GitHub Link

https://github.com/LuthoYRN/MNGLUT008_CBXLIS001_EEE3096S/blob/main/Prac3/main.c

Description of Implementation

In this practical, we further enhanced the initial STM32 firmware by implementing a range of functionalities: SPI-based EEPROM communication, generation of PWM signals, ADC polling, and LCD output.

1. SPI EEPROM Communication: We implemented the functionality to read and write to an external EEPROM using the SPI interface. A predefined array of binary data was written to specific memory addresses in EEPROM using the **write_to_address** function. Additionally, the **read_from_address** function was used to verify the correctness of the data stored in EEPROM. We compared the read values against the original data, given a mismatch, an error message "SPI ERROR" is displayed on the LCD.

2. PWM Signal Generation and Control: We added PWM control by using the ADC input from a potentiometer. The ADC values were polled periodically, and the result was converted into a PWM duty cycle through the **ADCtoCCR** function. The PWM signal controlled an LED's brightness by adjusting the duty cycle dynamically based on the potentiometer's position. The PWM implementation uses TIM3, Channel 3 and the CCR value is updated in each loop iteration to reflect real time changes in the potentiometer's input.

3. LCD Output: The **writeLCD** function was added to display the relevant information on a connected LCD module. This included the value read from EEPROM memory or an error message if the SPI read was incorrect. The LCD updated every time the EEPROM value was checked, with the display showing both the correct value and the error message.

4. Button Control and Frequency Adjustment: A button interrupt was implemented to toggle the LED blinking frequency between 2 Hz and 1 Hz. We dynamically adjusted this frequency using an EXTI interrupt and debouncing logic to avoid accidental multiple presses. The button-controlled frequency update altered the LED toggling, providing visual feedback based on the selected interval.

5. Timers for periodic tasks: We used TIM6 to handle the periodic toggling of LED7 based on the button-controlled frequency. TIM16 was also used to periodically check the EEPROM values, update the LCD with the current memory content, and ensure data integrity. These timers were configured to run in interrupt mode, ensuring that tasks such as EEPROM checking and LED blinking occurred without blocking other tasks.

Appendix

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2023 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include "stm32f0xx.h"
#include <lcd_stm32f0.c>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

// Definitions for SPI usage
#define MEM_SIZE 8192 // bytes
#define WREN 0b00000110 // enable writing
#define WRDI 0b00000100 // disable writing
#define RDSR 0b00000101 // read status register
#define WRSR 0b00000001 // write status register
#define READ 0b00000011
#define WRITE 0b00000010
/* USER CODE END PD */
/*our declared variables*/
uint32_t period = 500;    // Initial frequency period (500 ms, 2 Hz)
uint32_t previoustime = 0;
uint32_t adc_value=0;
static uint8_t binaryArray[6] = {
    0b10101010,  // 170 in decimal
    0b01010101,  // 85 in decimal
    0b11001100,  // 204 in decimal
    0b00110011,  // 51 in decimal
    0b11110000,  // 240 in decimal
    0b00001111   // 15 in decimal
};
uint16_t currentAddress = 0;
/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc;
```

```c

TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim6;
TIM_HandleTypeDef htim16;

/* USER CODE BEGIN PV */

// TODO: Define input variables


/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC_Init(void);
static void MX_TIM3_Init(void);
static void MX_TIM16_Init(void);
static void MX_TIM6_Init(void);
/* USER CODE BEGIN PFP */
void EXTI0_1_IRQHandler(void);
void TIM16_IRQHandler(void);
void writeLCD(char *char_in);

// ADC functions
uint32_t pollADC(void);
uint32_t ADCtoCCR(uint32_t adc_val);

// SPI functions
static void init_spi(void);
static void write_to_address(uint16_t address, uint8_t data);
static uint8_t read_from_address(uint16_t address);
static void spi_delay(uint32_t delay_in_us);
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{

  /* USER CODE BEGIN 1 */
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */
  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */
  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  init_spi();
  MX_GPIO_Init();
```

```
  MX_ADC_Init();
  MX_TIM3_Init();
  MX_TIM16_Init();
  MX_TIM6_Init();
  /* USER CODE BEGIN 2 */

  // Initialise LCD
  init_LCD();

  // Start timers
  HAL_TIM_Base_Start_IT(&htim6);
  HAL_TIM_Base_Start_IT(&htim16);

  // PWM setup
  uint32_t CCR = 0;
  HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3); // Start PWM on TIM3 Channel 3

  // TODO: Write all bytes to EEPROM using "write_to_address"
  for(int i = 0;i < 6; i++){
        write_to_address(i,binaryArray[i]);
  }
  /* USER CODE END 2 */
  //M16_IRQHandler();
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

      // TODO: Poll ADC

   adc_value = pollADC(); // Read ADC value from potentiometer
      // TODO: Get CRR
    CCR = ADCtoCCR(adc_value); // Convert ADC value to CCR value


  // Update PWM value
      __HAL_TIM_SetCompare(&htim3, TIM_CHANNEL_3, CCR);
  // Wait for delay ms
    HAL_Delay (period);

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  LL_FLASH_SetLatency(LL_FLASH_LATENCY_0);
  while(LL_FLASH_GetLatency() != LL_FLASH_LATENCY_0)
  {
  }
  LL_RCC_HSI_Enable();

   /* Wait till HSI is ready */
  while(LL_RCC_HSI_IsReady() != 1)
  {

  }
  LL_RCC_HSI_SetCalibTrimming(16);
  LL_RCC_HSI14_Enable();

```

```c
    /* Wait till HSI14 is ready */
  while(LL_RCC_HSI14_IsReady() != 1)
  {

  }
  LL_RCC_HSI14_SetCalibTrimming(16);
  LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);
  LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_1);
  LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_HSI);

    /* Wait till System clock is ready */
  while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_HSI)
  {

  }
  LL_SetSystemCoreClock(8000000);

    /* Update the time base */
  if (HAL_InitTick (TICK_INT_PRIORITY) != HAL_OK)
  {
    Error_Handler();
  }
  LL_RCC_HSI14_EnableADCControl();
}

/**
  * @brief ADC Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC_Init(void)
{

  /* USER CODE BEGIN ADC_Init 0 */
  /* USER CODE END ADC_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC_Init 1 */

  /* USER CODE END ADC_Init 1 */

  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and
number of conversion)
  */
  hadc.Instance = ADC1;
  hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc.Init.Resolution = ADC_RESOLUTION_12B;
  hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
  hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc.Init.LowPowerAutoWait = DISABLE;
  hadc.Init.LowPowerAutoPowerOff = DISABLE;
  hadc.Init.ContinuousConvMode = DISABLE;
  hadc.Init.DiscontinuousConvMode = DISABLE;
  hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc.Init.DMAContinuousRequests = DISABLE;
  hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  if (HAL_ADC_Init(&hadc) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure for the selected ADC regular channel to be converted.
  */
  sConfig.Channel = ADC_CHANNEL_6;
```

```c
  sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
  sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
  if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC_Init 2 */
  ADC1->CR |= ADC_CR_ADCAL;
  while(ADC1->CR & ADC_CR_ADCAL);                    // Calibrate the ADC
  ADC1->CR |= (1 << 0);                                    // Enable ADC
  while((ADC1->ISR & (1 << 0)) == 0);                // Wait for ADC ready
  /* USER CODE END ADC_Init 2 */

}

/**
  * @brief TIM3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM3_Init(void)
{

  /* USER CODE BEGIN TIM3_Init 0 */

  /* USER CODE END TIM3_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};

  /* USER CODE BEGIN TIM3_Init 1 */

  /* USER CODE END TIM3_Init 1 */
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 0;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 47999;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
  {
    Error_Handler();
  }
```

```
  /* USER CODE BEGIN TIM3_Init 2 */

  /* USER CODE END TIM3_Init 2 */
  HAL_TIM_MspPostInit(&htim3);

}

/**
  * @brief TIM6 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM6_Init(void)
{

  /* USER CODE BEGIN TIM6_Init 0 */

  /* USER CODE END TIM6_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM6_Init 1 */

  /* USER CODE END TIM6_Init 1 */
  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 8000-1;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 500-1;
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM6_Init 2 */
  NVIC_EnableIRQ(TIM6_IRQn);
  /* USER CODE END TIM6_Init 2 */

}

/**
  * @brief TIM16 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM16_Init(void)
{

  /* USER CODE BEGIN TIM16_Init 0 */

  /* USER CODE END TIM16_Init 0 */

  /* USER CODE BEGIN TIM16_Init 1 */

  /* USER CODE END TIM16_Init 1 */
  htim16.Instance = TIM16;
  htim16.Init.Prescaler = 8000-1;
  htim16.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim16.Init.Period = 1000-1;
  htim16.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim16.Init.RepetitionCounter = 0;
  htim16.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
```

```
  if (HAL_TIM_Base_Init(&htim16) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM16_Init 2 */
  NVIC_EnableIRQ(TIM16_IRQn);
  /* USER CODE END TIM16_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  LL_EXTI_InitTypeDef EXTI_InitStruct = {0};
  LL_GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOF);
  LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
  LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);

  /**/
  LL_GPIO_ResetOutputPin(LED7_GPIO_Port, LED7_Pin);

  /**/
  LL_SYSCFG_SetEXTISource(LL_SYSCFG_EXTI_PORTA, LL_SYSCFG_EXTI_LINE0);

  /**/
  LL_GPIO_SetPinPull(Button0_GPIO_Port, Button0_Pin, LL_GPIO_PULL_UP);

  /**/
  LL_GPIO_SetPinMode(Button0_GPIO_Port, Button0_Pin, LL_GPIO_MODE_INPUT);

  /**/
  EXTI_InitStruct.Line_0_31 = LL_EXTI_LINE_0;
  EXTI_InitStruct.LineCommand = ENABLE;
  EXTI_InitStruct.Mode = LL_EXTI_MODE_IT;
  EXTI_InitStruct.Trigger = LL_EXTI_TRIGGER_RISING;
  LL_EXTI_Init(&EXTI_InitStruct);

  /**/
  GPIO_InitStruct.Pin = LED7_Pin;
  GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
  GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_LOW;
  GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
  GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
  LL_GPIO_Init(LED7_GPIO_Port, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
  HAL_NVIC_SetPriority(EXTI0_1_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI0_1_IRQn);
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */
void EXTI0_1_IRQHandler(void)
{
// TODO: Add code to switch LED7 delay frequency
      uint32_t currentTime = HAL_GetTick();
       // Check if button is pressed and debounce time(100ms)  has passed
      if (LL_GPIO_IsInputPinSet(GPIOA, LL_GPIO_PIN_0) && (currentTime -
```

```
      previoustime > 500))
      {
             // Toggle between 500 ms (2 Hz) and 1000 ms (1 Hz) period
      if( period== 1000-1){
      period= 500-1;
      }
              // Update the previous time for debounce
      else{
      period =1000-1;
      }
      previoustime = currentTime;
      __HAL_TIM_SET_AUTORELOAD(&htim6,period);
 }
HAL_GPIO_EXTI_IRQHandler(Button0_Pin); // Clear interrupt flags
}

void TIM6_IRQHandler(void)
{
      // Acknowledge interrupt
      HAL_TIM_IRQHandler(&htim6);

      // Toggle LED7
      HAL_GPIO_TogglePin(GPIOB, LED7_Pin);
}

void TIM16_IRQHandler(void)
{
      // Acknowledge interrupt
      HAL_TIM_IRQHandler(&htim16);

      // TODO: Initialise a string to output second line on LCD


      // TODO: Change LED pattern; output 0x01 if the read SPI data is incorrect
      // Read the value from EEPROM at the current address

    uint8_t eepromValue = read_from_address(currentAddress);

    // Check if the read value matches the expected value from binaryArray
    if (eepromValue == binaryArray[currentAddress])
    {
        // Format and print the correct value to LCD
        char lcdLine[16];
        //f(lcdLine, sizeof(lcdLine), "EEPROM byte:\n%d", eepromValue);
        sprintf(lcdLine, "%d", eepromValue);
        writeLCD(lcdLine);
    }
    else
    {
        // Print SPI error message to LCD
        writeLCD("SPI ERROR!");
    }

    // Update the current address and wrap around if needed
    currentAddress = (currentAddress + 1) % 6;
}

// TODO: Complete the writeLCD function
void writeLCD(char *char_in){

  delay(3000);
lcd_command(CLEAR);
lcd_command(CURSOR_HOME);
      lcd_command(TWOLINE_MODE);
      lcd_putstring("EEPROM byte");
      lcd_command(LINE_TWO);
      lcd_putstring(char_in);
```

```
}

// Get ADC value
uint32_t pollADC(void){
      HAL_ADC_Start(&hadc); // start the adc
      HAL_ADC_PollForConversion(&hadc, 100); // poll for conversion
      uint32_t val = HAL_ADC_GetValue(&hadc); // get the adc value
      HAL_ADC_Stop(&hadc); // stop adc
      return val;
}

// Calculate PWM CCR value
uint32_t ADCtoCCR(uint32_t adc_val){
 // TODO: Calculate CCR valUE using an equation
 uint32_t value = (adc_value * 47999) / 4095;
return value;
}

void ADC1_COMP_IRQHandler(void)
{
      //adc_val = HAL_ADC_GetValue(&hadc); // read adc value
      HAL_ADC_IRQHandler(&hadc); //Clear flags
}

// Initialise SPI
static void init_spi(void) {

  // Clock to PB
  RCC->AHBENR |= RCC_AHBENR_GPIOBEN;      // Enable clock for SPI port

  // Set pin modes
  GPIOB->MODER |= GPIO_MODER_MODER13_1; // Set pin SCK (PB13) to Alternate Function
  GPIOB->MODER |= GPIO_MODER_MODER14_1; // Set pin MISO (PB14) to Alternate Function
  GPIOB->MODER |= GPIO_MODER_MODER15_1; // Set pin MOSI (PB15) to Alternate Function
  GPIOB->MODER |= GPIO_MODER_MODER12_0; // Set pin CS (PB12) to output push-pull
  GPIOB->BSRR |= GPIO_BSRR_BS_12;         // Pull CS high

  // Clock enable to SPI
  RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
  SPI2->CR1 |= SPI_CR1_BIDIOE;                                                    //
Enable output
  SPI2->CR1 |= (SPI_CR1_BR_0 |  SPI_CR1_BR_1);                              // Set Baud to
fpclk / 16
  SPI2->CR1 |= SPI_CR1_MSTR;                                                   // Set to
master mode
  SPI2->CR2 |= SPI_CR2_FRXTH;                                                     //
Set RX threshold to be 8 bits
  SPI2->CR2 |= SPI_CR2_SSOE;                                                   // Enable
slave output to work in master mode
  SPI2->CR2 |= (SPI_CR2_DS_0 | SPI_CR2_DS_1 | SPI_CR2_DS_2);      // Set to 8-bit mode
  SPI2->CR1 |= SPI_CR1_SPE;                                                    // Enable
the SPI peripheral
}

// Implements a delay in microseconds
static void spi_delay(uint32_t delay_in_us) {
  volatile uint32_t counter = 0;
  delay_in_us *= 3;
  for(; counter < delay_in_us; counter++) {
    __asm("nop");
    __asm("nop");
  }
}

// Write to EEPROM address using SPI
static void write_to_address(uint16_t address, uint8_t data) {
```

```
        uint8_t dummy; // Junk from the DR

        // Set the Write Enable latch
        GPIOB->BSRR |= GPIO_BSRR_BR_12; // Pull CS low
        spi_delay(1);
        *((uint8_t*)(&SPI2->DR)) = WREN;
        while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
        dummy = SPI2->DR;
        GPIOB->BSRR |= GPIO_BSRR_BS_12; // Pull CS high
        spi_delay(5000);

        // Send write instruction
        GPIOB->BSRR |= GPIO_BSRR_BR_12;                 // Pull CS low
        spi_delay(1);
        *((uint8_t*)(&SPI2->DR)) = WRITE;
        while ((SPI2->SR & SPI_SR_RXNE) == 0);          // Hang while RX is empty
        dummy = SPI2->DR;

        // Send 16-bit address
        *((uint8_t*)(&SPI2->DR)) = (address >> 8);      // Address MSB
        while ((SPI2->SR & SPI_SR_RXNE) == 0);          // Hang while RX is empty
        dummy = SPI2->DR;
        *((uint8_t*)(&SPI2->DR)) = (address);           // Address LSB
        while ((SPI2->SR & SPI_SR_RXNE) == 0);          // Hang while RX is empty
        dummy = SPI2->DR;

        // Send the data
        *((uint8_t*)(&SPI2->DR)) = data;
        while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
        dummy = SPI2->DR;
        GPIOB->BSRR |= GPIO_BSRR_BS_12; // Pull CS high
        spi_delay(5000);
}

// Read from EEPROM address using SPI
static uint8_t read_from_address(uint16_t address) {

        uint8_t dummy; // Junk from the DR

        // Send the read instruction
        GPIOB->BSRR |= GPIO_BSRR_BR_12;                 // Pull CS low
        spi_delay(1);
        *((uint8_t*)(&SPI2->DR)) = READ;
        while ((SPI2->SR & SPI_SR_RXNE) == 0);          // Hang while RX is empty
        dummy = SPI2->DR;

        // Send 16-bit address
        *((uint8_t*)(&SPI2->DR)) = (address >> 8);      // Address MSB
        while ((SPI2->SR & SPI_SR_RXNE) == 0);          // Hang while RX is empty
        dummy = SPI2->DR;
        *((uint8_t*)(&SPI2->DR)) = (address);           // Address LSB
        while ((SPI2->SR & SPI_SR_RXNE) == 0);          // Hang while RX is empty
        dummy = SPI2->DR;

        // Clock in the data
        *((uint8_t*)(&SPI2->DR)) = 0x42;                // Clock out some junk data
        while ((SPI2->SR & SPI_SR_RXNE) == 0);          // Hang while RX is empty
        dummy = SPI2->DR;
        GPIOB->BSRR |= GPIO_BSRR_BS_12;                 // Pull CS high
        spi_delay(5000);

        return dummy;                                               // Return read data
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
```

```
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```