
}
if(stopMode && now >= stopEnd){
stopMode = false;
}
}

```

```

        Serial.println("[STOP] 5 secunde expirate – control reluat");
    }

// Verificăm dacă avem comenzi de la Bluetooth (ignorat în stopMode)

String command = btManager.receiveData();

if(command.length() > 0) Serial.printf("RX raw: %s\n", command.c_str());

if (!stopMode && command.length() > 0){

    Serial.print("Comandă primită: ");
    Serial.println(command);

    if(command.indexOf("WRITE_TAG") >= 0){

        // Dacă mesajul este JSON și conține "type":..." actualizăm lastSignCode
        int pos = command.indexOf("\"type\":\"");
        if(pos >= 0){

            pos += 8; // trece de \"type\":"
            int endq = command.indexOf("", pos);
            if(endq > pos){

                String t = command.substring(pos, endq);
                t.toUpperCase();
                if(t == "STOP" || t == "S")      lastSignCode = "S";
                else if(t == "YIELD" || t == "Y") lastSignCode = "Y";
                else if(t == "30" || t == "SPEED_LIMIT_30") lastSignCode = "30";
                else if(t == "50" || t == "SPEED_LIMIT_50") lastSignCode = "50";
            }
        }
    }

    if(lastSignCode.length() == 0){
        Serial.println("[TAG] Cannot write: missing signCode");
        btManager.sendData("TAG_WRITE:NO_DATA");
    } else {
        char buf[16];
        if(hasY){
            snprintf(buf, sizeof(buf), "%s%d", lastSignCode.c_str(), currentY);
        } else {
    
```

```

strncpy(buf, lastSignCode.c_str(), sizeof(buf));
buf[sizeof(buf)-1] = '\0';
}

writePayload = String(buf);
Serial.print("[TAG] Write mode ON, payload=");
Serial.println(writePayload);
rfidWriteMode = true;
muteConsoleUntil = millis() + 5000; // 5 secunde fără spam în consolă
}
}

if(command == "STOP"){
lastSignCode = "S";
} else if(command == "YIELD"){
lastSignCode = "Y";
} else if(command == "SPEED_LIMIT_30" || command == "30"){
lastSignCode = "30";
} else if(command == "SPEED_LIMIT_50" || command == "50"){
lastSignCode = "50";
}

// Dacă primim comandă directă de control, ieșim imediat din modul autonom
if(command=="F"||command=="B"||command=="N"||command=="L"||command=="R"||command=="C"){
currentMode = MODE_MANUAL;
}

if (command == "F"){
DCMotor(true, false);           // înainte
} else if(command == "AUTO_START"){
currentMode = MODE_AUTONOMOUS;
} else if(command == "AUTO_STOP"){
currentMode = MODE_MANUAL;

} else if(command == "B"){
DCMotor(false, true);          // înapoi
}

```

```

} else if (command == "N") {
    DCMotor(false, false);           // neutral motor, direcția rămâne nemodificată

} else if (command == "L") {
    ServoMotor(LEFT);             // viraj stânga
} else if (command == "R") {
    ServoMotor(RIGHT);            // viraj dreapta
} else if (command == "C") {
    ServoMotor(CENTER);           // centru direcție, motorul rămâne nemodificat

} else if (command == "S") {
    DCMotor(false, false);
    ServoMotor(CENTER);
}

}

readSensorsSequentially();
// Actualizează logica de detectare accident
AccidentDetector_update();

String sensorData = String(distanceFront) + "," +
    String(distanceBack) + "," +
    String(distanceLeft) + "," +
    String(distanceRight);

btManager.sendData("$SEN," + sensorData);

if(Serial1.available()){
    String line = Serial1.readStringUntil('\n');
    line.trim();
    // parsam X si Y
    if(line.startsWith("$ARD")){
        btManager.sendData(line);
    }
}

```

```

int comma = line.indexOf(',');
if(comma > 5){
    arduinoCounter = line.substring(5, comma).toInt();
}
}

}

//Serial.println(mesaj);
// Actualizare stare modul RFID
RFIDManager_update();

// === Gestionare evenimente RFID (detectare / îndepărtare) ===
static bool cardReported = false;

bool present = isCardPresent();
if(present && lastCardID.length() > 0 && !cardReported){
    // Semnalăm o singură dată când intră în câmp
    char signCode = getSignCode(lastCardID);
    if(signCode == 'S'){
        lastStopSeen = millis();
        if(!stopMode && !stopLatched){
            stopLatched = true;
            Serial.println("[STOP] Semn STOP detectat – oprire 5 secunde");
            DCMotor(false,false);
            ServoMotor(CENTER);
            stopMode = true;
            stopEnd = millis() + 5000;
        }
    } else if(signCode == 'Y'){
        if(!yieldMode){
            yieldMode = true;
            Serial.println("[YIELD] Semn CEDEAZĂ detectat – reducere vitează");
            MOTOR_SPEED = YIELD_SPEED;
        }
    }
}

```

```

    ledcWrite(PIN_MOTOR_ENA, MOTOR_SPEED);

}

}

if(!rfidWriteMode){

    Serial.print("Card RFID detectat: ");
    Serial.println(lastCardID);
    btManager.sendData("RFID:" + lastCardID);

}

cardReported = true;

}

else if(!present && cardReported){

    // Dacă părăsește zona YIELD revenim la viteza normală

    if(yieldMode){

        yieldMode = false;

        Serial.println("[YIELD] Tag CEDEAZĂ îndepărtat – revenire viteză normală");

        MOTOR_SPEED = NORMAL_MOTOR_SPEED;

        ledcWrite(PIN_MOTOR_ENA, MOTOR_SPEED);

    }

    // Tag-ul a plecat din rază -> notificăm o singură dată

    if(!rfidWriteMode){

        Serial.print("Tag-ul: "); Serial.print(lastCardID); Serial.println(" a fost indepartat");

        btManager.sendData("RFID_REMOVED:" + lastCardID);

    }

    cardReported = false;

}

if(rfidWriteMode){

    uint8_t bytes[12] = {0};

    uint8_t Lstr = min((uint8_t)writePayload.length(), (uint8_t)12);

    memcpy(bytes, writePayload.c_str(), Lstr); // rest rămân 0

    Serial.print("[TAG] Writing EPC: "); Serial.println(writePayload);

    bool ok = RFID_writeEpc(bytes, 12); // scriem 6 words complete

    String notif = ok ? "TAG_WRITE:OK" : "TAG_WRITE:ERR";
}

```

```

Serial.print("[BLE] sendData: "); Serial.println(notif);
btManager.sendData(notif);
rfidWriteMode = false;
}

delay(10);
}

```

Modules.cpp

```

/**
 * Modules.cpp - Fișier de agregare pentru compilarea tuturor modulelor Elysium RC
 *
 * Acest fișier include toate implementările (.cpp) modulelor componente ale sistemului Elysium RC.
 * Este necesar deoarece mediul Arduino IDE compilează automat doar fișierele aflate în
 * directorul principal al sketch-ului, iar modulele noastre sunt organizate în subdirectoare
 * separate pe categorii funcționale.
 *
 * Structura modulelor este organizată în următoarele categorii:
 * - Core: Module fundamentale pentru gestionarea task-urilor și comunicației
 * - Motion control: Module pentru controlul motorului DC și servomotorului
 * - Sensors: Module pentru interfațarea cu senzorii (ultrasonic, RFID)
 * - Feedback: Module pentru notificări și alerte
 */

```

```

// Modul pentru gestionarea task-urilor FreeRTOS și sincronizarea lor
#include "../core/TaskManager.cpp"

// Notă: BluetoothManager este implementat direct în header, nu are fișier .cpp separat

```

```
// Modul pentru controlul motorului DC de propulsie
```

```

#include "../motion-control/DCMotor.cpp"
// Modul pentru controlul servomotorului de direcție
#include "../motion-control/ServoMotor.cpp"

// Modul pentru gestionarea senzorilor ultrasonici de distanță
#include "../sensors/UltrasonicSensors.cpp"
// Modul pentru interfațare cu cititorul RFID
#include "../sensors/RFIDManager.cpp"

// Modul pentru control buzzer și alerte sonore
#include "../feedback/BuzzerManager.cpp"
// Modul pentru detecția accidentelor și notificare semne de circulație
#include "../alerts/AccidentDetector.cpp"

```

AccidentDetector.cpp

```

#include "AccidentDetector.h"

#include "../sensors/UltrasonicSensors.h"
#include "ESP32_NOW.h"
#include "WiFi.h"
#include <esp_mac.h>
#include <string.h>

// Configurația canalului WiFi pentru comunicarea ESP-NOW
#define ESPNOW_WIFI_CHANNEL 6

// Parametrii pentru detecția accidentelor și trimiterea notificărilor
#define ACC_INVALID_MS 5000 // Declanșează accident dacă un senzor raportează -1 timp de 5 secunde
#define ACC_COOLDOWN_MS 100 // Perioada minimă între notificări consecutive (evită spam-ul)

```

```

// Structura mesajului transmis prin ESP-NOW către semnele de trafic

// Format identic cu cel folosit în firmware-ul semnelor pentru compatibilitate

typedef struct __attribute__((packed)) traffic_message {

    uint8_t targetId; // 0 = broadcast către toate semnele, alte valori pot identifica un semn specific
    char signType[20]; // Tipul mesajului: "ACCIDENT", "WARNING" etc.
    uint8_t priority; // Prioritatea: 1 = urgent (accident), 0 = informativ
} traffic_message;

// Clasa pentru gestionarea comunicației ESP-NOW în mod broadcast către toate semnele de trafic

class AccidentBroadcastPeer : public ESP_NOW_Peer {

public:

    // Constructor care initializează un peer ESP-NOW în mod broadcast
    AccidentBroadcastPeer(uint8_t channel, wifi_interface_t iface, const uint8_t *lmk) :
        ESP_NOW_Peer(ESP_NOW.BROADCAST_ADDR, channel, iface, lmk) {}

    // Initializează ESP-NOW și adaugă peer-ul pentru comunicare
    bool begin() {
        if (!ESP_NOW.begin() || !add()) {
            return false;
        }
        return true;
    }

    // Trimit un mesaj către toate dispozitivele din raza de acțiune
    bool send_message(const uint8_t *data, size_t len) {
        return send(data, len);
    }
};

// Instanță globală pentru comunicare ESP-NOW în mod broadcast
static AccidentBroadcastPeer broadcast_peer(ESPNOW_WIFI_CHANNEL, WIFI_IF_STA, NULL);

```

```

// Variabile pentru urmărirea stării senzorilor

static unsigned long invalidSince[4] = {0}; // Timestamp-uri pentru ultimele citiri invalide ale senzorilor

static unsigned long lastAccidentMillis = 0; // Timestamp pentru ultima notificare de accident transmisă

// Funcție pentru trimitera notificărilor de accident către semnele de trafic

static void sendAccident() {

    // Pregătește mesajul de notificare a accidentului

    traffic_message msg{};

    msg.targetId = 0; // 0 = broadcast către toate semnele din rază

    strncpy(msg.signType, "ACCIDENT", sizeof(msg.signType)); // Tipul mesajului: accident

    msg.priority = 1; // Prioritate maximă

    // Trimit mesajul folosind ESP-NOW și confirmare în Serial pentru debugging

    broadcast_peer.send_message(reinterpret_cast<uint8_t*>(&msg), sizeof(msg));

    Serial.println("[AccidentDetector] ACCIDENT transmis prin ESP-NOW!");

}

void AccidentDetector_init() {

    // Configurează WiFi în modul Station pentru ESP-NOW

    WiFi.mode(WIFI_STA);

    WiFi.setChannel(ESPNOW_WIFI_CHANNEL); // Configurează canalul WiFi pentru comunicație

    // Așteaptă ca modulul WiFi să fie inițializat complet

    while (!WiFi.STA.started()){

        delay(50);

    }

    // Inițializează comunicarea ESP-NOW pentru broadcast

    if (!broadcast_peer.begin()){

        // Eroare critică, este necesară repornirea sistemului

        Serial.println("[AccidentDetector] Eroare init ESP-NOW, se repornește în 5s");

        delay(5000);

        ESP.restart();

    }

}

```

```

        }

    }

void AccidentDetector_update() {
    // Obține timestamp-ul curent pentru calculul duratelor
    unsigned long now = millis();

    // Obține distanțele de la toți cei patru senzori ultrasonici
    long dists[4] = { distanceFront, distanceBack, distanceLeft, distanceRight };

    // Actualizează starea fiecărui senzor (valid sau invalid)
    for (int i = 0; i < 4; ++i) {
        if (dists[i] == -1) {
            // Dacă senzorul raportează eroare (-1), începe sau continuă cronometrarea perioadei invalide
            if (invalidSince[i] == 0) invalidSince[i] = now; // Începe cronometrarea
        } else {
            // Dacă senzorul raportează o valoare validă, resetează cronometrarea
            invalidSince[i] = 0;
        }
    }

    // Verifică dacă este necesară declansarea alarmei de accident
    bool trigger = false;
    for (int i = 0; i < 4; ++i) {
        // Dacă un senzor a fost invalid mai mult de pragul definit, activează alarma
        if (invalidSince[i] != 0 && (now - invalidSince[i] >= ACC_INVALID_MS)) {
            trigger = true;
            break;
        }
    }

    // Dacă s-a detectat un accident și a trecut perioada de cooldown, trimite notificarea
    if (trigger && (now - lastAccidentMillis > ACC_COOLDOWN_MS)) {

```

```

    sendAccident();

    lastAccidentMillis = now; // Actualizează timestamp-ul ultimei notificări
}

}

```

TaskManager.cpp

```

#include "TaskManager.h"

#include "../sensors/UltrasonicSensors.h"
#include "../feedback/BuzzerManager.h"

extern volatile bool rfidWriteMode;
extern volatile unsigned long muteConsoleUntil;

// Coada FreeRTOS pentru comunicare între task-uri
QueueHandle_t queue;

void Tasks_init(){
    Serial.println("\nCreare coadă mesaje...");
    queue = xQueueCreate(5, sizeof(char[50]));

    Serial.println("\nCreare taskuri...");
    // Creare task pentru monitorizarea obstacolelor cu prioritate scăzută
    xTaskCreate(
        obstacleDetectionTask,
        "ObstacleTask",
        2048,
        NULL,
        1,
        NULL
    );
}

```

```

void obstacleDetectionTask(void *parameter) {
    while (true) {
        long front = distanceFront;
        long back = distanceBack;
        long left = distanceLeft;
        long right = distanceRight;

        if(millis() >= muteConsoleUntil){
            String mesajComplet = String("Obstacle Task: ") +
                "Front: " + String(front) + " " +
                "Back: " + String(back) + " " +
                "Left: " + String(left) + " " +
                "Right: " + String(right);
            Serial.println(mesajComplet);
        }

        // Pauză între verificări pentru a permite rularea altor task-uri
        vTaskDelay(pdMS_TO_TICKS(500));
    }

    vTaskDelete(NULL);
}

```

AutonomousTask.cpp

```
// Rulează pașii prestabiliți de navigație autonomă și
// gestionează semnele STOP/YIELD, obstacolele și preluarea manuală a controlului.

#include "AutonomousTask.h"
#include "../motion-control/DCMotor.h"
#include "../motion-control/ServoMotor.h"
#include "TaskManager.h"
#include "../sensors/UltrasonicSensors.h"

// Semnale de control din programul principal
extern volatile bool stopMode;
extern volatile bool yieldMode;

// Obstacol proximitate
volatile bool rearObstacleDetected = false;
volatile bool frontObstacleDetected = false;
static unsigned long rearObstacleRemovalTime = 0;
static unsigned long frontObstacleRemovalTime = 0;
constexpr int REAR_OBSTACLE_THRESHOLD = 30; // cm
constexpr int FRONT_OBSTACLE_THRESHOLD = 30; // cm
constexpr int OBSTACLE_RESUME_DELAY = 3000; // ms

extern int MOTOR_SPEED; // definit în DCMotor.cpp

volatile OpMode currentMode = MODE_MANUAL;
volatile long arduinoCounter = 0; // pulsuri codor de la UNO

// Parametri autonavă
constexpr int AUTONOM_MOTOR_SPEED = 170;
constexpr long PULSES_PER_METER = 1800;
constexpr long PULSES_LONG = 1200;
```

```

constexpr long PULSES_SHORT    = 720;
constexpr long TURN_LEFT_PULSES = 1000;
constexpr long TURN_RIGHT_PULSES = 1200;
constexpr uint32_t PAUSE_MS     = 1000;

// Secvența de mișcări
enum AutoState{
    IDLE,
    LONG_FWD, PAUSE1,
    STEER_LEFT, PAUSE2,
    LEFT_TURN, PAUSE3,
    STEER_RIGHT, PAUSE4,
    RIGHT_TURN, PAUSE5,
    STEER_CENTER, PAUSE6
};

// Execută logica autonomă pe un task separat
static void autonomousTask(void *parameter) {
    int prevMotorSpeed = MOTOR_SPEED;
    AutoState state   = IDLE;
    long startCnt    = 0;
    uint32_t phaseTs = 0;

    for (;;) {
        // Ieșire rapidă dacă trecem pe control manual
        if (currentMode != MODE_AUTONOMOUS) {
            if (MOTOR_SPEED != prevMotorSpeed) MOTOR_SPEED = prevMotorSpeed;
            if (state != IDLE) { DCMotor(false, false); state = IDLE; }
            vTaskDelay(pdMS_TO_TICKS(50));
            continue;
        }

        // Suspendă secvența la STOP
    }
}

```

```

if (stopMode) { DCMotor(false, false); vTaskDelay(pdMS_TO_TICKS(50)); continue; }

// Gestionare obstacole față/spate

bool blockedRear = (state == RIGHT_TURN) && distanceBack > 0 && distanceBack <
REAR_OBSTACLE_THRESHOLD;

bool blockedFront = (state == LONG_FWD || state == LEFT_TURN) && distanceFront > 0 && distanceFront <
FRONT_OBSTACLE_THRESHOLD;

if (blockedRear) { rearObstacleDetected = true; DCMotor(false, false); }

if (blockedFront) { frontObstacleDetected = true; DCMotor(false, false); }

if (rearObstacleDetected) {

    if (distanceBack <= 0 || distanceBack >= REAR_OBSTACLE_THRESHOLD) {

        if (!rearObstacleRemovalTime) rearObstacleRemovalTime = millis();

        if (millis() - rearObstacleRemovalTime >= OBSTACLE_RESUME_DELAY) {

            rearObstacleDetected = false; rearObstacleRemovalTime = 0;

            if (state == RIGHT_TURN) DCMotor(false, true);

        }

    }

    vTaskDelay(pdMS_TO_TICKS(50));

    continue;

}

if (frontObstacleDetected) {

    if (distanceFront <= 0 || distanceFront >= FRONT_OBSTACLE_THRESHOLD) {

        if (!frontObstacleRemovalTime) frontObstacleRemovalTime = millis();

        if (millis() - frontObstacleRemovalTime >= OBSTACLE_RESUME_DELAY) {

            frontObstacleDetected = false; frontObstacleRemovalTime = 0;

            if (state == LONG_FWD || state == LEFT_TURN) DCMotor(true, false);

        }

    }

    vTaskDelay(pdMS_TO_TICKS(50));

    continue;

}

```

```

// Setează viteza de croazieră când nu cedăm prioritate
if (!yieldMode && MOTOR_SPEED != AUTONOM_MOTOR_SPEED) {
    prevMotorSpeed = MOTOR_SPEED;
    MOTOR_SPEED = AUTONOM_MOTOR_SPEED;
}

switch (state) {
    case IDLE:
        ServoMotor(CENTER);
        startCnt = arduinoCounter;
        DCMotor(true, false);
        state = LONG_FWD;
        break;

    case LONG_FWD:
        if (arduinoCounter - startCnt >= PULSES_LONG) {
            DCMotor(false, false); phaseTs = millis(); state = PAUSE1;
        }
        break;

    case PAUSE1:
        if (millis() - phaseTs >= PAUSE_MS) { ServoMotor(LEFT); phaseTs = millis(); state = STEER_LEFT; }
        break;

    case STEER_LEFT:
        if (millis() - phaseTs >= PAUSE_MS) { startCnt = arduinoCounter; DCMotor(true, false); state = LEFT_TURN; }
        break;

    case LEFT_TURN:
        if (arduinoCounter - startCnt >= TURN_LEFT_PULSES) { DCMotor(false, false); phaseTs = millis(); state = PAUSE3; }
        break;

    case PAUSE3:
        if (millis() - phaseTs >= PAUSE_MS) { ServoMotor(RIGHT); phaseTs = millis(); state = STEER_RIGHT; }

```

```

break;

case STEER_RIGHT:
    if (millis() - phaseTs >= PAUSE_MS) { startCnt = arduinoCounter; DCMotor(false, true); state = RIGHT_TURN; }
    break;

case RIGHT_TURN:
    if (startCnt - arduinoCounter >= TURN_RIGHT_PULSES) { DCMotor(false, false); phaseTs = millis(); state = PAUSE5; }
    break;

case PAUSE5:
    if (millis() - phaseTs >= PAUSE_MS) { ServoMotor(CENTER); phaseTs = millis(); state = STEER_CENTER; }
    break;

case STEER_CENTER:
    if (millis() - phaseTs >= PAUSE_MS) { phaseTs = millis(); state = PAUSE6; }
    break;

case PAUSE6:
    if (millis() - phaseTs >= PAUSE_MS) { startCnt = arduinoCounter; DCMotor(true, false); state = LONG_FWD; }
    break;
}

vTaskDelay(pdMS_TO_TICKS(20));
}
}

void AutonomousTask_init() {
    xTaskCreatePinnedToCore(autonomousTask, "AutonomousTask", 4096, nullptr, 1, nullptr, 1);
}

```

Aplicatia Android

MainActivity.kt

```
package com.example.elysiumrc

import android.Manifest
import android.content.pm.PackageManager
import android.os.Bundle
import android.view MotionEvent
import android.view.View
import android.widget ScrollView
import android.widget.Button
import android.widget.TextView
import android.widget.Toast
import android.widget.PopupMenu
import android.graphics.Color
import android.content.res.ColorStateList
import androidx.activity.enableEdgeToEdge
import androidx.activity.result.contract.ActivityResultContracts
import androidx.appcompat.app.AppCompatActivity
import androidx.core.app.ActivityCompat
import androidx.core.view.ViewCompat
import androidx.core.view.WindowInsetsCompat

class MainActivity : AppCompatActivity() {

    private val numberRegex = Regex("[+]?\\d+(?:\\.\\d+)?")
    private lateinit var bluetoothManager: BluetoothManager
    private lateinit var statusButton: Button
    private lateinit var distanceFrontText: TextView
    private lateinit var distanceBackText: TextView
    private lateinit var distanceLeftText: TextView
    private lateinit var distanceRightText: TextView
    private lateinit var consoleText: TextView
```

```

private lateinit var consoleScroll: ScrollView
private lateinit var counterText: TextView
private lateinit var xText: TextView
private lateinit var yText: TextView
private lateinit var zText: TextView
private lateinit var voltageText: TextView

private lateinit var buttonForward: Button
private lateinit var buttonBackward: Button
private lateinit var buttonLeft: Button
private lateinit var buttonRight: Button
private lateinit var buttonStop: Button
private lateinit var buttonMode: Button
private lateinit var buttonAutoStart: Button
private lateinit var buttonAutoStop: Button

private enum class DriveMode { MANUAL, AUTO, AUTONOMOUS }
private var currentMode = DriveMode.AUTO

private val REQUIRED_PERMISSIONS = arrayOf(
    Manifest.permission.BLUETOOTH,
    Manifest.permission.BLUETOOTH_ADMIN,
    Manifest.permission.BLUETOOTH_CONNECT,
    Manifest.permission.BLUETOOTH_SCAN
)
)

private val requestPermissionLauncher = registerForActivityResult(
    ActivityResultContracts.RequestMultiplePermissions()
) { permissions ->
    if (permissions.all { it.value }) {
        connectToESP32()
    } else {
        Toast.makeText(this, "Permisiiunile Bluetooth sunt necesare", Toast.LENGTH_LONG).show()
    }
}

```

```
    }

}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContentView(R.layout.activity_main)

    statusButton = findViewById(R.id.buttonStatus)
    distanceFrontText = findViewById(R.id.distanceFrontText)
    distanceBackText = findViewById(R.id.distanceBackText)
    distanceLeftText = findViewById(R.id.distanceLeftText)
    distanceRightText = findViewById(R.id.distanceRightText)
    counterText = findViewById(R.id.counterText)
    xText = findViewById(R.id.xText)
    yText = findViewById(R.id.yText)
    zText = findViewById(R.id.zText)
    voltageText = findViewById(R.id.voltageText)

    consoleText = findViewById(R.id.consoleText)
    consoleScroll = findViewById(R.id.consoleScroll)

    buttonForward = findViewById(R.id.buttonForward)
    buttonBackward = findViewById(R.id.buttonBackward)
    buttonLeft = findViewById(R.id.buttonLeft)
    buttonRight = findViewById(R.id.buttonRight)
    buttonStop = findViewById(R.id.buttonStop)
    buttonMode = findViewById(R.id.buttonMode)
    buttonAutoStart = findViewById(R.id.buttonAutoStart)
    buttonAutoStop = findViewById(R.id.buttonAutoStop)

    bluetoothManager = BluetoothManager(this)
```

```

setupControlButtons()
setupModeButton()

if (!bluetoothManager.isBluetoothSupported()) {
    Toast.makeText(this, "Bluetooth nu este suportat", Toast.LENGTH_LONG).show()
    finish()
    return
}

if (hasRequiredPermissions()) {
    connectToESP32()
} else {
    requestPermissionLauncher.launch(REQUIRED_PERMISSIONS)
}

ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets ->
    val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
    v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom)
    insets
}

statusButton.setOnClickListener {
    if (bluetoothManager.isConnected()) {
        bluetoothManager.disconnect()
        updateStatus("Deconectat")
    } else {
        if (hasRequiredPermissions()) {
            connectToESP32()
        } else {
            requestPermissionLauncher.launch(REQUIRED_PERMISSIONS)
        }
    }
}

```

```

buttonAutoStart.setOnClickListener { sendCommand("AUTO_START") }

buttonAutoStop.setOnClickListener { sendCommand("AUTO_STOP") }

buttonAutoStart.visibility = View.GONE
buttonAutoStop.visibility = View.GONE
}

private fun setupControlButtons() {
    buttonForward.setOnTouchListener { _, event ->
        when (event.action) {
            MotionEvent.ACTION_DOWN -> sendCommand("F")
            MotionEvent.ACTION_UP -> sendCommand("N")
        }
        true
    }

    buttonBackward.setOnTouchListener { _, event ->
        when (event.action) {
            MotionEvent.ACTION_DOWN -> sendCommand("B")
            MotionEvent.ACTION_UP -> sendCommand("N")
        }
        true
    }

    buttonLeft.setOnTouchListener{ _, event ->
        when (event.action) {
            MotionEvent.ACTION_DOWN -> sendCommand("L")      // Viraj stânga
            MotionEvent.ACTION_UP  -> sendCommand("C")      // Aduce roțile la centru
        }
        true
    }
}

```

```

// Buton dreapta (direcție)
buttonRight.setOnTouchListener { _, event ->
    when (event.action) {
        MotionEvent.ACTION_DOWN -> sendCommand("R")      // Viraj dreapta
        MotionEvent.ACTION_UP  -> sendCommand("C")      // Centru direcție
    }
    true
}

// Buton stop total (emergency)
buttonStop.setOnClickListener{
    sendCommand("S")           // Stop complet (motor + direcție)
}

private fun setupModeButton(){
    buttonMode.setOnClickListener {
        val popup = PopupMenu(this, buttonMode)
        popup.menu.add("Manual")
        popup.menu.add("Auto")
        popup.menu.add("Autonom")
        popup.setOnMenuItemClickListener { item ->
            when (item.title) {
                "Manual"  -> setDriveMode(DriveMode.MANUAL)
                "Auto"    -> setDriveMode(DriveMode.AUTO)
                "Autonom" -> setDriveMode(DriveMode.AUTONOMOUS)
            }
            true
        }
        popup.show()
    }
}

buttonAutoStart.setOnClickListener { sendCommand("AUTO_START") }

```

```

buttonAutoStop.setOnClickListener { sendCommand("AUTO_STOP") }

setDriveMode(currentMode)
}

private fun setDriveMode(mode: DriveMode) {
    currentMode = mode
    when (mode) {
        DriveMode.MANUAL -> {
            buttonMode.text = "Manual"
            buttonAutoStart.visibility = View.GONE
            buttonAutoStop.visibility = View.GONE
            sendCommand("MODE_MANUAL")
        }
        DriveMode.AUTO -> {
            buttonMode.text = "Auto"
            buttonAutoStart.visibility = View.GONE
            buttonAutoStop.visibility = View.GONE
            sendCommand("MODE_AUTO")
        }
        DriveMode.AUTONOMOUS -> {
            buttonMode.text = "Autonom"
            buttonAutoStart.visibility = View.VISIBLE
            buttonAutoStop.visibility = View.VISIBLE
            sendCommand("MODE_AUTONOMOUS")
        }
    }
}

private fun sendCommand(command: String) {
    bluetoothManager.sendCommand(command)
}

```

```

private fun hasRequiredPermissions(): Boolean {
    return REQUIRED_PERMISSIONS.all{
        ActivityCompat.checkSelfPermission(this, it) == PackageManager.PERMISSION_GRANTED
    }
}

private fun updateStatus(message: String) {
    val lower = message.lowercase()
    when {
        lower.contains("conectat") ->{
            statusButton.text = "Conectat"
            statusButton.backgroundTintList = ColorStateList.valueOf(Color.parseColor("#4CAF50"))
        }
        lower.contains("conectează") || lower.contains("conectare") ->{
            statusButton.text = "Conectare..."
            statusButton.backgroundTintList = ColorStateList.valueOf(Color.parseColor("#FFA500"))
        }
        else ->{
            statusButton.text = "Neconectat"
            statusButton.backgroundTintList = ColorStateList.valueOf(Color.parseColor("#FF0000"))
        }
    }
}

private fun handleIncomingData(data: String) {
    val line = data.trim()
    // Filtrare rapidă după prefix
    if (line.startsWith("\$SEN,")) {
        val nums = line.substring(5).split(",")
        if (nums.size == 4 && nums.all { numberRegex.matches(it.trim()) }) {
            runOnUiThread {
                distanceFrontText.text = nums[0]
                distanceBackText.text = nums[1]
            }
        }
    }
}

```

```

        distanceLeftText.text = nums[2]
        distanceRightText.text = nums[3]
    }
    return
}
}

if (line.startsWith("\$ARD,")) {
    val vals = line.substring(5).split(",")
    if (vals.size == 5 && vals.all { numberRegex.matches(it.trim()) }) {
        runOnUiThread {
            counterText.text = "Encoder rotativ: ${vals[0]}"
            xText.text = "Coord. X: ${vals[1]}"
            yText.text = "Coord. Y: ${vals[2]}"
            zText.text = "Coord. Z: ${vals[3]}"
            voltageText.text = "Tensiune baterie: ${vals[4]}"
        }
        return
    }
}

// Dacă nu a fost consumat mai sus, trimite la consolă
runOnUiThread {
    consoleText.append(line + "\n")
    if (consoleText.text.length > 4000) {
        consoleText.text = consoleText.text.takeLast(4000)
    }
    consoleScroll.post { consoleScroll.fullScroll(View.FOCUS_DOWN) }
}

private fun extractNumber(raw: String): String {
    val match = numberRegex.find(raw)

```

```
        return match?.value ?: raw.trim()
    }

private fun connectToESP32() {
    updateStatus("Se conectează...")
    bluetoothManager.connectToDevice(
        deviceName = "ElysiumRC",
        onConnected = {
            updateStatus("Conectat la ElysiumRC")
            bluetoothManager.setUserDataCallback { data ->
                handleIncomingData(data)
            }
        },
        onError = { error ->
            updateStatus("Eroare: $error")
        }
    )
}

override fun onDestroy(){
    super.onDestroy()
    bluetoothManager.disconnect()
}
```

BluetoothManager.kt

```
package com.example.elysiumrc

import android.Manifest
import android.bluetooth.BluetoothAdapter
import android.bluetooth.BluetoothDevice
import android.bluetooth.BluetoothSocket
import android.content.Context
import android.content.pm.PackageManager
import android.os.Handler
import android.os.Looper
import android.util.Log
import androidx.core.app.ActivityCompat
import java.io.IOException
import java.io.InputStream
import java.io.OutputStream
import java.util.*

class BluetoothManager(private val context: Context) {

    private val bluetoothAdapter: BluetoothAdapter? = BluetoothAdapter.getDefaultAdapter()
    private var bluetoothSocket: BluetoothSocket? = null
    private var outputStream: OutputStream? = null
    private var inputStream: InputStream? = null
    private var isConnected = false
    private val handler = Handler(Looper.getMainLooper())
    private var dataCallback: ((String) -> Unit)? = null

    companion object {
        private const val TAG = "BluetoothManager"
        private val UUID_SERIAL_PORT = UUID.fromString("00001101-0000-1000-8000-00805F9B34FB")
    }
}
```

```

fun isBluetoothSupported(): Boolean {
    return bluetoothAdapter != null
}

fun isBluetoothEnabled(): Boolean {
    return bluetoothAdapter?.isEnabled == true
}

fun setDataCallback(callback: (String) -> Unit) {
    dataCallback = callback
}

fun connectToDevice(deviceName: String, onConnected: () -> Unit, onError: (String) -> Unit) {
    if (ActivityCompat.checkSelfPermission(context, Manifest.permission.BLUETOOTH_CONNECT)
        != PackageManager.PERMISSION_GRANTED) {
        onError("Permisiunile Bluetooth nu sunt acordate")
        return
    }

    val pairedDevices = bluetoothAdapter?.bondedDevices
    val device = pairedDevices?.find { it.name == deviceName }

    Thread {
        try {
            bluetoothSocket = device.createRfcommSocketToServiceRecord(UUID_SERIAL_PORT)
            bluetoothSocket?.connect()
            outputStream = bluetoothSocket?.outputStream
            inputStream = bluetoothSocket?.inputStream
            isConnected = true

            Log.d(TAG, "Conectat cu succes la ${device.name}")
        } catch (e: IOException) {
            Log.e(TAG, "Eroare la conectare: ${e.message}")
            onError(e.message ?: "Eroare la conectare")
        }
    }.start()

    handler.post {
        onConnected()
    }
}

```

```

        onConnected()
    }

    startListening { data ->
        Log.d(TAG, "Date primite: $data")
    }

} catch (e: IOException) {
    Log.e(TAG, "Eroare la conectare: ${e.message}")
    handler.post {
        onError("Eroare la conectare: ${e.message}")
    }
}

}.start()

}

fun sendCommand(command: String) {
    if (! isConnected) {
        Log.e(TAG, "Nu se poate trimite comanda: dispozitivul nu este conectat")
        return
    }

    Thread {
        try {
            Log.d(TAG, "Trimitere comandă: $command")
            outputStream?.write((command + "\n").toByteArray())
            outputStream?.flush()
            Log.d(TAG, "Comandă trimisă cu succes")
        } catch (e: IOException) {
            Log.e(TAG, "Eroare la trimiterea comenzii: ${e.message}")
        }
    }.start()
}

```

```

private fun startListening(onDataReceived: (String) -> Unit) {
    Thread {
        val buffer = ByteArray(1024)
        while (isConnected) {
            try {
                val bytesRead = inputStream?.read(buffer)
                if (bytesRead != null && bytesRead > 0) {
                    val data = String(buffer, 0, bytesRead)
                    Log.d(TAG, "Date primite brut: $data")
                    handler.post {
                        dataCallback?.invoke(data)
                        onDataReceived(data)
                    }
                }
            } catch (e: IOException) {
                Log.e(TAG, "Eroare la citirea datelor: ${e.message}")
                break
            }
        }
    }.start()
}

fun disconnect() {
    isConnected = false
    try {
        bluetoothSocket?.close()
        Log.d(TAG, "Deconectat cu succes")
    } catch (e: IOException) {
        Log.e(TAG, "Eroare la inchiderea conexiunii: ${e.message}")
    }
}

```