Advanced Embedded Systems and Designs

Project Report

Project report submitted in partial fulfillment
of the requirements for the degree of

Bachelor of Technology
in
Electronics and Communication Engineering

Submitted by:

Barnabh Chandra Goswami
19UEC161

Course Instructor:
Dr. Deepak Nair

Department of Electronics and Communication Engineering
The LNM Institute of Information Technology, Jaipur

December 08, 2022

Table of Contents

List of Figures

Items required for this assignment

The apparatus which are used in this assignment are:

1. Arduino Uno Microcontroller
2. PIR Sensor
3. IR Proximity Sensor
4. DC Servo Motor
5. External DC Power Supply
6. Jumper Wires
7. 1 Switch
8. 1 LED
9. Breadboard
10. Resistors (330 Ω to 1 kΩ)

Chapter 1:

Question and Approach

1.1 Introduction

This project is about how:

"I created a Toll Booth system for one vehicle only which closes its the barricade on vehicle detection and on exiting from parking lot it counts the total number of vehicles parked there one at a time. These all things are implemented using the **FreeRTOS Library**"

Overall working of this will be discussed in further sections, but we can say in short that we will use IR (Infra-Red) Proximity Sensor for sensing the vehicle then operate the barricade using Servo Motor and at last after opening the barricade we will use PIR (Passive Infra-Red) Sensor to count the number of vehicles parked at that particular spot.

But let's discuss about some basic components and concepts use in this project:

1.2 Arduino Uno

Arduino Uno is a microcontroller board having a 8-bit ATmega328P microcontroller together with ATmega328P, it consists of different components like crystal oscillator, voltage regulator, serial communication etc, to support the microcontroller. Arduino Uno has 14 digital i/p or o/p pins (out of which 6 can be used as Pulse Wave Modulation outputs), 6 analog i/p pins, a USB connection, an influencing power barrel jack, an ICSP header and a push reset button.

Types of pins in Arduino Uno:

- Vin (Input Voltage)
- 5V supply
- 3.3V supply
- GND (Ground)
- Analog Pins (A_0 to A_5)
- Digital Pins (0 to 13)
- Serial Pins (1 and 0 as transmitter and receiver)
- External Interrupt Pins (2 and 3)
- PWM (Pulse Wave Modulation) Pins (3,5,6,9,10 and 11)

- SPI (Serial Peripheral Interface pin) Pins (10, 11, 12 and 13)
- LED Pin (13)
- AREF (analog reference pin) Pin

Figure 1: Diagram of the Arduino Uno. (Taken from [5])

1.3 Servo Motor

A servo motor is a type of motor that can rotate with pinpoint accuracy. This type of motor typically includes a control circuit that offers feedback on the current position of the motor shaft; this feedback allows servo motors to rotate with great precision. A servo motor is used when you wish to spin an object at a specified angle or distance. It is simply a motor that is controlled by a servo system.

Figure 2: Pin diagram of Servo Motor SG-90. (Taken from [8])

If the motor is powered by a DC power supply, it is referred to as a DC servo motor; if it is driven by an AC power supply, it is referred to as an AC servo motor. This lesson will solely cover the operation of a DC servo motor.

Figure 3: Diagram of SG-90 Servo Motor. (Taken from [7])

Aside from these basic categories, there are numerous different types of servo motors based on gear arrangement and operational characteristics. A servo motor is typically equipped with a gear arrangement that enables us to obtain a very high torque servo motor in tiny and lightweight designs. Because of these characteristics, they are utilised in a variety of applications like as toy cars, RC helicopters and planes, robotics, and so on.

1.4 IR (Infra-Red) Sensor

An infrared sensor is a type of electrical gadget that produces light in order to detect certain features of its surroundings. An IR sensor can detect motion as well as measure the heat of an item. These sensors merely measure infrared radiation rather than emitting it, which is known as a passive IR sensor. Typically, all objects in the infrared range emit some type of thermal radiation.

The operation of an infrared sensor is similar to that of an object detection sensor. This sensor contains an IR LED and an IR Photodiode. An IR LED is a type of transmitter that produces infrared radiations. This LED appears to be a regular LED, and the radiation it emits is not

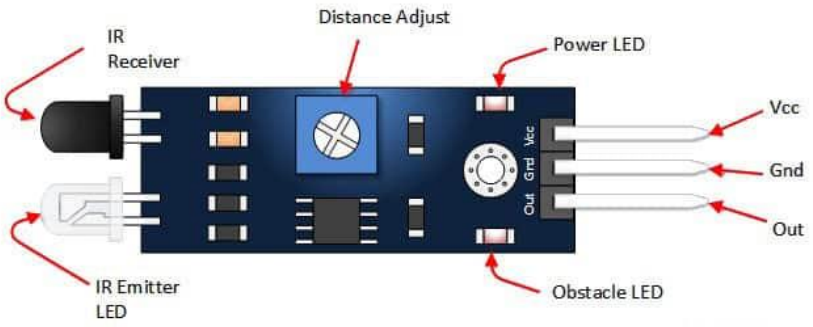

Figure 4: Schematic Diagram of IR Sensor. (Taken from [11])

visible to the naked eye. Infrared receivers detect radiation primarily through the use of an infrared transmitter. Infrared receivers are accessible as photodiodes. IR Photodiodes differ from conventional photodiodes in that they only sense IR radiation. There are various types of infrared receivers based on voltage, wavelength, packaging, and so on.

Figure 5: Diagram of IR Sensor. (Taken from [10])

1.5 PIR (Passive Infra-Red) Sensor

PIR sensors detect motion and are virtually always used to determine whether a human has moved into or out of the sensor's range. They are compact, cheap, low-power, simple to use, and do not wear out. As a result, they are frequently found in appliances and gadgets used in

households and companies. They are also known as PIR, "Passive Infrared," "Pyroelectric," or "IR motion" sensors.

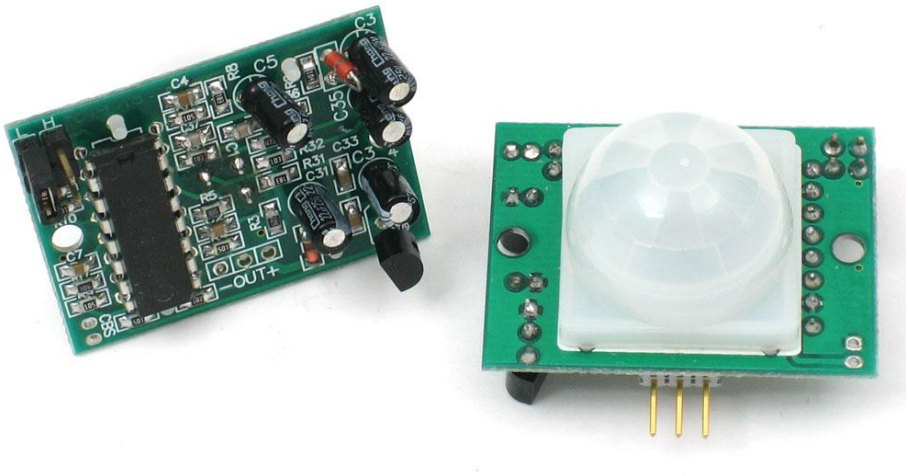

Figure 6: Diagram of PIR Sensor. (Taken from [12])

A PIR basically consists of a pyroelectric sensor (shown below as a round metal can with a rectangular crystal in the centre) that can detect infrared radiation. All things emit a small amount of radiation, and hotter things emit more radiation. The motion detector sensor is actually split in two. This is because we are trying to detect motion (change) instead of averaging IR values. The two halves are wired to cancel each other out. If one half sees more or less IR radiation than the other, the output will swing up and down.

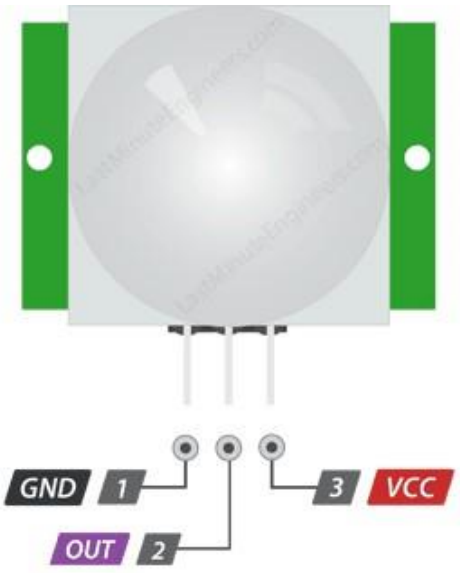

Figure 7: Schematic Diagram of PIR Sensor. (Taken from [13])

1.6 Queues in FreeRTOS

A queue is a data structure that supports data exchange between different tasks or between tasks and interrupts. It has a finite number of elements (defined at initialization) and operates in FIFO mode.

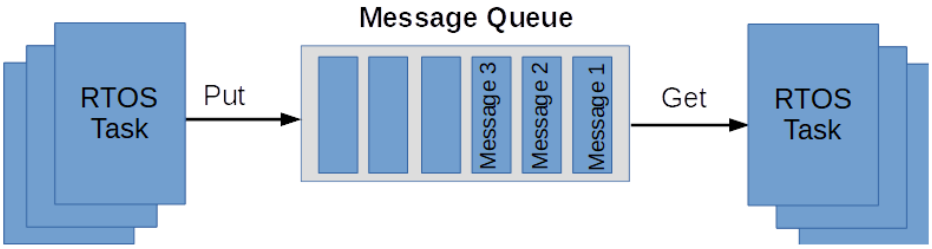

Figure 8: Working of queue. (Taken from [14])

1.7 Functions used in Queues in FreeRTOS

1.7.1 xQueueCreate

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength,
                           UBaseType_t uxItemSize;
```

Creates a new queue and returns a handle by which the queue can be referenced.

Each queue requires RAM that is used to hold the queue state, and to hold the items that are contained in the queue (the queue storage area). If a queue is created using xQueueCreate() then the required RAM is automatically allocated from the FreeRTOS heap. If a queue is created using xQueueCreateStatic() then the RAM is provided by the application writer, which results in a greater number of parameters, but allows the RAM to be statically allocated at compile time.

Parameters:

- uxQueueLength: The maximum number of items the queue can hold at any one time.
- uxItemSize: The size, in bytes, required to hold each item in the queue. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each queued item. Each item in the queue must be the same size.

Return values:

- If the queue is created successfully then a handle to the created queue is returned. If the memory required to create the queue could not be allocated then NULL is returned.

For more details you can refer to [15].

1.7.2 xQueueSend

```
BaseType_t xQueueSend(QueueHandle_t xQueue, const void *
        pvItemToQueue, TickType_t xTicksToWait);
```

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR() for an alternative which may be used in an ISR.

Parameters:

- xQueue: The handle to the queue on which the item is to be posted.
- pvItemToQueue: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- xTicksToWait: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if the queue is full and xTicksToWait is set to 0. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Return values:

- pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

For more details you can refer to [16].

1.7.3 xQueueReceive

```
BaseType_t xQueueReceive(QueueHandle_t xQueue, void *pvBuffer,
                    TickType_t xTicksToWait);
```

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Parameters:

- xQueue: The handle to the queue from which the item is to be received.
- pvBuffer: Pointer to the buffer into which the received item will be copied.
- xTicksToWait: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. Setting xTicksToWait to 0 will cause the function to return immediately if the queue is empty. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Return values:

- pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

For more details you can refer to [17].

1.8 Binary Semaphores in FreeRTOS

Binary semaphores are utilised for mutual exclusion as well as synchronisation. These and mutexes are quite similar yet differ slightly: Mutexes have a mechanism for prioritising inheritance, but binary semaphores do not.

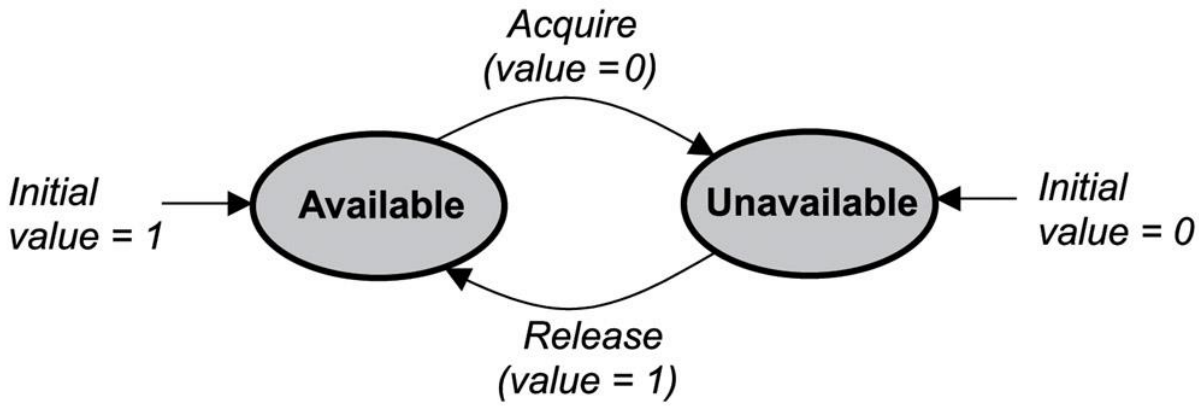

Figure 9: Working of the Binary Semaphore. (Taken from [6])

As a result, binary semaphores are a superior choice for establishing synchronisation (between tasks or between tasks and an interrupt), while mutexes are a better choice for implementing simple mutual exclusion. The description of mutexes as mutual exclusion mechanisms applies equally to binary semaphores.

Consider a binary semaphore to be a single-item queue. As a result, the queue can only be empty or filled (hence binary). Tasks and interruptions that use the queue don't care what's in

it; they just want to know if it's empty or full. This method can be used to synchronise (say) a task with an interrupt.

1.9 Functions to call and use Binary Semaphores

1.4.1 xSemaphoreCreateBinary

```
SemaphoreHandle_t xSemaphoreCreateBinary (void);
```

Creates a binary semaphore and returns a handle that can be used to refer to the semaphore. For this RTOS API function to be available, configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h or left undefined (in which case it will default to 1).

Return values:

- NULL: The semaphore could not be created because there was insufficient FreeRTOS heap available.
- Any other value: The semaphore was created successfully. The returned value is a handle by which the semaphore can be referenced.

For more details refer to [1] and [2].

1.4.2 xSemaphoreGive

```
xSemaphoreGive (SemaphoreHandle_t xSemaphore);
```

The semaphore must have previously been created with a call to xSemaphoreCreateBinary (), so when I use xSemaphoreGive () it means that our task gives the mutex back to the library or to the main terminal, so that other task can requiring the shared resources can access it.

Parameters:

- xSemaphore: A handle to the semaphore being taken - obtained when the semaphore was created.

Return values:

- pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

For more details refer to [4].

1.4.3 xSemaphoreTake

```
xSemaphoreTake (SemaphoreHandle_t xSemaphore, TickType_t
xTicksToWait);
```

The semaphore must have previously been created with a call to xSemaphoreCreateBinary (), so when I use xSemaphoreTake () it means that a particular task is using that semaphore as per the requirement and that time no other task can access it until that task doesn't give back the semaphore (shared resource).

Parameters:

- xSemaphore: A handle to the semaphore being taken - obtained when the semaphore was created.
- xTicksToWait: The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Return values:

- pdTRUE if the semaphore was obtained. pdFALSE if xTicksToWait expired without the semaphore becoming available.

For more details refer to [3].

1.5 Approach (How I started?)

First of all, I started with IR Sensor, where I used it as Proximity Sensor for vehicles which will be parked in a particular spot, after this I made a smart barricade using Servo Motor which is controlled and operated by IR Sensor and button provided by me (User Choice). When the vehicle comes into the Toll Booth area then the barricade will close automatically otherwise it will always be open in the default state.

Also, there is a red LED which will turn ON whenever IR Sensor detects the vehicle and only be turn OFF when the vehicle gets access to pass from Toll, which is done by the user by pressing the button according to his choice.

When the user pressed the button, the barricade will open and vehicle can go away from there, in this process the PIR Sensor also becomes ON so that it can count the number of vehicles parked in that area for a given amount of time which is again decided by user and when the counting is done, the PIR Sensor again goes into the OFF state.

We made PIR Sensor in Arduino Uno is such a way that whenever the button is pressed it turns ON and whenever button is release it turns OFF.

Thus, in the end, our program and circuit will tell us about the number of vehicles come across the Toll Booth and controls the barricade according to the IR Sensor and the user’s choice.

Chapter 2:

Code, Simulation and Results

2.1 Code written in Arduino Uno

```
#include <Arduino_FreeRTOS.h>
#include <Servo.h>
#include <semphr.h>
#include "queue.h"

QueueHandle_t var_queue;
SemaphoreHandle_t xBinarySemaphore;

#define IRSensor 2   // connect IR sensor to Arduino Pin 2
#define PIRSensor 7  // connect PIR sensor to Arduino Pin 7
#define Red_LED 13   // conect Red_LED to Arduino Pin 13
#define ServoPin 3
#define Button 4

Servo MyServo;

int i = 0;

void setup() {

  Serial.begin(9600);
  // put your setup code here, to run once:
  var_queue = xQueueCreate(1, sizeof(unsigned int));
  xBinarySemaphore = xSemaphoreCreateBinary();

  xTaskCreate(Task1, "IR_Sensor", 128, NULL, 3, NULL);
  xTaskCreate(Task2, "ServoMotor", 128, NULL, 2, NULL);
  xTaskCreate(Task3, "PIRSensor", 128, NULL, 1, NULL);

  xSemaphoreGive(xBinarySemaphore);
}

void Task1(void *pvparameters) {

  pinMode(IRSensor, INPUT);  // sensor pin INPUT
  pinMode(Red_LED, OUTPUT);  // Red_Led pin OUTPUT
```

```
  while (1) {

    int statusSensor = digitalRead(IRSensor);

    if (statusSensor == 1) {

      int Status_Send = 0;
      xQueueSend(var_queue, &Status_Send, 2);
      Serial.println("Vehicle is not there");
      vTaskDelay(1);
    }

    else {

      int Status_Send = 1;
      xQueueSend(var_queue, &Status_Send, 2);
      digitalWrite(Red_LED, HIGH);  // Red_LED High
      Serial.println("Vehicle is there");
      Serial.println("Red LED is ON");
      Serial.println("");
      vTaskDelay(1);
    }
  }
}

void Task2(void *pvparameters) {

  pinMode(Button, INPUT);

  MyServo.attach(ServoPin);

  int32_t StatusReceived = 0;
  int32_t button_state = 0;

  const TickType_t xTicksToWait = pdMS_TO_TICKS(100);

  while (1) {

    xQueueReceive(var_queue, &StatusReceived, xTicksToWait);
    vTaskDelay(1);
    Serial.print("Status received = ");
    Serial.println(StatusReceived);
    vTaskDelay(1);

    if (StatusReceived == 1) {

      MyServo.write(0);
      Serial.println("Gate Closed");
```

```
      Serial.println("");
      vTaskDelay(1);

      delay(100);
    }

    button_state = digitalRead(Button);
    int Gate_Opened = 0;

    if (button_state == HIGH) {

      MyServo.write(90);
      int Gate_Opened = 1;
      xQueueSend(var_queue, &Gate_Opened, 2);
      digitalWrite(Red_LED, LOW);  // Red_LED LOW
      Serial.println("Button is pressed");
      Serial.println("Gate Opened");
      Serial.println("Red LED Turns OFF");

      vTaskDelay(10);
    }
  }
}

void Task3(void *pvparameters) {

  int PIRState = LOW;  // we start, assuming no motion detected
  int val = 0;         // variable for reading the pin status
  int counter = 0;
  int CurrentState = 0;
  int PreviousState = 0;

  const TickType_t xTicksToWait = pdMS_TO_TICKS(100);

  pinMode(PIRSensor, INPUT);  // declare PIR Sensor as input

  int32_t GateOpenReceived = 0;


  while (1) {

    xQueueReceive(var_queue, &GateOpenReceived, xTicksToWait);
    Serial.print("GateOpenReceived = ");
    Serial.println(GateOpenReceived);
    vTaskDelay(1);

    if (GateOpenReceived == 1) {

      xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
```

```
      val = digitalRead(PIRSensor);  // read PIR Sensor input value
      if (val == HIGH) {             // check if the input is HIGH
        Serial.println("PIR Input is HIGH");
        vTaskDelay(1);
        if (PIRState == LOW) {
          // we have just turned on
          CurrentState = 1;
          // We only want to print on the output change, not state
          PIRState = HIGH;
          vTaskDelay(10);
        }
      }

      else {
        Serial.println("PIR Input is LOW");
        vTaskDelay(1);
        if (PIRState == HIGH) {
          // we have just turned of
          CurrentState = 0;
          // We only want to print on the output change, not state
          PIRState = LOW;
        }
      }

      if (CurrentState != PreviousState) {
        if (CurrentState == 1) {
          counter = counter + 1;
          Serial.println("");
          Serial.println("");
          Serial.print("Number of Vehicles = ");
          Serial.println(counter);
          Serial.println("");
          Serial.println("");
          vTaskDelay(100);
        }
      }

      xSemaphoreGive(xBinarySemaphore);
      vTaskDelay(1);
    }
  }
}

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

Some main important things to note that in the code we define IR Sensor, Servo Motor, Button, PIR Sensor and Red LED to the pin numbers 2, 3, 4, 7 and 13 respectively.

Then we use the queue structure for the communication between tasks like we transfer the value of IR Sensor Status to the other Task so that it can use it for other purposes (here as Button for User).

For our last task we use the Binary Semaphore as we don't want to interfere any other tasks in the last task so that it can execute completely without any pre-emption, whose task is to count the number of vehicles passes through PIR.

The basic working of the code has already been discussed in the section 1.5.

2.2 Images and Observations

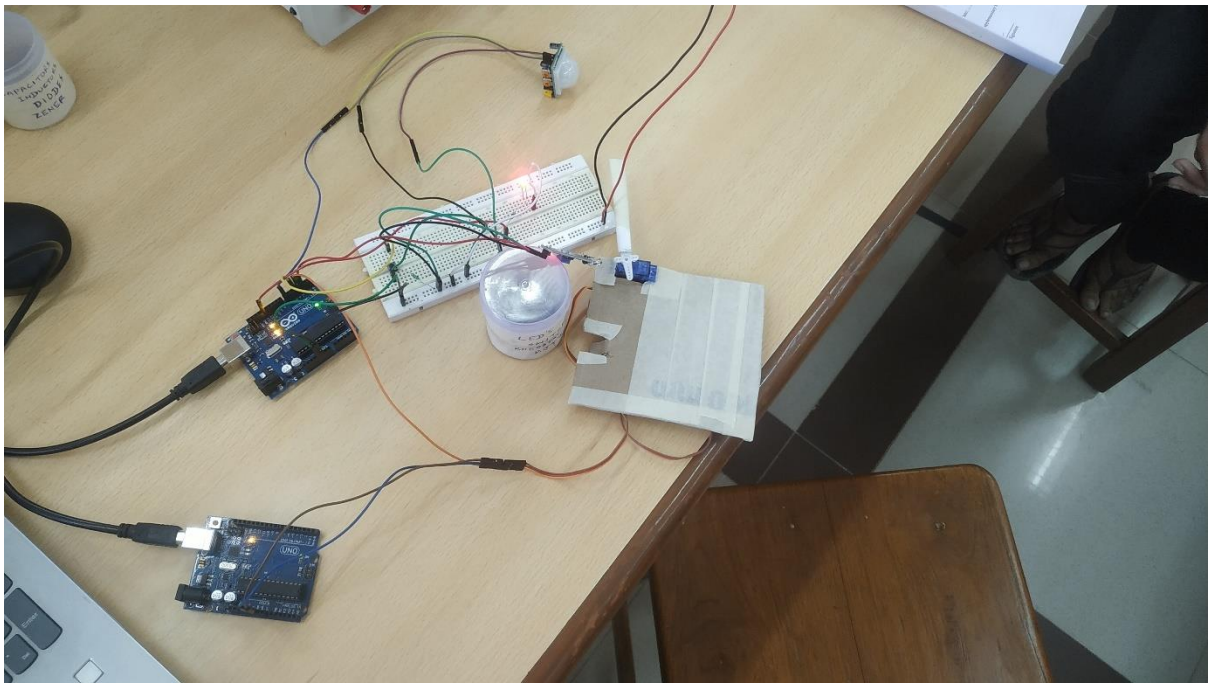

Figure 10: Full view of our project circuit interfaced with Arduino

As in the Figure 10 you can see that this is the overall circuit of our project, here we use the 100 Ω in the LEDs as well as in the switches in the form of pull up resistor and give external DC power supply to the Servo Moto via another Arduino Uno board because on using Standard DC power supply Servo Motor noise causes jitter in the voltage and due to this fluctuation the Servo Motor is not operating properly, that's why another Arduino Uno board is used because it has the tolerance power to run Servo Motor for a short period of time, although we know that it is a risky method.

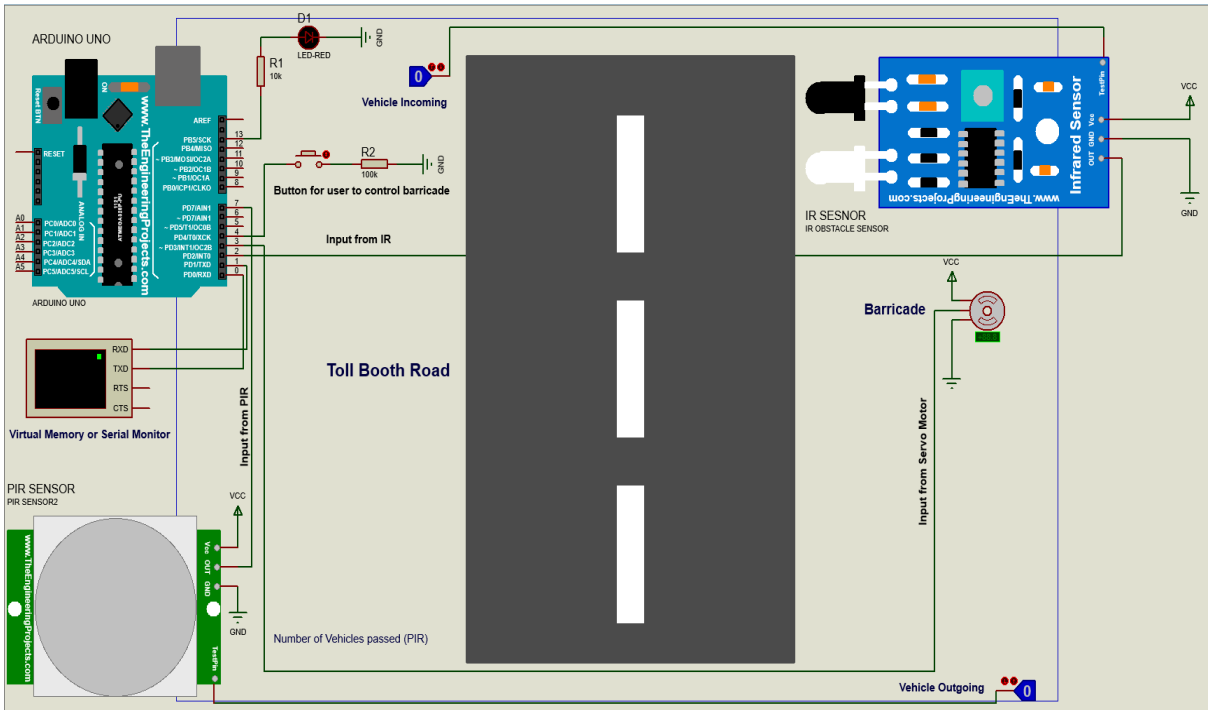

Figure 11: Schematic Diagram (Made in Proteus) of our overall circuit for getting better idea about the circuit.

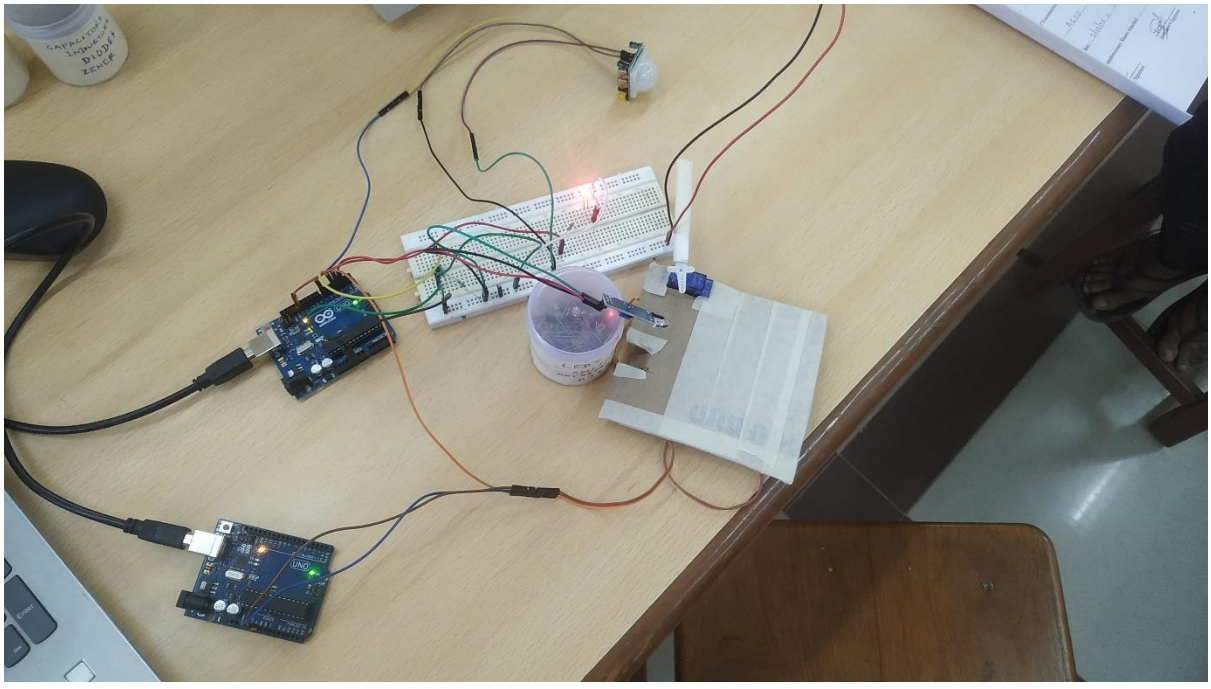

Figure 12: Circuit showing when the Task 1 is triggered (When IR Sensor senses something and gate is about to close).

In the Figure 12 we trigger the Task 1 event and IR Senses the Object (although not shown in the figure) and makes the Red LED turn ON and will close the Servo Motor.

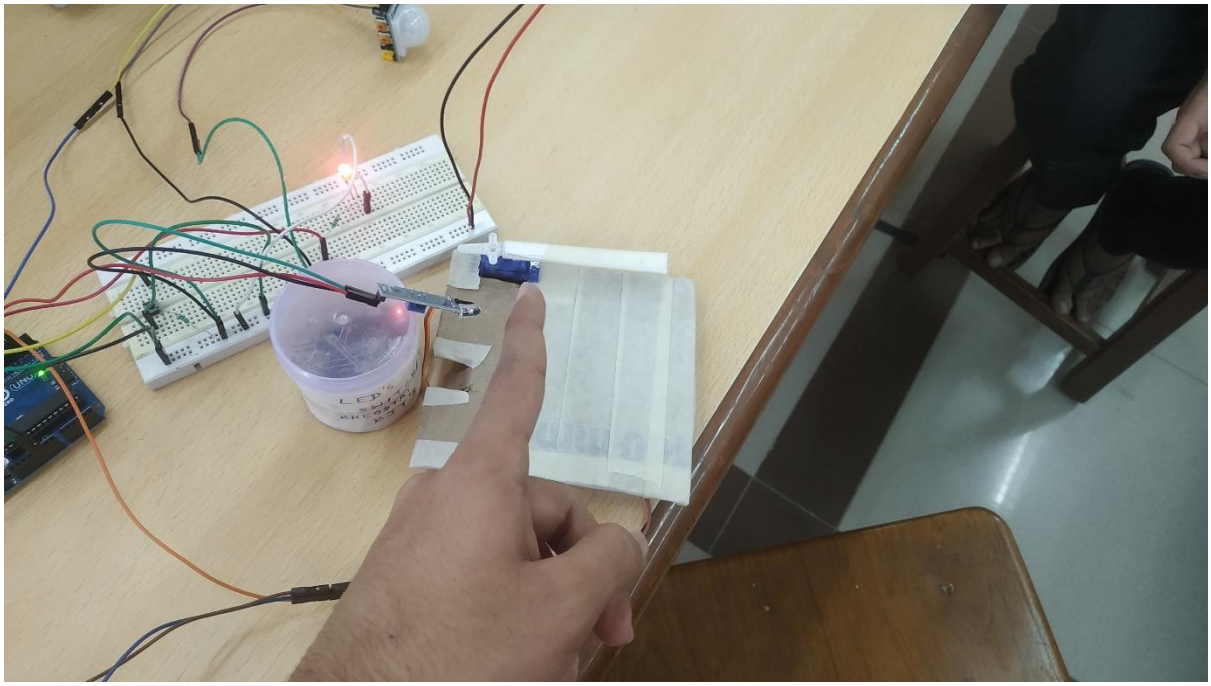

Figure 13: Sensing of IR and closing the barricade.

Here we can see that on object detection it closes the barricade and turn the Red LED ON.

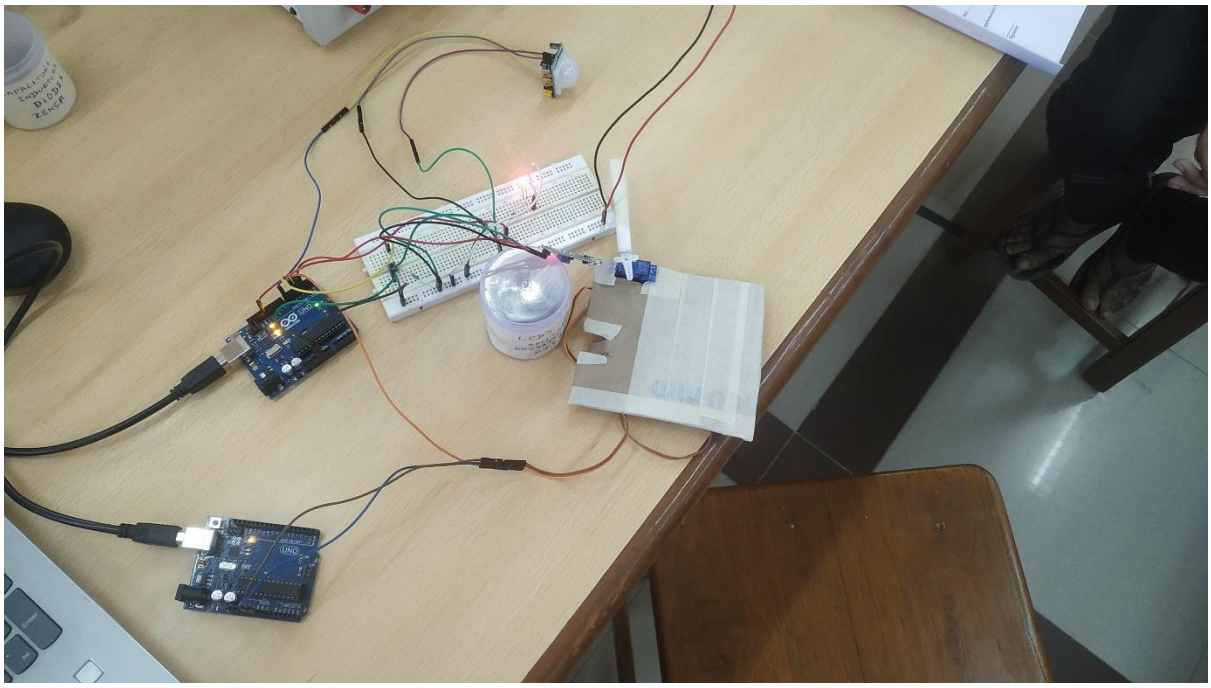

Figure 14: Pressing of button (Task 2) and counting of Vehicles through PIR (Task 3).

Here in this Figure 14, we can see that when we pressed the button gate opens and you will see a PIR Sensor after it, which counts the number of vehicles passed through its range.

Here are some screenshots which shows the results obtained from the Arduino Uno Serial monitor display in the PC.

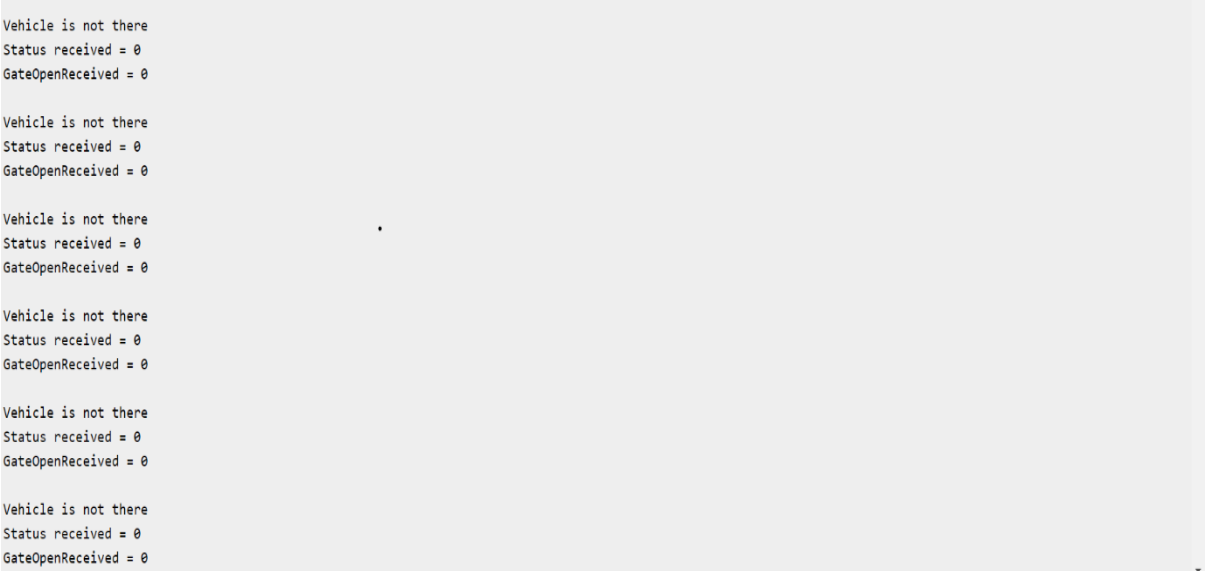

Figure 15: Serial Monitor Window of the Arduino showing that IR senses nothing till now.

Here IR Sensor didn't vehicle till now and is waiting for it, also it is connected with 2 things, first is as IR detects something it will turn the Red LED ON and second is that it will send a TRUE value (in Boolean Form) to the Task 2 so that it can use to communicate with Task 1 for User Button related operation. Here to send the data from Task 1 to Task 2 we use the Queue structure in FreeRTOS. That's why there is a Status Received = 0 in the above Serial Monitor.

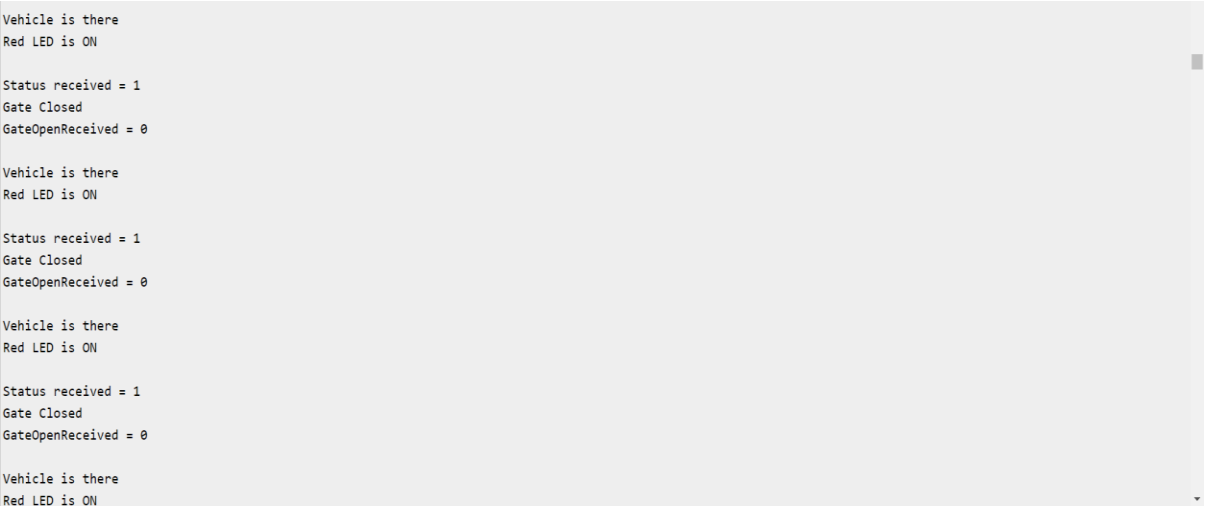

Figure 16: Serial Monitor Window of Arduino showing that in Task 1 IR detects a Vehicle.

Now as IR Sensor detects a Vehicle therefore Red LED will turn ON and it will send Status Received = 1 to Task 2 through which we will now control Servo Motor which will close the barricade.

GateOpenReceived = 0 (we will discuss what “GateOpenReceived” do).

```
Vehicle is there
Red LED is ON

Status received = 1
Button is pressed
Gate Opened
Red LED Turns OFF
GateOpenReceived = 1

PIR Input is LOW

Vehicle is there
Red LED is ON

Status received = 1
Button is pressed
Gate Opened
Red LED Turns OFF
GateOpenReceived = 1

PIR Input is LOW

Vehicle is there
Red LED is ON
```

Figure 17: Serial Monitor Window of Arduino showing that in Task 2 the Button is pressed.

When the user presses the Button acording to his will then barrciade will open and it sends this data to the Task 3 with the help of Queue alerting it that the gate is opened.. Red LED becomes OFF and GateOpenReceived = 1 implying that the message of gate opened reached to the Task 3.

“PIR input is LOW” shows that Task 3 has not received Gate Opened message therefore will not execute the function of PIR Sensor.

```
Number of Vehicles = 2

Status received = 0
Button is pressed
Gate Opened
Red LED Turns OFF
GateOpenReceived = 1

PIR Input is HIGH


Number of Vehicles = 3

Status received = 0
Button is pressed
Gate Opened
Red LED Turns OFF
GateOpenReceived = 1

PIR Input is HIGH


Number of Vehicles = 4
```

Figure 18: Serial Monitor Window of Arduino showing that the Task 3 has starting to execute the PIR Sensor’s Job.

Since the Task 3 gets notified, that Gate is opened, any number of vehicles cross from its range will increase the count and show to the Serial Monitor (Number of Vehicles), we will use Binary Semaphore here so that there will be no error in counting the number of vehicles since this will the last job and also lowest priority Task.

Conclusion

In this project we learn about how can we use queues, semaphores and demonstrate its working, by using it, we can handle, communicate the multiple tasks easily and also synchronised with each other but it also makes the code complex.

Since, an immediate response is given within a predetermined amount of time by a real-time operating system. For systems that must complete important tasks with a high priority and short deadlines, it is the most obvious choice.

The main benefit of implementing an RTOS in a microcontroller is the reduction in effort and time required for the development process, an RTOS manages resources, sets priorities, and maintains task coherence to make up for the microcontroller's limited capabilities, once you've decided to use an RTOS in your project, you can choose from a variety of different embedded RTOS solutions to find a solution that works for you. Any hardware platform has options, whether provided by closed-source initiatives or proprietary vendors.

It's ideal to always aim to streamline the development process to make it quicker and more effective. Using an RTOS is one method of accomplishing that for numerous projects.

The above project comes under the category of soft based Real-time systems (highest possibility), therefore not meeting the deadline will not result any critical consequences, we can use it for just stopping the vehicles in the Toll Booth to receive tax and where control of crossing (Barricade) is given to us, at last we can calculate that how much vehicles have crossed for a particular time period.

Bibliography

[1] FreeRTOS semaphore and mutex API functions vSemaphoreCreateBinary, xSemaphoreCreateCounting, xSemaphoreCreateMutex, xSemaphoreCreateRecursiveMutex, xSemaphoreTake, xSemaphoreTakeRecursive, xSemaphoreGive, xSemaphoreGiveRecursive, xSemaphoreGiveFromISR

[2] RTOS binary semaphore API (freertos.org)

[3] This page describes the xSemaphoreTake() FreeRTOS API function which is part of the RTOS semaphore API source code function set.

[4] This page describes the xSemaphoreGive() FreeRTOS API function which is part of the RTOS semaphore API source code function set. FreeRTOS is a professional grade open source RTOS for microcontrollers.

[5] Arduino UNO CH340 Board at Rs 250/piece | Girgaon | Mumbai | ID: 19651448730 (indiamart.com)

[6] 0602_0.jpg (1000×339) (embeddedlinux.org.cn)

[7] TowerPro SG 90 Micro Servo Motor : Amazon.in: Industrial & Scientific

[8] Servo Motor Basics, Working Principle & Theory (circuitdigest.com)

[9] IR sensor Working Principle and Applications | Robu.in

[10] IR Infrared Obstacle Avoider Sensor Module (makerbazar.in)

[11] What is IR Sensor - IR Reveiver and Transmitter, Pinout, Specifications (techzeero.com)

[12] Overview | PIR Motion Sensor | Adafruit Learning System

[13] How HC-SR501 PIR Sensor Works & How To Interface It With Arduino (lastminuteengineers.com)

[14] Communication Between RTOS Tasks - Open4Tech

[15] FreeRTOS - Open Source for Embedded Systems - FreeRTOS xQueueCreate() API function descriptions

[16] FreeRTOS - Open Source Software for Embedded Systems - FreeRTOS xQueueSend() API function descriptions

[17] FreeRTOS - Open Source Software for Embedded Systems - FreeRTOS xQueueReceive() API function description and example