Sem. I 2025-2026

Proiect PATR

Sistem de monitorizare a traficului intr-un tunel

Echipa: WildSouth

Grupa: 332AA

Data: January 9, 2026

Componența echipei și contribuția individuală (%)

Membrii	A	C	D	PR	CB	e-mail
Halandut Alexandru-Paul	100%	100%	100%	100%	100%	alexandru.halandut@stud.acs.upb.ro

A-analiza problemei si concepere solutie, C-implementare cod, D-editare documentatie, PR-proofreading, CB - contributie individuala totala (%).

Declaratie: Membrii echipei confirma ca lucrarea respecta regulile de onestitate academica.

Contents

Chapter 1

Introducere. Definire problema

Acest proiect are ca obiectiv simularea unui sistem de monitorizare si control al traficului intr-un tunel rutier, utilizand o platforma Arduino si o serie de senzori care permit detectarea situatiilor de risc si gestionarea fluxului de vehicule. Tunelurile sunt zone cu risc tehnic ridicat, deoarece spatiul inchis limiteaza disiparea fumului, gazelor si caldurii, ceea ce face esentiala detectarea timpurie a oricarui incident care ar putea afecta siguranta participantilor la trafic.

Sistemul implementat simuleaza modul de functionare al unui tunel real: monitorizeaza permanent vehiculele care intra si ies, verifica senzori de mediu si poate bloca accesul atunci cand apare o situatie periculoasa. Acest tip de monitorizare este specific infrastructurilor moderne, in care automatizarea contribuie la reducerea timpilor de reactie si la cresterea nivelului de siguranta.

Componentele sistemului

Tunelul din cadrul proiectului este echipat cu urmatoarele elemente functionale:

- **Doua perechi de senzori intrare-iesire** care numara vehiculele si permit determinarea numarului total de masini aflate simultan in tunel.
- **Senzori MQ2** pentru detectarea scurgerilor de gaze naturale sau a concentratiilor ridicate de fum.
- **Senzori de fum dedicati**, utilizati pentru confirmarea riscului de incendiu.
- **Un buton de panica**, care permite oprirea imediata a accesului in tunel.

- **Un mecanism extern de blocare**, controlat de un operator (de exemplu, serviciul de urgenta).

Aceste componente furnizeaza sistemului informatiile necesare pentru evaluarea starii tunelului si pentru luarea deciziilor de control asupra barierei de acces.

Conditii de blocare a accesului

Accesul in tunel se opreste automat in oricare dintre urmatoarele situatii:

1. **Depasirea numarului maxim admis** de vehicule aflate simultan in tunel.
2. **Detectarea unui incident intern**, precum fum sau gaze in concentratii periculoase.
3. **Activarea butonului de panica**, ceea ce semnaleaza o urgenta in interiorul tunelului.
4. **Blocarea manuala de catre operatorul extern**, atunci cand situatia o impune.

Aceste conditii reflecta protocoalele reale de securitate din tunelurile moderne si permit oprirea imediata a accesului pentru protectia utilizatorilor.

Functionalitati suplimentare

Pentru a simula comportamentul unui sistem complet de monitorizare a unui tunel rutier, proiectul include urmatoarele functionalitati extinse:

- **Alerte vizuale si/sau sonore** activabile automat in caz de pericol (ex.: LED rosu, semnal acustic).
- **Afisarea permanenta a starii tunelului** (sigur, nesigur, acces blocat) pe un display.
- **Actualizarea in timp real a numarului de vehicule**, inclusiv detectarea situatiilor de eroare (ex.: intrari fara iesiri).
- **Mod de operare manual versus automat**: sistemul poate functiona independent sau poate fi controlat de un operator.

- **Simularea unei proceduri de evacuare**, prin blocarea intrarii si mentinerea iesirii deschise in caz de urgenta.
- **Filtrarea falselor alarme**, necesitand confirmarea incidentului prin doi senzori diferiti (ex.: MQ2 + senzor fum).

Prin aceste functionalitati, proiectul nu se limiteaza doar la detectarea incidentelor, ci reproduce comportamentul unui sistem de securitate utilizat in infrastructuri reale, cu monitorizare continua si reactie rapida la evenimente critice.

Chapter 2

Analiza problemei

in aceasta sectiune sunt analizate cerintele si constrangerile sistemului de monitorizare a traficului dintr un tunel, precum si modul in care pot fi validate prin secvente de evenimente.

2.1 cerinte functionale

sistemul trebuie sa simuleze urmatoarele functionalitati

- numararea masinilor care intra si ies din tunel, folosind doua perechi de senzori intrare iesire (doua benzi)
- detectarea incidentelor in interiorul tunelului, pe baza a doi indicatori
  - scurgere de gaze naturale
  - fum (posibil incendiu)
- existenta unui buton de panica ce poate bloca accesul imediat
- posibilitatea ca un operator extern sa blocheze manual intrarea si sau iesirea
- respectarea unei capacitati maxime n de vehicule aflate simultan in tunel (in proiect n este fix, n=5)

2.2 reguli de control al accesului

pe baza cerintei, accesul in tunel trebuie oprit in urmatoarele situatii

1. depasirea numarului maxim n de masini in tunel
2. existenta unui incident intern (gaz sau fum)
3. activarea butonului de panica
4. blocarea manuala de catre operatorul extern

in cazul unui incident sau al panici, se permite doar iesirea din tunel. astfel, intrarea devine blocata automat, iar iesirea ramane permisa, cu exceptia situatiei in care operatorul o blocheaza explicit.

2.3 intrari si iesiri ale sistemului

2.3.1 intrari

intrari de tip eveniment care pot aparea in sistem

- evenimente de trafic
  - intrare banda 1
  - iesire banda 1
  - intrare banda 2
  - iesire banda 2
- evenimente de incident
  - gaz on, gaz off
  - fum on, fum off
- evenimente de panica
  - panic on, panic off
- comenzi operator
  - blocare intrare on off
  - blocare iesire on off

2.3.2 iesiri

iesirile sistemului sunt utilizate pentru a confirma comportamentul si pentru a putea testa solutia

- led intrare permisa (on inseamna ca intrarea este permisa)
- led iesire permisa (on inseamna ca iesirea este permisa)
- led alarma (on inseamna ca exista incident sau panica)
- mesaje in serial monitor care descriu evenimentele si starea curenta

2.4 stari si variabile relevante

starea tunelului este descrisa de urmatoarele variabile logice si numerice

- carsInside numarul curent de masini in tunel
- n capacitatea maxima admisa (n=5)
- gas indica scurgere de gaze (0 sau 1)
- smoke indica fum (0 sau 1)
- panic indica buton de panica activ (0 sau 1)
- opBlockIn blocare manuala intrare (0 sau 1)
- opBlockOut blocare manuala iesire (0 sau 1)

pe baza acestor variabile se calculeaza cele doua stari de iesire

- entryAllowed intrare permisa sau blocata
- exitAllowed iesire permisa sau blocata

2.5 constrangeri si cazuri limita

- carsInside nu trebuie sa devina negativ. o iesire atunci cand carsInside=0 este ignorata
- carsInside nu trebuie sa depaseasca n. orice intrare cand tunelul este plin este respinsa
- daca apare gaz sau fum sau panica, intrarea devine blocata imediat
- operatorul poate bloca intrarea chiar daca nu exista incident
- operatorul poate bloca iesirea, situatie utila pentru testare, desi in realitate iesirea ar trebui mentinuta libera in urgenta

2.6 secvente corecte de executie pentru validare

secventele de mai jos sunt folosite pentru a valida ca sistemul respecta cerintele

2.6.1 secventa 1 umplere pana la n si blocare intrare

- se executa 5 intrari (e1 sau e2 de 5 ori)
- carsInside devine 5
- entryAllowed devine 0, iesirea ramane 1
- orice intrare suplimentara este respinsa

2.6.2 secventa 2 incident intern si permitere doar iesire

- se fac 2 intrari, carsInside devine 2
- se activeaza gas on
- entryAllowed devine 0, exitAllowed ramane 1, alarma devine 1
- o comanda de intrare este respinsa
- o comanda de iesire este acceptata, carsInside scade

2.6.3 secventa 3 panica

- se fac intrari pana la o valoare oarecare
- se activeaza panic on
- intrarea se blocheaza, iesirea ramane permisa, alarma se aprinde
- dupa panic off si daca nu exista alte incidente, intrarea revine permisa doar daca carsInside¡n si operatorul nu blocheaza intrarea

2.6.4 secventa 4 blocare operator

- se activeaza bin on
- entryAllowed devine 0 chiar daca nu exista incident
- se activeaza bin off, iar daca nu exista alte conditii de blocare, entryAllowed revine 1
- se activeaza bout on, iesirea devine 0
- se activeaza bout off, iesirea revine 1

2.7 secvente gresite care trebuie prevenite

- intrare acceptata cand carsInside=n
- intrare acceptata cand gas=1 sau smoke=1 sau panic=1
- carsInside devine negativ prin iesiri repetate cand este deja 0
- entryAllowed ramane 1 dupa aparitia unui incident

2.8 observatii despre simulare

deoarece proiectul este orientat pe partea software, sistemul este testat in simulare folosind wokwi si serial monitor. evenimentele de senzori sunt emulate prin comenzi text, iar comportamentul in timp real este obtinut folosind mecanisme free rtos (task uri, cozi, semafoare, vtaskdelay), fara utilizarea functiei delay.

Chapter 3

Implementarea solutiei

In acest capitol este prezentata implementarea efectiva a aplicatiei, evidentiind structura codului si modul in care mecanismele FreeRTOS sunt folosite pentru a obtine comportamentul cerut. Codul este scris astfel incat sa poata fi rulat in Wokwi, folosind Serial Monitor pentru simularea senzorilor si a evenimentelor de trafic.

3.1 Structura fisierului si dependente

Aplicatia este implementata intr-un singur fisier `.ino`. Pentru programarea in timp real au fost utilizate urmatoarele biblioteci:

- `Arduino_FreeRTOS.h` pentru task-uri si planificator
- `queue.h` pentru cozi de comunicare intre task-uri
- `semphr.h` pentru mutex si semafoare
- `timers.h` pentru infrastructura de temporizare (fara `delay`)

3.2 Initializare (setup)

In `setup()` sunt realizate urmatoarele operatii:

- Initializarea comunicatiei seriale, folosita pentru comenzi si afisare
- Configurarea pinilor pentru LED-uri (intrare permisa, iesire permisa, alarma)

- Initializarea structurii de stare a tunelului (N, numar masini, senzori, blocari)
- Crearea cozilor `qEvents` si `qLog`
- Crearea mutex-ului `mtxState` pentru protectia starii
- Crearea semaforului de numarare `semCapacity` pentru limitarea la N masini
- Crearea task-urilor (Logger, CLI, Controller, Emulator)

Dupa aceste initializari, aplicatia ruleaza exclusiv pe baza task-urilor FreeRTOS, iar functia `loop()` ramane neutilizata.

3.3 Task-ul Logger

Task-ul Logger este singurul task care scrie pe Serial. Celelalte task-uri trimit mesaje in coada `qLog`, iar Logger le afiseaza in ordinea receptionarii, impreuna cu tick-ul FreeRTOS. Aceasta abordare reduce riscul de mesaje intercalate si creste trasabilitatea.

3.4 Task-ul CLI

Task-ul CLI citeste non blocking din Serial Monitor si interpreteaza comenzile introduse de utilizator. Comenzile sunt mapate pe evenimente, iar evenimentele sunt trimise catre Controller prin coada `qEvents`.

Comenzile implementate includ:

- `e1`, `x1`, `e2`, `x2` pentru intrari si iesiri pe cele doua benzi
- `gas on/off`, `smoke on/off`, `panic on/off` pentru incidente si panica
- `bin on/off`, `bout on/off` pentru blocari manuale operator
- `auto on/off`, `rate <ms>`, `burst <k>` pentru emulator
- `status` si `help` pentru afisari si ghidaj

Task-ul foloseste `vTaskDelay()` pentru a ceda procesorul periodic si a evita ocuparea inutila a CPU.

3.5 Task-ul Controller

Task-ul Controller preia evenimentele din `qEvents` si actualizeaza starea tunelului. Accesul la starea globala este protejat de `mtxState`. Pentru limitarea la N masini, intrarile sunt conditionate de semaforul `semCapacity`.

Regulile principale implementate sunt:

- Daca tunelul este plin, intrarea este blocata
- Daca exista gaz, fum sau panica, intrarea este blocata imediat
- Iesirea este permisa in mod normal chiar si in incident, insa poate fi blocata de operator
- Starea este afisata doar la modificare sau la comanda `status`

Dupa fiecare modificare relevanta, Controller recalculeaza `entryAllowed` si `exitAllowed` si actualizeaza LED-urile.

3.6 Task-ul Emulator

Task-ul Emulator simuleaza traficul fara senzori fizici. Cand emulatorul este activ, el genereaza periodic evenimente:

- Daca intrarea este permisa, genereaza intrari alternative pe cele doua benzi
- Daca intrarea este blocata, genereaza iesiri (daca exista masini) pentru a simula evacuarea

Perioada de generare este configurabila prin comanda `rate`.

3.7 Fragment de cod relevant

In continuare este prezentat codul complet al aplicatiei, conform implementarii descrise.

```
#include <Arduino_FreeRTOS.h>
#include <queue.h>
#include <semphr.h>
#include <timers.h>
#include <stdarg.h>

/*
  proiect patr tema tunel

  ideea principala
  sistemul este modelat pe evenimente, ca sa putem simula senzorii fara hardware
  un task citeste comenzi din serial si le transforma in evenimente
  un task controller aplica regulile si actualizeaza starea
  un task logger afiseaza pe serial, ca sa nu se amestece mesajele
  un task emulator poate genera automat trafic (intrari si iesiri) fara senzori

  reguli cerinta
  intrarea se opreste daca
  - se depaseste n=5 masini in tunel
  - exista incident in tunel (gaz sau fum)
  - butonul de panica este activ
  - operatorul blocheaza intrarea

  se permite doar iesirea din tunel in caz de incident sau panica
  adica, in acele situatii, intrarea devine blocata automat, iesirea ramane perm
  operatorul poate bloca si iesirea separat, pentru a simula interventii externe

  comenzi serial explicate
  e1  inseamna o masina intra in tunel pe banda 1
  x1  inseamna o masina iese din tunel pe banda 1
  e2  inseamna o masina intra in tunel pe banda 2
  x2  inseamna o masina iese din tunel pe banda 2

  gas on   simuleaza detectie scurgere gaze naturale
  gas off  simuleaza revenirea la normal, fara scurgere

  smoke on  simuleaza detectie fum (posibil incendiu)
  smoke off simuleaza revenirea la normal, fara fum

  panic on  simuleaza apasarea butonului de panica (alarma)
  panic off simuleaza dezactivarea butonului de panica

  bin on   operatorul blocheaza intrarea in tunel
```

```
  bin off  operatorul deblocheaza intrarea

  bout on  operatorul blocheaza iesirea din tunel
  bout off operatorul deblocheaza iesirea

  status  afiseaza starea curenta manual
  help    afiseaza lista de comenzi

  emulator trafic
  auto on   porneste emularea automata a traficului
  auto off  opreste emularea automata

  rate 800  seteaza perioada emulatorului in milisecunde
            adica la fiecare 800ms emulatorul incearca o intrare sau o iesire

  burst 3   trimite rapid 3 evenimente de intrare (ca un val de masini)
            util pentru testarea blocarii la n masini
*/

// configurare pini led
static const uint8_t PIN_GATE_IN_LED  = 7;   // led intrare permisa - led verde
static const uint8_t PIN_GATE_OUT_LED = 6;   // led iesire permisa - led albastr
static const uint8_t PIN_ALARM_LED    = 13;  // led alarma (incident sau panica)

// capacitate fixa n
static const uint16_t N_DEFAULT = 5;

// polling serial pentru comenzi
static const TickType_t CLI_POLL_MS = 20;

// tipuri de evenimente
enum EventType : uint8_t {
  EV_ENTRY_L1,
  EV_EXIT_L1,
  EV_ENTRY_L2,
  EV_EXIT_L2,

  EV_GAS_ON,
  EV_GAS_OFF,
  EV_SMOKE_ON,
  EV_SMOKE_OFF,
  EV_PANIC_ON,
  EV_PANIC_OFF,
```

```
  EV_OP_BLOCK_IN_ON,
  EV_OP_BLOCK_IN_OFF,
  EV_OP_BLOCK_OUT_ON,
  EV_OP_BLOCK_OUT_OFF,

  EV_STATUS_REQ,
  EV_HELP
};

struct Event {
  EventType type;
  int16_t value;
  TickType_t tick;
};

struct LogMsg {
  TickType_t tick;
  char text[96];
};

// resurse free rtos
static QueueHandle_t qEvents = NULL;
static QueueHandle_t qLog    = NULL;

static SemaphoreHandle_t mtxState = NULL;       // mutex pentru starea globala
static SemaphoreHandle_t semCapacity = NULL;    // counting semaphore pentru n lo

// starea sistemului
struct SystemState {
  uint16_t N;
  int16_t carsInside;

  bool gas;
  bool smoke;
  bool panic;

  bool opBlockIn;
  bool opBlockOut;

  bool entryAllowed;
  bool exitAllowed;
```

```
  // setari emulator
  bool emuOn;
  uint16_t emuRateMs;
  uint8_t emuLaneToggle;
};

static SystemState S;

// log in coada, pentru ca un singur task sa scrie pe serial
static void logf(const char* fmt, ...) {
  if (!qLog) return;

  LogMsg m;
  m.tick = xTaskGetTickCount();

  va_list args;
  va_start(args, fmt);
  vsnprintf(m.text, sizeof(m.text), fmt, args);
  va_end(args);

  xQueueSend(qLog, &m, 0);
}

// calculeaza regulile de acces
static void recomputeGates_locked() {
  const bool incident = (S.gas || S.smoke);
  const bool stopAccess = incident || S.panic;
  const bool full = (S.carsInside >= (int16_t)S.N);

  // intrarea se opreste daca e plin, incident, panica sau operatorul blocheaza
  S.entryAllowed = (!stopAccess) && (!S.opBlockIn) && (!full);

  // iesirea e permisa, dar operatorul o poate bloca separat
  S.exitAllowed  = (!S.opBlockOut);
}

// aplica iesiri pe pini, ca sa se vada in wokwi
static void applyOutputs_locked() {
  digitalWrite(PIN_GATE_IN_LED,  S.entryAllowed ? HIGH : LOW);
  digitalWrite(PIN_GATE_OUT_LED, S.exitAllowed  ? HIGH : LOW);

  const bool alarm = (S.gas || S.smoke || S.panic);
  digitalWrite(PIN_ALARM_LED, alarm ? HIGH : LOW);
```

```
}

// afiseaza un snapshot al starii
static void printState() {
  xSemaphoreTake(mtxState, portMAX_DELAY);
  SystemState snap = S;
  xSemaphoreGive(mtxState);

  logf("state - | n=%u - cars=%d - | - gas=%d - smoke=%d - panic=%d - | - bin=%d - bout=%d - | - in=%d
       snap.N, snap.carsInside,
       snap.gas, snap.smoke, snap.panic,
       snap.opBlockIn, snap.opBlockOut,
       snap.entryAllowed, snap.exitAllowed,
       snap.emuOn, snap.emuRateMs);
}

// trimite un eveniment in coada
static void enqueueEvent(EventType t) {
  if (!qEvents) return;
  Event ev;
  ev.type = t;
  ev.value = 0;
  ev.tick = xTaskGetTickCount();

  if (xQueueSend(qEvents, &ev, 0) != pdTRUE) {
    logf("warn: - qevents - full, - drop - event=%d", (int)t);
  }
}

// task logger, singurul care scrie pe serial, ca sa nu se amestece mesajele
static void TaskLogger(void*) {
  LogMsg m;
  for (;;) {
    if (xQueueReceive(qLog, &m, portMAX_DELAY) == pdTRUE) {
      Serial.print("[");
      Serial.print((unsigned long)m.tick);
      Serial.print("] - ");
      Serial.println(m.text);
    }
  }
}

// task controller, aici se modifica starea si se aplica regulile
```

```
static void TaskController(void*) {
  Event ev;

  for (;;) {
    if (xQueueReceive(qEvents, &ev, portMAX_DELAY) != pdTRUE) continue;

    bool changed = false;
    bool statusRequested = false;

    xSemaphoreTake(mtxState, portMAX_DELAY);

    switch (ev.type) {
      case EV_ENTRY_L1:
      case EV_ENTRY_L2: {
          recomputeGates_locked();
          const int lane = (ev.type == EV_ENTRY_L1) ? 1 : 2;

          if (!S.entryAllowed) {
            logf("entry blocked (lane %d).", lane);
            break;
          }

          if (xSemaphoreTake(semCapacity, 0) == pdTRUE) {
            S.carsInside++;
            changed = true;
            logf("entry ok (lane %d). carsinside=%d", lane, S.carsInside);
          } else {
            logf("entry refused: tunnel full.");
          }
          break;
        }

      case EV_EXIT_L1:
      case EV_EXIT_L2: {
          recomputeGates_locked();
          const int lane = (ev.type == EV_EXIT_L1) ? 1 : 2;

          if (!S.exitAllowed) {
            logf("exit blocked by operator (lane %d).", lane);
            break;
          }

          if (S.carsInside > 0) {
```

```
        S.carsInside--;
        xSemaphoreGive(semCapacity);
        changed = true;
        logf("exit ok (lane %d). carsinside=%d", lane, S.carsInside);
      } else {
        logf("exit ignored: carsinside already 0.");
      }
      break;
    }

  case EV_GAS_ON:
    S.gas = true;
    changed = true;
    logf("gas leak on");
    break;

  case EV_GAS_OFF:
    S.gas = false;
    changed = true;
    logf("gas leak off");
    break;

  case EV_SMOKE_ON:
    S.smoke = true;
    changed = true;
    logf("smoke on");
    break;

  case EV_SMOKE_OFF:
    S.smoke = false;
    changed = true;
    logf("smoke off");
    break;

  case EV_PANIC_ON:
    S.panic = true;
    changed = true;
    logf("panic on");
    break;

  case EV_PANIC_OFF:
    S.panic = false;
    changed = true;
```

```
      logf("panic off");
      break;


    case EV_OP_BLOCK_IN_ON:
      if (!S.opBlockIn) {
        S.opBlockIn = true;
        changed = true;
      }
      logf("operator: block entry on");
      break;

    case EV_OP_BLOCK_IN_OFF:
      if (S.opBlockIn) {
        S.opBlockIn = false;
        changed = true;
      }
      logf("operator: block entry off");
      break;

    case EV_OP_BLOCK_OUT_ON:
      if (!S.opBlockOut) {
        S.opBlockOut = true;
        changed = true;
      }
      logf("operator: block exit on");
      break;

    case EV_OP_BLOCK_OUT_OFF:
      if (S.opBlockOut) {
        S.opBlockOut = false;
        changed = true;
      }
      logf("operator: block exit off");
      break;

    case EV_STATUS_REQ:
      statusRequested = true;
      break;

    case EV_HELP:
      break;
  }
```

```
    if (changed) {
      recomputeGates_locked();
      applyOutputs_locked();
    }

    xSemaphoreGive(mtxState);

    if (changed || statusRequested) {
      printState();
    }

    if (ev.type == EV_HELP) {
      logf("commands:");
      logf("e1 entry lane 1, x1 exit lane 1");
      logf("e2 entry lane 2, x2 exit lane 2");
      logf("gas on/off, smoke on/off, panic on/off");
      logf("bin on/off operator block entry");
      logf("bout on/off operator block exit");
      logf("auto on/off traffic emulator");
      logf("rate <ms> emulator period");
      logf("burst <k> send k entry events");
      logf("status show state, help show commands");
    }
  }
}

// parseaza comenzi din serial
static void handleCommand(char* line) {
  while (*line == ' ' || *line == '\t') line++;
  if (*line == 0) return;

  for (char* p = line; *p; ++p) {
    if (*p >= 'A' && *p <= 'Z') *p = *p - 'A' + 'a';
  }

  if (strcmp(line, "e1") == 0) {
    enqueueEvent(EV_ENTRY_L1);
    return;
  }
  if (strcmp(line, "x1") == 0) {
    enqueueEvent(EV_EXIT_L1);
    return;
```

```
}
if (strcmp(line, "e2") == 0) {
  enqueueEvent(EV_ENTRY_L2);
  return;
}
if (strcmp(line, "x2") == 0) {
  enqueueEvent(EV_EXIT_L2);
  return;
}

if (strcmp(line, "status") == 0) {
  enqueueEvent(EV_STATUS_REQ);
  return;
}
if (strcmp(line, "help") == 0)    {
  enqueueEvent(EV_HELP);
  return;
}

char* a = strtok(line, "-");
char* b = strtok(NULL, "-");
if (!a) return;

if (b) {
  if (strcmp(a, "auto") == 0) {
    const bool on = (strcmp(b, "on") == 0 || strcmp(b, "1") == 0);
    xSemaphoreTake(mtxState, portMAX_DELAY);
    const bool prev = S.emuOn;
    S.emuOn = on;
    xSemaphoreGive(mtxState);

    if (on != prev) {
      logf("auto-%s", on ? "on" : "off");
      printState();
    } else {
      logf("auto-already-%s", on ? "on" : "off");
    }
    return;
  }

  if (strcmp(a, "rate") == 0) {
    const int r = atoi(b);
    if (r < 50 || r > 5000) {
```

```
      logf("rate invalid 50..5000");
      return;
    }

    xSemaphoreTake(mtxState, portMAX_DELAY);
    const uint16_t prev = S.emuRateMs;
    S.emuRateMs = (uint16_t)r;
    xSemaphoreGive(mtxState);

    if (prev != (uint16_t)r) {
      logf("auto rate set to %d ms", r);
      printState();
    } else {
      logf("auto rate unchanged");
    }
    return;
  }

  if (strcmp(a, "burst") == 0) {
    const int k = atoi(b);
    if (k <= 0 || k > 50) {
      logf("burst invalid 1..50");
      return;
    }

    logf("burst %d entries requested", k);
    for (int i = 0; i < k; i++) {
      xSemaphoreTake(mtxState, portMAX_DELAY);
      const bool lane = (S.emuLaneToggle++ & 1);
      xSemaphoreGive(mtxState);
      enqueueEvent(lane ? EV_ENTRY_L2 : EV_ENTRY_L1);
    }
    return;
  }

  const bool on = (strcmp(b, "on") == 0 || strcmp(b, "1") == 0);

  if (strcmp(a, "gas") == 0)    {
    enqueueEvent(on ? EV_GAS_ON    : EV_GAS_OFF);
    return;
  }
  if (strcmp(a, "smoke") == 0) {
    enqueueEvent(on ? EV_SMOKE_ON : EV_SMOKE_OFF);
```

```
      return;
    }
    if (strcmp(a, "panic") == 0) {
      enqueueEvent(on ? EV_PANIC_ON : EV_PANIC_OFF);
      return;
    }
    if (strcmp(a, "bin") == 0)    {
      enqueueEvent(on ? EV_OP_BLOCK_IN_ON   : EV_OP_BLOCK_IN_OFF);
      return;
    }
    if (strcmp(a, "bout") == 0)   {
      enqueueEvent(on ? EV_OP_BLOCK_OUT_ON : EV_OP_BLOCK_OUT_OFF);
      return;
    }
  }

  logf("unknown command. type help");
}

// task cli, citeste serial non blocking
static void TaskCLI(void*) {
  static char buf[80];
  uint8_t idx = 0;

  logf("cli ready. type help.");

  for (;;) {
    while (Serial.available() > 0) {
      const char c = (char)Serial.read();

      if (c == '\r' || c == '\n') {
        if (idx > 0) {
          buf[idx] = 0;
          handleCommand(buf);
          idx = 0;
        }
      } else {
        if (idx < sizeof(buf) - 1) buf[idx++] = c;
      }
    }

    vTaskDelay(pdMS_TO_TICKS(CLI_POLL_MS));
  }
```

```
}

// task emulator, genereaza evenimente periodic, fara hardware
static void TaskSensorEmulator(void*) {
  for (;;) {
    xSemaphoreTake(mtxState, portMAX_DELAY);
    const bool on = S.emuOn;
    const uint16_t rate = S.emuRateMs;
    const bool entryAllowed = S.entryAllowed;
    const bool exitAllowed  = S.exitAllowed;
    const int cars = S.carsInside;
    const bool lane = (S.emuLaneToggle & 1);
    if (on) S.emuLaneToggle++;
    xSemaphoreGive(mtxState);

    if (on) {
      if (entryAllowed) {
        enqueueEvent(lane ? EV_ENTRY_L2 : EV_ENTRY_L1);
      } else {
        if (cars > 0 && exitAllowed) {
          enqueueEvent(lane ? EV_EXIT_L2 : EV_EXIT_L1);
        }
      }
    }

    vTaskDelay(pdMS_TO_TICKS(rate));
  }
}

void setup() {
  Serial.begin(115200);
  Serial.println("boot-ok");

  pinMode(PIN_GATE_IN_LED, OUTPUT);
  pinMode(PIN_GATE_OUT_LED, OUTPUT);
  pinMode(PIN_ALARM_LED, OUTPUT);

  S.N = N_DEFAULT;
  S.carsInside = 0;

  S.gas = false;
  S.smoke = false;
  S.panic = false;
```

```
  S.opBlockIn = false;
  S.opBlockOut = false;

  S.emuOn = false;
  S.emuRateMs = 800;
  S.emuLaneToggle = 0;

  qEvents = xQueueCreate(25, sizeof(Event));
  qLog    = xQueueCreate(30, sizeof(LogMsg));
  mtxState = xSemaphoreCreateMutex();
  semCapacity = xSemaphoreCreateCounting(S.N, S.N);

  if (qEvents == NULL) Serial.println("err: qevents null");
  if (qLog == NULL)    Serial.println("err: qlog null");
  if (mtxState == NULL) Serial.println("err: mtxstate null");
  if (semCapacity == NULL) Serial.println("err: semcapacity null");

  xSemaphoreTake(mtxState, portMAX_DELAY);
  recomputeGates_locked();
  applyOutputs_locked();
  xSemaphoreGive(mtxState);

  xTaskCreate(TaskLogger,          "logger",     256, NULL, 2, NULL);
  xTaskCreate(TaskCLI,             "cli",        256, NULL, 1, NULL);
  xTaskCreate(TaskController,      "controller", 256, NULL, 3, NULL);
  xTaskCreate(TaskSensorEmulator, "emu",        256, NULL, 1, NULL);
}

void loop() {
}
```

3.8 Observatii privind rularea in Wokwi

Aplicatia a fost rulata in Wokwi folosind placa Arduino Mega 2560, deoarece implementarea utilizeaza mai multe task-uri si cozi, iar memoria disponibila pe Arduino Uno poate fi insuficienta pentru aceasta structura.

Interfata de testare este Serial Monitor, unde pot fi introduse comenzile descrise anterior. Starea sistemului este afisata in format compact, iar LED-urile ofera feedback vizual imediat pentru intrare, iesire si alarma.

Bibliografie

- ChatGPT – OpenAI Language Model
- https://lastminuteengineers.com/mq2-gas-senser-arduino-tutorial/
- https://www.youtube.com/watch?v=Z1vYp8NE9LI