**ROMÂNIA**

**MINISTERUL APĂRĂRII NAȚIONALE**

**ACADEMIA TEHNICĂ MILITARĂ „FERDINAND I”**

**Facultatea de Sisteme Informatice și Securitate Cibernetică**

**Departamentul de Calculatoare și Securitate Cibernetică**



***Utilizare senzor FLACĂRĂ ȘI BUZZER***

***Platforma de dezvoltare frdm-kl25z***

Std. sg. maj. Marcu-Ovidiu TOMA

Std. sg. maj. Dimitrie-Dragoș SANDU

Grupa C114B

**București**

**2024**

Cuprins

[1. Prezentarea senzorului DFR0076(Flame Sensor V2) 3](#_Toc156159207)

[2. Prezentarea Buzzerului 4](#_Toc156159208)

[3. Scop proiect 5](#_Toc156159209)

[4. Conectare senzor – placă de dezvoltare 6](#_Toc156159210)

[5. Descriere program 7](#_Toc156159211)

[5.1. Funcția main 7](#_Toc156159212)

[5.2. Modulul UART 9](#_Toc156159213)

[5.2.1. Inițializare (void UART0\_Init(uint32\_t baud\_rate)) 9](#_Toc156159214)

[5.2.2. Funcția de transmisie 10](#_Toc156159215)

[5.2.3. Funcția de tratare a întreruperii asociată modulului UART0 10](#_Toc156159216)

[5.3. Modulul PIT 11](#_Toc156159217)

[5.3.1. Funcția de inițializare 11](#_Toc156159218)

[5.3.2. Funcția de tartare a întreruperii pentru modulul PIT 12](#_Toc156159219)

[5.4. Modulul ADC 14](#_Toc156159220)

[5.5. Modulul Buzzer 18](#_Toc156159221)

[6. Scheme și diagrame 19](#_Toc156159222)

[6.1. Schema bloc a sistemului 19](#_Toc156159223)

[6.2. Diagrama de stări 20](#_Toc156159224)

[6.2.1. Diagrama de stări pentru LED-uri 20](#_Toc156159225)

[6.2.2. Diagrama de stări pentru senzor flacără și buzzer 20](#_Toc156159226)

[7. Interfața grafică 21](#_Toc156159227)

[8. Dificultăți întâmpinate 23](#_Toc156159228)

[9. Rezultatul proiectului 24](#_Toc156159229)

[10. Referințe 25](#_Toc156159230)

[11. Linkuri 25](#_Toc156159231)

# Prezentarea senzorului DFR0076(Flame Sensor V2)

https://wiki.dfrobot.com/Flame\_sensor\_SKU\_\_DFR0076

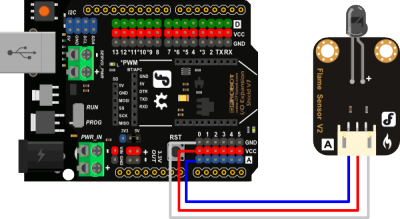
Senzorul DFR0076, produs de DFRobot, este proiectat pentru a detecta flăcări sau lumina în intervalul de lungimi de undă de la 760 nm la 1100 nm si are un unghi de detecție de 60 de grade cu o sensibilitate specială la spectrul flăcării. Funcționează în intervalul de temperatură de la -25°C la +85°C. Este important de menționat că senzorul nu trebuie plasat prea aproape de flacără pentru a evita deteriorarea. Versiunea nouă a senzorului prezintă modificări în maparea pinilor pentru portul analog. Utilizatorii trebuie să fie atenți la aceste schimbări atunci când utilizează scutul de extensie I/O.



*Fig. 1 Senzorul DFR0076*

**Specificații Tehnice**:

* Tensiune de Alimentare: 3.3V până la 5V
* Distanța de Detectare: 20 cm (4.8V) până la 100 cm (1V)
* Intervalul Lățimii de Bandă Spectrală: 760nm până la 1100nm
* Timp de Răspuns: 15us
* Interfață: Analogică
* Dimensiuni: 22x30 mm

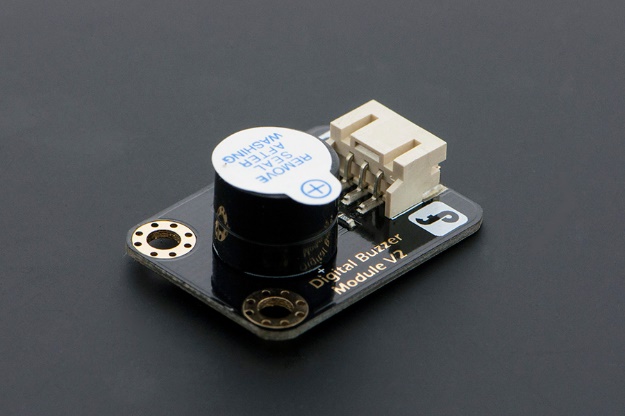


*Fig. 2 Diagrama de conectare a senzorului DFR0076*

|  |  |
| --- | --- |
| *Red* | *VCC* |
| *Black* | *GND* |
| *Blue* | *Analog signal output* |

# Prezentarea Buzzerului

Acest buzzer digital reprezintă cea mai simplă modalitate de a genera sunete în proiectele electronice. Este extrem de ușor de utilizat, deoarece poate fi activat cu impulsuri digitale de High/Low (Înalt/Jos).



*Fig. 3 Buzzer*

**Specificații Tehnice:**

* Tip: Digital
* Alimentare: 5V DC



*Fig. 4 Diagrama Buzzer*

# Scop proiect

Acest proiect are ca obiectiv principal dezvoltarea unei interfețe grafice în Python, care va ilustra în mod dinamic și intuitiv datele captate de la un senzor de flacără în timp real. Interfața va afișa un grafic clar și detaliat, oferind utilizatorilor o viziune imediată asupra intensității și proximității flăcării detectate.

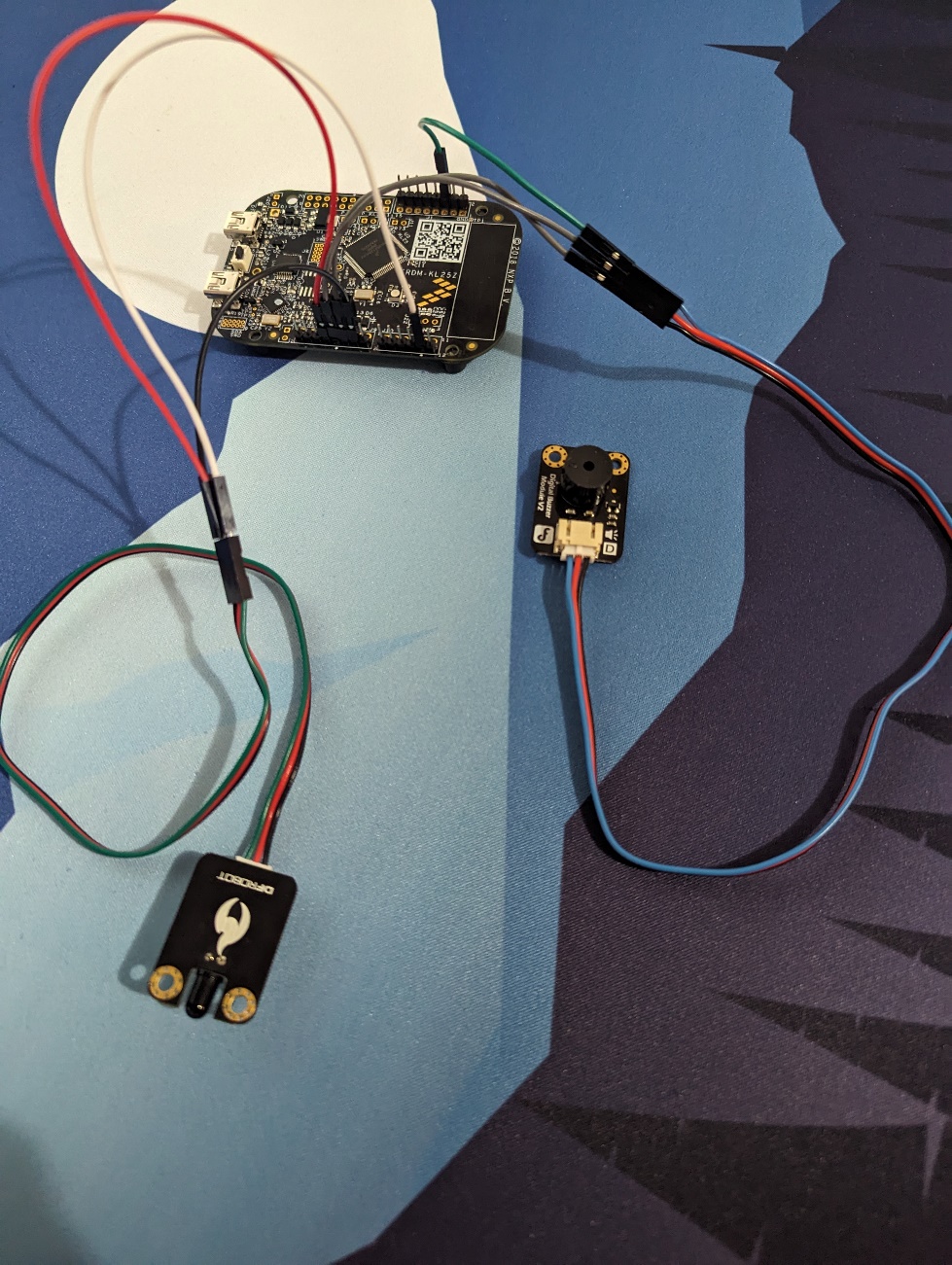
Proiectul va integra un buzzer, programat să emită trei tipuri diferite de sunete. Frecvența și repetiția acestor sunete vor varia (una, două sau trei repetări), bazându-se pe distanța la care se află sursa de flacără de senzor.

LED-ul inclus pe placa de dezvoltare va fi configurat pentru a afișa patru culori diferite (roșu, verde, albastru și galben). Ordinea afișării culorilor va fi configurabilă din interfața grafică.

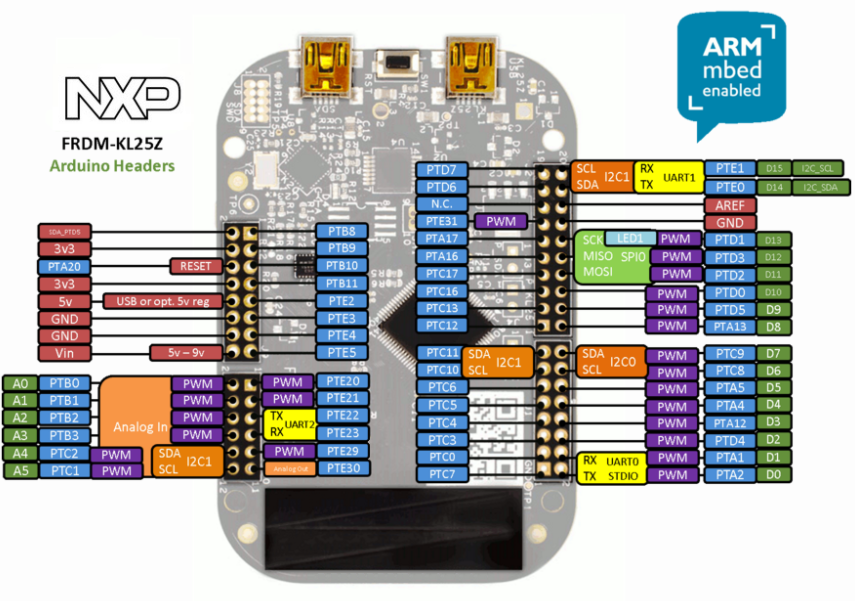
# Conectare senzor – placă de dezvoltare

Vom conecta senzorii astfel:

* Buzzerul:
  + Firul albastru-verde -> PTD4
  + Firul roșu-gri -> 5 V
  + Firul negru-gri -> GND
* Senzorul de flăcară:
  + Firul verde-alb -> PTC2
  + Firul roșu-roșu -> 3.3 V
  + Firul negru-negru -> GND



*Fig. 6 Conectare senzori*



*Fig. 7 Pini placa dezvoltare*

# Descriere program

## Funcția main

În fișierul main.c am inclus fișierele header în care sunt declarate funcțiile și variabilele ce urmează să fie utilizate: Buzzer.h(funcțiile de init\_buzzer, play\_buzzer), Adc.h(funcțiile ADC0\_Init(), ADC0\_IRQHandler, precum și variabilele externe adcFlag și analogInput), Uart.h(UART0\_Init()).

1. #include "Adc.h"

2. #include "Uart.h"

3. #include "ClockSettings.h"

4. #include "Buzzer.h"

5. int main() {

6. UART0\_Init(9600);

7. ADC0\_Init();

8. init\_buzzer();

9. SystemClock\_Configure();

10. SystemClockTick\_Configure();

11. uint8\_t prev\_parte\_zecimala = 0;

12. uint8\_t prev\_parte\_fractionara1 = 0;

13. for(;;) {

14. if(adcFlag==1)

15. {

16.

17. float measured\_voltage = (analogInput \* 3.3f) / 255;

18. uint8\_t parte\_zecimala = (uint8\_t) measured\_voltage;

19. uint8\_t parte\_fractionara1 = ((uint8\_t)(measured\_voltage \* 10)) % 10;

20. uint8\_t parte\_fractionara2 = ((uint8\_t)(measured\_voltage \* 100)) % 10;

21.

22. UART0\_Transmit(parte\_zecimala + 0x30);

23. UART0\_Transmit('.');

24. UART0\_Transmit(parte\_fractionara1 + 0x30);

25. UART0\_Transmit(parte\_fractionara2 + 0x30);

26. UART0\_Transmit(0x0A);

27. UART0\_Transmit(0x0D);

28.

29. if((prev\_parte\_zecimala==0 && prev\_parte\_fractionara1>=5) && (parte\_zecimala==0 && parte\_fractionara1<5))

30. {

31. play\_buzzer(500,1000);

32. }

33. else if((prev\_parte\_zecimala==0 && prev\_parte\_fractionara1<5)&& ((parte\_zecimala==0 && parte\_fractionara1>=5)||(parte\_zecimala>=1 && parte\_zecimala<3)))

34. {

35. play\_buzzer(500,3000);

36. play\_buzzer(2000,0);

37. play\_buzzer(500,3000);

38. }

39. else if(prev\_parte\_zecimala<3 && parte\_zecimala>=3)

40. {

41. play\_buzzer(500,5000);

42. play\_buzzer(2000,0);

43. play\_buzzer(500,5000);

44. play\_buzzer(2000,0);

45. play\_buzzer(500,5000);

46. }

47. else if(prev\_parte\_zecimala>=3 && parte\_zecimala<3)

48. {

49. play\_buzzer(500,3000);

50. play\_buzzer(2000,0);

51. play\_buzzer(500,3000);

52. }

53.

54. prev\_parte\_zecimala = parte\_zecimala;

55. prev\_parte\_fractionara1 = parte\_fractionara1;

56.

57. }

58. }

59.

60. }

**Workflow main:**

1. **Inițializarea Modulelor**: Codul începe prin inițializarea mai multor module hardware și software:
   * **UART0\_Init(9600)**: Inițializează comunicația serială UART cu o viteză de 9600 baud.
   * **ADC0\_Init()**: Inițializează un modul Analog-Digital Converter (ADC).
   * **init\_buzzer()**: Inițializează un buzzer (dispozitiv de emitere a sunetelor).
   * **SystemClock\_Configure()** și **SystemClockTick\_Configure()**: Configură ceasul sistemului și ceasul de tick al sistemului.
2. **Variabile de Stare**: Se definesc variabile pentru stocarea valorilor anterioare (precedente) ale tensiunii măsurate, respectiv **prev\_parte\_zecimala** și **prev\_parte\_fractionara1**.
3. **Bucla Infinită**: În cadrul unei bucle infinite (**for(;;)**), programul execută următoarele operații în mod repetitiv:
   * Verifică dacă flag-ul **adcFlag** este setat la 1, indicând o nouă citire disponibilă de la ADC.
   * Calculează tensiunea măsurată (**measured\_voltage**) bazată pe valoarea citită de la ADC (**analogInput**).
   * Descompune tensiunea măsurată în componentele ei zecimală și fracționară (**parte\_zecimala**, **parte\_fractionara1**, **parte\_fractionara2**).
   * Transmite valorile calculate prin UART, oferind o reprezentare umană citibilă a tensiunii (sub forma unui număr cu două zecimale).
4. **Logica pentru Buzzer**: Logica buzzerului tău este concepută pentru a emite tonuri diferite bazate pe schimbările valorii tensiunii măsurate, categorizate în trei subintervale definite: **0-0.5, 0.5-3, și >=3**. Aceste subintervale reprezintă diferite praguri de tensiune, iar codul reacționează la trecerea dintre aceste praguri astfel:
   * La trecerea din primul subinterval în al doilea (de exemplu, de la 0.4 la 0.6), buzzerul emite două tonuri.
   * La trecerea din al doilea subinterval în primul (de exemplu, de la 0.6 la 0.4), buzzerul emite un singur ton.
   * La trecerea din al doilea în al treilea subinterval (de exemplu, de la 2.9 la 3.1), buzzerul emite trei tonuri.
   * La trecerea din al treilea subinterval înapoi în al doilea (de exemplu, de la 3.1 la 2.9), buzzerul emite două tonuri.
   * La trecerea direct din al treilea subinterval în primul (de exemplu, de la 3.1 la 0.4), buzzerul emite un singur ton.

Codul utilizează funcția play\_buzzer() cu diferiți parametri pentru a genera tonuri la diferite frecvențe și durate. Pauza dintre tonuri a fost definita ca apelul functiei play\_buzzer() cu frecventa 0.

Frecvențele și duratele tonurilor sunt ajustate pentru a reflecta tipul de tranziție între subintervale.

1. **Actualizarea Valorilor Anterioare**: La finalul fiecărei iterații a buclei, valorile curente ale tensiunii sunt salvate ca valori anterioare pentru a fi folosite în iterația următoare.

## Modulul UART

### Inițializare (void UART0\_Init(uint32\_t baud\_rate))

Funcția stabilește sursa de ceas, activează semnalele de ceas pentru UART0 și portul A, și configurează pinii PTA1 și PTA2 pentru comunicarea UART. Această inițializare include setarea Baud Rate-ului, a numărului de biți de date la 8 și dezactivarea bitului de paritate, pregătind modulul pentru transmitere și recepție, iar apoi activează intreruperea pentru recepție.

În codul de mai jos se observă calcularea Baude Rate-ului cu osr-ul dat. (baud\_rate = parametru funcție)

1. //Configurare Baud Rate

2. uint32\_t osr = 4; // Over-Sampling Rate (numarul de esantioane luate per bit-time)

3.

4. //SBR - vom retine valoarea baud rate-ului calculat pe baza frecventei ceasului de sistem

5. // SBR - b16 b15 b14 [b13 b12 b11 b10 b09 b08 b07 b06 b05 b04 b03 b02 b01] &

6. // 0x1F00 - 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0

7. // 0 0 0 b13 b12 b11 b10 b09 0 0 0 0 0 0 0 0 >> 8

8. // BDH - 0 0 0 b13 b12 b11 b10 b09

9. // BDL - b08 b07 b06 b05 b04 b03 b02 b01

10.

11. //DEFAULT\_SYSTEM\_CLOCK is 48000000UL

12. uint32\_t sbr = (48000000UL / ((osr\*4)\*baud\_rate));

13.

14.

15. //(pag 725-726 Reference Manual)

16.

17. uint8\_t temp = UART0->BDH & ~(UART0\_BDH\_SBR(0x1F));

18. UART0->BDH = temp | UART0\_BDH\_SBR(((sbr & 0x1F00)>> 8));

19. UART0->BDL = (uint8\_t)(sbr & UART\_BDL\_SBR\_MASK);

20.

21. //(pag 737 Reference Manual)

22.

23. UART0->C4 |= UART0\_C4\_OSR(osr-1);

24.

### Funcția de transmisie

1. void UART0\_Transmit(uint8\_t data)

2. {

3. //Acest loop asteapta pana cand bitul TDRE (Transmitter Data Register Empty) din registrul UART0->S1 devine 1.

4. //Acest lucru indica faptul ca bufferul de transmitere este gol si poate accepta un nou octet pentru transmitere.

5. //(Reference manual pag 729)

6. while(!(UART0->S1 & UART0\_S1\_TDRE\_MASK));

7. UART0->D = data;

8. }

9.

### Funcția de tratare a întreruperii asociată modulului UART0

Funcția verifică dacă bitul RDRF (Receiver Data Register Full) din registrul de stare UART0->S1 este setat. Dacă acest bit este setat, înseamnă că s-a primit un octet de date și este pregătit să fie citit din registrul de date UART0->D. În funcție de valoarea primită se vor seta niște flag-uri care sunt date ca variabile externe în alte surse. Acestea determină modul de funcționare a programului (direcția LED-urilor, dacă LED-urile active, dacă Buzzer-ul redă sunetul). Acest input va fi transmis serial de către interfața grafică (Cap. 7).

1. void UART0\_IRQHandler(void)

2. {

3. if (UART0->S1 & UART\_S1\_RDRF\_MASK)

4. {

5. char receivedData = UART0->D;

6.

7. if (receivedData == 'a')

8. {

9. if (isOpen == 0)

10. isOpen = 1;

11. else

12. isOpen = 0;

13. }

14.

15. if (receivedData == 'b')

16. {

17. if (direction == 0)

18. direction = 1;

19. else

20. direction = 0;

21. }

22.

23. if (receivedData == 'c')

24. {

25. if (isBuzzer == 0)

26. isBuzzer = 1;

27. else

28. isBuzzer = 0;

29. }

30. }

31. }

## Modulul PIT

### Funcția de inițializare

Funcția PIT\_Init initializează perifericul PIT (Periodic Interrupt Timer). Ea activează semnalul de ceas pentru PIT, configurează modulul pentru a utiliza semnalul de ceas pentru tabloul de timere, oprește decrementarea valorilor numărătoarelor în modul de depanare, și setează valoarea numărătorului pe canalul 0 pentru o perioadă de 0.99 secunde. De asemenea, activează întreruperile pentru canalul 0, pornind timerul și configurează întreruperile asociate cu prioritatea 5, permitând astfel tratarea evenimentelor periodice generate de timerul PIT.

**Calcularea LDVAL**

1. #define DEFAULT\_SYSTEM\_CLOCK 20971520u /\* Default System clock value \*/

2. SIM->CLKDIV1 = 0;

3. SIM->CLKDIV1 |= 0x10010000;

…

4. PIT->CHANNEL[0].LDVAL = 0x9E6665;

Conform Manualului de Referință, avem câmpurile OUTDIV1 și OUTDIV4, care determină factorul cu care este împărțită frecvența pentru ceasul 1 și respectiv ceasul 4. Pentru a realiza o divizare la 2, ambele câmpuri trebuie setate la valoarea 0001, pe care o stabilim cu ajutorul măștii descrise anterior.



*Reference Manual(page 210)*

Apoi, pentru obținerea lui LDVAL (Load Value) divizăm frecvența DEFAULT\_SYSTEM\_CLOCK la 2, înmulțim rezultatul cu numărul de secunde din cerință (0,99), obținând astfel valoarea 10380901 (sau 0x9E6665 în hexazecimal).

### Funcția de tartare a întreruperii pentru modulul PIT

Funcția *PIT\_IRQHandler* gestionează secvența de aprindere a LED-urilor, utilizând modulul PIT (Timer de Perioadă Întreținută). Această funcție este apelată în momentul depășirii unui contor asociat canalului 0 al modulului PIT.

În cadrul acestei funcții:

1. Se verifică dacă flag-ul de depășire (TIF\_MASK) pentru canalul 0 al modulului PIT este activat. Dacă flag-ul este setat, înseamnă că a avut loc o depășire și gestionarul începe să trateze evenimentul asociat.
2. Flag-ul de depășire este apoi dezactivat (PIT->CHANNEL[0].TFLG &= PIT\_TFLG\_TIF\_MASK) pentru a reseta starea de depășire și a pregăti modulul pentru următoarea depășire.
3. Se activează culorile LED-urilor conform secvenței din cerință (RED\_LED, GREEN\_LED, și BLUE\_LED) prin setarea pinilor corespunzători din porturile GPIO (GPIOB\_PSOR și GPIOD\_PSOR).
4. Funcția verifică cele două variabile externe, care se setează din interfața grafică în funcția din modulul UART, mai exact isOpen (variabilă care reprezintă dacă LED-ruile trebuie aprinse sau nu) și direction (care determină ordinea în care va fi afișată secvența)

1. void PIT\_IRQHandler(void)

2. {

3.

4. if(PIT->CHANNEL[0].TFLG & PIT\_TFLG\_TIF\_MASK)

5. {

6. PIT->CHANNEL[0].TFLG &= PIT\_TFLG\_TIF\_MASK;

7.

8. GPIOB\_PSOR |= (1<<RED\_LED\_PIN);

9. GPIOB\_PSOR |= (1<<GREEN\_LED\_PIN);

10. GPIOD\_PSOR |= (1<<BLUE\_LED\_PIN);

11.

12.

13. if(isOpen == 1)

14. {

15. if(state == 0)

16. {

17. if(direction == 1)

18. GPIOB\_PCOR |= (1<<RED\_LED\_PIN);

19.

20. else

21. {

22. GPIOB\_PCOR |= (1<<RED\_LED\_PIN);

23. GPIOB\_PCOR |= (1<<GREEN\_LED\_PIN);

24. }

25.

26. state=1;

27. }

28. else if (state == 1)

29. {

30. if(direction == 1)

31. GPIOB\_PCOR |= (1<<GREEN\_LED\_PIN);

32.

33. else

34. {

35. GPIOD\_PCOR |= (1<<BLUE\_LED\_PIN);

36. GPIOB\_PCOR |= (1<<GREEN\_LED\_PIN);

37. }

38.

39. state=2;

40. }

41. else if (state == 2)

42. {

43. if(direction == 1)

44. {

45. GPIOD\_PCOR |= (1<<BLUE\_LED\_PIN);

46. GPIOB\_PCOR |= (1<<GREEN\_LED\_PIN);

47. }

48.

49. else

50. GPIOB\_PCOR |= (1<<GREEN\_LED\_PIN);

51.

52. state=3;

53. }

54. else if (state == 3)

55. {

56. if(direction == 1)

57. {

58. GPIOB\_PCOR |= (1<<RED\_LED\_PIN);

59. GPIOB\_PCOR |= (1<<GREEN\_LED\_PIN);

60. }

61.

62. else

63. GPIOB\_PCOR |= (1<<RED\_LED\_PIN);

64.

65. state=0;

66. }

67. }

68. }

69. }

Pinii corespunzători LED-urilor sunt inițializați în funcția *void RGBLed\_Init(void)* din fișierul Gpio.c în care se află configurarea modulului GPIO

## Modulul ADC

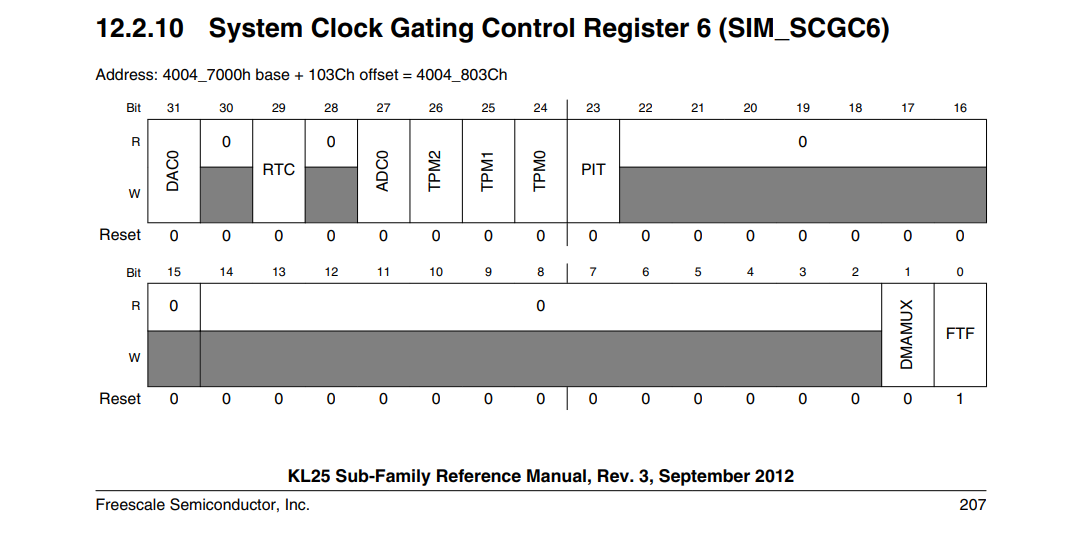
Modulul ADC de pe FRDM KL25Z este esențial pentru captarea datelor analogice și convertirea acestora în format digital. Acest convertor oferă flexibilitatea necesară pentru adaptarea la diferite cerințe de precizie și viteză, grație mai multor niveluri de rezoluție. Cu capacități de eșantionare multi-canale, permite monitorizarea simultană a mai multor semnale, fie că este vorba de măsurători directe sau diferențiale.

Având o interfață programabilă, este ușor de configurat pentru diferite scenarii de utilizare, iar funcțiile sale de declanșare manuală sau automată adaugă versatilitate în utilizare. Modulul este, de asemenea, conceput pentru a fi eficient din punct de vedere energetic, facilitând utilizarea în aplicații ce depind de baterie și necesită un consum redus de energie. În plus, integrarea strânsă cu alte module ale microcontrolerului face ca ADC-ul să fie o componentă centrală în proiectarea sistemelor încorporate cu FRDM KL25Z.

În proiectul acesta, am utilizat modulul ADC pentru a converti semnalele analogice provenite de la senzorul de flacară în date digitale interpretate de microcontroler.

Pentru a activa modulul ADC0, a fost necesar să alimentăm cu energie acest modul pe placa de dezvoltare. Acest pas a fost realizat prin setarea bitului adecvat la valoarea 1 în registrul SIM\_SCGC6, care este responsabil pentru controlul alimentării cu ceas a sistemului.

1. SIM->SCGC6 |= SIM\_SCGC6\_ADC0\_MASK



Am integrat o etapă de calibrare pentru modulul ADC, urmând procedurile recomandate în Manualul de Referință. Procesul de calibrare este esențial pentru a asigura precizia datelor, reducerea erorilor de măsurare și fiabilitatea pe termen lung a sistemului.

1. ADC0\_Calibrate();

Odată finalizată calibrarea, a fost necesar să configurăm modulul ADC cu setările specifice aplicației noastre. Ne-am îndreptat atenția către registrul CFG1 pentru configurare. Am ales o configurație pentru consum normal de energie și am utilizat ceasul plăcii de bază ca sursă de ceas pentru ADC, cu divizorul intern setat la 4, ceea ce ne-a permis să obținem o frecvență a ceasului de intrare de cel mult 4MHz, conform specificațiilor (pagina 494). În plus, am optat pentru o conversie de 8 biți și un timp scurt de eșantionare, ajustând biții în registrul de configurare pentru a reflecta aceste preferințe.

1. ADC0->CFG1 = 0x00;

2.

3.

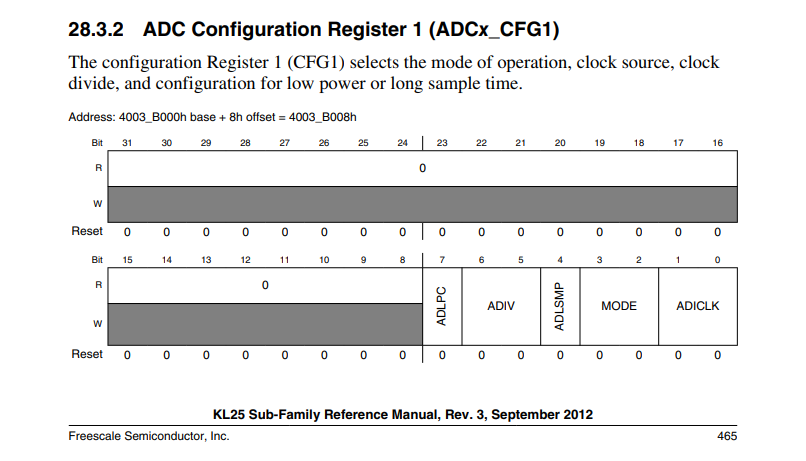
4. // Selectarea modului de conversie pe 8 biti single-ended --> MODE

5. // Selectarea sursei de ceas pentru generarea ceasului intern --> ADICLK

6. // Selectarea ratei de divizare folosit de periferic pentru generarea ceasului intern --> ADIV

7. // Set ADC clock frequency fADCK less than or equal to 4 MHz (PG. 494)

8. ADC0->CFG1 |= ADC\_CFG1\_MODE(0) | ADC\_CFG1\_ADICLK(0) | ADC\_CFG1\_ADIV(2);



Pentru a efectua conversii single-ended cu modulul ADC, este necesar să setăm bitul DIFF la 0. Acest bit se găsește în registrul de status și control 1 (ADC0\_SC1) al convertorului analog-digital. În acest registru, avem de asemenea opțiunea de a activa întreruperile după finalizarea fiecărei conversii, prin setarea câmpului AIEN. În plus, în acest registru specificăm și canalul de intrare pentru citirea datelor de la senzor; în cazul nostru, folosim canalul 11, corespunzător pinului PTC2, la care este conectat senzorul.

De asemenea, configurăm modulul ADC pentru a efectua conversii în mod continuu. Această setare ne permite să utilizăm întreruperile în paralel cu funcționarea modulului ADC, asigurând astfel o monitorizare eficientă și continuă a datelor de intrare de la senzor.

1. // DIFF = 0 --> Conversii single-ended (PG. 464)

2. ADC0->SC1[0] = 0x00;

3. ADC0->SC3 = 0x00;

4.

5. // Selectarea modului de conversii continue,

6. // pentru a-l putea folosi in tandem cu mecanismul de intreruperi

7. ADC0->SC3 |= ADC\_SC3\_ADCO\_MASK;

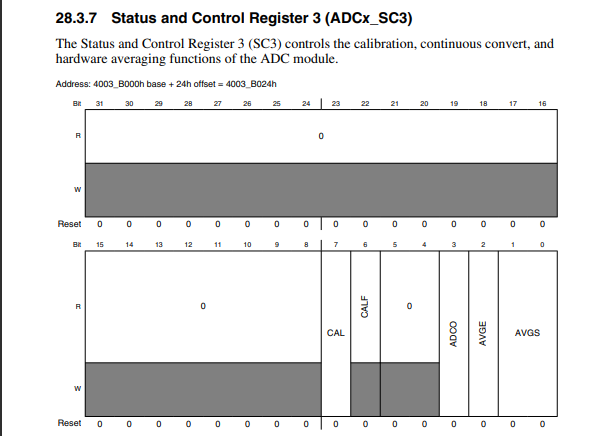
8.

9. // Activarea subsistemului de conversie prin aproximari succesive pe un anumit canal (PG.464)

10. ADC0->SC1[0] |= ADC\_SC1\_ADCH(ADC\_CHANNEL);

11. // Enables conversion complete interrupts

12. ADC0->SC1[0] |= ADC\_SC1\_AIEN\_MASK;



*Reference Manual (Page 472)*

Funcția **ADC0\_Calibrate()** este destinată calibrării modulului ADC de pe microcontrolerul FRDM KL25Z, un pas esențial pentru asigurarea acurateței în citirea semnalelor analogice. În această funcție, modulul ADC este configurat inițial pentru o conversie de 16 biți, cu setări specifice ale ceasului și divizorului. Procesul de calibrare începe automat prin activarea anumitor câmpuri în registrele SC3 și SC2, iar funcția așteaptă finalizarea acestei calibrări, monitorizând bitul COCO din registrul SC1. Dacă calibrarea se finalizează cu succes, se verifică absența erorilor evaluând bitul CALF. Se ajustează apoi offset-ul pentru partea pozitivă și negativă a ADC, calculând o valoare medie din rezultatele de calibrare individuale, la care se adaugă setarea bitului MSB. Aceste valori ajustate sunt stocate în registrele PG și MG, destinat partea pozitivă și respectiv negativă. După finalizarea acestor pași, calibrarea este dezactivată, iar funcția returnează un rezultat care indică dacă procesul a fost un succes sau nu. Această funcție joacă un rol important în precizia și fiabilitatea citirilor ADC în cadrul aplicației.

1. int ADC0\_Calibrate() {

2.

3. ADC0\_CFG1 |= ADC\_CFG1\_MODE(3) | // 16 bits mode

4. ADC\_CFG1\_ADICLK(1)| // Input Bus Clock divided by 2

5. ADC\_CFG1\_ADIV(3); // Clock divide by 8

6.

7. // The calibration will automatically begin if the SC2[ADTRG] is 0. (PG. 495)

8. ADC0->SC2 &= ~ADC\_SC2\_ADTRG\_MASK;

9. // Set hardware averaging to maximum, that is, SC3[AVGE]=1 and SC3[AVGS]=0x11 for an average of 32 (PG. 494)

10. ADC0->SC3 |= (ADC\_SC3\_AVGE\_MASK | ADC\_SC3\_AVGS(3));

11. // To initiate calibration, the user sets SC3[CAL] (PG. 495)

12. ADC0->SC3 |= ADC\_SC3\_CAL\_MASK;

13. // At the end of a calibration sequence, SC1n[COCO] will be set (PG. 495)

14. while(!(ADC0->SC1[0] & ADC\_SC1\_COCO\_MASK));

15. // At the end of the calibration routine, if SC3[CALF] is not

16. // set, the automatic calibration routine is completed successfully. (PG. 495)

17. if(ADC0->SC3 & ADC\_SC3\_CALF\_MASK){

18. return (1);

19. }

20. // ====== CALIBRATION FUNCTION (PG.495) =====

21. // 1. Initialize or clear a 16-bit variable in RAM.

22. uint16\_t calibration\_var = 0x0000;

23. // 2. Add the plus-side calibration results CLP0, CLP1, CLP2, CLP3, CLP4, and CLPS to the variable.

24. calibration\_var += ADC0->CLP0;

25. calibration\_var += ADC0->CLP1;

26. calibration\_var += ADC0->CLP2;

27. calibration\_var += ADC0->CLP3;

28. calibration\_var += ADC0->CLP4;

29. calibration\_var += ADC0->CLPS;

30. // 3. Divide the variable by two.

31. calibration\_var /= 2;

32. // 4. Set the MSB of the variable.

33. calibration\_var |= 0x8000;

34. // 5. Store the value in the plus-side gain calibration register PG.

35. ADC0->PG = ADC\_PG\_PG(calibration\_var);

36. // 6. Repeat the procedure for the minus-side gain calibration value.

37. calibration\_var = 0x0000;

38. calibration\_var += ADC0->CLM0;

39. calibration\_var += ADC0->CLM1;

40. calibration\_var += ADC0->CLM2;

41. calibration\_var += ADC0->CLM3;

42. calibration\_var += ADC0->CLM4;

43. calibration\_var += ADC0->CLMS;

44. calibration\_var /= 2;

45. calibration\_var |= 0x8000;

46. ADC0->MG = ADC\_MG\_MG(calibration\_var);

47.

48. // Incheierea calibrarii

49. ADC0->SC3 &= ~ADC\_SC3\_CAL\_MASK;

50. return (0);

51. }

52.

Funcția **ADC0\_Read()** este folosită pentru a citi o valoare de la modulul ADC (Convertor Analog-Digital) al microcontrolerului FRDM KL25Z. Aceasta inițiază o conversie ADC prin scrierea în registrul SC1A, cu condiția ca acest registru să nu aibă toți biții setați pe 1. Funcția apoi așteaptă ca procesul de conversie să înceapă și să se finalizeze, ceea ce este indicat de schimbarea stării bitului ADACT din registrul SC2 și setarea bitului COCO din SC1. Odată ce conversia este completă și rezultatul este transferat în registrele de date ale rezultatelor, funcția returnează această valoare. Aceasta este o parte esențială a procesului de prelucrare a datelor analogice, convertindu-le în format digital pentru ca microcontrolerul să le poată utiliza.

1. uint16\_t ADC0\_Read(){

2.

3. // A conversion is initiated following a write to SC1A, with SC1n[ADCH] not all 1's (PG. 485)

4. ADC0->SC1[0] |= ADC\_SC1\_ADCH(ADC\_CHANNEL);

5. // ADACT is set when a conversion is initiated

6. // and cleared when a conversion is completed or aborted.

7. while(ADC0->SC2 & ADC\_SC2\_ADACT\_MASK);

8. // A conversion is completed when the result of the conversion is transferred

9. // into the data result registers, Rn (PG. 486)

10. // If the compare functions are disabled, this is indicated by setting of SC1n[COCO]

11. // If hardware averaging is enabled, the respective SC1n[COCO] sets only if

12. // the last of the selected number of conversions is completed (PG. 486)

13. while(!(ADC0->SC1[0] & ADC\_SC1\_COCO\_MASK));

14. return (uint16\_t) ADC0->R[0];

15. }

16.

Funcția **ADC0\_IRQHandler()** este un handler de întrerupere pentru modulul ADC al microcontrolerului. Atunci când o conversie ADC este completată, această funcție este apelată automat. În interiorul funcției, valoarea citită de la ADC este salvată în variabila globală analogInput, iar un flag numit adcFlag este setat pe 1 pentru a indica faptul că o nouă valoare a fost citită și este gata pentru a fi prelucrată. După aceasta, întreruperile de la ADC sunt dezactivate prin resetarea bitului AIEN din registrul SC1. Această funcție este crucială pentru gestionarea asincronă a datelor venite de la ADC, permițând programului principal să știe când sunt disponibile date noi pentru procesare.

1. void ADC0\_IRQHandler(){

2.

3. analogInput=(uint16\_t)ADC0->R[0];

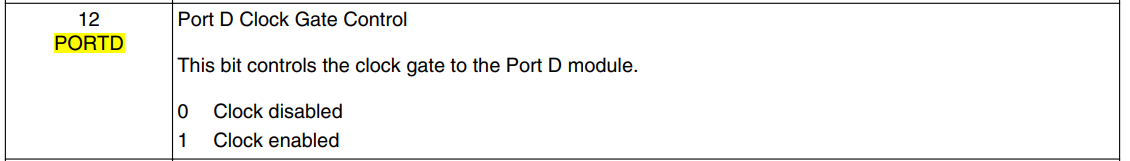
4. adcFlag=1;

5. ADC0->SC1[0] &= ~ADC\_SC1\_AIEN\_MASK;

6. }

## Modulul Buzzer

Funcția de inițializare a buzzerului, arată faptul că acesta este folosit ca și un modul GPIO, și este utilizat pe portul D.



*Reference Manual(page 206)*

Vom activa semnalul de ceas pentru portul D, utilizând pinul 4, pentru a permite comunicația cu buzzer-ul. Acest pas este crucial pentru funcționarea buzzer-ului și este descris de bitul Port D Clock Gate Control. Setând acest bit pe 1, asigurăm că portul D este alimentat cu semnal de ceas, astfel încât buzzer-ul să poată primi semnalele digitale necesare pentru a produce tonuri.

1. void init\_buzzer(void){

2.

3. SIM->SCGC5 |= SIM\_SCGC5\_PORTD\_MASK; // activez portul D

4. PORTD->PCR[buzzer] &= ~PORT\_PCR\_MUX\_MASK;

5. PORTD->PCR[buzzer] |= PORT\_PCR\_MUX(1); // GPIO

6. PTD->PDDR |= MASK(buzzer); // buzzer ca output

7.

Funcția **play\_buzzer** activează buzzer-ul pentru a produce un sunet, controlându-i frecvența și durata in functia main. Parametrul duration specifică cât timp buzzer-ul va suna, în timp ce frequency definește pauza dintre toggle-urile stării buzzer-ului, afectând astfel frecvența tonului. Funcția utilizează o buclă while pentru a repeta schimbarea stării (toggle) a buzzer-ului, creând astfel vibrații auditive.

1. void play\_buzzer(volatile int duration, volatile int frequency)

2. {

3. int cnt = 0;

4. while(cnt != duration){

5. PTD->PTOR = MASK(buzzer); // buzzerul va incepe sa bazaie

6. delay(frequency); // delay-ul stabilit mai sus

7. cnt++;

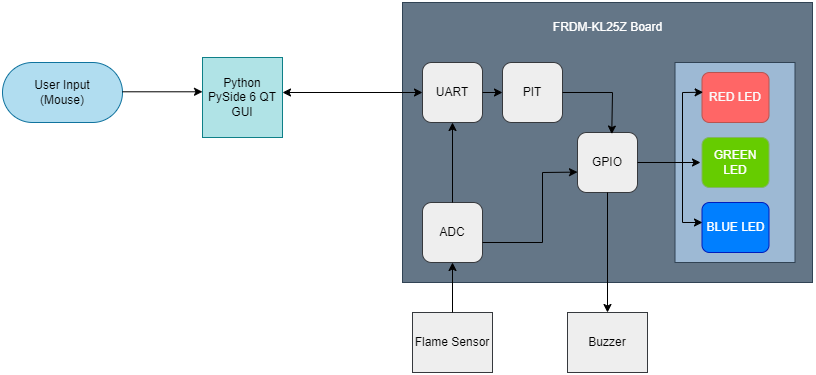
8. }

9. return;

10. }

# Scheme și diagrame

## Schema bloc a sistemului



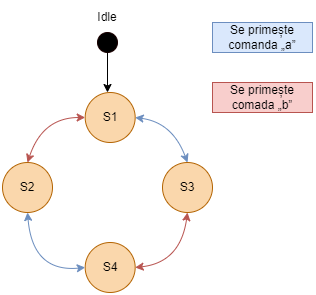
*Fig. 8 Schema bloc*

## Diagrama de stări

Avem două diagrame de stări deoarece aplicația are două flow-uri. Acestea se întâmplă în același timp însă nu sunt interdependente. Culorile chenarelor corespund culorilor săgeților din diagramă

### Diagrama de stări pentru LED-uri

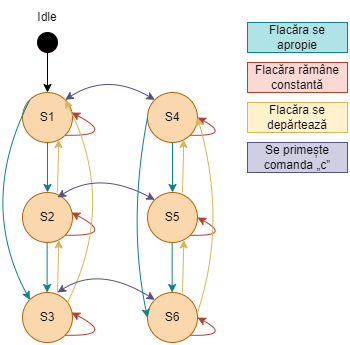
|  |  |  |
| --- | --- | --- |
| Stare | Stare LED-uri | Direcție LED-uri |
| S1 | Pronit | Normală |
| S2 | Pornit | Invers |
| S3 | Oprit | Normală |
| S4 | Oprit | Invers |



*Fig. 9 Schema bloc*

### Diagrama de stări pentru senzor flacără și buzzer

|  |  |  |
| --- | --- | --- |
| Stare | Culoare linie grafic | Buzzer |
| S1 | Verde | Acționat o dată |
| S2 | Galben | Acționat de 2 ori |
| S3 | Roșu | Acționat de 3 ori |
| S4 | Verde | Oprit |
| S5 | Galben | Oprit |
| S6 | Roșu | Oprit |



*Fig. 10 Schema bloc*

# Interfața grafică



*Fig. 11 Interfața grafică*

Interfața grafică este implementată în Python folosind biblioteca *PySide 6* pentru construirea interfeței, iar pentru afișarea graficului, se utilizează *pyqtgraph*. Acest grafic conține valorile primite de la un senzor de flacără.

Aceasta este conectată la la placă serial, folosind biblioteca *serial* din Python. Portul este cel identificat în Device Manager, iar baudrate-ul este 9600 (conform cerinței și configurării plăcii).

1. # Comunicarea seriala

2. self.serial\_port = serial.Serial()

3. self.serial\_port.baudrate = 9600

4. self.serial\_port.port = 'COM3'

5. self.serial\_port.open()

Cele trei butoane situate în partea dreaptă trimit semnale de comunicare serială către placă. Procedura este următoarea: „Toggle LEDs” trimite caracterul „a”, „Change LEDs direction” trimite caracterul „b”, iar „Toggle Buzzer” trimite caracterul „c”. După fiecare transmitere se resetează buffer-ul de output. Aceste intrări sunt receptate prin UART, utilizând întreruperi, de către placă. Ulterior, aceasta procesează informațiile și setează anumite indicatori (flaguri) care vor activa sau dezactiva funcționalitățile corespunzătoare în funcție de numele butoanelor. În momentul apăsării unui buton, se va efectua o modificare a culorii, având ca scop accentuarea vizuală a schimbării efectuate.

1. #Exemplu de cod pentru un buton

2.

3. button1 = QPushButton("Toggle LEDs")

4. button1.setStyleSheet(first\_color)

5.

6. self.button1\_c = 1

7. button1.clicked.connect(self.toggle\_leds)

8. def toggle\_leds(self):

9. self.serial\_port.write(b'a')

10. self.serial\_port.reset\_output\_buffer()

11. if self.button1\_c == 1:

12. self.button1.setStyleSheet(second\_color)

13. self.button1\_c = 0

14. else:

15. self.button1.setStyleSheet(first\_color)

16. self.button1\_c = 1

Pentru grafic se folosește un obiect de tip PlotWidget(), inițializat cu 20 de elemente. Configurările includ setarea fundalului la culoarea neagră și activarea afișării grid-ului.

1. plot\_widget = pg.PlotWidget()

2. plot\_widget.setBackground("black")

3. plot\_widget.showGrid(x=True, y=True, alpha=0.5)

4. self.bar\_plot = pg.BarGraphItem(x=range(1, 21), height=[0]\*20, width=0.3)

5. plot\_widget.addItem(self.bar\_plot)

6. self.plot\_data = [0]\*20

Pentru preluarea datelor de la placă avem funcția read\_serial\_data(self). Pentru ca aceasta să fie apelată periodic este creat

1. self.timer = QTimer(self)

2. self.timer.timeout.connect(self.read\_serial\_data)

3. self.timer.start(100) un timer.

Funcția citește caractere individual până când întâlnește caracterul '\r'. După identificarea acestui caracter, urmează patru caractere în formatul n.nn, care sunt citite și convertite într-o valoare de tip float. Această valoare este adăugată la lista de date a graficului, care este menținută la o lungime constantă de 20 de elemente. Apoi, în funcție de valoarea obținută, se adaugă în lista de culori una dintre opțiunile: verde, galben sau roșu, iar opțiunile graficului sunt actualizate (pozițiile, valoarea, lățimea și culorile barelor). În final, în caseta de jos (values) vor fi afișate datele primite. În cazul unor condiții de eroare, se afișează un mesaj de eroare corespunzător.

1. def read\_serial\_data(self):

2. char = self.serial\_port.read(1)

3. if char == b'\r':

4. bvalue = self.serial\_port.read(4).decode()

5. self.serial\_port.reset\_input\_buffer()

6. try:

7. value = float(bvalue)

8. self.plot\_data.append(value)

9. self.plot\_data = self.plot\_data[-20:]

10. x\_positions = range(1, len(self.plot\_data) + 1)

11.

12. color\_list = []

13. for val in self.plot\_data:

14. if val <= 0.5:

15. color\_list.append('g')

16. elif 0.5 < val <= 3.0:

17. color\_list.append('y')

18. else:

19. color\_list.append('r')

20.

21. self.bar\_plot.setOpts(x=x\_positions, height=self.plot\_data, width=0.9, brushes=color\_list)

22. self.text\_edit.insertPlainText(f"Received data: {value}\n")

23. except ValueError:

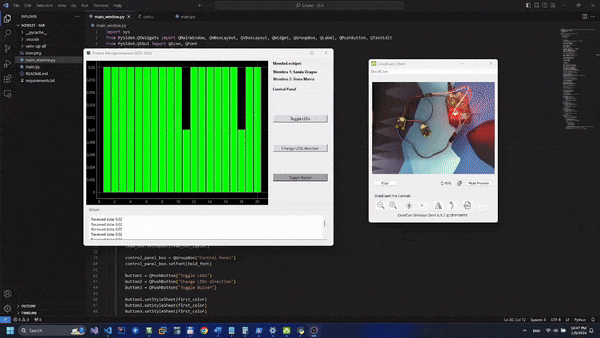
24. self.text\_edit.insertPlainText(f"Invalid data received: {bvalue}\n")

25.

# Dificultăți întâmpinate

* Problema: Calculam LDVAL și LED-urile nu se schimbau la secundele pe care le foloseam în formulă
* Soluție: SIM->CLKDIV1 |= 0x10010000; -> setarea câmpurile OUTDIV1 și OUTDIV4 pentru împărțirea frecvențelor ceasurilor la 2
* Problema: Nu primeam datele corect pe interfața serială
* Soluția: În configurarea interfeței UART, în ciuda faptului că DEFAULT\_SYSTEM\_CLOCK apărea ca fiind *20971520u* l-am luat tot ca fiind *48000000UL* și a funcționat
* Problema: Nu se aprindea secvența de LED-uri corect
* Soluția: Le-am setat pe toate în void RGBLed\_Init(void) cu PCOR
* Problema: Nu am putut filma ziua, razele soarelui dereglau senzorul
* Soluția: Am filmat după ce a apus soarele .

# Rezultatul proiectului



# Referințe

1. FRDM-KL25Z – Reference Manual
2. FRDM-KL25Z - Schematics
3. FRDM-KL25Z - UserManual
4. <https://wiki.dfrobot.com/Digital_Buzzer_Module__SKU__DFR0032_>
5. <https://wiki.dfrobot.com/Flame_sensor_SKU__DFR0076>
6. <https://www.pythonguis.com/tutorials/pyside6-plotting-pyqtgraph/>
7. <https://pyserial.readthedocs.io/en/latest/shortintro.html>

# Linkuri

* GitHub:

<https://github.com/marcu26/ProiectMicroProcesoare>

* Video:

<https://drive.google.com/file/d/1V_wYAeTyLueFmyha0ltpNOo_qpJsNJpy/view?usp=sharing> (cele două filmări au fost lipite într-una. 00:00 – 07:35 Sandu Dragoș, 07:35 – 15: 36 Toma Marcu)