**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:**

Power supply

RFID

**Micro controller**

Buzzer

IR Sensor

Sonic sensor

Gas Sensor

LCD Display

GSM

**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**