ROMÂNIA

MINISTERUL APĂRĂRII NAȚIONALE

ACADEMIA TEHNICĂ MILITARĂ „FERDINAND I”

Facultatea de Sisteme Informatice și Securitate Cibernetică

Departamentul de Calculatoare și Securitate Cibernetică

UTILIZARE SENZOR DIGITAL BUZZER PE BAZA SENZORULUI DE TEMPERATURĂ INTEGRAT

PLATFORMA DE DEZVOLTARE FRDM-KL25Z

Std. sg. maj. Ioana MISTRIC

Std. sg. maj. Teodora BANU

Grupa C114-D

București

2024

Cuprins

1. Introducere

1.1 Scopul proiectului

Acest proiect are ca scop dezvoltarea și implementarea unui sistem pe placa de dezvoltare FRDM-KL25Z, utilizând senzorul integrat de temperatură, ledul RGB integrat și un buzzer digital conectat la placă. De asemena, am inclus în dezvoltarea proiectului și senzorul touch integrat.

Obiectivul principal este de a crea un sistem capabil să utilizeze buzzer-ul pentru a indica temperaturile măsurate în interiorul microprocesorului și să controleze un LED RGB prin intermediul unei interfețe grafice. Am extins scopul inițial, incluzând acționarea manuală a buzzerului, prin intermediul senzorului touch, în cazul în care, prin reducere la absurd, transmisia temperaturii eșuează și este nevoie de intervenția utilizatorului pentru a declanșa ”alarma”.

Proiectul își propune, de asemenea, să demonstreze eficient utilizarea diferitelor periferice integrate pe placa de dezvoltare, îmbinând hardware-ul și software-ul pentru a realiza funcționalități specifice.

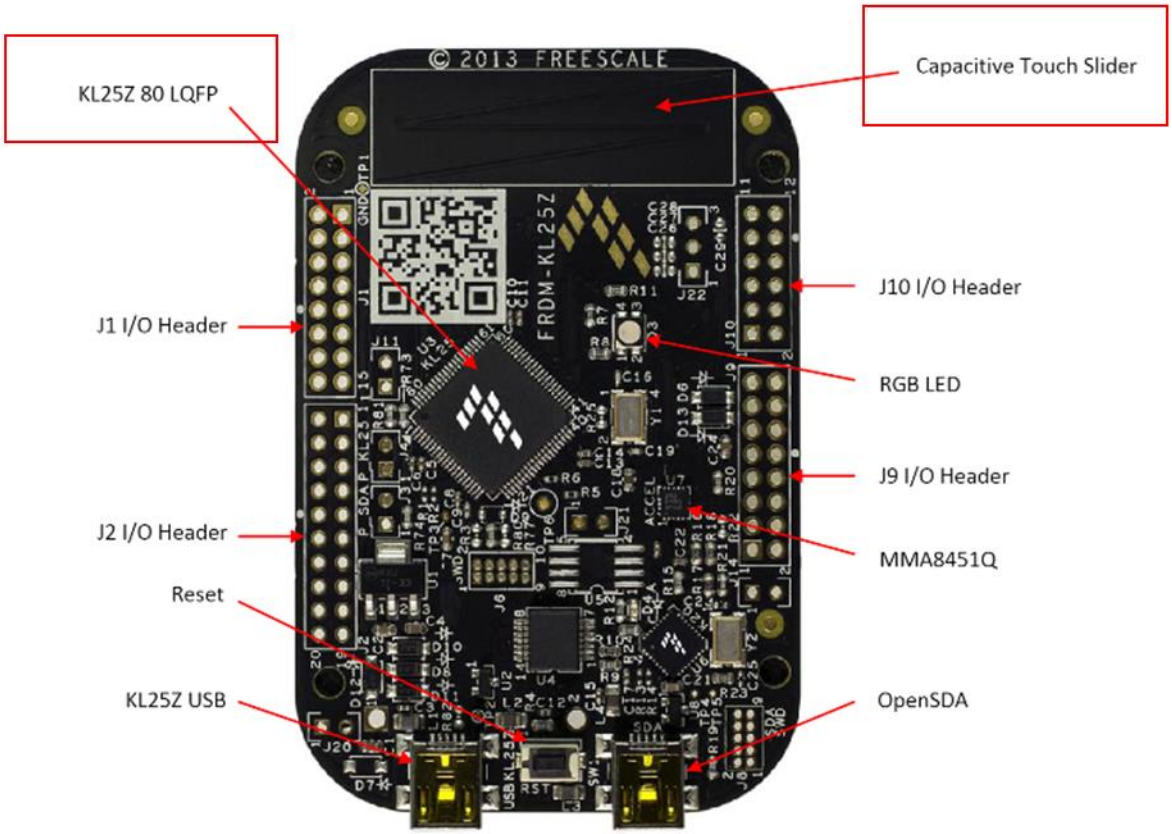
1.2 Descrierea generală a sistemului

Sistemul propus este structurat în jurul plăcii FRDM-KL25Z, integrând funcționalitățile sale native, precum senzorii de temperatură și touch. Prin intermediul acestor senzori, sistemul va putea detecta variațiile de temperatură și va folosi buzzer-ul pentru a emite semnale sonore corespunzătoare.

De asemenea, sistemul include o interfață grafică pentru vizualizarea intuitivă a datelor înregistrate de senzorul de temperatură, prin afișarea acestora în culori corespunzătoare intervalului de referință în care ne aflăm (0-20°C verde, 20-30°C galben, peste 30°C roșu). În plus, pune la dispoziție un buton dedicat acționării buzzer-ului, ca metodă suplimentară, în cazul în care cedează toate celelalte sisteme de alertă.

Proiectul îmbină tehnici de programare în C, respectiv Python cu principii de electronică, demonstrând aplicabilitatea tehnologiei microprocesoarelor în situații practice.

2. Descrierea Hardware



2.1 FRDM-KL25Z: Specificații și funcționalități

FRDM-KL25Z este o placă de dezvoltare versatilă, proiectată pentru a facilita experimentarea și dezvoltarea de aplicații încorporate. Placa este dotată cu un microcontroler ARM Cortex-M0+ care funcționează la o frecvență de până la 48 MHz, oferind un echilibru între performanță și consum de energie. Placa include 128 KB de memorie flash și 16 KB SRAM, suficient pentru dezvoltarea aplicațiilor de bază.

În ceea ce privește conectivitatea, placa dispune de multiple porturi de intrare/ieșire, incluzând GPIO, PWM, I2C, SPI și UART, oferind flexibilitate în conectarea diverselor periferice. Suportă diverse limbaje de programare, inclusiv C/C++ și Assembly, permițând dezvoltarea de aplicații complexe.

Astfel, integrând aceste caracteristici, FRDM-KL25Z este ideală pentru proiecte educaționale precum este proiectul în cauză.

2.2 Senzori integrați: temperatură și touch

2.2.1 Senzorul de temperatură

Placa FRDM-KL25Z include un senzor de temperatură integrat în modulul său ADC (Analog-Digital Converter). Senzorul are o funcție de transfer aproximativă, care relatează tensiunea măsurată de senzor la temperatura ambientală.

$$Temp = 25 - ((V_{TEMP} - V_{TEMP25}) \div m)$$

Ecuația specifică tensiunea la 25°C și o panta de temperatură (m), valorile acestora fiind disponibile în tabelul de caracteristici electrice ale ADC.

În aplicații, se citește tensiunea senzorului, se calculează și se compară cu valoarea de referință la 25°C.

Diferențele de tensiune sunt apoi folosite pentru a calcula temperatura ambientală, aplicând panta de temperatură adecvată. Această abordare permite măsurarea precisă a temperaturii, esențială pentru diverse aplicații de monitorizare și control.

2.2.2 Modul pentru detectarea atingerii

Placa FRDM-KL25Z este dotată cu un modul de intrare pentru detectarea atingerii (TSI) care oferă detectare capacitivă cu sensibilitate înaltă și robustețe îmbunătățită. Fiecare pin TSI implementează măsurători capacitive folosind scanări cu sursă de curent, încărcând și descărcând electrodul, o singură dată sau de mai multe ori. Un oscilator de referință numără timpul de scanare și stochează rezultatul într-un registru de 16 biți la finalizarea scanării. Modulul TSI poate fi declanșat periodic în modul de consum redus de energie și poate trezi CPU-ul la finalul scanării sau când rezultatul conversiei este în afara intervalului specificat de pragul TSI. Acesta oferă un modul solid pentru măsurători capacitive ce poate fi implementat în tastaturi tactile, rotative și glisoare.

Funcționalitățile TSI includ suport pentru până la 16 electrozi externi, detectarea automată a capacității electrodului în toate modurile de putere operaționale, oscilator intern pentru măsurători de înaltă precizie, trigger de scanare configurabil software sau hardware, și suport complet pentru biblioteca software de senzare tactilă Freescale (TSS).

Noi vom folosi acest senzor touch astfel încât să permitem utilizatorului să declanșeze „alarma” (buzzer-ul) în cazul în care acest lucru este necesar. Astfel,

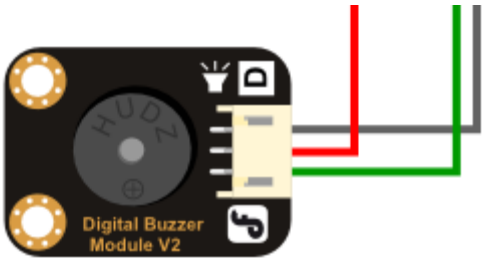
oferim o modalitate directă și eficientă utilizatorului de a interacționa cu placa de dezvoltare printr-o simplă atingere.

2.3 Buzzer Digital: Model și specificații

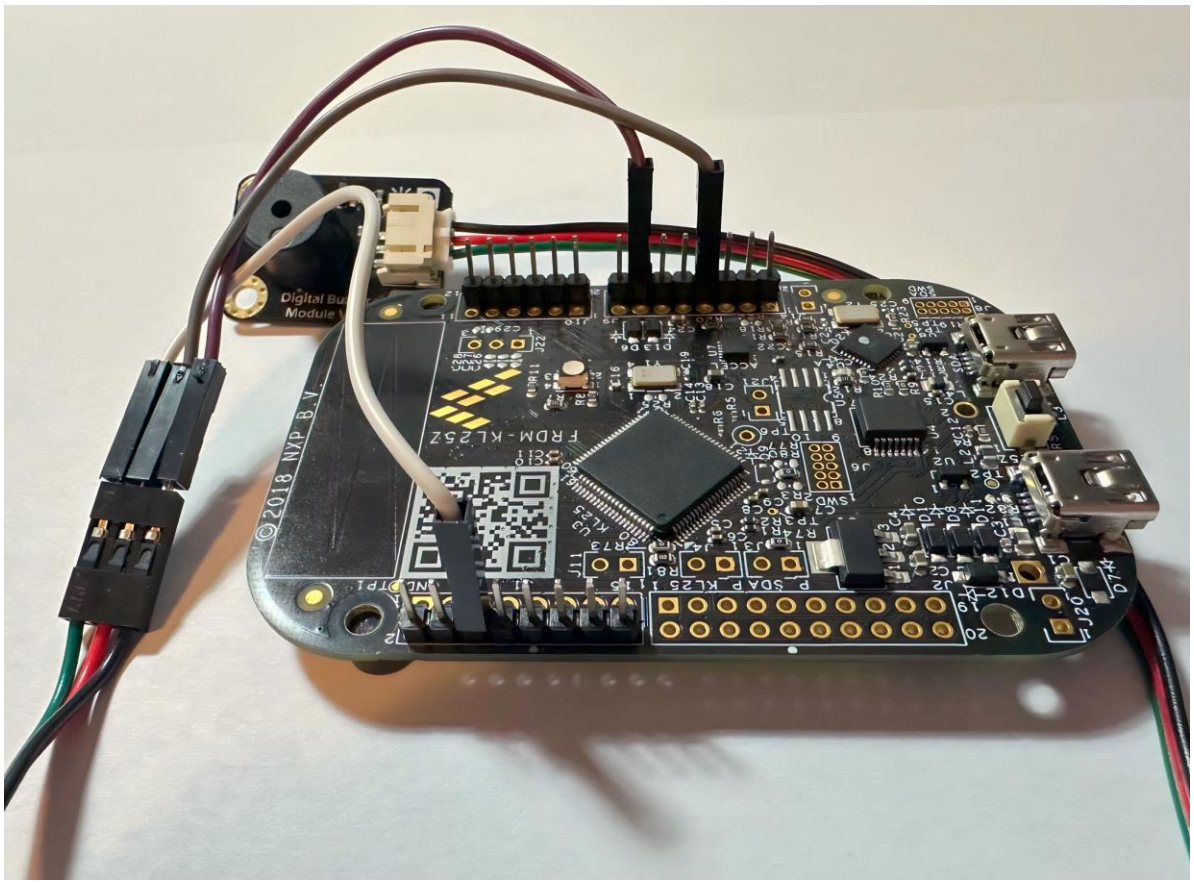
Modelul buzzer-ului digital utilizat în acest proiect este "Digital Buzzer Module v2" de la DFROBOT. Acest buzzer este proiectat pentru a emite sunete de intensitate reglabilă și poate fi controlat digital printr-un semnal PWM (Pulse Width Modulation) sau un semnal digital de înaltă/ joasă tensiune. Specificațiile tehnice includ o gamă largă de frecvențe audibile și este compatibil cu tensiunile standard de operare pentru plăcile de dezvoltare, cum ar fi 3.3V sau 5V.

Buzzer-ul este conectat la placa FRDM-KL25Z printr-unul dintre pinii săi GPIO (General Purpose Input/Output), care este configurat să genereze semnale pentru a controla buzzer-ul. Codul sursă implementat pe placa FRDM-KL25Z gestionează frecvența și durata impulsurilor, determinând astfel tonul și volumul sunetului emis de buzzer. Această metodă permite o varietate de sunete și alarme care pot fi utilizate pentru a indica diferite stări ale sistemului sau pentru a alerta utilizatorul.

Aplicațiile practice ale buzzer-ului digital sunt variate și se întind pe mai multe domenii ale tehnologiei și electronicii. De exemplu, buzzer-ele sunt adesea folosite în sisteme de alarmă pentru a oferi semnale auditive care pot fi detectate chiar și în medii zgomotoase, acesta fiind și contextul actual de utilizare.

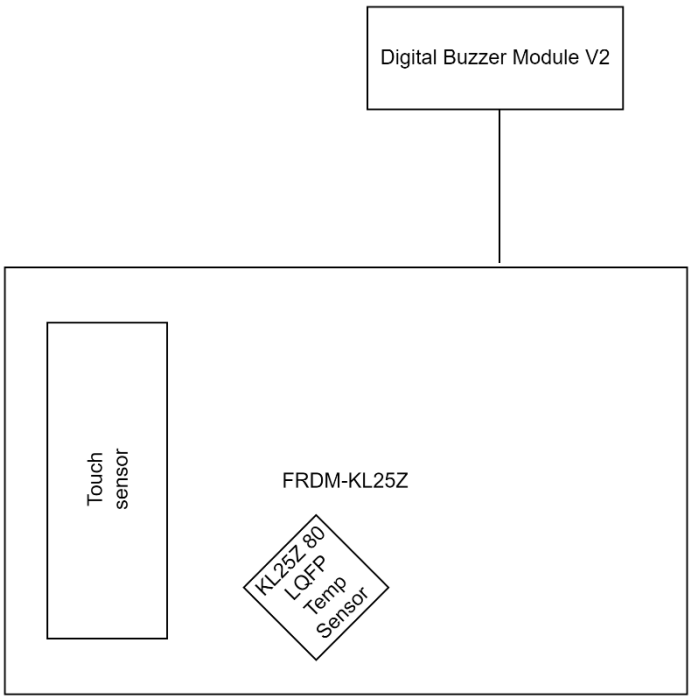


Realizarea conexiunii buzzer-ului la placa de dezvoltare implică o înțelegere a pinilor specifici și a funcțiilor. Pinul GND (Ground) este folosit pentru a completa circuitul electric, asigurând calea de întoarcere a curentului. Pinul de 3v3 (3.3 volți) furnizează tensiunea necesară alimentării buzzer-ului, în timp ce pinul PTD4 este utilizat pentru a trimite semnale de control de la microcontroler la buzzer. Este esențial să se asigure că firele sunt conectate corect și că buzzer-ul este configurat pentru a opera la nivelul de tensiune specificat. Această configurare permite nu doar alimentarea buzzer-ului, dar și modularea sunetului acestuia prin ajustarea duratei impulsurilor, ceea ce va influența tonul și volumul sunetului produs.

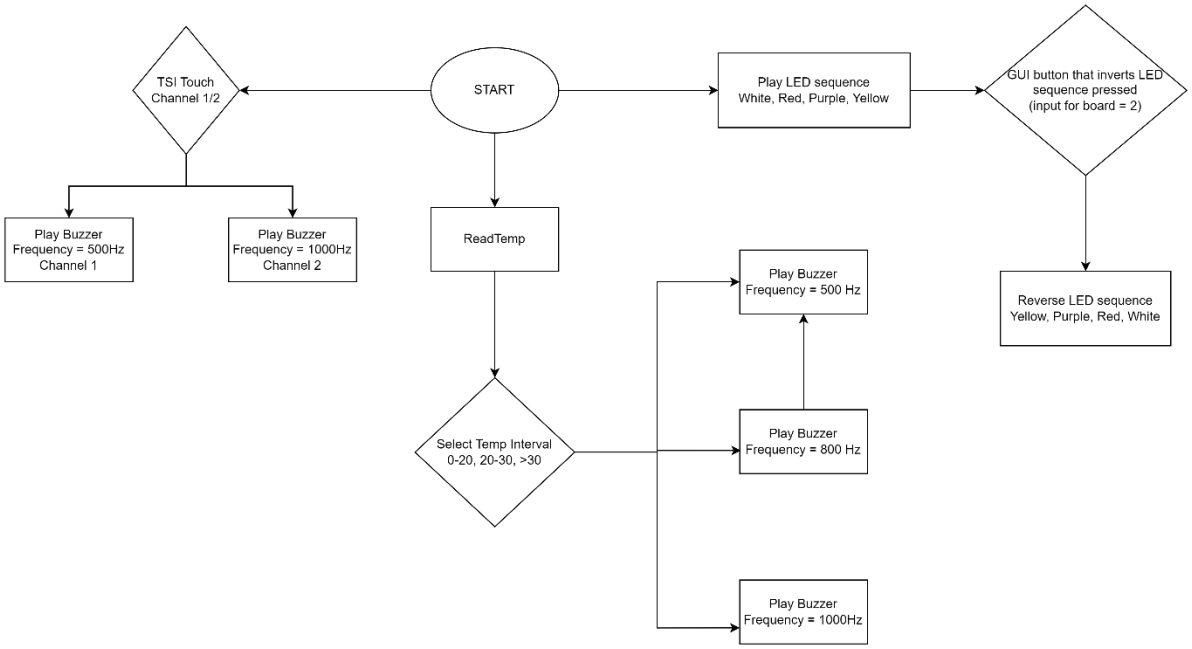


3. Configurarea Sistemului

Arhitectura sistemului conceput se caracterizează printr-o structură fundamentală, constând din două componente esențiale: placa de dezvoltare și un dispozitiv de tip buzzer digital. O ilustrație detaliată a plăcii de dezvoltare, împreună cu descrierea conexiunii sale, a fost furnizată anterior. În continuare, se prezintă diagrama bloc a sistemului proiectat.:



În vederea unei înțelegeri mai clare a proceselor aplicației, se recomandă consultarea următoarei diagrame de flux (flowchart), care include un overview asupra capabilităților proiectului dezvoltat:



3.1 Inițializarea și setările pentru FRDM-KL25Z necesare

Inițializarea și setările pentru FRDM-KL25Z sunt esențiale pentru funcționarea corectă a plăcii și perifericelor sale.

În fișierul **ClockSettings.c**, inițializarea ceasului de sistem este realizată prin configurarea Multiplier Generator Clock (MCG) pentru a folosi sursa internă de referință și pentru a seta frecvența dorită. Codul setează regiștrii MCG pentru a utiliza sursa de referință internă (MCGIRCLK) și pentru a defini factorii de multiplicare ai frecvenței (DRST_DRS) și fine trimming (DMX32).

```
1. void SystemClock_Configure(void) {
2.     MCG->C1 |= MCG_C1_CLKS(0);
3.     MCG->C1 |= MCG_C1_IREFS_MASK;
4.     MCG->C4 |= MCG_C4_DRST_DRS(1);
5.     MCG->C4 |= MCG_C4_DMX32(1);
6. }
```

Acest fragment de cod configurează ceasul de sistem al plăcii FRDM-KL25Z, ceea ce este un pas preliminar înainte de inițializarea modulului ADC și a altor periferice necesare pentru funcționarea senzorilor și a buzzerului.

Codul urmator configurează SysTick pentru a genera o întrerupere o dată la fiecare milisecundă, ceasul de sistem fiind setat la 48MHz. De asemenea, setează prioritatea întreruperii și activează timerul. Această funcție este esențială pentru gestionarea timpului în aplicații în timp real.

```
1. void SystemClockTick_Configure(void){
2.         SysTick->LOAD = (uint32_t)(48000000UL / 1000 - 1UL);
3.         NVIC_SetPriority(SysTick_IRQn, (1UL << __NVIC_PRIO_BITS) - 1UL);
4.         SysTick->VAL = 0UL;
5.         SysTick->CTRL |= (SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk |
SysTick_CTRL_ENABLE_Msk);
6. }
```

Pentru inițializarea UART, codul setează sursa ceasului pentru UART0, configurează pinii asociati PORTA pentru transmitere și recepție, și setează rata de baud necesară pentru comunicații. Întreruperile sunt activate pentru a permite transmiterea și recepția asincronă.

```
 1.  // Inițializează modulul UART0 cu o rată de baud specificată (9600)
 2. void UART0_Init(uint32_t baud_rate){
 3.     uint32_t osr; // Over Sampling Rate
 4.     uint32_t sbr; // Baud Rate Setting Value
 5.     uint8_t temp; // Temporary variable
 6.
 7.    // Selectează sursa ceasului pentru UART0 și activează ceasurile pentru UART0 și portul A
 8.     SIM->SOPT2 |= SIM_SOPT2_UART0SRC(01);
 9.     SIM->SCGC4 |= SIM_SCGC4_UART0_MASK;
10.     SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;
11.
12.     // Configurează pinii pentru UART0 Rx și Tx
13.     PORTA->PCR[1] &= ~PORT_PCR_MUX_MASK; // Curăță prima configurație a mux
14.     PORTA->PCR[1] |= PORT_PCR_MUX(2);    // Setează mux pentru UART0 Rx
15.     PORTA->PCR[2] &= ~PORT_PCR_MUX_MASK; // Curăță prima configurație a mux
16.     PORTA->PCR[2] |= PORT_PCR_MUX(2);    // Setează mux pentru UART0 Tx
17.
18.     // Dezactivează emițătorul și receptorul înainte de a configura setările
19.     UART0->C2 &= ~((UART0_C2_RE_MASK) | (UART0_C2_TE_MASK));
20.
21.       // Setează osr și sbr conform frecvenței ceasului și ratei de baud dorite
22.     osr = 15; // Over Sampling Rate
23.     sbr = 48000000UL / ((osr + 1)*baud_rate);
24.
25.     // Aplică setările de Baud Rate
26.     temp = UART0->BDH & ~(UART0_BDH_SBR(0x1F));
27.     UART0->BDH = temp | UART0_BDH_SBR(((sbr & 0x1F00)>> 8));
28.     UART0->BDL = (uint8_t)(sbr & UART_BDL_SBR_MASK);
29.
30.     // Aplică Over Sampling Rate
31.     UART0->C4 |= UART0_C4_OSR(osr);
32.
33.     // Setează setările de control la valorile lor implicite
34.     UART0->C1 = 0; // Setările de control 1
35.     UART0->C2 |= UART0_C2_TIE(0); // Dezactivează întreruperile pentru transmiterea datelor
36.     UART0->C2 |= UART0_C2_TCIE(0); //Dezactivează întreruperile pentru transmiterea completă
37.     UART0->C2 |= UART0_C2_RIE(1); // Activează întreruperile pentru recepția datelor
38.
39.     // Activează emițătorul și receptorul
40.     UART0->C2 |= ((UART_C2_RE_MASK) | (UART_C2_TE_MASK));
41.
42.     // Activează întreruperea la nivelul controlerului de întreruperi
43.     NVIC_EnableIRQ(UART0_IRQn);
44. }
```

Inițializarea ADC configurează ADC0 pentru a citi intrările analogice. Se realizează calibrarea, se setează rezoluția de conversie, se alege sursa ceasului și divizorul acestuia, și se activează întreruperile pentru conversii complete.

```c
/*  Funcția de inițializare a modulului ADC0  configurarea registrilor*/
void ADC0_Init(void)
{
    int cal_res;
    SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK;
    do
    {
        /* Calibrate ADC */
        cal_res = adc_cal();
    } while (cal_res == -1);

    /* Configure ADC */
    ADC0_SC1A |= (ADC_SC1_AIEN_MASK);  /* Interrupt enabled */
    ADC0_SC1A &= ~ADC_SC1_DIFF_MASK;   /* Single Ended ADC */
    ADC0_CFG1 = 0; /* Reset register */
    ADC0_CFG1 |= (ADC_CFG1_MODE(2)  |   /* 10 bits mode */
                        ADC_CFG1_ADICLK(1)|   /* Input Bus Clock/2 (24 Mhz) */
                        ADC_CFG1_ADIV(3)  |     /* Clock divide by 8 (3 Mhz) */
                                ADC_CFG1_ADLPC_MASK); /* Low power mode */
    ADC0_SC3  &= ~ADC_SC3_AVGE_MASK;
}
```

Inițializarea buzzer-ului configurează portul D pentru a opera cu buzzer-ul digital. Prin activarea ceasului pentru portul D și setarea pinului corespunzător buzzerului ca GPIO, se permite controlul digital al buzzerului. Pinul este setat ca ieșire, pregătindu-l pentru emiterea de semnale sonore atunci când este activat de placa FRDM-KL25Z. Această configurație este necesară pentru a permite sistemului să utilizeze buzzerul ca metodă de alertare sau feedback auditiv.

```c
void BUZZER_init(void)
    {
        SIM->SCGC5 |= SIM_SCGC5_PORTD_MASK; //Activează ceasul pentru portul D, necesar pentru operațiuni pe acest port
        PORTD->PCR[buzzer] &= ~PORT_PCR_MUX_MASK; //Configurează pinul buzzerului ca GPIO prin curățarea și setarea multiplexerului
        PORTD->PCR[buzzer] |= PORT_PCR_MUX(1);
        PTD->PDDR |= MASK(buzzer);
    }
```

```c
#define buzzer (4)
#define MASK(x) (1UL << (x))
```

Inițializarea LED-urilor setează porturile B și D pentru a controla LED-urile RGB ale plăcii. Se configurează pinii corespunzători ca GPIO și se setează ca ieșiri pentru a putea manipula LED-urile.

```c
void RRGBLED_init(void) {
    // Activează ceasul pentru porturile B și D
    SIM->SCGC5 |= SIM_SCGC5_PORTB_MASK | SIM_SCGC5_PORTD_MASK;
    // Setează pinii pentru LED-uri ca GPIO
    PORTB->PCR[RED_LED_PIN] = PORT_PCR_MUX(1);
    PORTB->PCR[GREEN_LED_PIN] = PORT_PCR_MUX(1);
    PORTD->PCR[BLUE_LED_PIN] = PORT_PCR_MUX(1);
    // Configurează pinii LED-urilor ca ieșiri
    PTB->PDDR |= (1 << RED_LED_PIN) | (1 << GREEN_LED_PIN);
    PTD->PDDR |= (1 << BLUE_LED_PIN);}
```

Inițializarea PIT configurează temporizatorul pentru a genera întreruperi la un interval setat, permițând executarea de cod la momente regulate, fiind util pentru timing și sarcini periodice.

```
 1.  void PIT_Init(void) {
 2.      // Enable the clock to the PIT module to allow access to its registers
 3.      SIM->SCGC6 |= SIM_SCGC6_PIT_MASK;
 4.
 5.      // Enable the PIT module (clear MDIS bit in PIT_MCR register)
 6.      PIT_MCR &= ~PIT_MCR_MDIS_MASK;
 7.
 8.      // Freeze the timer when debugging (set FRZ bit in PIT_MCR register)
 9.      PIT->MCR |= PIT_MCR_FRZ_MASK;
10.
11.      // Set the load value of timer 0 (how long the timer runs before triggering an
interrupt)
12.      PIT->CHANNEL[0].LDVAL = 0x9FFFFF;
13.
14.      // Enable timer 0 interrupts (set TIE bit in TCTRL register)
15.      PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TIE_MASK;
16.
17.      // Start timer 0 (set TEN bit in TCTRL register)
18.      PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TEN_MASK;
19.
20.      // Set the load value of timer 1 (how long the timer runs before triggering an
interrupt)
21.      PIT->CHANNEL[1].LDVAL = 0x63FFFF;
22.
23.      // Enable timer 1 interrupts (set TIE bit in TCTRL register)
24.      PIT->CHANNEL[1].TCTRL |= PIT_TCTRL_TIE_MASK;
25.
26.      // Start timer 1 (set TEN bit in TCTRL register)
27.      PIT->CHANNEL[1].TCTRL |= PIT_TCTRL_TEN_MASK;
28.
29.      // Clear any pending interrupts for the PIT
30.      NVIC_ClearPendingIRQ(PIT_IRQn);
31.
32.      // Set the priority for the PIT interrupts
33.      NVIC_SetPriority(PIT_IRQn, 5);
34.
35.      // Enable the PIT interrupt in the NVIC
36.      NVIC_EnableIRQ(PIT_IRQn);
37.  }
38.
```

Inițializarea TSI activează modulul de intrare pentru senzorul tactil și configurează porturile pentru a detecta atingerile. Setările de sensibilitate sunt ajustate pentru a optimiza detectarea.

```
 1. // Inițializează modulul Touch Sensing Input (TSI)
 2. void TSI_Init(void) {
 3.      SIM->SCGC5 |= SIM_SCGC5_TSI_MASK;  // Activează ceasul pentru modulul TSI
 4.      PORTB->PCR[16] = PORT_PCR_MUX(0);  // Configurează pinul 16 al portului B pentru TSI
 5.      PORTB->PCR[17] = PORT_PCR_MUX(0);  // Configurează pinul 17 al portului B pentru TSI
 6.
 7.      // Setează parametrii de configurare pentru TSI: modul, sarcina de referință,
 8.      // divizorul de tensiune, sarcina externă, prescalarea, numărul de scanări, activeazăTSI
 9.      TSI0->GENCS = TSI_GENCS_OUTRGF_MASK | TSI_GENCS_MODE(0) | TSI_GENCS_REFCHRG(4) |
10.                    TSI_GENCS_DVOLT(0) | TSI_GENCS_EXTCHRG(7) | TSI_GENCS_PS(4) |
11.                    TSI_GENCS_NSCN(31) | TSI_GENCS_TSIEN_MASK;
12.
13.      TSI0->GENCS |= TSI_GENCS_EOSF_MASK;  // Marchează sfârșitul operației de scanare
```

```
14. }
```

3.2 Detalii despre utilizarea senzorilor integrați

3.2.1 Senzorul de temperatură

Funcția **ADC0_Read** citește valoarea de la un canal specific al convertorului analog-digital (ADC). Inițializează o conversie ADC pe canalul ales, așteaptă până când conversia este completă și apoi returnează rezultatul.

```
1. uint16_t ADC0_Read(uint8_t ch){
2.     ADC0->SC1[0] = ADC_SC1_ADCH(ch);
3.     while(ADC0->SC2 & ADC_SC2_ADACT_MASK);
4.     while(!(ADC0->SC1[0] & ADC_SC1_COCO_MASK));
5.     ADC0->SC1[0] |= ADC_SC1_ADCH(31);
6.     return (uint16_t) ADC0->R[0];
7. }
```

Monitorizarea temperaturii ambientale se realizează prin măsurarea periodică cu ajutorul senzorului de temperatură integrat și conversia acestor măsurători în valori de temperatură utilizând coeficienți specifici sistemului.

Funcția **scan_temperature** gestionează acest proces de măsurare și, de asemenea, calibrează tensiunea de referință a bandgap-ului pentru a asigura acuratețea citirilor. După fiecare serie de măsurători, temperatura calculată este evaluată și, în funcție de diferite praguri de temperatură definite, starea sistemului este actualizată.

În funcție de această stare, buzzer-ul digital este activat cu diferite frecvențe și durate pentru a semnaliza vizual și auditiv schimbările de temperatură. Aceasta permite utilizatorilor să primească feedback imediat despre condițiile de mediu, facilitând acțiuni rapide în cazul unor schimbări critice ale temperaturii.

În plus, datele despre temperatura ambientală sunt transmise prin UART, permițând integrarea acestora în sisteme mai complexe de monitorizare sau control automat

```
 1. /* Funcții sau variabile statice, locale pentru acest fișier */
 2. static void monitor_temperature_and_play_buzzer(double temperature) {
 3.     static int currentStage = 0;
 4.     int newStage;
 5.
 6.     /* Determine the new stage based on the temperature */
 7.     if (temperature < STAGE_1_MAX_TEMP) { newStage = 1; }
 8.     else if (temperature < STAGE_2_MAX_TEMP) { newStage = 2; }
 9.     else { newStage = 3;}
10.
```

```
    /* Check if the stage has changed */
    if (newStage != currentStage) {
        /* Update the current stage */
        currentStage = newStage;

        /* Play buzzer based on the new stage */
        switch (newStage) {
            case 1:
                BUZZER_play(500, 500); /* Frequency 500 Hz, duration 500 ms */
                break;
            case 2:
                BUZZER_play(800, 500); /* Frequency 1000 Hz, duration 500 ms */
                break;
            case 3:
                BUZZER_play(1000, 500); /* Frequency 1500 Hz, duration 500 ms */
                break;

        }
    }
}

void scan_temperature(void) {
    /* Funcții sau variabile statice, locale pentru acest fișier */
    static uint8_t loopcntr;
        /* Funcții sau variabile statice, locale pentru acest fișier */
    static uint8_t bandgap_vltg_cal;

    loopcntr++;

    if (!bandgap_vltg_cal) {
        adc_result += ADC0_READ(BANDGAP_VOLTAGE_CHN);
        if (loopcntr >= 16) {
            adc_result_avg = adc_result >> 4;
            /* Vddcal = (VREF_FACTOR * 1000) / adc_result_avg; */
            Vddcal = 33; /* testing */
            Vtemp25 = 7173 / Vddcal;
            TempSlope = 1657 / Vddcal;

            loopcntr = 0;
            adc_result = 0;
            bandgap_vltg_cal = 1;
        }
    } else {
        adc_result += ADC0_READ(TEMPERATURE_SENSOR_CHN);
        if (loopcntr >= 16) {
            char temp_str[20];

            Vtemp = adc_result >> 4;
            temp = (25 - (((Vtemp - Vtemp25) * 100) / TempSlope));
            monitor_temperature_and_play_buzzer(temp);
            floatToString(temp, temp_str);
            UART0_Transmit_String(temp_str);
            loopcntr = 0;
            adc_result = 0;
        }
    }
    ADC0_SC1A |= ADC_SC1_ADCH(DISABLE_ADC); /* Disable module */
}

```

3.2.3 Senzorul touch

Inițializarea acestui modul de touch a fost explicată într-un subcapitol anterior, aparținând de configurarea inițială a modulelor mari utilizate în acest proiect. Vom continua cu modul în care se realizează citirea datelor și scopul acestora în proiectul de față. Funcția TSI_Read este utilizată pentru a citi date de la senzorul tactil astfel:

```
1. /* functia de citire a touch-ului pentru canallul 9 stanga suprafetei tactile*/
2. uint16_t TSI_Read1(uint8_t channel) {
3.     int16_t counter;
4.     TSI0->DATA = TSI_DATA_TSICH(channel);
5.     TSI0->DATA |= TSI_DATA_SWTS_MASK;
6.     while(!(TSI0->GENCS & TSI_GENCS_EOSF_MASK));
7.     TSI0->GENCS |= TSI_GENCS_EOSF_MASK;
8.     counter = (int16_t)(TSI0->DATA & TSI_DATA_TSICNT_MASK);
9.     if(counter > TOUCH_THRESHOLD)
10.    {
11.         BUZZER_play(500, 800);
12.    }
13.    return (uint16_t)counter;
14. }
15. /* functia de citire a touch-ului pentru canallul 10 dreapta suprafetei tactile*/
16. uint16_t TSI_Read2(uint8_t channel) {
17.    int16_t counter;
18.    TSI0->DATA = TSI_DATA_TSICH(channel);
19.    TSI0->DATA |= TSI_DATA_SWTS_MASK;
20.    while(!(TSI0->GENCS & TSI_GENCS_EOSF_MASK));
21.    TSI0->GENCS |= TSI_GENCS_EOSF_MASK;
22.    counter = (int16_t)(TSI0->DATA & TSI_DATA_TSICNT_MASK);
23.    if(counter > TOUCH_THRESHOLD)  /* intra in bucla doar daca TOUCH_THRESHOLD depaseste
valoarea 2100 care regleaza sensibilitatea suprafetei tactile  */
24.    {
25.         BUZZER_play(1000, 800);
26.    }
27.    return (uint16_t)counter;
28. }
29.
```

Această funcție demonstrează cum senzorul tactil poate fi folosit pentru a detecta atingeri și pentru a interacționa cu alte componente ale sistemului, cum ar fi buzzerul sau UART pentru notificări.

3.3 Conectarea buzzerului

Funcția BUZZER_play controlează activitatea buzzerului. În funcție de valoarea citită de la ADC, se stabilește durata pentru care buzzerul va fi activat. Buzzerul este comutat pentru a genera un sunet pe durata specificată.

```
1. /* Funcția care redă un sunet pe buzzer*/
2. void BUZZER_play(uint32_t frequency, uint32_t duration_ms)
3. {
4.     uint32_t period = 1000000 / frequency; /* Period in microseconds for half cycle */
5.     uint32_t duration = duration_ms * 1000; /* Convert duration to microseconds */
6.     uint32_t elapsed;
```

```
    for (elapsed = 0; elapsed < duration; elapsed += period)
    {
        PTD->PTOR = MASK(buzzer); /* Toggle the buzzer pin */
        delay(period / 2); /* Delay for half period */
    }
}

```

Buzzer-ul poate fi utilizat în tandem cu senzorii pentru a oferi feedback auditiv în diferite scenarii. De exemplu:

1. **Senzorul de Temperatură**: Buzzer-ul este activat când temperatura măsurată depășește un anumit prag, semnalizând astfel condiții de supraîncălzire sau alte schimbări critice de temperature. Această activare a buzzerului se realizează respectând 3 intervale de temperatură.
2. **Senzorul Touch**: În cazul senzorului de touch, buzzer-ul poate fi folosit pentru a confirma o atingere sau pentru a preveni o eroare de sistem, în care utilizatorul detectează temperaturi anormale înaintea ansamblului practic. De exemplu, un sunet scurt poate fi emis ca răspuns la o atingere, oferind feedback imediat utilizatorului.

4. Descrierea Software

4.1 Structura și organizarea codului sursă

Proiectul este structurat modular, cu fișiere sursă separate pentru fiecare componentă hardware și funcționalitate specifică. Avem Adc.c și Adc.h pentru gestionarea conversiilor analog-digital, Buzzer.c și Buzzer.h pentru operarea buzzerului, Tempsenzor.c și Tempsenzor.h pentru manipularea temperaturii , iar Touch.c și Touch.h pentru citirea intrărilor de la senzorul tactil. Pentru setările inițiale ale ceasului sistemului avem ClockSettings.c și ClockSettings.h, iar pentru controlul LED-ului RGB avem Led.c. Pit.c și Pit.h sunt folosite pentru temporizatorul programabil de intrare, în timp ce Uart.c și Uart.h gestionează comunicația serială. Fișierul principal main.c integrează toate aceste module.

Pe lângă acestea, sunt prezente și fișiere pentru interfața grafică (main_window.py și main.py). Această structură facilitează dezvoltarea și depanarea independentă a diferitelor părți ale proiectului.

4.2 Modulele utilizate (UART, GPIO, PIT, ADC)

- **UART (Universal Asynchronous Receiver-Transmitter)**: Gestionat prin **Uart.c** și **Uart.h**, modulul UART permite comunicația serială bidirecțională. Este configurat pentru a stabili parametrii de comunicație cum ar fi rata de baud și permite schimbul de date între microcontroler și alte dispozitive sau interfețe, esențial pentru debug și comenzi interactive.
- **GPIO (General-Purpose Input/Output)**: Prin intermediul **Led.c** și a altor fișiere, pinii GPIO sunt utilizați pentru controlul dispozitivelor externe, cum ar fi LED-urile și buzzerul. Acești pini sunt configurați ca intrări sau ieșiri în funcție de necesitățile circuitului și sunt esențiali pentru interfața hardware a sistemului.
- **PIT (Programmable Interval Timer)**: Cu **Pit.c** și **Pit.h**, PIT oferă funcționalitate de temporizare pentru a genera întreruperi la intervale regulate, facilitând astfel sarcini periodice și măsurarea timpului, vital pentru operațiunile în timp real.
- **ADC (Analog-to-Digital Converter)**: **Adc.c** și **Adc.h** administrează conversia semnalelor analogice de la senzorii de temperatură și touch în semnale digitale. Acest proces este crucial pentru monitorizarea și reacția adecvată a sistemului la datele de intrare analogice.

Fișierele **Tempsenzor.c** și **Tempsenzor.h** sunt utilizate pentru controlul temperaturii, integrând componentele definite în celelalte componente descrise.

4.3 Detalii despre implementarea interfetei

Aplicația dezvoltată în Python folosind biblioteca PySide6 implementează o interfață grafică pentru monitorizarea și controlul temperaturii, cu interacțiunea utilizatorului prin intermediul unei conexiuni seriale. Fereastra principală este intitulată cu anul academic al proiectului și include elemente vizuale care permit vizualizarea datelor de temperatură în timp real și controlul unui buzzer.

Funcționalitatea serială inițializează comunicația cu un port serial specific, iar un temporizator (QTimer) actualizează periodic informațiile de temperatură. Citirile sunt preluate de la un senzor prin portul serial și sunt afișate grafic, cu segmente colorate în funcție de intervalul de temperatură: verde pentru valori scăzute, galben pentru medii și roșu pentru temperaturi înalte.

```
    def init_serial(self):
        self.serial_port = serial.Serial(port='COM8', baudrate=9600, timeout=1)

    def get_color_for_temperature(self, temp):
        if temp < 20:
            return 'g'  # Verde
        elif 20 <= temp < 30:
            return 'y'  # Galben
        else:
            return 'r'  # Roșu

    def update_temperature(self):
        if self.serial_port.in_waiting:
            data = self.serial_port.readline().decode('utf8').strip()

            if data.startswith('T:'):
                temp_str = data[2:].strip()
                try:
                    temp = float(temp_str)
                    self.temperature.append(temp)
                    self.hour.append(self.hour[-1] + 1)

                    color = self.get_color_for_temperature(temp)
                    # Crearea unui nou segment și adăugarea lui la grafic
                    new_segment = self.plot_widget.plot(
                        self.hour[-2:], self.temperature[-2:], pen=mkPen(color, width=2))
                    self.line_segments.append(new_segment)

                    # O logică pentru a limita numărul de segmente stocate (opțional)
                    if len(self.line_segments) > 50:  # Exemplu: limitează la 50 de segmente
                        old_segment = self.line_segments.pop(0)
                        self.plot_widget.removeItem(old_segment)

                    self.debug_text_edit.append(f"Temperatura: {temp}°C")
                except ValueError as e:
                    self.debug_text_edit.append(f"Nu pot procesa datele.")
            else:
                self.debug_text_edit.append(f"{data}")
```

Interacțiunea utilizatorului este facilitată prin butoane care transmit comenzi către un microcontroler, oferind funcționalități precum 'Play Buzzer' și 'Stop Buzzer'. Un câmp de introducere permite trimiterea de date personalizate. Mesajele de debug și erorile sunt afișate într-o zonă de text dedicată, facilitând depanarea și monitorizarea stării aplicației.

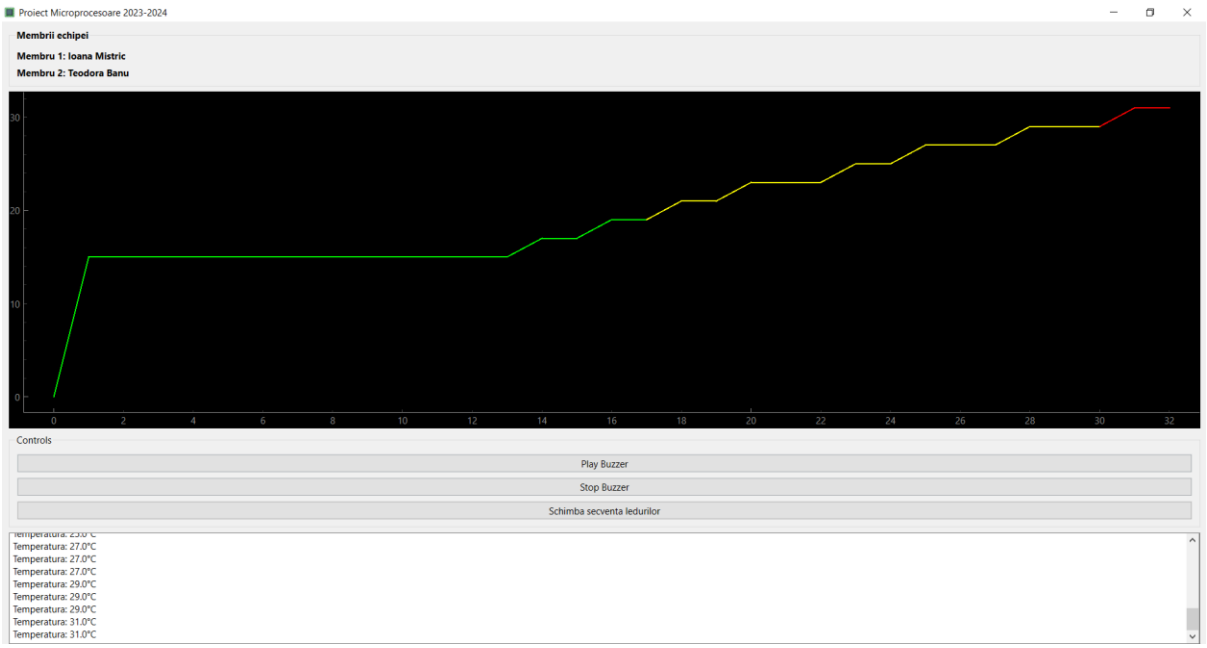
```
    def send_input(self):
        data = self.input_field.text()
        self.input_field.clear()
        self.send_serial_data(data)

    def send_serial_data(self, data):
        try:
            if self.serial_port.is_open:
                self.serial_port.write(data.encode())
        except Exception as e:
            self.debug_text_edit.append(f"Error sending data: {e}")

```

La închiderea aplicației, o rutină de curățare se asigură că portul serial este închis corespunzător, prevenind astfel posibilele probleme la reconectare sau la utilizările ulterioare. Această abordare asigură o încheiere elegantă a sesiunii de lucru și menține integritatea comunicației cu dispozitivele externe.

```
def closeEvent(self, event):
        if self.serial_port.is_open:
            self.serial_port.close()
        super().closeEvent(event)
```

5. Testare și rezultate

În urma rularii interfeței grafice putem observa cum este conturat graficul temperaturii înregistrată de senzorul integrat pe placa de dezvoltare.



De asemenea, fiecare temperatură înregistrată este afișată în câmpul de text pentru o analiză simplificată a graficului.

Putem observa și cele 3 butoane discutate anterior: cele doua legate de controlul buzzer-ului și cel pentru inversarea secvenței ledurilor. Aceste controale adăugate în interfața grafică facilitează interacțiunea utilizatorului cu proiectul în sine, adăugând funcționalități imperios necesare în cazul unei eventuale defecțiuni. Acestea vor fi detaliate mai amănunțit în videoclipul de prezentare, întrucât necesită înțelegere vizuală și auditivă.

Testarea interfeței grafice a evidențiat eficacitatea vizualizării datelor de temperatură, oferind utilizatorilor o reprezentare grafică intuitivă și imediată a informațiilor provenite de la senzor. A fost posibilă nu doar observarea dinamicii temperaturii în timp real dar și o analiză detaliată prin înregistrările afișate în zona de text. Această abordare facilitează nu numai monitorizarea ambientală dar și interacțiunea directă cu sistemul, prin intermediul controalelor dedicate pentru buzzer și LED-uri. Aceste caracteristici vor fi demonstrate în detaliu într-un videoclip de prezentare pentru a asigura o înțelegere completă atât vizuală, cât și auditivă.

6. Dificultăți întâmpinate

6.1 Configurarea ADC

Noi am rescris codul de la ADC aproape complet, întrucat pe cel din laborator nu am reușit să îl integrăm în proiecul nostru. Codul din laborator nu furniza corect și în timp real date legate de temperatură. Nu am reușit să identificăm cauza, astfel a trebuit sa studiem mai în detaliu aceeasta parte.

ADC0_Init() din laborator:

```
1. void ADC0_Init() {
2.
3.     // Activarea semnalului de ceas pentru modulul periferic ADC
4.     SIM->SCGC6 |= SIM_SCGC6_ADC0_MASK;
5.
6.     // Functia de calibrare
7.     ADC0_Calibrate();
8.
9.     ADC0->CFG1 = 0x00;
10.
11.    // Selectarea modului de conversie pe 16 biti single-ended --> MODE
12.    // Selectarea sursei de ceas pentru generarea ceasului intern --> ADICLK
13.    // Selectarea ratei de divizare folosit de periferic pentru generarea ceasului intern --
> ADIV
14.    // Set ADC clock frequency fADCK less than or equal to 4 MHz (PG. 494)
15.    ADC0->CFG1 |= ADC_CFG1_MODE(3) |
16.                              ADC_CFG1_ADICLK(0) |
17.                              ADC_CFG1_ADIV(2);
18.
19.    // DIFF = 0 --> Conversii single-ended (PG. 464)
20.    ADC0->SC1[0] = 0x00;
21.    ADC0->SC3 = 0x00;
22.
23.    // Selectarea modului de conversii continue,
24.    // pentru a-l putea folosi in tandem cu mecanismul de intreruperi
25.    ADC0->SC3 |= ADC_SC3_ADCO_MASK;
26.
27.     // Activarea subsistemului de conversie prin aproximari succesive pe un anumit canal
(PG.464)
28.    ADC0->SC1[0] |= ADC_SC1_ADCH(ADC_CHANNEL);
29.    // Enables conversion complete interrupts
30.    ADC0->SC1[0] |= ADC_SC1_AIEN_MASK;
31.
32.    NVIC_ClearPendingIRQ(ADC0_IRQn);
33.    NVIC_EnableIRQ(ADC0_IRQn);
34. }
```

Iar acesta este init facut de noi:

```
1. void ADC0_Init(void)
2. {
3.     int cal_res;
4.     SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK;
5.     do
6.     {
7.         /* Calibrate ADC */
8.         cal_res = adc_cal();
9.     } while (cal_res == -1);
```

```
10.
11.     /* Configure ADC */
12.     ADC0_SC1A |= (ADC_SC1_AIEN_MASK);  /* Interrupt enabled */
13.     ADC0_SC1A &= ~ADC_SC1_DIFF_MASK;   /* Single Ended ADC */
14.     ADC0_CFG1 = 0; /* Reset register */
15.     ADC0_CFG1 |= (ADC_CFG1_MODE(2)  |   /* 10 bits mode */
16.                           ADC_CFG1_ADICLK(1)|   /* Input Bus Clock/2 (24 Mhz) */
17.                           ADC_CFG1_ADIV(3)  |     /* Clock divide by 8 (3 Mhz) */
18.                                   ADC_CFG1_ADLPC_MASK); /* Low power mode */
19.     ADC0_SC3  &= ~ADC_SC3_AVGE_MASK;
20. }
```

Noi am schimbat partea de calibrare, astfel încât aceasta să se realizeze până se stabilizează complet, adică pana când calibrarea este reușită. Am schimbat pointerii către structuri în inițializări de regiștrii. Am schimbat de asemenea și modul de conversie prin a utiliza doar 10 biți, iar conversia să fie făcută in modul de consum redus de energie. De asemenea, noi nu am folosit conversii continue conform acestui câmp din documentație. Acest lucru, împreună cu conversia pe 10 biți și modul consum redus de energie au făcut diferența.

Pe partea de configurare temperatură a fost destul de complicat să înțelegem formulele din pdf-ul „In-built Temperature Sensor AN3031.pdf” din documentația de la MCULabs așa că a trebuit sa ne inspirăm și de pe internet unde am găsit un repository pe această temă și anume: https://github.com/mahphalke/learningmicro/blob/master/Embedded%20Tutorials/KL25Z_TemperatureSensor.rar

Repository-ul din care face parte este următorul: https://github.com/mahphalke/learningmicro/tree/master/Embedded%20Tutorials

Acest proiect a fost foarte de ajutor în configurarea senzorului de temperatură și a modulului ADC, de aici am preluat parte din modificările esențiale, pe care a trebuit sa le verificăm în conformitate cu documentația pentru a înțelege de ce se fac aceste configurări.

6.2 Afișarea graficului pe culori

Am întâmpinat probleme si la interfața grafică, la fișarea colorată a graficului, întrucat codul încercat inițial bloca interfața grafică. Pentru a afișa colorat, trebuie să facem distincția între date, astfel trebuie să reținem anumite segmente de date și să le ștergem pe parcurs. Această ștergere a rezolvat problema.

7. Concluzii

În concluzie, acest proiect a demonstrat capacitatea tehnologiei actuale de a crea sisteme complexe și interconectate. Interfața grafică a utilizatorului, împreună cu monitorizarea și controlul în timp real al temperaturii, ilustrează cum tehnologia poate îmbunătăți interacțiunea umană cu dispozitivele electronice. Prin implementarea comunicației seriale și a algoritmilor de prelucrare a datelor, proiectul subliniază importanța integrării software-ului cu hardware-ul.

Succesul acestui proiect stă în aplicabilitatea sa în lumea reală, deschizând calea spre noi cercetări și dezvoltări în domeniul microprocesoarelor. Reflectând asupra proiectului, este clar că progresul tehnologic ne oferă uneltele necesare pentru a construi soluții inovatoare și eficiente care să răspundă nevoilor în continuă schimbare ale societății.

Proiectul și-a atins scopul de a integra cunoștințele teoretice cu practica inginerească, punând bazele pentru progrese viitoare în domeniul microprocesoarelor. Acesta a arătat cum un sistem înglobat poate fi nu doar funcțional, dar și interactiv, facilitând comunicarea între om și mașină. În plus, aplicabilitatea sa în contexte reale validează relevanța și potențialul său educativ și practic. În final, proiectul reprezintă un exemplu concret al modului în care tehnologia poate fi folosită pentru a răspunde nevoilor specifice și dinamice ale utilizatorilor.

8. Resurse

În acest capitol sunt incluse linkuri externe care oferă informații suplimentare despre proiect, inclusiv codul sursă și un videoclip demonstrativ.

- Codul Sursă: https://github.com/echipa17/Proiect_Microprocesoare.git
- Videoclip Demonstrativ: https://youtu.be/-FgpAtF_460