

## GitHub Link

[https://github.com/LuthoYRN/MNGLUT008\\_CBXLIS001\\_EEE3096S/blob/main/Prac3/main.c](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 **writelnLCD** 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

```
1  /* USER CODE BEGIN Header */
2  /**
3   *
4   * @file           : main.c
5   * @brief          : Main program body
6   *
7   * @attention
8   *
9   * Copyright (c) 2023 STMicroelectronics.
10  * All rights reserved.
11  *
12  * This software is licensed under terms that can be found in the LICENSE file
13  * in the root directory of this software component.
14  * If no LICENSE file comes with this software, it is provided AS-IS.
15  *
16  *
17  */
18 /* USER CODE END Header */
19 /* Includes -----*/
20 #include "main.h"
21
22 /* Private includes -----*/
23 /* USER CODE BEGIN Includes */
24 #include <stdio.h>
25 #include "stm32f0xx.h"
26 #include <lcd_stm32f0.c>
27 /* USER CODE END Includes */
28
29 /* Private typedef -----*/
30 /* USER CODE BEGIN PTD */
31
32 /* USER CODE END PTD */
33
34 /* Private define -----*/
35 /* USER CODE BEGIN PD */
36
37 // Definitions for SPI usage
38 #define MEM_SIZE 8192 // bytes
39 #define WREN 0b00000110 // enable writing
40 #define WRDI 0b00000100 // disable writing
41 #define RDSR 0b00000101 // read status register
42 #define WRSR 0b00000001 // write status register
43 #define READ 0b00000011
44 #define WRITE 0b00000010
45 /* USER CODE END PD */
46 /*our declared variables*/
47 uint32_t period = 500; // Initial frequency period (500 ms, 2 Hz)
48 uint32_t previoustime = 0;
49 uint32_t adc_value=0;
50 static uint8_t binaryArray[6] = {
51     0b10101010, // 170 in decimal
52     0b01010101, // 85 in decimal
53     0b11001100, // 204 in decimal
54     0b00110011, // 51 in decimal
55     0b11110000, // 240 in decimal
56     0b00001111 // 15 in decimal
57 };
58 uint16_t currentAddress = 0;
59 /* Private macro -----*/
60 /* USER CODE BEGIN PM */
61
62 /* USER CODE END PM */
63
64 /* Private variables -----*/
65 ADC_HandleTypeDef hadc;
```

```

66
67 TIM_HandleTypeDef htim3;
68 TIM_HandleTypeDef htim6;
69 TIM_HandleTypeDef htim16;
70
71 /* USER CODE BEGIN PV */
72
73 // TODO: Define input variables
74
75
76 /* USER CODE END PV */
77
78 /* Private function prototypes -----*/
79 void SystemClock_Config(void);
80 static void MX_GPIO_Init(void);
81 static void MX_ADC_Init(void);
82 static void MX_TIM3_Init(void);
83 static void MX_TIM16_Init(void);
84 static void MX_TIM6_Init(void);
85 /* USER CODE BEGIN PFP */
86 void EXTI0_1_IRQHandler(void);
87 void TIM16_IRQHandler(void);
88 void writeLCD(char *char_in);
89
90 // ADC functions
91 uint32_t pollADC(void);
92 uint32_t ADCtoCCR(uint32_t adc_val);
93
94 // SPI functions
95 static void init_spi(void);
96 static void write_to_address(uint16_t address, uint8_t data);
97 static uint8_t read_from_address(uint16_t address);
98 static void spi_delay(uint32_t delay_in_us);
99 /* USER CODE END PFP */
100
101 /* Private user code -----*/
102 /* USER CODE BEGIN 0 */
103
104 /* USER CODE END 0 */
105
106 /**
107  * @brief The application entry point.
108  * @retval int
109  */
110 int main(void)
111 {
112
113     /* USER CODE BEGIN 1 */
114     /* USER CODE END 1 */
115
116     /* MCU Configuration-----*/
117
118     /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
119     HAL_Init();
120
121     /* USER CODE BEGIN Init */
122     /* USER CODE END Init */
123
124     /* Configure the system clock */
125     SystemClock_Config();
126
127     /* USER CODE BEGIN SysInit */
128     /* USER CODE END SysInit */
129
130     /* Initialize all configured peripherals */
131     init_spi();
132     MX_GPIO_Init();

```

```

133 MX_ADC_Init();
134 MX_TIM3_Init();
135 MX_TIM16_Init();
136 MX_TIM6_Init();
137 /* USER CODE BEGIN 2 */
138
139 // Initialise LCD
140 init_LCD();
141
142 // Start timers
143 HAL_TIM_Base_Start_IT(&htim6);
144 HAL_TIM_Base_Start_IT(&htim16);
145
146 // PWM setup
147 uint32_t CCR = 0;
148 HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3); // Start PWM on TIM3 Channel 3
149
150 // TODO: Write all bytes to EEPROM using "write_to_address"
151 for(int i = 0; i < 6; i++){
152     write_to_address(i, binaryArray[i]);
153 }
154 /* USER CODE END 2 */
155 //M16_IRQHandler();
156 /* Infinite loop */
157 /* USER CODE BEGIN WHILE */
158 while (1)
159 {
160
161     // TODO: Poll ADC
162
163     adc_value = pollADC(); // Read ADC value from potentiometer
164     // TODO: Get CCR
165     CCR = ADCtoCCR(adc_value); // Convert ADC value to CCR value
166
167
168     // Update PWM value
169     __HAL_TIM_SetCompare(&htim3, TIM_CHANNEL_3, CCR);
170     // Wait for delay ms
171     HAL_Delay (period);
172
173     /* USER CODE END WHILE */
174
175     /* USER CODE BEGIN 3 */
176 }
177 /* USER CODE END 3 */
178 }
179
180 /**
181  * @brief System Clock Configuration
182  * @retval None
183  */
184 void SystemClock_Config(void)
185 {
186     LL_FLASH_SetLatency(LL_FLASH_LATENCY_0);
187     while(LL_FLASH_GetLatency() != LL_FLASH_LATENCY_0)
188     {
189     }
190     LL_RCC_HSI_Enable();
191
192     /* Wait till HSI is ready */
193     while(LL_RCC_HSI_IsReady() != 1)
194     {
195     }
196     LL_RCC_HSI_SetCalibTrimming(16);
197     LL_RCC_HSI14_Enable();
198
199

```

```

200 /* Wait till HSI14 is ready */
201 while(LL_RCC_HSI14_IsReady() != 1)
202 {
203
204 }
205 LL_RCC_HSI14_SetCalibTrimming(16);
206 LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);
207 LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_1);
208 LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_HSI);
209
210 /* Wait till System clock is ready */
211 while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_HSI)
212 {
213
214 }
215 LL_SetSystemCoreClock(8000000);
216
217 /* Update the time base */
218 if (HAL_InitTick (TICK_INT_PRIORITY) != HAL_OK)
219 {
220     Error_Handler();
221 }
222 LL_RCC_HSI14_EnableADCControl();
223 }
224
225 /**
226  * @brief ADC Initialization Function
227  * @param None
228  * @retval None
229  */
230 static void MX_ADC_Init(void)
231 {
232
233     /* USER CODE BEGIN ADC_Init 0 */
234     /* USER CODE END ADC_Init 0 */
235
236     ADC_ChannelConfTypeDef sConfig = {0};
237
238     /* USER CODE BEGIN ADC_Init 1 */
239
240     /* USER CODE END ADC_Init 1 */
241
242     /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and
243     number of conversion)
244     */
245     hadc.Instance = ADC1;
246     hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
247     hadc.Init.Resolution = ADC_RESOLUTION_12B;
248     hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
249     hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
250     hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
251     hadc.Init.LowPowerAutoWait = DISABLE;
252     hadc.Init.LowPowerAutoPowerOff = DISABLE;
253     hadc.Init.ContinuousConvMode = DISABLE;
254     hadc.Init.DiscontinuousConvMode = DISABLE;
255     hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;
256     hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
257     hadc.Init.DMAContinuousRequests = DISABLE;
258     hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
259     if (HAL_ADC_Init(&hadc) != HAL_OK)
260     {
261         Error_Handler();
262     }
263
264     /** Configure for the selected ADC regular channel to be converted.
265     */
266     sConfig.Channel = ADC_CHANNEL_6;

```

```

267 sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
268 sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
269 if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
270 {
271     Error_Handler();
272 }
273 /* USER CODE BEGIN ADC_Init 2 */
274 ADC1->CR |= ADC_CR_ADCAL;
275 while(ADC1->CR & ADC_CR_ADCAL); // Calibrate the ADC
276 ADC1->CR |= (1 << 0); // Enable ADC
277 while((ADC1->ISR & (1 << 0)) == 0); // Wait for ADC ready
278 /* USER CODE END ADC_Init 2 */
279
280 }
281
282 /**
283  * @brief TIM3 Initialization Function
284  * @param None
285  * @retval None
286  */
287 static void MX_TIM3_Init(void)
288 {
289     /* USER CODE BEGIN TIM3_Init 0 */
290
291     /* USER CODE END TIM3_Init 0 */
292
293     TIM_ClockConfigTypeDef sClockSourceConfig = {0};
294     TIM_MasterConfigTypeDef sMasterConfig = {0};
295     TIM_OC_InitTypeDef sConfigOC = {0};
296
297     /* USER CODE BEGIN TIM3_Init 1 */
298
299     /* USER CODE END TIM3_Init 1 */
300     htim3.Instance = TIM3;
301     htim3.Init.Prescaler = 0;
302     htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
303     htim3.Init.Period = 47999;
304     htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
305     htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
306     if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
307     {
308         Error_Handler();
309     }
310     sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
311     if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
312     {
313         Error_Handler();
314     }
315     if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
316     {
317         Error_Handler();
318     }
319     sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
320     sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
321     if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
322     {
323         Error_Handler();
324     }
325     sConfigOC.OCMode = TIM_OCMODE_PWM1;
326     sConfigOC.Pulse = 0;
327     sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
328     sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
329     if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
330     {
331         Error_Handler();
332     }
333

```

```

334  /* USER CODE BEGIN TIM3_Init 2 */
335
336  /* USER CODE END TIM3_Init 2 */
337  HAL_TIM_MspPostInit(&htim3);
338
339  }
340
341  /**
342   * @brief TIM6 Initialization Function
343   * @param None
344   * @retval None
345   */
346  static void MX_TIM6_Init(void)
347  {
348
349      /* USER CODE BEGIN TIM6_Init 0 */
350
351      /* USER CODE END TIM6_Init 0 */
352
353      TIM_MasterConfigTypeDef sMasterConfig = {0};
354
355      /* USER CODE BEGIN TIM6_Init 1 */
356
357      /* USER CODE END TIM6_Init 1 */
358      htim6.Instance = TIM6;
359      htim6.Init.Prescaler = 8000-1;
360      htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
361      htim6.Init.Period = 500-1;
362      htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
363      if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
364      {
365          Error_Handler();
366      }
367      sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
368      sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
369      if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
370      {
371          Error_Handler();
372      }
373      /* USER CODE BEGIN TIM6_Init 2 */
374      NVIC_EnableIRQ(TIM6_IRQn);
375      /* USER CODE END TIM6_Init 2 */
376
377  }
378
379  /**
380   * @brief TIM16 Initialization Function
381   * @param None
382   * @retval None
383   */
384  static void MX_TIM16_Init(void)
385  {
386
387      /* USER CODE BEGIN TIM16_Init 0 */
388
389      /* USER CODE END TIM16_Init 0 */
390
391      /* USER CODE BEGIN TIM16_Init 1 */
392
393      /* USER CODE END TIM16_Init 1 */
394      htim16.Instance = TIM16;
395      htim16.Init.Prescaler = 8000-1;
396      htim16.Init.CounterMode = TIM_COUNTERMODE_UP;
397      htim16.Init.Period = 1000-1;
398      htim16.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
399      htim16.Init.RepetitionCounter = 0;
400      htim16.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;

```

```

401 if (HAL_TIM_Base_Init(&htim16) != HAL_OK)
402 {
403     Error_Handler();
404 }
405 /* USER CODE BEGIN TIM16_Init 2 */
406 NVIC_EnableIRQ(TIM16_IRQn);
407 /* USER CODE END TIM16_Init 2 */
408
409 }
410
411 /**
412  * @brief GPIO Initialization Function
413  * @param None
414  * @retval None
415  */
416 static void MX_GPIO_Init(void)
417 {
418     LL_EXTI_InitTypeDef EXTI_InitStruct = {0};
419     LL_GPIO_InitTypeDef GPIO_InitStruct = {0};
420 /* USER CODE BEGIN MX_GPIO_Init_1 */
421 /* USER CODE END MX_GPIO_Init_1 */
422
423     /* GPIO Ports Clock Enable */
424     LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOF);
425     LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
426     LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);
427
428     /**/
429     LL_GPIO_ResetOutputPin(LED7_GPIO_Port, LED7_Pin);
430
431     /**/
432     LL_SYSCFG_SetEXTISource(LL_SYSCFG_EXTI_PORTA, LL_SYSCFG_EXTI_LINE0);
433
434     /**/
435     LL_GPIO_SetPinPull(Button0_GPIO_Port, Button0_Pin, LL_GPIO_PULL_UP);
436
437     /**/
438     LL_GPIO_SetPinMode(Button0_GPIO_Port, Button0_Pin, LL_GPIO_MODE_INPUT);
439
440     /**/
441     EXTI_InitStruct.Line_0_31 = LL_EXTI_LINE_0;
442     EXTI_InitStruct.LineCommand = ENABLE;
443     EXTI_InitStruct.Mode = LL_EXTI_MODE_IT;
444     EXTI_InitStruct.Trigger = LL_EXTI_TRIGGER_RISING;
445     LL_EXTI_Init(&EXTI_InitStruct);
446
447     /**/
448     GPIO_InitStruct.Pin = LED7_Pin;
449     GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
450     GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_LOW;
451     GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
452     GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
453     LL_GPIO_Init(LED7_GPIO_Port, &GPIO_InitStruct);
454
455 /* USER CODE BEGIN MX_GPIO_Init_2 */
456 HAL_NVIC_SetPriority(EXTI0_1_IRQn, 0, 0);
457 HAL_NVIC_EnableIRQ(EXTI0_1_IRQn);
458 /* USER CODE END MX_GPIO_Init_2 */
459 }
460
461 /* USER CODE BEGIN 4 */
462 void EXTI0_1_IRQHandler(void)
463 {
464     // TODO: Add code to switch LED7 delay frequency
465     uint32_t currentTime = HAL_GetTick();
466     // Check if button is pressed and debounce time(100ms) has passed
467     if (LL_GPIO_IsInputPinSet(GPIOA, LL_GPIO_PIN_0) && (currentTime -

```



```

468     previoustime > 500))
469     {
470         // Toggle between 500 ms (2 Hz) and 1000 ms (1 Hz) period
471         if( period== 1000-1){
472             period= 500-1;
473         }
474         // Update the previous time for debounce
475         else{
476             period =1000-1;
477         }
478         previoustime = currentTime;
479         __HAL_TIM_SET_AUTORELOAD(&htim6,period);
480     }
481     HAL_GPIO_EXTI_IRQHandler(Button0_Pin); // Clear interrupt flags
482 }
483
484 void TIM6_IRQHandler(void)
485 {
486     // Acknowledge interrupt
487     HAL_TIM_IRQHandler(&htim6);
488
489     // Toggle LED7
490     HAL_GPIO_TogglePin(GPIOB, LED7_Pin);
491 }
492
493 void TIM16_IRQHandler(void)
494 {
495     // Acknowledge interrupt
496     HAL_TIM_IRQHandler(&htim16);
497
498     // TODO: Initialise a string to output second line on LCD
499
500
501     // TODO: Change LED pattern; output 0x01 if the read SPI data is incorrect
502     // Read the value from EEPROM at the current address
503
504     uint8_t eepromValue = read_from_address(currentAddress);
505
506     // Check if the read value matches the expected value from binaryArray
507     if (eepromValue == binaryArray[currentAddress])
508     {
509         // Format and print the correct value to LCD
510         char lcdLine[16];
511         //f(lcdLine, sizeof(lcdLine), "EEPROM byte:\n%d", eepromValue);
512         sprintf(lcdLine, "%d", eepromValue);
513         writeLCD(lcdLine);
514     }
515     else
516     {
517         // Print SPI error message to LCD
518         writeLCD("SPI ERROR!");
519     }
520
521     // Update the current address and wrap around if needed
522     currentAddress = (currentAddress + 1) % 6;
523 }
524
525 // TODO: Complete the writeLCD function
526 void writeLCD(char *char_in) {
527
528     delay(3000);
529     lcd_command(CLEAR);
530     lcd_command(CURSOR_HOME);
531     lcd_command(TWOLINE_MODE);
532     lcd_putstring("EEPROM byte");
533     lcd_command(LINE_TWO);
534     lcd_putstring(char_in);

```

```

535 }
536
537 // Get ADC value
538 uint32_t pollADC(void) {
539     HAL_ADC_Start(&hadc); // start the adc
540     HAL_ADC_PollForConversion(&hadc, 100); // poll for conversion
541     uint32_t val = HAL_ADC_GetValue(&hadc); // get the adc value
542     HAL_ADC_Stop(&hadc); // stop adc
543     return val;
544 }
545
546 // Calculate PWM CCR value
547 uint32_t ADCToCCR(uint32_t adc_val) {
548     // TODO: Calculate CCR value using an equation
549     uint32_t value = (adc_value * 47999) / 4095;
550     return value;
551 }
552
553 void ADC1_COMP_IRQHandler(void)
554 {
555     //adc_val = HAL_ADC_GetValue(&hadc); // read adc value
556     HAL_ADC_IRQHandler(&hadc); //Clear flags
557 }
558
559 // Initialise SPI
560 static void init_spi(void) {
561
562     // Clock to PB
563     RCC->AHBENR |= RCC_AHBENR_GPIOBEN; // Enable clock for SPI port
564
565     // Set pin modes
566     GPIOB->MODER |= GPIO_MODER_MODER13_1; // Set pin SCK (PB13) to Alternate Function
567     GPIOB->MODER |= GPIO_MODER_MODER14_1; // Set pin MISO (PB14) to Alternate Function
568     GPIOB->MODER |= GPIO_MODER_MODER15_1; // Set pin MOSI (PB15) to Alternate Function
569     GPIOB->MODER |= GPIO_MODER_MODER12_0; // Set pin CS (PB12) to output push-pull
570     GPIOB->BSRR |= GPIO_BSRR_BS_12; // Pull CS high
571
572     // Clock enable to SPI
573     RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
574     SPI2->CR1 |= SPI_CR1_BIDIOE; //
575
576     // Enable output
577     SPI2->CR1 |= (SPI_CR1_BR_0 | SPI_CR1_BR_1); // Set Baud to
578     fpclk / 16
579     SPI2->CR1 |= SPI_CR1_MSTR; // Set to
580     master mode
581     SPI2->CR2 |= SPI_CR2_FRXTH; //
582     Set RX threshold to be 8 bits
583     SPI2->CR2 |= SPI_CR2_SSOE; // Enable
584     slave output to work in master mode
585     SPI2->CR2 |= (SPI_CR2_DS_0 | SPI_CR2_DS_1 | SPI_CR2_DS_2); // Set to 8-bit mode
586     SPI2->CR1 |= SPI_CR1_SPE; // Enable
587     the SPI peripheral
588 }
589
590 // Implements a delay in microseconds
591 static void spi_delay(uint32_t delay_in_us) {
592     volatile uint32_t counter = 0;
593     delay_in_us *= 3;
594     for(; counter < delay_in_us; counter++) {
595         __asm("nop");
596         __asm("nop");
597     }
598 }
599
600 // Write to EEPROM address using SPI
601 static void write_to_address(uint16_t address, uint8_t data) {

```

```

602 uint8_t dummy; // Junk from the DR
603
604 // Set the Write Enable latch
605 GPIOB->BSRR |= GPIO_BSRR_BR_12; // Pull CS low
606 spi_delay(1);
607 *((uint8_t*)(&SPI2->DR)) = WREN;
608 while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
609 dummy = SPI2->DR;
610 GPIOB->BSRR |= GPIO_BSRR_BS_12; // Pull CS high
611 spi_delay(5000);
612
613 // Send write instruction
614 GPIOB->BSRR |= GPIO_BSRR_BR_12; // Pull CS low
615 spi_delay(1);
616 *((uint8_t*)(&SPI2->DR)) = WRITE;
617 while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
618 dummy = SPI2->DR;
619
620 // Send 16-bit address
621 *((uint8_t*)(&SPI2->DR)) = (address >> 8); // Address MSB
622 while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
623 dummy = SPI2->DR;
624 *((uint8_t*)(&SPI2->DR)) = (address); // Address LSB
625 while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
626 dummy = SPI2->DR;
627
628 // Send the data
629 *((uint8_t*)(&SPI2->DR)) = data;
630 while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
631 dummy = SPI2->DR;
632 GPIOB->BSRR |= GPIO_BSRR_BS_12; // Pull CS high
633 spi_delay(5000);
634 }
635
636 // Read from EEPROM address using SPI
637 static uint8_t read_from_address(uint16_t address) {
638
639     uint8_t dummy; // Junk from the DR
640
641     // Send the read instruction
642     GPIOB->BSRR |= GPIO_BSRR_BR_12; // Pull CS low
643     spi_delay(1);
644     *((uint8_t*)(&SPI2->DR)) = READ;
645     while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
646     dummy = SPI2->DR;
647
648     // Send 16-bit address
649     *((uint8_t*)(&SPI2->DR)) = (address >> 8); // Address MSB
650     while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
651     dummy = SPI2->DR;
652     *((uint8_t*)(&SPI2->DR)) = (address); // Address LSB
653     while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
654     dummy = SPI2->DR;
655
656     // Clock in the data
657     *((uint8_t*)(&SPI2->DR)) = 0x42; // Clock out some junk data
658     while ((SPI2->SR & SPI_SR_RXNE) == 0); // Hang while RX is empty
659     dummy = SPI2->DR;
660     GPIOB->BSRR |= GPIO_BSRR_BS_12; // Pull CS high
661     spi_delay(5000);
662
663     return dummy; // Return read data
664 }
665 /* USER CODE END 4 */
666
667 /**
668  * @brief This function is executed in case of error occurrence.

```

```

669 * @retval None
670 */
671 void Error_Handler(void)
672 {
673     /* USER CODE BEGIN Error_Handler_Debug */
674     /* User can add his own implementation to report the HAL error return state */
675     __disable_irq();
676     while (1)
677     {
678     }
679     /* USER CODE END Error_Handler_Debug */
680 }
681
682 #ifndef USE_FULL_ASSERT
683 /**
684 * @brief Reports the name of the source file and the source line number
685 * where the assert_param error has occurred.
686 * @param file: pointer to the source file name
687 * @param line: assert_param error line source number
688 * @retval None
689 */
690 void assert_failed(uint8_t *file, uint32_t line)
691 {
692     /* USER CODE BEGIN 6 */
693     /* User can add his own implementation to report the file name and line number,
694     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
695     /* USER CODE END 6 */
696 }
697 #endif /* USE_FULL_ASSERT */

```