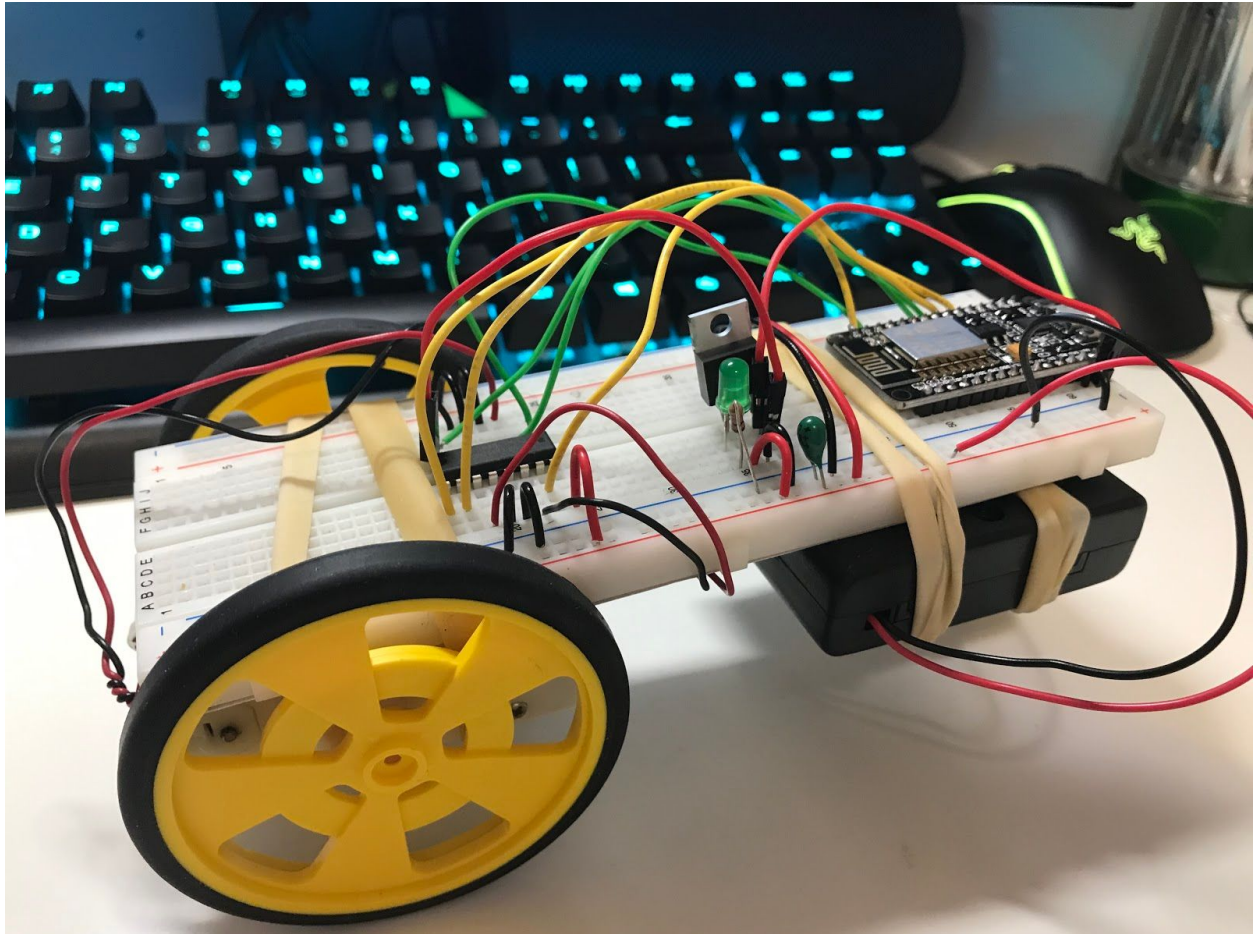


# NodeMCU RC Car

## *Software Implementation Guide*

ShangJin Li  
shli001@uw.edu



*Image Credit: S.J Li*

### **Introduction**

In this tutorial, we will be implementing the software for ESP8266-12 module to control a toy car. We will first investigate the structure and implementation of the Arduino code for the module, then we will proceed to the C# application that controls the car from Windows OS. Finally, we close with the source code for an Android app that serve as a reference for how similar ideas can be applied in Android app development.

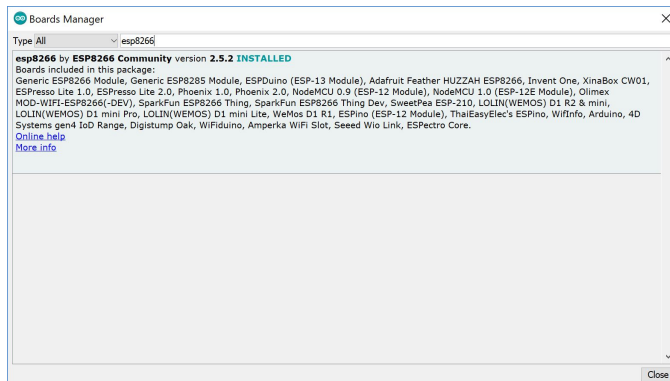
## **Part 1: Arduino Code**

### **Overview**

In this part of the tutorial, we will be implementing the Arduino code for the ESP8266 module. In order to upload the code to our module successfully, we first need to setup Arduino IDE for ESP8266-12. Then, we will go over the logic behind how the code works.

### **Setup**

1. In Arduino IDE, go to File->Preferences
2. On [Addition Board Manager Urls], paste in [http://arduino.esp8266.com/stable/package\\_esp8266com\\_index.json](http://arduino.esp8266.com/stable/package_esp8266com_index.json)
3. Click OK to save and exit the preference window
4. Click Tools->Board->Board Manager
5. Search for ESP8266 and install



6. Close the Boards Manager window
7. Go to Tools->Boards
8. Select "NodeMCU 1.0 (ESP-12E Module) as your board
9. Connect your ESP8266 with your computer, try to upload an empty sketch file and see if it works.

### **Code**

The complete project can be downloaded here: <https://github.com/miska12345/ESP8266RC>

## Code Explanation (1 of 6)

```
#include <ESP8266WiFi.h>

// Pulse-width-modulation pins
// These pins allow you to control motor speed
#define PWM1 D1
#define PWM2 D2

// General Purpose Input/Output
// Control motor on/off and direction
#define A0 D8
#define A1 D7
#define A2 D6
#define A3 D5
```

In the first part of the code we include the library required to initialize the ESP module and define the names for I/O pins that we will use.

```
// Server config
// Port to listen for connection
// See https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html
#define TCP_PORT 12345

// WiFi configuration
static const char *wfName = "name";           // Your Wi-Fi name
static const char *wfPass = "pass";          // Your Wi-Fi password
```

We then define the TCP port for ESP8266's WiFiServer (the port that it will listen for incoming connections). Then we set the name and password of our access point.

```
// InsCode Definition
// InsCode are all instruction send by remote client (i.e. App)
// Values defined here are of type String
// Other encodings, such as integer, are possible
#define INS_FORWARD_PRESSED "FORWARD_P"
#define INS_BACKWARD_PRESSED "BACKWARD_P"
#define INS_LEFT_PRESSED "LEFT_P"
#define INS_LEFT_RELEASED "LEFT_R"
#define INS_RIGHT_PRESSED "RIGHT_P"
#define INS_RIGHT_RELEASED "RIGHT_R"
#define INS_NULL "STOP"
```

Next, we define a set of instructions that we want the car to handle.

```

// Data Field
// GPIOMatrix is a multi-level array
// It is equivalent to uint8_t[2][2]
// Matrix view: +-----+
//               +-A0-----A1-+
//               +-A2-----A3-+
//               +-----+
typedef uint8_t GPIOMatrix[2][2];

// Direction is an abstraction for the direction of motors
// There are 3 states for the car's Direction
// 1. Forward: State where motors roll forward (relative)
// 2. Backward: State where motors roll backward (relative)
// 3. None: An idle state, also a default state
enum Direction {FORWARD, BACKWARD, NONE };

// Struct is a bundle of data
// CarMgr is a name we use for the struct manager_st
// This struct currently holds a field of type Direction
// It is considered best practice to place related data together in a data structure such as a struct
// Read more: https://www.geeksforgeeks.org/structures-c/
typedef struct manager_st {
    Direction DIR;
} CarMgr;

```

GPIOMatrix is a 2x2 uint8\_t array. Typedef makes it so that we can use GPIOMatrix in places where uint8\_t[2][2] is expected. The matrix view of GPIOMatrix tell us that A[0][0] correspond to value at I/O pin A0, A[0][1] correspond to A1...etc. Direction is an enum with three values, it is defined to represent direction of our car. CarMgr is a structure, and structure is just a bundle of related data. It holds the properties of our car. It is considered a best practice to group related properties of the car together. In case you wish to extend the functionality of this code, new properties of the car (such as speed) should be added to CarMgr to avoid confusion with other variables.

```

CarMgr *Manager; // A pointer to a CarMgr (which we defined earlier)
static WiFiServer Sserver(TCP_PORT); // NodeMCU's server instance, the provided TCP_PORT is the port we listen for incoming connections

// These GPIOMatrices are pre-defined to help ease programming
static const GPIOMatrix MATRIX_STOP = {LOW, LOW, LOW, LOW};
static const GPIOMatrix MATRIX_FORWARD = {HIGH, LOW, HIGH, LOW};
static const GPIOMatrix MATRIX_BACKWARD = {LOW, HIGH, LOW, HIGH};
static const GPIOMatrix MATRIX_FORWARD_LEFT = {LOW, LOW, HIGH, LOW};
static const GPIOMatrix MATRIX_FORWARD_RIGHT = {HIGH, LOW, LOW, LOW};
static const GPIOMatrix MATRIX_BACKWARD_LEFT = {LOW, LOW, LOW, HIGH};
static const GPIOMatrix MATRIX_BACKWARD_RIGHT = {LOW, HIGH, LOW, LOW};

```

Here we create two variables. Manager is a pointer to a CarMgr struct. A pointer is an address in memory where we can find data. Server is a WiFiServer instance. It represents the TCP server that we will use later in the code. All the MATRIX\_XXX are predefined GPIOMatrix. They help reduce redundancy when we change the direction of the car.



## Code Explanation (2 of 6)

### *Setup*

```
void setup() {  
    Serial.begin(9600);    // for Serial output  
    WiFiInitialize();     // Initialize the WiFi module  
    NodeMCUInitialize();   // Initialize the pins  
    Initialize();          // Initialize data structures  
    Serial.println("Initialization completed");  
}
```

Here we start serial communication and select the baud rate. We call Initializer methods to initialize the car, and then print a message to serial when finished.

## Code Explanation (3 of 6)

### *Initialization*

```
void WiFiInitialize() {  
    WiFi.begin(wfName, wfPass);  
  
    while (WiFi.status() != WL_CONNECTED) {  
        delay(500);  
    }  
    Serial.println("Connected!");  
    Serial.println(WiFi.localIP()); // Print IP to console  
    Server.begin();  
}  
  
void NodeMCUInitialize() {  
    pinMode(A0, OUTPUT);  
    pinMode(A1, OUTPUT);  
    pinMode(A2, OUTPUT);  
    pinMode(A3, OUTPUT);  
    pinMode(PWM1, OUTPUT);  
    pinMode(PWM2, OUTPUT);  
    analogWrite(PWM1, 1024);  
    analogWrite(PWM2, 1024);  
}  
  
void Initialize() {  
    Manager = new CarMgr;  
    if (Manager == NULL) {  
        Serial.println("Not enough RAM");  
        return;  
    }  
    Manager->DIR = Direction::NONE;  
}
```

WiFiInitialize() initializes WiFi related values. First it connect to the WiFi with name given by wfName and password given by wfPass. Then it starts the server. NodeMCUInitialize() define pin modes. Initialize() initialize our Manager pointer and set the default direction to None, meaning the car is not moving. Note that 'new' is a keyword in the C++ language used to allocate heap data. In this case, we allocate a CarMgr object on the heap and save the address of where that data is located to Manager.

### Code Explanation (3 of 6)

#### *Loop*

```
void loop() {
  WiFiClient c = Server.available();
  if (c) {
    // There is a connection
    while (c.connected()) {
      // 1. While the client is connected, we continuously receive data from the client
      while (c.available() > 0) {
        // 2. As soon as we reach the end of instruction (by seeing \r), we handle the instruction
        String data = c.readStringUntil('\r');
        InterpretIns(data);
      }
      // Go back to 1 if client is still online
    }
    // Client is offline, stop the motors
    InterpretIns(INS_NULL);
  }
}
```

In the loop section, we constantly check if a client is connected to ESP8266. If so, we receive instructions from the client and call InterpretIns to process it. If the client disconnected, we tell InterpretIns to process a STOP instruction so the car automatically stops when no one is controlling it.

## Code Explanation (4 of 6)

### *InterpretIns*

```
void InterpretIns(String ins) {
  // This function check to see if ins match any of the known instructions
  // If there is a matching instruction, motor states will be updated based on the instruction
  GPIOMatrix matrix; // A scheme, used to hold matrix used in the function

  // Note: str.indexOf(b) is a function that returns positive value if a substring (b) appears in str, and -1 otherwise
  // By checking if the return value is not -1, we are certain that the instruction can be interpreted this way
  if (ins.indexOf(INS_FORWARD_PRESSED) != -1) {
    Set(matrix, MATRIX_FORWARD);
    Manager->DIR = Direction::FORWARD;
  } else if (ins.indexOf(INS_BACKWARD_PRESSED) != -1) {
    Set(matrix, MATRIX_BACKWARD);
    Manager->DIR = Direction::BACKWARD;
  } else if (ins.indexOf(INS_LEFT_PRESSED) != -1) {
    if (Manager->DIR != Direction::NONE) {
      Manager->DIR == Direction::FORWARD ? Set(matrix, MATRIX_FORWARD_LEFT) : Set(matrix, MATRIX_BACKWARD_LEFT);
    }
  } else if (ins.indexOf(INS_RIGHT_PRESSED) != -1) {
    if (Manager->DIR != Direction::NONE) {
      Manager->DIR == Direction::FORWARD ? Set(matrix, MATRIX_FORWARD_RIGHT) : Set(matrix, MATRIX_BACKWARD_RIGHT);
    }
  } else if (ins.indexOf(INS_LEFT_RELEASED) != -1 || ins.indexOf(INS_RIGHT_RELEASED) != -1) {
    if (Manager->DIR != Direction::NONE) {
      Manager->DIR == Direction::FORWARD ? Set(matrix, MATRIX_FORWARD) : Set(matrix, MATRIX_BACKWARD);
    }
  } else {
    // Unknown instructions are treated as INS_NULL
    Set(matrix, MATRIX_STOP);
    Manager->DIR = Direction::NONE;
  }
  Apply(matrix); // Apply the current GPIO scheme to NodeMCU
}
```

InterpretIns is the core of remote control in this project. Given a string parameter ‘ins’, the function match ins with the known set of instructions. If a match is found, the function proceed to execute the instruction. For direction related instructions, we use the predefined GPIOMatrix to update direction for the wheels. If no match is found, an invalid instruction would cause the car to stop moving.

## Code Explanation (5 of 6)

### *Apply*

```
void Apply(const GPIOMatrix matrix) {
  digitalWrite(A0, matrix[0][0]);
  digitalWrite(A1, matrix[0][1]);
  digitalWrite(A2, matrix[1][0]);
  digitalWrite(A3, matrix[1][1]);
}
```

This function turns the value stored in GPIOMatrix into pin output. The convention on how this works has been covered when we typedef GPIOMatrix.

### Code Explanation (6 bof 6)

*Set*

```
void Set(GPIOMatrix matrix, const GPIOMatrix copyFrom) {  
    matrix[0][0] = copyFrom[0][0];  
    matrix[0][1] = copyFrom[0][1];  
    matrix[1][0] = copyFrom[1][0];  
    matrix[1][1] = copyFrom[1][1];  
}
```

---

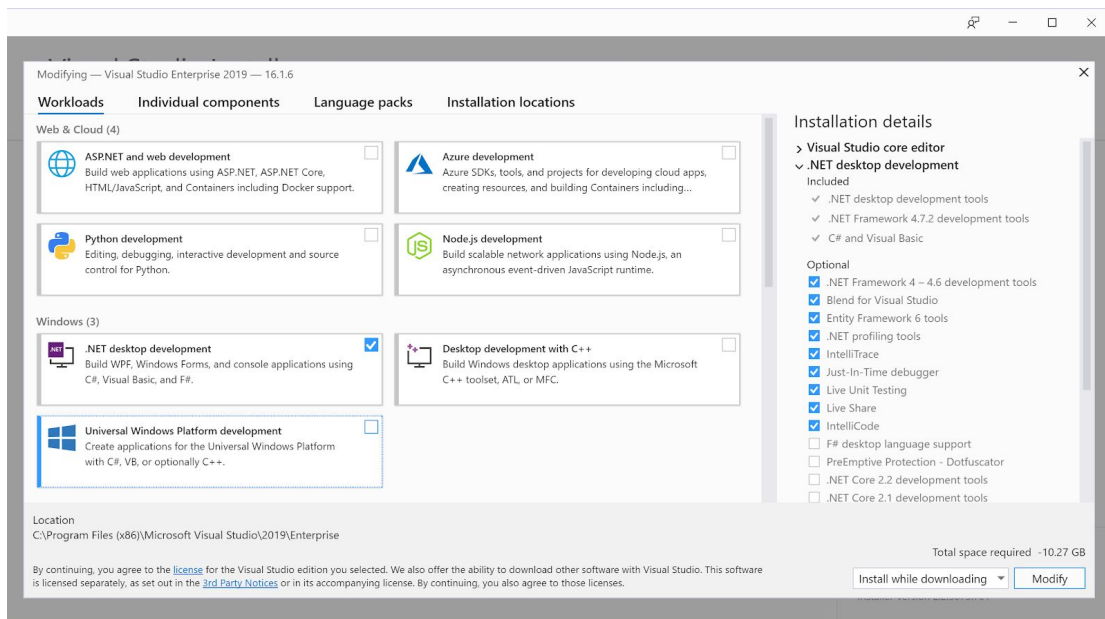
This is a helper function that is used to copy a given matrix.



## Part 2: C#

### Setup

1. Register for a Microsoft account (if you don't have one)
2. Download Visual Studio 2019 Community
  - ★ Visual Studio is an Integrated Development Environment for developing Windows applications. The Community version is free.
3. Launch the file you have downloaded
4. Login to your account as prompted
5. Install C# (Select what I have selected here) and save



6. Once the installation is done, run Visual Studio 2019. If you cannot find it on Desktop, try to search for it by clicking on the Windows icon and type Visual Studio.
7. By now you should have the Visual Studio IDE running.

### Crash Course - What is TCP and Socket?

The Transmission Control Protocol(TCP) is a protocol for one-to-one connections. In this context, socket refers to an internal endpoint for sending and receiving data on a computer network. It is associated with a specific socket address, namely the IP address and a port number.

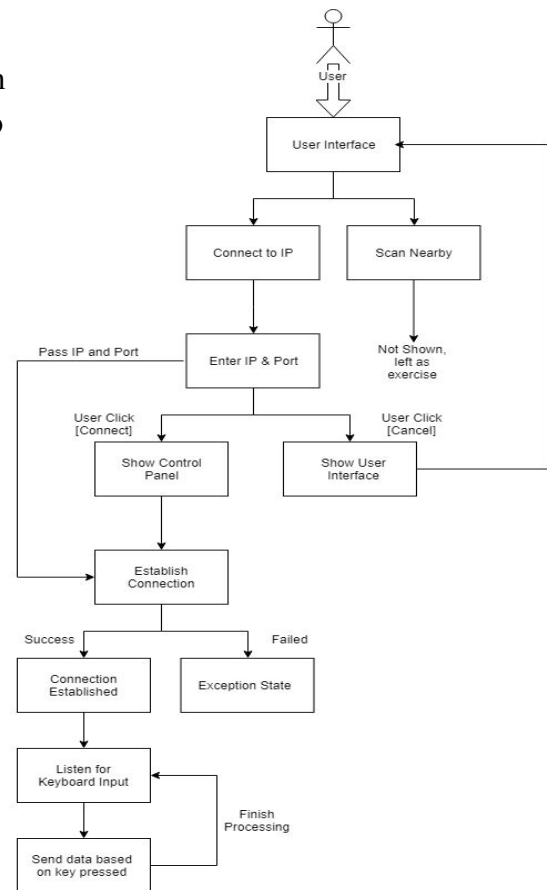
Trivially speaking, TCP is a convention that networking devices follow in order to “talk” to each other. Socket, in the context of TCP, represent one of the devices in the “conversation” and is labeled by an IP address and a port number. Socket allow us to send/receive data from/to a remote device. It is the underlying mechanism for all network-related operations such as browsing the Internet and playing online games.

## Structure and Design

Unlike Arduino code, C# projects tend to benefit from modular software design. That is, increase cohesion and decrease coupling between different components. To start, we will go over a flow-chart representation of the program that we will be implementing, and discuss how different modules (classes) interact with each other.

Notice that each component (box) have a single, well-defined purpose. The benefit of modular design is that it allows us to localize bugs in case some components fail to execute. Modularity is generally followed by industrial-level software engineers to prevent bugs from propagating into other parts of the software system, and is the reason why we practice this convention.

Some components, such as User Interface and Control Panel, have an associated Graphical User Interface (GUI). GUI is a visual representation of the C# WinForm class which we get to design. For this project, we will use a third party library called MetroModernUI to help us make a compelling user interface.



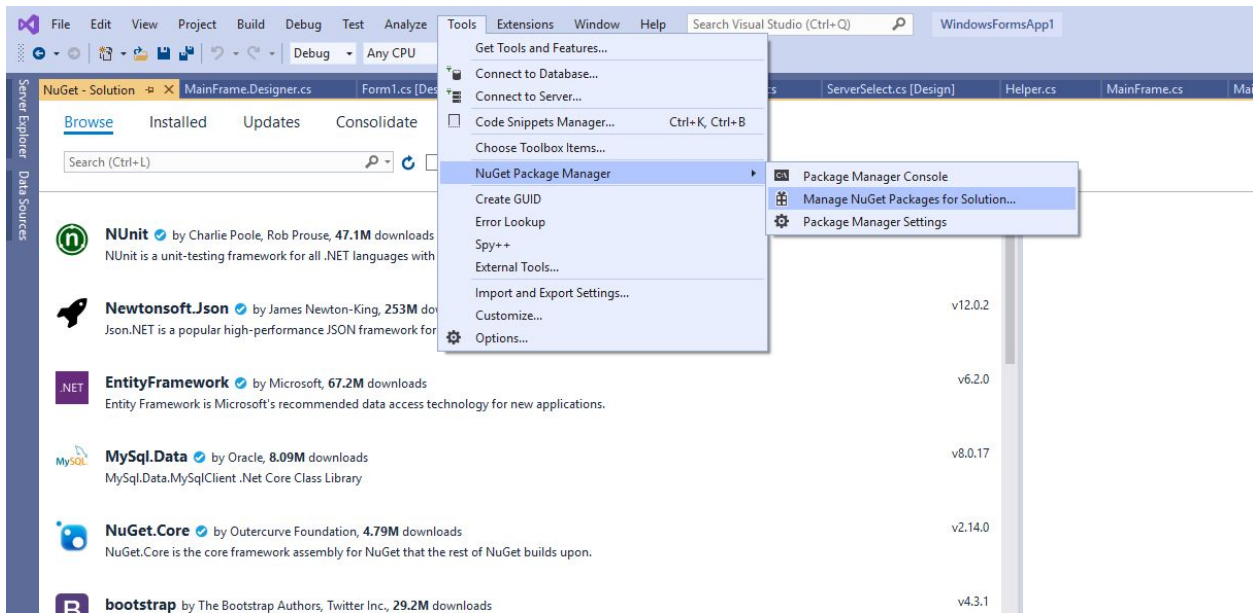
## Install MetroFramework (Optional)

MetroFramework is a third party library that helps you make compelling user-interface. The source files that you downloaded should have it installed. In case you are curious, to install MetroModernUI, we need to use the NuGet tool in Visual Studio (VS).

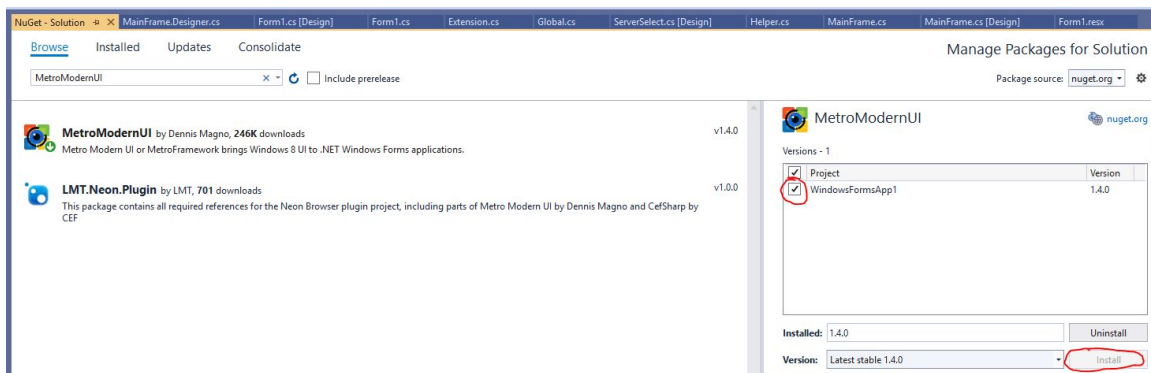
1. Open the project file you have downloaded (the .sln file) with VS

packages	7/17/2019 11:07 PM	File folder	
WindowsFormsApp1	7/24/2019 10:47 PM	File folder	
WindowsFormsApp1.sln	7/17/2019 11:06 PM	Visual Studio Solu...	2 KB

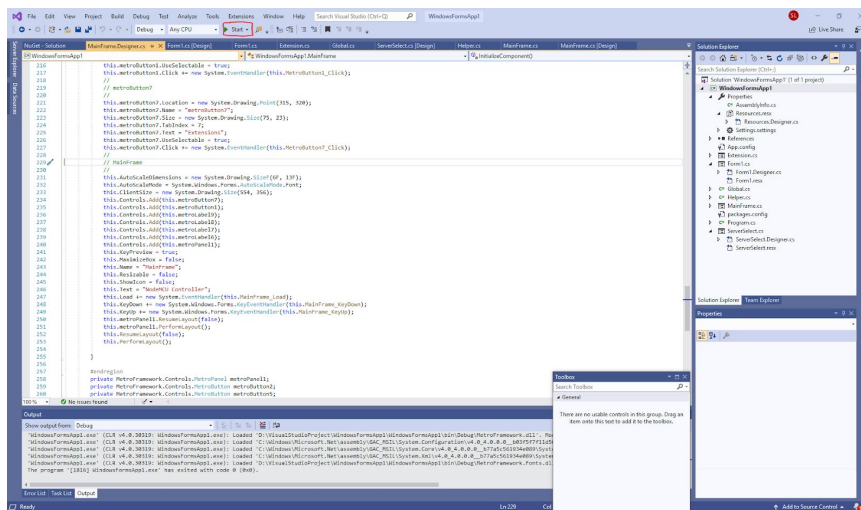
2. You can find NuGet by going to Tools->NuGet Package Manager->Manage NuGet Package for Solution.



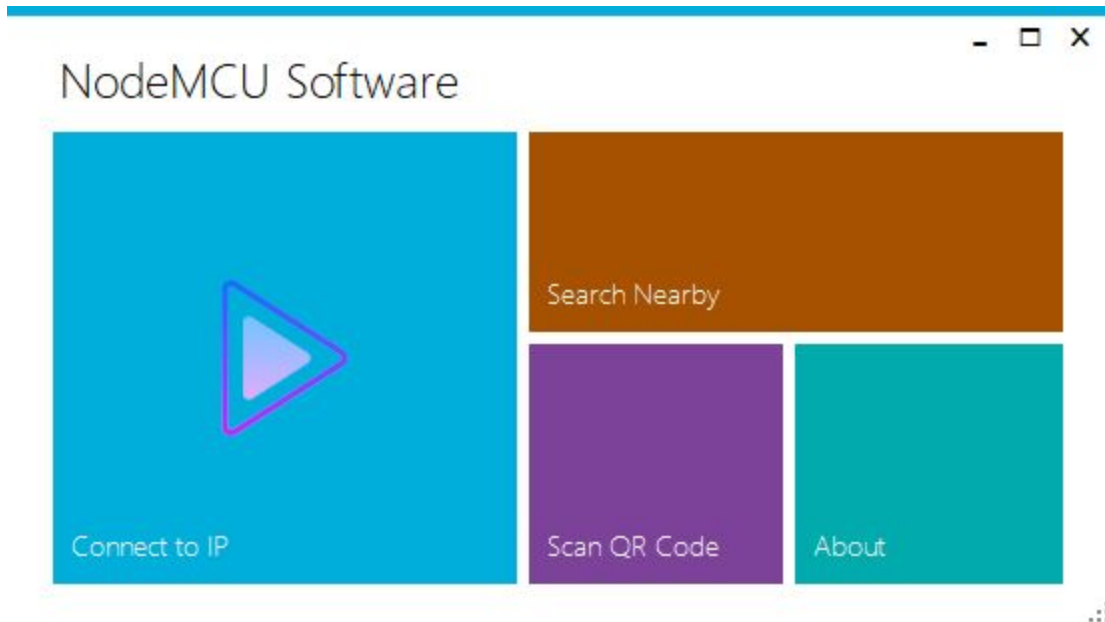
3. Search for MetroModernUI once you are in the NuGet window
4. Install the package to your solution



5. Try to run the code and see if it compiles

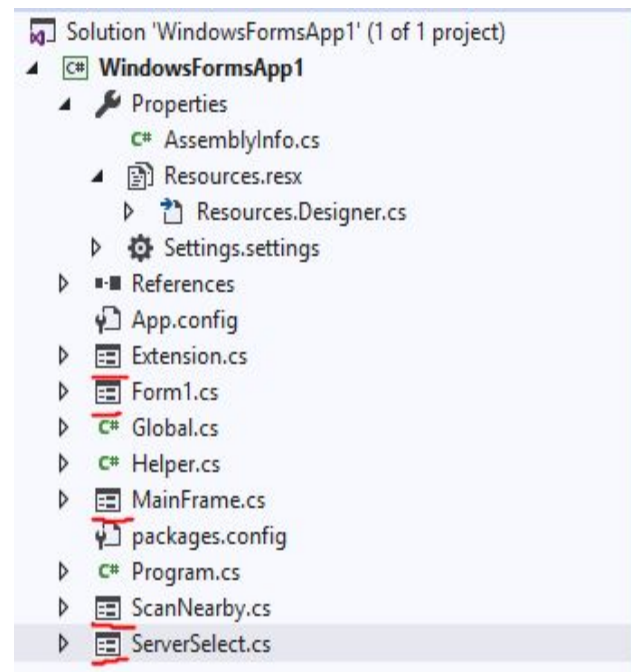


6. This is the user-interface for our application! If you see this, we can move on. (p.s you may exit the NodeMCU Software by clicking on the 'X' or click the red square (stop debugging) in VS)



### Navigation:

You can navigate between different components of the program by clicking on the items in the solution explorer. If you do not see this window, go to View->Solution Explorer. The items with a red underline are components with GUI. Again, if you click into it, VS will bring you to a GUI editor where you can edit the interface. **For source code associated with a particular GUI component, go to the GUI editor by double clicking any items with red underline, and then double click anywhere in the GUI editor to go to code view.**



### Overview

In this part of the tutorial, we will be implementing the C# application that controls the car.

## Code

The complete C# project can be downloaded here: <https://github.com/miska12345/ESP8266RC>

### Code Explanation (part 1 of 10)

*Form1.cs (User Interface)*

```
11 namespace WindowsFormsApp1
12 {
13     4 references
14     public partial class Form1 : MetroFramework.Forms.MetroForm
15     {
16         1 reference
17         public Form1()
18         {
19             InitializeComponent();
20         }
21
22         1 reference
23         private void Form1_Load(object sender, EventArgs e)
24         {
25             this.Theme = MetroFramework.MetroThemeStyle.Dark;
26         }
27
28         1 reference
29         private void MetroTile1_MouseClick(object sender, MouseEventArgs e)
30         {
31             ServerSelect window = new ServerSelect();
32             window.Show();
33         }
34
35         1 reference
36         private void MetroTile2_Click(object sender, EventArgs e)
37         {
38             ScanNearby window = new ScanNearby();
39             window.Show();
40         }
41     }
42 }
```

This part of the code deals with the features of User Interface (first block in the flow-chart). At line 11, we define a namespace called `WindowsFormsApp1`. A namespace is simply a declarative space that provides scope to the identifiers inside it. Line 25 and 31 are event handlers called by the `MetroFramework` when user click on the corresponding tiles. These handlers define what happens when a user clicks on one of the tiles. In this case, we can see that when the user clicks on `MetroTile1` (the Connect tile), a window called `ServerSelect` will be shown. Similarly, when the user clicks on `MetroTile2` (the Scan Nearby tile), a window called `ScanNearby` will be shown.

Side note: C# is an object-oriented programming language. The declaration format `TYPE Name = new Type();` declares an object and store it in `window`.



## Code Explanation (part 2 of 10)

*ServerSelect.cs (Enter IP & Port)*

```
21 private void MetroButton2_Click(object sender, EventArgs e)
22 {
23     this.Close();
24 }
25
26 1 reference
27 private void MetroButton1_Click(object sender, EventArgs e)
28 {
29     String server = metroTextBox1.Text;
30     if (server.Length == 0)
31     {
32         MetroFramework.MetroMessageBox.Show(this, "Please enter a non-empty IP address");
33         return;
34     }
35     int port = -1;
36     int.TryParse(metroTextBox2.Text, out port);
37     if (port == -1)
38     {
39         MetroFramework.MetroMessageBox.Show(this, "Please enter a valid port number (numbers only)");
40         return;
41     }
42     MainFrame main = new MainFrame(server, port);
43     main.Closed += (s, args) => this.Close();
44     main.Show();
45     this.Hide();
46 }
47
```

This part of the code defines the behavior of our program when the user is in the ServerSelect window. Line 26 defines an event handler for MetroButton2 (the Close button), which will be called by MetroFramework when the user clicks on the button. Line 31 is our first 'big' function. Similar to line 26, it defines an event handler for the MetroButton1 (the Connect button). Line 33 to 44 deals with input checking. If a user input an empty IP address or a negative port number, a message box will pop up and tell the user to fix it. If no errors detected, the function proceed to create a MainFrame, which is another GUI component, and display to the user. In case you are curious, the `main.Closed += ...` part means close the current window when MainFrame exits, and `this.Hide()` hides the current window from the user.

### Code Explanation (3 of 10)

*MainFrame.cs (Control Panel)*

```
15 7 references
16 public partial class MainFrame : MetroFramework.Forms.MetroForm
17 {
18     // Instructions
19     private readonly static String INS_FORWARD_PRESSED = "FORWARD_P";
20     private readonly static String INS_BACKWARD_PRESSED = "BACKWARD_P";
21     private readonly static String INS_LEFT_PRESSED = "LEFT_P";
22     private readonly static String INS_LEFT_RELEASED = "LEFT_R";
23     private readonly static String INS_RIGHT_PRESSED = "RIGHT_P";
24     private readonly static String INS_RIGHT_RELEASED = "RIGHT_R";
25     private readonly static String INS_NULL = "STOP";
26
27     private readonly static String INS_HORN_PRESSED = "HORN_P";
28     private readonly static String INS_HORN_RELEASED = "HORN_R";
29
30     private String server;
31     private int port;
32
33 2 references
34 public MainFrame(String ip, int remote_port)
35 {
36     InitializeComponent();
37
38     // Establish connection
39     server = ip;
40     port = remote_port;
41 }
```

As the comment suggests, all the private readonly static fields are defined to match the corresponding instructions for our ESP8266 RC car. You may recall that these strings also appeared in the Arduino code. This is not a coincidence because these instructions are sent and received to make remote control possible. Line 30 and 31 define what is called a field variable. They exist in the instance of the class MainFrame. Line 33 defines the constructor for MainFrame. Constructor is just a function that gets called as soon as the class is created. Line 38 and 39 save the parameters for later use. Notice how we can pass parameters into constructors, this is done by `new MainFrame(server, port)` from our code at ServerSelect.cs.

## Code Explanation (4 of 10)

*MainFrame.cs (Control Panel)*

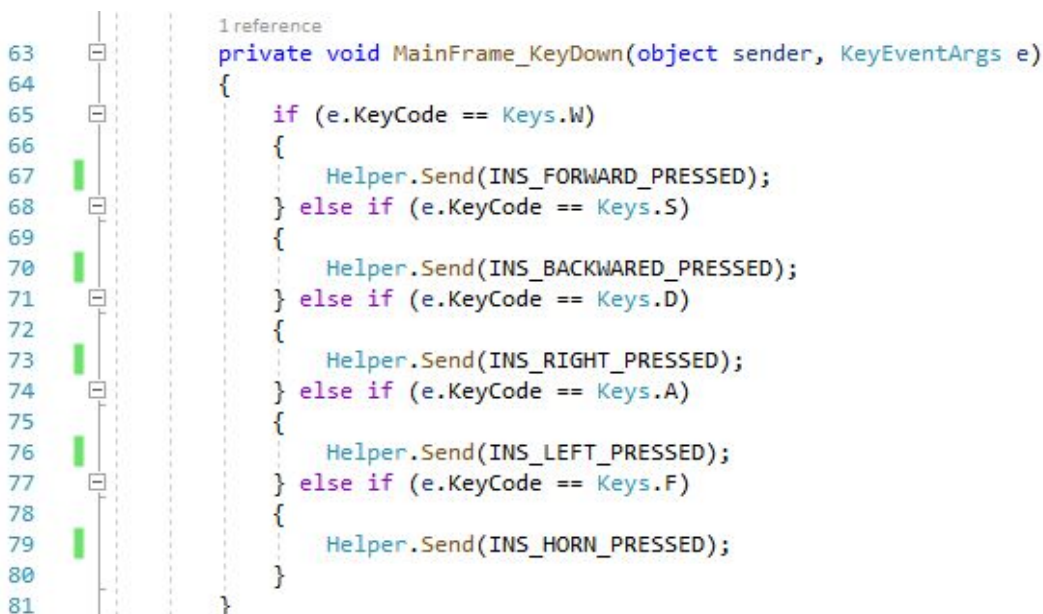


```
1 reference
42 private void MainFrame_Load(object sender, EventArgs e)
43 {
44     if (!Helper.PingHost(server))
45     {
46         MetroFramework.MetroMessageBox.Show(this, "Remote address is not responding");
47     } else
48     {
49         if (Helper.Start(server, port))
50         {
51             label_server.Text = server;
52             label_status.Text = "Connected";
53         } else
54         {
55             MetroFramework.MetroMessageBox.Show(this, "Connection cannot be established");
56             label_server.Text = "null";
57             label_status.Text = "Disconnected";
58         }
59     }
60 }
61 }
```

MainFrame\_Load is an event handler. Similar to the event handlers for buttons, this function is called when the GUI window has been loaded. In this function, we use Helper.PingHost(), a method provided in the Helper.cs, to check if the server is online. If so, we proceed to establish a connection using the TCP protocol. Helper.Start(server, port) is the function that establishes the connection. If a connection has been established, it will return true, otherwise it will return false. If you are interested, check out what is happening under the hood at Helper.cs. The exact details on how it works is outside the scope of this tutorial.

## Code Explanation (5 of 10)

*MainFrame.cs (Control Panel)*



```
1 reference
63 private void MainFrame_KeyDown(object sender, KeyEventArgs e)
64 {
65     if (e.KeyCode == Keys.W)
66     {
67         Helper.Send(INS_FORWARD_PRESSED);
68     } else if (e.KeyCode == Keys.S)
69     {
70         Helper.Send(INS_BACKWARD_PRESSED);
71     } else if (e.KeyCode == Keys.D)
72     {
73         Helper.Send(INS_RIGHT_PRESSED);
74     } else if (e.KeyCode == Keys.A)
75     {
76         Helper.Send(INS_LEFT_PRESSED);
77     } else if (e.KeyCode == Keys.F)
78     {
79         Helper.Send(INS_HORN_PRESSED);
80     }
81 }
```

MainFrame\_KeyDown is an event handler for the KeyDown event in Forms. This function is called when the user presses on any key on the keyboard. The KeyEventArgs contain information about which key is being pressed. We deal with user input by checking if the key is one of our hotkeys (namely, W, A, S, D, and F), and if so, we use Helper.Send(INSTRUCTION) to tell ESP8266 what has happened. As you may recall, the InterpretIns(...) function in Arduino then handles these instructions by checking if it matches one of the known instructions.

### Code Explanation (6 of 10)

MainFrame.cs (Control Panel)

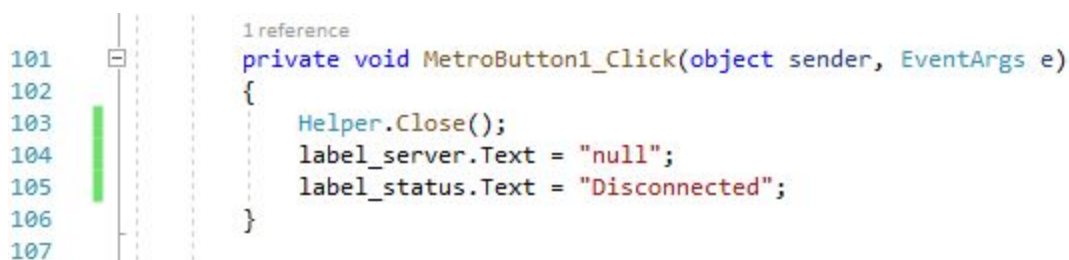


```
1 reference
private void MainFrame_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.W || e.KeyCode == Keys.S)
    {
        Helper.Send(INS_NULL);
    } else if (e.KeyCode == Keys.D)
    {
        Helper.Send(INS_RIGHT_RELEASED);
    } else if (e.KeyCode == Keys.A)
    {
        Helper.Send(INS_LEFT_RELEASED);
    } else if (e.KeyCode == Keys.F)
    {
        Helper.Send(INS_HORN_RELEASED);
    }
}
```

MainFrame\_Keyup is the counterpart of MainFrame\_KeyDown. It is an event handler for the KeyUp event in Forms. This function is called when the user releases a key on the keyboard. The KeyEventArgs contain information about which key is being pressed. Unlike in MainFrame\_KeyDown where we simply send key pressed instruction to ESP8266, this time we deal with a little bit of logic. Consider what happens when the user releases either W or S. The car should come to a stop because no direction is specified. This is why the first if statement handles an OR statement. It means if W OR S is released, we stop the car by sending INS\_NULL to ESP8266. The rest of the keys are self-explanatory.

### Code Explanation (7 of 10)

MainFrame.cs (Control Panel)



```
1 reference
private void MetroButton1_Click(object sender, EventArgs e)
{
    Helper.Close();
    label_server.Text = "null";
    label_status.Text = "Disconnected";
}
```

This function is yet another event handler. It handles what happen when a user clicks on the Disconnect button. In order to close a connection, we use `Helper.Close()` and then update the labels for our GUI.

## Code Explanation (8 of 10)

*Helper.cs*

```
36 9 references
37 public static bool Send(String content)
38 {
39     if (client == null)
40     {
41         return false;
42     } else if (content == null)
43     {
44         throw new ArgumentNullException("content cannot be null");
45     }
46
47     Byte[] bArray = Encoding.ASCII.GetBytes(content + '\r');
48     NetworkStream stream = client.GetStream();
49     stream.Write(bArray, 0, bArray.Length);
50     stream.Flush();
51     return true;
52 }
```

This is the function that we use to send data to a remote device. At line 38, we check if we have a valid connection with a remote socket. If so, we turn the `String` data into an array of bytes, and then send the bytes across the network. In case you are curious, `NetworkStream` is a class that provides buffering for data transmission. When we do `stream.write()`, we are actually writing to a buffer, not to the remote device. It is only when we have fill the buffer or we call `stream.Flush()` that the data gets sent.

```
79 1 reference
80 public static void ScanNearby(List<String> lst)
81 {
82     String ip = GetLocalIPv4();
83
84     //List < String > lst = new List<String>();
85     String[] ip_parts;
86     int Start;
87     if (ip.Length == 0)
88     {
89         return;
90     }
91     ip_parts = ip.Split('.');
92     if (ip_parts.Length < 3)
93     {
94         return;
95     }
96     Start = int.Parse(ip_parts[3]);
97     for (Start++; Start <= 20; Start++)
98     {
99         String tmpIP = ip_parts[0] + "." + ip_parts[1] + "." + ip_parts[2] + "." + Start.ToString();
100         if (PingHost(tmpIP))
101         {
102             lst.Add(tmpIP);
103         }
104     }
105 }
```



ScanNearby() is a function that scans for remote devices that are connected to the same Wi-Fi as the computer running this application. It does this by sending a ping signal to nearby IPv4 addresses and then add the addresses that responded to the ping signal to a result list. This function is used by ScanNearby.cs.

### Code Explanation (9 of 10)

*Program.cs*

```
0 references
10 static class Program
11 {
12     /// <summary>
13     /// The main entry point for the application.
14     /// </summary>
15     [STAThread]
16     static void Main()
17     {
18         Application.EnableVisualStyles();
19         Application.SetCompatibleTextRenderingDefault(false);
20         Application.Run(new Form1());
21     }
22 }
```

The Main() function is the entry point of our C# application. All this function does is that it displays our User Interface window as soon as the application started.

### Code Explanation (10 of 10)

*Global.cs*

```
8 references
9 public static class Global
10 {
11     public static bool HornEnabled = false;
12     public static bool LightEnabled = false;
13     public static bool SpeedEnabled = false;
14     public static bool RecordEnabled = false;
15 }
```

This class holds global variables that we will use for configuration purposes. Modification to the variables defined in this class (i.e. Global.HornEnabled = true) will affect components that use these variables. Be aware that although these variables are defined, they are not used by our application in the current release. If you wish to extend the functionality of our simple C# application, Extension.cs could be the place to start.

### Part 3: Android App

The project can be downloaded here: <https://github.com/miska12345/ESP8266RC>

**Brief Description:** This is a simple Android app that have a virtual joystick as its sole component. User move the joystick in order to control the ESP8266 car via remote connection using the TCP protocol.

**Note:** The app is intended to serve as a reference for how similar ideas can be applied in Android app development. Implementation details (a.k.a code explanations) will be omitted.

### Results

After uploading the Arduino code to ESP8266, disconnect the USB cable and start the car with battery. Wait until the wheels stop rotating, run the C# application and either connect by IP or scan nearby if you don't know the car's IP address. Once a connection has been established, press W, A, S, D to move the car.

Watch Demo @: [https://www.youtube.com/watch?v=zohX36Z\\_Hoc](https://www.youtube.com/watch?v=zohX36Z_Hoc)

If you have any questions, feel free to open an issue at the project's GitHub page!

## Reference

1. Arduino Documentation: <https://www.arduino.cc/en/main/docs>
2. Android Studio Joystick: <https://github.com/controlwear/virtual-joystick-android>
3. C# Socket Class:  
<https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket?view=netframework-k-4.8>
4. ESP8266 Documentation: <https://arduino-esp8266.readthedocs.io/en/latest/>
5. Java Socket: <https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html>
6. MetroModernUI: <https://www.nuget.org/packages/MetroModernUI/>
7. Modular Software Design: [https://en.wikipedia.org/wiki/Modular\\_programming](https://en.wikipedia.org/wiki/Modular_programming)
8. NodeMCU C# (UDP):  
<https://www.instructables.com/id/DIY-WIFI-RC-Car-With-ESP8266-and-Arduino-IDE/>
9. NodeMCU Arduino:  
<https://www.hackster.io/brian-lough/simple-wifi-controlled-rc-car-cebb87>
10. More about socket:  
<https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html>