Republic of Turkey

Kocaeli University

Faculty of Engineering

Electronic & Communication Eng.

Micro-Controllers Project

Wi-Fi ThingSpeak - Updating System using MSP430G2553 Microprocessor, LDR Sensor, Temperature Sensor (LM35) and ESP8266 Wi-Fi Module.

Submitted by

Zeynep ÖZÇELİK- 190208062

Supervised by

Ayhan KÜÇÜKMANİSA

Kocaeli, May 2022

Wi-Fi ThingSpeak - Updating System using MSP430G2553 microprocessor, LDR sensor, Temperature Sensor (LM35) and ESP8266 Wi-Fi Module.

---

1. Abstract

An embedded system that measures the temperature of the atmosphere using LM35 Temperature Sensor, and also detect the Light-Level in the atmosphere using LDR (Light Dependent Resistor), developed using the MSP430G2553 microprocessor, and ESP8266 Wi-Fi module.

The Data of the sensors and historical statistics are visualized on a live-updating cloud server (ThingSpeak).

2. Background

2.1 MSP430G2553

Texas Instruments MSP430G2553 is part of the MSP430 family of ultra-low-power microcontrollers featuring different sets of peripherals. The MSP430G2553 mixed signal microcontroller features a 16-bit RISC CPU, 16-bit registers, 16KB non-volatile memory, 512 bytes RAM, 10-bit ADC, UART, and two 16-bit timers. The G2x53 series is popular due to the low price and various package sizes. The MSP430G2553 acts as the master node (central hub) of this system– connecting the sensors and ESP8266.

2.2 ESP8266

ESP8266 WiFi module is a self-contained SoC with integrated TCP/IP protocol. The module is an extremely low-cost all-in-one solution for Wi-Fi connectivity for microcontrollers. The ESP8266 contains an API- Application Programming Interface (AT command set firmware) which allows any microcontroller to control the module using UART protocol. The ESP8266 is used in this project as a client to initiate TCP connections to a cloud server and transmit information via HTTP requests.

2.3 LM35 (Temperature Sensor)

The LM35 series are precision integrated-circuit temperature devices with an output voltage linearly-proportional to the Centigrade temperature. The LM35 device does not require any external calibration or trimming to provide typical accuracies of ±¼°C at room temperature and ±¾°C over a full −55°C to 150°C temperature range.

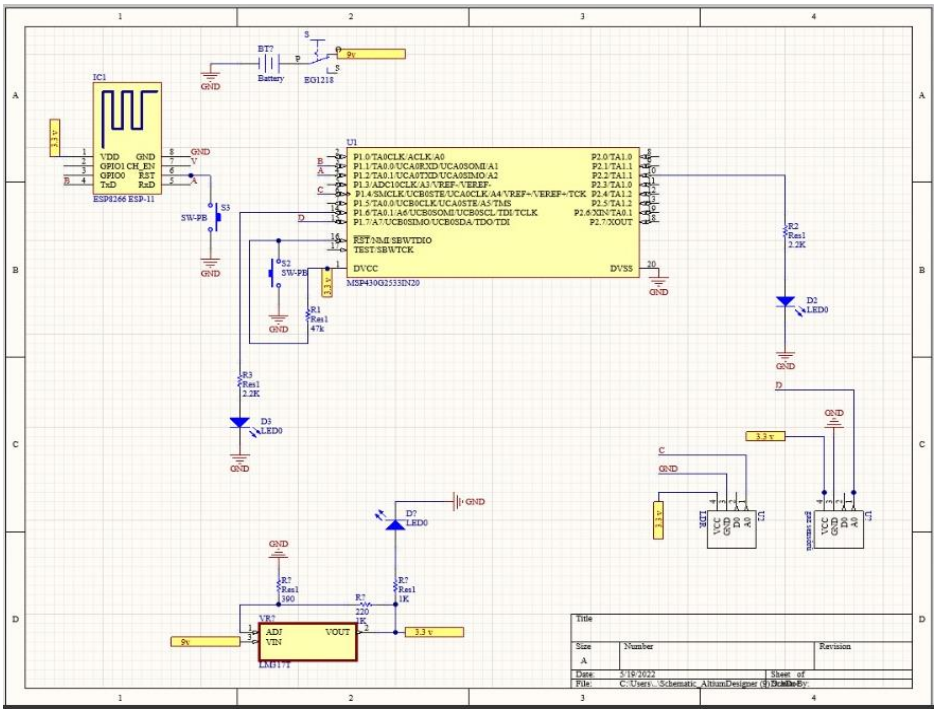
2.4 LDR Module (Light Dependent Resistor)

An LDR is a component that has a (variable) resistance that changes with the light intensity that falls upon it. This allows them to be used in light sensing circuits.

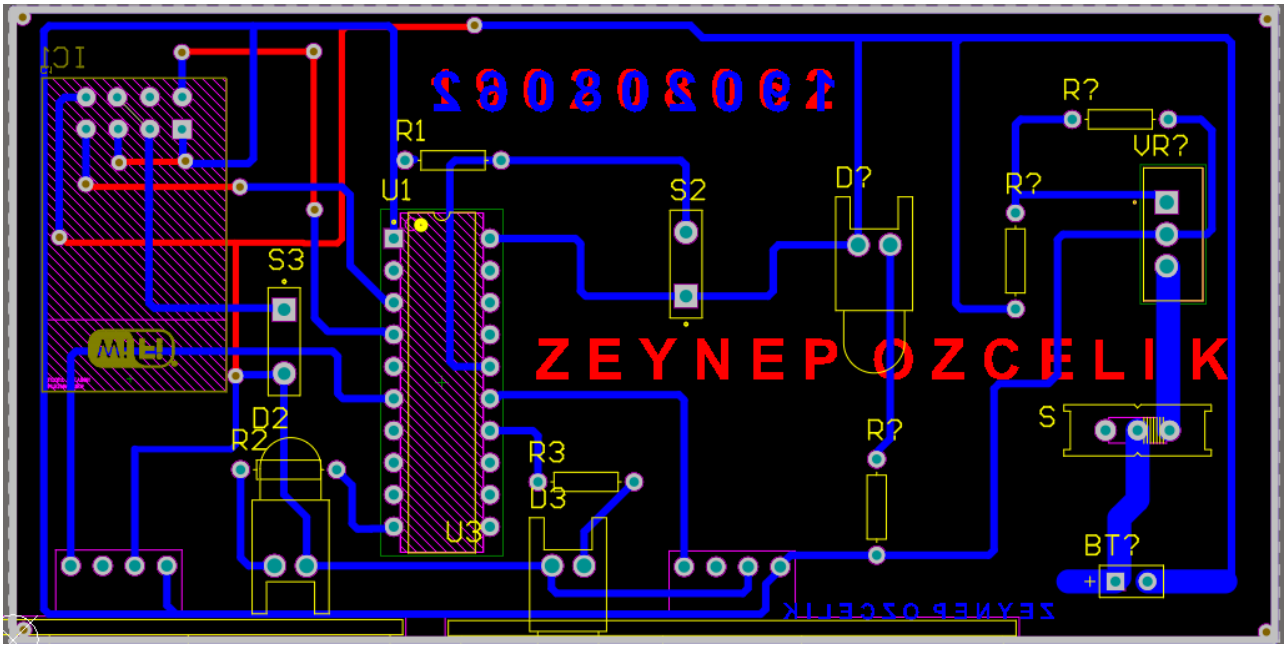
3. System Design

3.1 Schematic and PCB

This system's schematic was designed using Altium Designer 17.0 and can be seen in the Figure below... The circuit interfaces the MSP430 microprocessor with both the ESP8266 and the two sensors. The circuit provides several second-order design considerations such as resistors, and LED status indicators.



The PCB was designed for the system using Altium Designer 17.0. A 3D visualized render of the PCB is seen in Figure below:



3.2 MSP430G2553 Software

3.2.1 ADC Initialization (For both sensors)

The ADC ISR solves the distance equation using the ADC input. The MSP430 ADC registers are configured to the following parameters

ADC Register Initializations

Register	Bit	Description
ADC10CTL0	REFON	Reference generator: ON
ADC10CTL0	SREF_0	Select reference VR+ = AVCC and VR- = AVSS
ADC10CTL0	ADC10ON	ADC10 Enable: ON
ADC10CTL0	ADC10IE	ADC10 Interrupt: ENABLED
ADC10CTL1	INCH_4	Input channel A4 (P1.4) – LDR Sensor
ADC10CTL1	INCH_7	Input channel A7 (P1.7) – TMP Sensor

ADC10AE0	BIT4	ADC Analog enable A4- LDR Sensor
ADC10AE0	BIT7	ADC Analog enable A7- TMP Sensor

For LDR:

The levels of the light was set due to the ADC10MEM 10-bit register (Max. 1023) where 5 indicates a very high light and 1 indicates a very low light.

```
//LDR Levels ( 5 --> Light is so high , 1 --> Light is so low ) LDR = 0;
    if ( LDR_Result > 0 && LDR_Result < 100 ) { LDR = 5 ; }
    else if ( LDR_Result > 100 && LDR_Result < 250 ) { LDR = 4 ; }
    else if ( LDR_Result > 250 && LDR_Result < 450 ) { LDR = 3 ; }
    else if ( LDR_Result > 450 && LDR_Result < 650 ) { LDR = 2 ; }
    else if ( 650 < LDR_Result ) { LDR = 1 ;}
```

3.2.2 UART Initialization

UART is used to communicate (RX/TX) with the ESP8266 WiFi module. The ESP8266 communicates at 115200 baud with about 3.3V logic.

UART Register Initializations

Register	Bit	Description
DCOCTL	0	Select lowest DCOx and MODx settings
DCOCTL	CALDCO_1MHZ	Set DCO
BCSCTL1	CALBC1_1MHZ	Set DCO
P1SEL	BIT1 + BIT2	P1.1 = RXD, P1.2=TXD

P1SEL2	BIT1 + BIT2	P1.1 = RXD, P1.2=TXD
UCA0CTL1	UCSSEL_2	Set UART use SMCLK
UCA0BR0	8	1MHz 115200 baud
UCA0BR1	0	1MHz 115200 baud
UCA0MCTL	UCBRS2	Modulation UCBRSx = 5
UCA0MCTL	UCBRS0	Modulation UCBRSx = 5

Functions are created to send UART strings to/from the ESP8266.

Function	**Description**
void TX(**const** char *s)	Send a string via UART
void putc(**const** unsigned c)	Send a character via UART
void crlf(void)	Sends a carriage return and new line via UART

3.2.3 Timer Initialization

A timer is initialized to send distance data to the web server on a specified interval. The MSP430G2553 contains a 1MHz SMCLK. The maximum clock divider available is 8-times. Because the MSP430G2553 uses 16-bit registers the maximum period is 65535. Therefore, the slowest hardware frequency for a timer is 2Hz.

$$2\text{Hz} = \frac{(1\text{MHz})/8}{\text{Period}}$$

Solving for the timer period for a 2Hz interval results in a period of 62500. This is the maximum value before we reach an overflow of 65535.

Timer A Register Initializations

Register	Bit	Description
TA0CCTL0	CCIE	Enable interrupts for Timer A0
TA0CTL	TASSEL_2	Select SMCLK clock source
TA0CTL	MC_1	Set timer mode to UPMODE
TA0CTL	ID_3	Set clock divider to SMCLK/8
TA0CCR0	62500	Set capture compare register period to 62500

Using software, the timer can be acted upon in slower intervals. Because we know the timer interrupts every 2Hz (0.5 Second) we can use software to determine when to activate code. By the way, because of the "Timer Interrupt Processing Time -which contain some delay cycles- it exceeds the 2Hz (0.5 Second), it is about 25 second, so there will always be a pending interrupt so the code will act like there is a loop inside the timer interrupt function.

3.2.4 WiFi Functions

The program has functions for sending specific commands to the ESP8266 to send and format data for upload. The program requires definitions for Wi-Fi information such as SSID, Password, URL, etc. The program requires definitions for WiFi information such as SSID, Password, URL, etc.

WiFi Constant Definitions

Constant	Description
MAX_SEND_LENGTH	Determines maximum number of digits of sensor data to send (default 3)
BASE_URL_LDR	The URL for API request servicer – LDR sensor

BASE_URL_TMP	The URL for API request servicer – TMP sensor
TCP_REQUEST	Preformatted message to initiate TCP request
WIFI_NETWORK_SSID	SSID for WiFi network
WIFI_NETWORK_PASS	Password for WiFi network
ENABLE_CONFIG_WIFI	Enable to configure the WiFi network to settings

Functions are created perform WiFi related tasks.

Function	**Description**
void WifiConfigureNetwork(void)	Configure the network to defined settings
void WiFiSendTCP()	Initiate TCP request with web server
void WiFiSendLength(unsigned int data)	Send the length of the incoming TCP request to ESP8266 before upload- Was set as 50- Useless function

CODE OF THE PROJECT :

```
#include <msp430.h>

#define MAX_SEND_LENGTH   3     // 3 characters or 3 digits to send over WiFi

#define BASE_URL_LDR          "GET /update?key=XILWJ5EF3TLAJEOR&field3="      // Used in WiFiSendMessage_LDR Function to send the Data of LDR to field 3 in ThingSpeak Channel

#define BASE_URL_TMP          "GET /update?key=XILWJ5EF3TLAJEOR&field1="     // Used in WiFiSendMessage_TMP Function to send the Data of TMP to field 1 in ThingSpeak Channel

#define WIFI_NETWORK_SSID         "zeynep"    // Network SSID

#define WIFI_NETWORK_PASS         "303380zey"   // Network Password
```

```
#define ENABLE_CONFIG_WIFI       0          // Enable if you want to program the ESP8266 to connect to a new WiFi network

// Network configuration on ESP8266 only needs to run once. The ESP8266 keeps memory and will automatically connect to programmed networks

// Only enable if connecting to a new network or updating network settings // Disable after use


/* Functions' Prototypes */

void initUART(void); // UART configuration

void initLED(void);  // LEDs Configuration

void initADC_LDR(void); // ADC Configuration for LDR Sensor

void resetADC_LDR(void); // Reset ADC Configuration to Defult

void initADC_TMP(void); // ADC Configuration for TMP Sensor

void resetADC_TMP(void); // Reset ADC Configuration to Defult

void initUploadTimer(void); // Timer0_A0 Configuration

void putc(const unsigned c); // Output char to UART

void TX(const char *s); // Output string to UART

void crlf(void); // CarriageReturn and LineFeed (Which e10nds the line and starts a new line - sends the message to ESP8266)

void WifiConfigureNetwork(void); // Wifi Network Configuration

void WiFiSendMessage_LDR(int data); // Send LDR data

void WiFiSendMessage_TMP(int data); // Send TMP data

void WiFiSendLength(int data); // Send the Length of the HTTP Request -- Defult 50 digit

void WiFiSendTCP(void); // Send TCP Request

char singleDigitToChar(unsigned int digit); // Converts Single to chart to be sent by the Function putc();

// Integers

int LDR_Result ;

int LDR ;

int TMP_Result;

int TMP;

int sensorReadingLDR = 0;

int sensorReadingTMP = 0;

// Main Function

int main(void)

{

    WDTCTL = WDTPW + WDTHOLD;             // Stop WDT

   if (CALBC1_1MHZ == 0xFF)              // If calibration constant erased

   {
```

```
    while(1);                   // do not load, trap CPU!!
  }
  initUART();
  initLED();
 #if defined(ENABLE_CONFIG_WIFI)
    WifiConfigureNetwork();                     // Configure the ESP8266 WiFi settings
 #endif
  initUploadTimer();              // Enable timer to upload at 2Hz frequency
  __bis_SR_register(LPM0_bits + GIE);   // Enter low power mode with interrupts enabled
                                                // The UploadTimer will interrupt and start ADC sampling and WiFi uploading
}// Initialize TimerB for sending WiFi timer
void initUploadTimer(void)
{  TA0CCTL0 = CCIE;                                             // Enable Timer Interrupt
  TA0CTL = TASSEL_2 + MC_1 + ID_3;     // SMCLK/8, UPMODE, Compare mode
  TA0CCR0 = 62500;                // Timer period (125kHz/2Hz), slowest before overflow
}
void initADC_LDR(void)
{  ADC10CTL0&=~ENC;
  ADC10CTL0|=SREF_1+ADC10SHT_3+REFON+ADC10ON+ADC10IE;
  ADC10CTL1|=INCH_4+ADC10DIV_3+ADC10SSEL_0;
  ADC10AE0=BIT4; }
void resetADC_LDR(void)
{          ADC10CTL0 &= ~ENC ; // Disable Conversation - ADC10CTL0 REGISTER CANNOT BE MODYFIED WHILE EMC IS ENABLED.
  ADC10CTL0 = 0 ;   ADC10CTL1 = 0 ; ADC10AE0 =  0 ;
} void initADC_TMP(void) {
  ADC10CTL0&=~ENC;
  ADC10CTL0|=SREF_1+ADC10SHT_3+REFON+ADC10ON+ADC10IE;
  ADC10CTL1|=INCH_7+ADC10DIV_3+ADC10SSEL_0;
  ADC10AE0=BIT7;
}
void resetADC_TMP(void)
{
  ADC10CTL0 &= ~ENC ; // Disable Conversation - ADC10CTL0 REGISTER CANNOT BE MODYFIED WHILE EMC IS ENABLED.
  ADC10CTL0 = 0 ;   ADC10CTL1 = 0 ; ADC10AE0 =  0 ;
```

```
}
// Configure the ESP8266 for the current WiFi network
// This function needs to only run once. The ESP8266 will always connect to the network after being programmed.
void WifiConfigureNetwork(void)
{
        TX("AT+RST");                                   // Reset the ESP8266
        crlf();                                                 // CR LF
        __delay_cycles(2000000); // Delay
        TX("AT+CWMODE_DEF=1");          // Set mode to client (save to flash)
                                                                // Modem-sleep mode is the default sleep mode for
CWMODE = 1
                                                                // ESP8266 will automatically sleep and go into "low-power"
(15mA average) when idle
        crlf();                                                 // CR LF
        __delay_cycles(2000000); // Delay
        TX("AT+CWJAP_DEF=\"");     // Assign network settings (save to flash)
        TX(WIFI_NETWORK_SSID);
        TX("\",\"");
        TX(WIFI_NETWORK_PASS);          // Connect to WiFi network
        TX("\"");
        crlf();
        __delay_cycles(5000000); // Delay, wait for ESP8266 to fetch IP and connect }
// Transmits the TCP request
void WiFiSendTCP()
{
   // TCP Request to ThingSpeak  AT+CIPSTART="TCP","api.thingspeak.com",80
   TX("AT+CIPSTART=");
   while (!(IFG2&UCA0TXIFG)); UCA0TXBUF = '"';          // Transmit a byte
   TX("TCP");
   while (!(IFG2&UCA0TXIFG)); UCA0TXBUF = '"';
   TX(",");
   while (!(IFG2&UCA0TXIFG)); UCA0TXBUF = '"';
   TX("api.thingspeak.com");
   while (!(IFG2&UCA0TXIFG)); UCA0TXBUF = '"';
   TX(",");
```

```
    TX("80");
    crlf();
}

// Converts a single digit to its corresponding character
char singleDigitToChar(unsigned int digit)
{
    char returnDigit = '0';
    if(digit == 0) returnDigit = '0';
    else if(digit == 1) returnDigit = '1';  else if(digit == 2) returnDigit = '2'; else if(digit == 3) returnDigit = '3';
    else if(digit == 4) returnDigit = '4';  else if(digit == 5) returnDigit = '5'; else if(digit == 6) returnDigit = '6';
    else if(digit == 7) returnDigit = '7';  else if(digit == 8) returnDigit = '8'; else if(digit == 9) returnDigit = '9';
    else returnDigit = '0'; return returnDigit;
}

// The ESP8266 requires that the message request length be known to the processor before the request is sent. This tells the ESP8266 when the stop searching for data to transmit
// Function calculates the length of the message to be sent and transmits to ESP8266
void WiFiSendLength(int data) // Input the ADC Reading // Set as 50 digits
{
    TX("AT+CIPSEND=50"); // The start of the command, the next parameter is the length
    crlf();
}
void WiFiSendMessage_LDR(int data) // Input the ADC distance reading
{
    TX(BASE_URL_LDR); // Send the beginning part of the request
    // Split the data integer into separate characters
    unsigned int i;
    unsigned int reverseIndex = MAX_SEND_LENGTH - 1;
    char reverseBuffer[MAX_SEND_LENGTH - 1];
    for(i = 0; i < MAX_SEND_LENGTH; i++) reverseBuffer[i] = 0; // Initialize all indices at zero
    while(data > 0)
    {
        int operatedDigit ;
```

```
    operatedDigit = data % 10 ;
    reverseBuffer[reverseIndex] = singleDigitToChar(operatedDigit);
    reverseIndex--;
    data /= 10;
  }
  for(i = 0; i < MAX_SEND_LENGTH; i++)
  {
    if(reverseBuffer[i] != 0) putc(reverseBuffer[i]); // If initialized indice, transmit each individual character
  }
  crlf();
}
void WiFiSendMessage_TMP(int data) // Input the ADC distance reading
{
  TX(BASE_URL_TMP); // Send the beginning part of the request
  // Split the data integer into separate characters
  unsigned int i;
  unsigned int reverseIndex = MAX_SEND_LENGTH - 1;
  char reverseBuffer[MAX_SEND_LENGTH - 1];
  for(i = 0; i < MAX_SEND_LENGTH; i++) reverseBuffer[i] = 0; // Initialize all indices at zero
  while(data > 0)
  {
    unsigned int operatedDigit = data % 10;
    reverseBuffer[reverseIndex] = singleDigitToChar(operatedDigit);
    reverseIndex--;
    data /= 10;
  }
  for(i = 0; i < MAX_SEND_LENGTH; i++)
  {
    if(reverseBuffer[i] != 0) putc(reverseBuffer[i]); // If initialized indice, transmit each individual character
  }
  crlf();
}
// ADC10 interrupt service routine
#pragma vector=ADC10_VECTOR
```

```
__interrupt void ADC10_ISR(void)
{
   __bic_SR_register_on_exit(CPUOFF);
}
// Timer A0 interrupt service routine (UploadTimer)
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer0_A0 (void)
{
      P1OUT &= ~BIT6; P2OUT &= ~BIT2; // LEDS OFF
      initADC_LDR();
      __delay_cycles(1000000);
      ADC10CTL0 |= ENC + ADC10SC;        // Sampling and conversion start
      sensorReadingLDR = ADC10MEM ;
          LDR_Result = sensorReadingLDR;
      // LDR Levels ( 5 --> Light is so high , 1 --> Light is so low )
      LDR = 0;
      if ( LDR_Result > 0 && LDR_Result < 100 ) { LDR = 5 ; }
      else if ( LDR_Result > 100 && LDR_Result < 250 ) { LDR = 4 ; }
      else if ( LDR_Result > 250 && LDR_Result < 450 ) { LDR = 3 ; }
      else if ( LDR_Result > 450 && LDR_Result < 650 ) { LDR =2 ; }
      else if ( 650 > LDR_Result ) { LDR = 1 ;}
      // Send LDR level to Thingspeak server
      P2OUT |= BIT2; //  LED ON
      WiFiSendTCP();
      __delay_cycles(500000); // 0.5s delay
      WiFiSendLength(LDR);
      P2OUT ^= BIT2; // Toggle output LED
      __delay_cycles(500000); // 0.5s delay
      for ( int x = 0 ; x < 8 ; x++){
        WiFiSendMessage_LDR(LDR);}
      P2OUT |= BIT2; // Enable output LED
      __delay_cycles(500000); // 0.5s delay
      sensorReadingLDR = 0 ; LDR_Result = 0 ; LDR = 0 ;
```

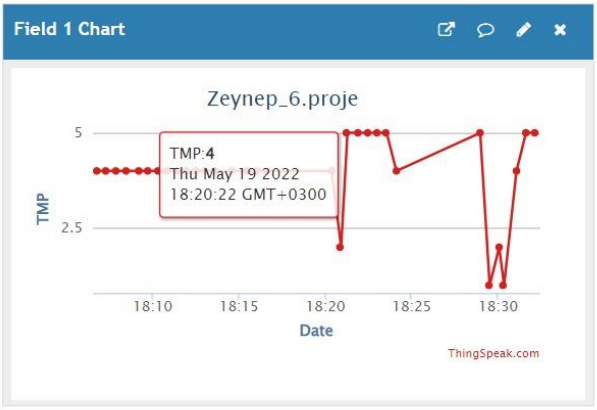
```
    // Second Sensor configuration
    P2OUT &= ~BIT2;
    __delay_cycles(3000000); // 2 s delay
    resetADC_LDR();
    initADC_TMP(); // Set ADC configs for the TMP sensor.
    __delay_cycles(1000000);
    ADC10CTL0 |= ENC + ADC10SC;         // Sampling and conversion start
    sensorReadingTMP = 0 ;
    sensorReadingTMP = ADC10MEM ;
    //__bis_SR_register(CPUOFF + GIE);    // LPM0, ADC10_ISR will force exit
    TMP_Result = sensorReadingTMP;
    TMP = 0 ;
    if ( TMP_Result > 0 && TMP_Result < 100 ) { TMP = 1 ; }
    else if ( TMP_Result > 100 && TMP_Result < 250 ) { TMP = 2 ; }
    else if ( TMP_Result > 250 && TMP_Result < 450 ) { TMP = 3 ; }
    else if ( TMP_Result > 450 && TMP_Result < 650 ) { TMP = 4 ; }
    else if ( 650 > TMP_Result ) { TMP = 1 ;}
    P1OUT |= BIT6; // Toggle output LED
    WiFiSendTCP();
    __delay_cycles(500000); // 0.5s delay
    WiFiSendLength(TMP);
    P1OUT ^= BIT6; // Toggle output LED
    __delay_cycles(500000); // 0.5s delay
    for ( int x = 0 ; x < 8 ; x++ ){
      WiFiSendMessage_TMP(TMP);}
    P1OUT |= BIT6; // Enable output LED
    __delay_cycles(500000); // 0.5s delay
    resetADC_TMP();
    __delay_cycles(3000000); // 2 s delay
    sensorReadingTMP = 0 ; TMP_Result = 0 ; TMP = 0 ;
}
// Initialize LED on P1.0 (indicator LED)
void initLED(void)
{
```

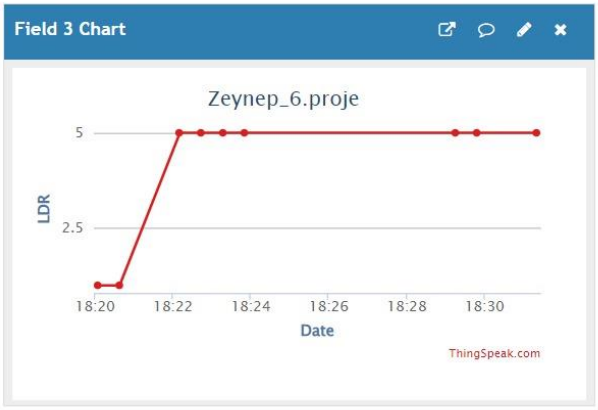
```
   P2DIR |= BIT2; // Set P1.0 to the output direction
   P2OUT |= BIT2; // Enable the LED
   P1DIR |= BIT6; // Set P1.0 to the output direction
   P1OUT |= BIT6; // Enable the LED
}
// Output char to UART
void putc(const unsigned c)
{
   while (!(IFG2&UCA0TXIFG)); // USCI_A0 TX buffer ready?
   UCA0TXBUF = c;
}
// Output string to UART
void TX(const char *s)
{
   while(*s) putc(*s++);
}
// CR LF
void crlf(void)
{ TX("\r\n"); }
void initUART(void)
{
   DCOCTL = 0;                          // Select lowest DCOx and MODx settings
   BCSCTL1 = CALBC1_1MHZ;                  // Set DCO
   DCOCTL = CALDCO_1MHZ;
   P1DIR = 0xFF;                        // All P1.x outputs
   P1OUT = 0;                           // All P1.x reset
   P1SEL = BIT1 + BIT2 + BIT4;             // P1.1 = RXD, P1.2=TXD
   P1SEL2 = BIT1 + BIT2;                  // P1.4 = SMCLK, others GPIO
   P2DIR = 0xFF;                        // All P2.x outputs
   P2OUT = 0;                           // All P2.x reset
   UCA0CTL1 |= UCSSEL_2;                  // SMCLK
   UCA0BR0 = 8;                         // 1MHz 115200
   UCA0BR1 = 0;                         // 1MHz 115200
   UCA0MCTL = UCBRS2 + UCBRS0;             // Modulation UCBRSx = 5
```

UCA0CTL1 &= ~UCSWRST; // **Initialize USCI state machine** }

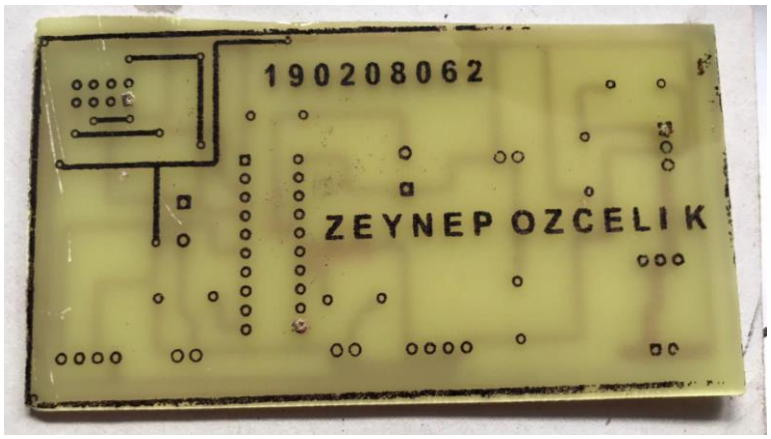
4. System Performance

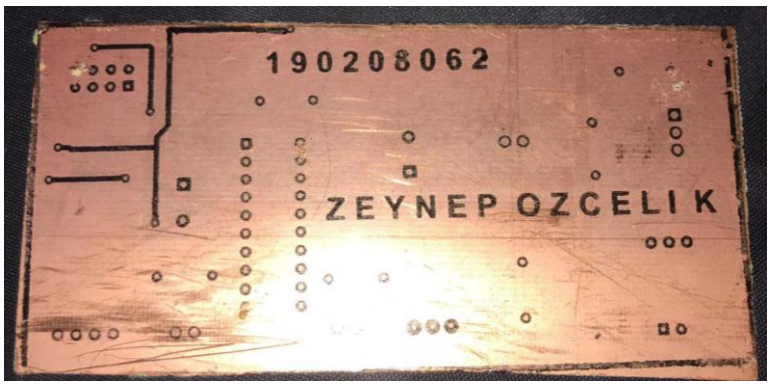
Performance can be watched through this link: https://thingspeak.com/channels/489473

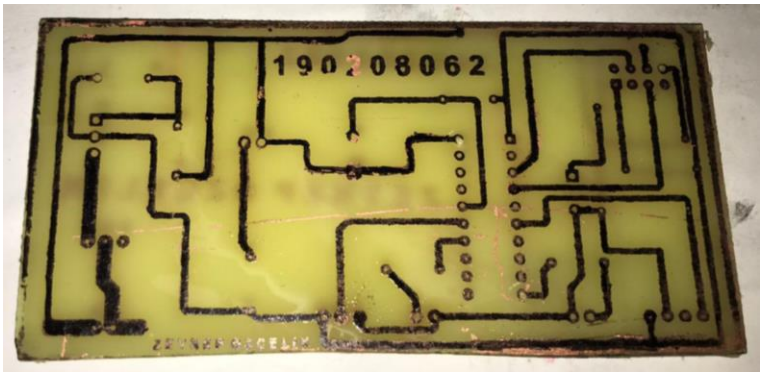


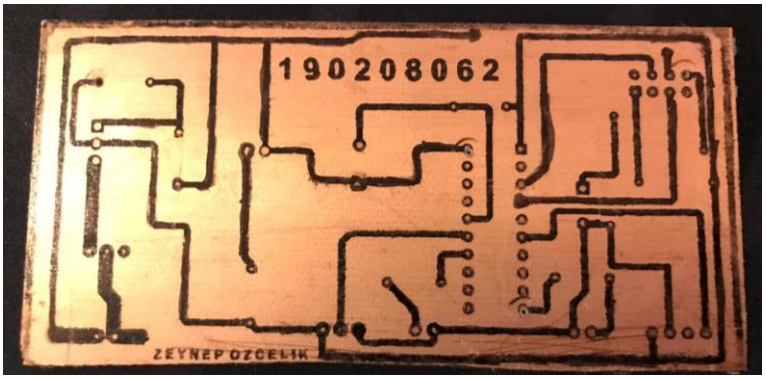


PCB Making Steps :









View of The Circuit :

References :

- https://thingspeak.com/channels/1739599/private_show
- https://www.google.com/search?q=esp8266+at+komutlar%C4%B1&oq=esp&aqs=chrome.1.69i57j35i39j69i59l2j46i433i512j69i60l2j69i61.2209j0j7&sourceid=chrome&ie=UTF-8
- https://www.youtube.com/watch?v=EEzcNje0Zk8
- https://www.google.com/search?q=msp430+family+user+guide&oq=msp430+&aqs=chrome.1.69i57j69i59l3j69i60l3j69i65.4483j0j7&sourceid=chrome&ie=UTF-8