Project 3 - Remember It! The Memory Game

Aaron Ahmadyar and Luis D. Garcia

CPE 316 - Section: 02 - Spring 2023

June 11, 2023

Dr. Paul Hummel

1 Behavior Description

"Remember It! The Memory Game" is a game designed for the Nucleo-L476RG STM32 board, featuring a joystick with x-y axis and a button, along with a 320x240 ILI9431 touch LCD screen connected via SPI1. The game utilizes a finite state machine with states including IDLE, Generate, Display, Read, Fail, and Pass.

The objective of the game is to reach the highest level possible, which is level 50. At the beginning, the user initiates the game by pressing the joystick. The game then enters the Generate state, where it randomly generates a pattern of blue rectangles using the RNG (Random Number Generator) of the STM32 board. These blue rectangles can appear at four different positions: top, bottom, left, or right of the screen.

After generating the pattern, the game transitions to the Display state, where it shows the blue rectangles on the LCD screen. The user needs to carefully observe and memorize the positions of the displayed rectangles. Once all the rectangles have been displayed, the game moves to the Read state.

In the Read state, the user is prompted to replicate the pattern by moving the joystick in the same direction and order as the rectangles appeared. The joystick's x-y axis is used to capture the user's input. If the user successfully reproduces the pattern, the game transitions to the Pass state, indicating that the user has progressed to the next level. In the Pass state, a green pass screen is displayed.

However, if the user fails to reproduce the pattern correctly, the game transitions to the Fail state, where a red fail screen is shown. The user can then choose to restart the game or exit.

As the user progresses to higher levels, the number of rectangles in the pattern increases, challenging the user's memory capabilities. The game continues until the user reaches level 50 or chooses to exit.

In summary, "Remember It! The Memory Game" is a game that tests the player's memory by displaying a sequence of randomly positioned blue rectangles, which the player must replicate by moving the joystick in the correct order. The game features a finite state machine, the Nucleo-L476RG STM32 board, a joystick, and a touch LCD screen.

2 System Specifications

2.1 Remember It! System Specifications

Microcontroller Unit (MCU)	STM32 Nucleo-L476RG
Processor Core	ARM Cortex-M4
Clock Speed	80 MHz
Flash Memory	1 MB
RAM	128 KB
Display	ILI9341 320x240 TFT LCD (3.3V)
Size	2.8 inches
Interface	SPI1
Touch Capability	Not specified
Input Devices	Joystick (3.3V)
Joystick X-Axis	ADC1
Joystick Y-Axis	ADC2
Joystick Button	GPIO (Input)
Communication	SPI
Power	Micro USB A (3.3V)
Frequency	80 MHz
Random Number Generator	STM32 RNG

Table 1: Remember It! System Specifications

3 System Schematic

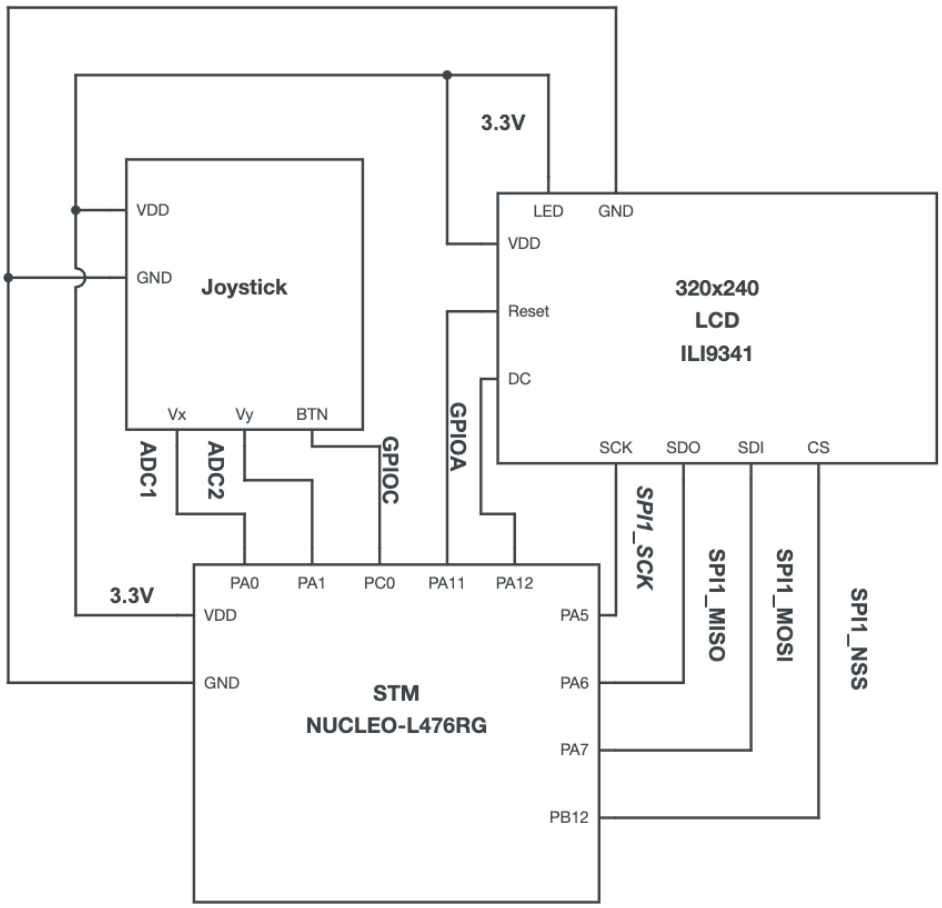


Figure 1: Remember It! Schematic of MCU and Peripherals.

Figure 1 illustrates the interfacing of the STM's Nucleo-L476RG with it's internal ADC1, ADC2, and SPI1 being used to interface with the Joystick and the 320x240 LCD ILI9341 display. ADC1 and ADC2 were the Joystick to determine the X and Y direcitons, while SPI1 was used to output the rectangles needed for the game to the LCD. Notice, however the configuration of the internal RNG of the Nucleo-L476RG is not shown due to being an internal hardware component, but it was utilized to randomize which rectangles the user has to memorize.

4 Software Architecture

4.1 Remember It! FSM

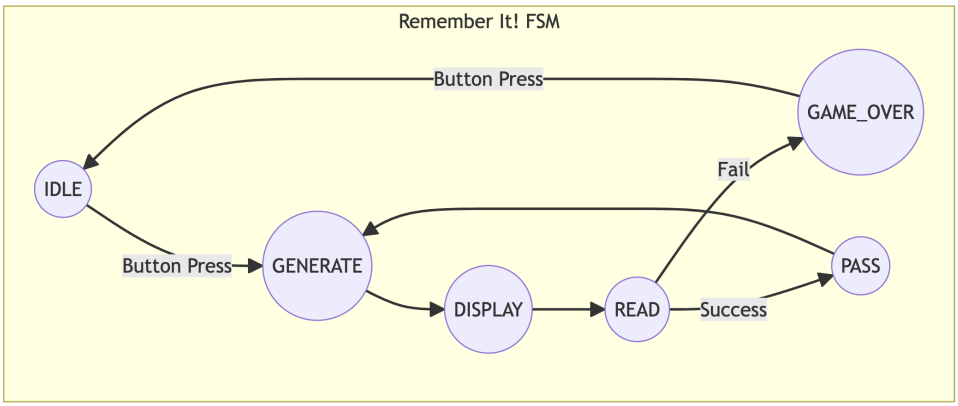


Figure 2: ADC12 Init Part 1

The FSM logic in the "Remember It" program controls the flow of the game and how it interacts with the ILI9341 LCD to display generated rectangles. The program starts in the IDLE state, waiting for a button press to initiate the game. Once the button is pressed, it transitions to the GENERATE state, where a random direction is generated and added to the cache. After that, it moves to the DISPLAY state.

In the DISPLAY state, the program iterates through the cache and displays each direction as a rectangle on the ILI9341 LCD. It briefly shows the rectangle for a period of time, then clears the screen, and moves on to the next direction. This process repeats until all directions in the cache have been displayed.

Next, the program transitions to the READ state, where it waits for the user to input the directions they observed. The user input is read from the joystick, and each inputted direction is displayed briefly on the LCD. If the input matches the corresponding direction in the cache, the program continues to the next input. However, if an incorrect direction is entered, the program transitions to the GAME_OVER state.

In the GAME_OVER state, a red screen is displayed on the ILI9341 LCD to indicate the game over status. The program then returns to the IDLE state, where it waits for a button press to restart the game.

If the user successfully inputs all the correct directions, the program transitions to the PASS state. In the PASS state, a green screen is displayed on the LCD to indicate success. If the maximum count of directions (50 in this case) is reached, the program resets the count and returns to the IDLE state. Otherwise, it goes back to the GENERATE state to generate the next random direction and continue the game.

The FSM logic controls the flow between these states, providing a game loop that generates and displays directions, accepts user input, and determines whether the game is over or the level is passed. The ILI9341 LCD is utilized to visually present the generated rectangles to the player, enhancing the interactive experience of the "Remember It" game.

4.2 Remember It! Flowchart

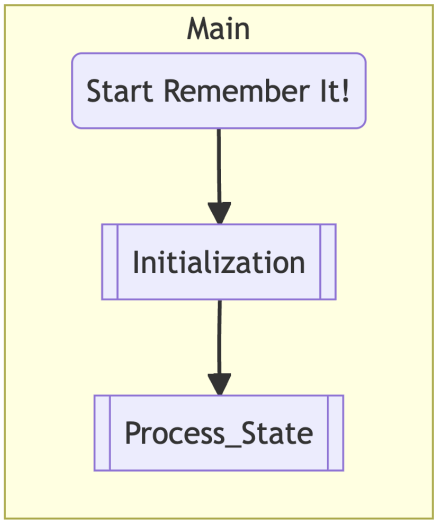


Figure 3: Remember It! Flowchart of Main

The Main subgraph represents the main program flow of the "Remember It!" application. It starts by initializing the application and proceeds to the Setup phase for initialization tasks. After Setup, it enters the Process_State subgraph.

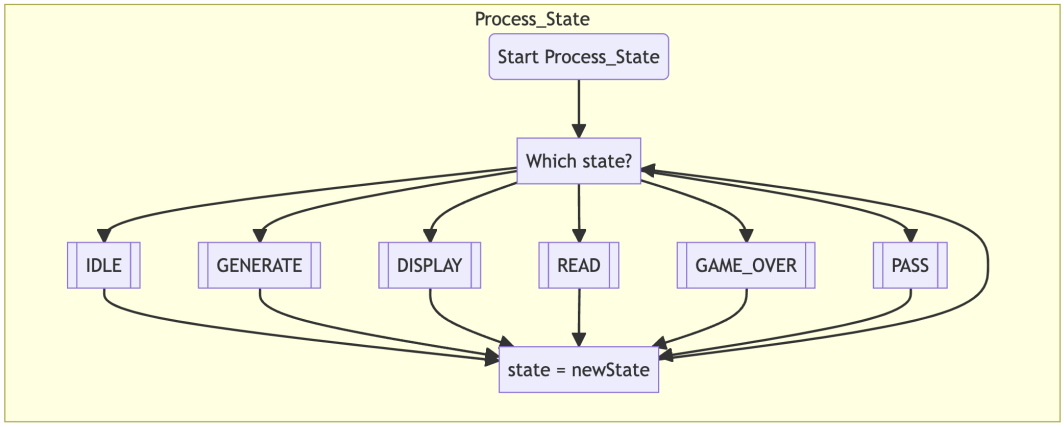


Figure 4: Remember It! Flowchart of Process_State

The Process_State subgraph represents the processing of different states in the application. It starts by querying which state to process. Depending on the queried state, it transitions to the corresponding state subgraph (IDLE, GENERATE, DISPLAY, READ, GAME_OVER, or PASS) to perform specific actions. After updating the state, it returns to the Query phase to check for the next state to process. This loop continues until the program execution is complete.

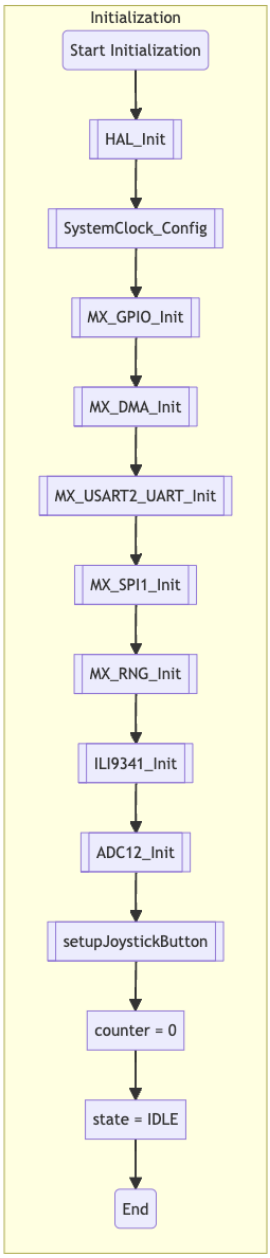


Figure 5: Remember It! Flowchart of Initialization

The Initialization subgraph represents the initialization process of various components in the system. It starts with the Start node and proceeds to perform the following tasks in sequence: HAL initialization, system clock configuration, GPIO initialization, DMA initialization, UART initialization, SPI initialization, RNG initialization, ILI9341 display initialization, ADC12 initialization, and setup of the joystick button. Each step sets up a specific component or module to ensure proper functioning of the system.

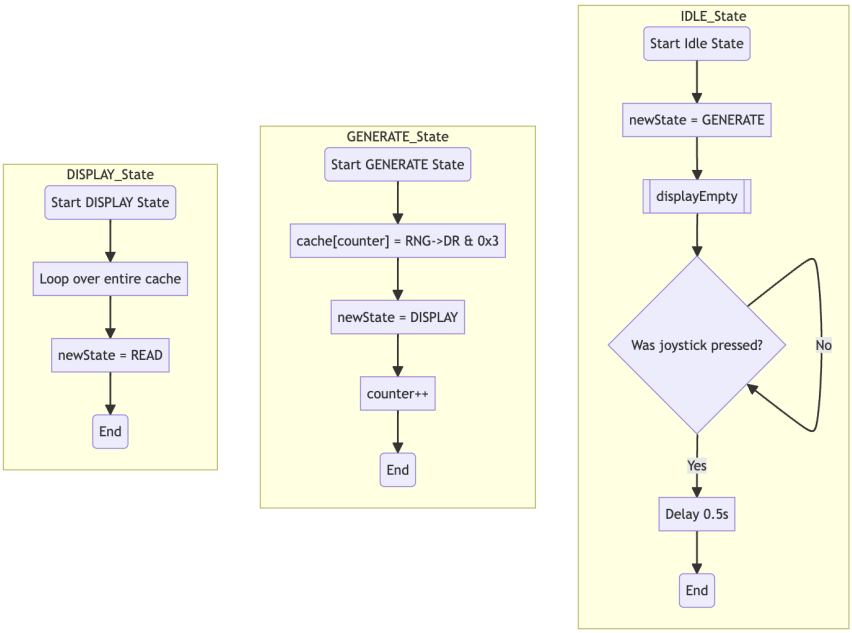


Figure 6: Remember It! Flowcharts of Display, Genereate, and IDLE States

The DISPLAY_State subgraph represents the display state of the program. It starts by initializing the state and then enters a loop that iterates over the entire cache. After each iteration, it transitions to the READ state. Eventually, it reaches the end. The GENERATE_State subgraph represents the generate state of the program. It starts by initializing the state and then updates the cache based on a random number generation. After updating the cache, it transitions to the DISPLAY state and increments the counter. Finally, it reaches the end. The IDLE_State subgraph represents the idle state of the program. It starts by initializing the state and transitions to the GENERATE state. If the joystick was not pressed, it checks again. If the joystick was pressed, it introduces a delay of 0.5 seconds and reaches the end.

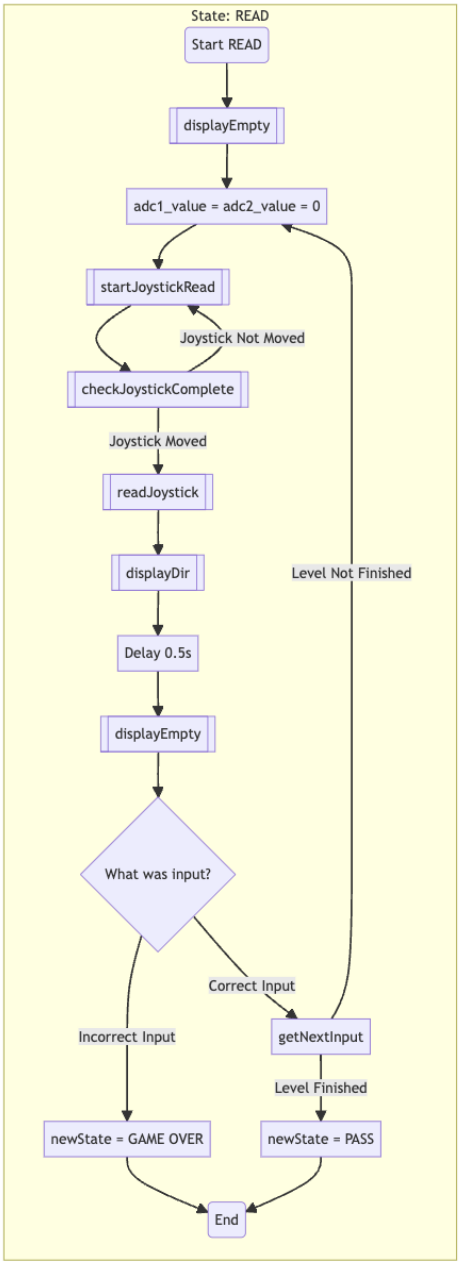


Figure 7: Remember It! Flowchart of Read State

The READ state in the flowchart represents a stage in the program where the user's input is read and compared to the expected solution. It starts by clearing the display, resetting the ADC values, and initiating the joystick reading. If the joystick is not moved, it continues to wait for the movement. Once a movement is detected, it displays the corresponding direction, waits for a delay of 0.5 seconds, clears the display, and proceeds to compare the input. If the input is incorrect, it transitions to the GAME_OVER state, indicating the end of the game. If the input is correct and the level is finished, it transitions to the PASS state. If the level is not finished, it resets the ADC values to read the next input.

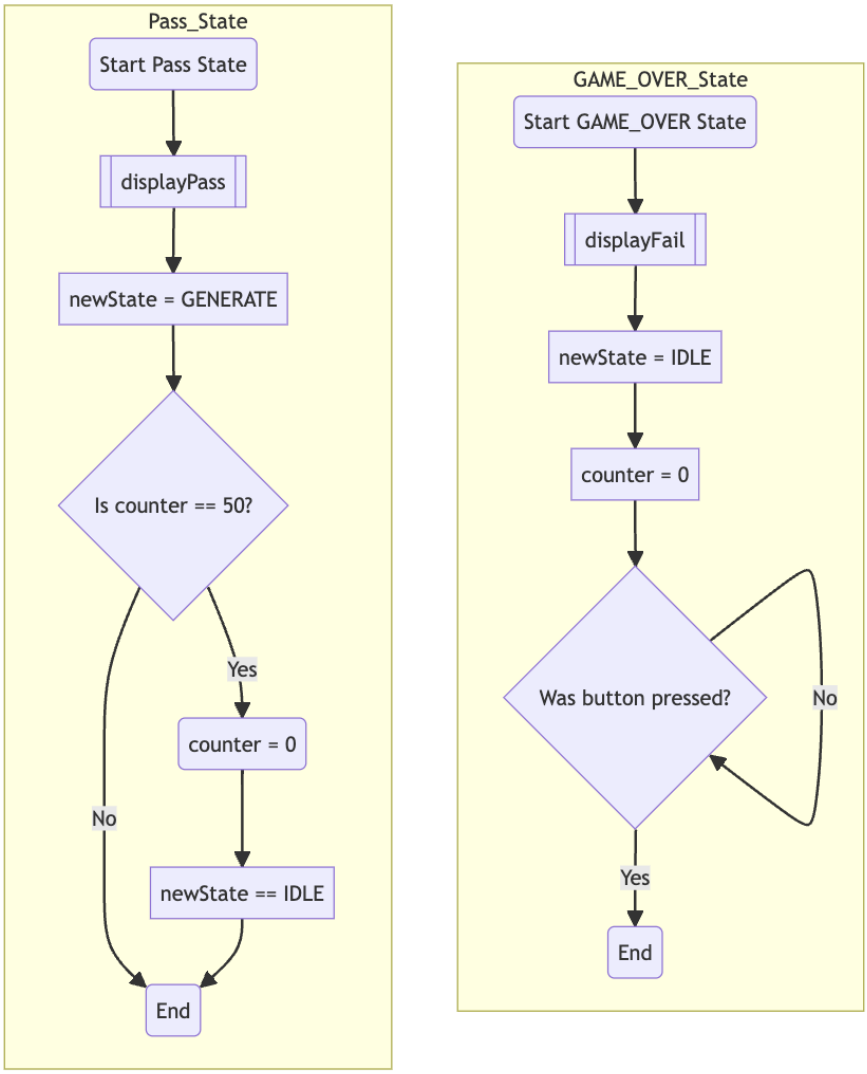


Figure 8: Remember It! Flowcharts of Pass and Game_Over States

The Pass_State subgraph represents the state when the game is passed successfully. It starts by initializing the state and displays the pass screen. Then, it transitions to the GENERATE state. It checks if the counter is equal to 50. If it's not, it reaches the end. If the counter is 50, it clears the counter and transitions to the IDLE state, reaching the end. The GAME_OVER_State subgraph represents the state when the game is over. It starts by initializing the state and displays the fail screen. Then, it transitions to the IDLE state. It clears the counter and checks if the button was pressed. If the button was not pressed, it continues checking. If the button was pressed, it reaches the end.

4.3 ADC Functions

4.3.1 ADC Init

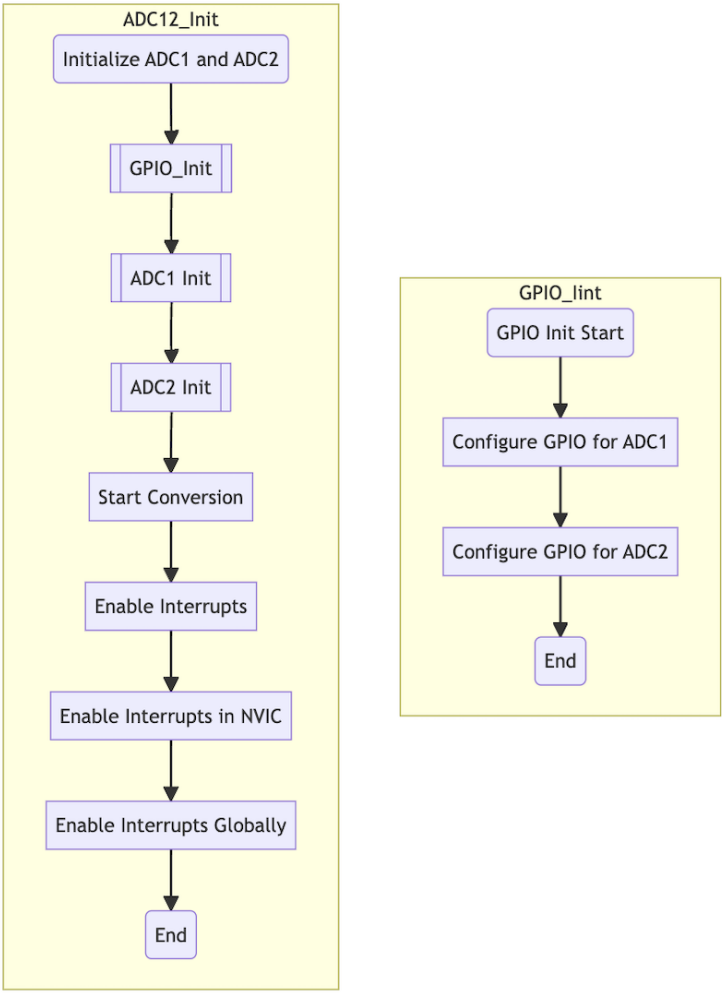


Figure 9: ADC12 Init Flowcharts for ADC12_Init and GPIO_Init

The first subgraph, ADC12_Init, represents the initialization process for ADC1 and ADC2. It starts with the initialization of GPIO using the GPIO_Init function. Then, it proceeds to initialize ADC1 and ADC2. After initializing the ADCs, the conversion is started, interrupts are enabled, and relevant configurations in the NVIC (Nested Vectored Interrupt Controller) are enabled. Finally, interrupts are globally enabled, and the initialization process concludes at the end. The second subgraph, GPIO_Init, represents the initialization process for GPIO (General Purpose Input/Output) configuration. It starts by configuring the GPIO pins for ADC1, followed by configuring the GPIO pins for ADC2. The initialization process concludes at the end.

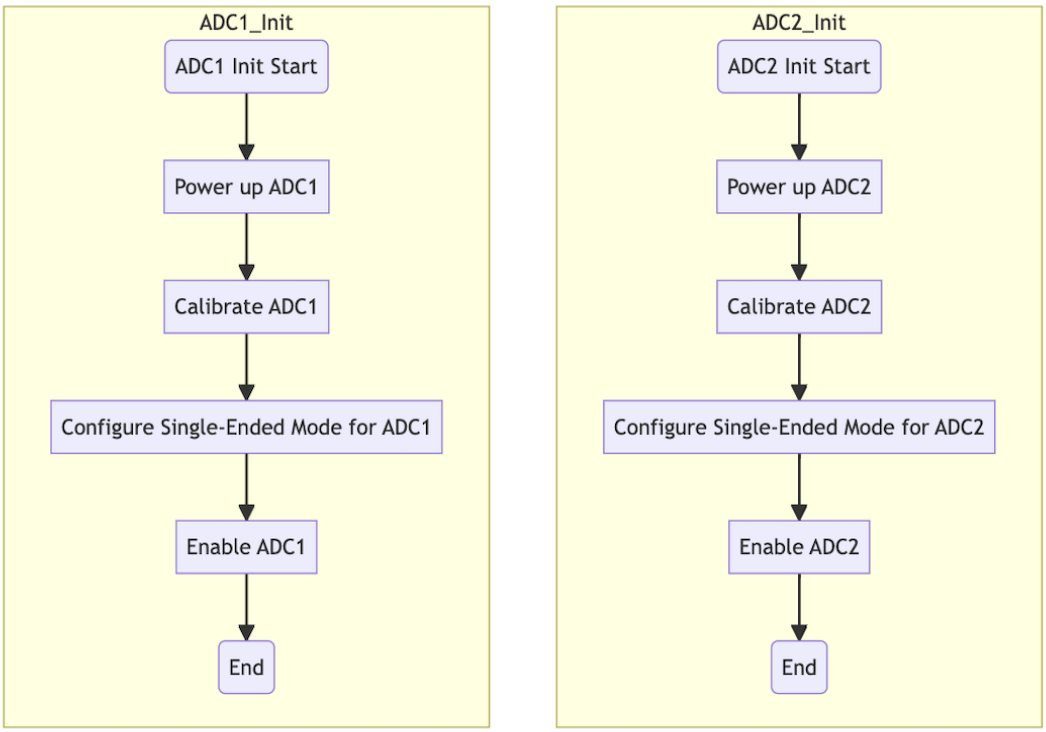


Figure 10: ADC12 Init Flowcharts for ADC1_Init and ADC2_Init

The first subgraph, ADC2_Init, represents the initialization process for ADC2. It starts by powering up ADC2, followed by calibrating it to ensure accurate readings. Then, it configures ADC2 to operate in single-ended mode and enables ADC2 for operation. The initialization process concludes at the end. The second subgraph, ADC1_Init, represents the initialization process for ADC1. Similar to ADC2_Init, it begins by powering up ADC1, followed by calibrating it. Then, it configures ADC1 to operate in single-ended mode and enables ADC1. The initialization process for ADC1 also concludes at the end.

4.3.2 ADC IRQ

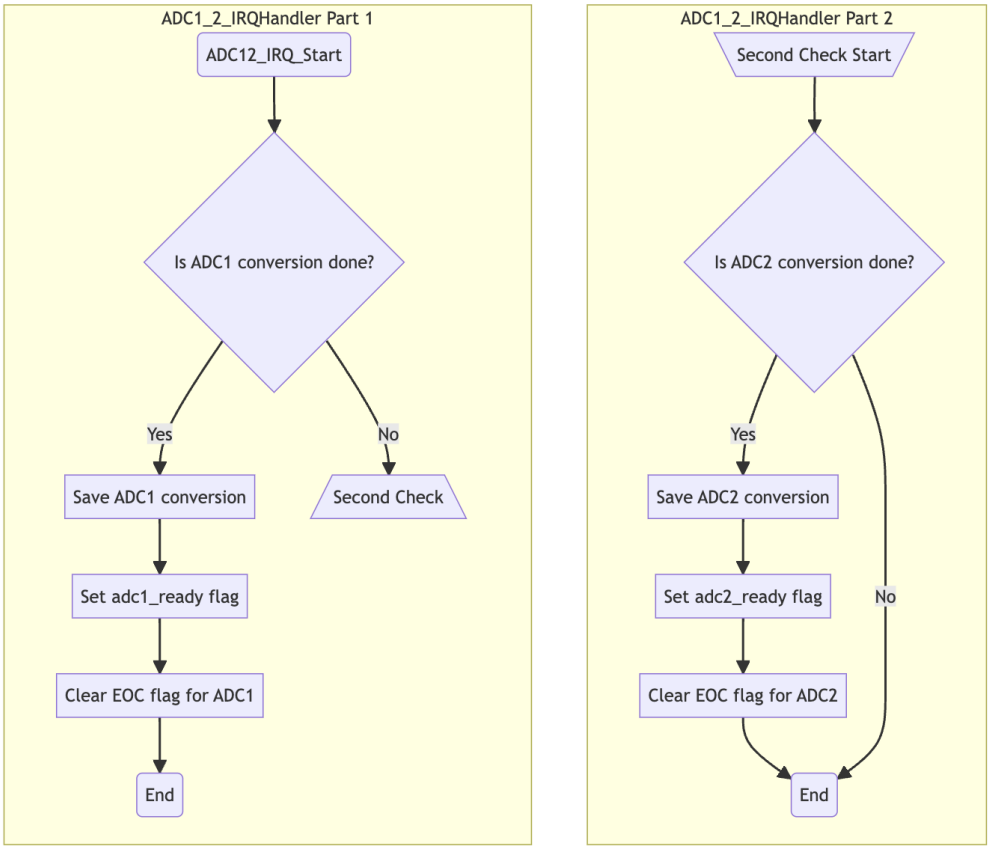


Figure 11: ADC12 IRQ Flowcharts

The first subgraph, ADC12_IRQHandler Part 1, represents the first part of the ADC1 and ADC2 interrupt handler. It starts with the ADC12_IRQ_Start and checks if the ADC1 conversion is done. If the conversion is done, the ADC1 result is saved, and the adc1_ready flag is set. The EOC (End of Conversion) flag for ADC1 is then cleared. If the ADC1 conversion is not done, it proceeds to the second check. Finally, the process concludes at the End1 node. The second subgraph, ADC12_IRQHandler Part 2, represents the second part of the ADC1 and ADC2 interrupt handler. It starts with the Second Check Start node and checks if the ADC2 conversion is done. If the conversion is done, the ADC2 result is saved, and the adc2_ready flag is set. The EOC flag for ADC2 is then cleared. If the ADC2 conversion is not done, the process also concludes at the End node.

4.4 LCD_Methods Functions

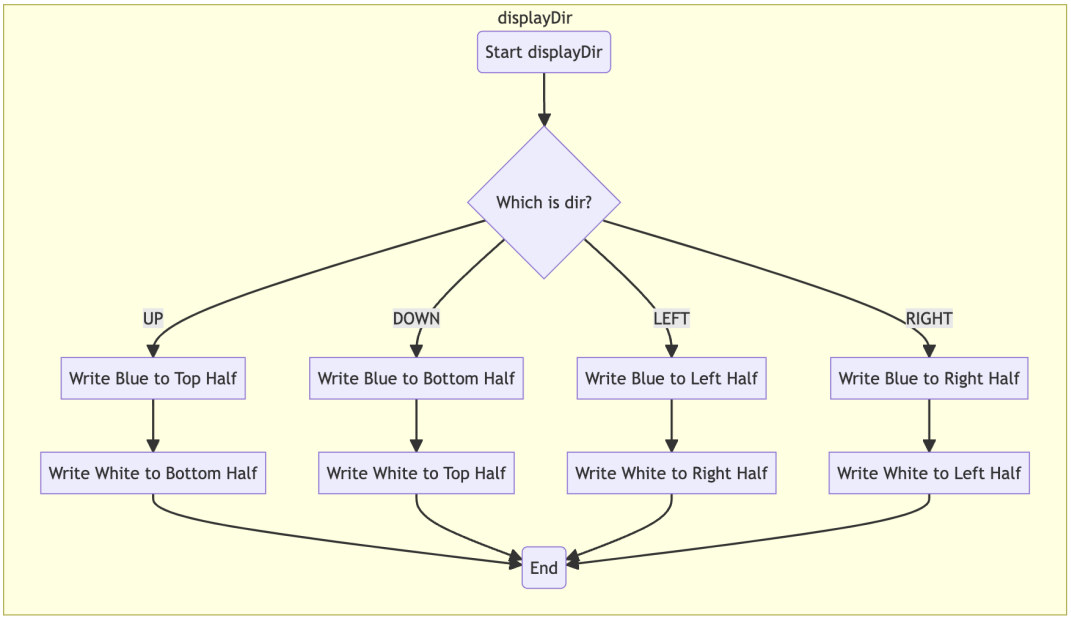


Figure 12: LCD_Methods displayDir Flowchart

This subgraph represents the process of displaying a direction on the screen. It starts by determining the direction to be displayed. Based on the direction, the corresponding action is taken: writing blue color to the specified half of the screen (top, bottom, left, or right). After writing the blue color, the process continues to write white color to the complementary half of the screen. Finally, the process reaches its end.

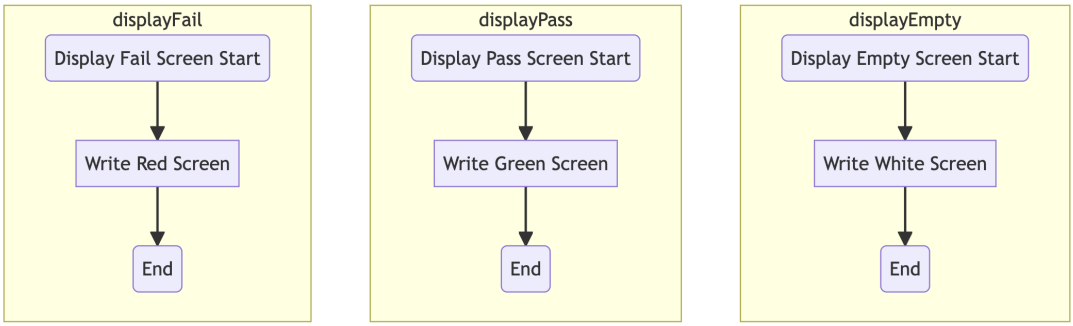


Figure 13: LCD_Methods displayFall, displayPass and displayEmpty Flowcharts

The first subgraph represents the process of displaying the fail screen. It starts by initiating the display of the fail screen, which is represented by a red screen. Once the red screen is written, the process reaches its end. The second subgraph represents the process of displaying the pass screen. It begins by initiating the display of the pass screen, which is represented by a green screen. After writing the green screen, the process reaches its end. The third subgraph represents the process of displaying the empty screen. It starts by initiating the display of the empty screen, which is represented by a white screen. Once the white screen is written, the process reaches its end.

4.5 Joystick Functions

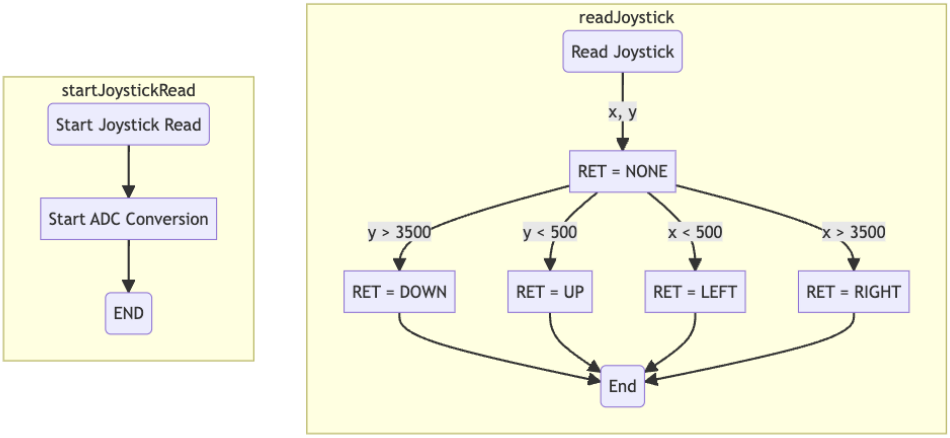


Figure 14: Start and Read Functions for the Joystick

The first subgraph represents the process of starting the joystick read. It initiates the ADC conversion to read the joystick values and indicates the end of the process. The second subgraph represents the process of reading the joystick's x and y values. Based on the values, it determines the joystick's direction and sets the corresponding return value (RET). If the y value is above 3500, the direction is DOWN; if it's below 500, the direction is UP; if the x value is below 500, the direction is LEFT; and if it's above 3500, the direction is RIGHT.

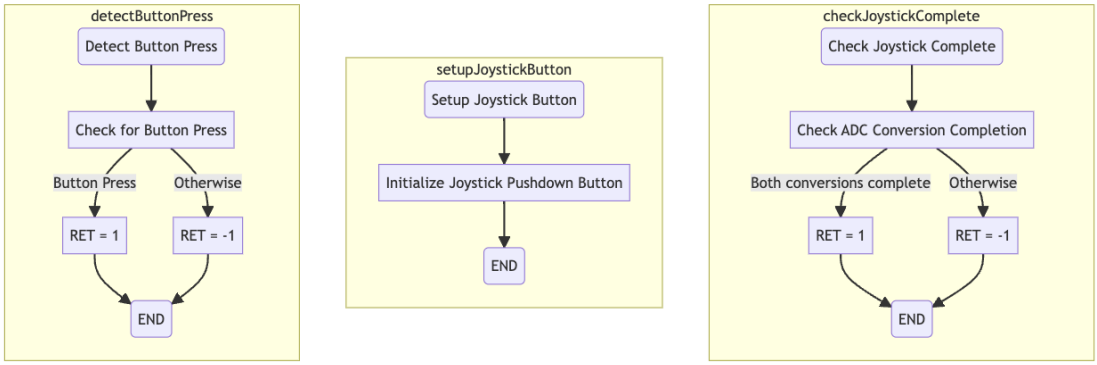


Figure 15: Button Press, Setup, and Completion Check Functions

The first subgraph represents the process of detecting a button press. It checks if a button press has occurred. If a button press is detected, it sets the return value (RET) to 1. Otherwise, it sets the return value to -1. The second subgraph represents the process of checking if the joystick conversion is complete. It verifies if both ADC conversions have finished. If both conversions are complete, it sets the return value (RET) to 1. Otherwise, it sets the return value to -1. The third subgraph represents the process of setting up the joystick button. It initializes the joystick pushdown button, ensuring it is ready for use.

5 Power Calculations

Device	Power Consumption (mW)
Display (ILI9341 TFT LCD)	198
Joystick	153

Table 2: Power Consumption of Devices

The power consumption values listed in Table 2 were obtained from the datasheets provided by the respective manufacturers [2, 3].

To calculate the average power consumption for the "Remember It!" system, we sum up the power consumption of all devices and divide it by the number of devices:

$$\text{Power of Peripheral} = \text{Voltage}_{\text{Peripheral}} + \text{CurrentDrawn}_{\text{Peripheral}} \tag{1}$$

$$\text{Total Power Consumption} = \text{Power}_{\text{Display}} + \text{Power}_{\text{Joystick}} \tag{2}$$

$$\text{Average Power Consumption} = \frac{\text{Total Power Consumption}}{\text{Number of Devices}} \tag{3}$$

In this case, the total power consumption is calculated as the sum of the individual power values, and the average power consumption is obtained by dividing the total power consumption by the number of devices (in this case, 5).

$$\text{Power}_{\text{Display}} = 0.06\text{A} \times 3.3\text{V} = 0.198\text{W} = 198\text{mW}$$

$$\text{Power}_{\text{Joystick}} = 0.051\text{A} \times 3.3\text{V} = 0.153\text{W} = 153\text{mW}$$

$$\text{Total Power Consumption} = 198\,\text{mW} + 153\text{mW} = 351\text{mW}$$

$$\text{Average Power Consumption} = \frac{351\text{mW}}{2} = 175.5\text{mW}$$

Answer: The average power consumption for the "Remember It!" system is approximately 176 mW.

6 User Manual: Remember It! The Memory Game

6.1 Introduction

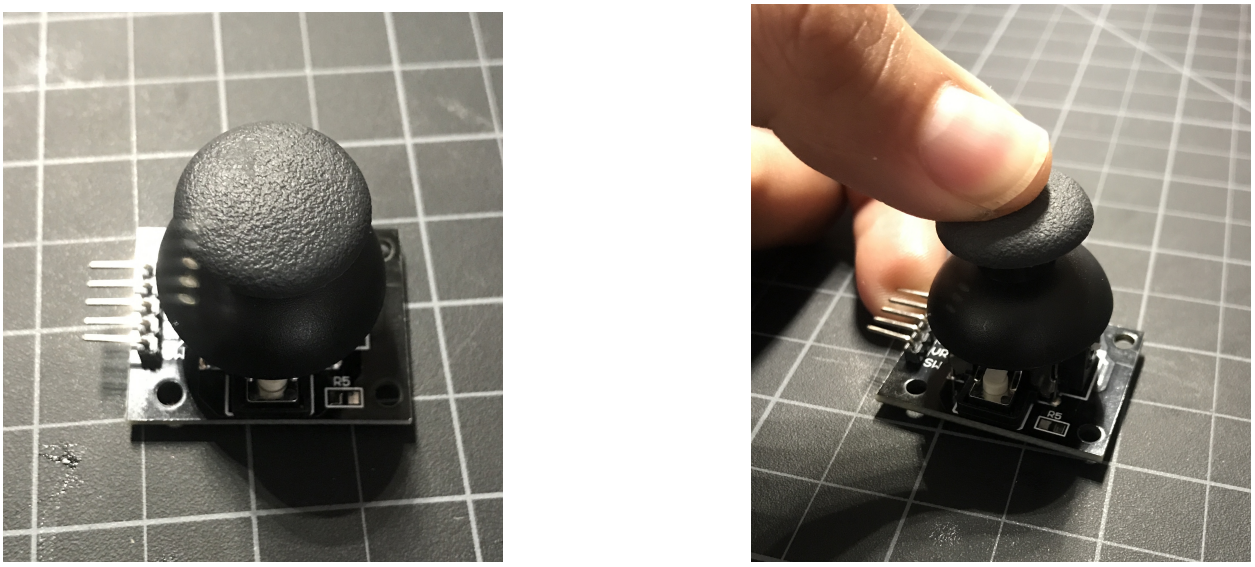
"Remember It! The Memory Game" is an engaging game that tests your memory skills. The game utilizes the Nucleo-L476RG STM32 board, a joystick with x-y axis and a button, and a 320x240 ILI9431 touch LCD screen. This user manual provides clear instructions on how to use the device and play the game effectively.

6.2 Getting Started

1. Connect the Nucleo-L476RG STM32 board to a power source using a suitable USB cable.
2. Ensure that the joystick and LCD screen are properly connected to the board.
3. Power on the board.

6.3 Game Controls

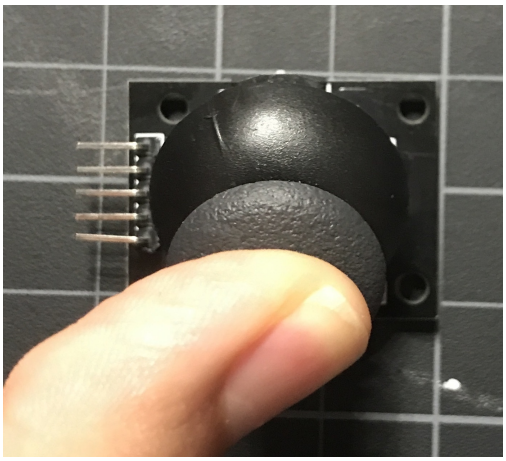
1. Joystick: Use the joystick's x-y axis to navigate and replicate the pattern. Press the Joystick button to start the game and progress through the screens.



(a) Joystick used in the Game

(b) Button Press of Joystick

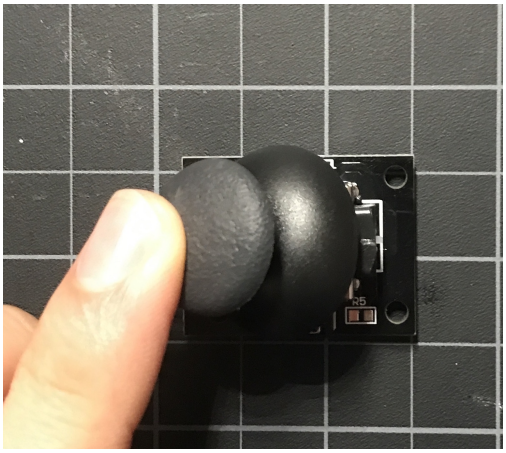
Figure 16: Joystick and Button Press of Joystick



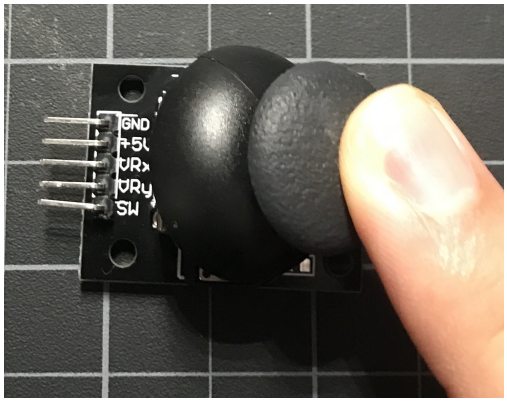
(a) Down Position of Joystick

(b) Up Position of Joystick

Figure 17: Down and Up Positions of the Joystick



(a) Left Position of Joystick



(b) Right Position of Joystick

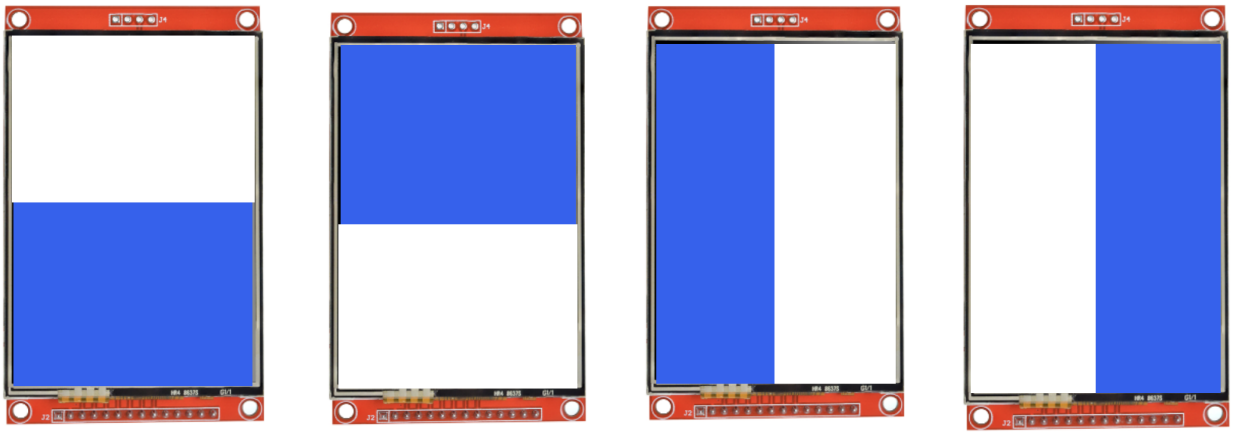
Figure 18: Left and Right Positions of the Joystick

6.4 Gameplay

1. Press the joystick button to start the game.
2. The game will display a blue rectangle at one of four possible positions: top, bottom, left, or right of the screen.
3. Memorize the position of the blue rectangle carefully.
4. After displaying the rectangle, the game will prompt you to replicate the pattern.
5. Use the joystick's x-y axis to move the cursor in the same direction and order as the rectangles appeared.
6. Be precise and accurate in reproducing the pattern.
7. If you successfully replicate the pattern, the game will display a green pass screen, indicating your progress to the next level.

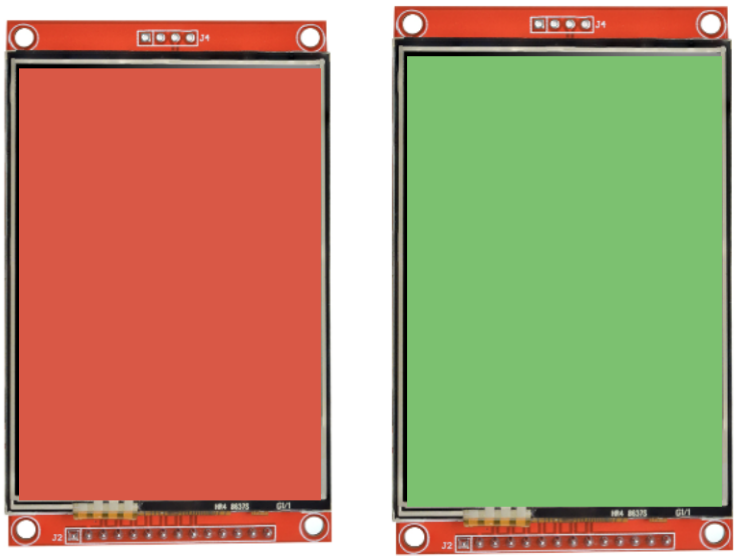
8. If you fail to replicate the pattern correctly, the game will display a red fail screen.
9. You can choose to restart the game or exit.
10. Demo Game Play

6.5 Gameplay Screens



(a) Bottom blue rectangle (b) Top blue rectangle (c) Left blue rectangle (d) Right blue rectangle

Figure 19: The four different blue rectangle configurations.



(a) Red Fail Screen (b) Green Pass Screen

Figure 20: Fail and Pass Screens

6.6 Scoring and Levels

1. The objective is to reach the highest level possible, which is level 50 where the game resets back to level 1.
2. Each level increases the number of rectangles in the pattern, making it more challenging.
3. Aim to achieve the highest score by accurately replicating the patterns.

6.7 Restarting or Exiting the Game

1. After reaching the fail or pass screen, you can choose to restart the game or exit.
2. Use the joystick to navigate through the options and press the button to select your choice.

6.8 Safety Instructions

1. The NUCLEO-L476RG STM32 board contains electronic components. Handle it with care and avoid subjecting it to extreme conditions.
2. When using the joystick, be gentle to prevent any damage.
3. Ensure that the board is properly connected and powered off when not in use.
4. Follow standard electrical safety guidelines of where the power supply that is supplying the STM32 board.

6.9 Troubleshooting

1. If you encounter any technical issues or difficulties with the game or device, refer to the manufacturer's documentation or seek technical assistance.

6.10 Conclusion

Congratulations on choosing "Remember It! The Memory Game"! Enjoy the game, challenge your memory skills, and strive for the highest level possible. Have fun and good luck!

Note: *This user manual provides a general overview of the game and device usage. For detailed technical information, refer to the respective documentation provided by the manufacturer or consult the user manual specific to your NUCLEO-L476RG STM32 board.*

7 Appendices

7.1 References

References

[1] STM32 Nucleo-L476RG Datasheet. Manufacturer: STMicroelectronics. Available online: https://www.st.com/resource/en/datasheet/stm32l476rg.pdf. Accessed: May 30, 2023.

[2] ILI9341 TFT LCD Datasheet. Manufacturer: ILITEK. Available online: http://www.lcdwiki.com/2.8inch_SPI_Module_ILI9341_SKU:MSP2807. Accessed: June 1, 2023.

[3] Joystick Datasheet. Manufacturer: Joystick Inc.

[4] SPI Datasheet. Manufacturer: SPI Corporation.

[5] STM32 RNG Datasheet. Manufacturer: STMicroelectronics.

[6] C. Corral, "STM32L476RG - Ultra-low-power with FPU Arm Cortex-M4 MCU 80 MHz with 1 Mbyte Flash, USB OTG, DFSDM, STM32L476RGT6TR," STMicroelectronics, May 2021. Available online: https://www.st.com/resource/en/datasheet/stm32l476rg.pdf. Accessed: May 30, 2023.

[7] "ST STM32L4x6 Reference Manual," ManualsLib. Available online: https://www.manualslib.com/manual/1317428/St-Stm32l4x6.html. Accessed: May 30, 2023.

[8] Nunocky, "Nucleo_L476RG_LCD_ILI9341," March 3, 2021. Available online: https://github.com/Nunocky/Nucleo_L476RG_LCD_ILI9341. Accessed: June 3, 2023.

[9] LCDWiki, "2.8inch SPI Module ILI9341 SKU:MSP2807." Available online: http://www.lcdwiki.com/2.8inch_SPI_Module_ILI9341_SKU:MSP2807. Accessed: June 1, 2023.

7.2 C Code

7.2.1 main.c

```
/*
 ******************************************************************************
 * @file           : main.c
 * @brief          : Main program body
 ******************************************************************************
 ******************************************************************************
 */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "ILI9341.h"
#include <stdint.h>
#include <stdlib.h>
#include "joystick.h"
#include "adc.h"

/* Private variables ---------------------------------------------------------*/
RNG_HandleTypeDef hrng;
SPI_HandleTypeDef hspi1;
DMA_HandleTypeDef hdma_spi1_tx;
UART_HandleTypeDef huart2;

// Define the ADC global variables
volatile uint16_t adc1_value = 0;
volatile uint8_t adc1_ready = 0;
volatile uint16_t adc2_value = 0;
volatile uint8_t adc2_ready = 0;

// State machine states
typedef enum {
    IDLE, GENERATE, DISPLAY, READ, GAME_OVER, PASS
} STATE;

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_SPI1_Init(void);
static void MX_RNG_Init(void);

// ADC interrupt handler
void ADC1_2_IRQHandler(void);



int main(void) {

    HAL_Init();
    SystemClock_Config();

    //touchscreen initialization
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART2_UART_Init();
    MX_SPI1_Init();
    MX_RNG_Init();
    ILI9341_Init();

    //joystick initialization
    ADC12_Init();
    setupJoystickButton();
```

```
    //keep track of state in state machine
    STATE state = IDLE;

    //valid directions for next level
    DIR options[4] = { UP, DOWN, LEFT, RIGHT };

    //keep track of solutions thus far
    DIR cache[50];

    //curent level or amount of blue flashes to memorize
    uint8_t cnt = 0;
    while (1) {

        //state machine
        switch (state) {

            //do nothing state, wait for button press
            case IDLE: {
                state = GENERATE;   //next state generate random direction
                displayEmpty(); //clear screen

                //wait for btn press to move one
                while (detectButtonPress() == -1);
                HAL_Delay(400);
                break;
            }

            //generate next random direction
            case GENERATE: {
                //get first 2 bits of RNG since only 4 options to choose from direction-wise
                cache[cnt] = options[RNG->DR & 0x3];
                state = DISPLAY;    //display pattern so far for user to mimic
                cnt++;
                break;
            }

            //display pattern to mimic
            case DISPLAY: {
                //display all paterns up to cnt
                for (uint8_t i = 0; i < cnt; i++) {
                    displayDir(cache[i]);
                    HAL_Delay(500);
                    displayEmpty();
                    HAL_Delay(50);
                }
                //go on to read state to read user input and check againt solution
                state = READ;
                break;
            }

            //read user input and compare to solution
            case READ: {
                displayEmpty(); //clear LCD

                //read # of user inputs equal to current level
                uint8_t i = 0;
                while (i < cnt) {
                    //reset joystick value
                    adc1_value = 1900;
                    adc2_value = 1900;  //reset add values

                    //wait for joystick joystick to be moved by user
                    while (readJoystick(adc1_value, adc2_value) == NONE) {
                        startJoystickRead();        //start adc conversion
                        while (!checkJoystickComplete(adc1_ready, adc2_ready)); //wait for both
    adc conversoin to be complete
                    }
                    //get joystick directoin if valid direction inputted
```

```
                    DIR dir = readJoystick(adc1_value, adc2_value);

                    //display input joystick direction
                    displayDir(dir);
                    HAL_Delay(500);
                    displayEmpty();

                    //if input not equal to answer game over
                    if (dir != cache[i]) {
                        state = GAME_OVER;
                        break;
                    }
                    //if correct joystick get next joystick input
                    else {
                        i++;

                        //if level finished exit state and get ready for next level
                        if (i == cnt) {
                            state = PASS;
                        }
                    }
                }
                break;
            } //end case

            //game over screen (displays red)
            case GAME_OVER: {
                displayFail();
                state = IDLE;       //go back to starting state
                cnt = 0;

                //wait for btn press to move one
                while (detectButtonPress() == -1)
                    ;
                HAL_Delay(200);
                break;
            }

            //pass level screen (displays green)
            case PASS: {
                displayPass();
                state = GENERATE;

                //reset if max count reached
                if (cnt == 50) {
                    cnt = 0;
                    state = IDLE;
                }
                HAL_Delay(500);
                break;
            }

            default:
                break;
        } //end switch
    }
}

void ADC1_2_IRQHandler(void) {
    //check if ADC1 conversion done
    if (ADC1->ISR & ADC_ISR_EOC) {
        // Save the digital conversion to a global variable
        adc1_value = ADC1->DR;
        // Set the global flag
        adc1_ready = 1;
        // Clear the end of conversion flag
        ADC1->ISR &= ~(ADC_ISR_EOC);
    }
```

```
    //check if ADC2 conversion done
    if (ADC2->ISR & ADC_ISR_EOC) {
        // Save the digital conversion of ADC2 to a global variable
        adc2_value = ADC2->DR;
        // Set the global flag for ADC2
        adc2_ready = 1;
        // Clear the end of conversion flag for ADC2
        ADC2->ISR &= ~(ADC_ISR_EOC);
    }
}

/**
 * @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
     */
    if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1)
            != HAL_OK) {
        Error_Handler();
    }

    /** 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 = 1;
    RCC_OscInitStruct.PLL.PLLN = 10;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
    RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
    RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != 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_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK) {
        Error_Handler();
    }
}

/**
 * @brief RNG Initialization Function
 * @param None
 * @retval None
 */
static void MX_RNG_Init(void) {
    /*
    Initialize random number generator generated from IOC
    */
```

```
    hrng.Instance = RNG;
    if (HAL_RNG_Init(&hrng) != HAL_OK) {
        Error_Handler();
    }
    /* USER CODE BEGIN RNG_Init 2 */

    /* USER CODE END RNG_Init 2 */

}

/**
 * @brief SPI1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_SPI1_Init(void) {

    /* USER CODE BEGIN SPI1_Init 0 */

    /* USER CODE END SPI1_Init 0 */

    /* USER CODE BEGIN SPI1_Init 1 */

    /* USER CODE END SPI1_Init 1 */
    /* SPI1 parameter configuration*/
    hspi1.Instance = SPI1;
    hspi1.Init.Mode = SPI_MODE_MASTER;
    hspi1.Init.Direction = SPI_DIRECTION_2LINES;
    hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
    hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
    hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
    hspi1.Init.NSS = SPI_NSS_SOFT;
    hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_4;
    hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
    hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
    hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
    hspi1.Init.CRCPolynomial = 7;
    hspi1.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
    hspi1.Init.NSSPMode = SPI_NSS_PULSE_DISABLE;
    if (HAL_SPI_Init(&hspi1) != HAL_OK) {
        Error_Handler();
    }
    /* USER CODE BEGIN SPI1_Init 2 */

    /* USER CODE END SPI1_Init 2 */

}

/**
 * @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;
    huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
    huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    if (HAL_UART_Init(&huart2) != HAL_OK) {
        Error_Handler();
    }
    /* USER CODE BEGIN USART2_Init 2 */

    /* USER CODE END USART2_Init 2 */

}

/**
 * Enable DMA controller clock
 */
static void MX_DMA_Init(void) {

    /* DMA controller clock enable */
    __HAL_RCC_DMA1_CLK_ENABLE();

    /* DMA interrupt init */
    /* DMA1_Channel3_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(DMA1_Channel3_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(DMA1_Channel3_IRQn);

}

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

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

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(LCD_BACKLIGHT_GPIO_Port, LCD_BACKLIGHT_Pin,
            GPIO_PIN_RESET);

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(LCD_CS_GPIO_Port, LCD_CS_Pin, GPIO_PIN_SET);

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOA, LCD_RESET_Pin | LCD_DC_Pin, GPIO_PIN_RESET);

    /*Configure GPIO pin : B1_Pin */
    GPIO_InitStruct.Pin = B1_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

    /*Configure GPIO pin : LCD_BACKLIGHT_Pin */
    GPIO_InitStruct.Pin = LCD_BACKLIGHT_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(LCD_BACKLIGHT_GPIO_Port, &GPIO_InitStruct);

    /*Configure GPIO pin : LCD_CS_Pin */
    GPIO_InitStruct.Pin = LCD_CS_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
```

```
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(LCD_CS_GPIO_Port, &GPIO_InitStruct);

    /*Configure GPIO pins : LCD_RESET_Pin LCD_DC_Pin */
    GPIO_InitStruct.Pin = LCD_RESET_Pin | LCD_DC_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */

/* 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 */
```

7.2.2 LCD_Methods.h

```
#ifndef LCD_METHODS_H
#define LCD_METHODS_H

#include "joystick.h"

#define WHITE 0
#define RED 1
#define GREEN 2
#define BLUE 3

void displayDir(DIR);
void displayPass(void);
void displayFail(void);
void displayEmpty(void);

#endif
```

7.2.3 LCD_Methods.c

```
#include "LCD_Methods.h"
#include "ILI9341.h"
#include "main.h"
#include <stdint.h>

//lookup tables from library representing LCD screen grid
extern const uint8_t defaultData_white[320 * 240 * 2];
extern const uint8_t defaultData_green[320 * 240 * 2];
extern const uint8_t defaultData_red[320 * 240 * 2];
extern const uint8_t defaultData_blue[320 * 240 * 2];


void displayDir(DIR dir) {
    /*
     Displays direction on LCD in BLUE using library functions.
     Pixel grid mapped 0 to 320 (bottom to top) on y axis and 0 to 240 on x-axis (left to right)
     :param dir: direction to display
     */

    switch (dir) {
        //display blue on top half of LCD
        case UP: {
            //write blue to top half
            ILI9341_setColumnAddress(160, 320);
            ILI9341_setPageAddress(0, 240);
            ILI9341_MemoryWrite(defaultData_blue, 320 * 240 * 2);

            //write white to bottom half
            ILI9341_setColumnAddress(0, 160);
            ILI9341_setPageAddress(0, 240);
            ILI9341_MemoryWrite(defaultData_white, 320 * 240 * 2);
            break;
        }

        //displays blue on bottom half of LCD
        case DOWN: {
            //set blue on bottom half
            ILI9341_setColumnAddress(0, 160);
            ILI9341_setPageAddress(0, 240);
            ILI9341_MemoryWrite(defaultData_blue, 320 * 240 * 2);

            //display white on top half
            ILI9341_setColumnAddress(160, 320);
            ILI9341_setPageAddress(0, 240);
            ILI9341_MemoryWrite(defaultData_white, 320 * 240 * 2);
            break;
        }

        //displays blue on left half
        case LEFT: {
            //display blue on left half
            ILI9341_setColumnAddress(0, 320);
            ILI9341_setPageAddress(0, 120);
            ILI9341_MemoryWrite(defaultData_blue, 320 * 240 * 2);

            //displays white on right half
            ILI9341_setColumnAddress(0, 320);
            ILI9341_setPageAddress(120, 240);
            ILI9341_MemoryWrite(defaultData_white, 320 * 240 * 2);
            break;
        }

        //displays blue on right half
        case RIGHT: {
```

```
            //display blue on right half
            ILI9341_setColumnAddress(0, 320);
            ILI9341_setPageAddress(120, 240);
            ILI9341_MemoryWrite(defaultData_blue, 320 * 240 * 2);

            //displays white on left half
            ILI9341_setColumnAddress(0, 320);
            ILI9341_setPageAddress(0, 120);
            ILI9341_MemoryWrite(defaultData_white, 320 * 240 * 2);
            break;
        }
        default: break;
    }//end switch
}

void displayEmpty(void) {
    /*
     Displays Green Screen to indicate success
    */
    //display white screen on LCD
    ILI9341_setColumnAddress(0, 320);
    ILI9341_setPageAddress(0, 240);
    ILI9341_MemoryWrite(defaultData_white, 320 * 240 * 2);
}

void displayPass(void) {
    /*
     Displays Green Screen to indicate success
    */
    //display full screen of green on LCD
    ILI9341_setColumnAddress(0, 320);
    ILI9341_setPageAddress(0, 240);
    ILI9341_MemoryWrite(defaultData_green, 320 * 240 * 2);
}

void displayFail(void) {
    /*
     Displays Red Screen to indicate success
    */
    //display full screen of red on LCD
    ILI9341_setColumnAddress(0, 320);
    ILI9341_setPageAddress(0, 240);
    ILI9341_MemoryWrite(defaultData_red, 320 * 240 * 2);
}
```

7.2.4 joystick.h

```
#ifndef JOYSTICK_H
#define JOYSTICK_H

#include <stdint.h>

typedef enum {
    NONE, UP, DOWN, LEFT, RIGHT
} DIR;

DIR readJoystick(uint16_t, uint16_t);
void startJoystickRead(void);
int8_t checkJoystickComplete(uint16_t, uint16_t);
void setupJoystickButton(void);
int8_t detectButtonPress(void);

#endif
```

7.2.5 joystick.c

```
#include "joystick.h"
#include <stdint.h>
#include "main.h"

DIR readJoystick(uint16_t x, uint16_t y) {
    /*
    Takes joystick output and maps to directoin
    :param x: ADC reading from joystick on x-axis, range: (0 ~ 4000)
    :param y: ADC reading from joystick on y-axis, range: (0 ~ 4000)
    :return: enum type specifying direction of joystick
    */
    DIR ret = NONE;

    //interpert ADC12 data to get direction
    if (y > 3500) {
        ret = DOWN;
    } else if (y < 500) {
        ret = UP;
    } else if (x < 500) {
        ret = LEFT;
    } else if (x > 3500) {
        ret = RIGHT;
    }
    return ret;
}

void startJoystickRead(void) {
    /*
    Starts adc conversion to read joystick x and y values
    */
    //start adc1 and adc2 conversion
    ADC1->CR |= ADC_CR_ADSTART;
    ADC2->CR |= ADC_CR_ADSTART;
}

int8_t checkJoystickComplete(uint16_t adc1_ready, uint16_t adc2_ready) {
    /*
    Checks user defined flags to see if ADC covnersions complete
    :param adc1_ready: flag for adc1 conversion
    :param adc2_ready: flag for adc2 conversion
    :return: 1 if ready, -1 if not
    */
    //if both  conversoin complete return 1
    if (adc1_ready && adc2_ready) {
        adc1_ready = 0;
        adc2_ready = 0;
        return 1;
    }
    //return -1 otherwise
    return -1;
}

void setupJoystickButton(void) {
    /*
    Initialize joystick pushdown button
    */
    // Turn on the clock for GPIOC
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOCEN;

    // Set MODE to input
    GPIOC->MODER &= ~(GPIO_MODER_MODE0);

    // Pull up resistor so idles high and drops to 0 when button pressed
    GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPD0);
```

```
    GPIOC->PUPDR |= (1 << GPIO_PUPDR_PUPD0_Pos);
}

int8_t detectButtonPress(void) {
    /*
    Checks for button press
    :return: 1 if button press, otherwise -1
    */
    //if button press return 1
    if ((GPIOC->IDR  & GPIO_IDR_ID0) == 0) {
        return 1;

    //return -1 if no button press
    } else {
        return -1;
    }
}
```

7.2.6 adc.h

```
#ifndef __ADC_H
#define __ADC_H

#ifdef __cplusplus
extern "C" {
#endif

#define NUM_SAMPLES 0x14        // equals 20
#define DELAY_TIME  0xFB0       // equals 4000
#define MULTIPLER   0.795       // the needed m in y = mx + b
#define DIFF        23.5        // the needed b in y = mx + b
#define DELAY       0xF4240     // equals 1000000
#define TENS        0xA         // equals 10
#define UPPER_VOLT  0x3E8       // equal 1000
#define LOWER_VOLT  0x64        // equal 100

void ADC12_Init(void);

#ifdef __cplusplus
}
#endif

#endif /* __ADC_H */
```

7.2.7 adc.c

```
#include "adc.h"
#include "stm32l4xx_hal.h"

void ADC12_Init(void) {
    /*
    Initialized adc1 and adc2
    */
    // Enable GPIOA clock
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;

    // Configure GPIO for channel 5 on analog input pin (PA0) ADC1
    // Clear PA0
    GPIOA->MODER &= ~(GPIO_MODER_MODE0_Msk);
    // Set pin as analog mode
    GPIOA->MODER |= (GPIO_MODER_MODE0_0 | GPIO_MODER_MODE0_1);
    // Set analog switch
    GPIOA->ASCR |= GPIO_ASCR_ASC0;
    // Configure GPIO for channel 6 on analog input (PA1) ADC2
    // Clear PA1
    GPIOA->MODER &= ~(GPIO_MODER_MODE1_Msk);
    // set pin as analog mode
    GPIOA->MODER |= (GPIO_MODER_MODE1_0 | GPIO_MODER_MODE1_1);
    // set analog switch
    GPIOA->ASCR |= GPIO_ASCR_ASC1;

    // Enable clock for ADC
    RCC->AHB2ENR |= RCC_AHB2ENR_ADCEN;
    //ADC will run at the same speed as CPU (CLK/1), Prescaler = 1
    ADC123_COMMON->CCR = (1 << ADC_CCR_CKMODE_Pos);

    /*--------------- Set up ADC1 Here ------------*/
    //Power up the ADC and voltage regulator
    ADC1->CR &= ~(ADC_CR_DEEPPWD);
    ADC1->CR |= (ADC_CR_ADVREGEN);
    // delay for voltage regulation startup time ~ 20 us
    for (uint32_t i = 0; i < DELAY_TIME; i++)
        ;
    // Calibrate the ADC1 and ensure ADC1 is disabled and single-ended mode
    ADC1->CR &= ~(ADC_CR_ADEN | ADC_CR_ADCALDIF);
    // start the calibration
    ADC1->CR |= ADC_CR_ADCAL;
    while (ADC1->CR & ADC_CR_ADCAL)
        ;
    // Configure for single-ended mode on channel 5 must be set before enabling the ADC1
    // Using ADC12_IN5 channel
    ADC1->DIFSEL &= ~(ADC_DIFSEL_DIFSEL_5);
    // Enable ADC1
    // Clear ready bit with a 1
    ADC1->ISR |= (ADC_ISR_ADRDY);
    // Enable ADC
    ADC1->CR |= ADC_CR_ADEN;
    while (!(ADC1->ISR & ADC_ISR_ADRDY))
        ;
    // Clear ready bit with a 1 (optional)
    ADC1->ISR |= (ADC_ISR_ADRDY);
    // Configure ADC1
    // set sequence for 1 conversion on channel 5
    ADC1->SQR1 |= (5 << ADC_SQR1_SQ1_Pos);
    // 12 bit resolution, software trigger, right align, single conversion
    ADC1->CFGR = 0;
    // Used to change sampling rate -> ADC1->SMPR1 = (SMP_640_5 << ADC_SMPR1_SMP5_Pos);

    /*--------------- Set up ADC2 Here --------------*/
    ADC2->CR &= ~(ADC_CR_DEEPPWD);
```

```
    ADC2->CR |= (ADC_CR_ADVREGEN);
    // delay for voltage regulation startup time ~ 20 us
    for (uint32_t i = 0; i < DELAY_TIME; i++)
        ;
    // Calibrate ADC2 and ensure ADC2 is disabled and single-ended mode
    ADC2->CR &= ~(ADC_CR_ADEN | ADC_CR_ADCALDIF);
    // Configure for single-ended mode on channel 6 must be set befor ennable the ADC2
    // Using ADC12_IN6 channel
    ADC2->DIFSEL &= ~(ADC_DIFSEL_DIFSEL_6);
    // Enable ADC2
    // Clear ready bit with a 1
    ADC2->ISR |= (ADC_ISR_ADRDY);
    // Enable ADC
    ADC2->CR |= ADC_CR_ADEN;
    while (!(ADC2->ISR & ADC_ISR_ADRDY))
        ;
    // Clear ready bit with a 1 (optional)
    ADC2->ISR |= (ADC_ISR_ADRDY);
    // Set sequence for 1 conversion on channel 6
    ADC2->SQR1 |= (6 << ADC_SQR1_SQ1_Pos);
    // 12 bit resolution, software trigger, right align, single conversion
    ADC2->CFGR = 0;

    // Start conversion
    ADC1->CR |= (ADC_CR_ADSTART);
    ADC2->CR |= (ADC_CR_ADSTART);
    // Enable interrupts for ADC
    // Interrupt on end of conversion
    ADC1->IER |= (ADC_IER_EOC);
    // Clear EOC flag with 1
    ADC1->ISR &= ~(ADC_ISR_EOC);
    // Interrupt on end of conversion
    ADC2->IER |= (ADC_IER_EOC);
    // Clear EOC flag with h2
    ADC2->ISR &= ~(ADC_ISR_EOC);
    // Enable interrupt in NVIC
    NVIC->ISER[0] = (1 << (ADC1_2_IRQn & 0x1F));
    // Enable interrupts globally
    __enable_irq();
}
```
