Project in 62583

Toilet rain water system

05.12.2021

Group 3:

Kim R. H. Christensen
s181554

Jørgen D. Greve
s181519

DTU - ELEKTRO
62583 - Programming of embedded wireless systems and sensors

Contents

1 Introduction 3
1.1 Problem definition 3
1.2 UN Goals 4

2 Design 4
2.1 System flow chart 4
2.2 Protocols 5
2.3 ZigBee protocol 5
2.4 ESP32 protocol 5
2.5 Energy consumption 5

3 Implementation 6
3.1 Case diagram 6

4 Test 6
4.1 MQTT publish 6
4.2 System test 8
4.3 Energy consumption 9

5 Conclusion 9

A Appendix 9
A.1 main.c code 9

1 Introduction

With the increased focus on maximizing the effort to preserve the earths resources there are a rise in making technology that enables users to do that. That could be street lights dimming down when there are no traffic or pedestrians or Smart Homes where automation takes care of for example controlling the heating of a house or turn off home appliances when there are no persons present in the room.

1.1 Problem definition

The problem for this project was to enable households to use rain water for the toilets instead of using mains. Though the underground water resources in Denmark are plentiful, there is still a limit to the amount of water that can be extracted in order to keep the environmental balance at a state where the underground water could be refilled naturally. Here our solution will enable households to reduce the usage of the mains water for sanitary purposes and instead make use of the water collected from precipitation, which in the future is predicted to increase due to climate changes.

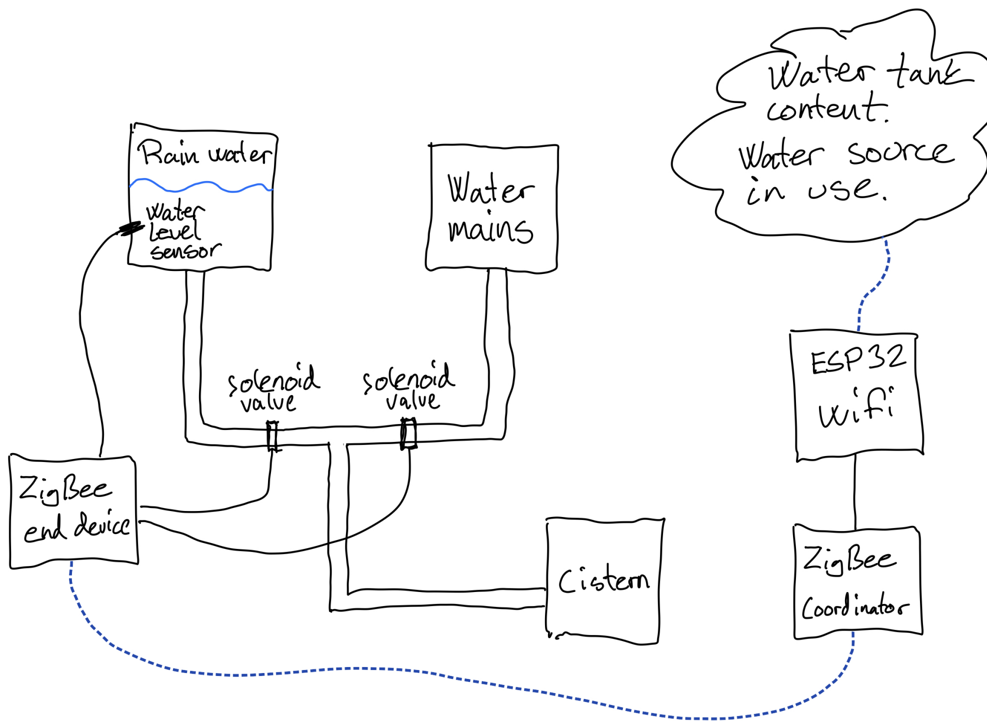

Figure 1: System overview

In figure 1 a schematic of the system is shown. A ZigBee S2C End Device should be responsible keeping track of the amount of water in the water reservoir as well as managing which water source that should be active, hence the water source should change to mains when the water reservoir is at a certain water level. All this is controlled by a ESP-32 that will communicate with the End Device via a ZigBee S2C Coordinator and upload the water level in the water reservoir and current water source to the internet.

In order to make the, above mentioned, data accessible the MQTT protocol should be used, here the ESP-32 will be used as the publisher. This decision was made in order to minimize the components necessary for the project as well as reducing the overall power consumption.

For the project two LEDs should simulate the two solenoid valves and a potentiometer should be used to simulate the water level in the water reservoir.

1.2 UN Goals

When choosing the topic for this project, the United Nations Sustainable Development Goals where consulted in order to create a product that would help reaching one or more of those.

Here the following two goals seemed to fit perfect with our idea for the project.
Goal 6, Clean water and sanitation
By utilizing rain water to flush the toilet instead of clean drinking water this project is helping to ensure that we do not waste precious drinking water and thereby contributing to UN goal 6 which is to "ensure availability and sustainable management of water and sanitation for all"
Furthermore it reduces the need for descaler because of the lack of calcium in rain water and thereby reducing the discharge of harmfull substances into the waste water.

Goal 12, Responsible consumption and production
By reducing the consumption of clean drinking water together with the reduction in the production of descaling products this project contributes towards UN goal 12 which is to "Ensure sustainable consumption and production patterns"

2 Design

2.1 System flow chart

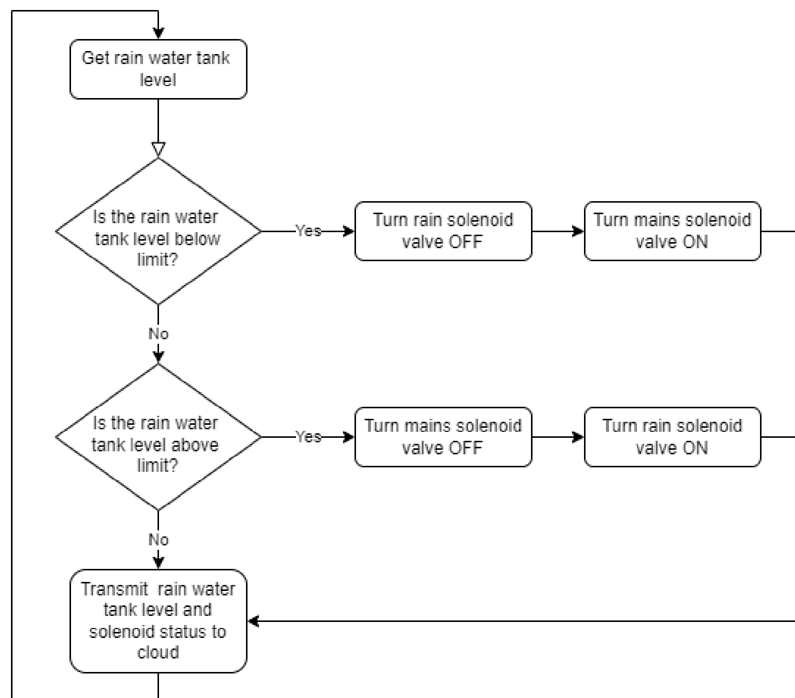

Figure 2: System flowchart

The system should read the water level in the rain water tank and then from some water level parameters decide if it should use water from the rain water tank or the water mains by turning solenoid valves on or off. It should also transmit the current rain water tank level and the solenoid valve status to a cloud server where it can be viewed by a user.
The rain water tank limit should be set so that it is not possible to run out of rain water before next water level check.

2.2 Protocols

Describe and document the different protocols used in communication for sensors to the esp, and esp communication protocol to gateway and to the cloud.

2.3 ZigBee protocol

The ZigBee protocol is build on top of the IEEE 802.15.4 protocol and consists of a Network Layer and Application Layer as seen in the figure 3 below.

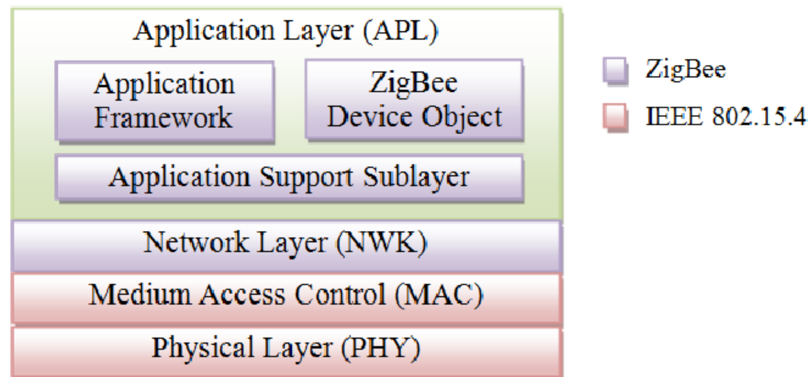

Figure 3: ZigBee Protocol

The Physical layer are responsible for the radio transmission configurations, like the output power and managing the channels. The Medium Control layer is responsible the communication between two devices and handles the data packets, it is not able to direct data to other devices than the closest one in the network. The Network Layer is responsible for security, and network structure. And the Application Layer handles the different profiles in the network, keeps track of which device the ZigBee which to communicate with and set the role in the network (Coordinator, Router or End Device).

2.4 ESP32 protocol

The ESP32 uses the TCP protocol to communicate data to Thingspeak. TCP works by first establishing a connection between the client and the server, then packets of data can be send and then either the server or the client closes the connection. If a data packet of data is lost it can be detected and the packet can be resend.

`https://www.digi.com/resources/documentation/Digidocs/90002002/Content/Reference/r_zb_stack.htm?TocPath=zi`

`s. 40 i Xbee manualen`

2.5 Energy consumption

The system should be designed to be as energy efficient as possible so that it does not contribute unnecessarily to pollution by burning fossile fuels to create energy. The system is built into a building where mains power is at hand. Therefore battery operation is unnecessary and due to the operation of solenoid valves which consume a considerable amount of electrical energy it would require a rater large battery.

3 Implementation

3.1 Case diagram

The ESP32 module is running FreeRTOS with two tasks.

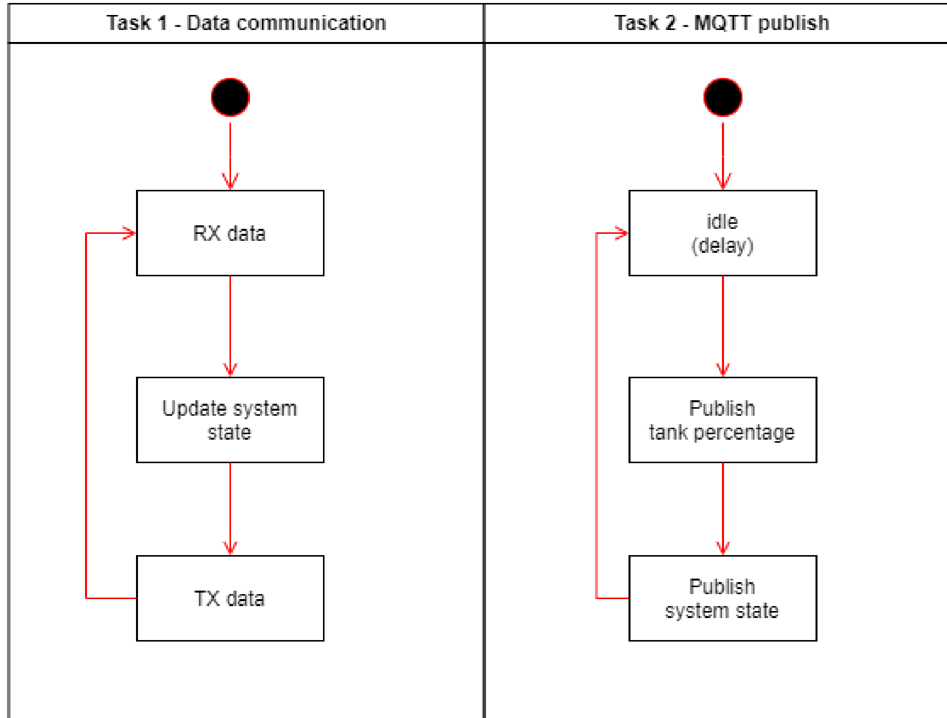

Figure 4: ESP32 case diagram

Task 1 is responsible for handling data communication. First it checks if any new data is available in the UART receive buffer. If there is new data and it comes from the XBee end-device, the rain water tank percentage and the system state is updated. The system state is a variable that is either 1 or 0 where 0 indicates that the system is using water mains as supply and 1 indicates that it uses rain water. After the system update, the ESP32 transmits data to the XBee end-device which then changes the solenoid valves accordingly.

Task 2 is responsible for publishing system information to Thingspeak.com. More specifically it publishes the current water level as a percentage and if it is using water from mains or rain water.

4 Test

4.1 MQTT publish

In this test we tested if the system could publish to Thingspeak. To simulate changing water level in the rain water tank we connected a signal generator to the ADC input on the XBee end-device. The signal generator was set up to output a sine wave with a period of 10 min and an amplitude of 1.2V which is the maximum the XBee ADC can measure.
On the below figure we can see that the sine curve in Field 1 is a bit angular, it should have been much more smooth. The data in Field 2 shows that we use rain water to begin with and then when the water level drops below the limit we switch to mains until the water is above the limit again. The MQTT publish task was at the time set to publish every 10 seconds but if we look at both graphs its easy to see that its somewhat irregular.

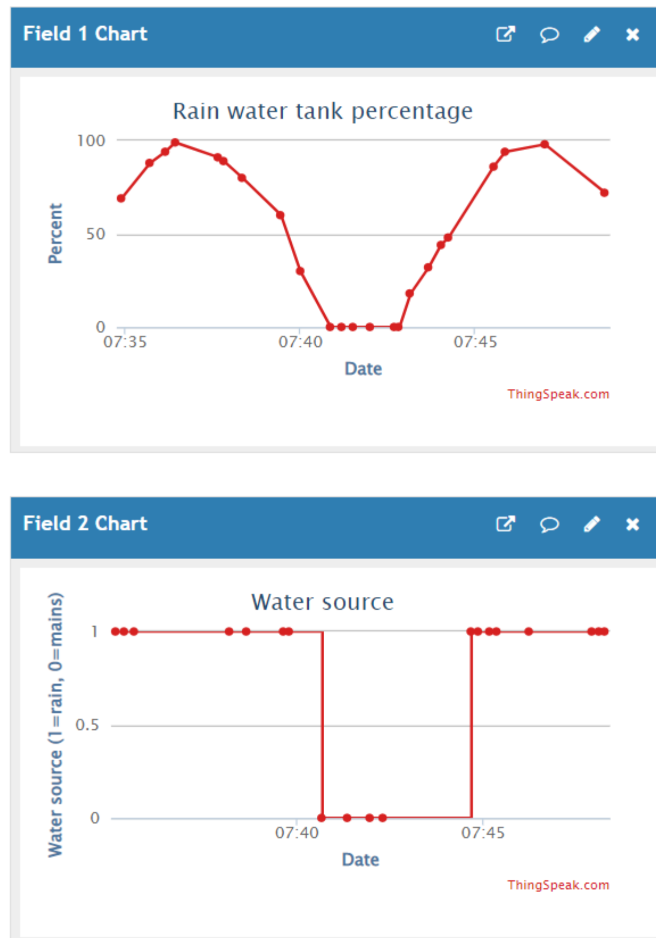

Figure 5: Thingspeak MQTT publish

We have had some difficulties concerning MQTT that we have not been able to solve before our deadline as can be viewed on the below figure. Given some more time we would of course have found a solution for that.

```
I (1592) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (4722) event: sta ip: 192.168.0.150, mask: 255.255.255.0, gw: 192.168.0.1
I (4722) MQTT_EXAMPLE: Other event id:7
I (5722) MQTT_EXAMPLE: Other event id:7
I (7952) MQTT_EXAMPLE: MQTT_EVENT_CONNECTED
I (7952) MQTT_EXAMPLE: sent subscribe successful, msg_id=25135
I (8152) MQTT_EXAMPLE: MQTT_EVENT_SUBSCRIBED, msg_id=25135
I (8162) MQTT_EXAMPLE: sent publish successful, msg_id=0
E (8362) MQTT_CLIENT: mqtt_message_receive: transport_read() error: errno=128
I (8362) MQTT_EXAMPLE: MQTT_EVENT_ERROR
E (8362) MQTT_CLIENT: mqtt_process_receive: mqtt_message_receive() returned -1
I (8372) MQTT_EXAMPLE: MQTT_EVENT_DISCONNECTED
I (8662) MQTT_EXAMPLE: MQTT_EVENT_CONNECTED
I (8672) MQTT_EXAMPLE: sent subscribe successful, msg_id=58416
I (8672) MQTT_EXAMPLE: sent publish successful, msg_id=0
Tank level is 99 percent

Using rain...

Tank level is 98 percent

I (17392) MQTT_EXAMPLE: Other event id:7
Using rain...
I (21262) MQTT_EXAMPLE: MQTT_EVENT_CONNECTED
I (21262) MQTT_EXAMPLE: sent subscribe successful, msg_id=16215
I (21262) MQTT_EXAMPLE: sent publish successful, msg_id=0

I (23372) MQTT_EXAMPLE: Other event id:7
I (23722) MQTT_EXAMPLE: MQTT_EVENT_CONNECTED
I (23722) MQTT_EXAMPLE: sent subscribe successful, msg_id=24836
I (23922) MQTT_EXAMPLE: MQTT_EVENT_SUBSCRIBED, msg_id=24836
I (23932) MQTT_EXAMPLE: sent publish successful, msg_id=0
E (24162) MQTT_CLIENT: mqtt_message_receive: transport_read() error: errno=128
I (24162) MQTT_EXAMPLE: MQTT_EVENT_ERROR
E (24162) MQTT_CLIENT: mqtt_process_receive: mqtt_message_receive() returned -1
I (24172) MQTT_EXAMPLE: MQTT_EVENT_DISCONNECTED
Tank level is 97 percent

Using rain...
```

Figure 6: MQTT errors from the Visual Studio terminal

4.2 System test

Because we did not have any water level sensor or solenoid valves these were simulated. The water level sensor was replaced with a potentiometer or a signal generator both capable of delivering a variable voltage between 0V to 1.2V. The solenoid valves were simulated with LED's (LED on => solenoid open). The green LED simulated the rain water tank solenoid valve and the red simulated the water mains solenoid valve. Except from the MQTT the whole system ran as expected. With 0V on the ADC of the XBee end-device the system chose to use water from the mains. When the voltage hit the defined limit, the system switched to use rain water as source until the voltage dropped below the limit again.

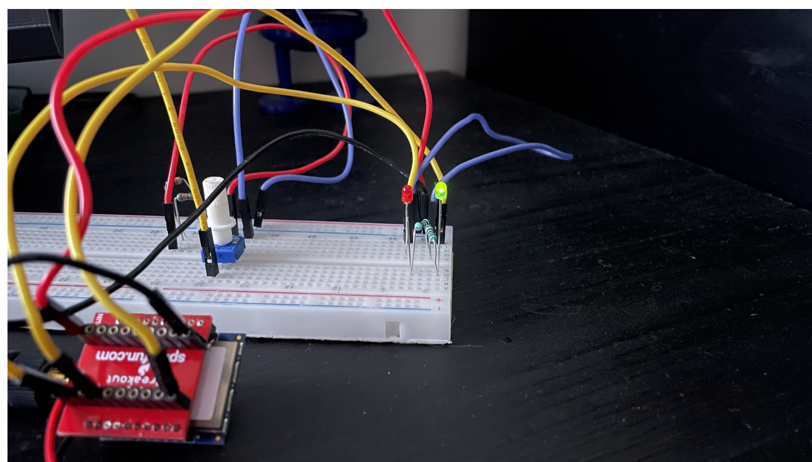

Figure 7: Rain water solenoid on

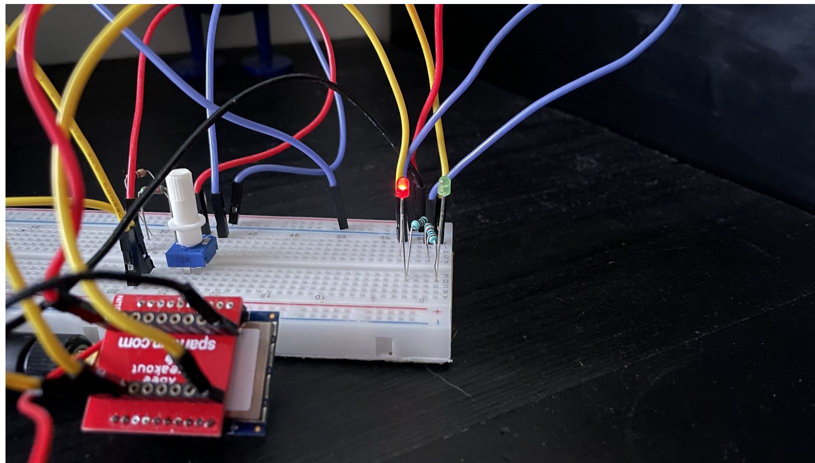

Figure 8: Water mains solenoid on

4.3 Energy consumption

Due to the fact that our system is powered from the mains we have not gone into more detail with power measurements but we do have some measurements. The XBee end-device consumes around 12 mAh but that is without the real solenoid valves which consumes a lot more. The XBee coordinator consumes about 27 mAh and according to our research the ESP32 module consumes around 160 - 260 mAh when it is active.

5 Conclusion

Despite the challenges with MQTT we all in all think that we succeeded in creating a system (or at least a proof of concept) that can handle the electrical part of a rain water toilet flush system. There are of course many other things to consider if the system should be implemented in real life.

A Appendix

A.1 main.c code

```
#include <stdio.h>
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include "esp_wifi.h"
#include "esp_system.h"
#include "nvs_flash.h"
#include "esp_event_loop.h"

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "freertos/queue.h"
#include "freertos/event_groups.h"

#include "lwip/sockets.h"
#include "lwip/dns.h"
#include "lwip/netdb.h"
```

```
#include "esp_log.h"
#include "mqtt_client.h"

#define SECOND (1000 / portTICK_PERIOD_MS)

/////////// UART ///////////////////
#include "driver/uart.h"
#include "soc/uart_struct.h"

static const int RX_BUF_SIZE = 1024;

#define TXD_PIN (GPIO_NUM_4)
#define RXD_PIN (GPIO_NUM_5)

/////////// XBEE API COMMANDS ////////
char solenoid_rain_on[] =   {0x7E,0x00,0x10,0x17,0x01,0x00,0x13,0xA2,0x00,0x41,0xB5,0xFE,0xA0,0xFF,
                             0xFE,0x02,0x44,0x34,0x05,0x22};
char solenoid_rain_off[] =  {0x7E,0x00,0x10,0x17,0x01,0x00,0x13,0xA2,0x00,0x41,0xB5,0xFE,0xA0,0xFF,
                             0xFE,0x02,0x44,0x34,0x04,0x23};
char solenoid_mains_on[] =  {0x7E,0x00,0x10,0x17,0x01,0x00,0x13,0xA2,0x00,0x41,0xB5,0xFE,0xA0,0xFF,
                             0xFE,0x02,0x44,0x33,0x05,0x23};
char solenoid_mains_off[] = {0x7E,0x00,0x10,0x17,0x01,0x00,0x13,0xA2,0x00,0x41,0xB5,0xFE,0xA0,0xFF,
                             0xFE,0x02,0x44,0x33,0x04,0x24};

/////////// RAIN TANK STATUS ////////
int system_state = 0;
/////////////////////////////////////

static const char *TAG = "MQTT_EXAMPLE";

static EventGroupHandle_t wifi_event_group;
static EventGroupHandle_t mqtt_event_group;
const static int CONNECTED_BIT = BIT0;

SemaphoreHandle_t print_mux = NULL;

char tank_perc_pub[46] = {0};
char sys_state_pub[46] = {0};

int change = 0;
uint16_t tank_percent = 0;

esp_mqtt_client_handle_t client;



///////////////////////////////// MQTT EVENT HANDLER ////////////////////////////////////
static esp_err_t mqtt_event_handler(esp_mqtt_event_handle_t event)
{
    esp_mqtt_client_handle_t client = event->client;
    int msg_id;

    switch (event->event_id)
    {
        case MQTT_EVENT_CONNECTED:
```

```
            ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");

            xEventGroupSetBits(mqtt_event_group, CONNECTED_BIT);
            /*
            msg_id = esp_mqtt_client_publish(client, "/topic/qos1", "data_3", 0, 1, 0);
            ESP_LOGI(TAG, "sent publish successful, msg_id=%d", msg_id);

            msg_id = esp_mqtt_client_subscribe(client, "/topic/qos0", 0);
            ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);

            msg_id = esp_mqtt_client_subscribe(client, "/topic/qos1", 1);
            ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);

            msg_id = esp_mqtt_client_unsubscribe(client, "/topic/qos1");
            ESP_LOGI(TAG, "sent unsubscribe successful, msg_id=%d", msg_id);
            */
            msg_id = esp_mqtt_client_subscribe(client, "channels/1572371/subscribe", 0);
           ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);

            break;
        case MQTT_EVENT_DISCONNECTED:
            ESP_LOGI(TAG, "MQTT_EVENT_DISCONNECTED");
            break;
        case MQTT_EVENT_SUBSCRIBED:
            ESP_LOGI(TAG, "MQTT_EVENT_SUBSCRIBED, msg_id=%d", event->msg_id);
            msg_id = esp_mqtt_client_publish(client, "/topic/qos0", "data", 0, 0, 0);
            ESP_LOGI(TAG, "sent publish successful, msg_id=%d", msg_id);
            break;
        case MQTT_EVENT_UNSUBSCRIBED:
            ESP_LOGI(TAG, "MQTT_EVENT_UNSUBSCRIBED, msg_id=%d", event->msg_id);
            break;
        case MQTT_EVENT_PUBLISHED:
            ESP_LOGI(TAG, "MQTT_EVENT_PUBLISHED, msg_id=%d", event->msg_id);
            break;
        case MQTT_EVENT_DATA:
            ESP_LOGI(TAG, "MQTT_EVENT_DATA");
            printf("TOPIC=%.*s\r\n", event->topic_len, event->topic);
            printf("DATA=%.*s\r\n", event->data_len, event->data);
            break;
        case MQTT_EVENT_ERROR:
            ESP_LOGI(TAG, "MQTT_EVENT_ERROR");
            break;
        default:
            ESP_LOGI(TAG, "Other event id:%d", event->event_id);
            break;
    }
    return ESP_OK;
}


//////////////////////////////// WIFI EVENT HANDLER ////////////////////////////////////
static esp_err_t wifi_event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) {
        case SYSTEM_EVENT_STA_START:
```

```
            esp_wifi_connect();
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            xEventGroupSetBits(wifi_event_group, CONNECTED_BIT);

            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            esp_wifi_connect();
            xEventGroupClearBits(wifi_event_group, CONNECTED_BIT);
            break;
        default:
            break;
    }
    return ESP_OK;
}


////////////////////////////////// WIFI INITIALIZE ////////////////////////////////////////
static void wifi_init(void)
{
    tcpip_adapter_init();
    wifi_event_group = xEventGroupCreate();
    ESP_ERROR_CHECK(esp_event_loop_init(wifi_event_handler, NULL));
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = CONFIG_WIFI_SSID,
            .password = CONFIG_WIFI_PASSWORD,
        },
    };
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config));
    ESP_LOGI(TAG, "start the WIFI SSID:[%s]", CONFIG_WIFI_SSID);
    ESP_ERROR_CHECK(esp_wifi_start());
    ESP_LOGI(TAG, "Waiting for wifi");
    xEventGroupWaitBits(wifi_event_group, CONNECTED_BIT, false, true, portMAX_DELAY);
}



////////////////////////////////// MQTT APP START ////////////////////////////////////////
static void mqtt_app_start(void)
{
    mqtt_event_group = xEventGroupCreate();
    esp_mqtt_client_config_t mqtt_cfg = {
        .uri = CONFIG_BROKER_URL,
        .port=1883,
        .client_id="EB8QAhQlHDYdMTQLByETGBY",
        .username="EB8QAhQlHDYdMTQLByETGBY",
        .password="pLPZfUVoOwIUi2TVUOYNhrfu",
        .event_handle = mqtt_event_handler
    };

#if CONFIG_BROKER_URL_FROM_STDIN
```

```
    char line[128];

    if (strcmp(mqtt_cfg.uri, "FROM_STDIN") == 0) {
        int count = 0;
        printf("Please enter url of mqtt broker\n");
        while (count < 128) {
            int c = fgetc(stdin);
            if (c == '\n') {
                line[count] = '\0';
                break;
            } else if (c > 0 && c < 127) {
                line[count] = c;
                ++count;
            }
            vTaskDelay(10 / portTICK_PERIOD_MS);
        }
        mqtt_cfg.uri = line;
        printf("Broker url: %s\n", line);
    } else {
        ESP_LOGE(TAG, "Configuration mismatch: wrong broker url");
        abort();
    }
#endif  /* CONFIG_BROKER_URL_FROM_STDIN */

    client = esp_mqtt_client_init(&mqtt_cfg);
    esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);
    esp_mqtt_client_start(client);
}


///////////////////////////////// INITIALIZE UART   //////////////////////////////////////
void uart_init() {
    const uart_config_t uart_config = {
        .baud_rate = 9600,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
    };
    uart_param_config(UART_NUM_1, &uart_config);
    uart_set_pin(UART_NUM_1, TXD_PIN, RXD_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    // We won't use a buffer for sending data.
    uart_driver_install(UART_NUM_1, RX_BUF_SIZE * 2, 0, 0, NULL, 0);
}



///////////////////////////////// SEND DATA ////////////////////////////////////////////////
int sendData(const char* logName, const char* data)
{
    const int len = strlen(data);
    const int txBytes = uart_write_bytes(UART_NUM_1, data, len);
    ESP_LOGI(logName, "Wrote %d bytes", txBytes);
    return txBytes;
}
```

```
/////////////////////////////// TASK DATA COMMS /////////////////////////////////////
static void data_comms(void *pvParameters)
{
    static const char *RX_TASK_TAG = "RX_TASK";
    esp_log_level_set(RX_TASK_TAG, ESP_LOG_INFO);
    uint8_t* data = (uint8_t*) malloc(RX_BUF_SIZE+1);
    uint16_t tank_level = 0;
    uint16_t tank_level_hi = 0;
    uint16_t tank_level_lo = 0;

    while(1)
    {
        xSemaphoreTake(print_mux, portMAX_DELAY);

        //////////////////////////// RX DATA ///////////////////////////////////////////////////////
        const int rxBytes = uart_read_bytes(UART_NUM_1, data, RX_BUF_SIZE, 1000 / portTICK_RATE_MS);

        if (rxBytes > 0)
        {
            data[rxBytes] = 0;

            ESP_LOG_BUFFER_HEXDUMP(RX_TASK_TAG, data, rxBytes, ESP_LOG_INFO);

           if (data[4] == 0x00 && data[5] == 0x13 && data[6] == 0xA2 && data[7] == 0x00 && data[8]
               == 0x41 && data[9] == 0xB5 && data[10] == 0xFE && data[11] == 0xA0)
           {

                tank_level_hi = data[21] << 8;
                tank_level_lo = data[22];
                tank_level = tank_level_hi + tank_level_lo;

                tank_percent = 0.0978 * tank_level;

                printf("Tank level is %i percent \n",tank_percent);

                sprintf(tank_perc_pub, "field1=%u&status=MQTTPUBLISH", tank_percent);
                sprintf(sys_state_pub, "field2=%u&status=MQTTPUBLISH", system_state);

                if (tank_percent > 100)
                {
                    tank_percent = 100;
                }
                if (tank_percent < 16)
                {
                    tank_percent = 0;
                }


                if (tank_percent < 10)
                {
                    system_state = 0;
                }
                if (tank_percent >= 10)
                {
```

```
                system_state = 1;
            }
            change = 1;

        }

    }

    xSemaphoreGive(print_mux);

    //////////////////////////// TX DATA ///////////////////////////////////////////////
    if (change == 1)
    {
        switch (system_state)
        {
        case 0: // Rain tank empty
            uart_write_bytes(UART_NUM_1, solenoid_rain_off, sizeof(solenoid_rain_off));
            vTaskDelay(500 / portTICK_PERIOD_MS);
            uart_write_bytes(UART_NUM_1, solenoid_mains_on, sizeof(solenoid_mains_on));
            printf("Using mains... \n");
            change = 0;
            break;

        case 1: // There is water in the rain tank
            uart_write_bytes(UART_NUM_1, solenoid_mains_off, sizeof(solenoid_mains_off));
            vTaskDelay(500 / portTICK_PERIOD_MS);
            uart_write_bytes(UART_NUM_1, solenoid_rain_on, sizeof(solenoid_rain_on));
            printf("Using rain... \n");
            change = 0;
            break;

        default:
            break;
        }
        change = 0;

    }

  }
  vTaskDelete(NULL);

}



//////////////////////////////// TASK MQTT PUBLISH ////////////////////////////////////
static void mqtt_publish(void *pvParameters)
{
    static int msg_id = 0;

    portTickType xLastWakeTime;

    xLastWakeTime = xTaskGetTickCount();

    // MQTT INIT
    esp_mqtt_client_config_t mqtt_cfg = {
```

```
        .uri = CONFIG_BROKER_URL,
        .port=1883,
        .client_id="EB8QAhQlHDYdMTQLByETGBY",
        .username="EB8QAhQlHDYdMTQLByETGBY",
        .password="pLPZfUVoOwIUi2TVUOYNhrfu",
        .event_handle = mqtt_event_handler
    };

    while(1)
    {
        xSemaphoreTake(print_mux, portMAX_DELAY);
        xEventGroupClearBits(mqtt_event_group, CONNECTED_BIT);

        esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);

        esp_mqtt_client_start(client);

        xEventGroupWaitBits(mqtt_event_group, CONNECTED_BIT, false, true, portMAX_DELAY);

        msg_id = esp_mqtt_client_publish(client, "channels/1572371/publish", tank_perc_pub, 0,
        0, 0);
        ESP_LOGI(TAG, "sent publish successful, msg_id=%d", msg_id);

        msg_id = esp_mqtt_client_publish(client, "channels/1572371/publish", sys_state_pub, 0,
        0, 0);
        ESP_LOGI(TAG, "sent publish successful, msg_id=%d", msg_id);

        esp_mqtt_client_stop(client);

        xSemaphoreGive(print_mux);

        // 10 s between every run
        vTaskDelayUntil( &xLastWakeTime, (10000 / portTICK_RATE_MS));

    }
    vTaskDelete(NULL);

}



///////////////////////////////// MAIN ////////////////////////////////////////////////////
void app_main()
{
    print_mux = xSemaphoreCreateMutex();

    ESP_LOGI(TAG, "[APP] Startup..");
    ESP_LOGI(TAG, "[APP] Free memory: %d bytes", esp_get_free_heap_size());
    ESP_LOGI(TAG, "[APP] IDF version: %s", esp_get_idf_version());

    esp_log_level_set("*", ESP_LOG_INFO);
    esp_log_level_set("MQTT_CLIENT", ESP_LOG_VERBOSE);
    esp_log_level_set("MQTT_EXAMPLE", ESP_LOG_VERBOSE);
    esp_log_level_set("TRANSPORT_TCP", ESP_LOG_VERBOSE);
    esp_log_level_set("TRANSPORT_SSL", ESP_LOG_VERBOSE);
    esp_log_level_set("TRANSPORT", ESP_LOG_VERBOSE);
```

```
    esp_log_level_set("OUTBOX", ESP_LOG_VERBOSE);

    ESP_ERROR_CHECK(esp_event_loop_create_default());

    uart_init();
    nvs_flash_init();
    wifi_init();
    mqtt_app_start();

    // xTaskCreate(pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCreatedTask)
    xTaskCreate(      data_comms,  "Data communication", 4048,  (void *)0,  3,         NULL);
    xTaskCreate(      mqtt_publish,  "MQTT Publish",     4048,  (void *)0,  2,         NULL);

}
```