SURVEILLANCE CAR USING ESP32 CAM MODULE | ESP32 CAMERA WI-FI CAR

REPORT BY:

SAHANA S (2022503049)

MAANASA PRATHAP CHANDER (2022503065)

SHREYA SANKARAN (2022503071)

AKSHAYA A (2022503579)

NEERAJA A (2022503581)

INDEX

Introduction:

The Surveillance Car using the ESP32 Cam module is a project that involves building a remote-controlled car equipped with a camera for surveillance purposes. The ESP32 Cam module, based on the ESP32 microcontroller, provides both the processing power and the capability to capture and stream video. The project allows users to remotely control the car over Wi-Fi, view live video footage, and potentially implement additional features for surveillance and exploration.

A surveillance car using ESP32-CAM is a system that utilizes the ESP32-CAM board and a car chassis to create a mobile surveillance device. The ESP32-CAM is a low-cost development board that integrates a small camera module and Wi-Fi connectivity. The car chassis allows the device to move around and capture video in different locations. The system can be controlled through a web interface hosted on the ESP32-CAM board. The web interface allows the user to control the car's movement, view live video streams, and take snapshots of the video feed. Additionally, the system can be programmed to detect motion using computer vision algorithms, such as object detection and tracking, and send alerts to the user. The surveillance car using ESP32-CAM has potential applications in home security, monitoring of remote locations, and industrial surveillance. With its low cost and easy-to-use interface, it provides a convenient solution for anyone who needs to monitor their surroundings remotely.

How It Works:

Initialization:

The ESP32 Cam module is powered up, and the initialization routine begins. Wi-Fi Connection: The ESP32 Cam connects to a preconfigured Wi-Fi network, making it accessible on the local network. Web Server Hosting: The ESP32 Cam sets up a simple web server that hosts a web page. User Interaction: Users access the web page served by the ESP32 Cam using a web browser. The web page includes controls for moving the car. Motor Control: Based on user inputs from the web interface, the ESP32 Cam sends commands to the motor driver to control the movement of the car. Live Video Streaming: The ESP32 Cam captures video footage using its integrated camera. The live video stream is made available through the web server. Remote Surveillance: Users can remotely monitor the video feed in real-time, effectively using the car for surveillance purposes.

Web controlled surveillance cars can be built using the ESP32-CAM module. Apart from the ESP32-Camera module, we will need 4 DC motors with its Robo car chassis and L293D motor driver module to build this Robocar. ESP32 is one of the popular boards to build IoT-based projects.

Here HTTP communication protocol is used to receive video streaming from the camera over the web browser using an IP address. The web page will also have buttons to move the car in Left, Right, Forward, and reverse directions and can vary the speed with light control as well.

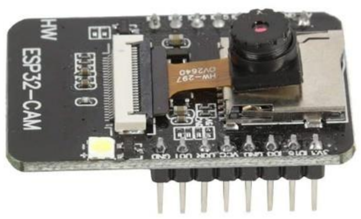
Uses and Applications:

This project is a practical application of IoT (Internet of Things) and robotics, providing an opportunity to learn about microcontroller programming, motor control, and wireless communication. It demonstrates the integration of hardware components and software logic to create a functional and remotely controlled surveillance car. A Surveillance Car using the ESP32 Cam module has various practical applications due to its ability to capture and transmit video over Wi-Fi. Here are some potential uses: Home Security: Deploy the surveillance car to monitor your home when you're away. You can remotely access the live video feed to check for any unusual activities or intruders. Remote Monitoring: Use the surveillance car to monitor remote locations. This could be useful for checking on vacation homes, construction sites, or other areas where real-time visual inspection is necessary. Educational Tool: The project can serve as an educational tool for learning about robotics, IoT, and programming. Students and hobbyists can explore concepts such as motor control, wireless communication, and camera integration. Pet Monitoring: Keep an eye on your pets while you're not at home.

Components Used In Our Project:

✓4WD Car kit (It has 4 motors, 4 wheels, and connectors)

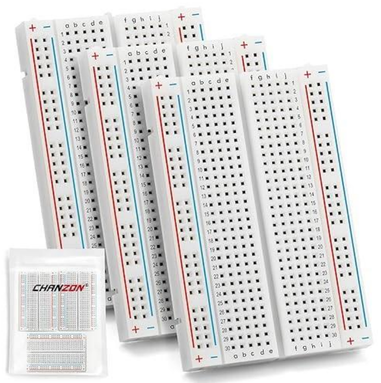
✓ESP32 Cam module



✓L298N motor driver module

✓7-12 V DC Battery (in our case lipo 3s battery)

✓Breadboard

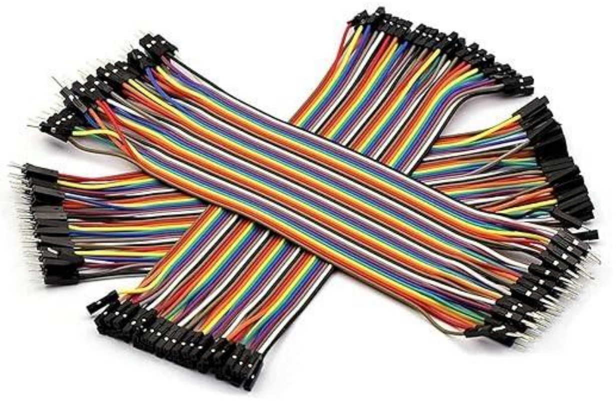


✓Arduino Uno (for uploading code)

✓Double sided tape

✓Glue gun

✓Jumper wires



Uses Of The Various Components Used In Our Project:

4WD Car kit

- Very handy and simple in assembling/disassembling.
  Strong components to withstand extreme terrain conditions.
  Transparent Car Acrylic Chassis.
  Attractive design.

ESP32 Cam module

- The ESP32-CAM can be widely used in intelligent IoT applications such as wireless video monitoring, Wi-Fi image upload, QR identification, and so on. The ESP32-CAM suit for IOT applications such as: Smart home devices image upload. Wireless monitoring.

L298N motor driver module

- The L298N is a dual H-Bridge motor driver which allows speed and direction control of two DC motors at the same time. The module can drive DC motors that have voltages between 5 and 35V, with a peak current up to 2A.

7-12 V DC Battery

- 12V 7Ah batteries are very popular deep cycle and general-purpose batteries, commonly used for powering medical equipment, security systems, UPS and other emergency systems, toys, scooters, fish finders, etc.

Breadboard

- A breadboard (sometimes called a plug block) is used for building temporary circuits. It is useful to designers because it allows components to be removed and replaced easily. It is useful to the person who wants to build a circuit to demonstrate its action, then to reuse the components in another circuit.

Arduino Uno

- Arduino UNO is a low-cost, flexible, and easy-to-use programmable open-source microcontroller board that can be integrated into a variety of electronic projects. This board can be interfaced with other Arduino boards, Arduino shields, Raspberry Pi boards and can control relays, LEDs, servos, and motors as an output.

Double sided tape

- It is designed to stick two surfaces together, often in a way which is not visible in the product, due to it being in between the objects rather than overlaying them. This allows for neater-looking projects and better craftsmanship. Double-sided tape can be either thin or dimensional.

Glue gun

- ♦ Aside from industrial uses of hot glue, glue guns are used in small joint attachment, surface lamination, and other product assembly applications in appliances, HVAC units, and mattresses. Glue guns can also be used to adhere to small joints in furniture assembly and woodworking projects.

Jumper wires

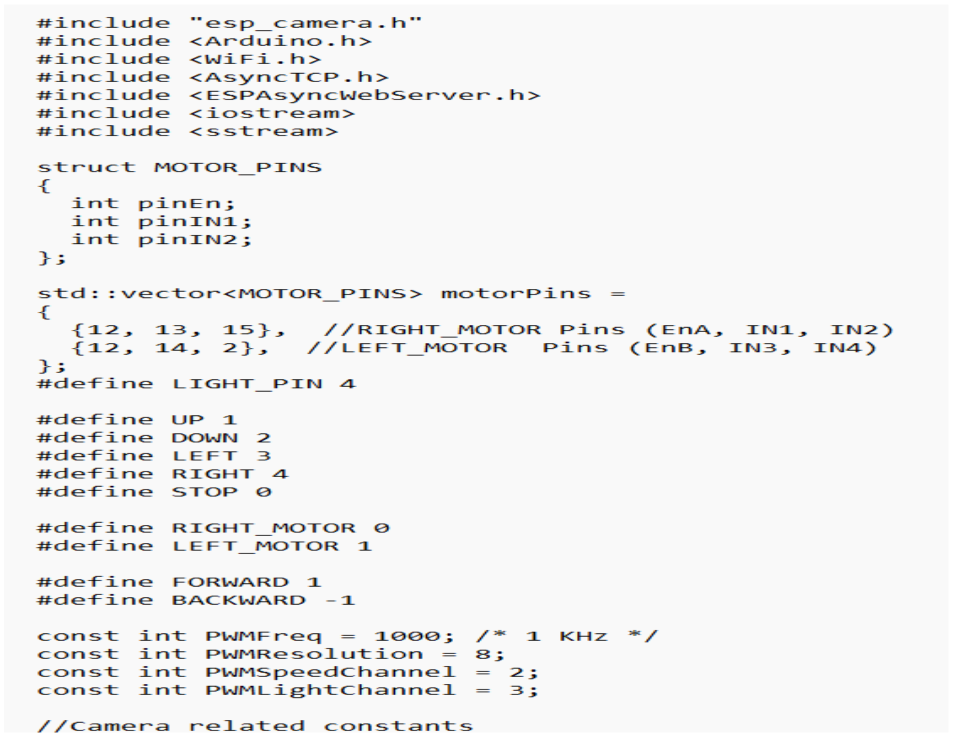
- ♦ A jumper wire is an electric wire that connects remote electric circuits used for printed circuit boards. By attaching a jumper wire on the circuit, it can be short-circuited and short-cut (jump) to the electric circuit.

Assembling Of The Car (Connections):

- ♦ Solder the wires to the gear motor.
- ♦ Mount all four motors on car chassis using connectors and screws.
- ♦ Join red to red and black to black wires of DC Motor on East side.
- ♦ Attach L298N motor driver module on car chassis now.

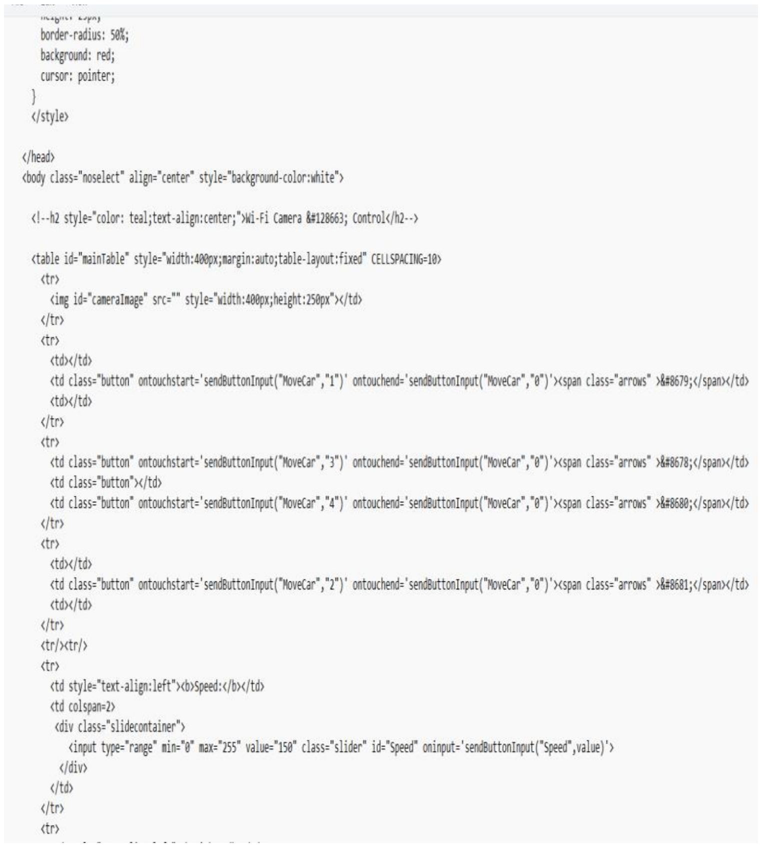
- Make the connections as per the circuit diagram.
- Connect the right-side motors to out1 and out2 pins of the L298N motor driver module.
- Connect the left-side motors to out3 and out4 pins of the L298N motor driver module.
- Attach dc power battery connectors to motor driver module, to+12 V pin and ground pin.
- Now connect L298N motor driver module to ESP32 cam pins as per table.
- Connect both enableA and B pins to ESP32 cam.
- Fix ESP32 cam module on car chassis using glue gun.
- Attach the wheels to the car now.

CODE 1:

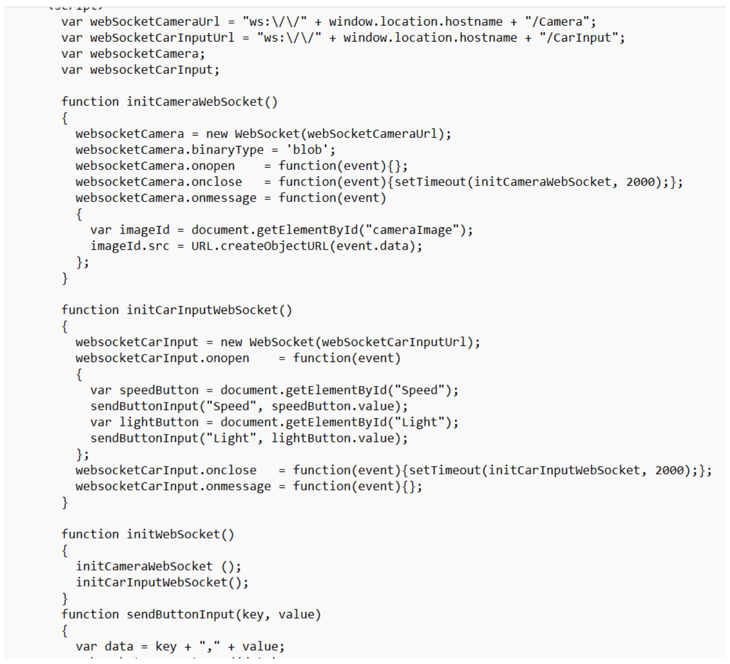


```
#include "esp_camera.h"
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <iostream>
#include <sstream>

struct MOTOR_PINS
{
  int pinEn;
  int pinIN1;
  int pinIN2;
};

std::vector<MOTOR_PINS> motorPins =
{
  {12, 13, 15},  //RIGHT_MOTOR Pins (EnA, IN1, IN2)
  {12, 14, 2},  //LEFT_MOTOR  Pins (EnB, IN3, IN4)
};
#define LIGHT_PIN 4

#define UP 1
#define DOWN 2
#define LEFT 3
#define RIGHT 4
#define STOP 0

#define RIGHT_MOTOR 0
#define LEFT_MOTOR 1

#define FORWARD 1
#define BACKWARD -1

const int PWMFreq = 1000; /* 1 KHz */
const int PWMResolution = 8;
const int PWMSpeedChannel = 2;
const int PWMLightChannel = 3;

//Camera related constants
```

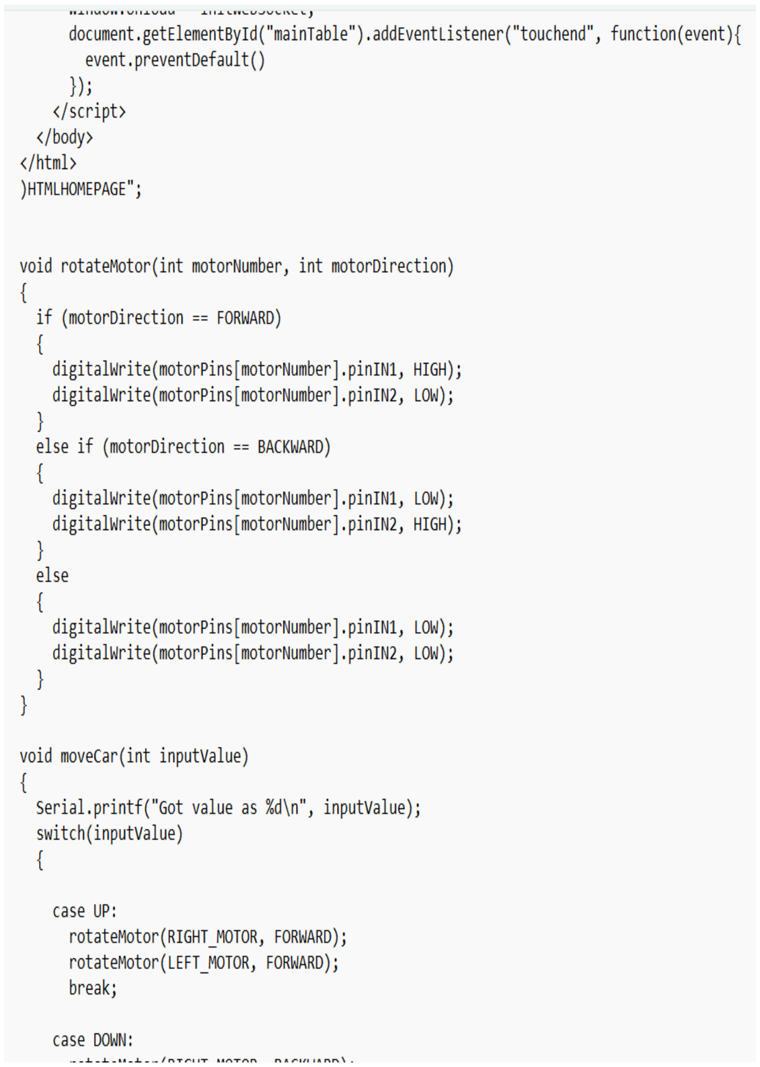
```
//Camera related constants
#define PWDN_GPIO_NUM     32
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM      0
#define SIOD_GPIO_NUM     26
#define SIOC_GPIO_NUM     27
#define Y9_GPIO_NUM       35
#define Y8_GPIO_NUM       34
#define Y7_GPIO_NUM       39
#define Y6_GPIO_NUM       36
#define Y5_GPIO_NUM       21
#define Y4_GPIO_NUM       19
#define Y3_GPIO_NUM       18
#define Y2_GPIO_NUM        5
#define VSYNC_GPIO_NUM    25
#define HREF_GPIO_NUM     23
#define PCLK_GPIO_NUM     22

const char* ssid     = "MyWiFiCar";
const char* password = "12345678";

AsyncWebServer server(80);
AsyncWebSocket wsCamera("/Camera");
AsyncWebSocket wsCarInput("/CarInput");
uint32_t cameraClientId = 0;

const char* htmlHomePage PROGMEM = R"HTMLHOMEPAGE(
<!DOCTYPE html>
<html>
  <head>
  <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no">
    <style>
    .arrows {
      font-size:40px;
      color:red;
    }
    td.button {
      background-color:black;
      border-radius:25%;
      box-shadow: 5px 5px #888888;
    }
    box-shadow: none;
  }
  .noselect {
    -webkit-touch-callout: none; /* iOS Safari */
      -webkit-user-select: none; /* Safari */
       -khtml-user-select: none; /* Konqueror HTML */
         -moz-user-select: none; /* Firefox */
          -ms-user-select: none; /* Internet Explorer/Edge */
              user-select: none; /* Non-prefixed version, currently
                                    supported by Chrome and Opera */
  }
  .slidecontainer {
    width: 100%;
  }
  .slider {
    -webkit-appearance: none;
    width: 100%;
    height: 15px;
    border-radius: 5px;
    background: #d3d3d3;
    outline: none;
    opacity: 0.7;
    -webkit-transition: .2s;
    transition: opacity .2s;
  }
  .slider:hover {
    opacity: 1;
  }

  .slider::-webkit-slider-thumb {
    -webkit-appearance: none;
    appearance: none;
    width: 25px;
    height: 25px;
    border-radius: 50%;
    background: red;
    cursor: pointer;
  }
  .slider::-moz-range-thumb {
    width: 25px;
```



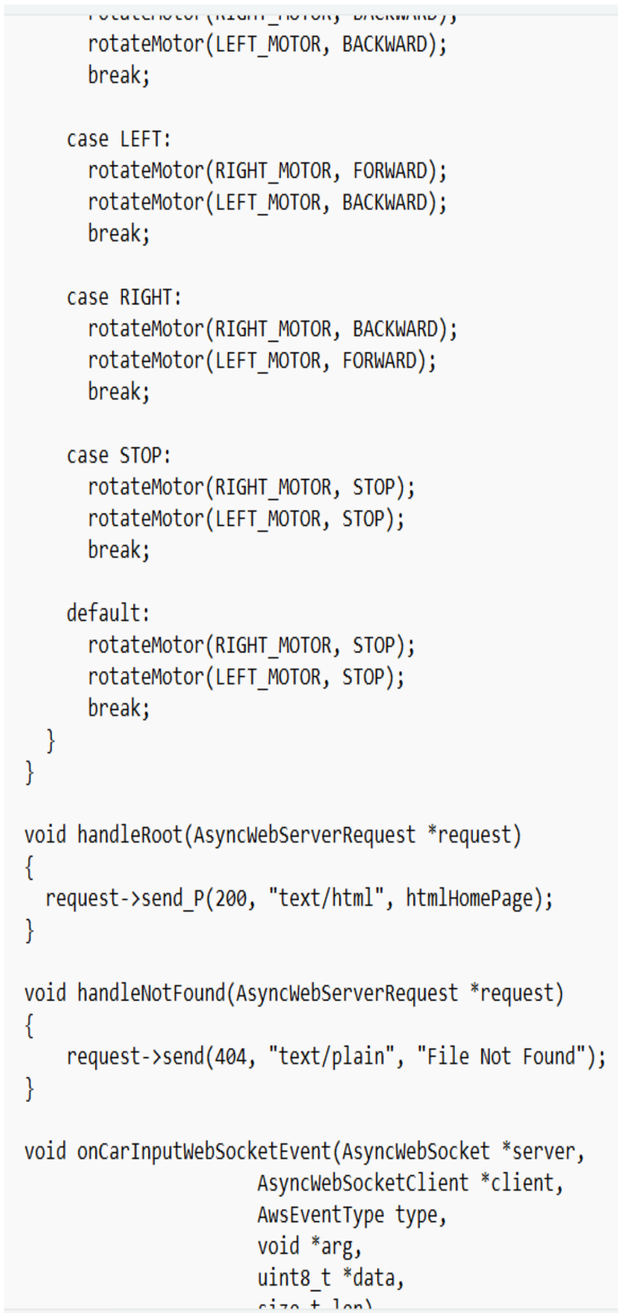
```
    border-radius: 50%;
    background: red;
    cursor: pointer;
  }
  </style>

</head>
<body class="noselect" align="center" style="background-color:white">

  <!--h2 style="color: teal;text-align:center;">Wi-Fi Camera &#128663; Control</h2-->

  <table id="mainTable" style="width:400px;margin:auto;table-layout:fixed" CELLSPACING=10>
    <tr>
      <img id="cameraImage" src="" style="width:400px;height:250px"></td>
    </tr>
    <tr>
      <td></td>
      <td class="button" ontouchstart='sendButtonInput("MoveCar","1")' ontouchend='sendButtonInput("MoveCar","0")'><span class="arrows" >&#8679;</span></td>
      <td></td>
    </tr>
    <tr>
      <td class="button" ontouchstart='sendButtonInput("MoveCar","3")' ontouchend='sendButtonInput("MoveCar","0")'><span class="arrows" >&#8678;</span></td>
      <td class="button"></td>
      <td class="button" ontouchstart='sendButtonInput("MoveCar","4")' ontouchend='sendButtonInput("MoveCar","0")'><span class="arrows" >&#8680;</span></td>
    </tr>
    <tr>
      <td></td>
      <td class="button" ontouchstart='sendButtonInput("MoveCar","2")' ontouchend='sendButtonInput("MoveCar","0")'><span class="arrows" >&#8681;</span></td>
      <td></td>
    </tr>
    <tr/><tr/>
    <tr>
      <td style="text-align:left"><b>Speed:</b></td>
      <td colspan=2>
       <div class="slidecontainer">
          <input type="range" min="0" max="255" value="150" class="slider" id="Speed" oninput='sendButtonInput("Speed",value)'>
        </div>
      </td>
    </tr>
    <tr>
```



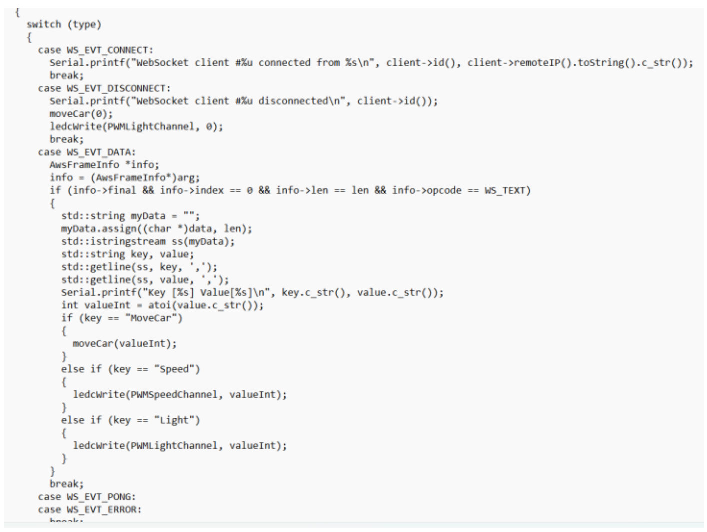
```
      var webSocketCameraUrl = "ws:\/\/" + window.location.hostname + "/Camera";
      var webSocketCarInputUrl = "ws:\/\/" + window.location.hostname + "/CarInput";
      var websocketCamera;
      var websocketCarInput;

      function initCameraWebSocket()
      {
        websocketCamera = new WebSocket(webSocketCameraUrl);
        websocketCamera.binaryType = 'blob';
        websocketCamera.onopen    = function(event){};
        websocketCamera.onclose   = function(event){setTimeout(initCameraWebSocket, 2000);};
        websocketCamera.onmessage = function(event)
        {
          var imageId = document.getElementById("cameraImage");
          imageId.src = URL.createObjectURL(event.data);
        };
      }

      function initCarInputWebSocket()
      {
        websocketCarInput = new WebSocket(webSocketCarInputUrl);
        websocketCarInput.onopen    = function(event)
        {
          var speedButton = document.getElementById("Speed");
          sendButtonInput("Speed", speedButton.value);
          var lightButton = document.getElementById("Light");
          sendButtonInput("Light", lightButton.value);
        };
        websocketCarInput.onclose   = function(event){setTimeout(initCarInputWebSocket, 2000);};
        websocketCarInput.onmessage = function(event){};
      }

      function initWebSocket()
      {
        initCameraWebSocket ();
        initCarInputWebSocket();
      }
      function sendButtonInput(key, value)
      {
        var data = key + "," + value;
```



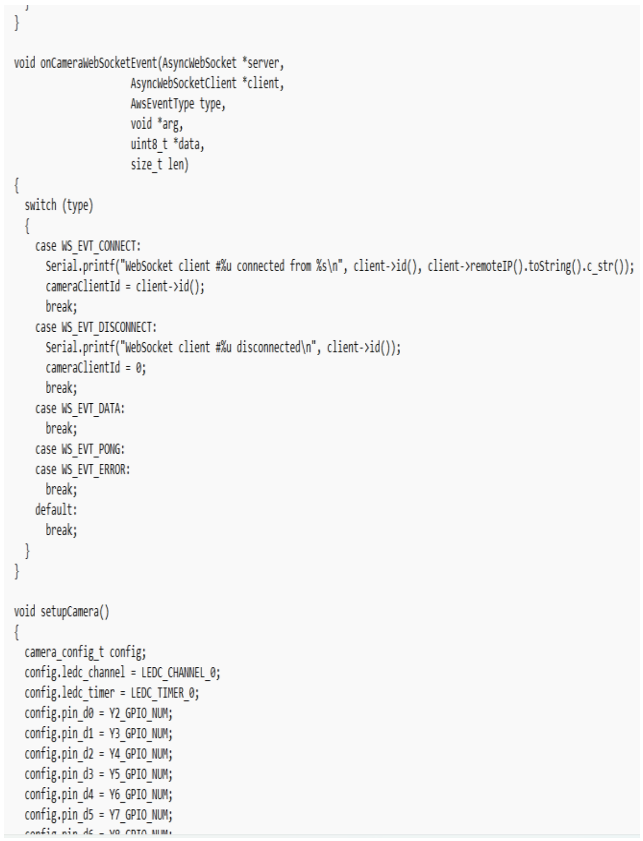
```
      document.getElementById("mainTable").addEventListener("touchend", function(event){
        event.preventDefault()
      });
    </script>
  </body>
</html>
)HTMLHOMEPAGE";


void rotateMotor(int motorNumber, int motorDirection)
{
  if (motorDirection == FORWARD)
  {
    digitalWrite(motorPins[motorNumber].pinIN1, HIGH);
    digitalWrite(motorPins[motorNumber].pinIN2, LOW);
  }
  else if (motorDirection == BACKWARD)
  {
    digitalWrite(motorPins[motorNumber].pinIN1, LOW);
    digitalWrite(motorPins[motorNumber].pinIN2, HIGH);
  }
  else
  {
    digitalWrite(motorPins[motorNumber].pinIN1, LOW);
    digitalWrite(motorPins[motorNumber].pinIN2, LOW);
  }
}

void moveCar(int inputValue)
{
  Serial.printf("Got value as %d\n", inputValue);
  switch(inputValue)
  {

    case UP:
      rotateMotor(RIGHT_MOTOR, FORWARD);
      rotateMotor(LEFT_MOTOR, FORWARD);
      break;

    case DOWN:
```



```
      rotateMotor(RIGHT_MOTOR, BACKWARD);
      rotateMotor(LEFT_MOTOR, BACKWARD);
      break;

    case LEFT:
      rotateMotor(RIGHT_MOTOR, FORWARD);
      rotateMotor(LEFT_MOTOR, BACKWARD);
      break;

    case RIGHT:
      rotateMotor(RIGHT_MOTOR, BACKWARD);
      rotateMotor(LEFT_MOTOR, FORWARD);
      break;

    case STOP:
      rotateMotor(RIGHT_MOTOR, STOP);
      rotateMotor(LEFT_MOTOR, STOP);
      break;

    default:
      rotateMotor(RIGHT_MOTOR, STOP);
      rotateMotor(LEFT_MOTOR, STOP);
      break;
  }
}

void handleRoot(AsyncWebServerRequest *request)
{
  request->send_P(200, "text/html", htmlHomePage);
}

void handleNotFound(AsyncWebServerRequest *request)
{
    request->send(404, "text/plain", "File Not Found");
}

void onCarInputWebSocketEvent(AsyncWebSocket *server,
                      AsyncWebSocketClient *client,
                      AwsEventType type,
                      void *arg,
                      uint8_t *data,
                      size_t len)
```



```
{
  switch (type)
  {
    case WS_EVT_CONNECT:
      Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
      break;
    case WS_EVT_DISCONNECT:
      Serial.printf("WebSocket client #%u disconnected\n", client->id());
      moveCar(0);
      ledcWrite(PWMLightChannel, 0);
      break;
    case WS_EVT_DATA:
      AwsFrameInfo *info;
      info = (AwsFrameInfo*)arg;
      if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT)
      {
        std::string myData = "";
        myData.assign((char *)data, len);
        std::istringstream ss(myData);
        std::string key, value;
        std::getline(ss, key, ',');
        std::getline(ss, value, ',');
        Serial.printf("Key [%s] Value[%s]\n", key.c_str(), value.c_str());
        int valueInt = atoi(value.c_str());
        if (key == "MoveCar")
        {
          moveCar(valueInt);
        }
        else if (key == "Speed")
        {
          ledcWrite(PWMSpeedChannel, valueInt);
        }
        else if (key == "Light")
        {
          ledcWrite(PWMLightChannel, valueInt);
        }
      }
      break;
    case WS_EVT_PONG:
    case WS_EVT_ERROR:
      break;
```



```
  }
}

void onCameraWebSocketEvent(AsyncWebSocket *server,
                      AsyncWebSocketClient *client,
                      AwsEventType type,
                      void *arg,
                      uint8_t *data,
                      size_t len)
{
  switch (type)
  {
    case WS_EVT_CONNECT:
      Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
      cameraClientId = client->id();
      break;
    case WS_EVT_DISCONNECT:
      Serial.printf("WebSocket client #%u disconnected\n", client->id());
      cameraClientId = 0;
      break;
    case WS_EVT_DATA:
      break;
    case WS_EVT_PONG:
    case WS_EVT_ERROR:
      break;
    default:
      break;
  }
}

void setupCamera()
{
  camera_config_t config;
  config.ledc_channel = LEDC_CHANNEL_0;
  config.ledc_timer = LEDC_TIMER_0;
  config.pin_d0 = Y2_GPIO_NUM;
  config.pin_d1 = Y3_GPIO_NUM;
  config.pin_d2 = Y4_GPIO_NUM;
  config.pin_d3 = Y5_GPIO_NUM;
  config.pin_d4 = Y6_GPIO_NUM;
  config.pin_d5 = Y7_GPIO_NUM;
  config.pin_reset = RESET_GPIO_NUM;
  config.xclk_freq_hz = 20000000;
  config.pixel_format = PIXFORMAT_JPEG;

  config.frame_size = FRAMESIZE_VGA;
  config.jpeg_quality = 10;
  config.fb_count = 1;

  // camera init
  esp_err_t err = esp_camera_init(&config);
  if (err != ESP_OK)
  {
    Serial.printf("Camera init failed with error 0x%x", err);
    return;
  }

  if (psramFound())
  {
    heap_caps_malloc_extmem_enable(20000);
    Serial.printf("PSRAM initialized. malloc to take memory from psram above this size");
  }
}

void sendCameraPicture()
{
  if (cameraClientId == 0)
  {
    return;
  }
  unsigned long  startTime1 = millis();
  //capture a frame
  camera_fb_t * fb = esp_camera_fb_get();
  if (!fb)
  {
      Serial.println("Frame buffer could not be acquired");
      return;
  }

  unsigned long  startTime2 = millis();
  wsCamera.binary(cameraClientId, fb->buf, fb->len);
  esp_camera_fb_return(fb);
```

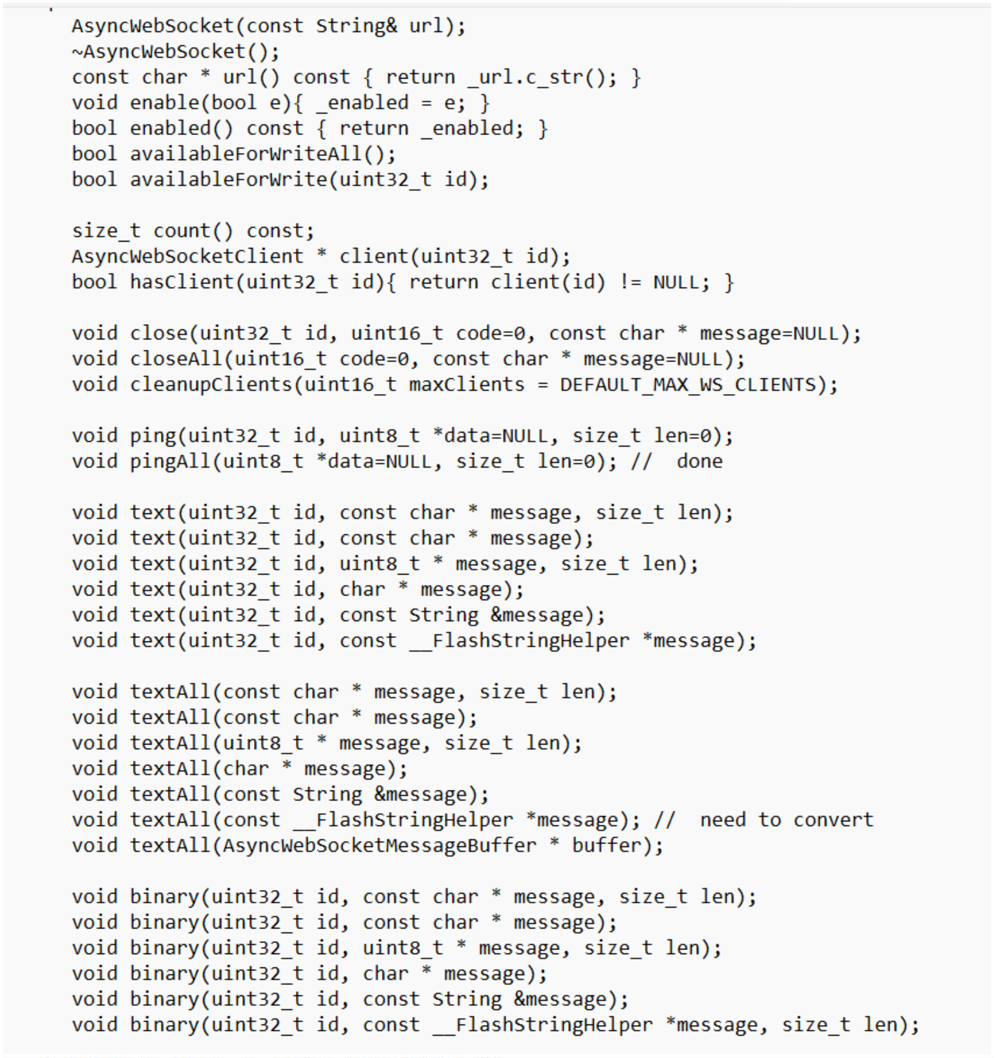
```
  //Wait for message to be delivered
  while (true)
  {
    AsyncWebSocketClient * clientPointer = wsCamera.client(cameraClientId);
    if (!clientPointer || !(clientPointer->queueIsFull()))
    {
      break;
    }
    delay(1);
  }

  unsigned long  startTime3 = millis();
  Serial.printf("Time taken Total: %d|%d|%d\n",startTime3 - startTime1, startTime2 - startTime1, startTime3-startTime2 );
}

void setUpPinModes()
{
  //Set up PWM
  ledcSetup(PWMSpeedChannel, PWMFreq, PWMResolution);
  ledcSetup(PWMLightChannel, PWMFreq, PWMResolution);

  for (int i = 0; i < motorPins.size(); i++)
  {
    pinMode(motorPins[i].pinEn, OUTPUT);
    pinMode(motorPins[i].pinIN1, OUTPUT);
    pinMode(motorPins[i].pinIN2, OUTPUT);

    /* Attach the PWM Channel to the motor enb Pin */
    ledcAttachPin(motorPins[i].pinEn, PWMSpeedChannel);
  }
  moveCar(STOP);

  pinMode(LIGHT_PIN, OUTPUT);
  ledcAttachPin(LIGHT_PIN, PWMLightChannel);
}


void setup(void)
{
  setUpPinModes();
  Serial.begin(115200);
  moveCar(STOP);

  pinMode(LIGHT_PIN, OUTPUT);
  ledcAttachPin(LIGHT_PIN, PWMLightChannel);
}


void setup(void)
{
  setUpPinModes();
  Serial.begin(115200);

  WiFi.softAP(ssid, password);
  IPAddress IP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(IP);

  server.on("/", HTTP_GET, handleRoot);
  server.onNotFound(handleNotFound);

  wsCamera.onEvent(onCameraWebSocketEvent);
  server.addHandler(&wsCamera);

  wsCarInput.onEvent(onCarInputWebSocketEvent);
  server.addHandler(&wsCarInput);

  server.begin();
  Serial.println("HTTP server started");

  setupCamera();
}


void loop()
{
  wsCamera.cleanupClients();
  wsCarInput.cleanupClients();
  sendCameraPicture();
  Serial.printf("SPIRam Total heap %d, SPIRam Free Heap %d\n", ESP.getPsramSize(), ESP.getFreePsram());
}
```

```
  moveCar(STOP);

  pinMode(LIGHT_PIN, OUTPUT);
  ledcAttachPin(LIGHT_PIN, PWMLightChannel);
}


void setup(void)
{
  setUpPinModes();
  Serial.begin(115200);

  WiFi.softAP(ssid, password);
  IPAddress IP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(IP);

  server.on("/", HTTP_GET, handleRoot);
  server.onNotFound(handleNotFound);

  wsCamera.onEvent(onCameraWebSocketEvent);
  server.addHandler(&wsCamera);

  wsCarInput.onEvent(onCarInputWebSocketEvent);
  server.addHandler(&wsCarInput);

  server.begin();
  Serial.println("HTTP server started");

  setupCamera();
}


void loop()
{
  wsCamera.cleanupClients();
  wsCarInput.cleanupClients();
  sendCameraPicture();
  Serial.printf("SPIRam Total heap %d, SPIRam Free Heap %d\n", ESP.getPsramSize(), ESP.getFreePsram());
}
```

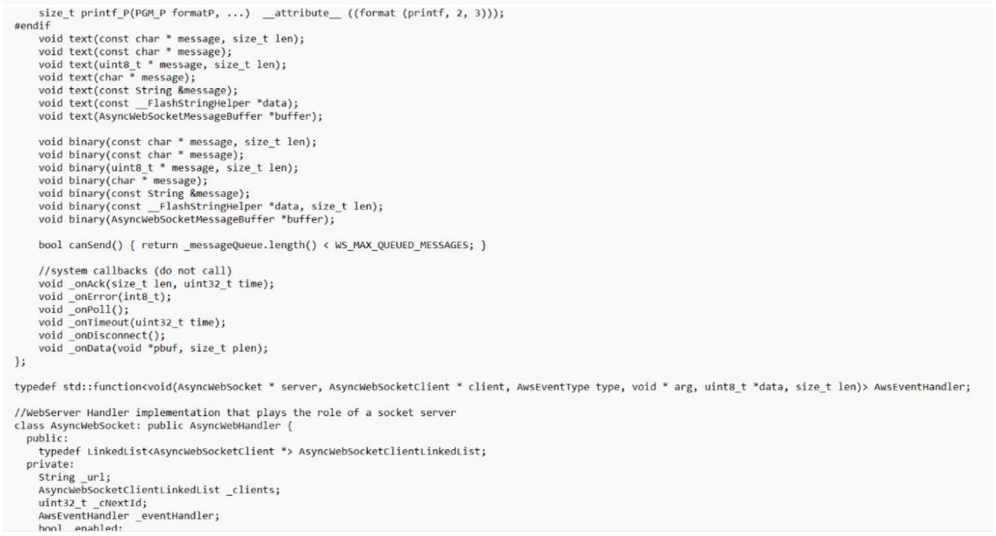
Code2:

```
#endif
    size_t printfAll_P(PGM_P formatP, ...)  __attribute__ ((format (printf, 2, 3)));

    //event listener
    void onEvent(AwsEventHandler handler){
      _eventHandler = handler;
    }

    //system callbacks (do not call)
    uint32_t _getNextId(){ return _cNextId++; }
    void _addClient(AsyncWebSocketClient * client);
    void _handleDisconnect(AsyncWebSocketClient * client);
    void _handleEvent(AsyncWebSocketClient * client, AwsEventType type, void * arg, uint8_t *data, size_t len);
    virtual bool canHandle(AsyncWebServerRequest *request) override final;
    virtual void handleRequest(AsyncWebServerRequest *request) override final;


    //  messagebuffer functions/objects.
    AsyncWebSocketMessageBuffer * makeBuffer(size_t size = 0);
    AsyncWebSocketMessageBuffer * makeBuffer(uint8_t * data, size_t size);
    LinkedList<AsyncWebSocketMessageBuffer *> _buffers;
    void _cleanBuffers();

    AsyncWebSocketClientLinkedList getClients() const;
};

//WebServer response to authenticate the socket and detach the tcp client from the web server request
class AsyncWebSocketResponse: public AsyncWebServerResponse {
  private:
    String _content;
    AsyncWebSocket *_server;
  public:
    AsyncWebSocketResponse(const String& key, AsyncWebSocket *server);
    void _respond(AsyncWebServerRequest *request);
    size_t _ack(AsyncWebServerRequest *request, size_t len, uint32_t time);
    bool _sourceValid() const { return true; }
};


#endif /* ASYNCWEBSOCKET_H_ */
```

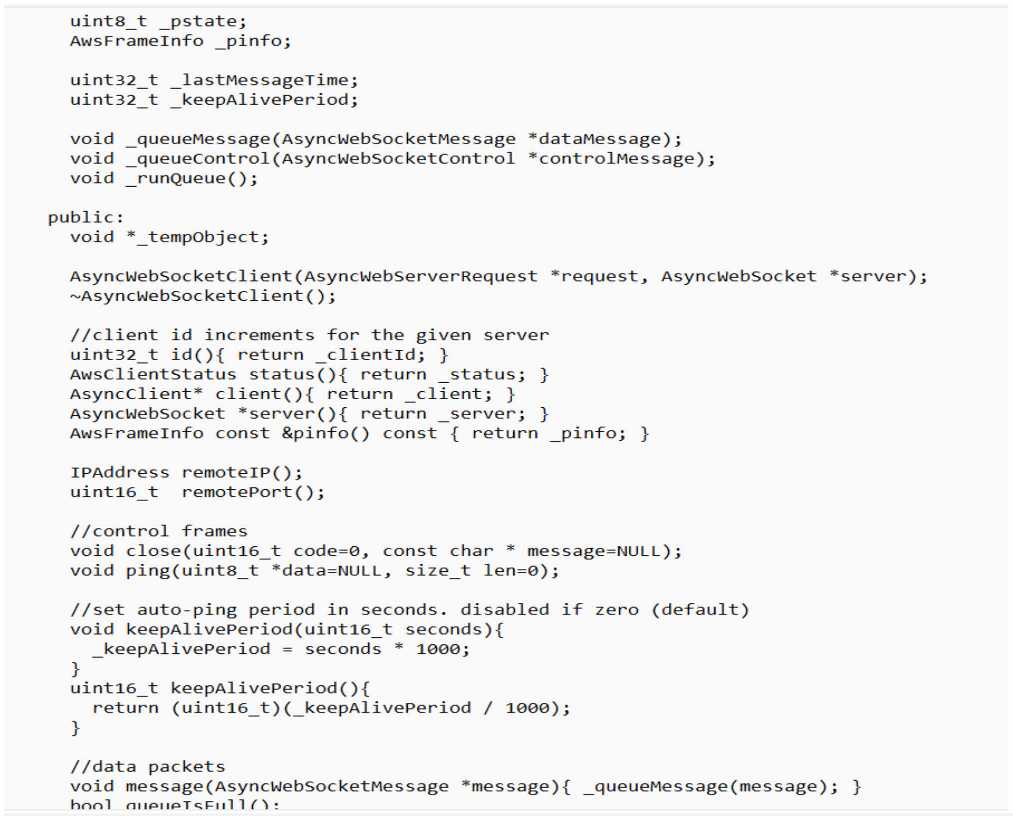
```
    void binaryAll(const char * message, size_t len);
    void binaryAll(const char * message);
    void binaryAll(uint8_t * message, size_t len);
    void binaryAll(char * message);
    void binaryAll(const String &message);
    void binaryAll(const __FlashStringHelper *message, size_t len);
    void binaryAll(AsyncWebSocketMessageBuffer * buffer);

    void message(uint32_t id, AsyncWebSocketMessage *message);
    void messageAll(AsyncWebSocketMultiMessage *message);

    size_t printf(uint32_t id, const char *format, ...)  __attribute__ ((format (printf, 3, 4)));
    size_t printfAll(const char *format, ...)  __attribute__ ((format (printf, 2, 3)));
#ifndef ESP32
    size_t printf_P(uint32_t id, PGM_P formatP, ...)  __attribute__ ((format (printf, 3, 4)));
#endif
    size_t printfAll_P(PGM_P formatP, ...)  __attribute__ ((format (printf, 2, 3)));

    //event listener
    void onEvent(AwsEventHandler handler){
      _eventHandler = handler;
    }

    //system callbacks (do not call)
    uint32_t _getNextId(){ return _cNextId++; }
    void _addClient(AsyncWebSocketClient * client);
    void _handleDisconnect(AsyncWebSocketClient * client);
    void _handleEvent(AsyncWebSocketClient * client, AwsEventType type, void * arg, uint8_t *data, size_t len);
    virtual bool canHandle(AsyncWebServerRequest *request) override final;
    virtual void handleRequest(AsyncWebServerRequest *request) override final;


    //  messagebuffer functions/objects.
    AsyncWebSocketMessageBuffer * makeBuffer(size_t size = 0);
    AsyncWebSocketMessageBuffer * makeBuffer(uint8_t * data, size_t size);
    LinkedList<AsyncWebSocketMessageBuffer *> _buffers;
    void _cleanBuffers();

    AsyncWebSocketClientLinkedList getClients() const;
};
```



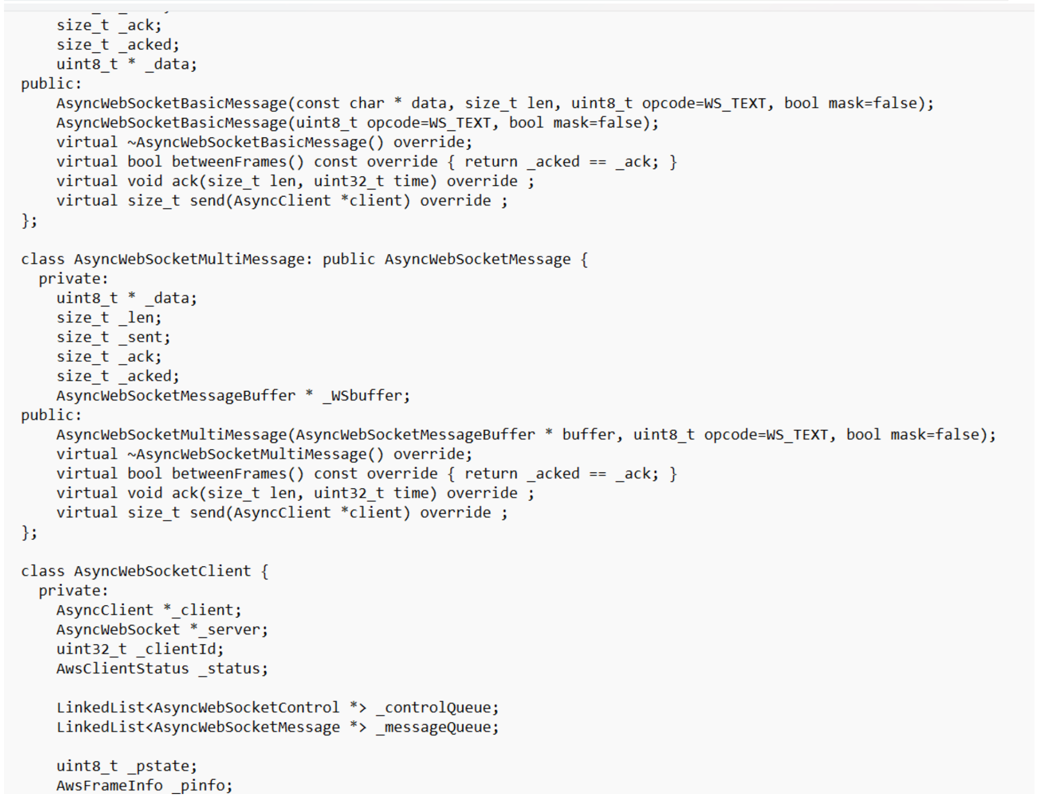
```
    AsyncWebSocket(const String& url);
    ~AsyncWebSocket();
    const char * url() const { return _url.c_str(); }
    void enable(bool e){ _enabled = e; }
    bool enabled() const { return _enabled; }
    bool availableForWriteAll();
    bool availableForWrite(uint32_t id);

    size_t count() const;
    AsyncWebSocketClient * client(uint32_t id);
    bool hasClient(uint32_t id){ return client(id) != NULL; }

    void close(uint32_t id, uint16_t code=0, const char * message=NULL);
    void closeAll(uint16_t code=0, const char * message=NULL);
    void cleanupClients(uint16_t maxClients = DEFAULT_MAX_WS_CLIENTS);

    void ping(uint32_t id, uint8_t *data=NULL, size_t len=0);
    void pingAll(uint8_t *data=NULL, size_t len=0); //  done

    void text(uint32_t id, const char * message, size_t len);
    void text(uint32_t id, const char * message);
    void text(uint32_t id, uint8_t * message, size_t len);
    void text(uint32_t id, char * message);
    void text(uint32_t id, const String &message);
    void text(uint32_t id, const __FlashStringHelper *message);

    void textAll(const char * message, size_t len);
    void textAll(const char * message);
    void textAll(uint8_t * message, size_t len);
    void textAll(char * message);
    void textAll(const String &message);
    void textAll(const __FlashStringHelper *message); //  need to convert
    void textAll(AsyncWebSocketMessageBuffer * buffer);

    void binary(uint32_t id, const char * message, size_t len);
    void binary(uint32_t id, const char * message);
    void binary(uint32_t id, uint8_t * message, size_t len);
    void binary(uint32_t id, char * message);
    void binary(uint32_t id, const String &message);
    void binary(uint32_t id, const __FlashStringHelper *message, size_t len);
```



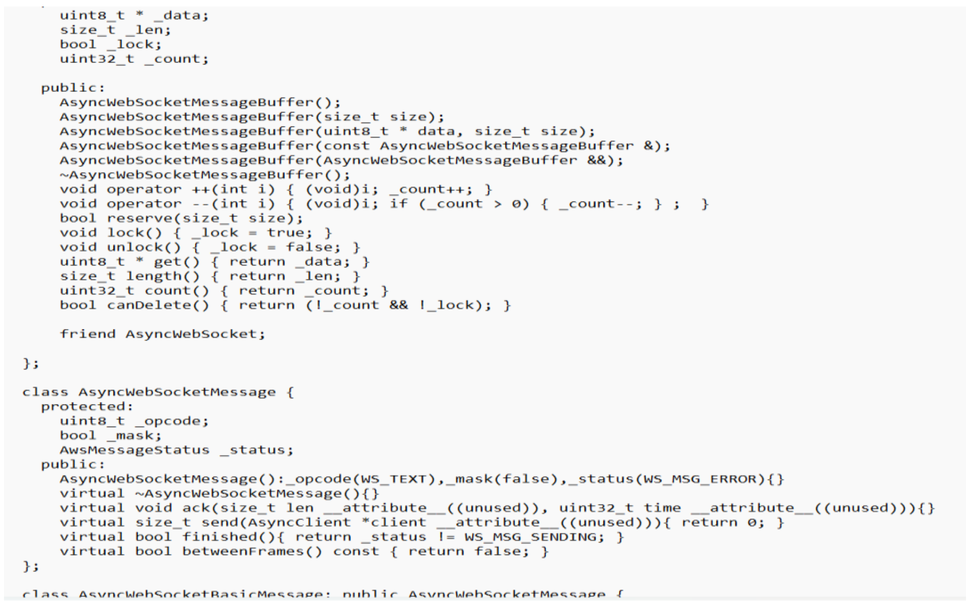
```
    size_t printf_P(PGM_P formatP, ...)  __attribute__ ((format (printf, 2, 3)));
#endif
    void text(const char * message, size_t len);
    void text(const char * message);
    void text(uint8_t * message, size_t len);
    void text(char * message);
    void text(const String &message);
    void text(const __FlashStringHelper *data);
    void text(AsyncWebSocketMessageBuffer *buffer);

    void binary(const char * message, size_t len);
    void binary(const char * message);
    void binary(uint8_t * message, size_t len);
    void binary(char * message);
    void binary(const String &message);
    void binary(const __FlashStringHelper *data, size_t len);
    void binary(AsyncWebSocketMessageBuffer *buffer);

    bool canSend() { return _messageQueue.length() < WS_MAX_QUEUED_MESSAGES; }

    //system callbacks (do not call)
    void _onAck(size_t len, uint32_t time);
    void _onError(int8_t);
    void _onPoll();
    void _onTimeout(uint32_t time);
    void _onDisconnect();
    void _onData(void *pbuf, size_t plen);
};

typedef std::function<void(AsyncWebSocket * server, AsyncWebSocketClient * client, AwsEventType type, void * arg, uint8_t *data, size_t len)> AwsEventHandler;

//WebServer Handler implementation that plays the role of a socket server
class AsyncWebSocket: public AsyncWebHandler {
  public:
    typedef LinkedList<AsyncWebSocketClient *> AsyncWebSocketClientLinkedList;
  private:
    String _url;
    AsyncWebSocketClientLinkedList _clients;
    uint32_t _cNextId;
    AwsEventHandler _eventHandler;
    bool _enabled;
```



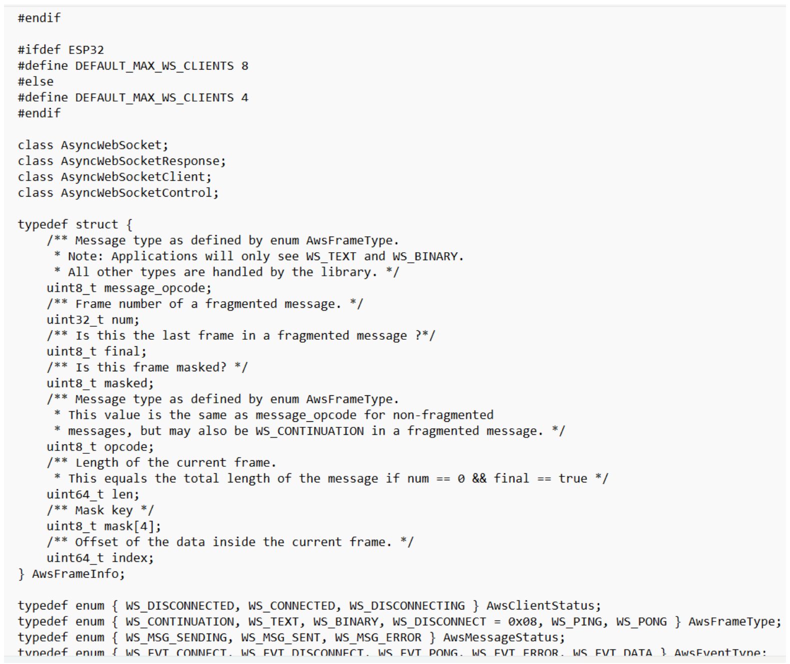
```
    uint8_t _pstate;
    AwsFrameInfo _pinfo;

    uint32_t _lastMessageTime;
    uint32_t _keepAlivePeriod;

    void _queueMessage(AsyncWebSocketMessage *dataMessage);
    void _queueControl(AsyncWebSocketControl *controlMessage);
    void _runQueue();

  public:
    void *_tempObject;

    AsyncWebSocketClient(AsyncWebServerRequest *request, AsyncWebSocket *server);
    ~AsyncWebSocketClient();

    //client id increments for the given server
    uint32_t id(){ return _clientId; }
    AwsClientStatus status(){ return _status; }
    AsyncClient* client(){ return _client; }
    AsyncWebSocket *server(){ return _server; }
    AwsFrameInfo const &pinfo() const { return _pinfo; }

    IPAddress remoteIP();
    uint16_t  remotePort();

    //control frames
    void close(uint16_t code=0, const char * message=NULL);
    void ping(uint8_t *data=NULL, size_t len=0);

    //set auto-ping period in seconds. disabled if zero (default)
    void keepAlivePeriod(uint16_t seconds){
      _keepAlivePeriod = seconds * 1000;
    }
    uint16_t keepAlivePeriod(){
      return (uint16_t)(_keepAlivePeriod / 1000);
    }

    //data packets
    void message(AsyncWebSocketMessage *message){ _queueMessage(message); }
    bool queueIsFull();
```



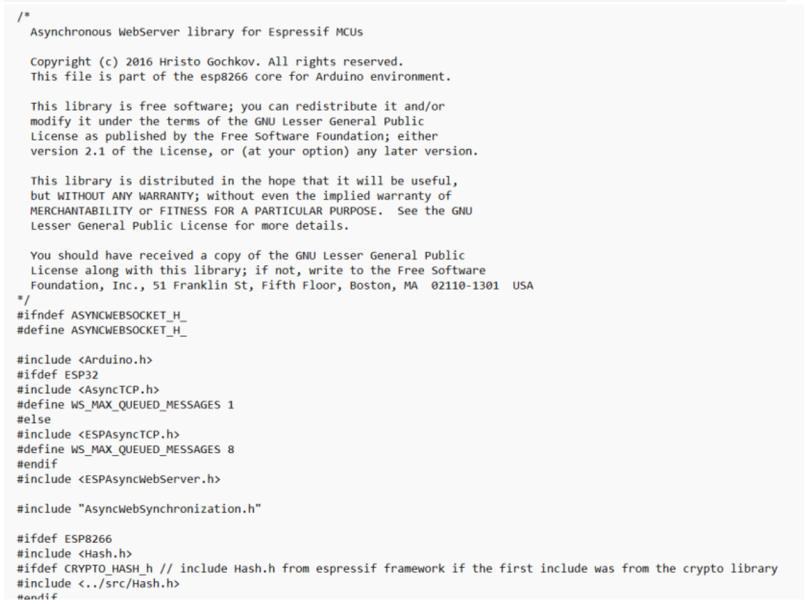
```
    size_t _ack;
    size_t _acked;
    uint8_t * _data;
public:
    AsyncWebSocketBasicMessage(const char * data, size_t len, uint8_t opcode=WS_TEXT, bool mask=false);
    AsyncWebSocketBasicMessage(uint8_t opcode=WS_TEXT, bool mask=false);
    virtual ~AsyncWebSocketBasicMessage() override;
    virtual bool betweenFrames() const override { return _acked == _ack; }
    virtual void ack(size_t len, uint32_t time) override ;
    virtual size_t send(AsyncClient *client) override ;
};

class AsyncWebSocketMultiMessage: public AsyncWebSocketMessage {
  private:
    uint8_t * _data;
    size_t _len;
    size_t _sent;
    size_t _ack;
    size_t _acked;
    AsyncWebSocketMessageBuffer * _WSbuffer;
public:
    AsyncWebSocketMultiMessage(AsyncWebSocketMessageBuffer * buffer, uint8_t opcode=WS_TEXT, bool mask=false);
    virtual ~AsyncWebSocketMultiMessage() override;
    virtual bool betweenFrames() const override { return _acked == _ack; }
    virtual void ack(size_t len, uint32_t time) override ;
    virtual size_t send(AsyncClient *client) override ;
};

class AsyncWebSocketClient {
  private:
    AsyncClient *_client;
    AsyncWebSocket *_server;
    uint32_t _clientId;
    AwsClientStatus _status;

    LinkedList<AsyncWebSocketControl *> _controlQueue;
    LinkedList<AsyncWebSocketMessage *> _messageQueue;

    uint8_t _pstate;
    AwsFrameInfo _pinfo;
```



```
    uint8_t * _data;
    size_t _len;
    bool _lock;
    uint32_t _count;

  public:
    AsyncWebSocketMessageBuffer();
    AsyncWebSocketMessageBuffer(size_t size);
    AsyncWebSocketMessageBuffer(uint8_t * data, size_t size);
    AsyncWebSocketMessageBuffer(const AsyncWebSocketMessageBuffer &);
    AsyncWebSocketMessageBuffer(AsyncWebSocketMessageBuffer &&);
    ~AsyncWebSocketMessageBuffer();
    void operator ++(int i) { (void)i; _count++; }
    void operator --(int i) { (void)i; if (_count > 0) { _count--; } ;  }
    bool reserve(size_t size);
    void lock() { _lock = true; }
    void unlock() { _lock = false; }
    uint8_t * get() { return _data; }
    size_t length() { return _len; }
    uint32_t count() { return _count; }
    bool canDelete() { return (!_count && !_lock); }

    friend AsyncWebSocket;

};

class AsyncWebSocketMessage {
  protected:
    uint8_t _opcode;
    bool _mask;
    AwsMessageStatus _status;
  public:
    AsyncWebSocketMessage():_opcode(WS_TEXT),_mask(false),_status(WS_MSG_ERROR){}
    virtual ~AsyncWebSocketMessage(){}
    virtual void ack(size_t len __attribute__((unused)), uint32_t time __attribute__((unused))){}
    virtual size_t send(AsyncClient *client __attribute__((unused))){ return 0; }
    virtual bool finished(){ return _status != WS_MSG_SENDING; }
    virtual bool betweenFrames() const { return false; }
};

class AsyncWebSocketBasicMessage: public AsyncWebSocketMessage {
```

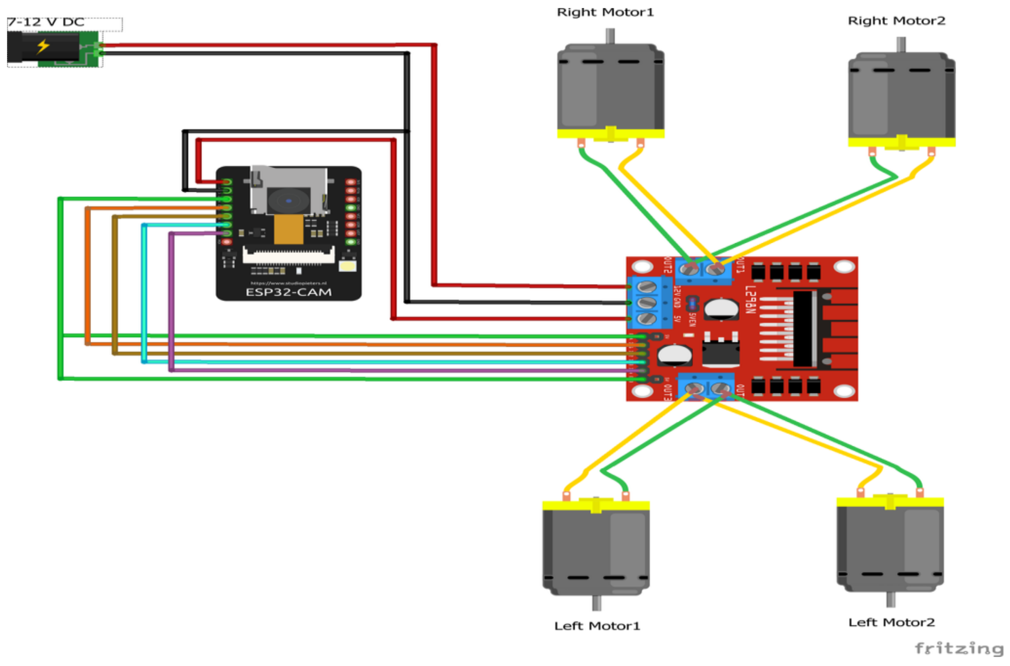


```
#endif

#ifdef ESP32
#define DEFAULT_MAX_WS_CLIENTS 8
#else
#define DEFAULT_MAX_WS_CLIENTS 4
#endif

class AsyncWebSocket;
class AsyncWebSocketResponse;
class AsyncWebSocketClient;
class AsyncWebSocketControl;

typedef struct {
    /** Message type as defined by enum AwsFrameType.
     * Note: Applications will only see WS_TEXT and WS_BINARY.
     * All other types are handled by the library. */
    uint8_t message_opcode;
    /** Frame number of a fragmented message. */
    uint32_t num;
    /** Is this the last frame in a fragmented message ?*/
    uint8_t final;
    /** Is this frame masked? */
    uint8_t masked;
    /** Message type as defined by enum AwsFrameType.
     * This value is the same as message_opcode for non-fragmented
     * messages, but may also be WS_CONTINUATION in a fragmented message. */
    uint8_t opcode;
    /** Length of the current frame.
     * This equals the total length of the message if num == 0 && final == true */
    uint64_t len;
    /** Mask key */
    uint8_t mask[4];
    /** Offset of the data inside the current frame. */
    uint64_t index;
} AwsFrameInfo;

typedef enum { WS_DISCONNECTED, WS_CONNECTED, WS_DISCONNECTING } AwsClientStatus;
typedef enum { WS_CONTINUATION, WS_TEXT, WS_BINARY, WS_DISCONNECT = 0x08, WS_PING, WS_PONG } AwsFrameType;
typedef enum { WS_MSG_SENDING, WS_MSG_SENT, WS_MSG_ERROR } AwsMessageStatus;
typedef enum { WS_EVT_CONNECT, WS_EVT_DISCONNECT, WS_EVT_PONG, WS_EVT_ERROR, WS_EVT_DATA } AwsEventType;
```



```
/*
  Asynchronous WebServer library for Espressif MCUs

  Copyright (c) 2016 Hristo Gochkov. All rights reserved.
  This file is part of the esp8266 core for Arduino environment.

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
*/
#ifndef ASYNCWEBSOCKET_H_
#define ASYNCWEBSOCKET_H_

#include <Arduino.h>
#ifdef ESP32
#include <AsyncTCP.h>
#define WS_MAX_QUEUED_MESSAGES 1
#else
#include <ESPAsyncTCP.h>
#define WS_MAX_QUEUED_MESSAGES 8
#endif
#include <ESPAsyncWebServer.h>

#include "AsyncWebSynchronization.h"

#ifdef ESP8266
#include <Hash.h>
#ifdef CRYPTO_HASH_h // include Hash.h from espressif framework if the first include was from the crypto library
#include <../src/Hash.h>
#endif
```

CIRCUIT DIAGRAM:



WORKING MODEL: