SMART PARKING SYSTEM USING RASPBERRY-PI (IOT)

(An IBM Project)

in partial fulfilment for the Course

(NAAN MUDHALVAN)

PROJECT REPORT

Submitted by

VINITHA V	**510921106045**
RAGAVI S	**510921106027**
NANDHINI B	**510921106017**
GAYATHRI M	**510921106301**

BACHELOR OF ENGINEERING IN ELECTRONICS AND COMMUNICATION ENGINEERING

GLOBAL INSTITUTE OF ENGINEERING & TECHNOLOGY

MELVISHARAM, RANIPET – 632509.

ANNA UNIVERSITY : CHENNAI 600025

NOV - 2023

CONTENT

INTRODUCTION

Problem Statement:

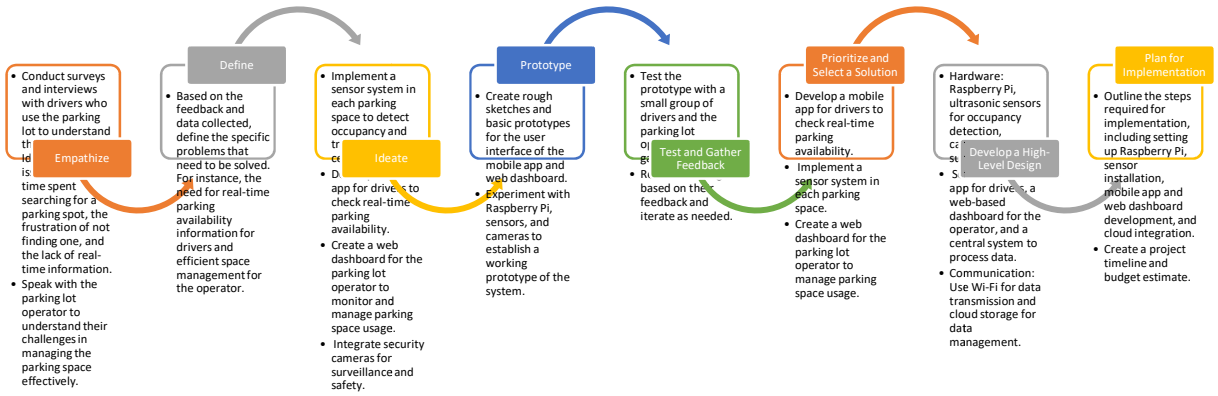
Problem:

In urban areas, finding available parking spaces can be a time-consuming and frustrating experience for drivers. It often leads to congestion, pollution, and wasted fuel. Additionally, parking lot operators may struggle to efficiently manage their parking spaces.

Scope:

Our project aims to create a Smart Parking System using Raspberry Pi to address this problem in a single parking IoT.

Design Thinking



INNOVATION

Ensure the cameras can connect to your network, whether through Wi-Fi, Ethernet, or cellular, depending on your

Power Efficiency: If using battery-powered cameras, check their power consumption and expected battery life to ensure they meet your

Line of Sight:

Ensure that there are no obstacles (e.g., trees, poles,walls)

For larger parking areas, consider using cameras with oblique (angled) views to cover more spaces efficiently.

Capture images at regular time intervals (e.g., every minute).

Choose an appropriate image file format (e.g., JPEG, PNG) based on your storage and processing requirements.

Define a data retention policy to manage the storage of captured images. Determine how long images will be retained and

Architecture:

Define ROIs within the images that correspond to parking spaces. These ROIs will be the areas where the

Ensure that your image processing software can interface with the cameras to receive images and trigger processing

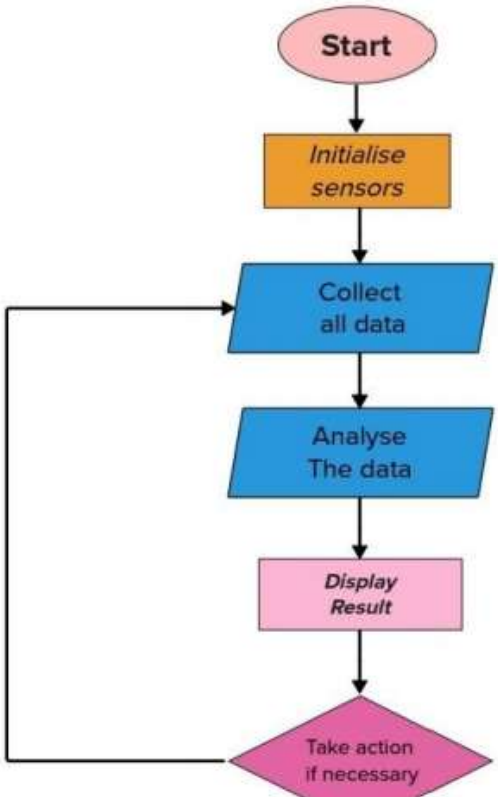
Ensure that your image processing system is scalable to accommodate additional cameras and parking spaces as your project

Design a database schema that can efficiently store parking space occupancy data. Consider tables for parking

Ensure compliance with data privacy regulations and policies, especially if your system collects personally identifiable

Implement analytics and reporting tools to gain insights from parking occupancy data, such as historical trends and

Flow chart:


Start
Initialise sensors
Collect all data
Analyse The data
Display Result
Take action if necessary

Real-Time Updates:

Display real-time information about parking

Provide notifications to alert users about the status of their reservations, approaching payment deadlines, and other

Implement security features to protect user data and ensure secure transactions if the UI involves payments or

Provide options for users to subscribe to specific parking spaces or areas to receive availability updates.

Use notifications to request feedback from users, such as post-parking experience surveys, and encourage them to rate their experience.

Opt-In and Opt-Out

Compliance and Privacy:

Ensure that your alert and notification system complies with

Determine how well your system can handle increased loads. Test its ability to accommodate more cameras, parking spaces, and

Verify that the system is accessible to users with disabilities. Ensure compliance with accessibility standards (e.g., WCAG) to

Maintain records of calibration settings, procedures, and results. Proper documentation ensures traceability and helps with

Use message queuing systems (e.g., RabbitMQ, Apache Kafka) to decouple components and handle asynchronous tasks,

Content Delivery Networks (CDNs):

Use CDNs to distribute content (e.g., images, videos) to users

Continuously gather user feedback and iterate on your system's scalability based on real-world usage patterns and changing

Adhere to data protection regulations (e.g., GDPR) and ensure that user data is handled with care, including anonymization and

Conduct regular penetration testing by ethical hackers to identify and remediate vulnerabilities before malicious actors can

Be mindful of user privacy and comply with relevant privacy regulations. Clearly communicate how user data will be used and

Implement real-time data streaming or messaging systems (e.g., Apache Kafka) to transmit data and events between system

Thoroughly test and validate integrations to ensure that data flows correctly, components work harmoniously, and the system

Establish a schedule for routine maintenance tasks, such as system backups, database optimization, and hardware inspections.

User Feedback and Bug Reporting:

Encourage users to provide feedback and report any issues they

Allocate sufficient budget and resources for maintenance and updates. Consider both planned maintenance tasks and unforeseen

CODING & SOLUTIONING

Components Required:

1.**Ultrasonic Sensor**:

The ultrasonic sensor is a device that uses ultrasonic sound waves to measure distances. It typically has two main components: a transmitter and a receiver. The transmitter emits ultrasonic pulses, and the receiver detects the reflection of these pulses to calculate the distance to an object.

2. **LEDs (Light Emitting Diodes)**:

LEDs are used as visual indicators in electronic projects. They emit light when an electrical current flows through them. In your project, you have three LEDs, often categorized by their colors, such as red, green, and blue.

3.** 330-ohm Resistor**:

A resistor is an electrical component that limits the flow of electric current. In your project, the 330-ohm resistor is used to limit the current flowing through the LEDs. It's important for preventing them from burning out.

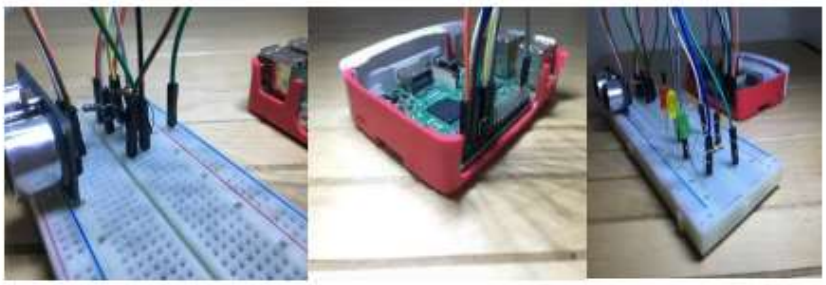
4. **10-ohm Resistor:** This resistor is likely used for a specific purpose in your project. The value of the resistor depends on the application and the specific requirements of the project. If you could provide more context, I can give more specific guidance.

5. **Jumper Wires**:

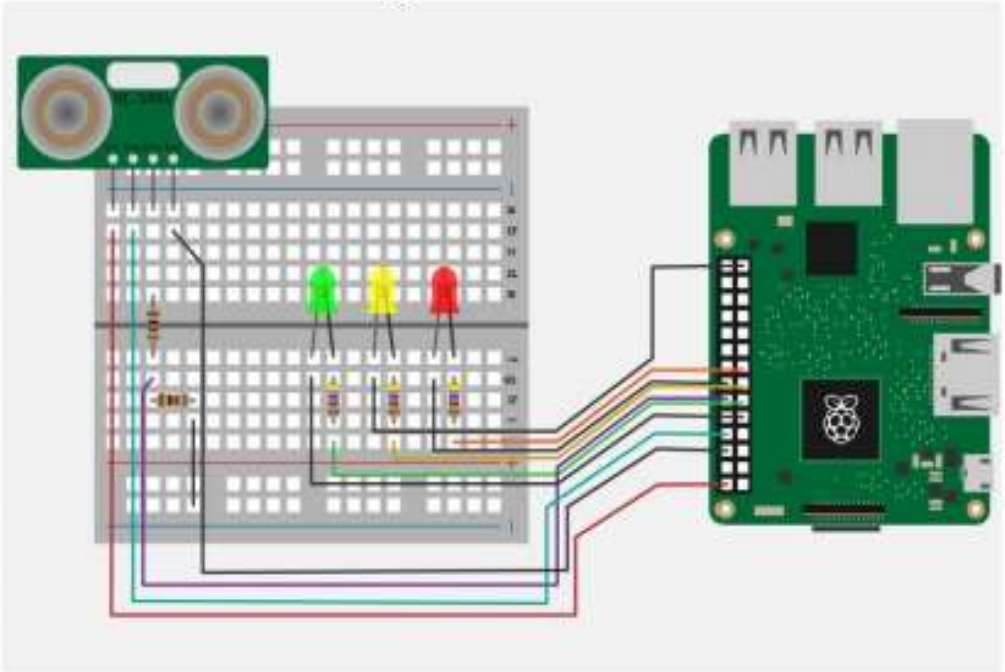
Jumper wires are used for making electrical connections between different components on a breadboard or connecting components to a Raspberry Pi's GPIO pins. They come in various lengths and can be male-to-male, male-to-female, or female-to-female.

6. **Half Breadboard**:

A breadboard is a tool for prototyping electronic circuits. The half breadboard provides a platform for you to connect and test your components without soldering. It has a grid of holes for inserting components and creating electrical connections.



Circuit Simulation Diagram:



Python Code used in Raspberry Pi

```
import RPi.GPIO as GPIO
import time
# Define GPIO pins
TRIG = 23  # Ultrasonic sensor trigger pin
ECHO = 24  # Ultrasonic sensor echo pin
LED1 = 17  # LED1 pin
LED2 = 27  # LED2 pin
LED3 = 22  # LED3 pin
# Set GPIO mode and setup
GPIO.setmode(GPIO.BCM)
GPIO.setup(TRIG, GPIO.OUT)
GPIO.setup(ECHO, GPIO.IN)
GPIO.setup(LED1, GPIO.OUT)
GPIO.setup(LED2, GPIO.OUT)
GPIO.setup(LED3, GPIO.OUT)
    GPIO.output(TRIG, False)


while GPIO.input(ECHO) == 0:
        pulse_start = time.time()
```

```
try:
    while True:
        distance = measure_distance()
        print("Distance: {:.2f} cm".format(distance))
        if distance < 10:
            GPIO.output(LED1, GPIO.HIGH)
            GPIO.output(LED2, GPIO.LOW)
            GPIO.output(LED3, GPIO.LOW)
        elif 10 <= distance < 20:
            GPIO.output(LED1, GPIO.HIGH)

            GPIO.output(LED2, GPIO.HIGH)
            GPIO.output(LED3, GPIO.LOW)
    else:
        GPIO.output(LED1, GPIO.HIGH)
        GPIO.output(LED2, GPIO.HIGH)
        GPIO.output(LED3, GPIO.HIGH)

except KeyboardInterrupt:
    GPIO.cleanuop()
```

Note: This code sets up the ultrasonic sensor and three LEDs. The LEDs will light up in different patterns depending on the measured distance from the ultrasonic sensor. You can modify the distance thresholds and LED patterns to suit your project's needs.

MOBILE APPLICATION DEPLOYMENT

INTRODUCTION:

Creating a full-fledged mobile app for smart parking is a complex task that requires extensive development and resources. Below, I'll provide a simplified outline of the steps.

STEPS TO BE FOLLOWED:

I'll provide a simplified outline of the steps you would need to follow:

1. Project Planning:

Define the scope and features of your smart parking app. Determine the user flow, core functionalities, and objectives.

2. Technology Stack:

Choose the technology stack. For a cross-platform app in Python, you can consider frameworks like Kivy or BeeWare. Alternatively, you can use native languages (Java/Kotlin for Android, Swift for iOS) for platform- specific apps.

3. User Interface (UI) Design:

Create wireframes and design the user interface. Decide on the layout, screens, and visual elements.

4. Backend development:

Set up the backend of your app, including:Database: Choose a database system to store information such as parking spot availability, user data, and transaction history.Server: Develop the server-side logic for handling user accounts, reservations, and payments.

5. User Authentication:

Implement user registration and authentication features to manage user accounts securely.

6. Real-time Data Integration:

Connect to real-time parking availability data sources such as sensors or APIs. Ensure that your app can fetch and display this data to users.

7. Payment Processing:

Implement secure payment processing for users who wish to reserve parking spots. Integrate payment gateways or services like Stripe or PayPal.

8. Geo-location Services:

Utilize geo-location services (e.g., GPS) to help users find available parking spots and provide navigation to their chosen parking location.

9. Frontend Development:

Develop the app's frontend using your chosen framework or native technologies. Implement the UI design, navigation, and user interactions.

10. Testing and Debugging:

Conduct thorough testing to identify and fix bugs and issues. Test on various devices and simulate real-world usage.

11. Deployment:

Prepare your app for deployment, including: Creating icons, splash screens, and other required assets. Generating platform-specific app packages (APK for Android, IPA for iOS).

12. Platform-specific Deployment:

For Android: Create a Google Play Developer account. Submit your app to the Google Play Store.

For iOS: Enroll in the Apple Developer Program. Submit your app to the Apple App Store

13. Maintenance and Updates:

Continuously update and maintain your app. Address user feedback, fix issues, and add new features.

14. Marketing and User Acquisition:

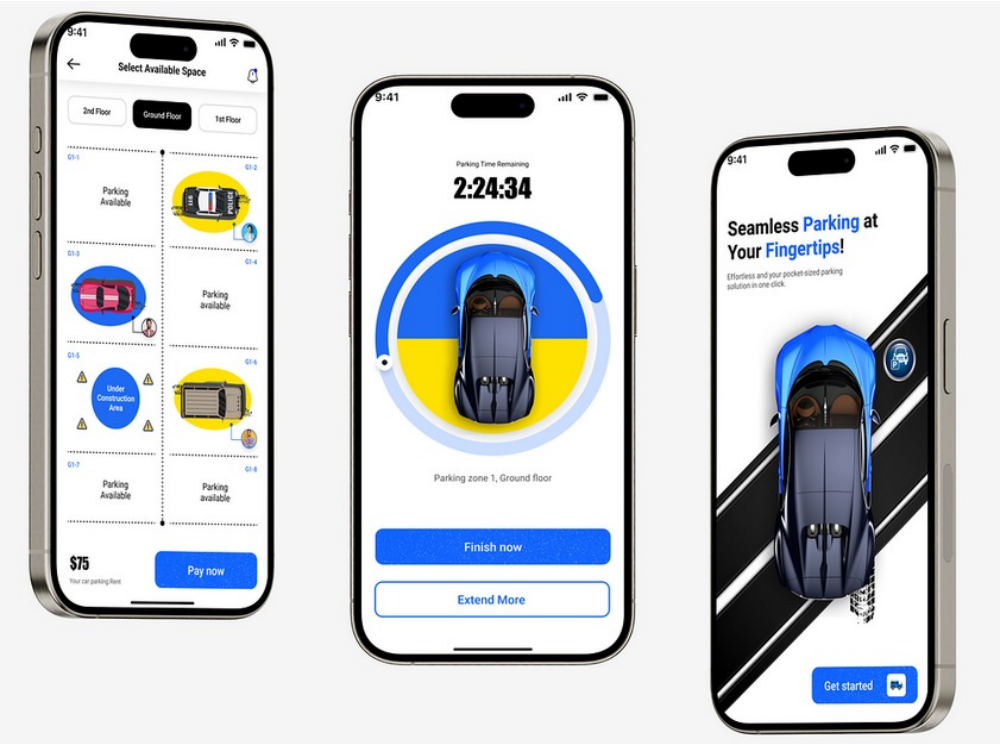
Promote your app through various marketing channels, including social media, advertising, and app store optimization (ASO).

15. Security and Legal Considerations:Ensure that your app is secure, and user data is protected with encryption and secure authentication. Address legal aspects like user privacy, terms of service, and compliance with data protection regulations. This is a simplified overview, and building a complete smart parking app requires a considerable amount of work and expertise. You may need to engage with designers, developers, and testers, and also consider issues like scalability and real-time data synchronization, depending on the scale of your project.

PROGRAM INTRODUCTION:

Creating a complete mobile app for smart parking in Python from scratch is a significant project and would require extensive code. However, I can provide a simplified example of a Python script using the Kivy framework to demonstrate a basic user interface for a smart parking app. Keep in mind that this is a minimal illustration and does not include features like real-time data, backend services,

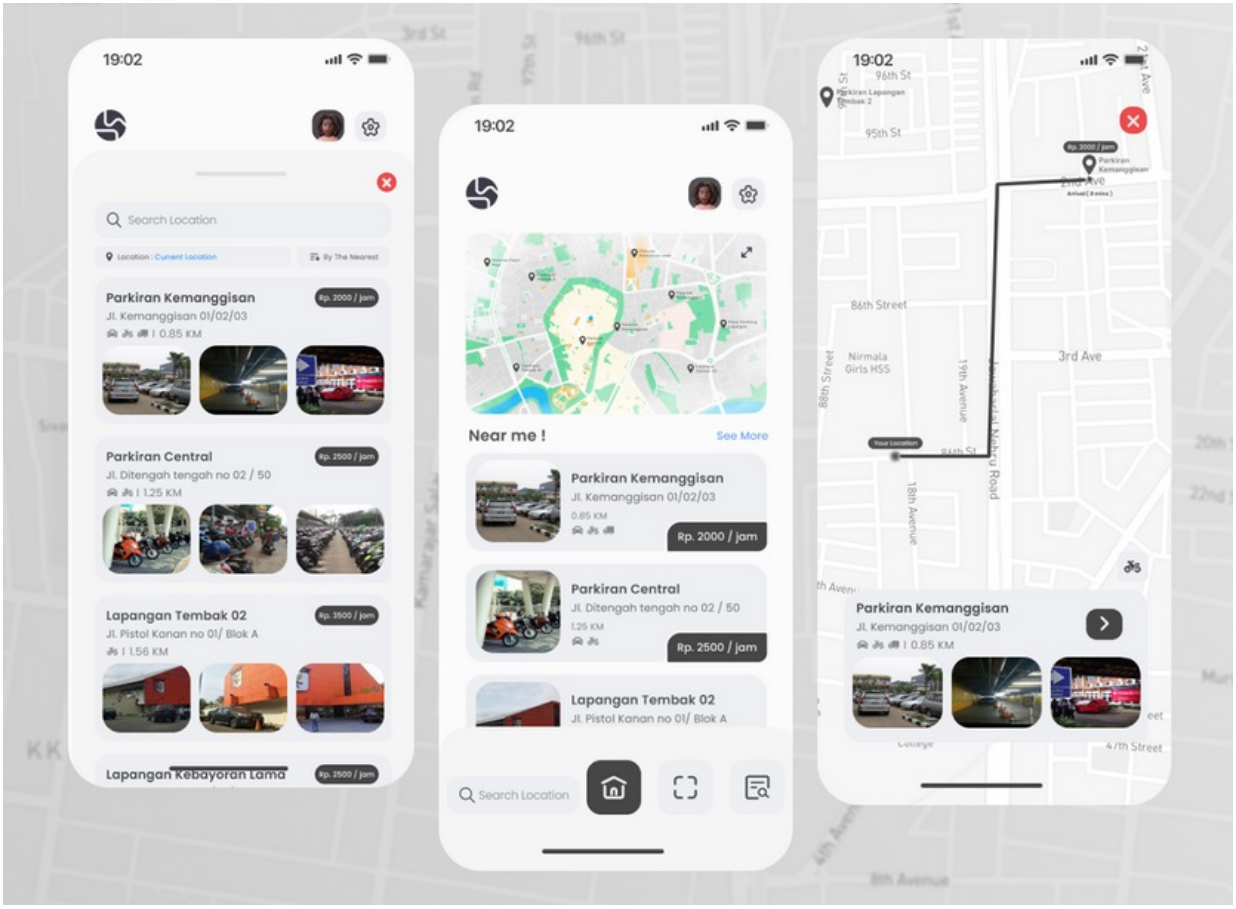
or actual parking reservation functionality. You'll need to expand upon this foundation for a full-fledged app.



CODING USING FLUTTER SOFTWARE:

```
# Import Kivy libraries
From kivy.app import App
From kivy.uix.boxlayout import BoxLayout
From kivy.uix.label import Label
From kivy.uix.button import Button

# Create a simple parking app
Class SmartParkingApp(App):
    Def build(self):
        # Main layout
        Layout = BoxLayout(orientation='vertical', padding=10, spacing=10)

        # Title label
        Title_label = Label(text='Smart Parking App', size_hint=(1, 0.1))

        # Parking information
        Parking_info_label = Label(text='Available parking spots: 10', size_hint=(1,0.1))
```

```
        # Reserve button
        Reserve_button = Button(text='Reserve Parking Spot',
        size_hint=(1, 0.1))
        Reserve_button.bind(on_press=self.reserve_parking)
        # Status label
        Self.status_label = Label(text='', size_hint=(1, 0.1))

        #Add widgets to the layout
        Layout.add_widget(title_label)
        Layout.add_widget(parking_info_label)
        Layout.add_widget(reserve_button)
        Layout.add_widget(self.status_label)
        Return layout

    Def reserve_parking(self, instance):
        # Placeholder function to simulate parking reservation
        Self.status_label.text = 'Parking spot reserved!'

# Run the app
If__name__== '__ main__ ':
    SmartParkingApp().run()
```



CONCLUSION

This code creates a simple mobile app using Kivy with a title, parking information, a button to reserve a parking spot (simulated), and a status label.Keep in mind that building a fully functional smart parking app would require integrating real-time data, payment processing, user authentication, and much more. This example is just a starting point to illustrate the basic structure of a mobile app.

APPENDIX

Github Project Link : https://github.com/Vinithavij/vinitha