Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Departamentul Ingineria Software și Automatică

RAPORT

Lucrare de laborator Nr.2
Disciplina: IoT
Tema: Sisteme de operare:
Secvențiale & Preemptive - FreeRTOS

A efectuat: st.gr.TI-212,
Muntean Mihai

A verificat : asist. univ.
Lupan Cristian

Chișinău 2024

Definirea problemei:

Realizarea unei aplicații pentru MCU care va rula minim 3 task-uri Secvențial, apoi cu FreeRTOS.

Aplicația va rula minim 3 task-uri printre care :

1. Button Led - Schimbare stare LED la detecția unei apăsări pe buton;
2. un al doilea Led Intermitent în faza în care LED-ul de la primul Task e stins;
3. Incrementare/decrementare valoare a unei variabile la apăsarea a doua butoane care va reprezenta numărul de recurențe/timp în care ledul de la al doilea task se va afla într-o stare;
4. Task-ul de Idle se va utiliza pentru afișarea stărilor din program, cum ar fi, afișare stare LED, și afișare mesaj la detecția apăsării butoanelor, o implementare fiind ca la apăsarea butonului sa se seteze o variabila, iar la afișare mesaj - resetare, implementând mecanismul provider/consumer.

Obiective:

1. Implimentarea mai multor funcționalități concomitente;
2. Analiza multitasking-ului pentru MCU;
3. Familiarizarea cu sistemul de operare FreeRTOS pentru microcontroalere.

Introducere

În contextul sistemelor de operare și al FreeRTOS, există două moduri principale de execuție a sarcinilor (task-urilor) pe un microcontroller: secvențial și preemptiv.

Într-un sistem secvențial (sau cooperativ), task-urile sunt executate unul după altul, într-o secvență fixă. Aici nu există o comutare automată între task-uri pe baza priorității sau a altor condiții, ci fiecare task trebuie să "cedeze" controlul manual după ce își termină execuția. Un astfel de sistem este simplu de implementat și de înțeles, însă poate fi ineficient pentru aplicații în timp real, deoarece un task care consumă mult timp poate bloca execuția altora. Fiecare task rulează la finalizarea celui anterior, iar timpul dintre execuții poate varia semnificativ, afectând comportamentul în timp real.

Într-un sistem preemptiv, cum este FreeRTOS, task-urile pot fi întrerupte automat pe baza unui algoritm de planificare. Acest lucru înseamnă că un task poate fi întrerupt pentru a permite unui alt task, de prioritate mai mare, să ruleze, chiar dacă primul task nu și-a terminat complet execuția. Această abordare este ideală pentru aplicații care au constrângeri de timp, unde anumite task-uri, ca cele critice, trebuie să fie executate imediat.

Materiale și metode:

Pentru a putea efectua acestă lucreare de laborator și ansambla un circuit fizic, au fost necesare următoarele materiale:

- 3 Led-uri;
- 3 Rezistoare;
- 3 Butoane;
- Placa Arduino Mega;
- Fire de contact;
- Un BreadBoard.

Fiecare LED este conectat pe breadboard astfel încât catodul (picioarul negativ) să fie legat la masă (GND) printr-o rezistență, iar anodul (picioarul pozitiv) să fie conectat la pini digitali diferiți de pe placa Arduino (pinul 13 pentru LED-ul roșu, 12 pentru LED-ul verde și 11 pentru LED-ul cyan). Rezistențele protejează LED-urile de suprasarcină.

Cele trei butoane sunt amplasate și conectate astfel încât fiecare să fie legat la masă pe una dintre picioare, iar celălalt picior să fie conectat la un pin digital pe Arduino. Aceasta permite detectarea acționării fiecărui buton individual de către placa Arduino.

Toate LED-urile și butoanele au legături la GND (masă) comună pe Arduino, iar restul firelor care duc la pinii digitali permit controlul și monitorizarea acestora. Arduino Mega este sursa de alimentare și control al circuitului, astfel încât, prin programare, fiecare LED poate fi aprins sau stins în funcție de starea butonului sau task-ul dedicat.

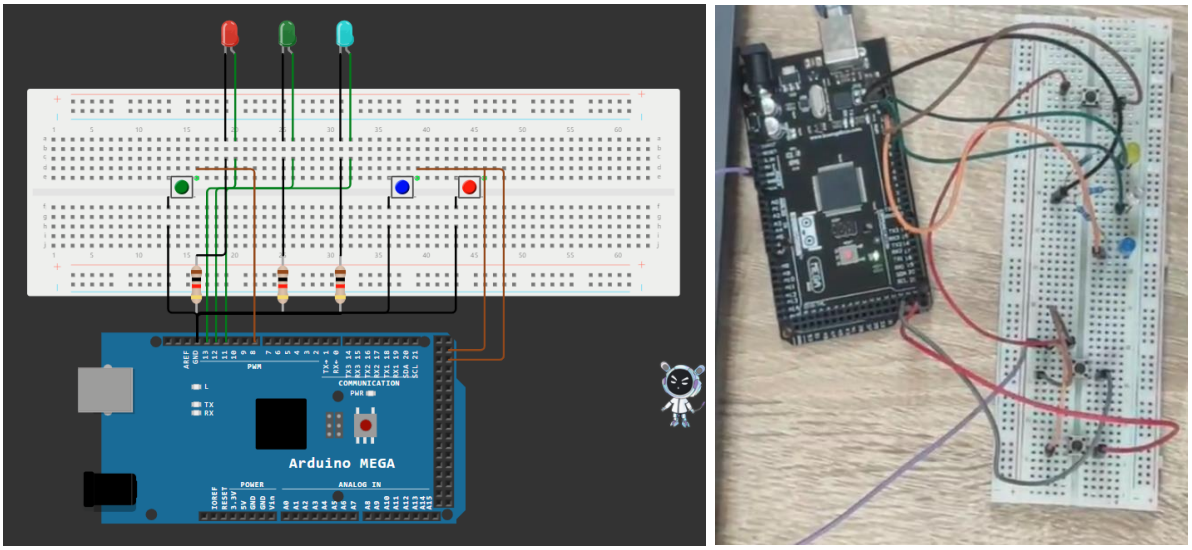


Figura 1 – Ansamblarea circuitului

Efectuarea lucrării:

Sarcina 1

Pentru sarcina 1, programul controlează trei LED-uri conectate la o placă Arduino Mega folosind butoane și comenzi seriale. De asemenea, codul include o sarcină de idle (inactivitate), care afișează starea curentă a sistemului la intervale regulate de timp.

```
void setup(){
  pinMode(MAIN_LED_PIN, OUTPUT);
  pinMode(INTERMITENT_LED_PIN, OUTPUT);
  pinMode(SERIAL_LED_PIN, OUTPUT);
  pinMode(MAIN_BUTTON_PIN, INPUT_PULLUP);
  pinMode(INC_BUTTON_PIN, INPUT_PULLUP);
  pinMode(DEC_BUTTON_PIN, INPUT_PULLUP);
  Serial.begin(115200);
}
```

Trei LED-uri sunt configurate ca ieșiri (OUTPUT): LED-ul principal pe pinul 13, LED-ul intermitent pe pinul 12 și LED-ul controlat prin comandă serială pe pinul 11. Butoanele sunt configurate ca intrări cu rezistențe de pull-up (INPUT_PULLUP), ceea ce înseamnă că atunci când butonul nu este apăsat, pinul citește HIGH, iar când este apăsat, citește LOW. După care se inițializată comunicarea serială la 115200 baund.

În ciclul principal sunt chemate cele 5 task-uri într-un mod secvențial cum și descrie sarcina.

```
void loop(){
  currentTime = millis();
  task1();
  task2();
  task3();
  task4();
  idleTask();
}
```

Funcția millis(), actualizează timpul din variabila currentTime, care este ulterior utilizată pentru a regula procesul de multitasking.

Pentru task-ul 1, butonul principal este verificat constant. Când acesta este apăsat (LOW), starea LED-ului principal se schimbă între aprins și stins.

```
void task1(){
  if(digitalRead(MAIN_BUTTON_PIN) == LOW){
    while(digitalRead(MAIN_BUTTON_PIN) == LOW);  // Așteaptă eliberarea butonului
    ledState = !ledState;  // Comută starea LED-ului principal
    digitalWrite(MAIN_LED_PIN, ledState);
  }
}
```

Ciclul *while* este utilizat pentru a preveni detectarea multiplelor apăsări consecutive.

Task-ul 2 răspunde de funcționarea intermitentă a celui de-al 2-lea LED. LED-ul intermitent se aprinde și se stinge la un interval de timp definit de variabila delayTime, care poate fi ajustată cu ajutorul butoanelor

de incrementare/decrementare din task-ul 3. Dacă LED-ul principal este aprins, LED-ul intermitent va rămâne stins.

```
void task2(){
  if (currentTime - prevTime_T1 > delayTime){
    if (ledState == HIGH){
      interLedState = LOW;
      digitalWrite(INTERMITENT_LED_PIN, interLedState);
    } else {
      digitalWrite(INTERMITENT_LED_PIN, interLedState =
                          interLedState == LOW ? HIGH : LOW);
    }
    prevTime_T1 = currentTime;
  }
}
void task3(){
  if(digitalRead(INC_BUTTON_PIN) == LOW){
    while(digitalRead(INC_BUTTON_PIN) == LOW);  // Așteaptă eliberarea butonului
    delayTime += step;  // Crește intervalul de timp pentru LED-ul intermitent
    digitalWrite(INC_BUTTON_PIN, HIGH);
  }
```

Apăsarea butonului de incrementare (INC_BUTTON_PIN) mărește timpul de întârziere al LED-ului intermitent.

```
  if(digitalRead(DEC_BUTTON_PIN) == LOW){
    if(delayTime > 100){
      while(digitalRead(DEC_BUTTON_PIN) == LOW);  // Așteaptă eliberarea
      delayTime -= step;  // Scade intervalul de timp
      digitalWrite(DEC_BUTTON_PIN, HIGH);
    }
  }
}
```

Apăsarea butonului de decrementare (DEC_BUTTON_PIN) scade acest timp, dar numai dacă intervalul curent este mai mare de 100 ms.

Task-ul 4, cel opțional, descrie controlul celui de-al 3-lea LED din terminalul serial.

```
void task3(){
  if(digitalRead(INC_BUTTON_PIN) == LOW){
    while(digitalRead(INC_BUTTON_PIN) == LOW);  // Așteaptă eliberarea butonului
    delayTime += step;  // Crește intervalul de timp pentru LED-ul intermitent
    digitalWrite(INC_BUTTON_PIN, HIGH);
  }
  if(digitalRead(DEC_BUTTON_PIN) == LOW){
    if(delayTime > 100){
      while(digitalRead(DEC_BUTTON_PIN) == LOW);  // Așteaptă eliberarea
      delayTime -= step;  // Scade intervalul de timp
      digitalWrite(DEC_BUTTON_PIN, HIGH);
    }
  }
}
```

Comenzile permise fiind led on și led off, care aprind și respectiv sting LED-ul 3.

Și în sfârșit task-ul IDLE (inactiv), afișează stările fiecărui LED în momentul procesării tasku-lui.

```
void idleTask(){
  if (currentTime - prevTime_Idle > idleInterval){
    prevTime_Idle = currentTime;

    printf("LED-ul principal este: %s \n", ledState == HIGH ? "Aprins" : "Stins");
    printf("LED-ul intermitent  este: %s \n", interLedState == HIGH ? "Aprins" : "Stins");
    printf("LED-ul serial  este: %s \n", serialLedState == HIGH ? "Aprins" : "Stins");
    printf("Numărul de recurențe/timp în care LED-ul intermitent se va afla într-o stare:
%d \n", delayTime);
  }
}
```

Sarcina 2.

În sarcina 2 codul inițial a fost modificat pentru a funcționa cu FreeRTOS, un sistem de operare în timp real (RTOS) care permite rularea mai multor sarcini (taskuri) în paralel, cu o gestionare mai eficientă a timpului și resurselor. FreeRTOS utilizează conceptul de sarcini care rulează în mod concurent, fiecare având o prioritate și o perioadă de întârziere între execuții.

```
void setup() {
  pinMode(MAIN_LED_PIN, OUTPUT);
  pinMode(INTERMITENT_LED_PIN, OUTPUT);
  pinMode(SERIAL_LED_PIN, OUTPUT);
  pinMode(MAIN_BUTTON_PIN, INPUT_PULLUP);
  pinMode(INC_BUTTON_PIN, INPUT_PULLUP);
  pinMode(DEC_BUTTON_PIN, INPUT_PULLUP);

  Serial.begin(9600);

  xTaskCreate(
    task1, "Task 1", 1000, NULL, 1, NULL
  );
  xTaskCreate(task2, "Task 2", 1000, NULL, 1, NULL);
  xTaskCreate(task3, "Task 3", 1000, NULL, 1, NULL);
  xTaskCreate(task4, "Task 4", 1000, NULL, 1, NULL);
  xTaskCreate(idleTask, "IDLE Task", 1000, NULL, 1, NULL);
}
```

Fiecare sarcină (task1, task2, etc.) este creată folosind funcția xTaskCreate(). Aceasta specifică:

- funcția care definește sarcina;
- numele sarcinii;
- dimensiunea stivei în bytes;
- parametrii sarcinii (NULL în acest caz, deoarece nu sunt necesari);
- prioritatea sarcinii;
- task handle-ul, care este folosit pentru gestionarea sarcinilor.

Notă: Funcția loop() este goală, deoarece FreeRTOS preia controlul asupra execuției programului, gestionând fiecare sarcină individual.

Task-urile, la rândul lor, deasemenea au suferit careva modificări. Funcțiile deja sunt definite într-o buclă infinită și pentru fiecare este stabilită o întârziere, prestabilită la început, înainte de a repeta procesul. Șablonul arată în felul următor:

```
void task(void * param){
  while(1){
    // codul taskului
    vTaskDelay(delayTime / portTICK_PERIOD_MS);
  }
}
```

Exemplu:

```
void task2(void * param)
{
  while(1){
    if (ledState ^ serialLedState )
    {
      interLedState = LOW;
      digitalWrite(INTERMITENT_LED_PIN, interLedState);
    }
    else
    {
      interLedState = (interLedState == LOW) ? HIGH : LOW ;
      digitalWrite(INTERMITENT_LED_PIN, interLedState);
    }
    vTaskDelay(interDelay / portTICK_PERIOD_MS );
  }
}
```

Codul de mai sus servește drept exemplu de modificare a task-urilor din sarcina 1, opțional a fost modificat algoritmul de aprindere a LED-ului intermited. Acesta acum va pulsa atunci când celelalte 2 LED-uri vor avea o stare egală: ori ambele stinse, sau aprinse. Codul în întregime este prezent în anexa 3.

Pentru a implimenta comunicarea între taskuri, mai precis identificarea apăsării unui buton să cheme UI (taskul IDLE) în care sunt prezentate stările Led-urilor, au fost utilizate semafoarele.

```
#include <semphr.h>
void setup() {
  ...
  xSemaphore = xSemaphoreCreateBinary();
  ...
 }

void idleTask(void * param)
{
  while (1) {
```

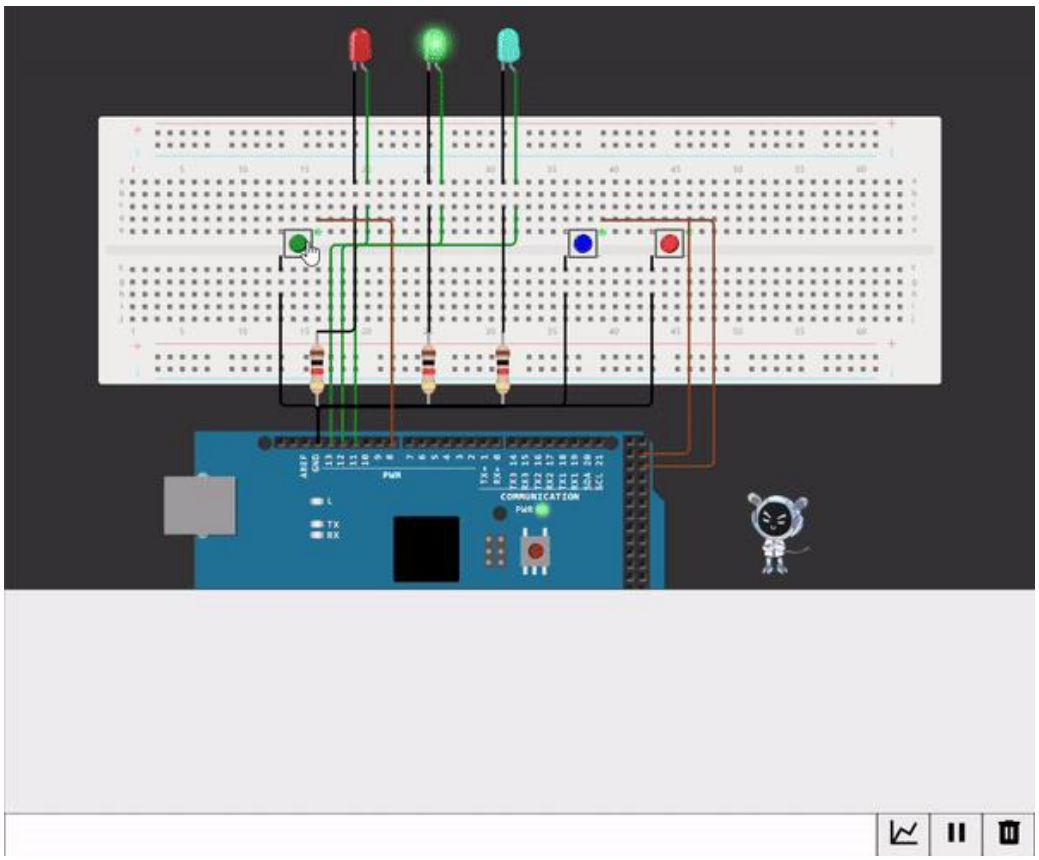
```
    if (xSemaphoreTake(xSemaphore, portMAX_DELAY) == pdTRUE) {
      printf(„...”);
      ...
    }
  }
}
```

Iar pentru a acorda semaforul folosim metoda `xSemaphoreGive(xSemaphore)` în interiorul taskurilor care generează date(modifică starea ledurilor, fie prin terminal fie prin apăsarea butonului).

```
void task1(void * param) {
  while (1) {
    if (digitalRead(MAIN_BUTTON_PIN) == LOW) {
      while (digitalRead(MAIN_BUTTON_PIN) == LOW);
      ledState = !ledState;
      digitalWrite(MAIN_LED_PIN, ledState);
      xSemaphoreGive(xSemaphore);
    }
    vTaskDelay(delayTime / portTICK_PERIOD_MS );
  }
}
```

Rezultate obținute:

Pentru ambele sarcini a fost utilizat același circuit, care operează comform așteptărilor.



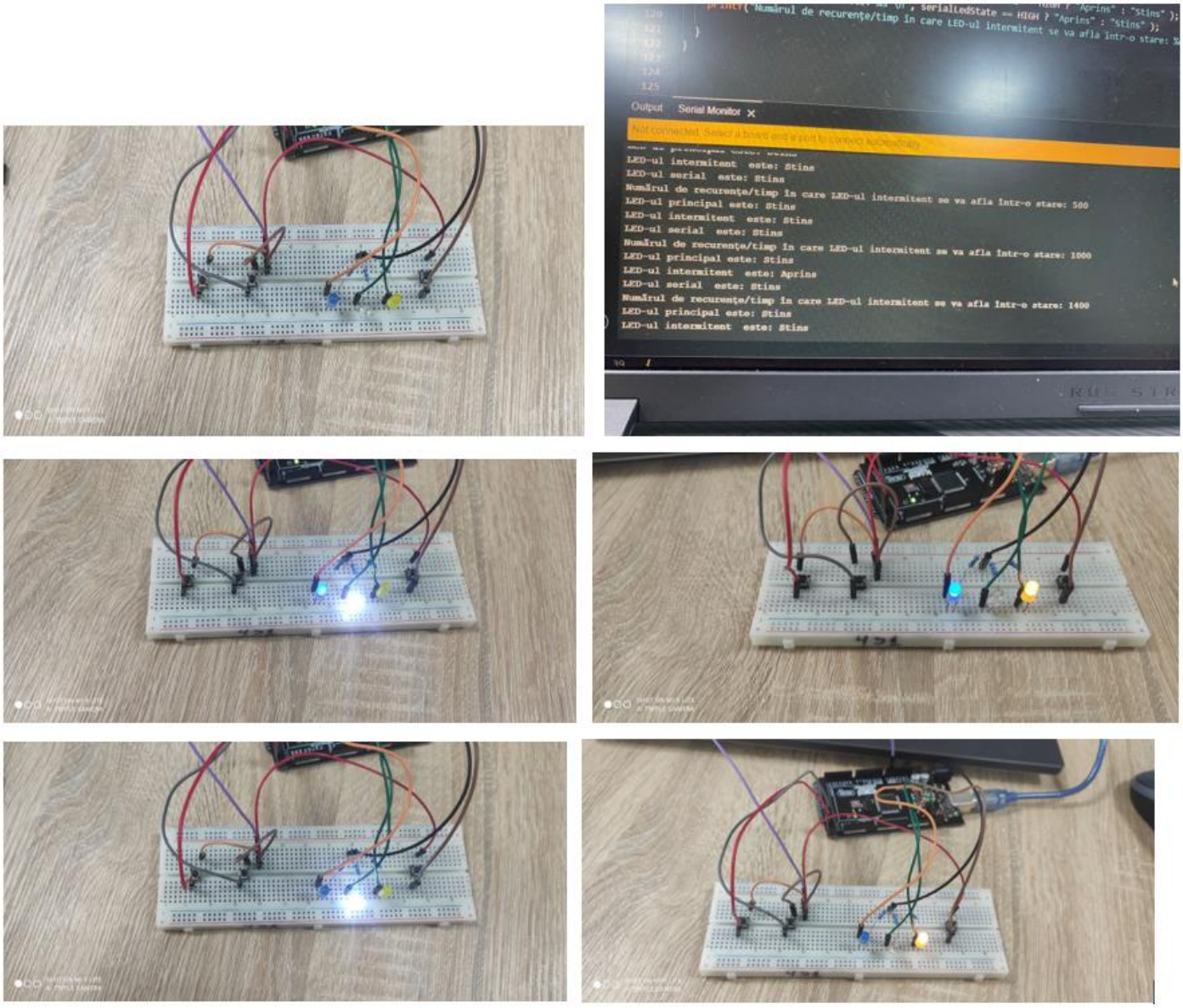


Figura 2 – Rezultatele pe circuit fizic

Rezultatele sunt prezentate în repozitoriul de pe GitHub. Anexa 4.

CONCLUZIE

În urma realizării acestei lucrări de laborator, am reușit să comparăm două metode de gestionare a task-urilor pe un microcontroller: execuția secvențială și utilizarea sistemului de operare FreeRTOS.

Prin implementarea sistemului secvențial, am observat că, în cazul în care un task este blocat — de exemplu, atunci când un buton este apăsat continuu — acest lucru duce la blocarea întregului circuit. Această abordare deși este simplă și ușor de înțeles, devine ineficientă în medii cu cerințe de timp real și poate provoca disfuncții semnificative. În contrast, utilizarea FreeRTOS ne-a permis să implementăm multitasking eficient, unde sarcinile pot rula în paralel și pot fi preemptive. Aceasta a dus la o utilizare mai bună a resurselor și la îmbunătățirea răspunsului sistemului la evenimente externe. Prin utilizarea preemptivității, am evitat problemele întâlnite în execuția secvențială, asigurându-ne că un task care rămâne blocat nu afectează funcționarea altor sarcini.

În concluzie, utilizarea FreeRTOS în dezvoltarea aplicațiilor pentru microcontrolere reprezintă un pas important pentru îmbunătățirea eficienței și performanțelor sistemelor în timp real. Aceasta face ca aplicațiile să fie mai robuste și mai potrivite pentru cerințe complexe.

BIBLIOGRAFII

1. Resursa electronică: https://docs.wokwi.com/?utm_source=wokwi – Regim de acces;
2. Resursa electronică: https://else.fcim.utm.md/course/view.php?id=343 – Regim de acces;
3. Resursa electronică: https://forum.arduino.cc/t/serial-print-and-printf/146256/14 - Regim de acces;

Anexa 1

Fișierul stdinout.h

```
#ifndef _STDINOUT_H
#define _STDINOUT_H

// no need to make an instance of this yourself
class initializeSTDINOUT
{
        static size_t initnum;
public:
        // Constructor
        initializeSTDINOUT();
};

// Call the constructor in each compiled file this header is included in
// static means the names won't collide
static initializeSTDINOUT initializeSTDINOUT_obj;

#endif
```

Fișierul stdinout.cpp

```
#if ARDUINO >= 100
#include "Arduino.h"
#else
#include "WProgram.h"
#endif
#include <stdio.h>
#include "stdinout.h"

// Function that printf and related will use to print
static int serial_putchar(char c, FILE *f)
{
  if (c == '\n') {
    serial_putchar('\r', f);
  }

  return Serial.write(c) == 1 ? 0 : 1;
}
// Function that scanf and related will use to read
static int serial_getchar(FILE *)
{
  // Wait until character is avilable
  while (Serial.available() <= 0) { ; }

  return Serial.read();
}

static FILE serial_stdinout;

static void setup_stdin_stdout()
```

```
{
  // Set up stdout and stdin
  fdev_setup_stream(&serial_stdinout, serial_putchar, serial_getchar, _FDEV_SETUP_RW);
  stdout = &serial_stdinout;
  stdin  = &serial_stdinout;
  stderr = &serial_stdinout;
}

// Initialize the static variable to 0
size_t initializeSTDINOUT::initnum = 0;

// Constructor that calls the function to set up stdin and stdout
initializeSTDINOUT::initializeSTDINOUT()
{
  if (initnum++ == 0) {
    setup_stdin_stdout();
  }
}
```

Anexa 2

Fișierul sketch.ino - Sarcina 1

```
#include "stdinout.h"
#define MAIN_LED_PIN 13
#define INTERMITENT_LED_PIN 12
#define SERIAL_LED_PIN 11
#define MAIN_BUTTON_PIN 8
#define INC_BUTTON_PIN 23
#define DEC_BUTTON_PIN 25

void task1();
void task2();
void task3();
void task4();
void idleTask();

byte ledState = LOW;
byte interLedState = LOW;
byte serialLedState = LOW;
bool btnPressed = false;
unsigned int currentTime = 0;
unsigned int prevTime_T1 = 0;
unsigned int prevTime_Idle = 0;
unsigned int delayTime = 1000;
unsigned int idleInterval = 2000;
unsigned int step = 100;
char command[10];

void setup(){
  pinMode(MAIN_LED_PIN, OUTPUT);
  pinMode(INTERMITENT_LED_PIN, OUTPUT);
  pinMode(SERIAL_LED_PIN, OUTPUT);
  pinMode(MAIN_BUTTON_PIN, INPUT_PULLUP);
  pinMode(INC_BUTTON_PIN, INPUT_PULLUP);
  pinMode(DEC_BUTTON_PIN, INPUT_PULLUP);
  Serial.begin(115200);
}

void loop(){
  currentTime = millis();
  task1();
  task2();
  task3();
  task4();
  idleTask();
}

void task1(){
  if(digitalRead(MAIN_BUTTON_PIN) == LOW){
    while(digitalRead(MAIN_BUTTON_PIN) == LOW);
    ledState = !ledState;
```

```
    digitalWrite(MAIN_LED_PIN, ledState);
  }
}


void task2()
{
  if (currentTime - prevTime_T1 > delayTime)
  {
    if (ledState == HIGH)
    {
      interLedState = LOW;
      digitalWrite(INTERMITENT_LED_PIN, interLedState);
    }
    else
    {
      digitalWrite(INTERMITENT_LED_PIN, interLedState =
                          interLedState == LOW ? HIGH : LOW);
    }
    prevTime_T1 = currentTime;
  }
}

void task3(){
  if(digitalRead(INC_BUTTON_PIN) == LOW){
    while(digitalRead(INC_BUTTON_PIN) == LOW);
    delayTime += step;
    digitalWrite(INC_BUTTON_PIN, HIGH);
  }
  if(digitalRead(DEC_BUTTON_PIN) == LOW){
    if(delayTime > 100){
      while(digitalRead(DEC_BUTTON_PIN) == LOW);
      delayTime -= step;
      digitalWrite(DEC_BUTTON_PIN, HIGH);
    }
  }
}

void task4() {
  if(Serial.available()){
    fgets(command, sizeof(command), stdin);
    command[strcspn(command, "\n")] = 0;
    printf("Received command: %s\n", command);

    if (strcasecmp(command, "led on") == 0) {
      serialLedState = HIGH;
      digitalWrite(SERIAL_LED_PIN, serialLedState);
    }
    else if (strcasecmp(command, "led off") == 0) {
      serialLedState = LOW;
      digitalWrite(SERIAL_LED_PIN, serialLedState);
    }
```

```
   else {
      printf("No such command available!\n");
    }
  }
}

void idleTask()
{
  if (currentTime - prevTime_Idle > idleInterval)
  {
    prevTime_Idle = currentTime;

    printf("LED-ul principal este: %s \n", ledState == HIGH ? "Aprins" : "Stins" );
    printf("LED-ul intermitent  este: %s \n", interLedState == HIGH ? "Aprins" : "Stins"
);
    printf("LED-ul serial  este: %s \n", serialLedState == HIGH ? "Aprins" : "Stins" );
    printf("Numărul de recurențe/timp în care LED-ul intermitent se va afla într-o stare:
%d \n", delayTime );

  }
}
```

Anexa 3

Fișierul sketch.ino – Sarcina 2

```
#include "stdinout.h"
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

#define MAIN_LED_PIN 13
#define INTERMITENT_LED_PIN 12
#define SERIAL_LED_PIN 11
#define MAIN_BUTTON_PIN 8
#define INC_BUTTON_PIN 23
#define DEC_BUTTON_PIN 25

void task1(void * param);
void task2(void * param);
void task3(void * param);
void task4(void * param);
void idleTask(void * param);

byte ledState = LOW;
byte interLedState = LOW;
byte serialLedState = LOW;

unsigned int interDelay = 500;
unsigned int delayTime = 500;
unsigned int step = 100;
char command[10];
```

```
SemaphoreHandle_t xSemaphore;

void setup() {
  pinMode(MAIN_LED_PIN, OUTPUT);
  pinMode(INTERMITENT_LED_PIN, OUTPUT);
  pinMode(SERIAL_LED_PIN, OUTPUT);
  pinMode(MAIN_BUTTON_PIN, INPUT_PULLUP);
  pinMode(INC_BUTTON_PIN, INPUT_PULLUP);
  pinMode(DEC_BUTTON_PIN, INPUT_PULLUP);

  xSemaphore = xSemaphoreCreateBinary();

  Serial.begin(9600);

  xTaskCreate(
    task1, // function name
    "Task 1", // description(task name)
    1000, // stack size
    NULL, // task parameters
    1, // task priority
    NULL // task handle
  );
  xTaskCreate(task2, "Task 2", 1000, NULL, 1, NULL);
  xTaskCreate(task3, "Task 3", 1000, NULL, 1, NULL);
  xTaskCreate(task4, "Task 4", 1000, NULL, 1, NULL);
  xTaskCreate(idleTask, "IDLE Task", 1000, NULL, 1, NULL);
}

void loop() {

}

void task1(void * param) {
  while (1) {
    if (digitalRead(MAIN_BUTTON_PIN) == LOW) {
      while (digitalRead(MAIN_BUTTON_PIN) == LOW);
      ledState = !ledState;
      digitalWrite(MAIN_LED_PIN, ledState);
      xSemaphoreGive(xSemaphore);
    }
    vTaskDelay(delayTime / portTICK_PERIOD_MS );
  }
}


void task2(void * param)
{
  while (1) {
    if (ledState ^ serialLedState )
    {
      interLedState = LOW;
```

```
      digitalWrite(INTERMITENT_LED_PIN, interLedState);
    }
    else
    {
      interLedState = (interLedState == LOW) ? HIGH : LOW ;
      digitalWrite(INTERMITENT_LED_PIN, interLedState);
    }
    vTaskDelay(delayTime / portTICK_PERIOD_MS );
  }
}

void task3(void * param) {
  while (1) {
    if (digitalRead(INC_BUTTON_PIN) == LOW) {
      while (digitalRead(INC_BUTTON_PIN) == LOW);
      delayTime += step;
      digitalWrite(INC_BUTTON_PIN, HIGH);
      xSemaphoreGive(xSemaphore);
    }
    if (digitalRead(DEC_BUTTON_PIN) == LOW) {
      if (delayTime > 100) {
        while (digitalRead(DEC_BUTTON_PIN) == LOW);
        delayTime -= step;
        digitalWrite(DEC_BUTTON_PIN, HIGH);
      }
      xSemaphoreGive(xSemaphore);
    }
    vTaskDelay(delayTime / portTICK_PERIOD_MS );
  }
}

void task4(void * param) {
  while (1) {
    if (Serial.available()) {
      fgets(command, sizeof(command), stdin);
      command[strcspn(command, "\n")] = 0;
      printf("Received command: %s\n", command);

      if (strcasecmp(command, "led on") == 0) {
        serialLedState = HIGH;
        digitalWrite(SERIAL_LED_PIN, serialLedState);
      }
      else if (strcasecmp(command, "led off") == 0) {
        serialLedState = LOW;
        digitalWrite(SERIAL_LED_PIN, serialLedState);
      }
      else {
        printf("No such command available!\n");
      }
      xSemaphoreGive(xSemaphore);
    }
    vTaskDelay(delayTime / portTICK_PERIOD_MS );
```

```
  }
}

void idleTask(void * param)
{
  while (1) {
    if (xSemaphoreTake(xSemaphore, portMAX_DELAY) == pdTRUE) {
      printf("LED-ul principal este: %s \n", ledState == HIGH ? "Aprins" : "Stins" );
     printf("LED-ul intermitent  este: %s \n", interLedState == HIGH ? "Aprins" : "Stins"
);
      printf("LED-ul serial  este: %s \n", serialLedState == HIGH ? "Aprins" : "Stins"
);
       printf("Numărul de recurențe/timp în care LED-ul intermitent se va afla într-o
stare: %d \n", delayTime );
    }
  }
}
```

ANEXA 4

https://github.com/MunMihai/IoT.git