Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

CEG4166/CSI4141/SEG4155 Real-Time System Design Winter 2024

Lab 4: Project's Alarm System with PIR Motion Detector and Buzzer

Submission Date: March 25th, 2024

Group Number: 5

Lab section: A01 (Tuesday PM)

Team Member List:

- Moise Baleke – 300207962
- Zinah Al-Saadi – 8867078
- Decaho Gbegbe – 300094197

Table of Contents

Introduction

Lab 4 requires is the continuation of the first module for the alarm system project created in the previous lab. To quickly recap lab 3, students were required to create an alarm system using a 4x4 Keypad and an LCD display connected to the nucleo-F446RE where the user is required to enter a four or six digit code to configure the system and once the system is configured successfully, it can enter two states, Armed if the correct code is inputted or Not Armed if the wrong code is inputted. This lab requires students to add a passive infrared PIR motion detector and a buzzer to the system integrated previously. The PIR sensor will be able to detect bodies moving in its field of view. The PIR sensor is split into two halves, with one half causes a positive differential change and the other causes a negative differential change. The two halves are wired in a way such that the result in cancelling each other out. The buzzer added will simply be used to provide the user with an audio alert when there is movement detected through the PIR.

Objectives

The main objectives of this lab are to:

- Reimplement the alarm system using nucleo-F446RE from the previous lab.
- Add a passive infrared PIR sensor to detect warm bodies passing by its field of view.
- Add a buzzer for audio cue when the PIR sensor has detected movement.

Problem Analysis

Lab 4 extends our existing alarm system by adding a Passive Infrared (PIR) motion detector and a buzzer. The PIR sensor detects motion using infrared radiation, which is naturally emitted by warm bodies. When this sensor captures a change in radiation—like when a person moves—it reacts. The buzzer, in turn, uses electrical signals to produce sound, serving as the alarm's auditory signal.

The challenge lies in integrating these components effectively. The system must interpret the PIR sensor's data accurately to minimize false alarms. It also requires a real-time software response to trigger the buzzer when true motion is detected. Task management becomes crucial because the system operates with multiple tasks that run concurrently. These tasks manage sensor readings, buzzer alerts, and user interactions for arming or disarming the system.

A critical feature is the system's 60-second delay after arming. This delay allows a user to exit the premises without triggering the alarm. The same delay applies when entering to disarm the alarm, demanding precise timing control within the system's software.

These new components must integrate with the already present keypad and OLED display, maintaining system responsiveness and reliability. The keypad sets the alarm, while the OLED displays the status. Together, they form a user interface for the alarm system, now enhanced with motion detection and sound alerts.

Solution Design

In the design of our alarm system, we plan a continuous monitoring strategy. The system checks the motion sensor's output regularly to detect any movement. If the sensor signals an intrusion, the system will enter an armed state and trigger the buzzer. This alert continues until a correct password is entered.

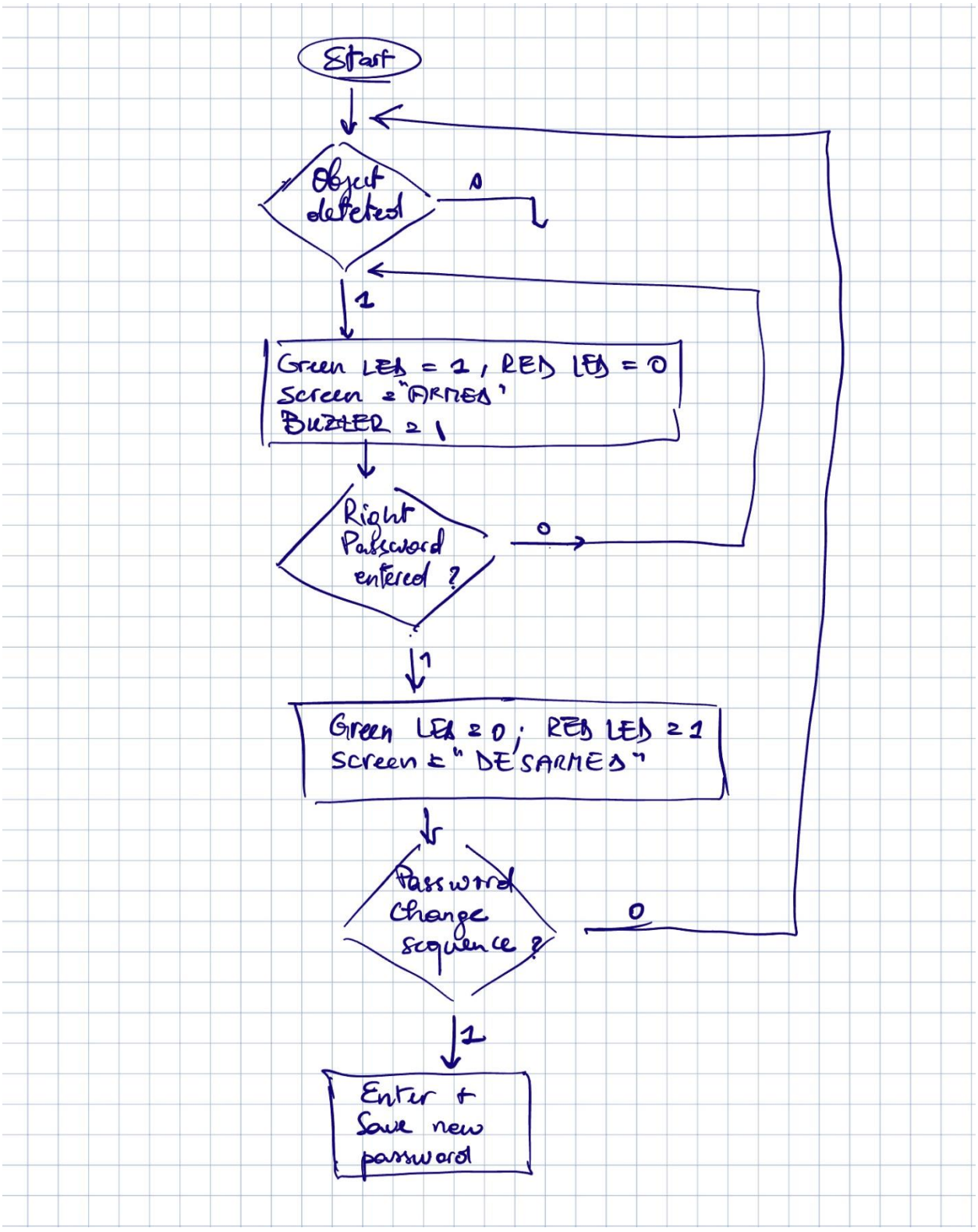
The disarm process involves password verification. Upon successful entry, the system deactivates the alarm. Moreover, the system provides an option to change the password if a specific sequence of keys is pressed immediately after disarming.

The system state is constantly displayed on an OLED screen and indicated by an LED, ensuring the user is always aware of whether the system is armed or disarmed.

To illustrate the solution clearly, we will create a flowchart. This diagram will visually map out the process from initial monitoring to potential state changes, including arming, disarming, and password modification stages.

Key methods, such as `pollMotionSensor`, `activateAlarm`, `checkPassword`, `disarmSystem`, and `changePasswordSequence`, will be implemented to ensure all these fucntions. These functions rely on the STM32 HAL library for managing hardware components.

The described flow can be illustrated in the following flow chart:


Start
Object detected
0
1
Green LED = 1, RED LED = 0
Screen = "ARMED"
BUZZER = 1
Right Password entered ?
0
1
Green LED = 0; RED LED = 1
Screen = "DESARMED"
Password Change sequence ?
0
1
Enter & Save new password

Component Usage

The following are a set of components used to complete lab 4

- **Nucleo-F446RE Development board:** This board incorporates the STM32 microcontroller. The microcontroller is essential in terms of monitoring the inputs and outputs from the passive infrared PIR motion detector as well as the buzzer.
- **PIR motion detector:** Created from a pyroelectric sensor, the PIR sensor can detect different level of infrared radiation. As a hotter object becomes closer to the sensor, it emits more radiation. This concept will be used through the detector to identify when a person or animal passes by the field of vision of the PIR sensor. PIR sensor is divided into two halves where one half is intercepted when a warm body passes by the detection area. The second half is intercepted when the warm body leaves the detection area
- **Buzzer:** a buzzer will be incorporated to be able to alarm the user by providing an audio representation when the PIR sensor has detected any sort of movement.
- **4x4 Keypad**: The keypad is a critical input device in our system, featuring 16 keys arranged into 4 rows and 4 columns. It is used for setting and entering a secure pin, which is a crucial aspect of the alarm system, allowing users to activate or deactivate the alarm state. Its design ensures a tactile interface for pin entry, which is more secure and reliable than touch interfaces in the context of security systems.
- **LCD Display**: The display provides immediate visual feedback to the user, echoing pin entry with asterisks to protect the pin's secrecy. It also displays the system's status, providing a clear indication of whether the alarm is armed or not. The choice of an LCD is deliberate, given its low power consumption and high visibility, which is crucial in security systems where clarity and power efficiency are key.
- **LED Lights**: As supplemental indicators, the LED lights offer a quick and intuitive status update. A red LED signifies an armed system, warning the user and potential intruders of the active security measures. Conversely, a green LED denotes the system's inactive state, indicating safety for entry. These color codes align with universal standards for stop (red) and go (green), ensuring the system's status is understood at a glance.

Algorithmic Approach

In our alarm system, Task1 plays a critical role by monitoring the PIR motion sensor. If motion is detected, Task1 updates the `systemArmed` global variable, engaging the system's armed state. It then activates the buzzer for an audible alert and uses LEDs for a visual indication. Importantly, Task1 also invokes `checkPassword` to validate the entered password. A correct entry disarms the system, resetting the `systemArmed` state and silencing the buzzer. Task1's procedural details are elaborated upon in Appendix Code A.1.1.

Task2 is responsible for the user interface. It updates the OLED display based on the `systemArmed` global variable, alternating between showing "ARMED" or "DESARMED" messages. This provides real-time feedback to the user, as Task2's code is detailed in Appendix Code A.1.2.

Task3 is designed to respond to user commands for changing the password. It detects a particular sequence of keys and initiates the password change routine. For the duration of the new password setup, Task3 ensures the system's other operations are paused, as shown in Appendix Code A.1.3.

To ensure the integrity of the `systemArmed` and `correctPassword` variables, mutexes have been implemented. These mutexes are critical as they prevent concurrent modifications of these variables by different tasks, which could lead to data inconsistency or race conditions.

The `keypad4X4`, `ssd1306`, and `fonts.h` libraries are fundamental to our system's functionality. `keypad4X4` is used for input processing, `ssd1306` for managing OLED interactions, and `fonts.h` for rendering text in various sizes. These libraries, detailed in the Appendix, simplify the interface with hardware components, allowing us to concentrate on the unique logic required for the system's security operations. Their inclusion supports a robust, maintainable codebase and reflects best practices in embedded systems by relying on established, reliable code for standard functionalities.

Implementation Evaluation

The evaluation of our implemented alarm system in a controlled environment facilitated by STM32CubeIDE's debug mode has revealed insights into the system's performance and highlighted areas for improvement. Using print statements as checkpoints, we confirmed the functionality of the primary features: motion detection, system arming, password input, and visual feedback via the OLED.

Task1 and Task2 demonstrated flawless operation, correctly interpreting sensor input to arm the system and updating the OLED display accordingly. As expected, Task1 armed the system upon detecting motion, activating the buzzer, while Task2 kept the user informed of the system's state in real-time. The implementation of these features aligned with our design expectations, demonstrating effective coding practices and utilization of the HAL library for hardware interaction.

However, our implementation faced a significant issue with the semaphore mechanism intended for managing access to shared resources. These semaphores unexpectedly led to a deadlock, halting the system’s responsiveness. The precise cause was not identified within the project's time constraints, thus preventing a resolution before the evaluation's conclusion. The Appendix references the associated code sections, particularly Task3 in Code A.1.3, where semaphore usage was critical.

Moreover, the system's efficacy was somewhat compromised by the current task prioritization. In retrospect, a more nuanced approach to assigning task priorities could enhance the system's responsiveness, especially in scenarios where motion detection should supersede the password change process. Optimizing task priorities could prevent situations where urgent security alerts need to pre-empt less critical system functions.

In summary, the code's execution, excluding semaphore-related complications, met our projected functional criteria with regards to system status updates and user interaction through the keypad and screen. The execution of our code underscores the importance of a thorough testing phase, particularly for concurrency and resource-sharing features like semaphores. Future iterations will aim to resolve the semaphore deadlock and refine task priority settings to achieve a more robust and efficient alarm system.

Results and Validation

Our project's results and the validation phase were constrained by material limitations. Specifically, we could only test the keypad-password system's interactivity with the screen display and the motion detection's LED indicator independently. Due to the unavailability of additional hardware, the intended simultaneous operation of these systems could not be fully evaluated.

Despite these constraints, individual tests of the system components yielded valuable insights. The keypad-password interface successfully accepted input and displayed the system state on the OLED screen without issues. The motion detection system, while functioning in isolation, demonstrated the capability to trigger the LED indicator as expected upon detecting motion.

However, intermittent delays were observed during the operation of these isolated tests. The delays were particularly noticeable when the system was recovering from a trigger event or processing input. These irregularities suggest that the task prioritization within our real-time operating system may not have been optimally configured. The use of long delay functions within the task processes likely contributed to these inefficiencies. These functions, while simple to implement, can be detrimental to the system's responsiveness, as they halt task execution for their duration, which in a multi-threaded environment can lead to significant performance degradation.

This experience has highlighted the critical importance of task management in real-time systems, especially regarding priority assignment and the careful use of delay functions. Optimal prioritization ensures that critical tasks, such as motion detection, are given precedence over less time-sensitive tasks, such as password input or screen updates. Moreover, implementing non-blocking mechanisms or utilizing interrupt-driven designs could significantly reduce latency and improve system responsiveness.

Moving forward, refinement of the task scheduling algorithm and a re-evaluation of the delays introduced into the processes are essential steps. Adjusting these elements will enhance the system's ability to perform its functions promptly and reliably, thus improving overall functionality and user experience. Future iterations of the project will also aim to secure the necessary materials for a comprehensive system test and integration.

Links to videos:

https://uottawa-my.sharepoint.com/personal/mbale067_uottawa_ca/Documents/lab4ceg4166?csf=1&web=1&e=5jI0Hh

Addressing Challenges

During the course of this lab project, we encountered several challenges, particularly in the integration of different system components and task scheduling within the real-time operating environment.

A key challenge was the integration of the motion detection system and the keypad-password interface. Due to material constraints, these components had to be tested independently rather than as a cohesive unit. This prevented us from fully assessing the interaction between the detection, alerting, and disarming mechanisms of the alarm system. To mitigate this issue, we simulated the system's operations as closely as possible within the available means, ensuring that each component functioned as expected in isolation.

Another significant challenge was the semaphore-induced deadlock that stalled the system's operations. Semaphores are crucial for managing concurrent access to shared resources but can lead to complex synchronization issues. Our initial semaphore implementation inadvertently led to a blocking scenario where tasks awaiting semaphore release were indefinitely suspended. Recognizing this, we identified the need for a more refined approach to inter-task communication and synchronization. Future solutions will focus on non-blocking algorithms and possibly utilize interrupt-driven mechanisms to avoid such deadlocks.

Task prioritization also proved to be a complex issue. Inadequate prioritization led to noticeable system delays, particularly when lower-priority tasks were executed before critical ones. This experience taught us the importance of a well-designed priority scheme in real-time systems, where timely response to events is paramount.

The lessons learned from these challenges were invaluable. They highlighted the need for comprehensive testing, particularly in systems where hardware and software components must work in concert. We also recognized the importance of adopting flexible design strategies that allow for the adjustment of task priorities and synchronization mechanisms as the system's requirements evolve. These insights will undoubtedly inform our approach to future projects, guiding us toward creating more robust and responsive real-time systems.

Conclusion

In Lab 4, we encountered critical learning opportunities, particularly with semaphore management and task prioritization within our real-time system. Errors such as semaphore deadlocks, while challenging, shed light on the intricacies of resource sharing between concurrent tasks. Despite these hurdles, the alarm system was successful in its fundamental functions of arming and disarming in response to sensor inputs and user prompts.

Material limitations necessitated a deviation from our comprehensive integration plan, compelling us to test system components separately. This divergence from the planned design underscored the importance of flexibility in the face of unforeseen constraints.

The project's success was marked by its ability to fulfill key objectives, despite the noted delays affecting system responsiveness. Lessons learned from these challenges, particularly the importance of rigorous testing and adaptable design strategies, will guide our future endeavors. The insights gained from the practical application of theoretical knowledge to real-world scenarios have equipped us with a deeper understanding of designing and implementing effective real-time systems.

Workload Distribution

The table below shows the work undertaken by each team member

Student name	Student number	Working percent
Moise Baleke	300207962	100%
Zinah Al-Saadi	8867078	100%
Decaho Gbegbe	300094197	100%

Appendix

A1.1 Main.c

```
/* USER CODE BEGIN Header */
/**

******************************************************************************
*
  * @file           : main.c
```

```
  * @brief          : Main program body
******************************************************************************
*
  * @attention
  *
  * Copyright (c) 2024 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"
#include "cmsis_os.h"
#include "keypad4X4.h"
#include "ssd1306.h"
#include "fonts.h"
#include <stdio.h>

/* Private includes ----------------------------------------------------------
-*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

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

/* USER CODE END PTD */

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

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------
-*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------
-*/
UART_HandleTypeDef huart2;

/* Definitions for Task1 */
```

```c
osThreadId_t Task1Handle;
const osThreadAttr_t Task1_attributes = {
  .name = "Task1",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for Task2 */
osThreadId_t Task2Handle;
const osThreadAttr_t Task2_attributes = {
  .name = "Task2",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for Task3 */
osThreadId_t Task3Handle;
const osThreadAttr_t Task3_attributes = {
  .name = "Task3",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};
/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
void StartTask1(void *argument);
void StartTask2(void *argument);
void StartTask3(void *argument);

/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
uint8_t dataTask1[] = "Hi, Task1\r\n";
uint8_t dataTask2[] = "Hello, Task2\r\n";
uint8_t dataTask3[] = "Great, Task3\r\n";

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */
```

```c
  /* 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 */
  MX_GPIO_Init();
  MX_USART2_UART_Init();

  SSD1306_Init();
    char snum[5];

    SSD1306_GotoXY (0,0);
    SSD1306_Puts ("NIZAR", &Font_11x18, 1);
    SSD1306_GotoXY (0, 30);
    SSD1306_Puts ("MOHIDEEN", &Font_11x18, 1);
    SSD1306_UpdateScreen();
    HAL_Delay (1000);

    SSD1306_ScrollRight(0,7);
    HAL_Delay(3000);
    SSD1306_ScrollLeft(0,7);
    HAL_Delay(3000);
    SSD1306_Stopscroll();
    SSD1306_Clear();
    SSD1306_GotoXY (35,0);
    SSD1306_Puts ("SCORE", &Font_11x18, 1);

   MX_I2C1_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */

  /* Init scheduler */
  osKernelInitialize();

  /* USER CODE BEGIN RTOS_MUTEX */
  /* add mutexes, ... */
  /* USER CODE END RTOS_MUTEX */

  /* USER CODE BEGIN RTOS_SEMAPHORES */
  /* add semaphores, ... */
  /* USER CODE END RTOS_SEMAPHORES */
```

```c
  /* USER CODE BEGIN RTOS_TIMERS */
  /* start timers, add new ones, ... */
  /* USER CODE END RTOS_TIMERS */

  /* USER CODE BEGIN RTOS_QUEUES */
  /* add queues, ... */
  /* USER CODE END RTOS_QUEUES */

  /* Create the thread(s) */
  /* creation of Task1 */
  Task1Handle = osThreadNew(StartTask1, NULL, &Task1_attributes);

  /* creation of Task2 */
  Task2Handle = osThreadNew(StartTask2, NULL, &Task2_attributes);

  /* creation of Task3 */
  Task3Handle = osThreadNew(StartTask3, NULL, &Task3_attributes);

  /* USER CODE BEGIN RTOS_THREADS */
  /* add threads, ... */
  /* USER CODE END RTOS_THREADS */

  /* USER CODE BEGIN RTOS_EVENTS */
  /* add events, ... */
  /* USER CODE END RTOS_EVENTS */

  /* Start scheduler */
  osKernelStart();

  /* We should never get here as control is now taken by the scheduler */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
        //HAL_UART_Transmit(&huart2, dataTask1, 7, 1000);
        //HAL_Delay(1000);


  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
```

```
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 180;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Activate the Over-Drive mode
  */
  if (HAL_PWREx_EnableOverDrive() != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
```

```
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

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

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5, GPIO_PIN_RESET);

  /*Configure GPIO pins : PB3 PB4 PB5 */
  GPIO_InitStruct.Pin = GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */
int phase1 = 10000;
int phase2 = 2000;
int phase3 = 10000;
int phase4 = 3000;

GPIO_PinState sensorState = HAL_GPIO_ReadPin(GPIOD, GPIO_PIN_12);

#define PASSWORD_LENGTH 6
```

```c
const char correctPassword[PASSWORD_LENGTH + 1] = "123456"; // Correct
password, plus null terminator
int systemArmed = 0;
char armedMessage[] = "ARMED";
char desarmedMessage[] = "DESARMED";


int checkPassword(void) {

    char enteredPassword[PASSWORD_LENGTH + 1] = {0};
    int count = 0;

    while (1) {
        char key = Get_Key(); // Get one character from keypad

        if (key >= '0' && key <= '9') { // Check if the key is a digit
            if (count < PASSWORD_LENGTH) {
                enteredPassword[count++] = key; // Store the digit
            }
        } else if (key == '*') { // Check if the '*' is pressed
            if (count == PASSWORD_LENGTH) { // Ensure we have the right
number of digits
                enteredPassword[count] = '\0'; // Null-terminate the string
                // Check if the entered password is correct
                if (strcmp(enteredPassword, correctPassword) == 0) {
                    // Correct password

                  return 1;
                } else {
                    // Incorrect password

                  return 0;
                }
            }
            // Reset for next attempt
            memset(enteredPassword, 0, PASSWORD_LENGTH + 1);
            count = 0;
        }
        HAL_Delay(200); // Simple debounce delay
    }
}
/* USER CODE END 4 */

/* USER CODE BEGIN Header_StartTask1 */
/**
  * @brief  Function implementing the Task1 thread.
  * @param  argument: Not used
  * @retval None
  */
/* USER CODE END Header_StartTask1 */
/* handles motion sensor and buzzer */
void StartTask1(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
```

```
        /* Motion detected */
        if(sensorState == GPIO_PIN_SET){
              /*
               * Toggle Green LED on, Toggle red LED off
               * send "ARMED" message to screen
               * turn on buzzer
               * */
              systemArmed = 1;
              HAL_GPIO_WritePin(GPIOD, GPIO_PIN_0, GPIO_PIN_SET); // Set PD0
high for red LED
              HAL_GPIO_WritePin(GPIOD, GPIO_PIN_1, GPIO_PIN_SET); // Set PD1
high for green LED
              HAL_GPIO_WritePin(GPIOD, GPIO_PIN_15, GPIO_PIN_SET); // Set PD0
high for Buzzer
              //send Armed message to screen
              /* Check for right password */
              while(1){
                    if(checkPassword() == 1){
                          break;
                    }else{
                          delay(1000);
                    }
              }

        }
  }
  /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_StartTask2 */
/**
* @brief Function implementing the Task2 thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartTask2 */
/* Handles screen */
void StartTask2(void *argument)
{
  /* USER CODE BEGIN StartTask2 */
  /* Infinite loop */
  for(;;)
  {
        if(systemArmed == 1){
              SSD1306_Puts (armedMessage, &Font_16x26, 1);
        }else if(systemArmed == 0){
              SSD1306_Puts (desarmedMessage, &Font_16x26, 1);
        }else{
              //to be completed
        }

  }
  /* USER CODE END StartTask2 */
}

/* USER CODE BEGIN Header_StartTask3 */
/**
```

```c
* @brief Function implementing the Task3 thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartTask3 */
void StartTask3(void *argument)
{
  /* USER CODE BEGIN StartTask3 */
  /* Infinite loop */
  for(;;)
  {
        HAL_UART_Transmit(&huart2, dataTask3, sizeof(dataTask3), 1000);
        osDelay(phase1);
        osDelay(phase2);
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_3);
        osDelay(phase3);
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_3);
        osDelay(phase2);
  }
  /* USER CODE END StartTask3 */
}

/**
  * @brief  Period elapsed callback in non blocking mode
  * @note   This function is called  when TIM6 interrupt took place, inside
  * HAL_TIM_IRQHandler(). It makes a direct call to HAL_IncTick() to
increment
  * a global variable "uwTick" used as application time base.
  * @param  htim : TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM6) {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */

  /* USER CODE END Callback 1 */
}

/**
  * @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 */
```