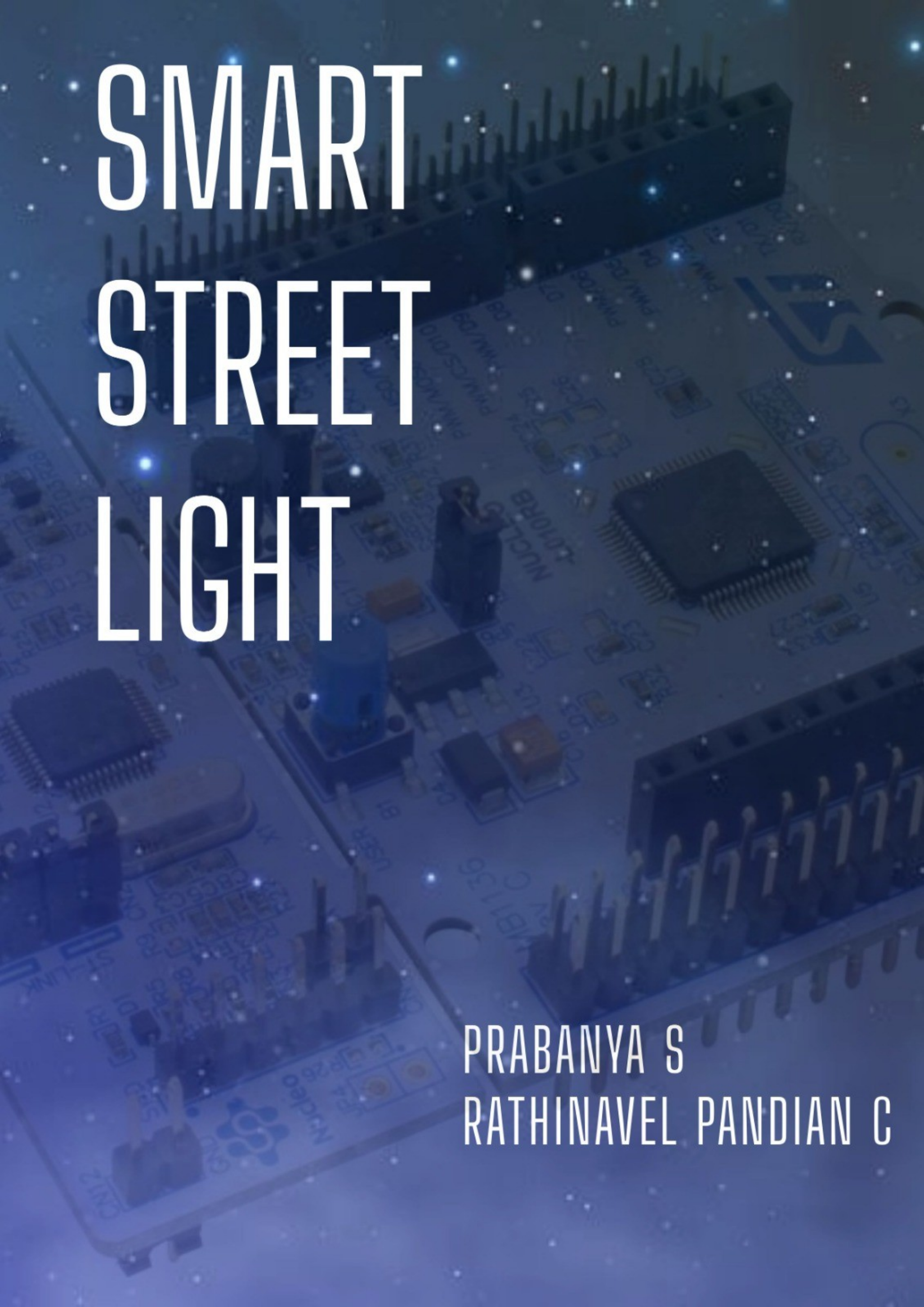
SMART STREET LIGHT

PRABANYA S
RATHINAVEL PANDIAN C



SMART STREET LIGHT

Project Summary

The project involves using an STM32 microcontroller to manage and monitor four street lights. These lights are connected via a 4-channel relay module, allowing precise control over each light's on/off state.

Wireless Communication is facilitated through a LoRa module, which enables long-range, low-power data transmission between the STM32 microcontroller and a Rugged Board. The system continuously transmits the status of each street light to the Rugged Board. This board stores the data in the cloud for remote monitoring and analysis.

The Rugged Board can issue two types of commands either "on" or "off" for each street light. These commands are relayed back to the STM32 microcontroller via the LoRa module, enabling real-time control over the street lights.

An LCD, connected to the STM32 via I2C, displays the current state of all four street lights. This user-friendly interface allows for easy monitoring and management of the lighting system.

Hardware Used and their Specification

- STM32 NUCLEO F446RE Microcontroller

  Core
  ARM® Cortex®-M4 CPU with DSP and FPU
  Frequency: Up to 180 MHz
  Memory
  Flash memory: 512 KB
  SRAM: 128 KB (112 KB + 16 KB backup SRAM)

- 4 channel relay

  Control Signal
  Input Voltage: Typically 3.3V to 5V DC
  Relay Output
  Load Voltage: Up to 250V AC or 30V DC
  Load Current: Typically up to 10A per channel

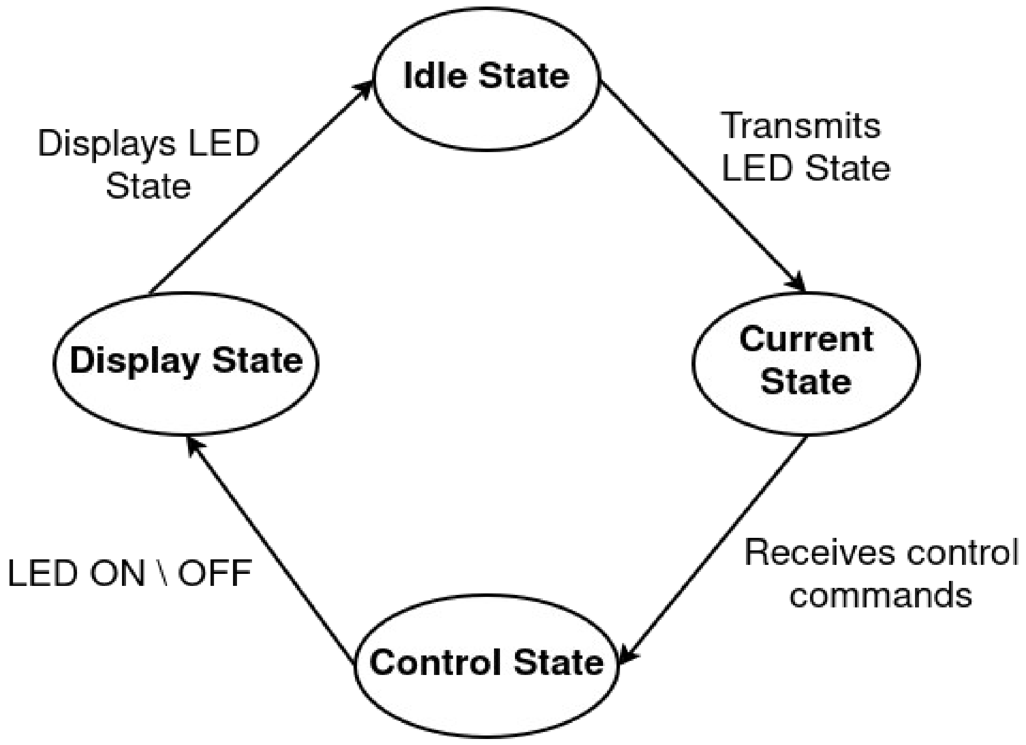
- LCD – I2C module

  Display Type
  Character LCD : 16x2
  InterfaceI2C
  Interface : Uses I2C protocol for communication, reducing the number of pins required

- LoRa module

  Communication
  Data Rate : LoRa modulation up to 300 kbps.
  Power Requirements
  Operating Voltage : 1.8V to 3.6V

Block diagram for finite state machine



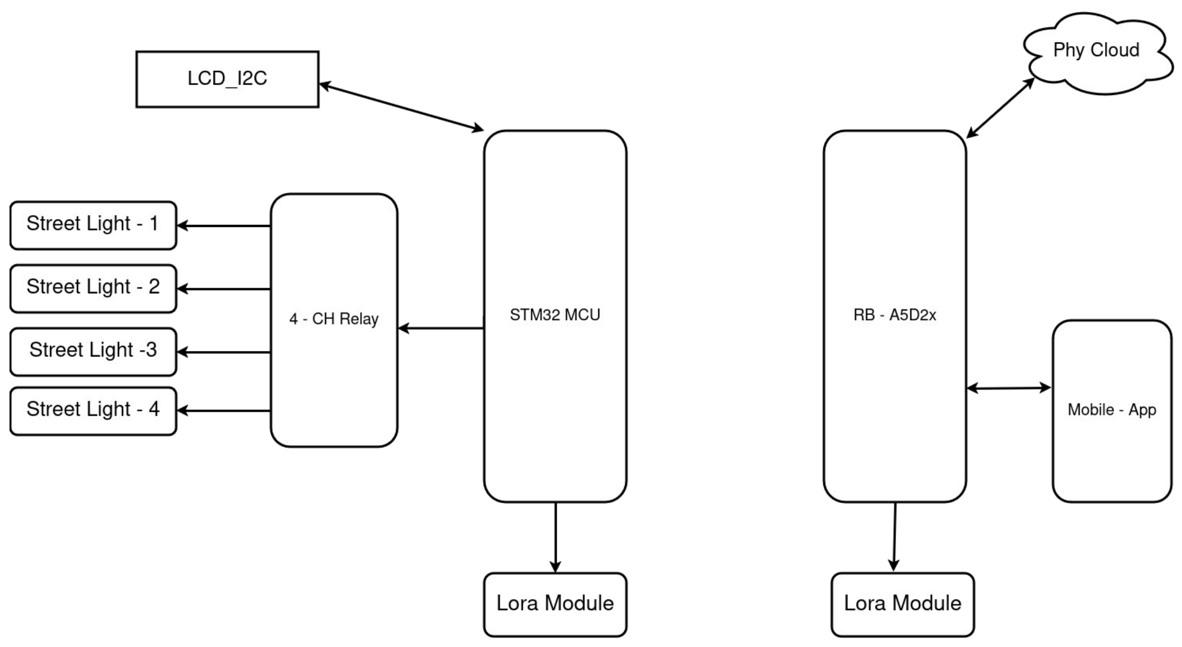
Transition State Diagram

Input	Output			
	Led 1	Led 2	Led 3	Led 4
Relay 1 ON	On	Off	Off	Off
Relay 2 ON	On	On	Off	Off
Relay 3 ON	On	On	On	Off
Relay 4 ON	On	On	On	On
Relay 1 OFF	Off	On	On	On
Relay 2 OFF	Off	Off	On	On
Relay 3 OFF	Off	Off	Off	On
Relay 4 OFF	Off	Off	Off	Off

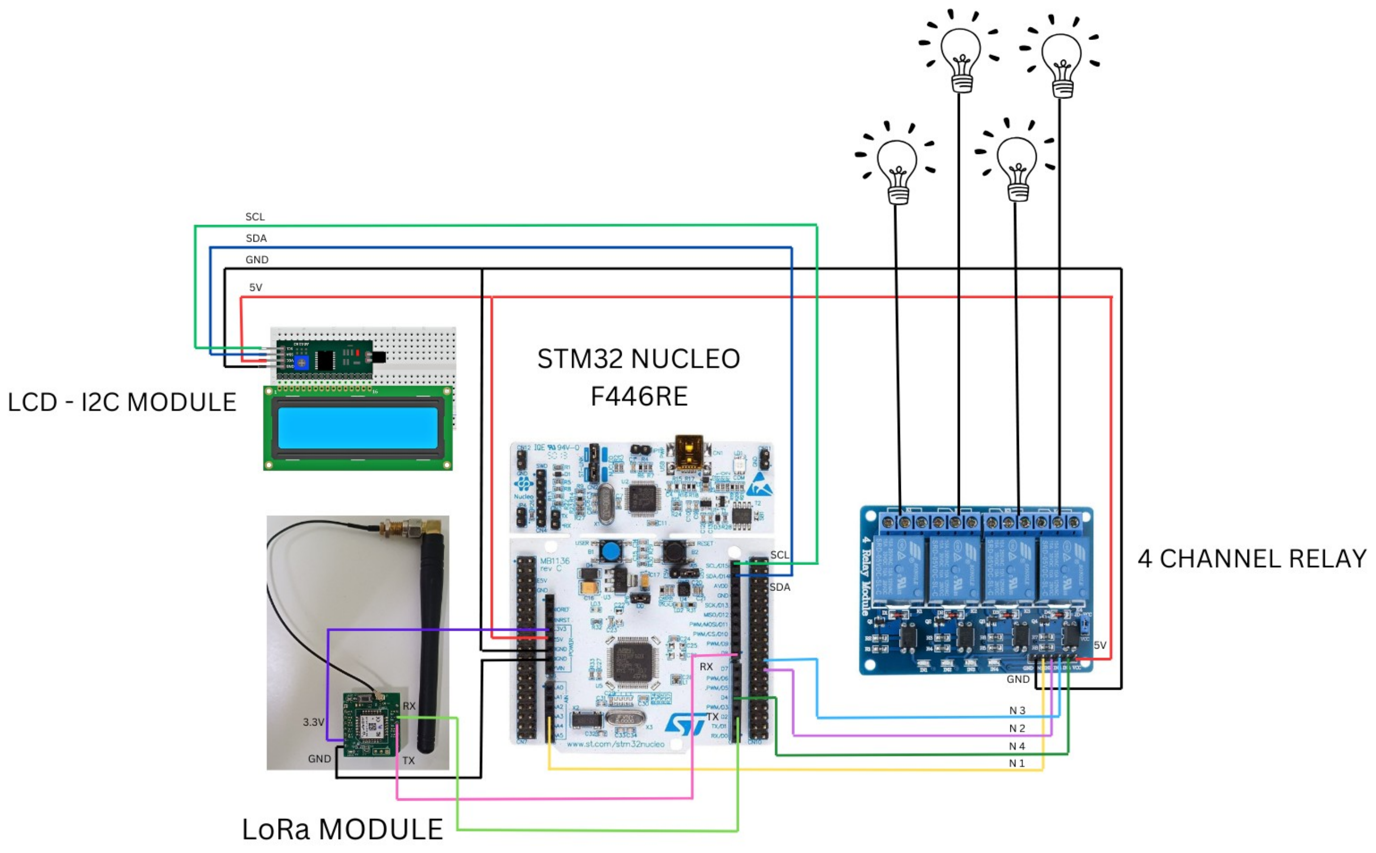
Finite state machine implementation

```
#include<stdio.h>
typedef enum
{
   Idle_State,
   Current_State,
   Control_State,
   Display_State
}eSystemState;

typedef enum
{
    Led_State_Event,
    Control_Event,
    On_Off_Event,
    Display_Event
}eSystemEvent;

eSystemState LedStateHandle(void)
{
     return Current_State;
}

eSystemState ControlHandler(void)
{
     return Control_State;
}

eSystemState OnOffHandler(void)
{
      return Display_State;
}

eSystemState DisplayHandler(void)
{
       return Idle_State;
}

int main(int argc,char *argv[])
{
    eSystemState eNextState = Idle_State;
    eSystemEvent eNewEvent;
    while(1)
    {
       eSystemEvent eNewEvent = ReadEvent();
       switch(eNextState)
       {
         case Idle_State:
          {
               if(Led_State_Event == eNewEvent)
               {
                  eNextState = LedStateHandler();
                }
              }
           break;
          case Current_State:
          {
                if(Control_Event == eNewEvent)
                {
                   eNextState = ControlHandler();
                 }
}
 break;
case Control_State:
{
     if(On_Off_Event == eNewEvent)
     {
         eNextState = OnOffHandler();
      }
  }
   break;
  case Display_State:
  {
      if(Display_Event == eNewEvent)
      {
          eNextState = DisplayHandler();
       }
    }
     break;
     default:
          break;
      }
    }
     return 0;
     }
```

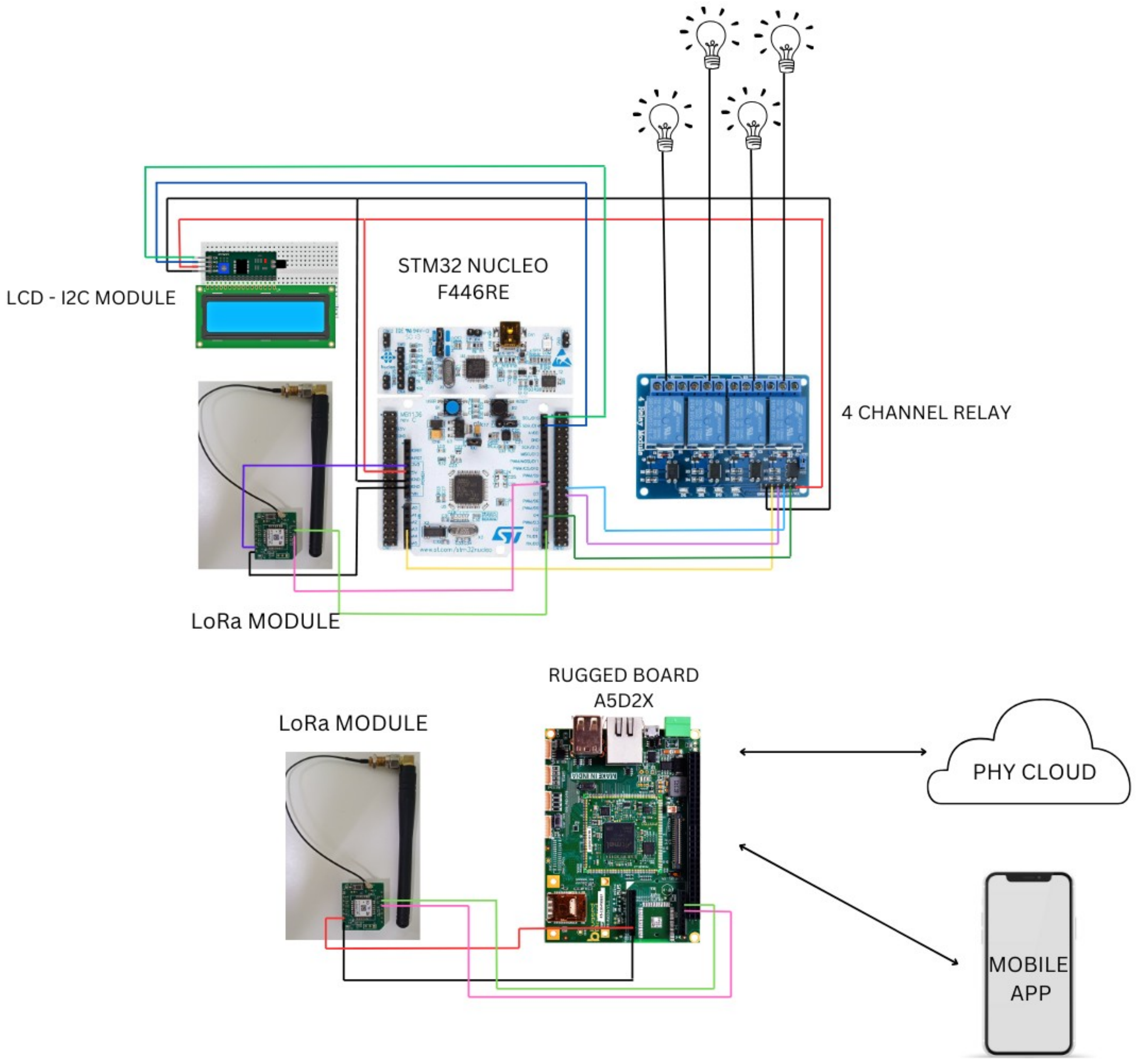
Block Diagram



Connection Diagram



Low Level Block Diagram



Peripheral Explanation

Microcontroller

The STM32 Nucleo-F446RE is a versatile development board featuring the STM32F446RE microcontroller, which is based on the ARM Cortex-M4 core running at up to 180 MHz. It offers 512 KB of Flash memory and 128 KB of SRAM, making it suitable for complex applications requiring high performance and real-time processing. The board includes a wide range of connectivity options, such as UART, I2C, SPI, and CAN, as well as USB OTG support, allowing it to interface with various peripherals and function as both a USB device and host. The board is ideal for a variety of applications, from IoT and industrial control to consumer electronics.

The STM32 Nucleo-F446RE also supports the ST-LINK/V2-1 debugger and programmer, which is integrated on the board, providing seamless debugging and programming capabilities without the need for an external programmer. It is compatible with a range of development environments, including STM32CubeIDE, Keil MDK, and IAR Embedded Workbench, offering flexibility for developers. The board's low-power modes and power consumption monitoring features make it suitable for energy-efficient applications

4 – Channel Relay

A 4-channel relay module allows you to control four separate electrical circuits using low-power signals from a microcontroller like an STM32. Each relay on the module can switch devices such as lights, motors, or other high-voltage equipment on and off, making it ideal for controlling multiple devices in a project. The relays are triggered by low-voltage control signals, typically 3.3V or 5V, and provide electrical isolation between the control and power circuits, ensuring safe operation. This makes the 4-channel relay module a versatile component for managing higher voltage and current loads in embedded systems, while still being controlled by the low-power logic of the microcontroller.

The module typically has LED indicators for each relay channel, allowing you to visually monitor the on/off state of each relay. It also offers screw terminal connectors for securely attaching the high-voltage wires, ensuring a reliable connection. With its ability to control multiple devices simultaneously, the 4-channel relay module is commonly used in home automation, industrial control, and IoT applications, where it serves as an interface between low-power control systems and high-power devices.

LCD – I2C Module

An LCD I2C module integrates a standard character LCD with an I2C communication interface, simplifying the connection between the display and a microcontroller like an STM32. By using the I2C protocol, the module reduces the number of required connections from the usual 16 pins to just four: VCC, GND, SDA, and SCL, streamlining the wiring and saving valuable GPIO pins. The module typically features standard display sizes such as 16x2 or 20x4 characters, which are suitable for displaying text and simple information. Additionally, the I2C interface allows for multiple devices to share the same bus, with each device identified by a unique address, making it convenient for expanding the system with other I2C peripherals. This combination of reduced wiring complexity and efficient communication makes the LCD I2C module a popular choice for embedded systems and user interfaces.

It includes a dedicated I2C controller on the module, which handles the communication protocol and minimizes the load on the microcontroller's processing resources. The module often comes with a contrast adjustment potentiometer to fine-tune the display visibility according to different lighting conditions. Furthermore, it typically supports various commands and functions, such as clearing the display, moving the cursor, and scrolling text, providing flexibility for dynamic and interactive user interfaces. The reduced pin count and simplified wiring make it especially valuable in compact or complex projects where space and pin availability are constraints.

LoRa Module

A LoRa (Long Range) module is a wireless communication device that utilizes LoRa technology to transmit data over long distances with low power consumption. It operates in unlicensed frequency bands, such as 868 MHz or 915 MHz, and is capable of sending data across several kilometers, making it ideal for IoT applications where long-range communication is required. LoRa modules are designed to provide robust and reliable communication in challenging environments, offering high interference immunity and the ability to penetrate through buildings and obstacles. These modules support low data rates, which contribute to their extended range and energy efficiency, making them suitable for applications like smart cities, remote monitoring, and agricultural systems. LoRa modules can be easily integrated with microcontrollers like STM32 via UART, SPI, or I2C interfaces, allowing for flexible deployment in various wireless sensor networks and communication systems.

In addition to long-range communication, LoRa modules offer advanced features such as adaptive data rate (ADR) to optimize performance based on network conditions, and built-in error correction mechanisms to ensure data integrity over vast distances. They are often used in conjunction with LoRaWAN (LoRa Wide Area Network) protocol, which provides a standardized framework for secure, scalable, and low-power wide-area networks. LoRa modules are ideal for battery-powered devices, as they support sleep modes that further reduce power consumption, enabling years of operation on a single battery. The versatility of LoRa technology allows it to be deployed in a wide range of applications, from environmental monitoring and asset tracking to smart metering and industrial automation. Additionally, the open ecosystem of LoRa enables developers to create custom network architectures and integrate the modules into existing systems with ease, making LoRa a powerful tool for creating resilient, long-range wireless networks.

Rugged Board

The Rugged Board A5D2x is a high-performance development platform centered around the Microchip SAMA5D2 ARM Cortex-A5 processor, making it well-suited for industrial and IoT applications requiring robust and reliable performance. Designed for use in harsh environments, the board features extended temperature ranges and industrial-grade construction. It offers a wide array of connectivity options, including Ethernet, USB, multiple UART, SPI, I2C, and CAN interfaces, as well as support for SD cards, enabling flexible integration with various peripherals and systems. The Rugged Board A5D2x is also optimized for low-power consumption, making it ideal for energy-efficient and battery-powered applications. With its powerful processing capabilities and extensive I/O options, it provides a versatile platform for developing advanced embedded systems and resilient IoT solutions.

In addition to its robust design and extensive connectivity, the Rugged Board A5D2x supports advanced security features, including hardware encryption and secure boot, ensuring the protection of sensitive data and safeguarding against unauthorized access. Its compatibility with Linux and a variety of real-time operating systems allows developers to leverage a rich ecosystem of software and tools, facilitating rapid development and deployment of applications. The board also offers expansion headers for adding custom modules, enabling tailored solutions for specific industrial or IoT needs. With its combination of durability, processing power, and flexibility, the Rugged Board A5D2x is an excellent choice for mission-critical applications in fields such as automation, transportation, and remote monitoring.

MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for low-bandwidth, high-latency, or unreliable networks, making it ideal for IoT (Internet of Things) applications. It operates on a publish/subscribe model, where devices (clients) send messages (publish) to a central broker, and other devices can receive those messages (subscribe) based on specified topics. This architecture reduces network traffic and allows for efficient communication between a large number of devices. MQTT is optimized for minimal resource consumption, enabling devices with limited processing power and memory to communicate effectively. Its simplicity, coupled with robust features like Quality of Service (QoS) levels and persistent sessions, makes MQTT a popular choice for real-time data exchange in applications such as smart homes, industrial automation, and remote monitoring systems.

MQTT is highly scalable, capable of supporting networks ranging from a few devices to thousands, making it suitable for both small-scale and large-scale IoT deployments. The protocol's support for different Quality of Service (QoS) levels allows developers to balance between message delivery guarantees and network bandwidth usage, depending on the application's requirements. MQTT is also well-suited for battery-operated devices, as it minimizes data transmission and can maintain connections with minimal power consumption. The protocol's retain and last will message features enhance reliability by ensuring that critical messages are delivered even if the device disconnects. MQTT's flexibility and support for a wide range of programming languages and platforms have made it a standard in IoT ecosystems, enabling seamless communication between devices across diverse environments. Its use in cloud-based services further extends its capabilities, allowing for real-time data integration and control in distributed systems.

Code

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"


/* 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 ---------------------------------------------------------*/
I2C_HandleTypeDef hi2c1;
UART_HandleTypeDef huart1;
UART_HandleTypeDef huart2;
osThreadId TransmitHandle;
osThreadId ReceiveHandle;
osMessageQId messageQueueHandle;
/* 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);
static void MX_I2C1_Init(void);
static void MX_USART1_UART_Init(void);
void StartTransmit(void const * argument);
void StartReceive(void const * argument);
/* USER CODE BEGIN PFP */
/* USER CODE END PFP */
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
/* Task1 FSM States */
typedef enum {
   STATE_INIT,
   STATE_SEND_DATA,
   STATE_WAIT_TRANSMIT
} Transmit_State;
/* Task2 FSM States */
typedef enum {
   STATE_WAIT,
   STATE_RECEIVE
} Receive_State;
#define RELAY1_PIN GPIO_PIN_0
#define RELAY1_PORT GPIOB
```

```
#define RELAY2_PIN GPIO_PIN_1
#define RELAY2_PORT GPIOB
#define RELAY3_PIN GPIO_PIN_2
#define RELAY3_PORT GPIOB
#define RELAY4_PIN GPIO_PIN_4
#define RELAY4_PORT GPIOB
/* USER CODE END 0 */
/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */
  /* USER CODE END 1 */
  /* MCU Configuration--------------------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* USER CODE BEGIN Init */
  /* USER CODE END Init */
  /* Configure the system clock */
  SystemClock_Config();
  /* USER CODE BEGIN SysInit */
  /* USER CODE END SysInit */
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_I2C1_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  HAL_GPIO_WritePin(RELAY1_PORT, RELAY1_PIN, GPIO_PIN_SET);
  HAL_GPIO_WritePin(RELAY2_PORT, RELAY2_PIN, GPIO_PIN_SET);
  HAL_GPIO_WritePin(RELAY3_PORT, RELAY3_PIN, GPIO_PIN_SET);
  HAL_GPIO_WritePin(RELAY4_PORT, RELAY4_PIN, GPIO_PIN_SET);
  /* USER CODE END 2 */
  /* USER CODE BEGIN RTOS_MUTEX */
  /* add mutexes, ... */
  /* USER CODE END RTOS_MUTEX */
  /* USER CODE BEGIN RTOS_SEMAPHORES */
  /* add semaphores, ... */
  /* USER CODE END RTOS_SEMAPHORES */
  /* USER CODE BEGIN RTOS_TIMERS */
  /* start timers, add new ones, ... */
  /* USER CODE END RTOS_TIMERS */
  /* USER CODE BEGIN RTOS_QUEUES */
  /* add queues, ... */
  /* Create the message queue */
   osMessageQDef(messageQueue, 10, uint8_t); // Queue with 10 items of type
uint8_t
```

```
  messageQueueHandle = osMessageCreate(osMessageQ(messageQueue), NULL);
  /* USER CODE END RTOS_QUEUES */
  /* Create the thread(s) */
  /* definition and creation of defaultTask */
  osThreadDef(Transmit, StartTransmit, osPriorityNormal, 0, 128);
  TransmitHandle = osThreadCreate(osThread(Transmit), NULL);
  osThreadDef(Receive, StartReceive, osPriorityNormal, 0, 128);
  ReceiveHandle = osThreadCreate(osThread(Receive), NULL);
  /* USER CODE BEGIN RTOS_THREADS */
  /* add threads, ... */
  /* USER CODE END RTOS_THREADS */
  /* 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 */
  }
  /* 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_VOLTA
GE_SCALE3);
  /** 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_DEFAUL
T;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 16;
  RCC_OscInitStruct.PLL.PLLN = 336;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  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_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) !=
HAL_OK)
  {
    Error_Handler();
  }
}
/**
  * @brief I2C1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_I2C1_Init(void)
```

```
{
  /* USER CODE BEGIN I2C1_Init 0 */
  /* USER CODE END I2C1_Init 0 */
  /* USER CODE BEGIN I2C1_Init 1 */
  /* USER CODE END I2C1_Init 1 */
  hi2c1.Instance = I2C1;
  hi2c1.Init.ClockSpeed = 100000;
  hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
  hi2c1.Init.OwnAddress1 = 0;
  hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c1.Init.OwnAddress2 = 0;
  hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(&hi2c1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN I2C1_Init 2 */
  /* USER CODE END I2C1_Init 2 */
}
/**
  * @brief USART1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART1_UART_Init(void)
{
  /* USER CODE BEGIN USART1_Init 0 */
  /* USER CODE END USART1_Init 0 */
  /* USER CODE BEGIN USART1_Init 1 */
  /* USER CODE END USART1_Init 1 */
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 9600;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART1_Init 2 */
  /* USER CODE END USART1_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;
  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};
  /* 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(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
  /*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOB,          GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|
GPIO_PIN_4, 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 : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
  /*Configure GPIO pins : PB0 PB1 PB2 PB4 */
GPIO_InitStruct.Pin=                    GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|
GPIO_PIN_4;
  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 4 */
/* USER CODE END 4 */
/* USER CODE BEGIN Header_StartDefaultTask */
/**
  * @brief  Function implementing the defaultTask thread.
  * @param  argument: Not used
  * @retval None
  */
/* USER CODE END Header_StartDefaultTask */
void StartTransmit(void const * argument)
{
        Transmit_State state = STATE_INIT;
        uint8_t dataToSend = 0;
        osEvent event;
        for( ; ;)
        {
                switch(state)
                {
                case STATE_INIT:
                        state = STATE_SEND_DATA;
                        break;
                case STATE_SEND_DATA:
                        for(dataToSend = 0 ; dataToSend < 4; dataToSend++)
                 osMessagePut(messageQueueHandle, dataToSend, osWaitForever);
                        state = STATE_WAIT_TRANSMIT;
                        break;
                case STATE_WAIT_TRANSMIT:
                        event= osMessageGet(messageQueueHandle, osWaitForever);
                        uint8_t receivedData = event.value.v;
                        TransmitCommand(receivedData);
                        state = STATE_INIT;
                        break;
                }
        }
}
void StartReceive(void const * argument)
{
        Receive_State state = STATE_WAIT;
        for( ; ; )
        {
                switch(state)
                {
                case STATE_WAIT:
                        state = STATE_RECEIVE;
                        break;
                case STATE_RECEIVE:
                        ReceiveResponse();
                        state = STATE_WAIT;
                        break;
                }
        }
}
/**
  * @brief  Period elapsed callback in non blocking mode
  * @note   This function is called  when TIM1 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 == TIM1) {
    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 */
```

lora_module.c

```
#include "lora_module.h"
#include "string.h"
#include "stdio.h"
#define RELAY1_PIN GPIO_PIN_0
#define RELAY1_PORT GPIOB
#define RELAY2_PIN GPIO_PIN_1
#define RELAY2_PORT GPIOB
#define RELAY3_PIN GPIO_PIN_2
#define RELAY3_PORT GPIOB
```

```
#define RELAY4_PIN GPIO_PIN_4

#define RELAY4_PORT GPIOB

char rxBuffer[256];

volatile uint16_t rxIndex = 0;

extern UART_HandleTypeDef huart1;

volatile uint8_t relay1_state = 0; // 0 for OFF, 1 for ON

volatile uint8_t relay2_state = 0;

volatile uint8_t relay3_state = 0;

volatile uint8_t relay4_state = 0;

char cmdQueue[][256] = {

    "AT\r\n",

    "AT+MODE=TEST\r\n",

    "AT+TEST=RFCFG,866,SF12,125,12,15,14,ON,OFF,OFF\r\n",

    "AT+TEST=RXLRPKT\r\n"

};

void TransmitCommand(uint8_t cmdIndex) {

printf("Transmitting Command: %s", cmdQueue[cmdIndex]);  // Debug print
for command

                                        HAL_UART_Transmit_IT(&huart1,  (uint8_t
*)cmdQueue[cmdIndex], strlen(cmdQueue[cmdIndex]));

}

void ReceiveResponse(void) {

    HAL_UART_Receive_IT(&huart1, (uint8_t *)&rxBuffer[rxIndex], 1);

}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == USART1) {
        if (rxBuffer[rxIndex] == '\n') {
            rxBuffer[rxIndex + 1] = '\0'; // Null-terminate the buffer
            char *payloadStart = strstr((char *)rxBuffer, "+TEST: RX \"");
                        if (payloadStart) {
                            payloadStart += strlen("+TEST: RX \"");
                            char extractedData[256];
                            char *endQuote = strchr(payloadStart, '\"');
                            if (endQuote) {
                                size_t length = endQuote - payloadStart;
                                strncpy(extractedData, payloadStart, length);
                                extractedData[length] = '\0';
                                char asciiString[128];
                                size_t i, j;
                                for (i = 0, j = 0; i < length; i += 2) {
                                    unsigned int hexValue;
                                    sscanf(&extractedData[i], "%2x", &hexValue);
                                    asciiString[j++] = (char)hexValue;
                                }
                                asciiString[j] = '\0';
                                printf("Last Received Data: %s\n", asciiString);
                                if (strcmp(asciiString, "SL1:ON ") == 0) {
                                     HAL_GPIO_WritePin(RELAY1_PORT, RELAY1_PIN,
GPIO_PIN_RESET);
                                    relay1_state = 1;
```

```
                    } else if (strcmp(asciiString, "SL1:OFF") == 0) {
                        HAL_GPIO_WritePin(RELAY1_PORT, RELAY1_PIN,
GPIO_PIN_SET);
                        relay1_state = 0;
                    } else if (strcmp(asciiString, "SL2:ON ") == 0) {
                        HAL_GPIO_WritePin(RELAY2_PORT, RELAY2_PIN,
GPIO_PIN_RESET);
                        relay2_state = 1;
                    } else if (strcmp(asciiString, "SL2:OFF") == 0) {
                        HAL_GPIO_WritePin(RELAY2_PORT, RELAY2_PIN,
GPIO_PIN_SET);
                        relay2_state = 0;
                    } else if (strcmp(asciiString, "SL3:ON ") == 0) {
                        HAL_GPIO_WritePin(RELAY3_PORT, RELAY3_PIN,
GPIO_PIN_RESET);
                        relay3_state = 1;
                    } else if (strcmp(asciiString, "SL3:OFF") == 0) {
                        HAL_GPIO_WritePin(RELAY3_PORT, RELAY3_PIN,
GPIO_PIN_SET);
                        relay3_state = 0;
                    } else if (strcmp(asciiString, "SL4:ON ") == 0) {
                        HAL_GPIO_WritePin(RELAY4_PORT, RELAY4_PIN,
GPIO_PIN_RESET);
                        relay4_state = 1;
                    } else if (strcmp(asciiString, "SL4:OFF") == 0) {
                        HAL_GPIO_WritePin(RELAY4_PORT, RELAY4_PIN,
GPIO_PIN_SET);
```

```
                        relay4_state = 0;
                    }
                    UpdateLCD();
                }
            }
          // Handle received data (you can pass control to another module here)
          rxIndex = 0; // Reset buffer index for the next message
        } else {
          rxIndex++;
        }
        ReceiveResponse(); // Continue receiving
    }
}
```

lora_module.h

```
#ifndef LORA_MODULE_H
#define LORA_MODULE_H
extern char rxBuffer[256];
#include "main.h"
extern char rxBuffer[256];
//extern volatile uint16_t rxIndex;
void TransmitCommand(uint8_t cmdIndex);
void ReceiveResponse(void);
void lora_config(void);
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);
#endif // LORA_MODULE_H
```

lcd_module.c

```
#include "lcd_module.h"
#include "stdio.h"
#include "stm32f4xx_hal.h"
#define I2C_ADDRESS_LCD 0x27 << 1
extern I2C_HandleTypeDef hi2c1;
extern volatile uint8_t relay1_state;
extern volatile uint8_t relay2_state;
extern volatile uint8_t relay3_state;
extern volatile uint8_t relay4_state;
void lcd_send_cmd(char cmd) {
    // Implementation for sending commands to the LCD
          char data_u, data_l;
              uint8_t data_t[4];
              data_u = (cmd & 0xf0);
              data_l = ((cmd << 4) & 0xf0);
              data_t[0] = data_u | 0x0C;  // en=1, rs=0
              data_t[1] = data_u | 0x08;  // en=0, rs=0
              data_t[2] = data_l | 0x0C;  // en=1, rs=0
              data_t[3] = data_l | 0x08;  // en=0, rs=0
```

```
            HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS_LCD, (uint8_t
*)data_t, 4, 100);
}
void lcd_send_data(char data) {
   // Implementation for sending data to the LCD
        char data_u, data_l;
           uint8_t data_t[4];
           data_u = (data & 0xf0);
           data_l = ((data << 4) & 0xf0);
           data_t[0] = data_u | 0x0D;  // en=1, rs=1
           data_t[1] = data_u | 0x09;  // en=0, rs=1
           data_t[2] = data_l | 0x0D;  // en=1, rs=1
           data_t[3] = data_l | 0x09;  // en=0, rs=1
            HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDRESS_LCD, (uint8_t
*)data_t, 4, 100);
}
void lcd_init(void) {
        lcd_send_cmd(0x01);  //LCD Reset
   lcd_send_cmd(0x02);  // initialize LCD in 4-bit mode
   lcd_send_cmd(0x28);  // 2 line, 5*7 matrix
   lcd_send_cmd(0x0c);  // display on, cursor off
   lcd_send_cmd(0x80);  // force cursor to beginning (1st line)
}
void lcd_send_string(char *str) {
   while (*str) lcd_send_data(*str++);
```

```
    //lcd_send_cmd(0x01);  //LCD Reset
}
void UpdateLCD(void) {
    char buffer[32];
    // Set cursor to the first line and print SL1 and SL2 states
    lcd_send_cmd(0x80);  // move cursor to the first line
    snprintf(buffer, sizeof(buffer), "SL1:%s  SL2:%s",
             relay1_state ? " on" : "off",
             relay2_state ? " on" : "off");
    lcd_send_string(buffer);
    // Set cursor to the second line and print SL3 and SL4 states
    lcd_send_cmd(0xC0);  // move cursor to the second line
    snprintf(buffer, sizeof(buffer), "SL3:%s  SL4:%s",
             relay3_state ? " on" : "off",
             relay4_state ? " on" : "off");
    lcd_send_string(buffer);
}
```

lcd_module.h

```
#ifndef LCD_MODULE_H
#define LCD_MODULE_H
#include "main.h"
void lcd_init(void);
void lcd_send_cmd(char cmd);

void lcd_send_data(char data);

void lcd_send_string(char *str);

void UpdateLCD(void);

#endif // LCD_MODULE_H
```

Code Flow

Initially in main function all 4 relays are set to off state. As the program is done with freeRTOS two tasks (StartTransmit, StartReceive) and one queue(messageQueue) is created. osKernelStart() is a function in CMSIS-RTOS that starts the RTOS (Real-Time Operating System) kernel.

The StartTransmit task is designed to periodically send and receive data through a message queue. It continuously puts data into a queue and retrieves data from the same queue, then processes that data by calling the TransmitCommand function.

The TransmitCommand function is used to transmit AT commands and configure LoRa module.

The StartReceive task is designed to handle the reception of data

The ReceiveResponse function is responsible for initiating the reception of data via UART in interrupt mode.

⇩

The HAL_UART_RxCpltCallback function is a callback function called when a UART reception interrupt completes. It extracts the data received through lora module and converts it from hexadecimal value to string and stores it in asciistring. Then asciistring is compared and relay is control based on it. UpdateLCD() function is called.

⇩

UpdateLCD() function is used to display the status of the all four street light

Future Enhancement

In future RTC (Real Time Clock) and motion sensors can be integrated and develop a project that works more efficiently and reduce the consumption of power.

RTC can be used to monitor and send the real time to the cloud by that street light can be on only during night time and off during day time. Microcontroller can also be put in sleep mode during day and woke up mode during night time which also helps in reducing power consumption.

Sensor integrated with PWM (Pulse Width Modulation) can be used to control the intensity of the light so that it glows 100% when there are people and 25% when no one is there.