**Raspberry-Pi-Project**

**Table of Contents**

[1. NetSim Simulation of IoT Networks with Real-world Sensor Data from Raspberry Pi 2](#_Toc141522878)

[Introduction 2](#_Toc141522879)

[Project overview 2](#_Toc141522880)

[Objective 2](#_Toc141522881)

[Scope 2](#_Toc141522882)

[Sensor Integration with Raspberry Pi 2](#_Toc141522883)

[Sensor Description 2](#_Toc141522884)

[Wiring and Connection to Raspberry Pi 4](#_Toc141522885)

[Communication Protocols Used 5](#_Toc141522886)

[How sensor data is acquired and processed by the Raspberry Pi 6](#_Toc141522887)

[Server Implementation 6](#_Toc141522888)

[Explanation of server.c code 6](#_Toc141522889)

[How the server collects and handles data from the sensors 6](#_Toc141522890)

[Communication between Raspberry Pi and the NetSim Simulator 7](#_Toc141522891)

[NetSim Simulator Configuration 7](#_Toc141522892)

[Overview of NetSim Simulator and its features 7](#_Toc141522893)

[Client.c code in the NetSim simulator 8](#_Toc141522894)

[Creating and Running Sensor Network Scenario in NetSim 9](#_Toc141522895)

[Results and Discussions 11](#_Toc141522896)

[Appendix: NetSim asource code modifications 13](#_Toc141522897)

# 1. NetSim Simulation of IoT Networks with Real-world Sensor Data from Raspberry Pi

## Introduction

### Project overview

* The project aims to integrate real-world sensor data from Raspberry Pi into an IoT network simulation of NetSim,enabling real-time monitoring and analysis.
* The key components of the project are the Raspberry Pi, four sensors (temperature, humidity, ultrasonic, and light-dependent sensor), and the NetSim simulator.
* The sensors will be connected to the Raspberry Pi using the GPIO pins. The data from the sensors will be collected and transmitted to the NetSim simulator.

### Objective

* Demonstrate the seamless functionality of the integrated sensor network within the NetSim simulation environment.
* Evaluate the performance and reliability of the integrated sensor network under different scenarios.

### Scope

* The project includes developing code to read data from real-world sensors (temperature, humidity, ultrasonic, and light-dependent sensor) connected to the Raspberry Pi.
* Analyzing the performance of the integrated sensor network under simulated scenarios and network conditions

## Sensor Integration with Raspberry Pi

### ****Sensor Description****

In this section, we will discuss the sensors integrated with the Raspberry Pi and how their data is acquired and processed. The sensors included are:

A blue electronic device with red buttons

Description automatically generated

Figure 1‑1: Different type of Sensors

**a) Ultrasonic Sensor:**

The ultrasonic sensor is a distance measuring device that uses sound waves to determine the distance between the sensor and an object.

**Functionality:** The ultrasonic sensor emits high-frequency sound pulses and measures the time taken for the echo to return after bouncing off an object.

**Real-World Applications:** Ultrasonic sensors are commonly used in robotics, automotive parking systems, industrial automation, and object detection applications.

**b) DHT11 Sensor:**

The DHT11 is a digital temperature and humidity sensor that provides reliable and low-cost sensing capabilities

* The temperature sensor is designed to measure the ambient temperature of its surroundings. It provides accurate temperature readings, making it ideal for various environmental monitoring applications.

**Functionality**: The temperature sensor operates by detecting changes in temperature, which cause variations in its electrical properties.

**Real-World Applications:** The temperature sensor finds applications in weather monitoring, indoor climate control, industrial automation, and more.

* The humidity sensor, also known as a hygrometer, measures the moisture content or relative humidity in the atmosphere

**Functionality:** The humidity sensor functions by sensing the changes in the moisture level, which affects its electrical characteristics.

**Real-World Applications:** Humidity sensors are widely used in weather stations, greenhouses, HVAC systems, and industries where humidity control is critical.

**c) LDR Sensor:**

The light-dependent sensor, also known as an LDR (Light-Dependent Resistor) or photoresistor, detects changes in ambient light intensity.

**Functionality:** The LDR's resistance changes with variations in light intensity, allowing it to detect the presence or absence of light.

**Real-World Applications:** Light-dependent sensors are used in streetlights, security systems, camera exposure control, and automatic lighting systems.

### Wiring and Connection to Raspberry Pi

All sensors (Ultrasonic, DHT11, and LDR) are connected to the Raspberry Pi using specific GPIO pins as follows:

1. **Ultrasonic Sensor**

**VCC** Pin connected to 5v (Physical Pin 2)

**TRIG** Pin connected to GPIO 17 (Physical Pin 11)

**ECHO** Pin connected to GPIO 27 (Physical Pin 13)

**GND** Pin connected GND (Physical Pin 6)

1. **DHT11 Sensor (Temperature and Humidity)**

**VCC** Pin connected to 5v (Physical Pin 4)

**DATA** Pin connected to GPIO 22 (Physical Pin 15)

**GND** Pin connected GND (Physical Pin 20)

1. **Light-Dependent Resistor (LDR):**

**VCC** Pin connected to 3v3 (Physical Pin 1)

**DATA** Pin connected to GPIO 04 (Physical Pin 7)

**GND** Pin connected GND (Physical Pin 39)

A circuit board with many wires

Description automatically generated

Figure 1‑2: Connection Diagram of Sensors

### Communication Protocols Used

1. **Ultrasonic Sensor**

GPIO (General Purpose Input/Output) pins are used to communicate with the Ultrasonic sensor.One GPIO pin triggers the sensor, and another GPIO pin receives the echo signal.Distance to an object is calculated based on the time taken for the echo signal to return.

1. **DHT11 Sensor (Temperature and Humidity)**

The DHT11 sensor communicates using a one-wire digital interface via a single GPIO pin.It sends temperature and humidity data to the Raspberry Pi through the data pin.

1. **Light-Dependent Resistor (LDR):**

The LDR sensor is connected to a digital input pin of the Raspberry Pi GPIOinterface.The sensor provides a digital output that varies with light intensity.The digital output is directly read by the Raspberry Pi, and its value corresponds to the light intensity detected by the sensor

### How sensor data is acquired and processed by the Raspberry Pi

**Data Acquisition**:

* Ultrasonic Sensor: The Raspberry Pi triggers the sensor and measures the time taken for the echo to return. It calculates the distance based on the speed of sound in air.
* DHT11 Sensor: The Raspberry Pi reads digital data from the sensor using a specific protocol, extracting humidity and temperature values.
* LDR Sensor: The Raspberry Pi reads the digital output from the sensor to obtain the light intensity value.

**Data Processing:**

* Ultrasonic Sensor: No additional processing is required for the distance data.
* DHT11 Sensor: The raw data is processed to obtain humidity and temperature values.
* LDR Sensor: No significant data processing is applied to the light intensity readings.

**Data Transmission:**

* The Raspberry Pi communicates with the NetSim simulator using socket programming and sends the sensor data as packets over the network.

## Server Implementation

### Explanation of server.c code

The server.c code serves as the central component for collecting data from the sensors connected to the Raspberry Pi and transmitting this data to the client, which is the NetSim simulator. The code uses the WiringPi library for GPIO interfacing.

**Note***: WiringPi is a C/C++ library that provides an easy-to-use interface to the GPIO pins on the Raspberry Pi. It was originally developed by Gordon Henderson, and it is now maintained by a team of developers.*

Inside the main loop, the server reads data from each sensor and forms a response containing temperature, humidity, ultrasonic distance, and light intensity values. Before sending the data, it performs a validity check to ensure the sensor readings are within the expected range.

In summary, server.c serves as a bridge between the physical sensors on the Raspberry Pi and the NetSim simulator, ensuring the real-world sensor data is integrated into the simulated IoT network environment.

**Note***: The entire server.c code will be provided in the appendix at the end of the document*.

### How the server collects and handles data from the sensors

The server collects data from the connected sensors on the Raspberry Pi and sends it to the NetSim simulator through a socket connection. It ensures that the data is valid and includes temperature, humidity, ultrasonic distance, and light intensity values

### Communication between Raspberry Pi and the NetSim Simulator

A raspberry logo with a black background

Description automatically generated

Figure 1‑3: Communication between Raspberry Pi and NetSim

The communication between the Raspberry Pi and the NetSim simulator is established through a socket connection. The server.c code running on the Raspberry Pi acts as the server, while the NetSim simulator's client.c code acts as the client.

* **Server**: The server.c code initializes a socket and binds it to a specific IP address and port on the Raspberry Pi. It listens for incoming connections from the NetSim client.
* **Client**: The NetSim client.c code initializes a socket and connects it to the server's IP address and port.
* **Data Transmission**: Once the connection is established, the server collects sensor data from the Raspberry Pi and sends it to the NetSim client. The client receives the data and processes it for simulation within the NetSim environment.
* **Real-time Integration**: The real-world sensor data from the Raspberry Pi is now integrated seamlessly into the virtual IoT network simulation in NetSim, enabling real-time interactions between the virtual sensor nodes and the server

This communication mechanism ensures that the simulated IoT network in NetSim receives and processes real-world sensor data from the Raspberry Pi, creating a more realistic and practical simulation environment.

## NetSim Simulator Configuration

### Overview of NetSim Simulator and its features

NetSim is a network simulator tool enables users to virtually create a network comprising of devices, links, applications etc.that can be used to model and simulate different types of networks, including wireless networks, wired networks, and sensor networks. and study the behavior and performance of the Network.

NetSim comes with a wide range of features:

* **Comprehensive Technology Coverage**: NetSim supports a wide range of network technologies, including internetworks, legacy, and cellular networks, mobile adhoc networks, software-defined networks, wireless sensor networks, internet of things, cognitive radio networks, LTE networks, 5G NR (Pro version), VANET (Pro version), and more.
* **Performance Reporting:** The simulator provides detailed performance metrics for networks and sub-networks, enabling users to analyze and evaluate network behavior.
* **Powerful Visualization Tools:** NetSim includes a packet animator, allowing users to visualize the flow of packets in the simulated network, making it easier to understand the data transmission process.
* **External Interfacing:** NetSim offers external interfacing with SUMO, MATLAB, and Wireshark, providing flexibility in data analysis and integration with other tools.
* **Integrated Debugging (Standard and Pro versions):** Users can write and debug their custom code, linking it to NetSim using Visual Studio, facilitating advanced research and experimentation.

### Client.c code in the NetSim simulator

In the NetSim simulator **Client.c** file is added to the Application project

The "client.c" code serves as a client application responsible for connecting to a server and receiving data from it. It uses the Winsock library for socket communication on Windows systems.

* **Initialization:** The code initializes Winsock, creates a client socket, and sets up the server's address
* **Data Reception:**The client continuously reads data from the server using the "recv" function and stores it in the "buffer" variable.
* **Connection Closing:** After data reception is complete, the client closes the socket and cleans up Winsock resources.

**Note***: The entire client.c code will be provided in the appendix at the end of the document*.

**Environment variable Configuration**

Raspberry Pi Server IP will be added in Environment Variables of windows system were we are Running NetSim Simulator

1. Go to start search Run → Enter the command “SystemPropertiesAdvanced” and then click on OK.

2. Click the Environment Variables → Add the following Variables.

Variable name - RPi-IP  
Variable value - 192.168.0.84 **( IP-Address of your Rapberry Pi system)**

A screenshot of a computer

Description automatically generated

Figure 1‑4: Environmental Variable settings

### Creating and Running Sensor Network Scenario in NetSim

1. The NetSim-Raspbery-Pi-Interfacing-v13.3 comes with a sample network configuration that are already saved. To open this example, Go to Your work in the home screen of NetSim and click on the Experiment.
2. The saved network scenario consists

* 4 Sensor Nodes(Four different sensors)
* 1 LOWPAN\_Gateway
* 1 Router
* 1 Wired-Node(Server)

A graph with lines and dots

Description automatically generated

Figure 1‑5: Virtual network scenario created in NetSim with real sensor data from Raspberry Pi.

**Note***: This network scenario is created in Internet of things Network in NetSim*

1. Application Properties Configuration

|  |  |
| --- | --- |
| Application Properties | |
| Application Type | SENSOR |
| Source ID | 1,2,3,4(Four different Sensor ID,s) |
| Destination ID | 5 (Server ID) |

Table 1-Application Properties

1. Before running the NetSim scenario please assure that server code is running in Raspberry-Pi system

Example to run Server code on raspbery pi system:

* Save Server.c code on Documents
* Open the terminal from the Documents

Give the follwing commands:-

**gcc<filename.c>-o<filename> -lwiringPi -lm**

**sudo ./filename**

**Note***: Here filename.c is the name of your server.c code*

**A screenshot of a computer

Description automatically generated**

Figure 1‑6: Running Server.c code on Raspberry pi system

Once you started the Server.c code it starts listening client to connect

1. After starting the server, you can run the NetSim example. As the scenario begins running, you can go back to the Raspberry Pi system, where you will notice that the server status changes to 'Connected

A screenshot of a computer

Description automatically generated

Figure 1‑7: After Client is connected to server

### Results and Discussions

After the simulation is completed, we can check the results using Wireshark captured files. In the Result Dashboard, On the left side, Packet Capture →Simulation→We can see all captured packects of all different sensors

A screenshot of a computer

Description automatically generated

Figure 1‑8: Packet capture outputs in Result Dashboard

We can access the pcap files for each individual sensor. These pcap files contain the real payload data transmitted during the simulation, allowing us to examine and interpret the sensor data in detail.

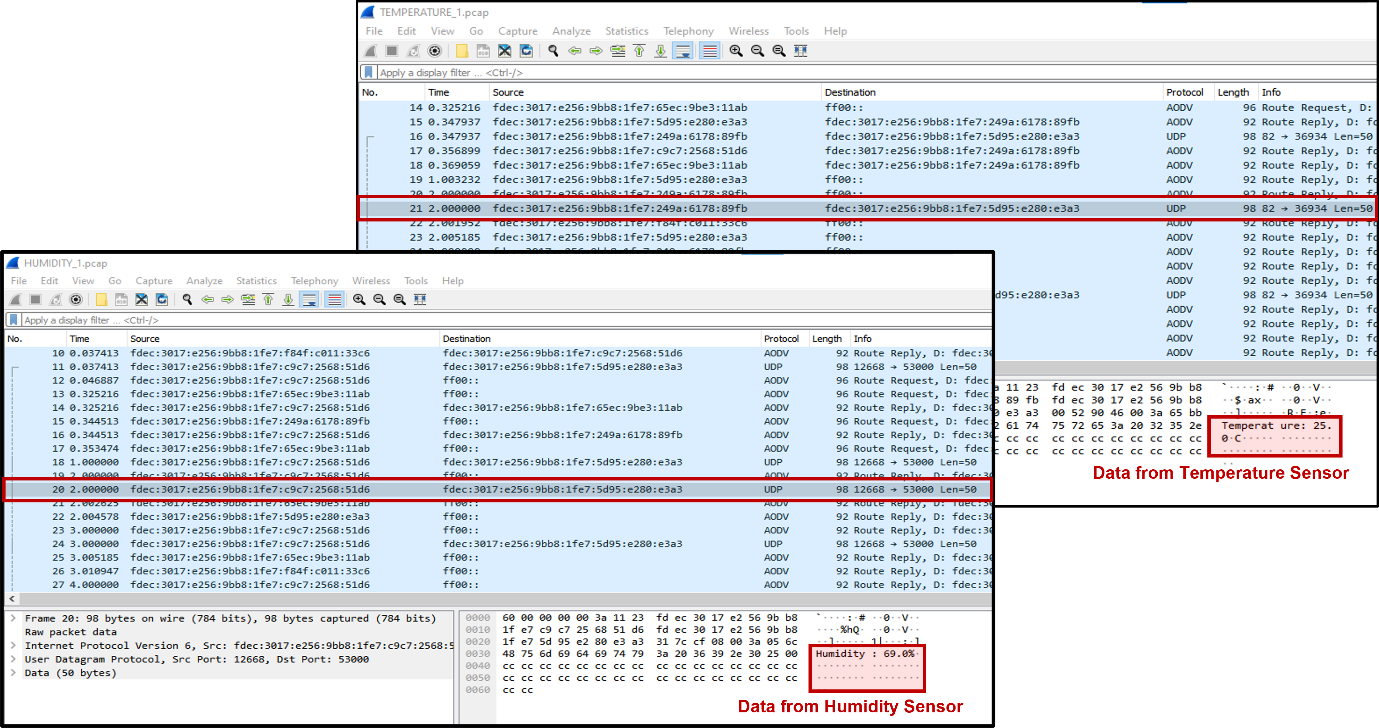


Figure 1‑9: Temperature and Humidity sensor data

A screenshot of a computer

Description automatically generated

Figure 1‑10: Ultrasonic and LDR sensor data

In the provided above Wireshark capture file screenshots, the payload data contains information about temperature,humidity,Ultrasonic and LDR sensor readings. These readings are the actual values acquired from the physical sensors connected to the Raspberry Pi. By examining these captured packets, we can verify the successful transmission of sensor data from the Raspberry Pi server to the NetSim simulator

## Appendix: NetSim source code modifications

MS Visual Studio Development environment is required for editing and building NetSim source codes. Please see this link on setting up Visual Studio

<https://support.tetcos.com/support/solutions/articles/14000138721-what-components-ofvisual-studio-community-2022-to-install-and-configure-to-work-with-netsim-source-c>

To open our project source code section, in NetSim home screen to →your work → source code → open code.

NetSim comes with inbuilt low-level functions to capture packets. This code is not open for user modification. The code to access the payload/header and to modify the payload/header is open to users and can be modified. We show below the source code changes we have made in red.

**Server.c code at the Raspberry pi system**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#include <wiringPi.h>

#define PORT 12345

#define LDR\_PIN 7 // Physical Pin 7

#define TRIG\_PIN 0 // Physical Pin 11 (GPIO 17)

#define ECHO\_PIN 2 // Physical Pin 13 (GPIO 27)

#define DHT\_PIN 3 // Physical Pin 15 (GPIO 22)

#define DHT\_TYPE DHT11 // DHT11 Sensor

int readLDRValue()

{

pinMode(LDR\_PIN, INPUT);

int pulse\_count = 0;

int start\_time = millis();

while ((millis() - start\_time) < 1000) // Count pulses for 1 second

{

if (digitalRead(LDR\_PIN) == LOW)

{

pulse\_count++;

}

}

return pulse\_count; // Use pulse count as an estimate of light intensity

}

float readUltrasonicDistance()

{

pinMode(TRIG\_PIN, OUTPUT);

pinMode(ECHO\_PIN, INPUT);

digitalWrite(TRIG\_PIN, LOW);

delay(30);

digitalWrite(TRIG\_PIN, HIGH);

delayMicroseconds(20);

digitalWrite(TRIG\_PIN, LOW);

while (digitalRead(ECHO\_PIN) == LOW)

{

continue;

}

long startTime = micros();

while (digitalRead(ECHO\_PIN) == HIGH)

{

continue;

}

long travelTime = micros() - startTime;

float distance = travelTime / 58.0; // Speed of sound in air is approximately 340 m/s or 29 microseconds per centimeter

return distance;

}

void readDHT11Data(float \*humidity, float \*temperature)

{

int data[5] = {0, 0, 0, 0, 0};

pinMode(DHT\_PIN, OUTPUT);

digitalWrite(DHT\_PIN, LOW);

delay(18);

digitalWrite(DHT\_PIN, HIGH);

delayMicroseconds(40);

pinMode(DHT\_PIN, INPUT);

// Wait for sensor response

while (digitalRead(DHT\_PIN) == HIGH)

delayMicroseconds(1);

// Sensor should pull low for 80us

while (digitalRead(DHT\_PIN) == LOW)

delayMicroseconds(1);

// Sensor should pull high for 80us

while (digitalRead(DHT\_PIN) == HIGH)

delayMicroseconds(1);

// Read 40 bits of data

for (int i = 0; i < 40; i++)

{

// Data bit starts with a low state for 50us

while (digitalRead(DHT\_PIN) == LOW)

delayMicroseconds(1);

// Measure the length of the high state to determine if it's a 0 or 1

int counter = 0;

while (digitalRead(DHT\_PIN) == HIGH)

{

delayMicroseconds(1);

counter++;

}

// Store the bit by shifting data array

data[i / 8] <<= 1;

if (counter > 30)

data[i / 8] |= 1;

}

// Verify checksum

if (data[4] == ((data[0] + data[1] + data[2] + data[3]) & 0xFF))

{

\*humidity = (float)data[0];

\*temperature = (float)data[2];

}

else

{

\*humidity = -1.0; // Error value for humidity

\*temperature = -1.0; // Error value for temperature

}

}

int main()

{

if (wiringPiSetup() == -1)

{

printf("Failed to initialize WiringPi library.\n");

return 1;

}

int server\_fd, new\_socket, valread;

struct sockaddr\_in address;

int addrlen = sizeof(address);

char buffer[1024] = {0};

char response[1024] = {0};

// Create socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0)

{

perror("socket failed");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

// Bind the socket to given IP and port

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0)

{

perror("bind failed");

exit(EXIT\_FAILURE);

}

// Listen for incoming connections

if (listen(server\_fd, 3) < 0)

{

perror("listen failed");

exit(EXIT\_FAILURE);

}

printf("Server listening on port %d\n", PORT);

// Accept incoming connection

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t \*)&addrlen)) < 0)

{

perror("accept failed");

exit(EXIT\_FAILURE);

}

printf("Connected\n");

while (1)

{

// Read LDR sensor data and calculate light intensity

int ldrValue = readLDRValue();

// Read Ultrasonic sensor data

float ultrasonicDistance = readUltrasonicDistance();

// Read DHT11 sensor data

float humidity, temperature;

readDHT11Data(&humidity, &temperature);

// Send the sensor data to the client

if (humidity != -1.0 && temperature != -1.0 && ldrValue!= -1.0 && ultrasonicDistance>0)

{

sprintf(response, "\nTemperature: %.1f°C, Humidity: %.1f%% Ultrasonic Distance: %.2f cm light Intensity: %d ", temperature, humidity,ultrasonicDistance,ldrValue);

}

else

{

sprintf(response, "Failed to retrieve data from sensor.\n");

}

send(new\_socket, response, strlen(response), 0);

delay(1000);

}

close(new\_socket);

close(server\_fd);

return 0;

}

**Client.c code is file added to the Application Project in NetSim**

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <winsock2.h>

#include "Application.h"

#pragma comment(lib, "ws2\_32.lib")

#define PORT 12345

char\* SERVER\_IP;

WSADATA wsaData;

SOCKET sock;

struct sockaddr\_in server;

char buffer[1024] = { 0 };

void Client\_init()

{

SERVER\_IP= getenv("RPi\_IP");

// Initialize Winsock

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

{

fprintf(stderr, "Failed to initialize Winsock\n");

return 1;

}

// Create socket

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) == INVALID\_SOCKET)

{

fprintf(stderr, "Failed to create socket\n");

WSACleanup();

return 1;

}

server.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);

server.sin\_family = AF\_INET;

server.sin\_port = htons(PORT);

// Connect to server

if (connect(sock, (struct sockaddr\*)&server, sizeof(server)) == SOCKET\_ERROR)

{

fprintf(stderr, "Connection failed\n");

closesocket(sock);

WSACleanup();

return 1;

}

}

char\* client\_run()

{

// Receive and print data from server

while (1)

{

memset(buffer, 0, sizeof(buffer));

int bytesRead = recv(sock, buffer, sizeof(buffer) - 1, 0);

if (bytesRead <= 0)

break;

//fprintf(stderr, "%s", buffer);

return buffer;

}

}

void client\_end()

{

closesocket(sock);

WSACleanup();

}

Changes in **copy\_payload() in Application.c in Application Project in NetSim**

void copy\_payload(UINT8 real[],NetSim\_PACKET\* packet,unsigned int\* payload, ptrAPPLICATION\_INFO info)

{

u\_short i;

uint32\_t key = 16;

char\* buffer= client\_run();

char temp\_val[BUFSIZ];

char humidi\_val[BUFSIZ];

char distan\_val[BUFSIZ];

char light\_val[BUFSIZ];

if (payload)

{

if (buffer)

{

char\* DNAME = DEVICE\_NAME(pstruEventDetails->nDeviceId);

//STart

char\* tempStart = strstr(buffer, "Temperature: ");

char\* humidityStart = strstr(buffer, "Humidity: ");

char\* distanceStart = strstr(buffer, "Ultrasonic Distance: ");

char\* lightStart = strstr(buffer, "light Intensity: ");

if (strcmp(DNAME, "TEMPERATURE") == 0) {

if (tempStart != NULL) {

tempStart += strlen("Temperature: ");

double temperature;

sscanf(tempStart, "%lf%%", &temperature);

//printf("Temperature: %.1lf°C", temperature);

sprintf(temp\_val, "Temperature: %.1lf°C", temperature);

for (i = 0; i < sizeof(temp\_val); i++)

{

if (info->encryption == Encryption\_XOR)

real[i] = xor\_encrypt(temp\_val[i], 16);

else

real[i] = temp\_val[i];

}

}

}

else if (strcmp(DNAME, "HUMIDITY") == 0) {

if (humidityStart != NULL) {

humidityStart += strlen("Humidity: ");

double humidity;

sscanf(humidityStart, "%lf%%", &humidity);

//printf();

sprintf(humidi\_val, "Humidity: %.1lf%%", humidity);

for (i = 0; i < sizeof(humidi\_val); i++)

{

if (info->encryption == Encryption\_XOR)

real[i] = xor\_encrypt(humidi\_val[i], 16);

else

real[i] = humidi\_val[i];

}

}

}

else if (strcmp(DNAME, "ULTRASONIC") == 0) {

if (distanceStart != NULL) {

distanceStart += strlen("Ultrasonic Distance: ");

double ultrasonicDistance;

sscanf(distanceStart, "%lf cm", &ultrasonicDistance);

//printf();

sprintf(distan\_val, "Distance: %.1lf cm", ultrasonicDistance);

for (i = 0; i < sizeof(distan\_val); i++)

{

if (info->encryption == Encryption\_XOR)

real[i] = xor\_encrypt(distan\_val[i], 16);

else

real[i] = distan\_val[i];

}

}

}

else if (strcmp(DNAME, "LIGHT") == 0) {

if (lightStart != NULL) {

lightStart += strlen("light Intensity: ");

double lightIntensity;

sscanf(lightStart, "%lf", &lightIntensity);

//printf();

sprintf(light\_val, "Intensity: %.1lf", lightIntensity);

for (i = 0; i < sizeof(light\_val); i++)

{

if (info->encryption == Encryption\_XOR)

real[i] = xor\_encrypt(light\_val[i], 16);

else

real[i] = light\_val[i];

}

}

}

}

if (info->encryption == Encryption\_TEA)

encryptBlock(real, payload, &key);

else if (info->encryption == Encryption\_AES)

aes256(real,payload);

else if(info->encryption==Encryption\_DES)

des(real,payload);

}

}

