Bare-Metal based STM32/ESP01 IoT node

Jose Miguel Estrada Aguirre
Embedded Systems Engineer
Merida, Yucatán. México
Email: Miguelest072018@outlook.com

Abstract

This paper is the document in which I will demonstrate how was developed the design of a circuit to obtain an interface with the ESP-01 by using the STM32F103B microcontroller which will bring us the possibility to control the module via UART communication using AT commands.

The development of the minimum circuit and firmware is done over the STM32F103B integrated in the ”blue pill” board using entirely Bare-Metal register programming, which will allow us to have entirely control over every register in this microcontroller.

The definition of each function to develop the communication is explained in this document, where the different attributes and specifications according to the reference manual were carried out.

Index Terms

ESP01, Thingspeak, IoT, TCP, AT, Commands, ESP01.

CONTENTS

Bare-Metal based STM32/ESP01 IoT node

I. Introduction

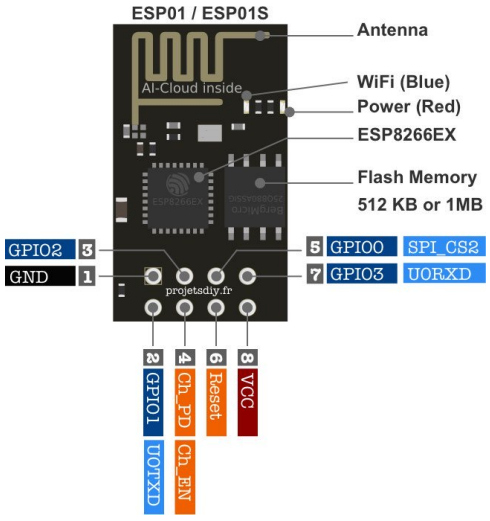
The ESP01 microcontroller from Espressif Systems is a Wi-Fi ready-to-use board that allows the internet connection programming it through AT commands using UART. This practice makes use of it to follow the proper steps to achieve an Internet connection using a 32-bit ARM microcontroller with AT commands. The platform to be connected is Thingspeak, such platform has some characteristics that we are going to show over the process of this practice.

II. ESP01

ESP-01 WiFi module is developed by Ai-thinker Team. core processor ESP8266 in smaller sizes of the module encapsulates Tensilica L106 integrates industry-leading ultra low power 32-bit MCU micro, with the 16-bit short mode,

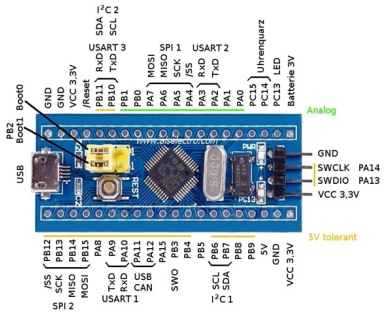
ESP-01 is a high integration wireless SOCs, designed for space and power constrained mobile platform It provides unsurpassed ability to embed Wi-FI application, with the lowest cost, and minimal space requirement with AT command capability.

Alternately, serving as a Wi-Fi adapter, wireless internet access can be added to any micro controller-based design with simple connectivity (SPI/SDIO or I2C/UART interface). Users can use the add modules to an existing device networking, or building a separate network controller. wireless SOCs, designed for space and power constrained mobile platform unsurpassed ability to embed Wi-Fi capabilities within other systems, or to function as a standalone application.

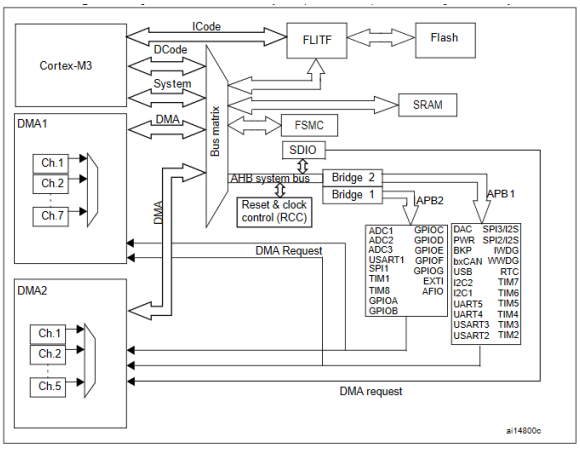


III. STM32F103B

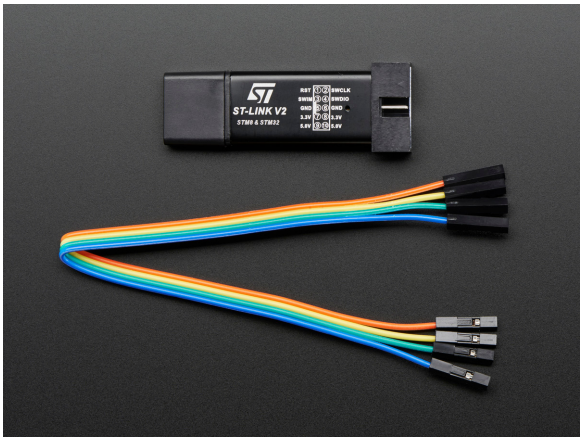
The STM32F103B medium-density performance line family incorporates the high-performance ARM, 32-bit RISC core operating at a 72 MHZ frequency, high-speed embedded memories and an extensive range of enhanced I/Os and peripherals connected to two APB buses.This microcontroller has a wide range of applications such as motor drives, application control, medical and handheld equipment, PC and gaming peripherals, GPS platforms, industrial applications, PLCs, inverters, printers, scanners, alarm system, video intercoms, and HVACs.



The structure of this module comprises of buses, DMA, inner static random access memory unit inner flash memory which operates as a master device, and all other devices linked with it, slaves.



The modules are divided into two clock domains determined by the name of the prescalers APB2 and APB1, which are dependant from the global clock and can be configured in any prescaler configuration.

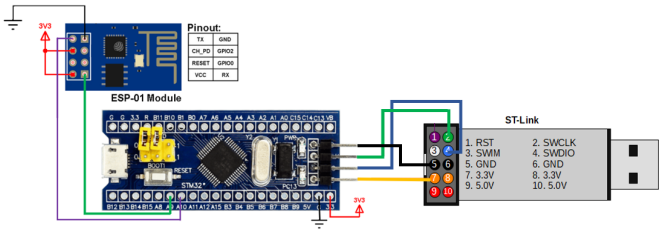


STM Devices are widely programmed using a tool named "STLINK" which vary through its different versions. in our case, the usage of the STLINK V2 was needed to load the code to the microcontroller.

IV. Minimum Circuit

The STM32 blue-pill is directly connected to the STLINK programmer, which provides voltages from 3.3 to 5 volts to our requirements which is enough for the circuit.

Once we have both circuits enabled the interface between them must be established. this can be done by connecting the UART pins over the blue pill board UART peripheral. The pin A9 and A10 corresponds to the Tx and Rx pin of such peripheral.



The minimum circuit is shown above, and we can identify that it is quite simple due the two-wire connection interface done between both microcontrollers and the facilities that the STLINK provides.

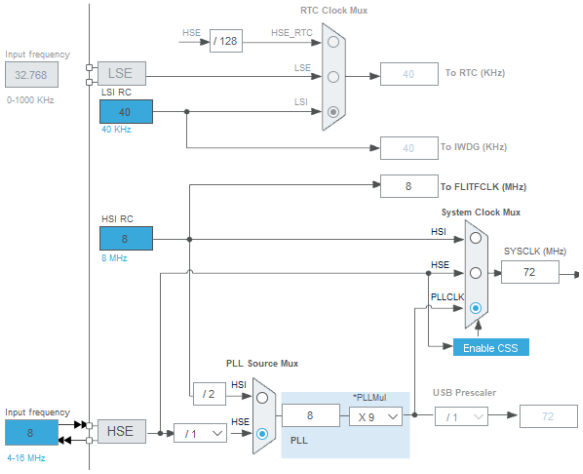
V. Minimum Firmware

Once the physical layer of the circuit is done, both microcontrollers must be configured in such a way that UART communication is done to send AT commands to the ESP01 and realize a communication protocol.

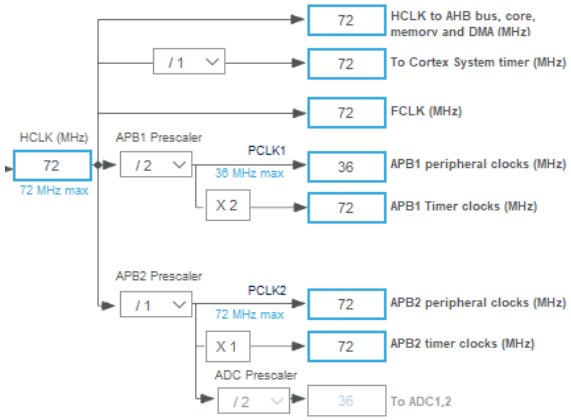
A. MCU Firmware

First at all, the STM32F1 chip can be programmed with a Hardware Abstraction Layer which determines a certain behavior of the modules, but it has different structures which make us dependents to the diverse complexity into the libraries that conforms the HAL architecture of STM Cube MX. We can take advantage of the capabilities of the STM32F1 using the register-based programming which is quite like assembly register programming but in addition with the c environment allow us to have the absolute control over the microcontroller.

1) Clock configuration: This specific microcontroller requires a certain clock configuration at the beginning of the program to configure the global clock of the system which almost has a 72 MHz velocity which can be configured to have the most performance and speed or to have the less power consumption in the board.



The image above shows that this STM32F1 has different oscillator sources. For this practice we used an HSE external oscillator which carry the main frequency of 8MHz to the PLL which multiplies the frequency to achieve the 72MHz to be configured. The clock configuration will be critical for the next following steps due the dependencies that every single module has with the clock and the prescaler that conditions each clock signal.



2) GPIO and peripheral Clock configuration: The GPIO's in STM32 must have enabled it clock sources to drive properly this microcontroller, in this case the register APBx peripheral clock enable register(APBxENR), APB2ENR from RCC allow us to enable and/or disable the prescaler clock source from some modules of this microcontroller that are in the APB2 domain.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved										TIM11 EN	TIM10 EN	TIM9 EN	Reserved		
										rw	rw	rw			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC3 EN	USART 1EN	TIM8 EN	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	IOPG EN	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.	AFIO EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

By setting the bit 2 and bit 5 we are going to enable the clock source for port D which handles the input signal of the HSE external oscillator and the port A which will handle the UART communication pins. Regarding the clock sources of each module, which as the same way to the GPIO's, the USART1 clock enable instruction is the APB2 domain too for enabling the UART module. Just by setting the 14th bit we can enable its clock source.

In the other hand, for developing a interface module we have the necessity to find a timing method to create delays. For this the Timer2 module can help to do it because it can be configured as delay function, which will be explained below, for this we need to enable the clock source for the general-purpose timer (Timer2) module, which is allocated in the APB1 domain in the APB1ENR Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		DAC EN	PWR EN	BKP EN	Res.	CAN EN	Res.	USB EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART3 EN	USART2 EN	Res.
		rw	rw	rw		rw		rw	rw	rw	rw	rw	rw	rw	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWD GEN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

By setting the bit 0 of such register, the configuration of the timer can be done.

```
RCC->APB2ENR |= (1<<5);
RCC->APB2ENR |= (1<<2);

RCC->APB1ENR |= (1<<14);
RCC->APB1ENR |= (1<<0);
```

3) Timer delay configuration: For creating our timer, we need to establish the clock timing that the timer will have, in this case the register TIMx prescaler (TIMx_PSC). TIM2_PSC allow us to divide the input frequency of the timer2 from APB1 into any frequency according to the following function:

$$Clock frequency = \frac{APB1CLK}{PSC[15:0]+1}$$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Where PSC [15:0] is the TIM2_PSC register determined by a 16-bit length number. The APB1 Clock for this timer is 72 MHz, if we use a prescaler of 71 we can obtain a 1MHz clear signal which will be helpful to deploy an easy delay function because the timer its exactly 1 microsecond length. Each timer has its auto reload register (ARR) which determines if it will be blocked if its cleared or unblocked if its reset, for our purpose we do not want a blocked timer so just by setting the 16-bit reset value FFFF in hexadecimal we can ensure we will not have a blocked timer.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

With the Timer configured we can set the first less significant bit of the control register 1 (CR1) to enable the timer2 and start the update process that has inside of the microcontroller.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Once we enabled the Timer, we need to ensure that it is properly configured, by pure luck this microcontroller help us to know about it through the status register (SR) of the Timer. The bit 0 represents the update interrupt flag indicating when the system is configuring the Timer, by some conditional we can obtain a feedback about the connection and do changes if needed.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

```
TIM2->PSC |= 72-1;
TIM2->ARR |= 0xffff;
TIM2->CR1 |= (1<<0);
while (!(TIM2->SR & (1<<0)));
```

B. UART Firmware

For configuring the UART module in this microcontroller is just follow the steps provided by the data sheet of the microcontroller, which are almost complete enough to cover different topics, so the most basic and important steps are:

- GPIO configuration: As we know, UART protocol requires two pins which are Tx and Rx, it is important to determine the direction of each pin so we can set the GPIO mode using the GPIO configuration register (GPIOx_CRH/GPIOx_CRL) depending to the port and the number of the pin, The Tx and Rx defined by the board are in A9 and A10.
  According to the following table we can observe that such registers has two independent registers MODEy and CNFy that configures each single pin according to our requirements.

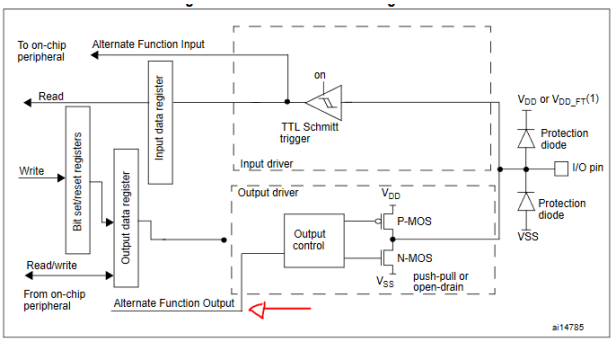
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:30, 27:26, 23:22, 19:18, 15:14, 11:10, 7:6, 3:2 **CNFy[1:0]:** Port x configuration bits (y= 8 .. 15)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table.*
In input mode (MODE[1:0]=00):
00: Analog mode
01: Floating input (reset state)
10: Input with pull-up / pull-down
11: Reserved
In output mode (MODE[1:0] > 00):
00: General purpose output push-pull
01: General purpose output Open-drain
10: Alternate function output Push-pull
11: Alternate function output Open-drain

Bits 29:28, 25:24, 21:20, 17:16, 13:12, 9:8, 5:4, 1:0 **MODEy[1:0]:** Port x mode bits (y= 8 .. 15)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table.*
00: Input mode (reset state)
01: Output mode, max speed 10 MHz.
10: Output mode, max speed 2 MHz.
11: Output mode, max speed 50 MHz.

A9 is the Tx pin, so it must be configured as a very fast output pin configured in the MODE9[1:0] register as 3 to perform the fastest output velocity achieving the best performance. In this case, the Tx pin must be set as an Alternate Function Input Output (AFIO) to enable the UART Tx driver which handle the transmission just by setting the CNF9[1:0] register as 2.



A10 is the Rx pin, so it must be configured as an input pin, which is done by configuring the MODE10[1:0] as 0 and the CNF10[1:0] as 1 to be a floating input.

```
GPIOA->CRH|= (1<<7)| (0<<6)| (1<<5)|(1<<4);
GPIOA->CRH|= (0<<11)|(1<<10)|(0<<9)|(0<<8);
```

- Control configuration: The UART modules of the STM32 bring us the total control of the module by using three different control registers to handle the USART protocol. in this case we only need to enable the UART, the Tx pin and the Rx pin.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

We can enable the UART1 module by setting the 13th bit, the Tx pin in the 3rd bit and the Rx pin in the 2nd bit. By resetting the M register in the 12th bit we can ensure a 8 bit length communication and the wake up UART responses must be set on the 11th bit.

```
USART1->CR1 = 0x00;
USART1->CR1 |= (1<<13);
USART1->CR1 |= (1<<2);
USART1->CR1 |= (1<<3);
USART1->CR1 |= ~(1<<12);
USART1->CR1 |= (1<<11);
```

- Baud Rate configuration:
  The baudrate to be configured in the UART is settle by the Baud Rate Register (BRR), which is composed by two sub registers, the mantissa, and the fraction part. Those numbers represent the prescaler needed to simulate the required baudrate from the clock which sources the module.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The following function determines the mantissa value and an approximation to the fraction.

$$TxRxbaud = mantissa.fractionaprox = \frac{fclk}{16 * baudrate}$$

For achieving the UART communication with the ESP01 we need to establish a 115200 baudrate communication. regarding the 72MHz that APB2 preescaler provides, we can obtain the mantissa 39:

$$mantissa.fractionaprox = \frac{72,000,000Hz}{16 * 115200} = 39.0625$$

Also, we got 0.0625 as an approximation of the fraction, to ensure the proper value in 16 bit lenght to make a precise division of the frequency the following function determines the fraction value:

$$fraction = 16 * fractionaprox$$

which gave us a result of:

$$fraction = 16 * 0.0625 = 1$$

so finally, the mantissa sub register must be set to 39 and the fraction sub register set to 1 achieving a 115200 baud rate generation.

```
USART1->BRR = (39<<4)|(1<<0);
```

1) UART subroutines: The UART modules inside of the STM32 has a specific register to transmit and receive data, such register is called Data Register (DR), which contains the Received or Transmitted data character, depending on whether it is read from or written to. The Data register performs a double function (read and write) since it is composed of two registers, one for transmission (TDR) and one for reception (RDR).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							DR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

For getting aware about what is happening in the module, the Status Register (SR) will give us handful information to handle the proper transmission and UART flow to ensure the best communication regarding our needs.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

- Send character
  The function used to send a character is simple, by using the data register we can assign an 8-bit length value to being transmitted, at the same time, the Status register gets aware that a transmission is being made, so we can use a conditional to wait until the transmission is complete by using the Transmission Complete sub register (TC) from the status register in bit 6.

```
void UART1_sendChar(uint8_t ch)
{
        USART1->DR = ch;
        while(!(USART1->SR & (1<<6)));
}
```

- Send string
  The function that determines how a string should be sent is by the usage of the send character function each single character of the string to be sent, this can be easily implement by a conditional that iterates the string to send it until the string is empty.

```
void UART1_sendString(char *string)
{
        while (*string)
        UART1_sendChar(*string++);
}
```

- Receive string
  The last function will be used for the next projects, and it helps to receive the string response of the ESP01 to get feedback about the AT commands sent and how behave the ESP01, such function storage all received data values into a buffer in a certain amount of time, which help us to handle the long-time responses that this module can have.

```
void UART1_getString(int us){
        tic = uwTick;
        while(uwTick - tic < us){
                r = USART1->DR;
                if(prevchar != r){
                        prevchar = r;
                        buffer[i] = r;
                        i++;
                }
        }
        for(int a = 0; a < i; a++){
                buffer[a]=0;
        }
        prevchar = 0;
        i = 0;
        r = 0;
}
```

2) AT commands: Once we got the firmware structure the next step is to follow the WIFI transmission by using the the AT commands from the ESP01 in the following order:

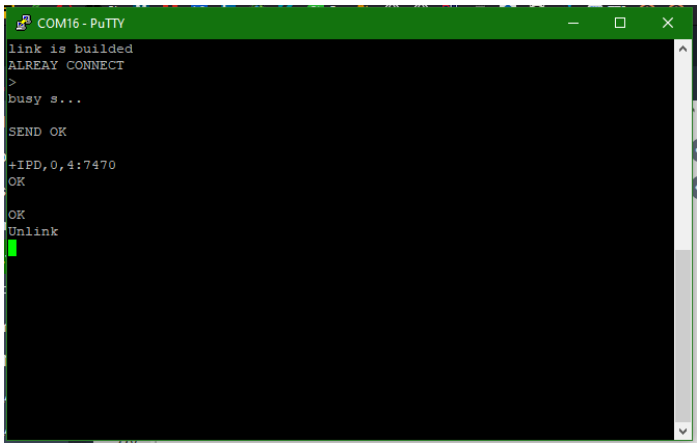
- AT: To confirm that the serial port works.
- ATE0: To disable Echo callbacks
- AT+CWMODE=1 : softAP+station mode, the command indicates which is the current mode of operation
- AT+CWJAP="SSID","Password" :SSID and password of router, This command connects the ESP module to a WiFi network. For this command, be careful with the special characters in the name of the Network or in the password.
- AT+CIPMUX: This command is used to enable multiple connections.
- AT+CIPSTART="TCP","192.168.101.110",8080 : protocol, server IP and port, this command indicates the IP assigned to the module. This command only works in operation mode 1 or 3, that is, when the ESP8266 is configured as a client or as a client / server. If the command is executed before connecting to a network, it will return as ip zeros and only the value of the MAC address.
- AT+CIPSEND= 0,47 ,this module sends data to the server.

VI. Results

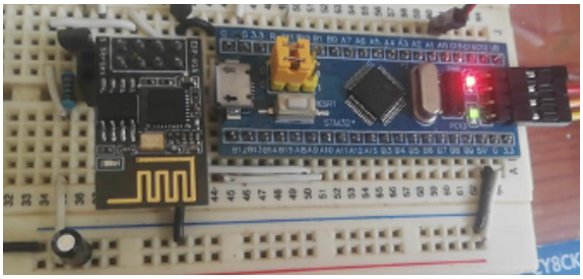
The bugs related of the usage of the commands was the first situation I faced, timing between instructions is mandatory due that the WIFI module takes its time to execute each command. So, the usage of delays about 200 milliseconds was enough to carry out a high performance.

The ThingSpeak platform has a limit of 15 seconds per transmission so we must ensure that we handle transmission waiting times longer than 15 seconds which can be carry out by using delays.

The result transmission was successfully due the communication to the server was achieved.



Now we can connect any sensor to send values over the internet using the STM32F103B.



VII. Conclusion

The process taken to develop this project shown us how a IoT device can be performed by using entirely Bare-Metal programming by using AT commands with an ESP01 , the integration of the different modules regarding its registers can be configured for developing the best communication between devices and the ESP01 was not an exception, which in addition to the theory saw in the subject we could make an IoT device with this STM microcontroller, that means that if we handle the minimum circuit and firmware properly we can embed WIFI to any existing microcontroller in the market.

References

[1] RM0008 Reference manual, STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx andSTM32F107xx advanced Arm-based 32-bit MCUs.

[2] Medium-density performance line ARM®-based 32-bit MCU STM32F103xB Datasheet.

VIII. Code

```
#include "main.h"

void SystemClock_Config(void);
void GPIO_clk(void);
void Delay_timer_Config(void);
void UART_Config(void);

void delay_us(uint16_t us);
void delay_ms(uint16_t ms);

void UART1_sendChar(uint8_t ch);
void UART1_sendString(char *string);
void UART1_getString(int us);

void handle (void);

int tic = 0;
int a = 0;
uint8_t r = 0;
int i = 0;
uint8_t prevchar = 0;
char buffer[100];
```

```
int main(void)
{
  SystemClock_Config();
  GPIO_clk();
  Delay_timer_Config();
  UART_Config();

  while (1)
  {
    handle();
  }
}

void UART_Config(void)
{
    GPIOA->CRH|= (1<<7)| (0<<6)| (1<<5)| 1<<4);
    GPIOA->CRH|= (0<<11)|(1<<10)|(0<<9)|(0<<8);
    USART1->CR1 = 0x00;
    USART1->CR1 |= (1<<13);
    USART1->CR1 |= (1<<2);
    USART1->CR1 |= (1<<3);
    USART1->CR1 |= ~(1<<12);
    USART1->CR1 |= (1<<11);
    USART1->BRR = (39<<4)|(01<<0);
}

void GPIO_clk(void)
{
    RCC->APB2ENR |= (1<<5);
    RCC->APB2ENR |= (1<<2);
    RCC->APB1ENR |= (1<<14);
    RCC->APB1ENR |= (1<<0);
}


void Delay_timer_Config(void)
{
        TIM2->PSC |= 72-1;
        TIM2->ARR |= 0xffff;
        TIM2->CR1 |= (1<<0);
        while (!(TIM2->SR & (1<<0)));
}

void delay_us(uint16_t us)
{
        TIM2->CNT = 0;
        while(TIM2->CNT < us);
}

void delay_ms(uint16_t ms)
{
        for(uint16_t i = 0; i < ms; i++)
        {
                delay_us(1000);
        }
}
```

```
void UART1_sendChar(uint8_t ch)
{
        USART1->DR = ch;
        while(!(USART1->SR & (1<<6)));
}

void UART1_sendString(char *string)
{
        while (*string) UART1_sendChar(*string++);
}

void UART1_getString(int us)
{
        tic = uwTick;
        while(uwTick - tic < us)
        {
                r = USART1->DR;
                if(prevchar != r)
                {
                        prevchar = r;
                        buffer[i] = r;
                        i++;
                }
        }
        for(int a = 0; a < i; a++)
        {
                buffer[a]=0;
        }
        prevchar = 0;
        i = 0;
        r = 0;
}




void handle (void)
{
    for(int i = 0; i < 2; i++)delay_ms(5000);
    UART1_sendString("AT+CWMODE=1\r");
    UART1_getString(200);
    delay_ms(200);
    UART1_sendString("AT+CWJAP=\"Students\",\"P0l1t3cn1c4.b1s\"\r");
    UART1_getString(200);
    delay_ms(200);
    UART1_sendString("AT+CIPMUX=1\r");
    UART1_getString(200);
    delay_ms(200);
    UART1_sendString("AT+CIPSTART=0,\"TCP\",\"184.106.153.149\",80\r");
    UART1_getString(200);
    delay_ms(200);
    UART1_sendString("AT+CIPSEND=0,47\r");
    UART1_getString(200);
    delay_ms(200);
    UART1_sendString("GET /update?apikey=3O8IDUM5E65QR7TW&field1=00\r\n");
    UART1_getString(1000);
}
```