IOT PROJECT PHASE 2

TITLE: DEVELOPING SMART TOILETS USING IOT

1.INTRODUCTION

In our country, people do not have enough knowledge of using toilets. This leads to several diseases, such as Malaria, Hepatitis, Flu, Cholera, Streptococcus, Typhoid, etc. we introduce the concept in the IOT called "Swachh Shithouse" The term Swachh means 'Clean'. Then the term Shithouse means 'Toilet'. It is introduced to use and maintain the toilets in the clean and hygienic way. The project is based on IOT concepts using different sensors like smell sensor, dirt sensor, sonic sensor, RFID reader, Database. Using this material, we are trying to provide the clean toilets and create the awareness among the people.

In the second phase of the project using IOT, in this paper we are going to provide the clean toilet. This paper can create the awareness among the people about the clean and hygienic toilets. This paper can ensure the responsibilities of the sweeper. Finally, this concept is the one of the stepping stone to the "Clean and diseases free India".

There are two parts are involved here. They are,

1. Automatic Flusher Part (AFP)

2. Server part

HARDWARE REQUIREMENTS:

- Microcontroller
- Power supply
- LCD display
- Buzzer
- Infrared sensor
- Sonic sensor
- Gas sensor
- RFID
- GSM modem

SOFTWARE REQUIREMENTS:

- Embedded C

WORKING PRINCIPLE:

- In the first phase, IR sensor is used to discover the dirt present in the toilet.
- Here the set of sample images are given as input.
- After using the toilet, the sensor senses the basin of the toilet.
- Then it relates the sensed image with the input image.
- If the dirt present, it increases the alarm.
- Then the user wants to be clean the waste. Through this activity, people can get the awareness about the toilet management.
- In the second phase, Figaro sensor is used to perceive the unwanted gases present in the toilet.
- In the Figaro sensor, a particular range is to be stable earlier manner. If the range gets extended, it can send the alert message to the sweeper. Then they cleaned it by using proper fragrant.
- In the third phase, RFID reader (Radio Frequency Identification) is used to observe the sweeper's activities (absence and presence in the toilet cleaning).
- Initially, the sweeper wants to show his/her individuality tag in front of RFID reader. It can be shown before and after cleaning the toilet.

- Then the first phase gets initiated and senses for the dirt presence in the toilet.
- If the dirt gets noticed, it raises the alarm.
- Through this monitoring activity, the sweeper can realize their roles and responsibilities. Then they protect the people by disposing all the unwanted materials (dirt, unwanted gases) present in the toilet.
- In the final phase, the sonic sensor is used to detect the depth of the septic tank.
- Here, the range of septic tank is fixed prior manner.
- If the sewage reached with the range, then it directs message to an organization.
- All the message transfer can be done by the GSM (Global System for Communication)

Block Diagram:

Program Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <unistd.h>

#include <limits.h>

#include <string.h>

#include <math.h>


#include "freertos/FreeRTOS.h"

#include "freertos/task.h"

#include "freertos/semphr.h"

#include "freertos/event_groups.h"

#include "esp_log.h"


#include "aws_iot_config.h"

#include "aws_iot_log.h"

#include "aws_iot_version.h"
```

```
#include "aws_iot_mqtt_client_interface.h"
#include "aws_iot_shadow_interface.h"

#include "core2forAWS.h"

#include "wifi.h"
#include "fft.h"
#include "ui.h"

#include "ads1115.h"
#include "dht.h"

static const char *TAG = "MAIN";

#define READY   "READY"
#define UNCLEAN "UNCLEAN"
#define BUSY  "BUSY"
#define INITIAL_STATE "INITIAL_STATE"

#define STARTING_ROOMTEMPERATURE 0.0f
#define STARTING_ROOMHUMIDITY 0.0f
#define STARTING_ROOMCO 0.0f
#define STARTING_ROOMNO2 0.0f
#define STARTING_ROOMNH3 0.0f
#define STARTING_ROOMSO2 0.0f
#define STARTING_ROOMCH4 0.0f
#define STARTING_SOUNDLEVEL 0x00
#define STARTING_FANSTATUS false
#define STARTING_TOILETSTATUS INITIAL_STATE

#define CO_CHNNEL 0
#define NH3_CHNNEL 1
```

```
#define NO2_CHNNEL 2
#define SO2_CHNNEL 3

static const dht_sensor_type_t sensor_type = DHT_TYPE_DHT11;
static const gpio_num_t dht_gpio = GPIO_NUM_26;



extern const uint8_t aws_root_ca_pem_start[] asm("_binary_aws_root_ca_pem_start");
extern const uint8_t aws_root_ca_pem_end[] asm("_binary_aws_root_ca_pem_end");

char HostAddress[255] = AWS_IOT_MQTT_HOST;

uint32_t port = AWS_IOT_MQTT_PORT;

SemaphoreHandle_t xMaxNoiseSemaphore;

void iot_subscribe_callback_handler(AWS_IoT_Client *pClient, char *topicName, uint16_t topicNameLen,
                                    IoT_Publish_Message_Params *params, void *pData) {
    ESP_LOGI(TAG, "Subscribe callback");
    ESP_LOGI(TAG, "%.*s\t%.*s", topicNameLen, topicName, (int) params->payloadLen, (char *)params->payload);
}

void disconnect_callback_handler(AWS_IoT_Client *pClient, void *data) {
    ESP_LOGW(TAG, "MQTT Disconnect");
    ui_textarea_add("Disconnected from AWS IoT Core...", NULL, 0);

    IoT_Error_t rc = FAILURE;

    if(NULL == pClient) {
        return;
    }
```

```
    if(aws_iot_is_autoreconnect_enabled(pClient)) {
        ESP_LOGI(TAG, "Auto Reconnect is enabled, Reconnecting attempt will start now");
    } else {
        ESP_LOGW(TAG, "Auto Reconnect not enabled. Starting manual reconnect...");
        rc = aws_iot_mqtt_attempt_reconnect(pClient);
        if(NETWORK_RECONNECTED == rc) {
            ESP_LOGW(TAG, "Manual Reconnect Successful");
        } else {
            ESP_LOGW(TAG, "Manual Reconnect Failed - %d", rc);
        }
    }
}


static bool shadowUpdateInProgress;


void ShadowUpdateStatusCallback(const char *pThingName, ShadowActions_t action,
Shadow_Ack_Status_t status,
                                const char *pReceivedJsonDocument, void *pContextData) {
    IOT_UNUSED(pThingName);
    IOT_UNUSED(action);
    IOT_UNUSED(pReceivedJsonDocument);
    IOT_UNUSED(pContextData);


    shadowUpdateInProgress = false;


    if(SHADOW_ACK_TIMEOUT == status) {
        ESP_LOGE(TAG, "Update timed out");
    } else if(SHADOW_ACK_REJECTED == status) {
        ESP_LOGE(TAG, "Update rejected");
    } else if(SHADOW_ACK_ACCEPTED == status) {
        ESP_LOGI(TAG, "Update accepted");
    }
}
```

```
void exhaustFan_Callback(const char *pJsonString, uint32_t JsonStringDataLen, jsonStruct_t *pContext) {
    IOT_UNUSED(pJsonString);
    IOT_UNUSED(JsonStringDataLen);

    //char * status = (char *) (pContext->pData);
    //bool * status = (bool *) (pContext->pData);
    bool status = (bool *) (pContext->pData);

    if(pContext != NULL) {
        ESP_LOGI(TAG, "Delta - fanStatus state changed to %s", status? "ON":"OFF");
    }

    //if(strcmp(status, HEATING) == 0) {
    if(status == false) {
        ESP_LOGI(TAG, "setting one side LEDs to green");
        //Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_LEFT, 0x00FF00);
        //Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_RIGHT, 0xFF0000);
        //Core2ForAWS_Sk6812_Show();
    } else if(status == true) {
        //ESP_LOGI(TAG, "setting one side LEDs to orange");
        //Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_LEFT, 0x00FFFF);
        //Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_RIGHT, 0x0000FF);
        //Core2ForAWS_Sk6812_Show();
    } else {
        //ESP_LOGI(TAG, "clearing side LEDs");
        //Core2ForAWS_Sk6812_Clear();
        //Core2ForAWS_Sk6812_Show();
    }
}

void toilet_status_Callback(const char *pJsonString, uint32_t JsonStringDataLen, jsonStruct_t *pContext) {
```

```
    IOT_UNUSED(pJsonString);
    IOT_UNUSED(JsonStringDataLen);

    char * status = (char *) (pContext->pData);

    if(pContext != NULL) {
        ESP_LOGI(TAG, "Delta - toiletStatus state changed to %s", status);
    }

    if(strcmp(status, BUSY) == 0) {
        ESP_LOGI(TAG, "setting side LEDs to Yellow");
        Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_LEFT, 0xFFFF00);
        Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_RIGHT, 0xFFFF00);
        Core2ForAWS_Sk6812_Show();
    } else if(strcmp(status, UNCLEAN) == 0) {
        ESP_LOGI(TAG, "setting side LEDs to Red");
        Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_LEFT, 0xFF0000);
        Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_RIGHT, 0xFF0000);
        Core2ForAWS_Sk6812_Show();
    } else if(strcmp(status, READY) == 0) {
        ESP_LOGI(TAG, "clearing side Green");
        Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_LEFT, 0x00FF00);
        Core2ForAWS_Sk6812_SetSideColor(SK6812_SIDE_RIGHT, 0x00FF00);
        //Core2ForAWS_Sk6812_Clear();
        Core2ForAWS_Sk6812_Show();
    }
}

float temperature = STARTING_ROOMTEMPERATURE;
float humidity = STARTING_ROOMHUMIDITY;
float nitrogen_dioxide = STARTING_ROOMNO2;
float ammonia = STARTING_ROOMNH3;
```

```
float carbon_monoxide = STARTING_ROOMCO;
float sulfur_dioxide = STARTING_ROOMSO2;
float methane = STARTING_ROOMCH4;
uint8_t soundBuffer = STARTING_SOUNDLEVEL;
uint8_t reportedSound = STARTING_SOUNDLEVEL;
bool fan_status = STARTING_FANSTATUS;
char toilet_status[14] = STARTING_TOILETSTATUS;

void read_temperature(){

    int16_t temperature_data = 0;
    int16_t humidity_data = 0;

    if (dht_read_data(sensor_type, dht_gpio, &humidity_data, &temperature_data) == ESP_OK){
        temperature = (float) temperature_data/10;
        humidity = (float) humidity_data/10;
    }

}

void read_airquality(){

    int16_t adc0, adc1, adc2;
    //float nitrogen_dioxide, ammonia, carbon_monoxide;
    adc0 = ADS1115_readADC_SingleEnded(CO_CHNNEL);
    carbon_monoxide = ADS1115_computeVolts(adc0);
    adc1 = ADS1115_readADC_SingleEnded(NH3_CHNNEL);
    ammonia = ADS1115_computeVolts(adc1);
    adc2 = ADS1115_readADC_SingleEnded(NO2_CHNNEL);
    nitrogen_dioxide = ADS1115_computeVolts(adc2);

}
```

```
uint16_t _baseNH3;
uint16_t _baseCO;
uint16_t _baseNO2;

typedef enum
{
CH_CO,
CH_NO2,
CH_NH3
} channel_t;

typedef enum
{
CO,
NO2,
NH3,
   CH4
} gas_t;

void airquality_calibrate ()
{
// The number of seconds that must pass before
// than we will assume that the calibration is complete
// (Less than 64 seconds to avoid overflow)
uint8_t seconds = 10;

// Tolerance for the average of the current value
uint8_t delta = 2;

// Measurement buffers
```

```
uint16_t bufferNH3 [seconds];
uint16_t bufferCO [seconds];
uint16_t bufferNO2 [seconds];

// Pointers for the next item in the buffer
uint8_t pntrNH3 = 0;
uint8_t pntrCO = 0;
uint8_t pntrNO2 = 0;

// The current floating amount in the buffer
   uint32_t fltSumNH3 = 0;
uint32_t fltSumCO = 0;
uint32_t fltSumNO2 = 0;

// Current measurement
uint16_t curNH3;
uint16_t curCO;
uint16_t curNO2;

// Flag of stability of indications
bool isStableNH3 = false;
bool isStableCO = false;
bool isStableNO2 = false;

// We kill the buffer with zeros
for (int i = 0; i <seconds; ++ i)
{
bufferNH3 [i] = 0;
bufferCO [i] = 0;
bufferNO2 [i] = 0;
}
```

```
// Calibrate
do
{
vTaskDelay(pdMS_TO_TICKS(1000));

unsigned long rs = 0;

vTaskDelay(pdMS_TO_TICKS(50));
for (int i = 0; i <3; i ++)
{
vTaskDelay(pdMS_TO_TICKS(1));
rs += ADS1115_readADC_SingleEnded(NH3_CHNNEL);
}

curNH3 = rs / 3;  //printf("cur NH3 Rs : %d", curNH3);
rs = 0;

vTaskDelay(pdMS_TO_TICKS(50));
for (int i = 0; i <3; i ++)
{
vTaskDelay(pdMS_TO_TICKS(1));
rs += ADS1115_readADC_SingleEnded(CO_CHNNEL);
}

curCO = rs / 3;  //printf("cur co Rs : %d", curCO);
rs = 0;

vTaskDelay(pdMS_TO_TICKS(50));
for (int i = 0; i <3; i ++)
{
vTaskDelay(pdMS_TO_TICKS(1));
rs += ADS1115_readADC_SingleEnded(NO2_CHNNEL);
```

```
}

curNO2 = rs / 3; //printf("cur NO2 Rs : %d", curNO2);

// Update the floating amount by subtracting the value,
// to be overwritten, and adding a new value.

fltSumNH3 = fltSumNH3 + curNH3 - bufferNH3 [pntrNH3];
fltSumCO = fltSumCO + curCO - bufferCO [pntrCO];
fltSumNO2 = fltSumNO2 + curNO2 - bufferNO2 [pntrNO2];
      //printf("\tNH3 %d\n", fltSumNH3);

// Store d buffer new values
bufferNH3 [pntrNH3] = curNH3;
bufferCO [pntrCO] = curCO;
bufferNO2 [pntrNO2] = curNO2;

// Define flag states
isStableNH3 = abs (fltSumNH3 / seconds - curNH3) <delta;
isStableCO = abs (fltSumCO / seconds - curCO) <delta;
isStableNO2 = abs (fltSumNO2 / seconds - curNO2) <delta;

// Pointer to a buffer
pntrNH3 = (pntrNH3 + 1)% seconds;
pntrCO = (pntrCO + 1)% seconds;
pntrNO2 = (pntrNO2 + 1)% seconds;
} while (! isStableNH3 ||! isStableCO ||! isStableNO2);

_baseNH3 = fltSumNH3 / seconds;
_baseCO = fltSumCO / seconds;
_baseNO2 = fltSumNO2 / seconds;
}
```

```
uint16_t getBaseResistance (channel_t channel)
{
switch (channel)
{
case CH_NH3:
return _baseNH3;
case CH_CO:
return _baseCO;
case CH_NO2:
return _baseNO2;
}

return 0;
}

uint16_t getResistance (channel_t channel)
{
unsigned long rs = 0;
int counter = 0;
   int i = 0;

switch (channel)
{
case CH_CO:
for (i = 0; i <100; i ++)
{
rs += ADS1115_readADC_SingleEnded(CO_CHNNEL);
counter ++;
vTaskDelay(pdMS_TO_TICKS(2));
}
      break;
```

```
case CH_NO2:
for (i = 0; i <100; i ++)
{
rs += ADS1115_readADC_SingleEnded(NO2_CHNNEL);
counter ++;
vTaskDelay(pdMS_TO_TICKS(2));
}
      break;
case CH_NH3:
for (i = 0; i <100; i ++)
{
rs += ADS1115_readADC_SingleEnded(NH3_CHNNEL);
counter ++;
vTaskDelay(pdMS_TO_TICKS(2));
}
      break;
}

return counter != 0? rs / counter: 0;
}

float getCurrentRatio (channel_t channel)
{
float baseResistance = (float) getBaseResistance (channel);
float resistance = (float) getResistance (channel);

return resistance / baseResistance * (1023.0 - baseResistance) / (1023.0 - resistance);

return -1.0;
}

float measure_in_ppm (gas_t gas)
```

```
{
float ratio;
float c = 0;

switch (gas)
{
case CO:
ratio = getCurrentRatio (CH_CO);
c = pow (ratio, -1.179) * 4.385;
break;
case NO2:
ratio = getCurrentRatio (CH_NO2);
c = pow (ratio, 1.007) / 6.855;
break;
case NH3:
ratio = getCurrentRatio (CH_NH3);
c = pow (ratio, -1.67) / 1.47;
break;
  case CH4:
    ratio = getCurrentRatio (CH_CO);
    c = pow (ratio, -4.093) * 0.837;
    break;
}

return isnan (c)? -1: c;
}

void read_airquality_ppm(){

    carbon_monoxide = measure_in_ppm(CO);
    nitrogen_dioxide = measure_in_ppm(NO2);
    ammonia = measure_in_ppm(NH3);
```

```
    methane = measure_in_ppm(CH4);

}

void read_sulfur_dioxide(){

    int rl = 10;
    float r0 = 76.63;
    int value = ADS1115_readADC_SingleEnded(SO2_CHNNEL);
    float rs = ( ( 5.0 * rl ) - ( rl * value ) ) / value;
    float ratio = rs/r0;
    ratio = ratio * 0.3611;
    float SO2_PPM = (146.15 * (2.868 - ratio) + 10);
    sulfur_dioxide = SO2_PPM;
}



// helper function for working with audio data
long map(long x, long in_min, long in_max, long out_min, long out_max) {
    long divisor = (in_max - in_min);
    if(divisor == 0){
        return -1; //AVR returns -1, SAM returns 0
    }
    return (x - in_min) * (out_max - out_min) / divisor + out_min;
}

void microphone_task(void *arg) {
    static int8_t i2s_readraw_buff[1024];
    size_t bytesread;
    int16_t *buffptr;
    double data = 0;
```

```
    Microphone_Init();
    uint8_t maxSound = 0x00;
    uint8_t currentSound = 0x00;

    for (;;) {
        maxSound = 0x00;
        fft_config_t *real_fft_plan = fft_init(512, FFT_REAL, FFT_FORWARD, NULL, NULL);
        i2s_read(I2S_NUM_0, (char *)i2s_readraw_buff, 1024, &bytesread, pdMS_TO_TICKS(100));
        buffptr = (int16_t *)i2s_readraw_buff;
        for (uint16_t count_n = 0; count_n < real_fft_plan->size; count_n++) {
            real_fft_plan->input[count_n] = (float)map(buffptr[count_n], INT16_MIN, INT16_MAX, -1000, 1000);
        }
        fft_execute(real_fft_plan);

        for (uint16_t count_n = 1; count_n < AUDIO_TIME_SLICES; count_n++) {
            data = sqrt(real_fft_plan->output[2 * count_n] * real_fft_plan->output[2 * count_n] + real_fft_plan-
>output[2 * count_n + 1] * real_fft_plan->output[2 * count_n + 1]);
            currentSound = map(data, 0, 2000, 0, 256);
            if(currentSound > maxSound) {
                maxSound = currentSound;
            }
        }
        fft_destroy(real_fft_plan);

        // store max of sample in semaphore
        xSemaphoreTake(xMaxNoiseSemaphore, portMAX_DELAY);
        soundBuffer = maxSound;
        xSemaphoreGive(xMaxNoiseSemaphore);
    }
}

void aws_iot_task(void *param) {
    IoT_Error_t rc = FAILURE;
```

```
char JsonDocumentBuffer[MAX_LENGTH_OF_UPDATE_JSON_BUFFER];
size_t sizeOfJsonDocumentBuffer = sizeof(JsonDocumentBuffer) / sizeof(JsonDocumentBuffer[0]);

jsonStruct_t temperatureHandler;
temperatureHandler.cb = NULL;
temperatureHandler.pKey = "temperature";
temperatureHandler.pData = &temperature;
temperatureHandler.type = SHADOW_JSON_FLOAT;
temperatureHandler.dataLength = sizeof(float);

jsonStruct_t humidityHandler;
humidityHandler.cb = NULL;
humidityHandler.pKey = "humidity";
humidityHandler.pData = &humidity;
humidityHandler.type = SHADOW_JSON_FLOAT;
humidityHandler.dataLength = sizeof(float);

jsonStruct_t carbonMonoxideHandler;
carbonMonoxideHandler.cb = NULL;
carbonMonoxideHandler.pKey = "carbon_monoxide";
carbonMonoxideHandler.pData = &carbon_monoxide;
carbonMonoxideHandler.type = SHADOW_JSON_FLOAT;
carbonMonoxideHandler.dataLength = sizeof(float);

jsonStruct_t ammoniaHandler;
ammoniaHandler.cb = NULL;
ammoniaHandler.pKey = "ammonia";
ammoniaHandler.pData = &ammonia;
ammoniaHandler.type = SHADOW_JSON_FLOAT;
ammoniaHandler.dataLength = sizeof(float);
```

```
jsonStruct_t nitrogenDioxideHandler;
nitrogenDioxideHandler.cb = NULL;
nitrogenDioxideHandler.pKey = "nitrogen_dioxide";
nitrogenDioxideHandler.pData = &nitrogen_dioxide;
nitrogenDioxideHandler.type = SHADOW_JSON_FLOAT;
nitrogenDioxideHandler.dataLength = sizeof(float);

jsonStruct_t sulfurDioxideHandler;
sulfurDioxideHandler.cb = NULL;
sulfurDioxideHandler.pKey = "sulfur_dioxide";
sulfurDioxideHandler.pData = &sulfur_dioxide;
sulfurDioxideHandler.type = SHADOW_JSON_FLOAT;
sulfurDioxideHandler.dataLength = sizeof(float);

jsonStruct_t methaneHandler;
methaneHandler.cb = NULL;
methaneHandler.pKey = "methane";
methaneHandler.pData = &methane;
methaneHandler.type = SHADOW_JSON_FLOAT;
methaneHandler.dataLength = sizeof(float);

jsonStruct_t soundHandler;
soundHandler.cb = NULL;
soundHandler.pKey = "sound";
soundHandler.pData = &reportedSound;
soundHandler.type = SHADOW_JSON_UINT8;
soundHandler.dataLength = sizeof(uint8_t);

jsonStruct_t exhaustFanActuator;
exhaustFanActuator.cb = exhaustFan_Callback;
exhaustFanActuator.pKey = "fan_status";
exhaustFanActuator.pData = &fan_status;
```

```
    exhaustFanActuator.type = SHADOW_JSON_BOOL;
    exhaustFanActuator.dataLength = sizeof(bool);

    jsonStruct_t toiletStatusActuator;
    toiletStatusActuator.cb = toilet_status_Callback;
    toiletStatusActuator.pKey = "toilet_status";
    toiletStatusActuator.pData = &toilet_status;
    toiletStatusActuator.type = SHADOW_JSON_STRING;
    toiletStatusActuator.dataLength = strlen(toilet_status)+1;

    ESP_LOGI(TAG, "AWS IoT SDK Version %d.%d.%d-%s", VERSION_MAJOR, VERSION_MINOR,
VERSION_PATCH, VERSION_TAG);

    // initialize the mqtt client
    AWS_IoT_Client iotCoreClient;

    ShadowInitParameters_t sp = ShadowInitParametersDefault;
    sp.pHost = HostAddress;
    sp.port = port;
    sp.enableAutoReconnect = false;
    sp.disconnectHandler = disconnect_callback_handler;

    sp.pRootCA = (const char *)aws_root_ca_pem_start;
    sp.pClientCRT = "#";
    sp.pClientKey = "#0";

    #define CLIENT_ID_LEN (ATCA_SERIAL_NUM_SIZE * 2)
    char *client_id = malloc(CLIENT_ID_LEN + 1);
    ATCA_STATUS ret = Atecc608_GetSerialString(client_id);
    if (ret != ATCA_SUCCESS){
        ESP_LOGE(TAG, "Failed to get device serial from secure element. Error: %i", ret);
        abort();
    }
```

```
    ui_textarea_add("\nDevice client Id:\n>> %s <<\n", client_id, CLIENT_ID_LEN);

    /* Wait for WiFI to show as connected */
    xEventGroupWaitBits(wifi_event_group, CONNECTED_BIT,
                        false, true, portMAX_DELAY);

    ESP_LOGI(TAG, "Shadow Init");

    rc = aws_iot_shadow_init(&iotCoreClient, &sp);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "aws_iot_shadow_init returned error %d, aborting...", rc);
        abort();
    }

    ShadowConnectParameters_t scp = ShadowConnectParametersDefault;
    scp.pMyThingName = client_id;
    scp.pMqttClientId = client_id;
    scp.mqttClientIdLen = CLIENT_ID_LEN;

    ESP_LOGI(TAG, "Shadow Connect");
    rc = aws_iot_shadow_connect(&iotCoreClient, &scp);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "aws_iot_shadow_connect returned error %d, aborting...", rc);
        abort();
    }
    ui_textarea_add("\nConnected to AWS IoT Core and pub/sub to the device shadow state\n", NULL, 0);

    xTaskCreatePinnedToCore(&microphone_task, "microphone_task", 4096, NULL, 1, NULL, 1);

    /*
     * Enable Auto Reconnect functionality. Minimum and Maximum time of Exponential backoff are set in
aws_iot_config.h
```

```
     *  #AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL
     *  #AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL
     */
    rc = aws_iot_shadow_set_autoreconnect_status(&iotCoreClient, true);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "Unable to set Auto Reconnect to true - %d, aborting...", rc);
        abort();
    }

    // register delta callback for roomOccupancy
    rc = aws_iot_shadow_register_delta(&iotCoreClient, &toiletStatusActuator);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "Shadow Register Delta Error");
    }

    // register delta callback for fanStatus
    rc = aws_iot_shadow_register_delta(&iotCoreClient, &exhaustFanActuator);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "Shadow Register Delta Error");
    }

    // loop and publish changes
    while(NETWORK_ATTEMPTING_RECONNECT == rc || NETWORK_RECONNECTED == rc || SUCCESS ==
rc) {
        rc = aws_iot_shadow_yield(&iotCoreClient, 200);
        if(NETWORK_ATTEMPTING_RECONNECT == rc || shadowUpdateInProgress) {
            rc = aws_iot_shadow_yield(&iotCoreClient, 1000);
            // If the client is attempting to reconnect, or already waiting on a shadow update,
            // we will skip the rest of the loop.
            continue;
        }

        // START get sensor readings
```

```
        // sample temperature, convert to fahrenheit
        //MPU6886_GetTempData(&temperature);
        //temperature = (temperature * 1.8)  + 32 - 50;

/*********************************************************************************************
*/
        read_temperature();
        //read_airquality();
        read_airquality_ppm();
        read_sulfur_dioxide();

/*********************************************************************************************
*/
        // sample from soundBuffer (latest reading from microphone)
        xSemaphoreTake(xMaxNoiseSemaphore, portMAX_DELAY);
        reportedSound = soundBuffer;
        xSemaphoreGive(xMaxNoiseSemaphore);


        // END get sensor readings


        ESP_LOGI(TAG,
"*********************************************************************************************
*");
        ESP_LOGI(TAG, "On Device: Toilet status: %s", toilet_status);
        ESP_LOGI(TAG, "On Device: Fan status: %s", fan_status? "ON" : "OFF");
        ESP_LOGI(TAG, "On Device: Temperature: %f C", temperature);
        ESP_LOGI(TAG, "On Device: Sound: %d", reportedSound);
        ESP_LOGI(TAG, "On Device: Humidity: %f %%", humidity);
        ESP_LOGI(TAG, "On Device: Carbon monoxide: %f ppm", carbon_monoxide);
        ESP_LOGI(TAG, "On Device: Ammonia: %f ppm", ammonia);
        ESP_LOGI(TAG, "On Device: Nitrogen dioxide: %f ppm", nitrogen_dioxide);
        ESP_LOGI(TAG, "On Device: Sulfur dioxide: %f ppm", sulfur_dioxide);
        ESP_LOGI(TAG, "On Device: Methane: %f ppm", methane);
        ESP_LOGI(TAG,
"*********************************************************************************************
*");
```

```
      rc = aws_iot_shadow_init_json_document(JsonDocumentBuffer, sizeOfJsonDocumentBuffer);

      if(SUCCESS == rc) {

         rc = aws_iot_shadow_add_reported(JsonDocumentBuffer, sizeOfJsonDocumentBuffer, 10,
&temperatureHandler,

                                          &humidityHandler, &carbonMonoxideHandler, &ammoniaHandler,
&nitrogenDioxideHandler, &sulfurDioxideHandler,

                                          &methaneHandler, &soundHandler, &toiletStatusActuator,
&exhaustFanActuator);

         if(SUCCESS == rc) {

            rc = aws_iot_finalize_json_document(JsonDocumentBuffer, sizeOfJsonDocumentBuffer);

            if(SUCCESS == rc) {

               ESP_LOGI(TAG, "Update Shadow: %s", JsonDocumentBuffer);

               rc = aws_iot_shadow_update(&iotCoreClient, client_id, JsonDocumentBuffer,

                                          ShadowUpdateStatusCallback, NULL, 10, true);

               shadowUpdateInProgress = true;

            }

         }

      }

      ESP_LOGI(TAG,
"*******************************************************************************************
*");

      ESP_LOGI(TAG, "Stack remaining for task '%s' is %d bytes", pcTaskGetTaskName(NULL),
uxTaskGetStackHighWaterMark(NULL));



      vTaskDelay(pdMS_TO_TICKS(15000));

   }



   if(SUCCESS != rc) {

      ESP_LOGE(TAG, "An error occurred in the loop %d", rc);

   }



   ESP_LOGI(TAG, "Disconnecting");

   rc = aws_iot_shadow_disconnect(&iotCoreClient);
```

```
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "Disconnect error %d", rc);
    }

    vTaskDelete(NULL);
}

void app_main()
{
    Core2ForAWS_Init();
    Core2ForAWS_Display_SetBrightness(80);
    Core2ForAWS_LED_Enable(1);

    ADS1115_I2CInit();
    ADS1115_setGain(GAIN_TWOTHIRDS);
    airquality_calibrate ();

    xMaxNoiseSemaphore = xSemaphoreCreateMutex();

    ui_init();
    initialise_wifi();

    xTaskCreatePinnedToCore(&aws_iot_task, "aws_iot_task", 4096*2, NULL, 5, NULL, 1);
}
```

Conclusion:

In this project, we explored multiple regression models to predict the usable condition of the restroom.

Team mates :

P.S.LAKSHANA

A.LEKHASRI

M.VIJAYALAKSHMI

S.M.MOUNIKA

Z.FARHEEN FATHIMA