Project with Arduino: Automatic parking

1. The Team

For this project are involved 4 students from different degree courses:

- Carlo Barone from Control Systems engineering
- Cristian Voltan from Control Systems engineering
- Federico Ambrosi from Aerospace Engineering
- Giovanni Bortolotto from Electronic Engineering

2. Objectives

The main objective of the project is to improve in the students the knowledge about the functioning and the possible applications of servo motors and ultrasonic distance sensors. Moreover, the team gets practice with the control of electric actuators in a possible complex scenario.

The second objective is then to realize a simplified version of a fully automated car parking, where a central computational unit manages the entrance and exit of cars up to reaching the full capacity of the park. This is reached through the actuation of gates, the feedback of position sensors, and the output information on a digital display.

3. Components of the project

Project Components

The components utilized in the final project are listed below:

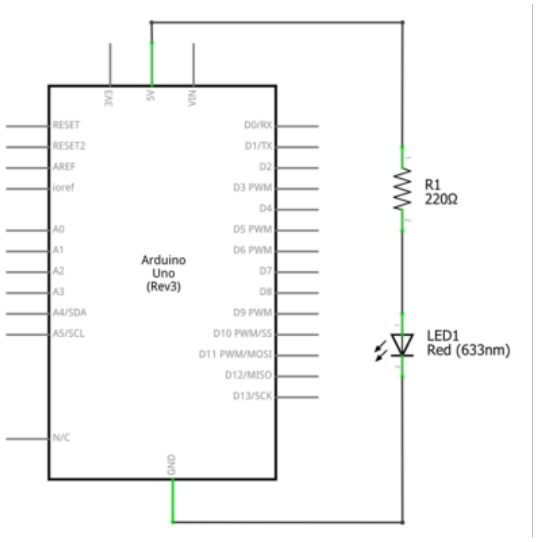
- 1x Red LED
- 1x Green LED
- 2x Yellow LEDs
- 4x Resistors
- 2x Servo motors
- 2x Ultrasonic sensors (HC-SR04)
- 1x Arduino Uno board
- 1x Breadboard
- Assorted jumper wires
- 1x LCD display

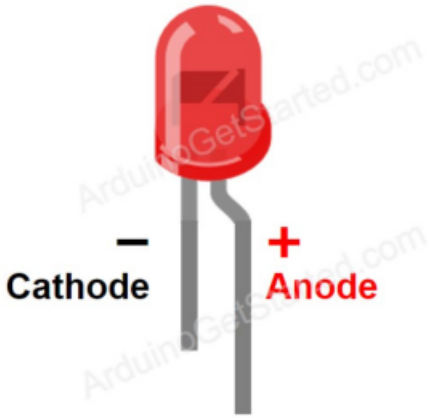
Component Descriptions

In the following section, we provide a brief description of the principal electronic components utilized in this system.

LED

An LED is a semiconductor that emits light when current flows through it. It is polarized, meaning it works only if connected correctly. It has two legs: the Anode (+), which is the longer leg connected to the power source (Arduino pin) through a resistor, and the Cathode (-), the shorter leg connected to Ground (GND). A way to cennect the LED with the arduino and a figure of LED with its terminals is show below:





ULTRASONIC SENSOR

For our automated parking system, we use the HC-SR04 ultrasonic sensor. This component allows the system to detect the presence of a vehicle and measure the exact distance to obstacles.

Function

The sensor operates on the sonar principle. It consists of two main components resembling small speakers: the **Transmitter** (Trigger), which sends out a burst of high-frequency ultrasonic waves, and the **Receiver** (Echo), which detects the returning signal. The measurement cycle begins when the Arduino activates the Trigger to fire a sound wave; this wave travels through the air, hits an object like a car, and bounces back. Finally, the reflected wave is captured by the Echo pin, allowing the system to calculate the distance based on the time taken.

Pin Connection

It has 4 pins to connect to Arduino board:

1. **VCC** (5 V)

2. **GND** (0 V)
3. **Trig** (Transmitter)
4. **Echo** (Receiver)

Calculating the distance

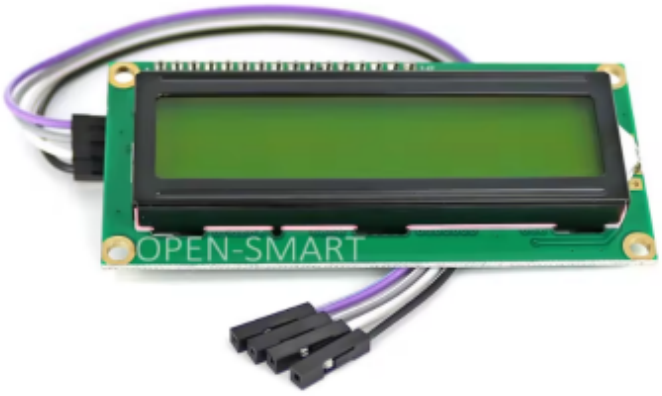
Arduino measures the Time of Flight**,** the time it takes for the sound to travel to the object and back. Since we know the speed of sound in air (approx. 343 m/s), we calculate the distance (D) using this formula:

$$D = \frac{Time \cdot Speed\ of\ Sound}{2}$$

We divide by 2 because the sound travels a round trip (to the object and back to the sensor).

LCD DISPLAY

The LCD (Liquid Crystal Display) is a screen used to visualize data. We use a 16x2 model, meaning it can display 2 rows of 16 characters each. Inside, liquid crystals align to block or allow light from the backlight to pass through, forming letters and numbers. In our project, it connects to Arduino (via an I2C module to use only 4 wires) and displays the real-time count of available parking spots.



Pin connection:

1. **GND (Ground):** Connects to the ground (0V) to complete the electrical circuit.
2. **VCC (Power):** Connects to the 5V power supply from the Arduino to turn on the display and backlight.
3. **SDA (Serial Data):** The data line used to send information (characters and numbers) from the Arduino to the screen.
4. **SCL (Serial Clock):** The clock line that synchronizes the timing of data transmission between the Arduino and the LCD.

Servomotor

The Servomotor is a precise actuator used to control angular position. Unlike standard motors that spin continuously, a servo rotates to a specific angle (usually between 0° and 180°) and holds that position. Inside, it contains a small DC motor, a gearbox for high torque, and a control circuit. In our project, it acts as the automatic barrier, lifting up to let cars enter and lowering to block the entrance.

Pin connection:

1. **Signal (Orange):** Receives the PWM (Pulse Width Modulation) control signal from an Arduino digital pin to tell the motor which angle to move to.
2. **VCC (Red):** Connects to the 5V power supply.
3. **GND (Brown or Black):** Connects to Ground (0V).

4. Project Design

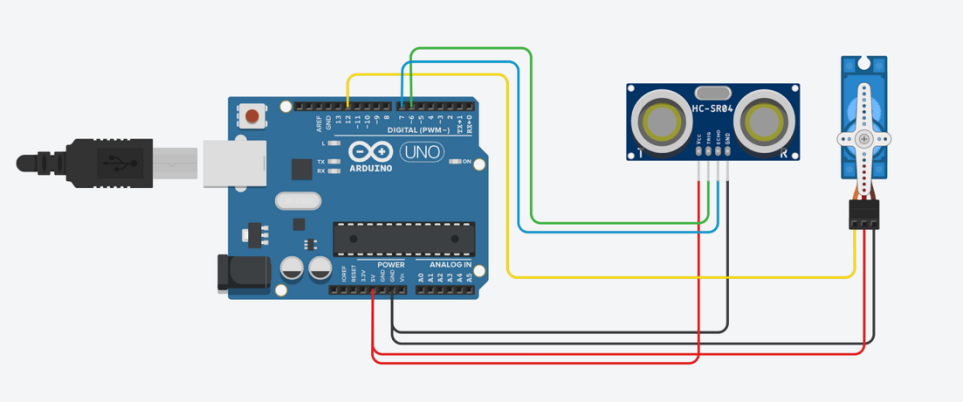
The design of the project is carried out in two consecutive phases: first just an understanding of the electronic components, and the coupling of all of them. Then, once the whole system with all its complexities works, the structural design phase begins.

4.1 The circuit

The idea of the automatic park is initially limited to a gate that opens when a vehicle arrives in front of a distance sensor and then closes when the vehicle is not detected anymore. This first configuration requires

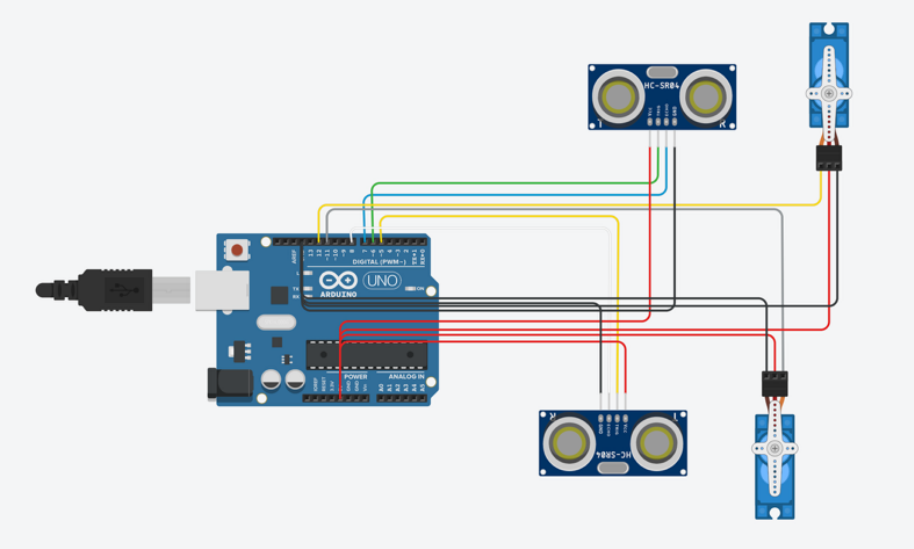
just one servo motor for the gate and one distance sensor. With a closed loop the output distance from the sensor is measured and only when the condition of a detected distance below a certain threshold is satisfied, the servo motor is actuated.

The first simple configuration can be represented with the following circuit:



In a second phase the team decide to increase a bit the complexity to the system, by adding another gate. In this way one represents the entrance and the other represents the exit of the parking. Of course, both gates are coupled with their own distance sensor and can work also simultaneously.

Here it is a schematic representation of the second configuration:

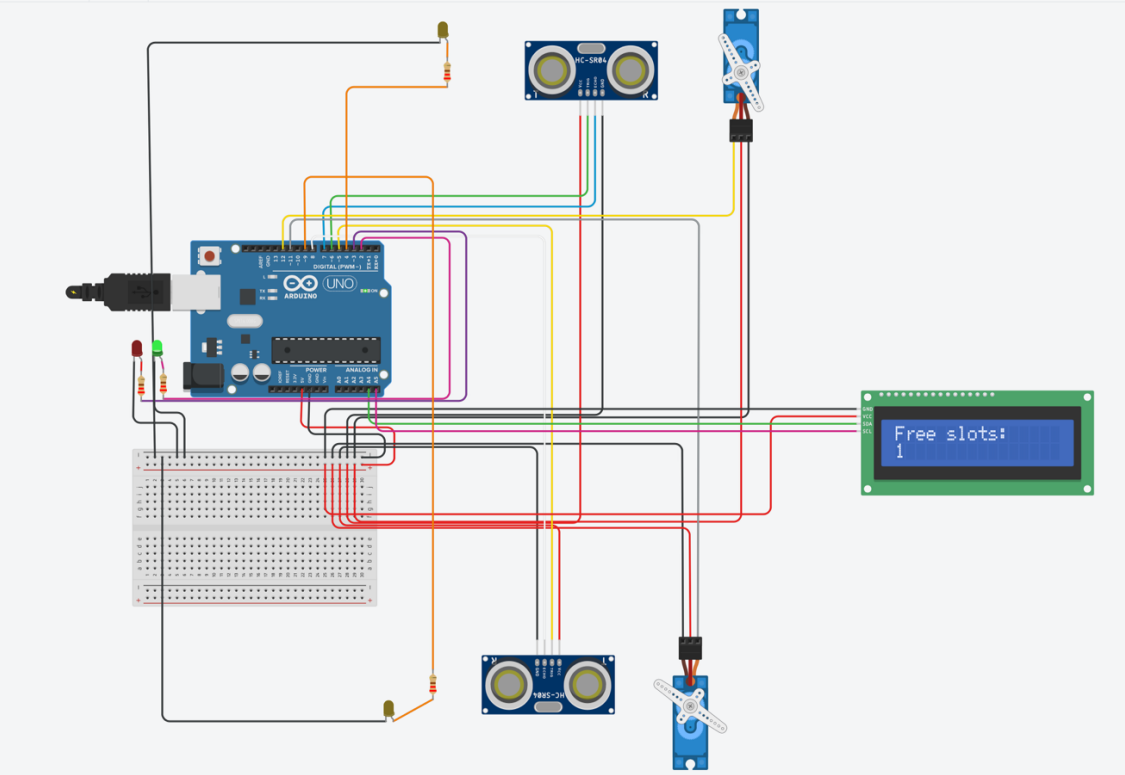


With the second configuration, the idea of an automatic park starts to have a shape. To improve it, the addition of a led-matrix display is needed. The idea is to use it to provide some useful information about the status of the parking, and the number of free slots.

Moreover, to have a more clear and faster visual information, the team decide to add also two led-diodes, like a green-red traffic light. These lights work parallel to the display, with the scope to give information about the available places.

Finally, to increase one last time the complexity and fidelity of the project, another two led are added in the proximity of the gates. The idea is to represent a safety flashing light that warns about the movement of the gate.

The final circuit that the team managed to realize is here represented (a brief description will follow):



In the circuit above, the servo motor and the sensor at the top represent the entrance, while the ones at the bottom represent the exit. In particular, the digital ports 6 and 7 of the Arduino board are dedicated to the entrance sensor and the port 12 is dedicated to the entrance servo motor. Instead, the exit motor and sensor are connected respectively to the digital ports 11 and 5-8.

For what concerning the display, it is connected to the analog ports 4 and 5, while the red-green traffic light led are connected to the digital ports 2 and 3. Finally, the ports 4 and 9 are dedicated to the yellow flashing lights of each gate.

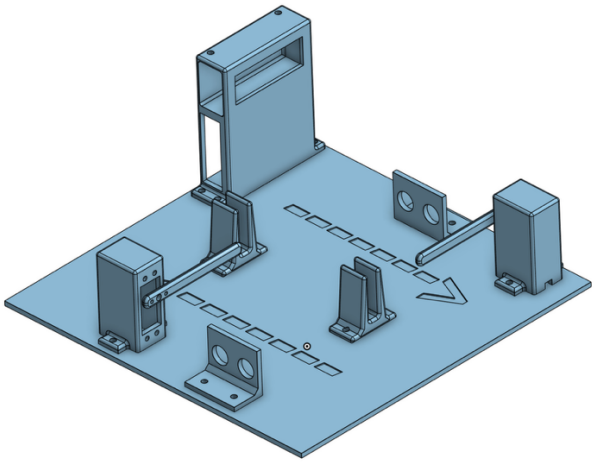
The power for all drives is then supplied directly from the Arduino board, with the 5 volts source. All power connections are managed by a breadboard, that shares positive and ground terminals.

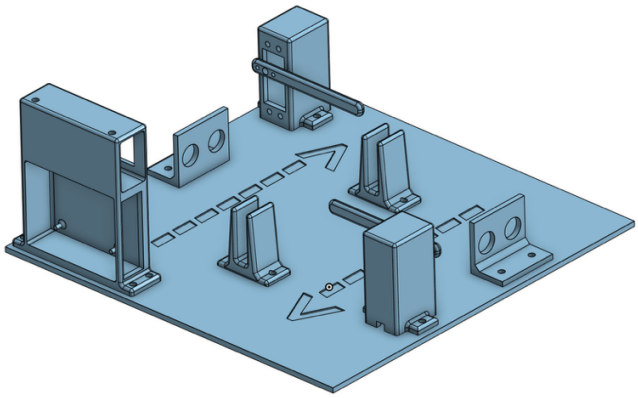
4.2 The structure

Once the circuit is completed and functioning, the team decide to design a structure that could support all the components, and position them in such a way that they could represent a realistic automatic parking.

To do that, different support structures are designed and sketched on a CAD software, to have a better visual overview of all the assembled components. Moreover, in this way the team have the possibility to modify all the dimensions, up to the final structural configuration, that arises parallel to the last circuit configuration.

An overview assembly of the hole project is here represented:





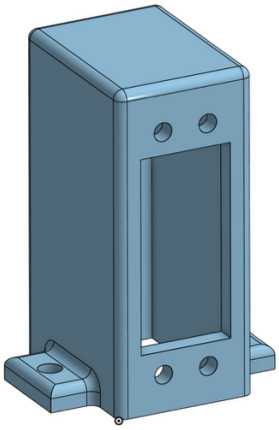
The main structural elements are:

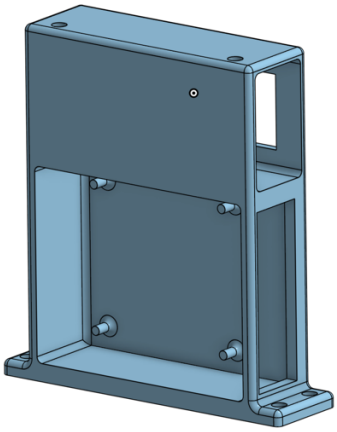
- One base plate 30x30cm
- One displaying unit for the support of the Arduino board, the display and the green-red signal lights
- Two middle vertical supports for the breadboard
- Two supports for the ultrasonic sensors
- Two structural towers for the servo motors and the opening bars

All these components, except from the base plate, are 3d printed in PLA, while the plate is made of wood.

For the mechanical connections, all the pieces are bolted together to the plate with some screws, all passing through 5 mm holes in the plastic elements. The servo motors and the bars are also bolted with screws, while the other electronic elements (Arduino board, breadboard, ultrasonic sensors) are simply mechanical jointed together.

Here are some more detailed images of the two main structural elements:





Code description

Initialization Section

Library Imports

At the beginning of the code, three essential Arduino libraries are included:

- #include <Servo.h>
  This library allows the Arduino to control servo motors. In this project, two servomotors are used to raise and lower the parking barriers at the entrance and exit.
- #include <Wire.h>
  This library enables communication using the I²C protocol. It is required because the LCD display communicates with the Arduino through I²C.
- #include <LiquidCrystal_I2C.h>
  This library provides functions to control an LCD display with an I²C interface. It will be used later to show the number of available parking spaces.

Together, these libraries allow the Arduino to control motors and display information on the LCD.

Servo Objects

- Servo Servo1;
- Servo Servo2;

Two servo objects are created, one for each barrier (entrance and exit).

Ultrasonic Sensor Pins

The parking system uses two ultrasonic proximity sensors to detect cars:

- **Sensor 1 (entrance)**
- #define trigPin1 6
- #define echoPin1 7

trigPin1 sends a pulse and echoPin1 receives the reflected signal to measure distance.

- **Sensor 2 (exit)**
- #define trigPin2 5
- #define echoPin2 8

These pins perform the same function for the sensor at the exit. Each sensor is therefore connected with a trigger pin and an echo pin, which allows distance calculation.

Barrier States (Finite State Machine)

enum State { DOWN, GOING_UP, WAITING, GOING_DOWN };

Here, an **enumeration (enum)** is used to define the possible states for each barrier.
An *enum* is a programming tool that allows you to create your own type with predefined values.
In this case, each barrier can be in one of these 4 states:

- **DOWN** – barrier is fully lowered (closed)

- **GOING_UP** – barrier is currently rising
- **WAITING** – barrier is up and waiting before closing
- **GOING_DOWN** – barrier is currently lowering

Two state variables are then initialized:

- `State state1 = DOWN;`
- `State state2 = DOWN;`

This means that both the entrance and exit barriers start in the **closed** position.

Servo Position Variables

- `int pos1 = 25;`
- `int pos2 = 55;`

The initial positions of the servos are not 0° because the physical installation requires these angles to keep the barriers perfectly horizontal when closed.
Every servo and mechanical setup is slightly different, so these values ensure that the barrier starts correctly aligned.

LEDs and Their States

The system uses four LEDs:

- `#define greenLed 2`
- `#define redLed 3`
- `#define led1 4`
- `#define led2 9`

Usage in this project:

- **greenLed:** indicates that entry is allowed
- **redLed:** indicates that entry is denied (parking full)
- **led1** and **led2:** additional indicators for barrier movement.

Timing Variables

```
unsigned long waitStart1 = 0;
unsigned long waitStart2 = 0;
```

- **waitStart1** and **waitStart2** store the exact moment when each barrier finishes opening.

These two `waitStart` variables are essential because, **each barrier must stay open for 3 seconds before closing again**.
By saving the timestamp when the barrier reaches the fully raised position, the program can calculate how long it has been open and close it automatically after the required time.

Parking Management Variables

```
int capacity = 2;
int carsInside = 1;
int freeSlots = 0;
```

These three variables allow the system to keep track of the parking status:

- **capacity:** total number of available parking spots in the garage

- **carsInside:** number of cars currently inside
- **freeSlots:** number of free spaces

These values are crucial because they determine:

1. Whether the entrance barrier is allowed to open
2. The message shown on the LCD display

By updating these variables each time a car enters or exits, the system can accurately show the number of available spots in real time.

LCD Display Initialization

```
LiquidCrystal_I2C lcd(0x27, 16, 2);
```

This initializes a **16×2 I²C LCD** with address **0x27.**

The LCD will later show messages such as:

- "Free slots:"
- The number of available parking spaces

This gives the user visual feedback on the parking status.

Sensor Sensitivity and Movement Delay

```
int tolleranza = 10;
int delayMov = 50;
unsigned long lastMove1 = 0;
unsigned long lastMove2 = 0;
```

- **tolleranza** defines the distance (in centimeters) below which the system considers that an object, such as a car, is present near the sensor.
- **delayMov** sets the delay between increments of servo movement.
- **lastMove1** and **lastMove2** store timestamps to control how often each servo updates its position.

These variables ensure that:

- Sensor readings are stable and do not produce false detections.
- Servo motors move gradually and smoothly rather than jumping to a position instantly.

Ultrasonic sensors do not always provide perfectly accurate readings.
Sometimes they return:

- very short distances even when nothing is there
- values that jump suddenly
- a single measurement that is completely wrong

This happens because ultrasonic sensors are sensitive to noise, reflections, and environmental conditions.
If we trusted every single reading, even one incorrect value would be enough to open the barrier unintentionally.
This would cause serious problems in the parking system, such as:

- the barrier going up even when no car is present
- wrong increases or decreases in the number of cars inside
- inconsistent system behavior

To prevent this, the code uses two stability-check functions:

```
bool stableDetect1(long cm) { ... }
bool stableDetect2(long cm) { ... }
```

These functions act like a **filter**.

How they work

Instead of accepting a single sensor reading, each function:

1. Checks if the value is within the valid detection range (less than the tolerance and greater than zero)
2. Counts how many *consecutive* valid readings it receives
3. Only returns `true` if the sensor confirms the presence of an object at least **two times in a row**

In other words, the system requires **confirmation** before deciding that a car is actually present.

This technique:

- Eliminates false positives
- Reduces the impact of sudden noise
- Ensures that the barrier only reacts to stable, repeated detections
- Makes the entire parking system more reliable

Object Detection Helper

```
bool objectDetected(long cm) {
  if (cm == 0) return false;
  return (cm < tolleranza);
}
```

This is a simpler detection rule used in some parts of the program.
However, because it can still be influenced by noise, the more advanced functions `stableDetect1` and `stableDetect2` are used when higher reliability is required.

LCD Update Function

```
void updateLCD() { ... }
```

This function updates the content of the LCD display by:

1. Clearing the screen
2. Printing `"Free slots:"` on the first line
3. Printing the value of `freeSlots` on the second line

Its purpose is to constantly provide the user with real-time information about how many parking spaces are available.

Setup

The setup() function initializes serial communication for debugging. Then, it configures the pins of the two ultrasonic sensors and all LEDs as inputs or outputs.

The two servomotors controlling the entrance and exit barriers are attached to pins 12 and 11, and each servo is set to its initial position (pos1 and pos2), ensuring both barriers start correctly aligned.

Next, the system calculates the initial number of free parking spots based on the total capacity and the number of cars currently inside.
The LCD display is initialized, its backlight is turned on, and updateLCD() is called to show the number of available spaces on the screen.

Loop

Sensor Reading and LED Status

At the beginning of the loop, the program measures the distance from both ultrasonic sensors.
Then it checks whether the parking still has free spots.

- If **freeSlots > 0**, the green LED turns on and the red LED turns off → cars are allowed to enter.
- If **freeSlots = 0**, the red LED turns on, and the green LED turns off → parking is full.

This gives immediate visual feedback to drivers.

Barrier 1 Logic (Finite State Machine)

Barrier 1 (the entrance barrier) is controlled using a **finite state machine** with four states:

1. **DOWN**
2. **GOING_UP**
3. **WAITING**
4. **GOING_DOWN**

Each state describes a phase in the barrier's movement, and the program switches between states depending on sensor readings and timing conditions.

1. State: DOWN

This is the default state: the barrier is fully lowered.

While in this state, the system checks two conditions:

- A car is detected near the sensor (stableDetect1(cm1)),
- There is at least one free parking space (freeSlots > 0)

If both conditions are true, the barrier is allowed to open, and the system switches to **GOING_UP**.

2. State: GOING_UP

In this state, the barrier gradually rises. Instead of moving the servo instantly, the system increases the angle step by step (every delayMov ms).
This ensures smoother movement and avoids jerky or unnatural servo behavior.

During the upward movement:

- The barrier angle $pos1$ increases
- LED1 blinks to signal movement
- When the barrier reaches the fully raised position ($pos1 >= 100$), movement stops
- The system switches to **WAITING**

The blinking LED provides visual confirmation that the barrier is currently moving.

3. State: WAITING

Now the barrier is fully up.
The system waits until the car passes beyond the sensor.

This is monitored by checking if the measured distance is **greater than the tolerance**, meaning the object (car) is no longer under the sensor.
When the car moves away, the system records the current time ($waitStart1 = now$), and switches to **GOING_DOWN**.

This ensures the barrier does not close while the car is still underneath.

4. State: GOING_DOWN

After the barrier has been up for a certain amount of time, it begins to close.

The barrier only starts moving down after **3000 ms (3 seconds)** since the car left the sensor area.
This prevents the barrier from closing too early and gives the car time to pass safely.

During the downward movement:

- The angle decreases gradually ($pos1 -= 3$)
- LED1 blinks again
- When the barrier reaches its closed position ($pos1 <= 25$), the state resets to **DOWN**

At this moment, an important update occurs:

- The car is counted as **inside** the parking ($carsInside++$)
- The number of available spots is recalculated
- The LCD is updated to show the new number of free slots

This ensures that the parking count remains accurate each time a car enters.

Just like the entrance barrier, the exit barrier is also controlled using a **finite state machine** with the same four states:
DOWN, GOING_UP, WAITING, GOING_DOWN.
However, the logic is slightly different because this barrier is used when cars leave the parking area.

Diagnostic Serial Output

At the end of the loop, the program sends detailed diagnostic information to the Serial Monitor:

- Sensor distances (cm1 and cm2)
- Current servo positions (pos1 and pos2)

- State numbers for both barriers (state1 and state2)

This is useful for:

- Debugging
- Observing real-time barrier behavior
- Verifying that sensors and servos respond correctly

The serial output acts as a valuable monitoring tool during testing and calibration.

In the next pages the code will be showed:

```
#include <Servo.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

Servo Servo1;
Servo Servo2;

// Sensore 1
#define trigPin1 6
#define echoPin1 7

// Sensore 2
#define trigPin2 5
#define echoPin2 8

// Stati sbarra
enum State { DOWN, GOING_UP, WAITING, GOING_DOWN };

State state1 = DOWN;
State state2 = DOWN;

int pos1 = 25;
int pos2 = 55;
bool stateled2= false;
bool stateled1= false;

// Led
#define greenLed 2
#define redLed 3
#define led1 4
#define led2 9

int accensione;
int spegnimento;

unsigned long waitStart1 = 0;
unsigned long waitStart2 = 0;
```

```
// =============== PARCHEGGIO ===============
int capacity = 2;       // totale posti disponibili
int carsInside = 1;     // auto attualmente nel parcheggio
int freeSlots = 0;      // posti liberi
// ==========================================

LiquidCrystal_I2C lcd(0x27, 16, 2);  // LCD I2C 16x2

int tolleranza = 10;

int delayMov = 50;    // tempo tra un movimento e l'altro
unsigned long lastMove1 = 0;
unsigned long lastMove2 = 0;

long readUltrasonic(int trig, int echo) {
  digitalWrite(trig, LOW);
  delayMicroseconds(5);
  digitalWrite(trig, HIGH);
  delayMicroseconds(10);
  digitalWrite(trig, LOW);
  long duration = pulseIn(echo, HIGH, 25000); // timeout di sicurezza
  return duration / 58;
}

// Caso di lettura non corretta del sensore
bool stableDetect1(long cm) {
  static int count1 = 0;
  if (cm > 0 && cm < tolleranza) count1++;
  else count1 = 0;
  return count1 >= 2;
}

bool stableDetect2(long cm) {
  static int count2 = 0;
  if (cm > 0 && cm < tolleranza) count2++;
  else count2 = 0;
  return count2 >= 2;
}
```

```
bool objectDetected(long cm) {
  if (cm == 0) return false;   // 0 significa nessuna eco → nessun oggetto
  return (cm < tolleranza);
}
void updateLCD() {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Free slots:");
  lcd.setCursor(0, 1);
  lcd.print(freeSlots);
}

void setup() {
  Serial.begin(9600);

  pinMode(trigPin1, OUTPUT);
  pinMode(echoPin1, INPUT);
  pinMode(trigPin2, OUTPUT);
  pinMode(echoPin2, INPUT);
  pinMode(greenLed, OUTPUT);
  pinMode(redLed, OUTPUT);
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);


  Servo1.attach(12);
  Servo2.attach(11);
  Servo1.write(pos1);
  Servo2.write(pos2);


  freeSlots = capacity - carsInside; // inizializza
  lcd.init();      // Inizializza il display
  lcd.backlight(); // Accende la retroilluminazione
  updateLCD();

}
```

```
      }
      Servo1.write(pos1);
    }
    break;

  case WAITING:
    if (cm1 ⩾ tolleranza) {
      waitStart1 = now;
      state1 = GOING_DOWN;
    }
    break;

  case GOING_DOWN:
    if (now - waitStart1 > 3000) {
      lastMove1 = now;
      pos1 -= 3;
      digitalWrite(led1, stateled1);
      stateled1 = !stateled1;
      if (pos1 ⩽ 25) {
        pos1 = 25;
        state1 = DOWN;
        digitalWrite(led1, LOW);
        carsInside++;
        freeSlots = capacity - carsInside;
        updateLCD();
      }
      Servo1.write(pos1);
    }
    break;
}

// ----------------------- SBARRA 2 -------------------------
switch (state2) {

  case DOWN:
    if (stableDetect2(cm2) && freeSlots < 2) {
        state2 = GOING_UP;
    }
```

```
    break;

  case GOING_UP:
    if (now - lastMove2 > delayMov) {
      lastMove2 = now;
      pos2 += 3;
      digitalWrite(led2, stateled2);
      stateled2 = !stateled2;
      if (pos2 ⩾ 155) {
        pos2 = 155;
        digitalWrite(led2, LOW);
        state2 = WAITING;

      }
      Servo2.write(pos2);
    }
    break;

  case WAITING:
    if (cm2 ⩾ tolleranza) {
      waitStart2 = now;
      state2 = GOING_DOWN;
    }
    break;

  case GOING_DOWN:
    if (now - waitStart2 > 3000) {
      lastMove2 = now;
      pos2 -= 3;
      digitalWrite(led2, stateled2);
      stateled2 = !stateled2;
      if (pos2 ⩽ 55) {
        pos2 = 55;
        digitalWrite(led2, LOW);
        state2 = DOWN;
        carsInside--;
        freeSlots = capacity - carsInside;
        updateLCD();
```

```
        }
        Servo2.write(pos2);
      }
      break;
  }\

  // Stampa diagnostica
  Serial.print("Cm1: "); Serial.print(cm1);
  Serial.print("\t Pos1: "); Serial.print(pos1);
  Serial.print("\t State1: "); Serial.print(state1);

  Serial.print("   |   Cm2: "); Serial.print(cm2);
  Serial.print("\t Pos2: "); Serial.print(pos2);
  Serial.print("\t State2: "); Serial.println(state2);
}
```