**SMART PARKING**

Phase 3: **Development Part 1**

In this part you will begin building your project.

Start building the IoT sensor system and Raspberry Pi integration.

Configure IoT sensors (e.g., ultrasonic sensors) to detect parking space occupancy.

Write Python scripts on Raspberry Pi to collect data from sensors and send it to the cloud or mobile app server.

Creating a smart parking system with a Raspberry Pi and an ultrasonic sensor involves monitoring parking spaces and detecting whether they are occupied or vacant. Here's a Python code example using a Raspberry Pi and an ultrasonic sensor (HC-SR04) to build a simple smart parking system. In this example, we'll assume you have one parking space to monitor:

This code configures the Raspberry Pi to use an ultrasonic sensor to detect objects in a parking space. When an object is detected within a certain distance, it indicates that the parking space is occupied using an LED. You can adjust the threshold distance in the code according to your specific setup.Make sure you have the required components (Raspberry Pi, HC-SR04 ultrasonic sensor, and an LED) connected correctly. Also, ensure that you have the required libraries installed, such as RPi.GPIO.

1. **Select Sensors**: Choose the sensors based on the data you want to collect (e.g., temperature, humidity, motion). Ensure they are compatible with Raspberry Pi.

2. **Raspberry Pi Setup**: Get a Raspberry Pi (any model should work).Install an operating system (Raspberry Pi OS is a good choice). Update and upgrade the system: sudo apt-get update sudo apt-get upgrade

3. **Connect Sensors**: Wire the sensors to the Raspberry Pi's GPIO pins. Refer to the datasheets and documentation for correct wiring.

4. **Python Programming**: Write Python code to read data from the sensors. Libraries like GPIO Zero and Adafruit Circuit Python can be helpful.

5. **Data Storage**: Decide where to store the sensor data. Options include local storage, cloud services, or databases.

6. **Data Transmission**: If you're sending data to the cloud, use IoT protocols like MQTT or HTTP to transmit the data.

7. **Data Visualization**: Create a dashboard or use a service to visualize the data. Tools like Grafana or ThingSpeak can help.

8. **Automation**: Implement rules and automation based on sensor data (e.g., sending alerts when a threshold is reached).

9**. Security**: Secure your Raspberry Pi by changing default passwords and using encryption for data transmission.

10. **Power Supply**: Ensure your Raspberry Pi and sensors have a reliable power source.

11. **Testing and Calibration**: Test the system, calibrate sensors, and ensure data accuracy.

12**. Monitoring**: Implement remote monitoring and error handling for the system.

13. **Scalability:** Consider how to scale the system if you want to add more sensors or devices.

14. **Maintenance:** Regularly update software and perform maintenance as needed.

**ROLE OF IOT IN SMART PARKING:**

**1. Real-Time Parking Space Monitoring:**

Sensors and Devices: IoT-enabled sensors (e.g., ultrasonic, infrared, magnetic) are deployed in parking spaces to monitor occupancy in real time.

Immediate Data Collection: Sensors detect and transmit data about vehicle presence or absence, enabling instant updates on parking space availability.

**2. Parking Space Availability Prediction:**

**Data Analysis**: Utilizing historical data and real-time sensor data, IoT systems can analyze patterns to predict future parking space availability.

**Enhanced User Experience:** Drivers can access predictions to plan their parking in advance, reducing time spent searching for a spot.

**3. Dynamic Pricing and Demand Management:**

**Data-Driven Pricing Models:** IoT-enabled systems can adjust parking prices based on demand, time of day, special events, or other factors to optimize utilization and revenue.

**Encouraging Efficient Use:** Dynamic pricing encourages drivers to park in less busy areas, distributing demand and easing congestion.

**4. Efficient Traffic Flow:**

**Integration with Traffic Systems:** IoT systems can integrate with traffic management systems to optimize traffic flow, guiding drivers to available parking spaces in real time.

**Reduced Congestion:** Efficient traffic flow leads to reduced traffic congestion and pollution, benefiting both drivers and the environment.

**5. Automated Payments and Ticketing:**

**Seamless Transactions:** IoT can enable automated payments for parking, reducing manual processes and enhancing user convenience.

**Digital Ticketing:** Integration with mobile apps allows users to pay for parking digitally, eliminating the need for physical tickets.

**6. Remote Monitoring and Maintenance:**

**Condition Monitoring:** IoT can monitor the health and performance of parking equipment, enabling predictive maintenance and minimizing downtime**.**

**Cost Efficiency:** Remote monitoring helps in optimizing maintenance schedules and reducing operational costs.

**7. Data Analytics and Insights:**

**Data Utilization:** IoT generates vast amounts of data related to parking patterns, user behavior, and system performance.

**Data Analytics:** Advanced analytics provide valuable insights for city planning, infrastructure improvements, and policy decision-making.

**8. Integration with Navigation Apps:**

**Seamless Navigation:** Integration with popular navigation apps provides real-time parking information to drivers, guiding them to available parking spots.

**Enhanced User Experience:** Drivers can plan their routes considering parking availability, reducing stress and improving the overall experience**.**

**9. Environmental Impact Reduction:**

**Reduced Emissions:** By reducing the time spent searching for parking spots, IoT-enabled smart parking contributes to decreased fuel consumption and lower emission.

**ROLE OF RASPBERRY PI IN SMARTPARKING**

1. **Central Controller**:

Data Processing Hub: Raspberry Pi acts as the central controller that processes data received from IoT sensors deployed in parking spaces.

Decision Making: It analyzes sensor data to determine parking space occupancy and availability.

2. **Sensor Data Aggregation**:

Data Collection: Raspberry Pi interfaces with IoT sensors (e.g., ultrasonic, infrared) to collect real-time data on parking space occupancy.

Data Fusion: It aggregates data from multiple sensors to get a comprehensive view of parking availability.

3**. Connectivity**:

Network Interface: Raspberry Pi connects to the internet via Wi-Fi, Ethernet, or other means, allowing for communication with centralized servers, cloud platforms, or other devices.

Data Transmission: It facilitates the transmission of parking occupancy data to servers for further processing and analysis.

4. **Communication Protocol Implementation**:

MQTT, HTTP, or other Protocols: Raspberry Pi implements communication protocols like MQTT or HTTP to transmit data to cloud servers, enabling seamless communication with other components of the system.

5. **Data Processing and Analysis**:

Algorithm Execution: Raspberry Pi executes algorithms to analyze the collected parking occupancy data, detecting available parking spaces and patterns.

Data Interpretation: It processes data to generate insights, such as predicting future parking availability based on historical data and current trends.

6. **Integration with Cloud/Server**:

Data Transmission: Raspberry Pi securely transmits processed data to a cloud server or local server for storage, further analysis, and long-term data retention.

Integration Interfaces: It integrates with cloud APIs to ensure seamless data transfer and synchronization.

7. **Integration with User Interfaces**:

API Integration: Raspberry Pi interfaces with the cloud or server to retrieve processed parking data, making it available for integration with user interfaces.

Real-Time Updates: It ensures that the user interface (web or mobile app) receives real-time updates on parking space availability.

8. **User Experience Enhancement**:

Real-Time Information: Raspberry Pi enables the display of real-time parking space availability on user interfaces, improving the user experience by helping drivers find available parking spots easily.

Integration with Navigation Apps: Raspberry Pi can integrate with navigation applications to provide real-time parking information to drivers.

9. **Scalability and Flexibility**:

Modular Integration: Raspberry Pi allows for the integration of additional sensors, modules, or functionalities, making the system scalable and adaptable to changing requirements.

10. **Energy Efficiency and Cost-Effectiveness**:

Low Power Consumption: Raspberry Pi devices are energy-efficient, contributing to cost-effective operation in the smart parking system.

CODING :

#!/usr/bin/python

import time

import RPi.GPIO as GPIO

import time

import os,sys

from urllib.parse import urlparse

import paho.mqtt.client as paho

GPIO.setmode(GPIO.BOARD)

GPIO.setwarnings(False)

define pin for lcd

'''

# Timing constants

E\_PULSE = 0.0005

E\_DELAY = 0.0005

delay = 1

# Define GPIO to LCD mapping

LCD\_RS = 7

LCD\_E = 11

LCD\_D4 = 12

LCD\_D5 = 13

LCD\_D6 = 15

LCD\_D7 = 16

slot1\_Sensor = 29

slot2\_Sensor = 31

GPIO.setup(LCD\_E, GPIO.OUT) # E

GPIO.setup(LCD\_RS, GPIO.OUT) # RS

GPIO.setup(LCD\_D4, GPIO.OUT) # DB4

GPIO.setup(LCD\_D5, GPIO.OUT) # DB5

GPIO.setup(LCD\_D6, GPIO.OUT) # DB6

GPIO.setup(LCD\_D7, GPIO.OUT) # DB7

GPIO.setup(slot1\_Sensor, GPIO.IN)

GPIO.setup(slot2\_Sensor, GPIO.IN)

# Define some device constants

LCD\_WIDTH = 16 # Maximum characters per line

LCD\_CHR = True

LCD\_CMD = False

LCD\_LINE\_1 = 0x80 # LCD RAM address for the 1st line

LCD\_LINE\_2 = 0xC0 # LCD RAM address for the 2nd line

LCD\_LINE\_3 = 0x90# LCD RAM address for the 3nd line

def on\_connect(self, mosq, obj, rc):

self.subscribe("Fan", 0)

def on\_publish(mosq, obj, mid):

print("mid: " + str(mid))

mqttc = paho.Client() # object declaration

# Assign event callbacks

mqttc.on\_connect = on\_connect

mqttc.on\_publish = on\_publish

url\_str = os.environ.get('CLOUDMQTT\_URL', 'tcp://broker.emqx.io:1883')

url = urlparse(url\_str)

mqttc.connect(url.hostname, url.port)

'''

Function Name :lcd\_init()

Function Description : this function is used to initialized lcd by sending the different commands

'''

def lcd\_init():

# Initialise display

lcd\_byte(0x33,LCD\_CMD) # 110011 Initialise

lcd\_byte(0x32,LCD\_CMD) # 110010 Initialise

lcd\_byte(0x06,LCD\_CMD) # 000110 Cursor move direction

lcd\_byte(0x0C,LCD\_CMD) # 001100 Display On,Cursor Off, Blink Off

lcd\_byte(0x28,LCD\_CMD) # 101000 Data length, number of lines, font size

lcd\_byte(0x01,LCD\_CMD) # 000001 Clear display

time.sleep(E\_DELAY)

'''

Function Name :lcd\_byte(bits ,mode)

Fuction Name :the main purpose of this function to convert the byte data into bit and send to lcd port

'''

def lcd\_byte(bits, mode):

# Send byte to data pins

# bits = data

# mode = True for character

# False for command

GPIO.output(LCD\_RS, mode) # RS

# High bits

GPIO.output(LCD\_D4, False)

GPIO.output(LCD\_D5, False)

GPIO.output(LCD\_D6, False)

GPIO.output(LCD\_D7, False)

if bits&0x10==0x10:

GPIO.output(LCD\_D4, True)

if bits&0x20==0x20:

GPIO.output(LCD\_D5, True)

if bits&0x40==0x40:

GPIO.output(LCD\_D6, True)

if bits&0x80==0x80:

GPIO.output(LCD\_D7, True)

# Toggle 'Enable' pin

lcd\_toggle\_enable()

# Low bits

GPIO.output(LCD\_D4, False)

GPIO.output(LCD\_D5, False)

GPIO.output(LCD\_D6, False)

GPIO.output(LCD\_D7, False)

if bits&0x01==0x01:

GPIO.output(LCD\_D4, True)

if bits&0x02==0x02:

GPIO.output(LCD\_D5, True)

if bits&0x04==0x04:

GPIO.output(LCD\_D6, True)

if bits&0x08==0x08:

GPIO.output(LCD\_D7, True)

# Toggle 'Enable' pin

lcd\_toggle\_enable()

'''

Function Name : lcd\_toggle\_enable()

Function Description:basically this is used to toggle Enable pin

'''

def lcd\_toggle\_enable():

# Toggle enable

time.sleep(E\_DELAY)

GPIO.output(LCD\_E, True)

time.sleep(E\_PULSE)

GPIO.output(LCD\_E, False)

time.sleep(E\_DELAY)

'''

Function Name :lcd\_string(message,line)

Function Description :print the data on lcd

'''

def lcd\_string(message,line):

# Send string to display

message = message.ljust(LCD\_WIDTH," ")

lcd\_byte(line, LCD\_CMD)

for i in range(LCD\_WIDTH):

lcd\_byte(ord(message[i]),LCD\_CHR)

lcd\_init()

lcd\_string("welcome ",LCD\_LINE\_1)

time.sleep(0.5)

lcd\_string("Car Parking ",LCD\_LINE\_1)

lcd\_string("System ",LCD\_LINE\_2)

time.sleep(0.5)

lcd\_byte(0x01,LCD\_CMD) # 000001 Clear display

# Define delay between readings

delay = 5

while 1:

# Print out results

rc = mqttc.loop()

slot1\_status = GPIO.input(slot1\_Sensor)

time.sleep(0.2)

slot2\_status = GPIO.input(slot2\_Sensor)

time.sleep(0.2)

if (slot1\_status == False):

lcd\_string("Slot1 Parked ",LCD\_LINE\_1)

mqttc.publish("slot1","1")

time.sleep(0.2)

else:

lcd\_string("Slot1 Free ",LCD\_LINE\_1)

mqttc.publish("slot1","0")

time.sleep(0.2)

if (slot2\_status == False):

lcd\_string("Slot2 Parked ",LCD\_LINE\_2)

mqttc.publish("slot2","1")

time.sleep(0.2)

else:

lcd\_string("Slot2 Free ",LCD\_LINE\_2)

mqttc.publish("slot2","0")

time.sleep(0.2)



