Experiment No. – 6

Title: Get the status of a bulb at a remote place (on the LAN) through web.

Aim: To get the status of a bulb/led using a webserver and iot

Hardware Requirements: Arduino Uno board, Micro-IoT sensor actuator board, Power adaptor and relay (bulb).

Software Requirements: Arduino IDE

Theory:

ESP8266 Module :

The ESP8266 Wi-Fi Module is used to connect with any available internet hotspot and control the LED light. The ESP8266 Wi-Fi Module is a self contained SOC with integrated TCP/IP protocol stack that can provide access to a Wi-Fi network. The ESP8266 is capable of either hosting an application or off loading all Wi-Fi networking functions from another application processor. Each ESP8266 module comes pre-programmed with an AT command set firmware. This firmware can be used to communicate to the ESP8266 module via **AT command** But, if the Arduino IDE is used, this firmware will be over written. If the Arduino IDE is once used to program the ESP module, then AT commands cannot be used to configure it. Hence, the module is flashed with the default firmware so that first the AT commands can be used to configure it.

The module comes available in two models – ESP-01 and ESP-12. ESP-12 has 16 pins available for interfacing while ESP-01 has only 8 pins available for use.

The ESP-01 model has the following pin configuration –

Pin Number	Pin Name	Pin Function
1	Ground	Ground
2	GPIO1	General purpose IO, Serial Tx1
3	GPIO2	General purpose IO
4	CH_PD	Active High Chip Enable
5	GPIO0	General purpose IO, Launch Serial Programming Mode if Low while Reset or Power ON
6	RESET	Active Low External Reset Signal
7	GPIO3	General purpose IO, Serial Rx
8	VCC	Power Supply

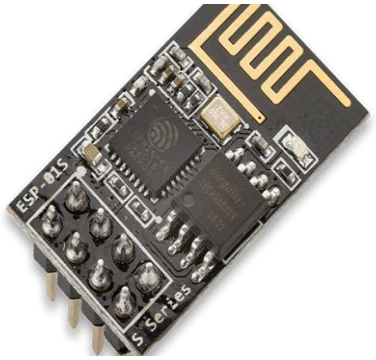
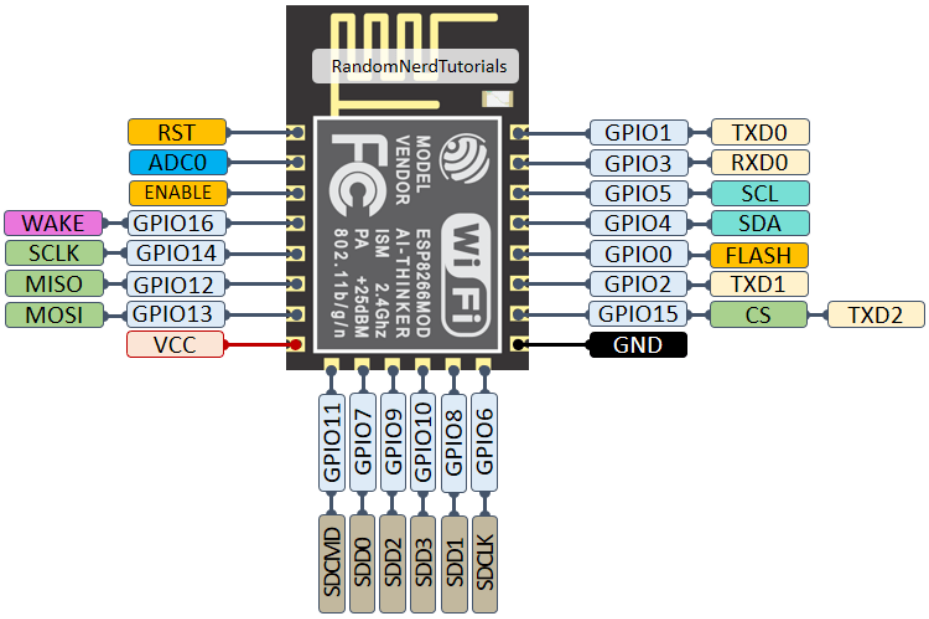


Table listing pin configuration of ESP8266 ESP-01 Wi-Fi Modem

The ESP-12 model has the following pin configuration –



Pin Number	Pin Name	Pin Function
1	RESET	Active Low External Reset Signal
2	ADC(TOUT)	ADC Pin Analog Input
3	CH_PD	Active High Chip Enable
4	GPIO16	General purpose IO
5	GPIO14	General purpose IO
6	GPIO12	General purpose IO
7	GPIO13	General purpose IO
8	VCC	Power Supply
9	Ground	Ground
10	GPIO15	General purpose IO, should be connected to ground for booting from internal flash
11	GPIO1	General purpose IO, Serial Tx1
12	GPIO0	General purpose IO, Launch Serial Programming Mode if Low while Reset or Power ON
13	GPIO4	General purpose IO
14	GPIO5	General purpose IO
15	GPIO3	General purpose IO, Serial Rx
16	GPIO1	General purpose IO, Serial Tx

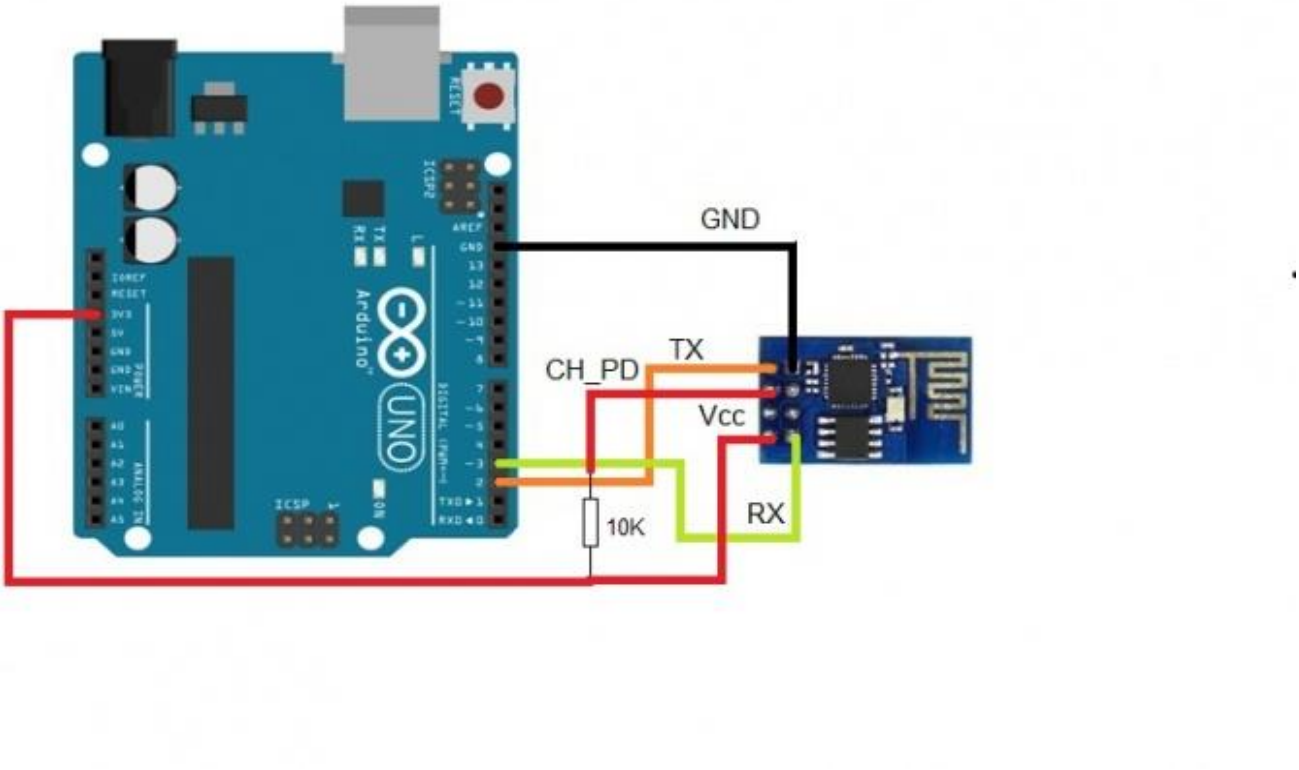
Table listing pin configuration of ESP8266 ESP-12 Wi-Fi Modem

The Chip Enable and VCC pins of the module are connected to the 3.3V DC while Ground pin is connected to the common ground. The RESET pin is connected to the ground via a tactile switch where the pin is supplied VCC through a pull up resistor by default. The Tx and Rx pins of the module are connected to the RXI and TDX terminals of the FTDI USB to Serial converter. An LED is connected to GPIO2 pin of the module via a pull up resistor of 1KΩ.

Features of ESP8266

- 802.11 b/g/n protocol
- Wi-Fi Direct (P2P), soft-AP
- Integrated TCP/IP protocol stack
- Integrated TR switch, balun, LNA, power amplifier and matching network
- Integrated PLL, regulators, and power management units
- +19.5dBm output power in 802.11b mode
- Integrated temperature sensor
- Supports antenna diversity
- Power down leakage current of < 10uA
- Integrated low power 32-bit CPU could be used as application processor
- Wake up and transmit packets in < 2ms
- Standby power consumption of < 1.0mW (DTIM3)

Circuit Diagram:



Interfacing of ESP8266 Module with Arduino

The **ESP8266 module** works with 3.3V only, anything more than 3.7V would kill the module hence be cautions with your circuits. The best way to program an **ESP-01** is by using the FTDI board that supports 3.3V programming. If you don't have one it is recommended to buy one or for time being you can also use an Arduino board. One commonly problem that every one faces with ESP-01 is the powering up problem. The module is a bit power hungry while programming and hence you can power it with a 3.3V pin on

Arduino or just use a potential divider. So it is important to make a small voltage regulator for 3.31v that could supply a minimum of 500mA.

Procedure:

Step 1: Connect the Arduino board to the Micro-IoT Sensor board using the FRC cable provided with the board.

Step 2: Connect the Power supply adaptor and power on the circuit.

Step 3: Open Arduino IDE and create a new sketch (program) using the above pins.

Step 4: In the Arduino IDE go to tools◊Port and select the appropriate COM port.

Step 5: Make sure to create a Hotspot or Wi-Fi connection.er the credentials of the Wi-Fi or hotspot like SSID and password in the program. In the Arduino IDE click on the upload button ( ) to compile and download the code into the Arduino UNO. When successfully downloaded the code will start running.

Step 6: This experiment uses the ESP8266 Wi-Fi module. To connect the module to the circuit slide the switch SW6 between 1-2 (WiFi_ON) . Reset the ESP8266 by pressing the switch SW10. Now reset the Arduino board. Open the serial monitor @115200 baud rate and observe the IP address.

Step 7: Using the browser open the ip address and click on the ON / OFF commands.

Conclusion:

Code:

```
/*
simple demo to get the status of a bulb/led using a webserver and iot

 A simple web server that lets you turn on and of an bulb/LED via a web page.
 This sketch will print the IP address of your ESP8266 module (once connected)
 to the Serial monitor. From there, you can open that address in a web browser
 to turn on and off the LED on pin 0.

 */

#include "WiFiEsp.h"

// Emulate Serial1 on pins 6/7 if not present
#ifndef HAVE_HWSERIAL1
#include "SoftwareSerial.h"
SoftwareSerial Serial1(8, 7); // RX, TX
#endif

char ssid[] = "Shreeparth1";            // your network SSID (name)
char pass[] = "shreepart1";        // your network password
int status = WL_IDLE_STATUS;

int ledStatus = LOW;

WiFiEspServer server(80);

// use a ring buffer to increase speed and reduce memory allocation
RingBuffer buf(8);

void setup()
{
  Serial.begin(115200);   // initialize serial for debugging
  Serial1.begin(9600);    // initialize serial for ESP module
  WiFi.init(&Serial1);    // initialize ESP module

  pinMode(9, OUTPUT);
  pinMode(A1, OUTPUT);

  digitalWrite(9, 0);
  digitalWrite(A1, 0);


  // check for the presence of the shield
  if (WiFi.status() == WL_NO_SHIELD) {
```

```
    Serial.println("WiFi shield not present");
    // don't continue
    while (true);

  }

  // attempt to connect to WiFi network
  while (status != WL_CONNECTED) {
    Serial.print("Attempting to connect to WPA SSID: ");
    Serial.println(ssid);
    // Connect to WPA/WPA2 network
    status = WiFi.begin(ssid, pass);
  }

  Serial.println("You're connected to the network");
  printWifiStatus();

  // start the web server on port 80
  server.begin();
}


void loop()
{
  WiFiEspClient client = server.available();  // listen for incoming clients

  if (client) {                             // if you get a client,
    Serial.println("New client");           // print a message out the serial port
    buf.init();                             // initialize the circular buffer
    while (client.connected()) {            // loop while the client's connected
      if (client.available()) {             // if there's bytes to read from the client,
        char c = client.read();             // read a byte, then
        buf.push(c);                        // push it to the ring buffer

        // printing the stream to the serial monitor will slow down
        // the receiving of data from the ESP filling the serial buffer
        //Serial.write(c);

        // you got two newline characters in a row
        // that's the end of the HTTP request, so send a response
        if (buf.endsWith("\r\n\r\n")) {
          sendHttpResponse(client);
          break;
        }
```

```
        // Check to see if the client request was "GET /H" or "GET /L":
        if (buf.endsWith("GET /H")) {
          Serial.println("Turn bulb ON");
          ledStatus = HIGH;
          digitalWrite(9, HIGH);
          digitalWrite(A1, HIGH);

        }
        else if (buf.endsWith("GET /L")) {
          Serial.println("Turn bulb OFF");
          ledStatus = LOW;
          digitalWrite(9, LOW);
          digitalWrite(A1, LOW);
          }
       }
     }

    // close the connection
    client.stop();
    Serial.println("Client disconnected");
  }
}


void sendHttpResponse(WiFiEspClient client)
{
  // HTTP headers always start with a response code (e.g. HTTP/1.1 200 OK)
  // and a content-type so the client knows what's coming, then a blank line:
  client.println("HTTP/1.1 200 OK");
  client.println("Content-type:text/html");
  client.println();

  // the content of the HTTP response follows the header:
  client.print("The BULB is : ");
 // client.print(ledStatus);
 if(ledStatus==0)
 {
  client.print("OFF");
  }
  else
  {
    client.print("ON");
   }
  client.println("<br>");
  client.println("<br>");
```

```
  client.println("Click <a href=\"/H\">here</a> turn the BULB on<br>");
  client.println("Click <a href=\"/L\">here</a> turn the BULB off<br>");

  // The HTTP response ends with another blank line:
  client.println();
}

void printWifiStatus()
{
  // print the SSID of the network you're attached to
  Serial.print("SSID: ");
  Serial.println(WiFi.SSID());

  // print your WiFi shield's IP address
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);

  // print where to go in the browser
  Serial.println();
  Serial.print("To see this page in action, open a browser to http://");
  Serial.println(ip);
  Serial.println();
}
```