CMR University

Project Report

Smart Parking System

Submitted in partial fulfillment of the requirements

For the award of the degree

Bachelor of Computer Application

(VI Semester)

DEVELOPED BY

Name: Bala krishna M **Reg No: 21DBCAG017**	**Name: Kamesh E P** **Reg No: 21DBCAG054**	**Name: Akshay S** **Reg No: 21DBCAG007**

GUIDED BY

Prof. Aurangjeb Khan
Assistant Professor,
SSCS, CMRU, Bangalore

2021 – 2024

School of Science and Computer Studies

#5, Bhuvanagiri, OMBR Layout, Bangalore- 560 043, Karnataka – India

CMR University

CMRU

Certificate

This is to certify that **Kamesh E P** and **21DBCAG054** Belonging to **VI Semester, BCA** course has satisfactorily completed the project titled **“Smart Parking System”** in partial fulfillment of Practical prescribed by the School of Science and Computer Studies for the Course code **8CAPS4010** and Course name **Capstone** during the academic year 2021 – 2024

Prof. Aurangjeb Khan	**Prof. Jayanthi.M**	**Dr. Ashok Kumar**
Project Guide	**Program Coordinator**	**Director**

SEAL:

Name: Kamesh E P
Reg No: 21DBCAG054
Date of Practical Examination: 03/06/2024

School of Science and Computer Studies

#5, Bhuvanagiri, OMBR Layout, Bangalore- 560 043, Karnataka - India

CMR University

Declaration

The project titled **Smart Parking System** Developed by us in the partial fulfillment of **VI Semester, BCA** course, is an authentic work carried out by us under the guidance of **Prof. Aurangjeb Khan** School of Science and Computer Studies, CMR University, Bangalore.

We declare that the project has not been submitted to any degree or diploma to the above said university or any other university.

Signature: ……………………………………

Name: Kamesh E P
Reg No: 21DBCAG054

I certify that all the above statements given by the candidate are true to the best of my knowledge and belief.

Signature: ……………………………………

Prof. Aurangjeb Khan
Project Guide

School of Science Studies

#5, Bhuvanagiri, OMBR Layout, Bangalore- 560 043, Karnataka – India

ACKNOWLEDGEMENT

We take this opportunity to express our gratitude to all those who have given their moral support during our entire project.

Firstly, we wish to express our profound thanks to **Dr. Sabita Ramamurthy**, Chancellor, CMR University, Bangalore, for providing us all the facilities required in completion of our project.

I am greatly indebted to **Dr. Ashok Kumar** Director, School of Science Studies, Bangalore, for her encouragement and suggestion at every step of our project work.

Our Sincere thanks to **Prof. Aurangjeb Khan** project guide, without whom this project is unimaginable, for guiding us with keen interest and constant encouragement at every stage during the course of our project work.

Finally, yet importantly, we want to thank all our friends and our family members who have helped us directly or indirectly in the successful completion of this project.

Name: Kamesh E P
Reg No: 21DBCAG054

CONTENTS

1. Introduction

SMART PARKING SYSTEM USING IOT

A Smart Parking System powered by IoT (Internet of Things) revolutionizes traditional parking management, offering a seamless, efficient, and user-friendly experience. Leveraging cutting-edge technology, it transforms the way we interact with parking spaces, optimizing utilization, reducing congestion, and enhancing convenience for both drivers and parking lot operators.

At its core, this system integrates various IoT devices such as sensors, cameras, and connectivity modules to monitor and manage parking spaces in real-time. These sensors detect the presence or absence of vehicles, transmitting data to a centralized platform via the internet. Through advanced algorithms and data analytics, the system provides valuable insights into parking space availability, occupancy patterns, and usage trends.

Key Components:

- **ESP32 microcontroller:** The ESP32 is the brain of the system. It's a powerful, low-cost, and Wi-Fi enabled microcontroller that can be programmed to read data from sensors, control outputs, and communicate with a network.

- **Sensors:** Sensors are used to detect the presence or absence of a vehicle in a parking space. Common sensor options include:
  - **Infrared (IR) sensors:** IR sensors emit infrared light and detect reflected light. They can be used to detect if a vehicle is blocking the sensor's beam.

- **Actuators:** Actuators are used to control physical elements of the parking system based on the data from the sensors. Examples of actuators include:

  - **Servo motors:** These motors can be used to control a barrier gate at the entrance/exit of the parking lot.
  - **LED lights:** LED lights can be used to indicate the availability of a parking space (e.g., green for vacant, red for occupied).

- **Communication Module (Optional):** An additional communication module like an Ethernet shield or LoRa module can be used to connect the ESP32 to the cloud or a local server for data logging, visualization, and integration with mobile applications.

- **Power Supply:** The entire system will need a power supply to operate. This can be a wall adapter, a battery pack, or even solar panels for a more eco-friendly solution.

2. Objectives

Smart Parking involves the use of low cost sensors, real-time data and applications that allow users to monitor available and unavailable parking spots. The goal is to automate and decrease time spent manually searching for the optimal parking floor, spot and even lot. Some solutions will encompass a complete suite of services such as online payments, parking time notifications and even car searching functionalities for very large lots. A parking solution can greatly benefit both the user and the lot owner.

- **Optimized parking**: Users find the best spot available, saving time, resources and effort. The parking lot fills up efficiently and space can be utilized properly by commercial and corporate entities.
- **Reduced traffic**: Traffic flow increases as fewer cars are required to drive around in search of an open parking space.
- **Reduced pollution**: Searching for parking burns around one million barrels of oil a day. An optimal parking solution will significantly decrease driving time, thus lowering the amount of daily vehicle emissions and ultimately reducing the global environmental footprint.
- **Increased Safety**: Parking lot employees and security guards contain real-time lot data that can help prevent parking violations and suspicious activity. License plate recognition cameras can gather pertinent footage. Also, decreased spot-searching traffic on the streets can reduce accidents caused by the distraction of searching for parking.
- **Decreased Management Costs**: More automation and less manual activity saves on labor cost and resource exhaustion.
- **Enhanced User Experience**: A smart parking solution will integrate the entire user experience into a unified action. Driver's payment, spot identification, location search and time notifications all seamlessly become part of the destination arrival process.

3. System analysis

System analysis for a Smart Parking System involves a thorough examination of various aspects to ensure the system meets its objectives effectively. Here's a structured approach to conducting system analysis:

- **Understanding the Context:** Define the scope and context of the smart parking system. Determine whether it will cover on-street parking, parking garages, or both.
- **Gathering Requirements:** Engage with stakeholders to gather requirements through interviews, surveys, and workshops.
- **Data Management:** Define data collection methods, including sensor data, vehicle information, and user interactions.

3.1 Identification of the Need

In today's urban landscape, finding a parking spot can feel like a battle. Here's why smart parking systems are becoming increasingly necessary:

- **Limited Parking, Growing Demand:** Cities are bursting at the seams, and the number of vehicles keeps rising. This creates a fundamental mismatch - there simply aren't enough parking spaces to go around.

- **Circling the Block Blues:** Drivers waste precious time and fuel endlessly searching for a vacant spot. This frustrating experience not only affects drivers' moods but also contributes significantly to traffic congestion and air pollution.

- **Parking Management Headaches:** Traditional parking methods like meter systems are inconvenient for drivers and labor-intensive for operators. They require constant maintenance, enforcement, and cash collection.

- **Real-time Parking Availability:** Imagine using an app or navigation system to see exactly where open spaces are located! This eliminates the time-wasting hunt and reduces frustration.

- **Optimizing Space Usage:** Sensors can detect occupied and vacant spots, guiding drivers directly to them. This not only benefits drivers but also allows for data-driven pricing strategies that encourage efficient use of parking facilities.

- **Streamlined Management:** Smart systems automate many parking management tasks, like enforcement and payment. This frees up parking operators' time and resources for other tasks, leading to a smoother operation.

3.2 Preliminary Investigation

This investigation aims to assess the need and feasibility of implementing a smart parking system in your target area. Here's a roadmap to guide you:

1. Problem Definition & Scope:

- **Challenges:** Identify the specific parking issues you want to address. Is it congestion, inefficient space use, or long search times?

- **Target:** Specify what type of parking the system will cover (on-street, off-street, public, private).
- **Project Size:** Estimate the area's size and budget for the project.

2. Data Collection:

a) Parking Usage:

- **Surveys:** Conduct parking occupancy surveys at different times to understand usage patterns.
- **Violations:** Analyze existing parking violation data to identify areas with high demand or misuse.

b) Infrastructure:

- **Map Parking:** Create a detailed map showing the existing parking layout and capacity.
- **Power & Network:** Assess the availability of power and communication networks for sensor installation.

3. Technology Exploration:

- **Sensor Technology:** Research different smart parking sensor options (magnetic, ultrasonic, video) considering suitability, cost, and environment.
- **Data Communication:** Evaluate reliable data communication protocols (LoRaWAN, cellular) for sensor data transmission.
- **Software & Apps:** Explore existing smart parking software platforms and mobile app solutions for managing parking data and user interaction.

4. Stakeholder Identification:

- **List Stakeholders:** Identify individuals and groups impacted by the system, such as:
  - Drivers
  - Parking Facility Owners/Operators
  - Local Authorities
  - Residents/Businesses in the Area

- **Gather Input:** Seek their feedback on the implementation and use of the smart parking system.

5. Preliminary Cost-Benefit Analysis:

- **Investment Estimation:** Estimate the initial cost for hardware, software, installation, and maintenance.
- **Projected Benefits:** Consider potential benefits like:
    - Reduced traffic congestion and fuel consumption
    - Increased parking revenue for operators
    - Improved efficiency in parking management
    - Enhanced driver experience

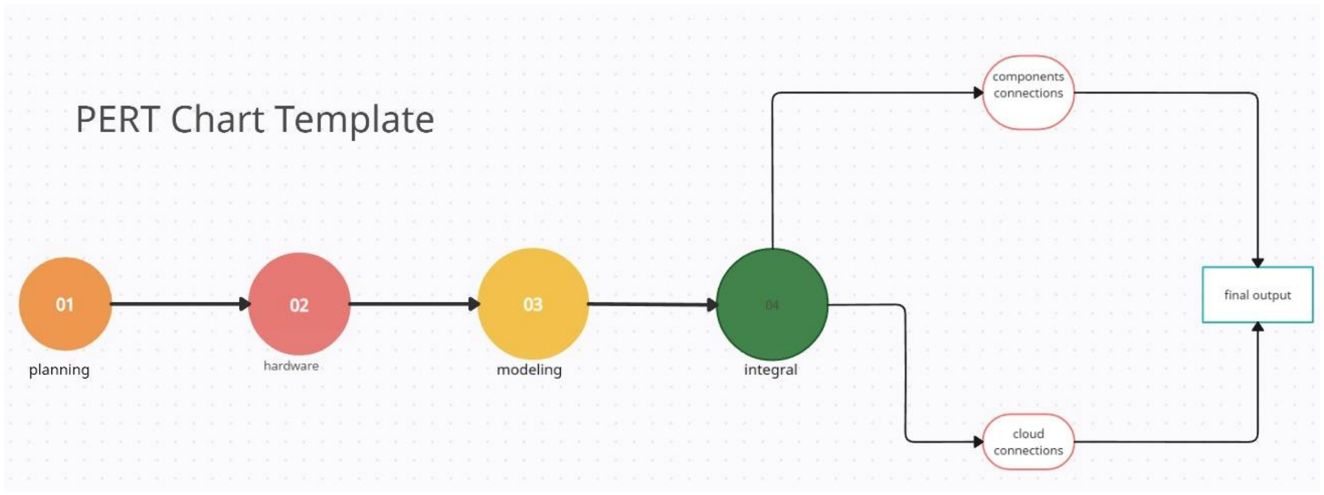
6. Regulatory Considerations:

- Research any existing regulations or permits needed for installing and operating the system in your area.

7. Next Steps:

- Based on the investigation, determine the feasibility of implementing a smart parking system.
- If viable, develop a detailed proposal outlining the specific technology, implementation plan, and projected outcomes.

3.3 Pert Chart



- **Planning:** This phase involves defining the project scope, objectives, and requirements. It includes tasks such as requirement gathering, feasibility analysis, and project planning.

- **Hardware:** In this phase, the hardware components required for the smart parking system, such as sensors, ESP32 modules, and actuators, are selected and procured. Tasks include hardware design, component selection, and procurement.

- **Modeling:** Here, the software models and algorithms required for the smart parking system are developed. This may include designing algorithms for parking slot detection, occupancy management, and user interfaces.

- **Integral:** This phase involves integrating the hardware and software components developed in the previous phases. Tasks include hardware-software integration, testing, and debugging.

- **Components Connections:** This sub-phase focuses on establishing connections between the hardware components, such as sensors, actuators, and ESP32 modules. Tasks include wiring, soldering, and configuring the connections.

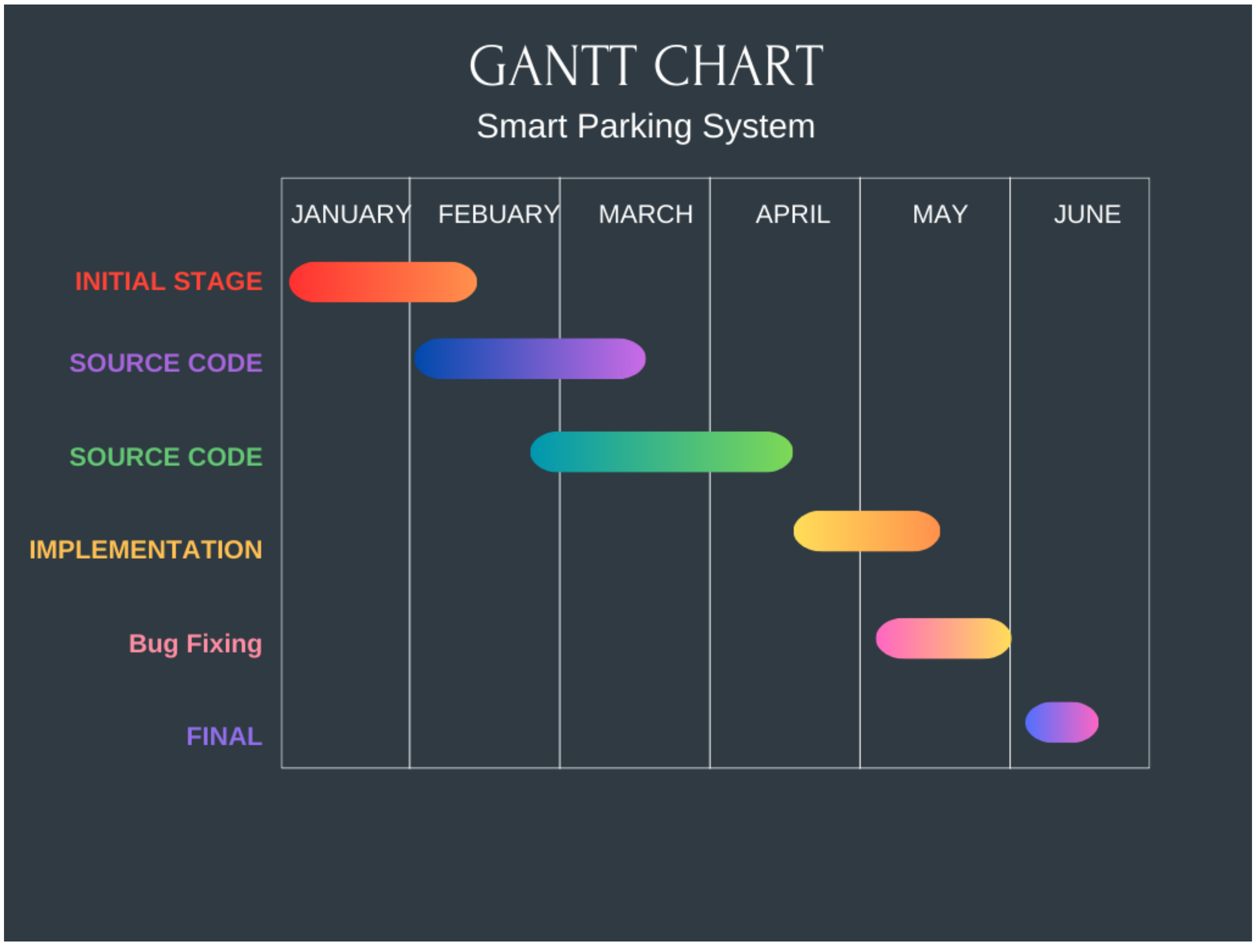
- **Cloud Connections:** In this sub-phase, connections to cloud services for data storage, processing, and remote access are established. This may involve setting up APIs, configuring cloud platforms, and ensuring data security.

- **Final Output:** The final phase involves testing the entire system and delivering the smart parking solution. Tasks include system testing, user acceptance testing, documentation, and deployment.

Each arrow in the chart represents a dependency between the phases or tasks. For example, hardware cannot be selected and procured until the planning phase is completed. Similarly, modeling must be finished before integration can begin. However, some tasks within a phase may be performed concurrently, as denoted by the branches in the chart.

3.4 Gantt Chart

A Gantt chart is a fundamental project management tool used to visually represent a project's schedule over time. It essentially functions as a bar chart with two key components:



- **Project Timeline:**

The chart spans from January to June.

Each month represents a time interval.

- **Stages and Tasks:**

The Gantt chart outlines the development of a Smart Parking System.

Key stages and tasks include:

- **Initial Stage (January):** Likely involves project planning, requirements gathering, and feasibility analysis.
- **Source Code (February - March):** Development of the system's code.
- **Implementation (May):** Deployment and integration of the system.
- **Bug Fixing (May):** Addressing any issues or defects.
- **Final (June):** Completion and delivery of the system.
- **Visualization:**

Each colored bar represents a task's duration.

The chart visually shows the sequence of tasks and their overlap.

- **Purpose:**

Gantt charts help project managers track progress, allocate resources, and manage dependencies

3.5 Feasibility Study

A feasibility study for smart parking involves a comprehensive assessment to determine the practicality, viability, and potential success of implementing a smart parking system in a specific location or area. This study typically examines various factors, including technical, financial, operational, legal, and environmental aspects, to evaluate whether the smart parking solution is feasible and beneficial. The primary goal of the feasibility study is to provide decision-makers with the information needed to determine whether to proceed with the implementation of the smart parking project.

4.1 Technical Feasibility

- **Hardware:** Assess the availability and compatibility of hardware components like sensors, cameras, and network infrastructure required for the smart parking system. Evaluate if the existing infrastructure can support the integration of new technology.

- **Software:** Determine the feasibility of developing or integrating software for real-time monitoring, data analysis, and user interfaces. Consider factors such as scalability, security, and interoperability with existing systems.

- **Connectivity:** Evaluate the feasibility of establishing reliable connectivity (e.g., Wi-Fi, cellular networks) to ensure seamless communication between parking sensors, central servers, and user interfaces.

- **Power Supply:** Assess the availability and reliability of power sources to support continuous operation of sensors, cameras, and other electronic components.

4.2. Economic Feasibility

- **Cost Estimation:** Conduct a comprehensive cost analysis, including the initial investment in hardware (sensors, communication devices), software development, installation, and ongoing maintenance and support.
- **Revenue Projection:** Estimate potential revenue streams from the smart parking system, such as parking fees, subscription models, or advertising partnerships.
- **Return on Investment (ROI)**: Calculate the expected ROI based on the projected costs and revenues over a specified period, considering factors like user adoption rates and market demand.
- **Risk Assessment:** Identify potential risks and uncertainties that could impact the financial viability of the project, such as changes in technology, regulatory requirements, or competition.

4.2 Operational Feasibility

- **User Needs Analysis:** Assess the needs and preferences of both parking operators and users to ensure that the smart parking system meets their requirements.

- **Process Evaluation:** Evaluate the operational processes involved in deploying and managing the smart parking system, including sensor installation, data collection, payment processing, and customer support.

- **Training and Support:** Develop a plan for training staff and providing ongoing support to ensure smooth operation and user satisfaction.
- **Regulatory Compliance:** Ensure compliance with relevant regulations and standards related to parking management, data privacy, and security to minimize legal and operational risks.

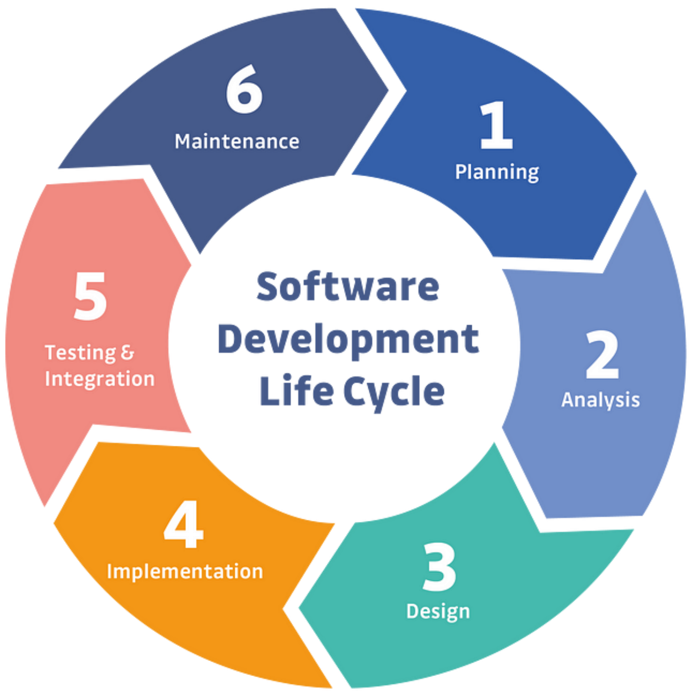
Here are some additional things to consider for operational feasibility:

- **User Accessibility and Inclusivity**: Ensure that the smart parking system is accessible to all users, including those with disabilities, by providing alternative methods for payment and information access. Consider offering features such as reserved parking spaces for people with disabilities and integration with navigation apps for users with visual impairments.

- **Service Level Agreements (SLAs) and Performance Metrics:** Establish SLAs with service providers and define performance metrics to monitor the effectiveness and reliability of the smart parking system. This includes metrics such as sensor accuracy, response time for detecting available parking spaces, and system uptime.

- **Customer Support and Feedback Mechanisms:** Implement robust customer support channels, including helplines, email support, and in-app chat, to address user inquiries, technical issues, and feedback promptly. Regularly solicit feedback from users to identify areas for improvement and enhance user satisfaction.

4.SDLC (Software Development Life Cycle)



The ESP32 is a powerful and versatile microcontroller well-suited for developing a smart parking system. Here's a breakdown of a possible SDLC (Software Development Life Cycle) for your project:

1. Planning and Requirements Gathering:

- Define the project scope and objectives. What functionalities do you want in your system (e.g., basic occupancy detection, real-time availability display)?
- Identify stakeholders (developers, users, parking management) and their needs.
- Research existing smart parking solutions and relevant regulations.
- Gather information about the parking environment (number of spaces, layout, potential network limitations).

2. System Design and Architecture:

- Design the overall system architecture, including hardware components (ESP32 board, sensors, network modules) and software components (data collection, processing, communication).
- Choose appropriate sensors based on your requirements, considering factors like detection method, range, and cost.
- Select a communication protocol (e.g., Wi-Fi, Bluetooth) for data transmission between ESP32 and other components.
- Design the data storage and processing strategy. Will data be stored locally on the ESP32 or transmitted to a cloud platform?

3. Development and Implementation:

- Develop the software for the ESP32 using the Arduino IDE or other compatible platforms. This may involve:
    - Code for sensor data acquisition and processing.
    - Communication protocols for data transmission.
    - User interface logic for a mobile app (if applicable) or a display system showing parking availability.

4. Testing and Integration:

- Conduct thorough unit testing of individual software modules on the ESP32.
- Integrate all hardware and software components and perform system-level testing to ensure functionality under various conditions.
- Test the system in a simulated parking environment (if possible) before real-world deployment.

5. Deployment and Maintenance:

- Deploy the smart parking system in the actual parking lot, ensuring proper sensor installation and network connectivity.
- Develop a maintenance plan for periodic checks of sensor functionality, system performance, and software updates.
- Monitor system performance and user feedback to identify areas for improvement or future enhancements.

Additional Considerations for ESP32 Development:

- **Power Management:** The ESP32 offers various power-saving modes to optimize battery life, especially important for long-term deployments with battery-powered sensors.
- **Security:** Implement security measures to prevent unauthorized access to sensor data or system manipulation.
- **Scalability:** Consider the scalability of your design if you plan to expand the system to more parking spaces in the future.

Tools and Resources:

- Arduino IDE (https://www.arduino.cc/) for ESP32 development
- ESP32 development resources and libraries from Espressif Systems (https://www.espressif.com/)
- Cloud platforms like Thingspeak (if applicable for data storage and processing)

5. SRS (Software Requirement Specification)

Product's purpose:

This document outlines the software requirements for an IoT-based smart parking system utilizing the ESP32 microcontroller. The system aims to improve parking management efficiency and driver experience by providing real-time parking availability information.

Details of the requirements:

The smart parking system will consist of the following components:

- **Hardware:**
  - ESP32 microcontroller
  - Infrared sensor
  - Servo SG90 Motor
  - LCD I2C Display
  - Jump Wires
  - Bread Board

- **Software:**
  - Arduino IDE
  - ThingSpeak Cloud server

Functional Requirements:

- **Sensor Data Acquisition:**
  - The system shall continuously collect data from ultrasonic sensors at a predefined interval (e.g., 1 second) to detect vehicle presence in each parking space.

- ➢ The system shall implement error handling mechanisms to identify and report sensor malfunctions (e.g., signal loss, out-of-range readings).

- **Parking Space Status:**

  - ➢ The system shall determine the real-time occupancy status (occupied/vacant) of each parking space based on sensor data and pre-defined thresholds (e.g., distance to object).

- **Data Communication:**

- The system shall provide options for data communication:

  - ➢ Local data visualization (optional): The system shall transmit real-time parking space occupancy data to control LEDs (or other visual indicators) to display the status (e.g., green for vacant, red for occupied).
  - ➢ Remote data transmission (optional): The system shall transmit parking space occupancy data to a cloud server using a lightweight messaging protocol (e.g., MQTT) for remote access and data storage.

Non-Functional Requirements

- **Performance:**

  - ➢ The system shall achieve a low latency in detecting vehicle occupancy changes (e.g., within 1 second).

  - ➢ The system shall minimize network bandwidth usage by efficiently transmitting data packets (especially if using a cloud server).

- **Reliability:**

  - ➢ The system shall operate continuously with minimal downtime.

- The system shall be resistant to sensor malfunctions and provide error handling to maintain data integrity.

- **Security:**
  - The system shall implement security measures to protect data transmission from unauthorized access (if applicable with cloud server communication). This may involve password protection or encryption.

Dependencies

- The system functionality depends on the chosen ultrasonic sensors and their communication protocols.

- Integration with a cloud server and mobile application development requires additional software development.

Future Enhancements

- Integrate with a payment system for online parking reservations.
- Implement real-time parking guidance using directional arrows or on-screen instructions within the mobile application.
- Develop a historical data analysis module to understand parking usage patterns and optimize resource allocation.

6. Module Description

The ESP32 microcontroller acts as the heart of the smart parking system, orchestrating various functionalities to deliver real-time parking information. Here's a detailed breakdown of its key roles:

IR Sensor for Vehicle Detection:

- The ESP32 communicates with the IR sensor to determine vehicle presence in a parking space. Unlike ultrasonic sensors, IR sensors typically detect interruptions in an infrared beam caused by a vehicle entering or exiting the space.

- The ESP32 reads the sensor's digital output (high/low) to identify a blocked beam (vehicle present) or an unblocked beam (vacant space).

- Similar to ultrasonic sensors, error handling routines are crucial to account for sensor malfunctions or external light interference that might affect readings.

Servo Motor for Gate Control (Optional):

- The ESP32 can be programmed to control a servo motor for applications like opening and closing a barrier gate at the parking entrance/exit.
- Based on pre-defined logic (e.g., space availability or user authentication), the ESP32 sends control signals to the servo motor, instructing it to move to a specific position (open/closed gate).

I2C LCD for Local Display:

- The ESP32 utilizes the I2C (Inter-Integrated Circuit) communication protocol to interact with the LCD (Liquid Crystal Display). I2C allows efficient data transfer with minimal required pins.

- The ESP32 transmits data to the LCD, controlling what characters or messages are displayed. This can be used to show real-time parking availability information (e.g., total vacant spaces) or other relevant messages.

System Functionalities:

- **Parking Space Status:** Similar to the previous scenario with ultrasonic sensors, the ESP32 processes IR sensor data to determine parking space occupancy (occupied/vacant).

- **Local Data Visualization:** The ESP32 transmits data to the I2C LCD, displaying the overall parking status (e.g., number of vacant spaces) or individual space statuses (depending on the number of IR sensors deployed).

- **Gate Control (Optional):** The ESP32 controls the servo motor to manage the gate based on pre-defined logic. This could involve opening the gate upon vehicle entry detection or requiring user authentication (not covered in this basic configuration) before allowing entry.

Additional Considerations:

- **Number of IR Sensors:** The number of IR sensors deployed depends on the desired granularity of the system. You can have one sensor per space for individual status display or fewer sensors strategically placed to cover multiple spaces for a general occupancy indication.

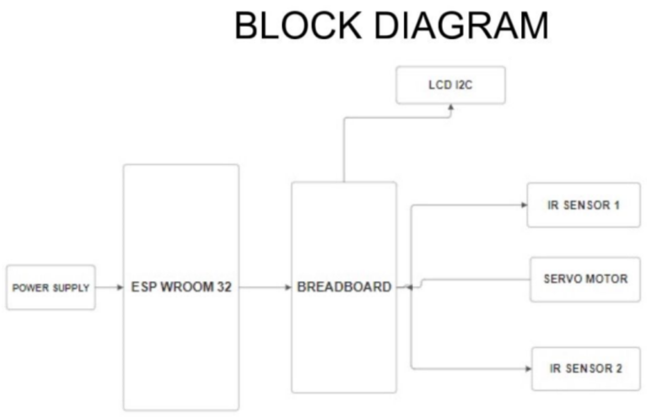
- **Power Management:** While the ESP32 is relatively power-efficient, consider optimizing code and adding sleep modes to conserve battery life, especially for portable applications.

Comparison with Ultrasonic Sensors:

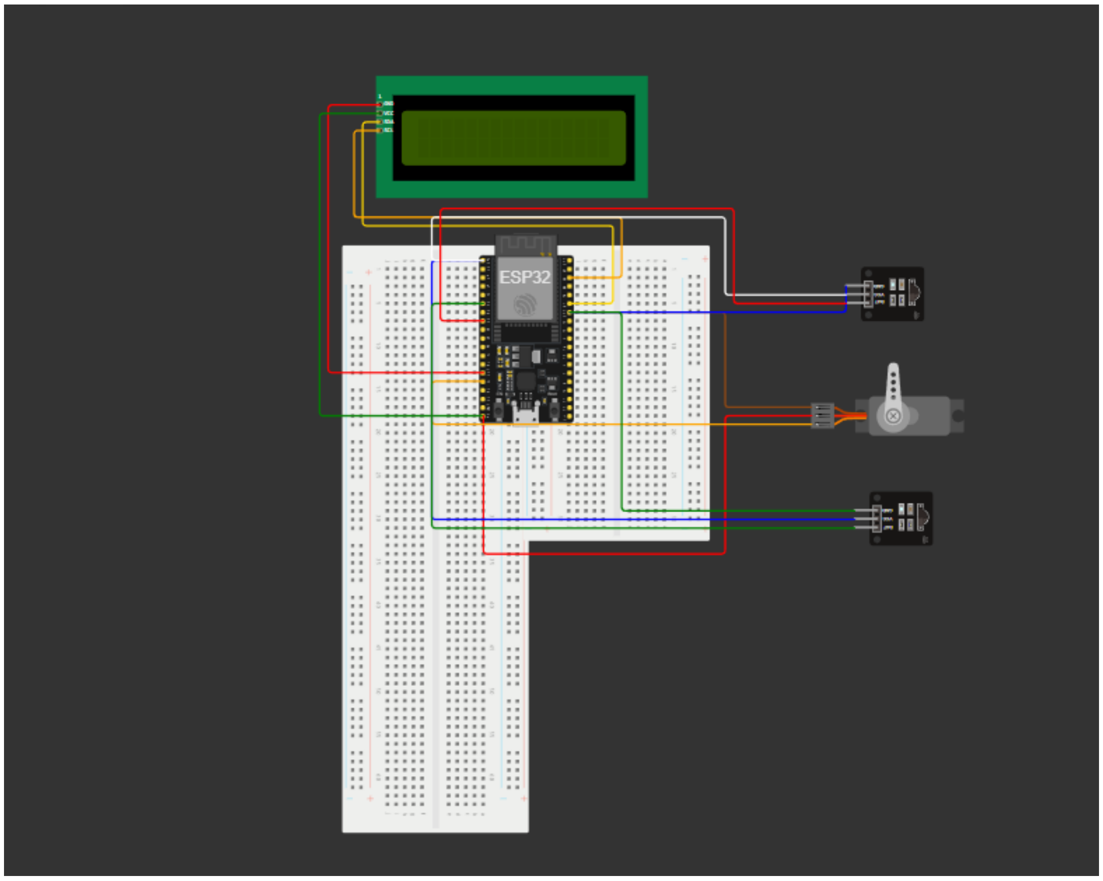
- IR sensors generally have a shorter detection range compared to ultrasonic sensors.
- They might be more susceptible to external light interference depending on the sensor type and placement.
- However, IR sensors offer a simpler and potentially more cost-effective solution for basic parking space occupancy detection.

7.Design

Block Diagram

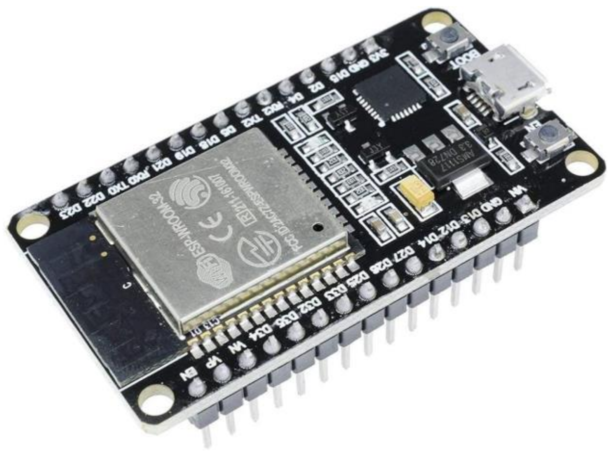


Circuit Diagram



Components Used

1.ESP32:



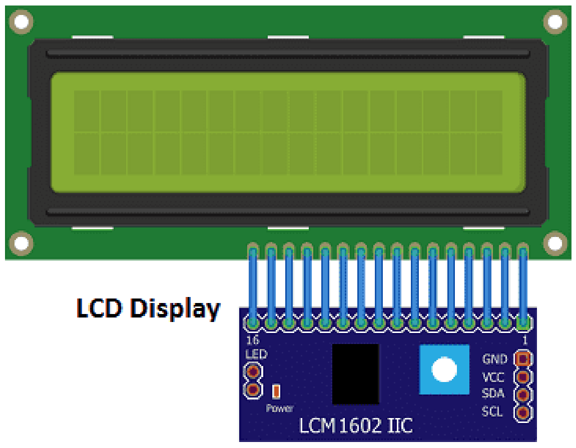
The ESP32 microcontroller, developed by Espressif Systems, stands out in the realm of embedded systems due to its versatile features and capabilities. Here's an elaborate look at its key components and functionalities:

1. **Dual-Core Processor:** The ESP32 is equipped with a dual-core processor, which enhances its processing power and multitasking capabilities. This feature allows the microcontroller to handle multiple tasks simultaneously, making it suitable for applications requiring real-time processing or complex computations.
2. **Connectivity Options:** One of the standout features of the ESP32 is its built-in support for various connectivity options, including Wi-Fi and Bluetooth. This enables seamless communication with other devices and networks, making it ideal for IoT (Internet of Things) projects that require wireless connectivity.
3. **GPIO Pins:** The ESP32 offers a wide range of GPIO (General Purpose Input/Output) pins, providing flexibility for interfacing with external components such as sensors, actuators, displays, and more. These pins can be configured to perform different functions based on the

requirements of the project, allowing for customization and versatility in design.

4. **Support for Communication Protocols:** In addition to Wi-Fi and Bluetooth, the ESP32 supports a variety of communication protocols such as SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), UART (Universal Asynchronous Receiver-Transmitter), and more. This makes it compatible with a wide range of sensors, peripherals, and communication modules, facilitating seamless integration into IoT ecosystems.
5. **Versatility for IoT Projects:** The combination of its dual-core processor, connectivity options, GPIO pins, and support for communication protocols makes the ESP32 well-suited for a diverse range of IoT applications.

2.LCD I2C:



1. **LCD Type:** The LCD is typically a character-based display, commonly available in configurations such as 16x2 (16 characters per line, 2 lines) or 20x4 (20 characters per line, 4 lines). These displays can show alphanumeric characters and symbols.

2. **I2C Interface:** The I2C interface (also known as TWI - Two-Wire Interface) is a popular serial communication protocol for connecting peripheral devices to microcontrollers. It requires only two wires for communication - SDA (Serial Data Line) and SCL (Serial Clock Line).

3. **I2C Addressing**: Each I2C device on the bus has a unique 7-bit address. LCDs with I2C interfaces typically have configurable addresses, allowing you to connect multiple LCDs to the same I2C bus without address conflicts.

4. **Backlight Control:** Many LCDs with I2C interfaces also feature an integrated LED backlight. The intensity of the backlight can usually be controlled through software, allowing you to adjust the brightness of the display.

5. **Power Requirements:** LCDs with I2C interfaces are generally low-power devices, operating at voltage levels compatible with microcontrollers (usually 5V or 3.3V).

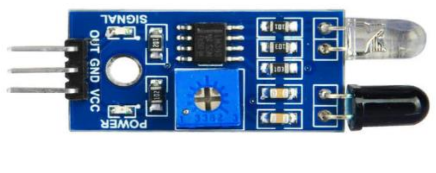
6. **Library Support:** There are various libraries available for interfacing LCDs with I2C interfaces to popular microcontroller platforms such as Arduino, Raspberry Pi, and ESP32. These libraries simplify the process of initializing and controlling the display, allowing you to focus on your application logic.

7. **Features:** LCDs with I2C interfaces often come with additional features such as custom character support, cursor control, and scrolling capabilities. These features enhance the flexibility and usability of the display in various applications.

8. **Compatibility:** LCDs with I2C interfaces are compatible with a wide range of microcontroller platforms that support the I2C protocol. This includes popular platforms such as Arduino, Raspberry Pi, ESP8266, ESP32, STM32, and more.

9. **Applications:** LCDs with I2C interfaces are commonly used in embedded systems, IoT devices, DIY projects, and educational purposes for displaying real-time data, status information, sensor readings, and user interfaces.

3.IR Sensor:



Infrared (IR) sensors are devices that can detect and measure infrared radiation, which lies beyond the visible spectrum of light. Here's some general information about them:

1. **Operating Principle:** IR sensors typically work based on one of two principles:

2. **Passive Infrared (PIR):** These sensors detect changes in infrared radiation caused by movement of objects within their detection range. They consist of a pyroelectric sensor, which generates an electric charge when exposed to infrared radiation, and a circuit that detects changes in this charge.
3. **Active Infrared (IR):** Active IR sensors emit infrared radiation and then detect the reflected or emitted radiation to determine the presence or absence of objects in their detection range.
4. **Detection Range:** The detection range of an IR sensor depends on its design, sensitivity, and the wavelength of the infrared radiation it is designed to detect. PIR sensors typically have a detection range of a few meters, while active IR sensors may have longer ranges depending on the power of the emitted infrared radiation.

Applications:

1. **Motion Detection:** PIR sensors are commonly used in motion detection systems for security, automatic lighting, and occupancy

sensing applications. They can trigger an alarm or activate lighting when motion is detected.

2. **Proximity Sensing:** Active IR sensors are used for proximity sensing and object detection in various applications such as robotics, automation, and industrial safety systems.

Temperature Measurement: Some IR sensors are designed for non-contact temperature measurement based on the amount of infrared radiation emitted by an object, commonly used in industrial processes, HVAC systems, and medical devices.

Types of IR Sensors:

1. **Single Element Sensors:** These sensors consist of a single infrared detector and are often used for simple applications such as motion detection.
2. **Array Sensors:** These sensors contain multiple infrared detectors arranged in a grid pattern, providing more detailed information about the distribution of infrared radiation in their field of view.
3. **Pyroelectric Sensors:** These sensors generate an electric charge in response to changes in temperature caused by infrared radiation, making them suitable for PIR applications.
4. **Infrared Thermopile Sensors:** These sensors measure the temperature of objects based on the heat absorbed by a thermopile detector array.

5. **Considerations:** When using IR sensors, factors such as ambient temperature, humidity, and interference from other sources of infrared radiation should be considered to ensure reliable operation.

4.Servo Motor:

The SG90 is a popular and widely used micro servo motor. Here's some information about it:

1. **Size and Form Factor:** The SG90 servo motor is a small-sized servo commonly used in hobbyist projects and small-scale robotics. It typically measures around 23mm x 12mm x 29mm and weighs approximately 9 grams.

2. **Operating Voltage:** The operating voltage of the SG90 servo motor is commonly in the range of 4.8V to 6V. It is compatible with most commonly available power sources such as AA batteries, rechargeable batteries, or DC power supplies.

3. **Torque and Speed:** The SG90 servo motor typically provides a torque output of around 1.5kg/cm (approximately 17 oz/in) at 4.8V. The speed of the motor is approximately 0.1 sec/60° at no load.

4. **Control Interface:** The SG90 servo motor uses a standard 3-wire control interface:

5. **Power (VCC):** Connected to the positive terminal of the power supply.
6. **Ground (GND):** Connected to the ground terminal of the power supply.
7. **Control Signal (PWM):** Connected to a microcontroller or servo controller to send PWM (Pulse Width Modulation) signals for controlling the position of the servo motor shaft.
8. **Operating Range:** The SG90 servo motor typically has a rotation range of approximately 180 degrees, although the actual range may vary slightly depending on the specific model and manufacturer.

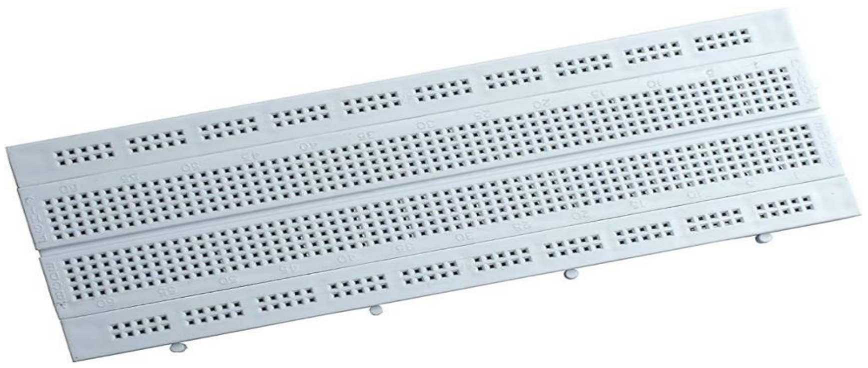
9. **Construction:** The SG90 servo motor consists of a small DC motor, a gearbox, a control circuit, and a potentiometer for position feedback. The gearbox helps increase torque output and reduce the speed of the motor.

10. **Usage:** The SG90 servo motor is commonly used in various hobbyist and DIY projects, including robotics, remote-controlled vehicles, model airplanes, drones, robotic arms, and more. Its small size,

lightweight, and ease of use make it suitable for a wide range of applications.

11. **Compatibility:** The SG90 servo motor is compatible with most microcontroller platforms, including Arduino, Raspberry Pi, and others. It can be easily controlled using PWM signals generated by these microcontrollers.

5.BreadBoard:



A breadboard, also known as a plugblock, serves as a fundamental tool in electronics prototyping, allowing engineers, hobbyists, and students to construct temporary circuits quickly and conveniently. Its versatility and ease of use make it invaluable for designing, testing, and iterating electronic circuits before final implementation on more permanent platforms. Here's a detailed elaboration on the functionality and benefits of breadboards:

1. **Functionality:** A breadboard consists of a plastic base with numerous holes arranged in a grid pattern. These holes are interconnected internally in specific configurations, typically following a common layout. The holes are designed to accommodate the leads of electronic components such as resistors, capacitors, integrated circuits (ICs), and jumper wires. By inserting components into the breadboard's

holes and connecting them with jumper wires, users can create electrical connections to build circuits without soldering.

Key Features:

1. **Removability:** One of the primary advantages of breadboards is that components can be easily inserted and removed without the need for soldering. This allows for rapid experimentation and iteration during circuit design and testing.

2. **Reusable Components:** Breadboards enable users to build circuits temporarily to demonstrate their functionality, after which components can be reused in other circuits. This promotes resource efficiency and cost-effectiveness, particularly for hobbyists or educational settings with limited budgets.

3. **Interconnectivity:** Breadboards feature internal connections (often referred to as buses or rails) that span across multiple rows or columns of holes. These connections allow for easy distribution of power and ground signals to different sections of the breadboard, simplifying circuit layout and wiring.

4. **Flexibility:** Breadboards come in various sizes and configurations, catering to different project requirements. Some breadboards feature multiple power rails and ground buses, while others may include built-in components such as LEDs or push-button switches for prototyping specific types of circuits.

5. **Visual Clarity:** Breadboards typically have labeled rows and columns, making it easier for users to organize and document their circuits. This enhances clarity and facilitates troubleshooting and debugging during circuit development.

6.Jump Wires:

Jumper wires are essential components used in electronics prototyping and wiring projects. Here's some information about them:

1. **Purpose:** Jumper wires are used to establish electrical connections between components on a breadboard, circuit board, or between different points in an electronic circuit. They help in creating temporary or permanent connections, enabling signal transmission, power distribution, and data transfer within the circuit.

Types:

1. **Male-to-Male (M-M):** These jumper wires have pins or connectors at both ends, allowing them to connect two female headers or pins.

2. **Male-to-Female (M-F):** These jumper wires have a pin or connector at one end and a socket or receptacle at the other end, enabling them to connect a male pin to a female header.
3. **Female-to-Female (F-F):** These jumper wires have sockets or receptacles at both ends, allowing them to connect two male pins or headers.
4. **Lengths:** Jumper wires are available in various lengths, typically ranging from a few centimeters to several inches. The length of the jumper wire determines the distance between the components it connects and helps in organizing the wiring on the breadboard or circuit board.
5. **Wire Gauge:** The wire gauge of jumper wires varies depending on the application and current requirements. Thicker gauge wires can carry higher currents without overheating, while thinner gauge wires are suitable for low-power applications and signal transmission.

6. **Colors:** Jumper wires are often color-coded to facilitate easy identification and organization of connections. Common colors include red, black, blue, green, yellow, white, and others. Color-coding helps in visually distinguishing between power, ground, signal, and other types of connections.

7. **Materials:** Jumper wires are typically made of stranded copper wire for flexibility and durability. The insulation material can be PVC

(Polyvinyl Chloride), silicone, or other thermoplastics, providing electrical insulation and protection against short circuits.

8. **Usage:** Jumper wires are widely used in electronics prototyping, breadboarding, circuit debugging, and educational projects. They are commonly used with development boards, microcontrollers, sensors, LEDs, motors, and other electronic components to create functional circuits and systems.

9. **Customization:** Some jumper wire kits or sets allow users to customize the length and type of connectors according to their specific requirements. Users can trim the wires to desired lengths and attach different types of connectors as needed.

SOFTWARE USED:-

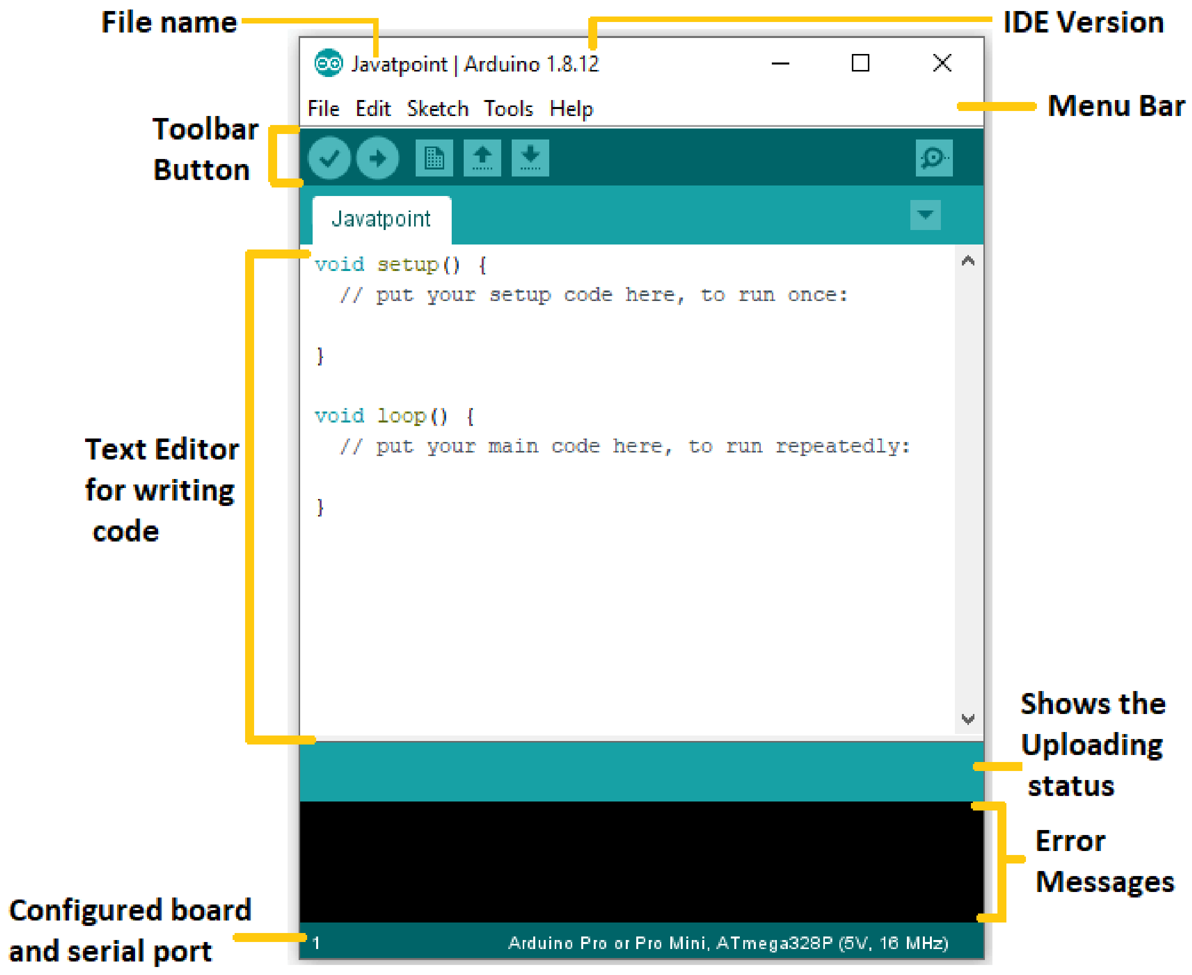
Arduino IDE

The Arduino IDE is an open-source software, which is used to write and upload code to the Arduino boards. The IDE application is suitable for different operating systems such as **Windows, Mac OS X, and Linux**. It supports the programming languages C and C++. Here, IDE stands for **Integrated Development Environment**.

The program or code written in the Arduino IDE is often called as sketching. We need to connect the Genuino and Arduino board with the IDE to upload the sketch written in the Arduino IDE software. The sketch is saved with the extension '.ino.'

Most Arduino boards consist of an Atmel 8-bit AVR microcontroller (ATmega8, ATmega168, ATmega328, ATmega1280, or ATmega2560) with varying amounts of flash memory, pins, and features. The 32-bit Arduino Due, based on the Atmel SAM3X8E was introduced in 2012. The boards use single or double-row pins or female headers that facilitate connections for programming and incorporation into other circuits. These may connect with add-on modules termed *shields*.Arduino microcontrollers are pre-programmed with a bootloader that simplifies the uploading of programs to the on-chip flash memory.

The Arduino IDE will appear as:


File name
IDE Version
Javatpoint | Arduino 1.8.12
File Edit Sketch Tools Help
Menu Bar
Toolbar Button
Javatpoint
Text Editor for writing code
Shows the Uploading status
Error Messages
Configured board and serial port
1
Arduino Pro or Pro Mini, ATmega328P (5V, 16 MHz)

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

- **Menu Bar:** Provides access to various features of the IDE, such as file management (e.g., creating, opening, saving sketches), editing options (e.g., copy, paste, undo, redo), debugging tools (e.g., finding and fixing errors in your code), and board selection (specifying the type of Arduino board you are connected to).
- **Toolbar:** Contains buttons for commonly used functions, such as verifying and uploading code (checking for errors in your code and transferring it to your Arduino board), opening and saving files, and creating new projects.
- **Text Editor:** The main window where you write your code. It includes features like syntax highlighting (making it easier to read code by coloring keywords and text according to their function), code completion (suggesting complete code statements as you type), and

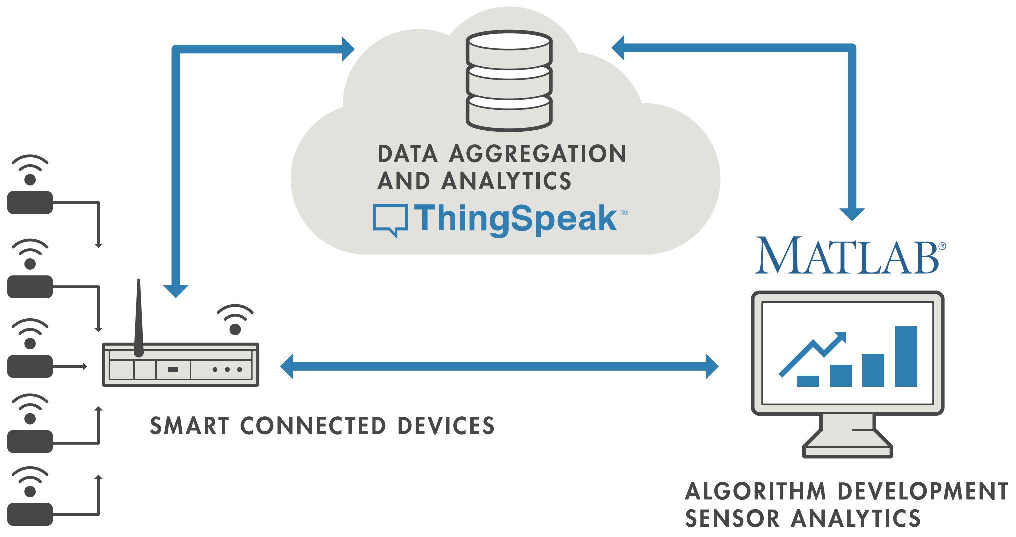
indentation (automatically formatting your code to improve readability).

- **File Name:** This field displays the name of the current sketch or project file that you are working on.
- **IDE Version:** This field displays the version of the Arduino IDE that you are currently using. In the image, it shows that Arduino version 1.8.12 is being used.
- **Sketch:** A sketch is the term used for a Arduino program. It is a text file that contains the code you write to control the behavior of your Arduino board.
- **void setup() { ... }:** This is a special function in Arduino code that is called once at the beginning of the program. You can use this function to initialize variables (storing data for use in your program), pin modes (configuring the electrical behavior of the Arduino's pins), and other settings for your program. For example, you might use the setup() function to define which Arduino pins will be used as inputs or outputs for sensors and actuators.
- **void loop() { ... }:** This is another special function in Arduino code that is called repeatedly after the setup function runs. The code within the loop() function continuously runs over and over again as long as your Arduino board is powered on. You can use this function to write the code that controls the main behavior of your program. For example, you might use the loop() function to read data from a sensor, process it, and then control an actuator like an LED or a motor.
- **Serial Monitor:** Allows you to communicate with your Arduino board by sending and receiving text messages. You can use it to print messages to the serial console on your computer screen, send data to the board, and monitor the output from your program. This is useful for debugging your code (finding and fixing errors) by allowing you to see what values your program is printing.
- **Uploading status:** Shows the status of the upload process, such as compiling (checking your code for errors), uploading (transferring the code to your Arduino board), and done (indicating that the code has been successfully uploaded).

- **Error messages:** This area will show any errors that occur during the compilation process. These error messages can help you identify and fix mistakes in your code.
- **Configured board:** This section displays the type of Arduino board that is currently selected in the IDE. The image shows that an Arduino Pro or Pro Mini, ATmega328P (5V, 16 MHz) board is selected. It is important to select the correct board type in the IDE so that the code is compiled with the appropriate settings for your board.
- **Messages:** This area shows additional messages from the IDE, such as board connection messages. For example, it might show a message indicating that the IDE has successfully connected to your Arduino board.

ThingSpeak Cloud:

ThingSpeak is an **IoT (Internet of Things) analytics platform** service offered by MathWorks, allowing you to collect sensor data from your IoT devices, send it to the cloud for storage, visualize it in real-time, and analyze it further. Here's a deeper dive into ThingSpeak's functionalities and what it offers:



Capabilities of ThingSpeak:

- **Data Acquisition from Diverse Devices:**
    - ThingSpeak is versatile in terms of compatible devices. You can connect microcontrollers like Arduino and Raspberry Pi, as well as various sensors and industrial instruments, to send data to ThingSpeak's cloud platform.
    - It supports two main methods for sending data:
- **REST API (Web Service):** This method allows you to send data from your device using HTTP requests. Libraries are available for various programming languages to simplify the process of interacting with ThingSpeak's API.

    - **MQTT (Message Queuing Telemetry Transport):** This is a lightweight protocol specifically designed for sending small messages from devices to the cloud. It's energy-efficient and works well with devices that have limited processing power or battery life.

- **Data Management and Storage:**
  - ThingSpeak provides secure storage for your sensor data in the cloud.
  - The amount of free storage space depends on the plan you choose (with limitations on the free tier). You can access and manage your stored data through the ThingSpeak platform's web interface or programmatically using its API.
  - ThingSpeak also offers features like data privacy control. You can choose to keep your data private or share it publicly to collaborate with others.
- **Data Visualization Tools:**
  - ThingSpeak offers a variety of tools to create informative visualizations of your sensor data.
  - You can generate charts and graphs to see trends, monitor changes, and identify patterns in your data. These visualizations can be customized with different display options like timeframes, data selection, and chart types (line graphs, bar graphs, etc.).
  - Visualizations are useful for gaining insights from your sensor data and can be easily shared with others for collaborative analysis.
- **Data Analysis and Processing:**
  - ThingSpeak integrates with MathWorks' MATLAB software, a powerful tool for data analysis.
  - You can leverage MATLAB for advanced analysis on your sensor data, such as filtering noise, performing statistical calculations, and even using machine learning algorithms to extract knowledge from your data.
  - While MATLAB requires a separate license, ThingSpeak itself offers basic data analysis features like calculating averages, minimums, and maximums within your data sets.
- **Alerts and Automated Actions:**

  - ThingSpeak allows you to set up alerts based on your sensor data.
  - You can define thresholds for your data (e.g., temperature exceeding a certain limit), and if those thresholds are crossed, ThingSpeak can send you email or SMS notifications, alerting you to potential issues.
  - ThingSpeak can also be configured to trigger actions based on your data. For instance, if a temperature sensor detects a critical overheating condition, ThingSpeak could automatically send a command to turn on a cooling system or trigger an alarm.

Advantages of Using ThingSpeak:

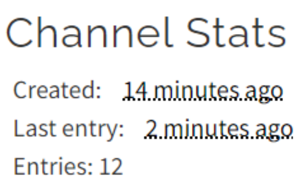
- **Simple and User-Friendly:** ThingSpeak offers an intuitive web interface for configuring data channels, sending data, and visualizing and analyzing your sensor data.
- **Free Tier Available:** A free tier with basic functionalities is available, making it accessible for hobbyists, students, and personal projects.
- **Supports Various Devices and Protocols:** ThingSpeak can work with a wide range of devices and sensors, and its support for REST API and MQTT allows for flexibility in integrating different devices.
- **Cloud-Based Platform:** There's no need to set up and maintain your own server infrastructure to manage and store your sensor data. ThingSpeak takes care of the backend, allowing you to focus on collecting data and deriving insights from it.
- **Powerful Data Analysis with MATLAB Integration:** For users who require advanced data analysis capabilities, ThingSpeak's integration with MATLAB opens doors to various mathematical and machine learning techniques.

Considerations When Using ThingSpeak:

- **Free Tier Limitations:** The free tier has limitations on data storage capacity, the number of channels you can create, and the frequency of data updates. Upgrading to a paid plan unlocks higher capacities and features.

- **Vendor Lock-in:** If you heavily rely on MATLAB for complex data analysis, you might be tied to the ThingSpeak platform for your data management and visualization needs.

7.Form Design/Snap

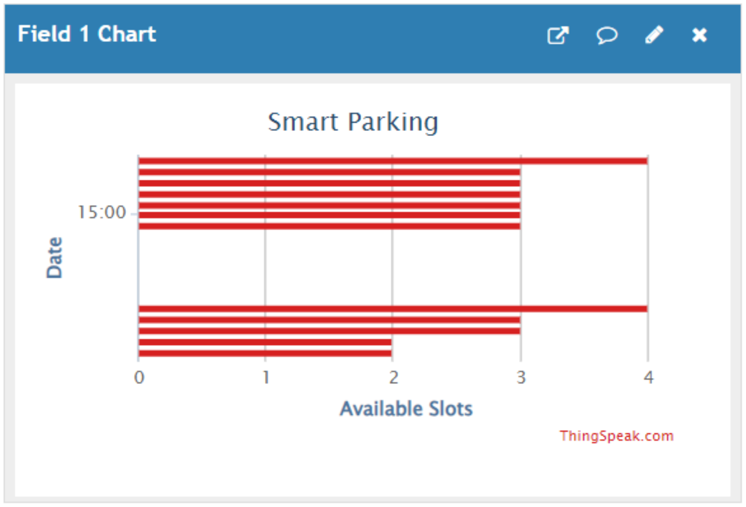


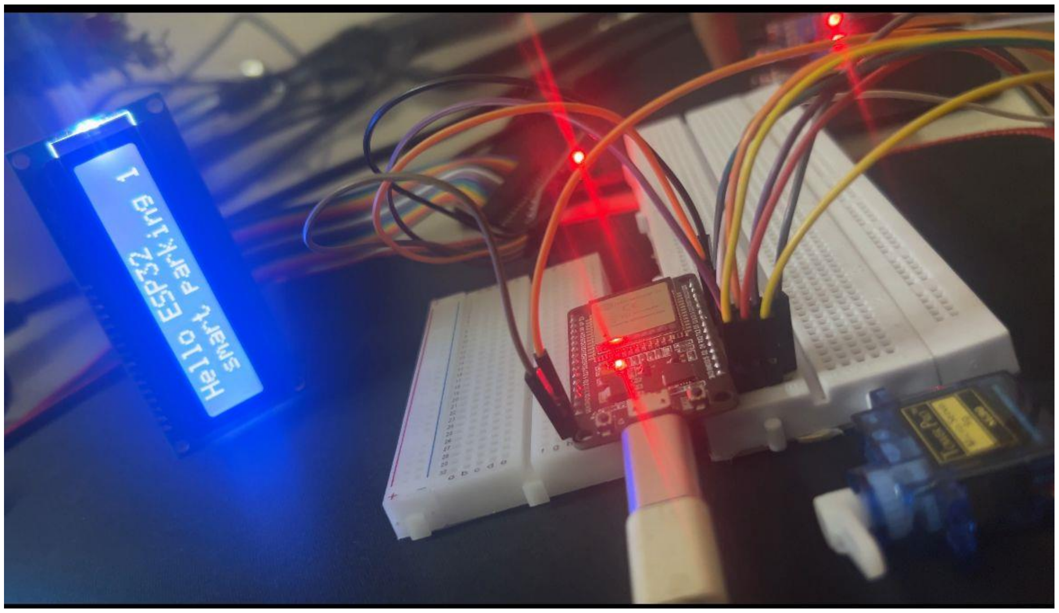
Channel Stats

Created: 14 minutes ago

Last entry: 2 minutes ago

Entries: 12





8.Coding

```
#include <Wire.h>

#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);  // Define I2C address,
columns and rows

#include <ESP32_Servo.h>

Servo myservo;

int IR1 = 33;
int IR2 = 35;

int totalSlots = 4;  // Total number of parking slots (changed
to totalSlots)
int availableSlots = totalSlots;  // Track available slots

int flag1 = 0;
int flag2 = 0;
```

```
void setup() {
   Serial.begin(9600);
  lcd.init();  // Initialize the LCD
  lcd.backlight(); // Open the backlight

  pinMode(IR1, INPUT);
  pinMode(IR2, INPUT);

  myservo.attach(13);
  myservo.write(0); // Set initial servo position (can be
adjusted)

  lcd.setCursor(0, 0);
  lcd.print("  SMART PARKING  ");
  lcd.setCursor(0, 1);
  lcd.print("  SYSTEM ");
  delay(2000);
  lcd.clear();

}
```

```
void loop() {
 if (digitalRead(IR1) == LOW && !flag1) {  // Check if car
enters and slot available
   if (availableSlots > 0) {
     flag1 = 1;
     if (!flag2) {
       myservo.write(90);
       delay(2000);
       myservo.write(0);  // Open barrier
       availableSlots--;  // Decrement available slots
     }
   } else {
     lcd.setCursor(0, 0);
     lcd.print("  SORRY :(  ");
     lcd.setCursor(0, 1);
     lcd.print(" Parking Full ");
     delay(2000);
     lcd.clear();
   }
 }
```

```
  if (digitalRead(IR2) == LOW && !flag2) {  // Check if car
exits
    if (availableSlots < totalSlots) {
      flag2 = 1;
      if (!flag1) {
        myservo.write(90);
        delay(2000);
        myservo.write(0); // Close barrier
        availableSlots++;  // Increment available slots
      }
    }
  }

  if (flag1 && flag2) {  // Reset flags after both sensors
triggered
    delay(1000);
    myservo.write(0);  // Set servo to initial position
    flag1 = 0;
    flag2 = 0;
  }

  lcd.setCursor(0, 0);
  lcd.print("  WELCOME!  ");
  lcd.setCursor(0, 1);
  lcd.print("Slot Left: ");
  lcd.print(availableSlots);
  delay(500); // Adjust display update frequency
}
```

ThinkSpeak Cloud:-

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <ESP32Servo.h>
#include <WiFi.h>
#include "ThingSpeak.h"

const char* ssid = "Tony";   // your network SSID (name)
const char* password = "invalid password";   // your network
password

const char* server = "api.thingspeak.com";
const unsigned long myChannelNumber = 2566289;
const char * myWriteAPIKey = "TCQRQ93UE6K7R79H";
```

```
WiFiClient client;

LiquidCrystal_I2C lcd(0x27, 16, 2);  // Define I2C address,
columns and rows
Servo myservo;

int IR1 = 33;
int IR2 = 35;
int totalSlots = 4;  // Total number of parking slots
int availableSlots = totalSlots;  // Track available slots
int flag1 = 0;
int flag2 = 0;

unsigned long lastTime = 0;
unsigned long timerDelay = 30000;

void setup() {
  Serial.begin(115200);
  lcd.init();
  lcd.backlight();
  pinMode(IR1, INPUT);
```

```
 pinMode(IR2, INPUT);
 myservo.attach(13);
 myservo.write(0);
 WiFi.mode(WIFI_STA);
 ThingSpeak.begin(client);
}

void loop() {
 if ((millis() - lastTime) > timerDelay) {
  if(WiFi.status() != WL_CONNECTED){
    Serial.print("Attempting to connect");
    while(WiFi.status() != WL_CONNECTED){
     WiFi.begin(ssid, password);
     delay(5000);
    }
    Serial.println("\nConnected.");
  }

  ThingSpeak.setField(1, availableSlots); // Available
Parking Slots
```

```
    int x = ThingSpeak.writeFields(myChannelNumber,
myWriteAPIKey);

    if(x == 200){
      Serial.println("Channel update successful.");
    }
    else{
      Serial.println("Problem updating channel. HTTP error
code " + String(x));
    }

    lastTime = millis();
  }

  if (digitalRead(IR1) == LOW && !flag1) {
    if (availableSlots > 0) {
      flag1 = 1;
      if (!flag2) {
        myservo.write(90);
        delay(2000);
        myservo.write(0);
        availableSlots--;
      }
    } else {
      lcd.setCursor(0, 0);
      lcd.print("  SORRY :(  ");
      lcd.setCursor(0, 1);
      lcd.print(" Parking Full ");
      delay(2000);
      lcd.clear();
    }
  }

  if (digitalRead(IR2) == LOW && !flag2) {
    if (availableSlots < totalSlots) {
      flag2 = 1;
      if (!flag1) {
        myservo.write(90);
        delay(2000);
        myservo.write(0);
        availableSlots++;
      }
    }
  }
```

```
  if (flag1 && flag2) {
    delay(1000);
    myservo.write(0);
    flag1 = 0;
    flag2 = 0;
  }

  lcd.setCursor(0, 0);
  lcd.print("  WELCOME!  ");
  lcd.setCursor(0, 1);
  lcd.print("Slot Left: ");
  lcd.print(availableSlots);
  delay(500);
}
```

9.Testing:

A well-tested smart parking system using an ESP32 unit ensures reliable and accurate operation. Here's a breakdown of the testing process, incorporating the best aspects of unit, integration, and system testing:

1. Unit Testing

- **IR Sensors:**
  - **Functionality:** Test for basic detection of presence/absence of a vehicle. Use blackbody sources or calibrated IR emitters to simulate real-world scenarios.
  - **Response Time:** Measure the time it takes for a sensor to detect a change in IR radiation. Aim for a fast response time for real-time parking availability.
  - **Accuracy:** Check for false positives/negatives by placing the sensor at different distances and angles from a vehicle. Calibrate if necessary.

- **Servo Motor:**
  - **End-to-End Travel Time:** Measure the time it takes for the servo to move from one extreme position to the other. This helps determine speed and suitability for your application.
  - **Stall Detection:** Implement current thresholds or retry logic to detect and handle situations where the servo encounters resistance. This prevents damage and ensures smooth operation.
  - **Holding Power:** Test the servo's ability to hold the weight of barriers or signage it needs to move. This ensures reliable operation.

- **LCD I2C Module:**
  - **Basic Functionality:** Verify if pixels, contrast, and brightness can be adjusted.

- **Text Display:** Test displaying various characters, text, and scrolling capabilities. This ensures clear communication of parking status.
- **Custom Messages:** Implement code to handle different scenarios (e.g., "Occupied," "Available," "System Error") and test their display.

2. Integration Testing

- **IR Sensor and Servo Integration:** Simulate sensor inputs during development to streamline testing. Verify that the servo responds correctly to the IR sensor's signals, indicating a parked vehicle.
- **IR Sensor, Servo, and LCD Integration:** Test if the system can detect a vehicle (IR sensor), trigger the servo movement (based on sensor data), and accurately display the parking status (e.g., "Occupied") on the LCD.
- **Communication Protocols:** If the system interacts with other devices (e.g., phone app for payment), test communication protocols (e.g., Bluetooth, Wi-Fi) for reliability and data integrity.

3. System Testing

- **Stress Testing:** Simulate real-world usage patterns by repeatedly detecting parked vehicles, actuating the servo, and updating the LCD. This helps identify potential bottlenecks or software issues.
- **Environmental Testing:** Test the system's functionality under different temperatures and humidity conditions to ensure it operates reliably in your target environment.
- **Power Supply:** Test if the power supply can deliver sufficient current under load (especially when multiple servos are activated) to avoid system malfunctions.

4.Test Results

The smart parking system utilizes an ESP32 microcontroller, an IR sensor, and cloud computing. When a car approaches a parking space, the IR sensor detects its presence and signals the ESP32. The ESP32 then activates a servo motor to open the barrier arm, allowing the car to enter. When the car leaves, the IR sensor again signals the ESP32, which closes the barrier arm.LCD Displays Slots Availability and Real-time data is sent to an Cloud for monitoring and accessibility.

10.Data validation checks

Data validation checks for a Smart Parking System using ESP32, along with additional considerations to enhance robustness:

Sensor Data Validation:

- **Range Checks:** Validate sensor readings against expected ranges. For example, an ultrasonic sensor reading might be invalid if it's less than the minimum parking space height (e.g., 6.5 ft for most vehicles) or greater than the maximum possible distance (typically around 13 ft for ultrasonic sensors).
- **Noise Checks:** Implement checks to identify and discard noisy sensor data. This could involve using techniques like averaging multiple readings over a short period (e.g., 10 readings) or checking for sudden spikes/dips in values that deviate significantly from the average.
- **Calibration Checks:** Regularly perform calibration checks on sensors to ensure their accuracy over time. This might involve using a reference voltage source for voltage sensors or a known distance object (e.g., a wall at a specific distance) for ultrasonic sensors. Calibration procedures should be documented and performed at defined intervals (e.g., weekly, monthly) depending on sensor type and environmental conditions.

Communication Data Validation:

- **Checksums:** Utilize checksums like CRC (Cyclic Redundancy Check) to detect errors introduced during data transmission between the ESP32 and other devices (e.g., mobile app server). Consider implementing error correction techniques alongside checksums for critical data transmissions (e.g., occupancy updates during peak hours) to automatically rectify errors.

- **Data Format Validation:** Ensure received data adheres to the expected format (e.g., JSON, message protocol). This could involve checking for missing fields, invalid data types (e.g., string entered for a numeric field), or unexpected characters. Define a validation schema to ensure data conforms to the expected structure.

- **Data Consistency Checks:** Validate data for internal consistency. For example, a car entering a parking lot should trigger a space occupancy change from available to occupied, and the total occupied spaces shouldn't exceed the available parking spots. Implement checks to identify inconsistencies that might indicate sensor malfunctions or communication errors.

User Input Validation:

- **Input Range Checks:** If the system accepts user input (e.g., car license plate number), ensure it falls within the expected range of characters or format (e.g., alphanumeric characters, specific number of characters depending on local regulations).
- **Data Type Checks:** Validate that user-entered data is of the correct type (e.g., numbers for parking slot IDs). Provide clear instructions and user interfaces that guide users towards entering data in the correct format.
- **Authorization Checks:** Implement mechanisms to restrict unauthorized access or actions (e.g., user authentication for managing parking reservations). Consider using secure tokens or encryption for sensitive data like user credentials.

Additional Considerations:

- **Timestamp Validation:** Verify timestamps for plausibility to identify potential clock drifts or data inconsistencies. Implement time synchronization mechanisms (e.g., NTP servers) to ensure all system components operate on a consistent time base.
- **Data Logging:** Log sensor readings, communication data, and user interactions for debugging and anomaly detection purposes. Store logs securely and implement mechanisms to archive and retrieve logs for troubleshooting and analysis.

- **Error Handling:** Implement proper error handling routines to gracefully handle invalid data and provide informative feedback to the user or system administrator. Error messages should be specific and actionable, guiding users towards resolving issues or notifying administrators of potential system malfunctions.

11. Implementation

Implementing a smart parking system using ESP32, a microcontroller with built-in Wi-Fi and Bluetooth capabilities, involves several steps. Here's a generalized outline of the process:

1. **Requirement Analysis:** Understand the requirements of the smart parking system. Determine the number of parking spaces, the type of sensors to be used (such as ultrasonic sensors or infrared sensors), and the desired features like real-time monitoring, mobile app integration, etc.

2. **Hardware Selection:** Choose the necessary hardware components, including ESP32 microcontrollers, sensors, actuators (if needed), and any additional peripherals like LED displays or buzzers.

3. **Prototype Design:** Design a prototype of the smart parking system, including the layout of sensors within the parking spaces, the placement of ESP32 microcontrollers, and the overall system architecture.

4. **Hardware Setup:** Assemble the hardware components according to the prototype design. Install the ESP32 microcontrollers and sensors in each parking space, ensuring they are securely mounted and properly connected.

5. **Software Development:** Develop the software for the smart parking system. This includes firmware for the ESP32 microcontrollers to read sensor data, communicate with a central server or gateway, and control actuators as necessary. Additionally, develop server-side software for data processing, storage, and user interface.

6. **Network Configuration:** Configure the Wi-Fi network for the ESP32 microcontrollers to connect to. Set up the necessary network infrastructure, such as access points or routers, and configure security settings as required.

7. **Integration and Testing:** Integrate the hardware and software components together to create a functioning smart parking system. Test the system thoroughly to ensure that sensors accurately detect parking occupancy, ESP32 microcontrollers communicate with the central server, and the user interface functions as expected.

8. **Deployment:** Deploy the smart parking system in a controlled environment for testing and evaluation purposes. This may involve installing the system in a specific parking lot or facility and providing access to testers or evaluators.

9. **Evaluation and Feedback:** Gather feedback from users and testers to evaluate the performance of the smart parking system. Identify any

issues or areas for improvement and make necessary adjustments to the hardware or software.

10. **Documentation and Training:** Document the implementation process, including hardware setup, software configuration, and troubleshooting steps. Provide training for administrators and users on how to operate the smart parking system effectively.

12.Security measures taken

When implementing a Smart Parking System (SPS) using ESP32 microcontrollers in IoT, ensuring robust security measures is paramount to protect against potential threats and vulnerabilities. Here are specific security measures tailored for such a system:

1. **Secure Authentication:** Implement strong authentication mechanisms for accessing the smart parking system, including user authentication for administrators and authorization mechanisms to restrict access to sensitive functions and data.

2. **Encryption of Communication:** Ensure that all communication between ESP32 devices, gateways, and backend servers is encrypted using protocols like HTTPS or MQTT with TLS/SSL. This prevents eavesdropping and data tampering during transmission.

3. **Device Identity Management:** Assign unique identities to each ESP32 device within the system and implement device authentication mechanisms, such as digital certificates or API keys, to verify the identity of devices connecting to the network.

4. **Access Control:** Enforce access control policies to limit the privileges of users and devices within the smart parking system. Use role-based access control (RBAC) to assign specific permissions based on user roles and responsibilities.

5. **Secure Firmware Updates:** Implement secure over-the-air (OTA) update mechanisms to deploy firmware updates to ESP32 devices. Ensure that updates are signed, encrypted, and authenticated to prevent unauthorized modifications or tampering with device firmware.

6. **Data Encryption at Rest:** Encrypt sensitive data stored on ESP32 devices, such as configuration settings or access credentials, to protect against unauthorized access in case of physical theft or tampering.

7. **Intrusion Detection and Prevention:** Deploy intrusion detection systems (IDS) or anomaly detection algorithms to monitor network traffic and detect potential security breaches or suspicious activities in real-time.

8. **Secure Boot and Firmware Integrity Checking:** Implement secure boot mechanisms to verify the integrity and authenticity of device firmware during the boot-up process. Perform regular checks to ensure that firmware remains unaltered and free from tampering.

9. **Regular Security Audits and Updates:** Conduct regular security audits and vulnerability assessments to identify and address potential security weaknesses in the smart parking system. Stay updated with security patches and firmware updates released by the ESP32 manufacturer to mitigate known vulnerabilities.

13.Cost estimation of the project

Hardware Costs:

1.ESP32 Microcontroller: Rs 350

2. IR Sensor: Rs 150

3.LCD I2C Screen: Rs 250

4.Servo Motor SG90: Rs 70

5.Bread Board: Rs 130

6.Jump Wire: Rs 250

Total Hardware Costs: Rs 1200

Miscellaneous Costs:

1.Component Pickup: Rs 450

2.CardBoard ,Black chart,Tap: Rs 200

Total Miscellaneous Costs: Rs 650

Total Estimated Cost(Including Miscellaneous Costs): Rs 1850

14.Maintenance

An IoT-based smart parking system relies on a network of sensors, software, and cloud connectivity to function smoothly. To ensure its optimal performance and longevity, a well-defined maintenance plan is crucial. Here's a breakdown of key maintenance aspects:

Preventive Maintenance: Proactive Measures for Peak Performance

- **Sensor Care:** The backbone of the system, sensors require regular attention. Schedule periodic calibration (especially for ultrasonic and magnetic sensors) and cleaning to ensure accurate data on vehicle presence. Environmental factors like dust, debris, or extreme weather can affect sensor performance. Strategic placement and cleaning routines can minimize these issues.
- **Software Updates:** Stay updated! Regularly schedule software updates for central systems and sensor devices. These updates address bugs, patch vulnerabilities, and introduce new features like improved parking guidance or mobile payment integrations.
- **Inspections:** Conduct routine inspections of all system components, including signage, payment terminals, gates (if applicable), and mechanical parts. Look for signs of wear and tear, damage, or malfunction. Early detection allows for timely repairs, preventing bigger problems and disruptions.
- **Battery Backups:** Ensure backup batteries for critical components like sensors and emergency lights are operational. Develop a replacement schedule based on their lifespan to prevent outages. Consider smart batteries that monitor their health and provide pre-emptive warnings.
- **Data Analytics:** Leverage data! By systematically analyzing system data (sensor readings, error logs, usage patterns), you can identify potential issues before they escalate. This proactive approach, known as predictive maintenance, helps prevent downtime and repairs, saving time and money.

Corrective Maintenance: Addressing Issues Promptly

- **Swift Repair:** When malfunctions or breakdowns occur, address them promptly to minimize disruption and ensure system integrity. Maintain a stock of essential spare parts for common repairs to expedite the process. This minimizes downtime and gets the system back to normal operation quickly.
- **Emergency Preparedness:** Be prepared for emergencies! Establish clear procedures for handling power outages, system failures, or malfunctioning equipment. Regularly test these protocols to ensure a coordinated and effective response. This includes designating roles, having a communication plan, and outlining troubleshooting and restoration steps.

Additional Considerations:

- **Maintenance Contracts:** Consider outsourcing maintenance to a qualified service provider specializing in IoT systems. They offer expertise, resources, and ensure adherence to best practices.
- **Documentation:** Maintain clear and up-to-date documentation on the system, including user manuals, maintenance schedules, troubleshooting guides, and warranty information. This facilitates efficient troubleshooting and repairs.
- **Training:** Provide training to personnel responsible for system operation and maintenance. This ensures they understand the system's functionality, can identify potential issues, and perform basic troubleshooting steps.

Benefits of Maintenance:

For Users:

- **Seamless Parking Experience:** Regular maintenance minimizes breakdowns and ensures the system functions as intended. Users can find available spots quickly and pay for parking efficiently, avoiding frustration caused by malfunctions or outages.
- **Accurate Information:** Calibrated sensors provide precise real-time data on parking availability. This reduces the time users spend searching for spots, leading to a more convenient experience.

- **Reduced Inconvenience:** A well-maintained system minimizes technical glitches or outages that can disrupt the parking process for users. They can expect a smooth experience from entry to payment.

For Operators:

- **Cost-Effectiveness:** Preventive maintenance catches minor issues before they snowball into expensive repairs. Replacing parts on a schedule avoids costly downtime due to unexpected failures. This proactive approach significantly reduces operational costs in the long run.
- **Extended System Life:** Regular maintenance promotes the longevity of the system's components, maximizing its lifespan and return on investment. By addressing minor problems promptly and preventing major breakdowns, operators can get the most value out of their smart parking system investment.
- **Improved Efficiency:** A well-maintained system operates more efficiently. This can translate to lower energy consumption (e.g., from sensor optimization) and reduced maintenance costs over time.

For the Environment:

- **Reduced Emissions:** By minimizing the time drivers spend searching for parking, traffic congestion and associated emissions are curbed. This contributes to a cleaner environment, especially in urban areas with high traffic volumes.
- **Data-Driven Sustainability:** Maintenance data can be used to identify areas for improvement, like optimizing lighting or energy usage in parking facilities. This can lead to a more sustainable operation with a lower environmental footprint.

15. Limitation and future enhancement

1. **Limited Accuracy:** Existing sensor technologies may have limitations in accurately detecting vehicle presence and occupancy, leading to occasional inaccuracies in parking availability information.

2. **High Initial Cost:** The implementation of an SPS can involve significant upfront costs for hardware installation, software development, and infrastructure setup, which may be a barrier for widespread adoption, especially in smaller municipalities or businesses.

3. **Dependency on Connectivity:** SPS heavily relies on network connectivity for data transmission between sensors, gateways, and the central management system. Connectivity issues or network outages can disrupt system operation and affect real-time parking information availability.

4. **Scalability Challenges:** Scaling up an SPS to accommodate a larger number of parking spaces or expanding to cover a wider area can pose logistical challenges and require additional investments in hardware, software, and infrastructure.

5. **Limited Integration with Navigation Systems:** Integration with navigation systems to provide real-time parking availability information to drivers is still limited in many areas. Enhancements in this area can improve user experience and help reduce traffic congestion.

6. **Maintenance and Upkeep:** SPS require regular maintenance to ensure proper functioning of sensors, cameras, and other hardware components.

This ongoing maintenance effort can be resource-intensive and may require skilled personnel.

7. **Data Privacy and Security Concerns:** Collecting and storing data about parking behavior and vehicle movements raise privacy concerns. Additionally, securing the SPS against cyber threats and unauthorized access is crucial to protect sensitive information.

Future enhancements to Smart Parking Systems in IoT could include:

1. **Advanced Sensor Technologies:** Integration of more advanced sensor technologies such as LiDAR or computer vision for improved accuracy in vehicle detection and occupancy sensing.

2. **AI and Machine Learning:** Utilizing AI and machine learning algorithms to analyze historical parking data, predict parking demand, and optimize parking space allocation in real-time.

3. **Edge Computing:** Leveraging edge computing capabilities to process data closer to the source (e.g., sensors), reducing latency and dependency on cloud services, especially in environments with limited connectivity.

4. **Blockchain for Security:** Implementing blockchain technology to enhance security and data integrity, ensuring tamper-proof records of parking transactions and enhancing trust among stakeholders.

5. **Enhanced User Interfaces:** Developing user-friendly mobile applications and web interfaces with intuitive navigation and real-time updates on parking availability, directions to available spaces, and payment options.

6. **Environmental Sustainability:** Integrating environmental sensors into SPS to monitor air quality, noise levels, and carbon emissions, enabling cities to implement sustainable urban planning strategies.

7. **Integration with Smart City Initiatives:** Aligning SPS with broader smart city initiatives, such as intelligent transportation systems, urban mobility planning, and sustainability efforts, to create synergies and maximize societal benefits.

16.Bibliography

- Mehala Chandran et al. (2019). An IoT Based Smart Parking System. Journal of Physics: Conference Series, 5.

- ElakyaR, Juhi Seth, Pola Ashritha and R Namith, “Smart Parking System using IoT,” International Journal of Engineering and Advanced Technology, vol. 9, issue 1, pp. 6091-95, 2019.
- Chethana.N.Gowda, Aishwarya.C.S, Sruthi Boggarapu, Mrudulla P Y, Lakshmi H. R., “A Survey on Smart City Application: An IoT Based Car Parking System”, International Journal of Creative Research Thoughts (IJCRT), vol. 6, issue 2, pp. 1057-62, 2018.
- Basavaraj Chougula, Arun Tigadi, Sushant Jadhav and Gujanatti Rudrappa, “Automatic Smart Parking and Reservation System Using IoT”, Bioscience, Biotechnology Research Communication, vol. 13, issue 13, pp. 107-113, 2020.Gautam, S. (2018, august 24). https://blog.getmyparking.com/2018/08/24/a-history-of-parking-garages/. Retrieved april 30, 2021, from https://blog.getmyparking.com/2018/08/24/a-history-of-parking-garages/

- Lueth, K. L. (2014, december 19). https://iot-analytics.com/internet-of-things-definition/. Retrieved april 30, 2021, from https://iot-analytics.com/: https://iot-analytics.com/internet-of-things-definition/