```
# Instructions:
# 1. Type your response to each question below the question's heading;
#    i.e., write question 1's response below "** Question 1:"
# 2. Do not modify this template;
#    this may result in your submission not being graded and a score of 0.
# 3. Do not change the file type of this file; i.e., do not change to rich text,
#    a Word document, PDF, etc.  You will submit this text file, as a .txt file,
#    on Canvas for this homework assignment.

** Question 1:

#include "stm32f0xx.h"
#include "adc.h"

void ADC_Init(void) {
    // Enable GPIOA for PA0 (analog input)
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
    GPIOA->MODER |= GPIO_MODER_MODER0;
    GPIOA->PUPDR &= ~GPIO_PUPDR_PUPDR0;

    // Enable ADC clock
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;

    // Enable HSI14 (ADC clock source)
    RCC->CR2 |= RCC_CR2_HSI14ON;
    while (!(RCC->CR2 & RCC_CR2_HSI14RDY));

    // Ensure ADC disabled before config
    if (ADC1->CR & ADC_CR_ADEN) {
        ADC1->CR |= ADC_CR_ADDIS;
    }
    while (ADC1->CR & ADC_CR_ADEN);

    // Calibrate ADC
    ADC1->CR |= ADC_CR_ADCAL;
    while (ADC1->CR & ADC_CR_ADCAL);

    // Configure ADC channel and sampling
    ADC1->CHSELR = ADC_CHSELR_CHSEL0;
    ADC1->CFGR1  = 0x00;
    ADC1->SMPR |= ADC_SMPR_SMP; // 239.5 cycles for stable conversion

    // Enable ADC
    ADC1->CR |= ADC_CR_ADEN;
    while (!(ADC1->ISR & ADC_ISR_ADRDY));
}

uint16_t ADC_In(void) {
    ADC1->CR |= ADC_CR_ADSTART;
    while (!(ADC1->ISR & ADC_ISR_EOC));
    return (uint16_t)ADC1->DR;
}

** Question 2:

/**
 ******************************************************************************
 * @file           : main.c
 * @brief          : Lab 3 – Phase 2 (Position Acquisition System)
```

```
 * @author         : Mohamed Khalifa – EGEC 450
 * @description    : Samples the slide potentiometer using the ADC every 100 ms
 *                   and displays the position on the LCD in centimeters.
 ******************************************************************************
 */

#include "stm32f0xx.h"
#include "lcd.h"
#include "adc.h"
#include <stdint.h>

volatile uint16_t ADCMail = 0;
volatile uint8_t  ADCFlag = 0;

static void delay_ms(uint32_t n) {
    for (uint32_t i = 0; i < n * 800; i++) {
        __NOP();
    }
}

static void LED_Init(void) {
    RCC->AHBENR |= RCC_AHBENR_GPIOCEN;
    GPIOC->MODER &= ~(3U << (8 * 2));
    GPIOC->MODER |=  (1U << (8 * 2));
}

static void SysTick_Init_10Hz(void) {
    SysTick->LOAD = (48000000 / 10) - 1;  // 100 ms period
    SysTick->VAL  = 0;
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk   |
                    SysTick_CTRL_ENABLE_Msk;
}

static uint32_t Position_From_Sample(uint16_t sample) {
    return ((uint32_t)sample * 2000U) / 4095U; // scale to 0–2 cm
}

void SysTick_Task(void) {
    GPIOC->ODR ^= (1 << 8);
    uint16_t sample = ADC_In();
    ADCMail = sample;
    ADCFlag = 1;
    GPIOC->ODR ^= (1 << 8);
}

int main(void) {
    LED_Init();
    LCD_Init();
    ADC_Init();
    SysTick_Init_10Hz();

    LCD_Command(0x80);
    LCD_OutString("Pos (cm):");

    while (1) {
        if (ADCFlag) {
            uint16_t sample = ADCMail;
            ADCFlag = 0;
```

```
            uint32_t pos_thousandths = Position_From_Sample(sample);
            LCD_Command(0xC0);
            LCD_OutUFix(pos_thousandths);
        }
    }
}
```

** Question 3:

Modular design is a programming approach that divides a large system into smaller, independent modules that each perform a specific function [1]. This approach simplifies debugging, improves readability, and enables code reuse.

In this lab, the system was divided into clear modules:
- adc.c handled analog sampling,
- lcd.c handled display output,
- main.c coordinated timing and communication using the mailbox.

This separation made the project easier to debug — for example, ADC linearity issues were isolated and fixed without affecting the LCD logic. Modular design also allowed reusing the same ADC driver for future labs or projects.

** Question 4:

At higher sampling frequencies, the current mailbox system would not work properly because it stores only one value at a time. If a new ADC sample arrives before the main loop clears the ADCFlag, the new data overwrites the old one, causing data loss [2].

This design is acceptable for low-rate sampling (e.g., 10 Hz) but not for high-speed acquisition. For faster sampling, a data buffering method is needed to retain multiple samples between interrupts.

** Question 5:

An effective alternative is a circular buffer (ring buffer) or DMA (Direct Memory Access) approach [2][3].

A circular buffer allows multiple ADC samples to be stored in a queue-like structure. The interrupt writes samples into the buffer, and the main program reads them asynchronously without overwriting data.
If DMA is used, the hardware automatically transfers ADC results into memory without CPU intervention, freeing the processor for other tasks.

To implement a circular buffer, global arrays and head/tail pointers would replace the mailbox variables. Each SysTick_Task() ISR would push new data into the buffer, and the main loop would pop and process it.
DMA would instead require enabling ADC1 DMA mode (ADC_CFGR1_DMAEN) and configuring the DMA controller to auto-fill a memory array.

References:

[1] Lecture 17: Sampling and ADC – EGEC 450, California State University, Fullerton, Fall 2025.
[2] Lecture 18: Data Acquisition and Fixed-Point Representation, California State University, Fullerton, Fall 2025.

[3] STMicroelectronics, STM32F0 Reference Manual RM0091, Rev. 16, 2023.
[4] OpenAI ChatGPT, Assistance with ADC driver calibration and linearity debugging for STM32F0, conversation with Mohamed Khalifa, Nov 2025.