UNIVERSIDAD
NACIONAL
DE LA PLATA

Facultad de Ingeniería - UNLP
Circuitos Digitales y Microcontroladores (E1305 / E0305)
Curso 2022 - Trabajo Práctico Nº4

Grupo 2

Alumnos: Santana Valentin, Guerrero Nicolás

Índice:

1. Control de un LED RGB

1.1 Propuesta

Realizar un programa para controlar la intensidad y el color del LED RGB con la técnica de PWM. En el kit de clases el mismo se encuentra conectado a los terminales PB5, PB2 y PB1 (RGB) a través de resistencias de limitación de 220 ohms y en forma ánodo común.

1.2 Resolución del problema

Lo primero que debemos tener en cuenta es que el LED RGB tiene una configuración de conexión en el kit de forma ánodo común, por lo cual las señales a enviar deberán estar en bajo para poder activar el LED. De igual manera, habrá que tener en cuenta los voltajes para el led de manera que este pueda funcionar de manera correcta.

Voltajes y corrientes de los led

Mediante los voltímetros del Proteus, medimos los voltajes en cada led, donde los resultados fueron:
Led Rojo: 2.93 V
Led Verde: 1.66 V
Led Azul: 1.66 V

Luego, para obtener la corriente máxima de cada uno, despejamos la misma en la fórmula V = I * R. Los datos que ya tenemos son los voltajes y las resistencias, que son las mismas para todos (220 ohms), por lo que la corriente que circula por cada led es:
Led Rojo: 13.3 mA
Led Verde: 7.55 mA
Led Azul: 7.55 mA

Para simularlo en Proteus decidimos agregar un osciloscopio para poder analizar las frecuencias de los PWM y verificar que funcionen de manera correcta, de esta manera el esquema fue el siguiente:

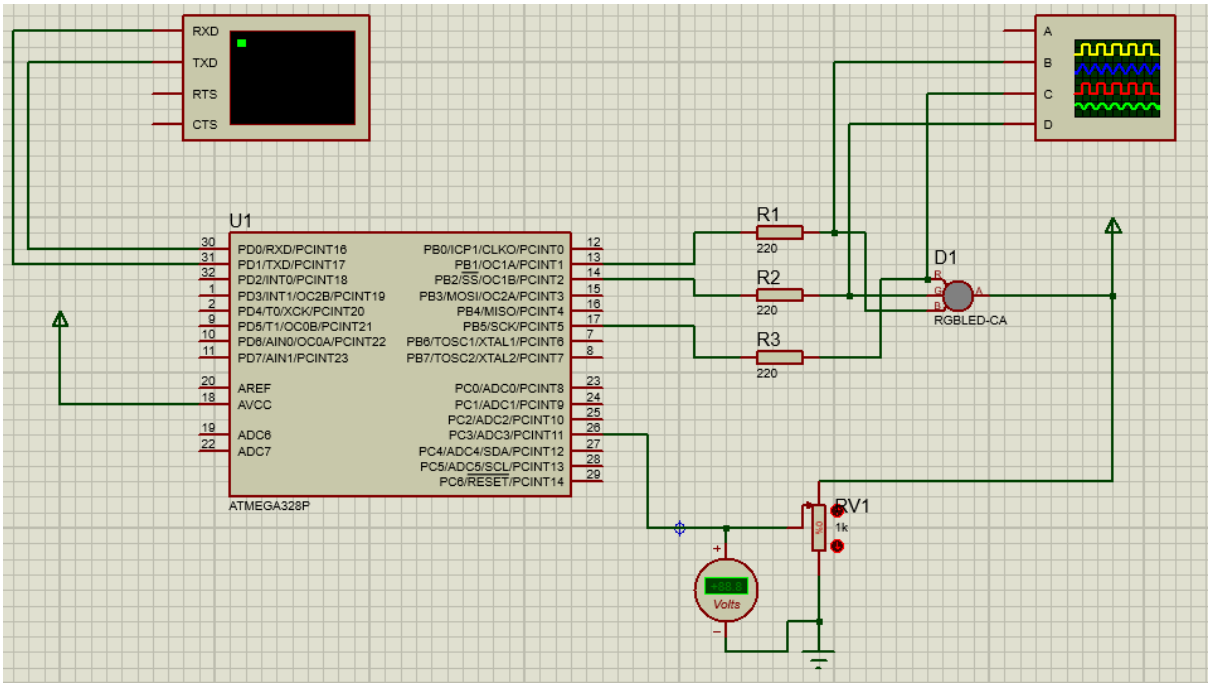

Figura 1.1 Conexión de los componentes en proteus.

2. Implementación

Para su implementación decidimos que nuestro programa se divida en diferentes aspectos:

- Por un lado, el UART será el encargado de administrar la transmisión de datos para la comunicación entre el usuario y el programa, además se encargará de verificar que los datos ingresados por el usuario sean correctos (que estén dentro de rango y sean números) para poder configurar la salida PWM.

- La librería Clock será la encargada de inicializar el Timer 0 y 1, y de recibir los datos brindados por el UART para poder configurar los módulos de los OCROA, OCR1B y OCR1A que como resultado hará que el ciclo de trabajo de la onda cuadrática varíe según la necesidad del usuario.

- El ADC se encargará de comunicarse con el potenciómetro, decodificar la información y almacenarla para luego modificar la intensidad del brillo del led según como el usuario lo prefiera.

2.1 UART

Utilizamos la librería del entregable anterior “serialPort.c” junto con su cabecera “serialPort.h”, para poder interactuar con el periférico UART. Junto con este periférico, utilizaremos otro llamado ADC.

Funciones que utilizamos de la librería nombrada en el apartado anterior:
void SerialPort_TX_Interrupt_Enable(void) → Activa las interrupciones del transmisor.
void SerialPort_RX_Interrupt_Enable(void) → Activa las interrupciones del receptor.
void SerialPort_TX_Interrupt_Disable(void) → Desactiva las interrupciones del transmisor.
void SerialPort_Send_Data(char) → Transmite un carácter.

Dentro de estas funciones se encuentra la lógica utilizada para activar o desactivar los bits del Registro B de control y estado del UART, en función de lo requerido.

UCSRB:	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80

Figura 1.2 UART Control and Status Register B.

Mediante la interfaz serie, se selecciona la escala de color de cada LED (de 0 a 255), en la cual se implementan tres funciones:
• **uint8_t esChar(char *)** → Recibe una cadena de caracteres por parámetro y constata que estos sean números. De ser así retorna un 0, de lo contrario un 1.
• **uint8_t verificador()** → Como su nombre lo indica, hace las verificaciones correspondientes tales como, en primer lugar, utilizando la función *esChar*, corroborar que lo recibido mediante el buffer sean numeros, y que estos esten dentro del rango de valores permitidos (de 0 a 255). Si hubieron inconvenientes se retorna un 0, caso contrario un 1.
• **void definirColor()** → Si la función verificador retornó un 1, envía un mensaje de confirmación de cambio de color.

2.2 ADC

2.2.1 Inicialización y configuración del ADC

El ADC es el encargado de convertir la magnitud analógica del potenciómetro en valores digitales. Para lograr esto, configuramos este periférico, en primer lugar utilizando una función **void initADC()** que puede desglosarse en 3 partes:

(1) ADCSRA |= (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)|(1<<ADEN);

Seteamos los bits ADPS0~2 para obtener un preescaler de 128 bits, ya que al dividir la frecuencia de reloj del MCU por este, obtenemos 16Mhz/128 = 125Khz, lo que garantiza la mejor performance, y seteamos el bit ADEN para habilitar el uso del ADC.

(2) ADMUX |= (1<<REFS0)|(1<<MUX1)|(1<<MUX0)|(1<<ADLAR);

Seteamos los bits MUX0~1 para habilitar como entrada el pin ADC3, ya que en el arduino utilizado para la verificación, el potenciómetro estaba conectado al mismo. Luego, hacemos lo mismo para el bit ADLAR, a fin de obtener un resultado de 8 bits justificado a izquierda, y para REFS0, con el objetivo de seleccionar el pin AVCC, el cual está conectado a una fuente de 5V.

(3) DIDR0= (1<<ADC3D);

Por último, configuramos el registro DIDR0 para hacer que el pin ADC3 sea un pin analógico de entrada.

En segundo lugar, implementamos una función **uint8_t getData()** que se encarga de obtener el valor del potenciómetro, mediante el mecanismo de polling, de la siguiente manera:

Inicia la conversión escribiendo un 1 en el bit ADSC del ADCSRA, luego espera a que la conversión finalice encuestando el bit ADIF del ADCSRA. Una vez que ADIF =1, se lee el resultado en el ADC register y se borra el flag escribiendo 1 en ADIF. Para finalizar se retornan los 8 bits de la parte alta del ADC mediante ADCH.

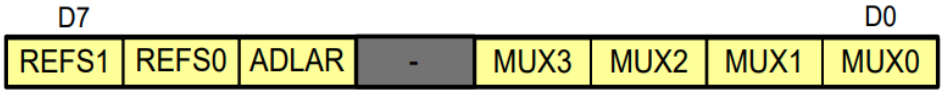

Figura 2.2.2.1 ADC Multiplexer Selection Register

Figura 2.2.2.2 ADC Control and Status Register A.

2.2.2 Administración de la intensidad del LED

Para poder realizar este mecanismo, implementamos 3 funciones:

void setValor(int, int) → Se le pasa por parámetro el valor obtenido en el UART, y se guarda en la posicion de un vector que es también pasada por parámetro dependiendo si es R(0), G(1) o B(2).

uint8_t getValor(int) → Devuelve el valor guardado en el vector anteriormente descrito.

void brillo() → Utiliza la función **getData()** explicada anteriormente y multiplica el valor por 100 y lo divide por 255 para obtener el porcentaje de la intensidad utilizada. Luego mediante un for, multiplica los valores RGB guardados en el vector por la intensidad y los divide por 100 para finalmente actualizar los valores OCR0A, OCR1A, OCR1B con los nuevos ciclos de trabajo.

2.3 PWM(Timer 0 y 1)

Para configurar las señales PWM utilizamos los Timer 0 y 1. El microcontrolador Atmega 328p tiene la salida de los OCR1A y OCR1B en los pines PB1 y PB2 por lo que los led azul y verde pueden ser configurados activando los modos PWM que brinda el Timer 1. Para el caso del led rojo que está conectado al puerto PB5 será necesario recurrir a interrupciones por software para poder simular la señal cuadrática por lo que se utilizara el Timer 0 para esta función.

Para Timer 1:

Se configuraron los puertos para generar una señal PWM de 8 bits en modo invertido. Como se pidió una frecuencia de 50hz o superior, se eligio un preescaler de 1024 ya que:

f pwm = f clk/(256*N) → f pwm= 16Mhz/(256*1024)= 61 hz

Una vez que el usuario setea los valores por consola estos a través de las funciones **SET** creadas se definen los valores de los OCR1A y OCR1B para que se genere el ciclo de trabajo correspondiente.

Para Timer 0:

Para poder simular la onda PWM invertida se configuró el modo fast PWM con un preescaler de 1024 para cumplir con la frecuencia de las otras dos ondas generadas(61 hz).

Además se utilizaron dos interrupciones: una por comparador A y otra por el comparador B.

El objetivo de esto es que mientras un módulo se encarga de generar un ciclo de periodo, el otro módulo es el encargado de generar el ciclo de trabajo para lograr la onda cuadrada.

En este caso, el OCR0B es el que se encarga de generar un ciclo de periodo enviando una señal baja cada vez que se activa la interrupción del comparador B. El OCR0A es el encargado de generar el ciclo de trabajo deseado por el usuario por lo que cada vez que ocurra su interrupción por comparador A, esta envía una señal en alto.

La elección de que se eligiera mandar una señal baja a la interrupción de un ciclo de periodo y una señal alta a la interrupción ciclo de trabajo se debe a que, como la conexión está hecha en forma ánodo común, se simuló el comportamiento de el modo PWM invertido es el más indicado para su correcto funcionamiento.

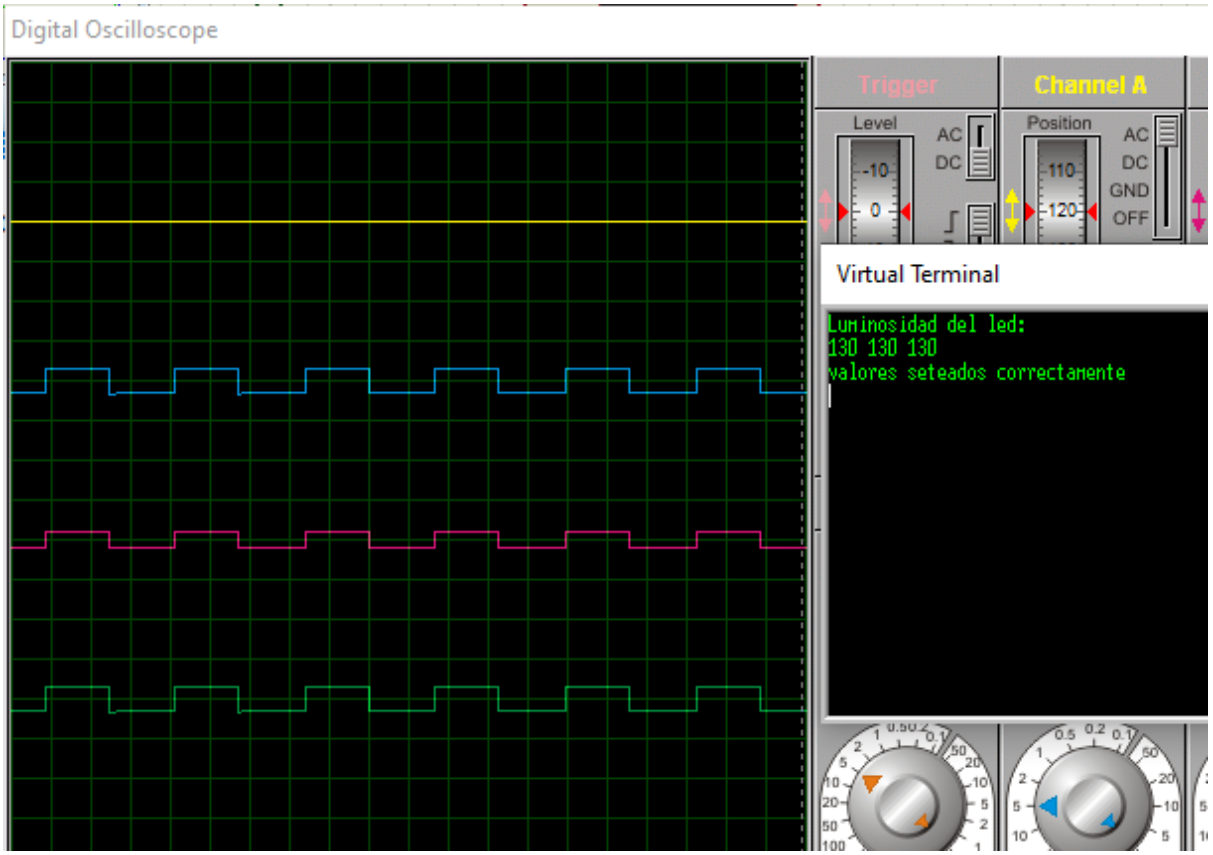

Figura 2.3.1 ondas PWM

2.3.1 Valor 0

A través de la simulación hemos encontrado un error en el caso de que se seleccione el valor 0 para el LED RGB en el cual en vez de enviar una señal en bajo constante se generan picos de subida y bajada, lo cual genera que, aunque sea mínimo, envíe suficiente voltaje para que el led continue prendido.

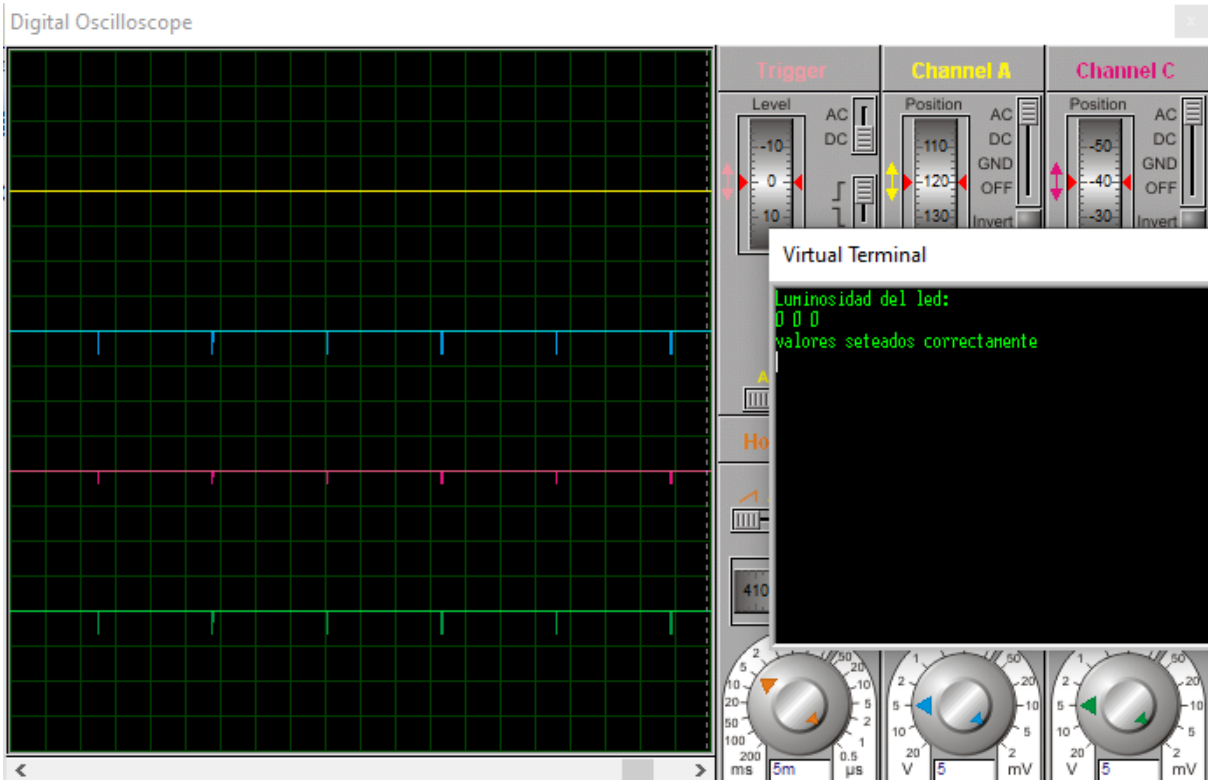

Figura 2.3.2 ADC Control and Status Register A

Esto se debe a las ecuaciones para calcular el ciclo de trabajo. El modo invertido, el cual utilizamos, no es posible llegar al 100%. Como el LED RGB está conectado de forma ánodo común se activa en bajo por lo cual la señal alta mostrada en el osciloscopio no es más que un 0% de ciclo de trabajo. Entonces es normal que si deseamos activar el led se envíe una señal en alto, lo cual en el osciloscopio reflejara una señal en bajo. El inconveniente es que para hacer esto es necesario lograr un ciclo de trabajo del 100%, pero como no es permitido por el modo invertido se generan esos picos mostrados en la **Figura 2.3.2.**

$$\text{Duty Cycle} = \frac{OCR0 + 1}{256} \times 100$$

Modo no invertido

$$\text{Duty Cycle} = \frac{255 - OCR0}{256} \times 100$$

Modo invertido

Para solucionar esto decidimos para el caso del timer 0 desactivar la interrupción de los comparadores en el caso de que el valor sea cero y enviar directamente una señal en alto que en el osciloscopio reflejara una señal baja.

De igual manera para el Timer 1 en caso de que el valor ingresado por el usuario sea cero, se desactivaran los puertos que generan la señal PWM invertida y se enviará en su lugar una señal en alto que en el osciloscopio reflejará una señal baja.

3. main

En este trabajo volvimos a utilizar la arquitectura background foreground, que se basa en interrupciones que pueden ser por eventos o temporizadas que se comunicaran a través de flags globales. La única interrupción implementada para este fin, es la de recepción que se encuentra en el UART.
Nuestro programa principal se encarga de inicializar los periféricos UART y ADC, y, los timers 0 y 1, llamando a sus respectivas funciones de inicialización. Luego se activan las interrupciones globales y mediante un super loop se espera a que ocurra un evento. Este último espera a que se reciba una cadena de datos a través del receptor de datos del UART. La interrupción RX será la encargada de activar el flag **FLAG_LINEA_RECIBIDA** cuando se termine de recibir datos. En ese momento iniciará la tarea **definirColor()** que fue explicada con anterioridad.
Además la función **brillo()** se encuentra también dentro de este super loop y no depende de ningún evento, sino que siempre está ejecutándose para poder tener un correcto valor de la intensidad dada por el potenciómetro.

4. Verificación

En este apartado solamente mostraremos los resultados obtenidos una vez completado el programa.

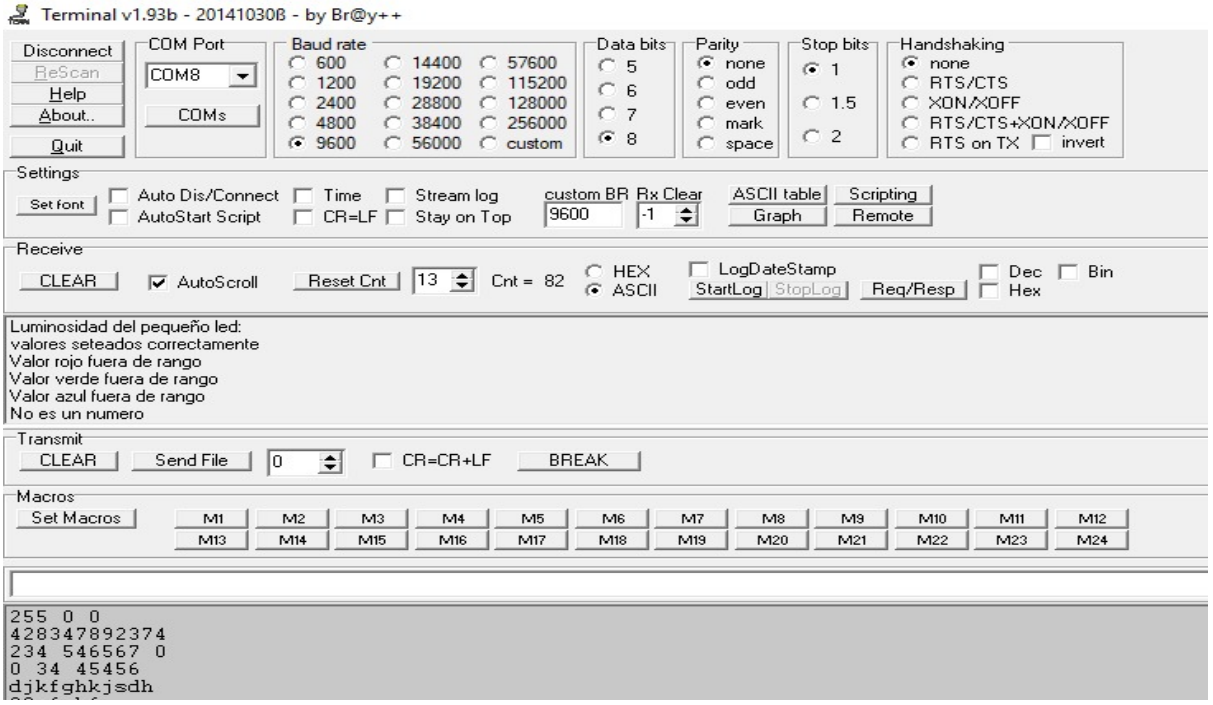

Figura 4.2 Valores probados en la terminal.

Como se ve en la imagen a la hora de seleccionar un valor máximo y mínimo el problema de los picos en la señal fue solucionado.

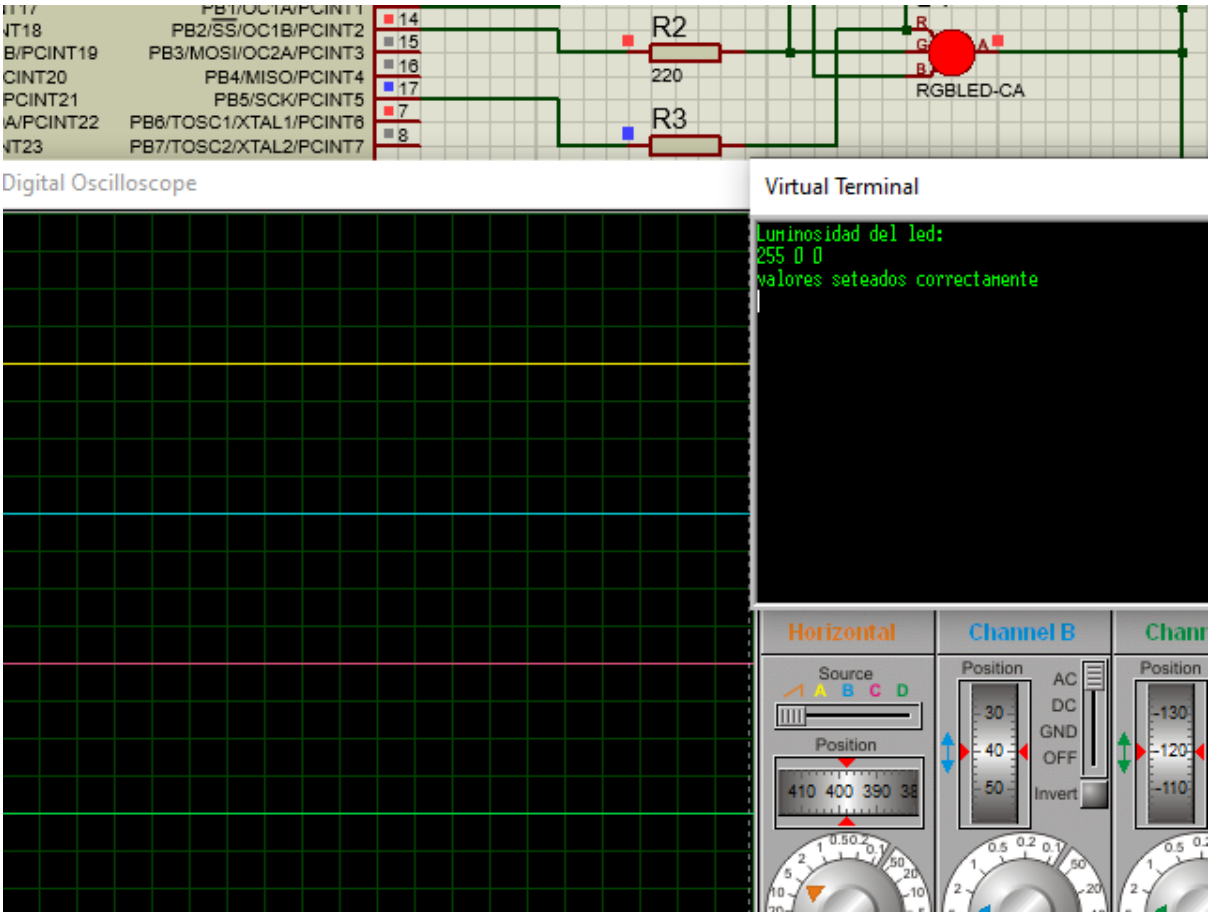

Figura 4.2 Valores probados en la terminal.

5. Anexo

5.1 Archivos cabecera

main.h

```
#ifndef MAIN_H
#define MAIN_H

#define F_CPU 16000000UL

#include "Clock.h"
```

```
#include "UART.h"
#include "ADC.h"
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>


uint8_t FLAG_LINEA_RECIBIDA;


#endif
```

Clock.h

```
#ifndef CLOCK_H
#define CLOCK_H

#include <avr/io.h>
#include <avr/interrupt.h>
void initTimer1(void);
void initTimer0(void);
void setPWMrojo(uint8_t);
void setPWMazul(uint8_t);
void setPWMverde(uint8_t);

#endif
```

ADC.h

```
#ifndef ADC_H
#define ADC_H

#include "main.h"


void initADC(void);
uint8_t getData(void);
void setValor(int,int);
uint8_t getValor(int);
void brillo();

#endif
```

UART.h

```
#ifndef UART_H
#define UART_H
	#include <avr/io.h>
	#include <string.h>
	#include <avr/interrupt.h>
	#include "main.h"
	#include "ADC.h"
	#include "serialPort.h"
	#include "clock.h"
	void init_periferico(void);
	void definirColor(void);
	uint8_t verificador(void);
	uint8_t esChar(char *);
#endif
```

SerialPort.h

```
/*
 * serialPort.h
 *
 * Created: 07/10/2020 03:02:42 p. m.
 *  Author: vfperri
 */

#ifndef SERIALPORT_H_
	#define SERIALPORT_H_

	// ------------------- Includes ----------------------------

	// Archivo de cabecera del Microcontrolador
	#include <avr/io.h>

	// Interrupciones del Microcontrolador
	#include <avr/interrupt.h>

	// -------- Prototipos de funciones Publicas ---------------

	// Inicializacion de Puerto Serie
	void SerialPort_Init(uint8_t);

	// Inicializacion de Transmisor
	void SerialPort_TX_Enable(void);
	void SerialPort_TX_Interrupt_Enable(void);
	void SerialPort_TX_Interrupt_Disable(void);
```

```
	// Inicializacion de Receptor
	void SerialPort_RX_Enable(void);
	void SerialPort_RX_Interrupt_Enable(void);

	// Transmision
	void SerialPort_Wait_For_TX_Buffer_Free(void);  // Pooling -
Bloqueante hasta que termine de transmitir.
	void SerialPort_Send_Data(char);
	void SerialPort_Send_String(char *);
	void SerialPort_Send_uint8_t(uint8_t);
	void SerialPort_send_int16_t(int val,unsigned int field_length);
//This function writes a integer type value to UART
                                          //-32768 y 32767

	// Recepcion
	void SerialPort_Wait_Until_New_Data(void);	// Pooling -
Bloqueante, puede durar indefinidamente!
	char SerialPort_Recive_Data(void);

	//Driver P.C.
	void SerialPort_Write_Char_To_Buffer ( char Data );
	void SerialPort_Write_String_To_Buffer( char *  STR_PTR );
	void SerialPort_Send_Char (char dato);
	void SerialPort_Update(void);
	char SerialPort_Get_Char_From_Buffer (char * ch);
	char SerialPort_Get_String_From_Buffer (char * string);
	char SerialPort_Receive_data (char * dato);
#endif /* SERIALPORT_H_ */
```

7.2 Archivos ".c"

main.c

```
#include "main.h"

int main(void)
{
	initADC();
	initTimer1();
	initTimer0();
	init_periferico();
	sei();
	DDRB = 0x26; // defino como salida PB1, PB2, PB5
```

```
	//DDRC &= ~(1<<PINC3); //defino como entrada el PC3
	while(1)
	{
		if(FLAG_LINEA_RECIBIDA){
			definirColor();
			FLAG_LINEA_RECIBIDA=0;
		}
		brillo();
	}

}
```

Clock.c

```
#include "Clock.h"

void initTimer1(){
	TCCR1A=(1<<COM1A1)|(1<<COM1A0)|(1<<WGM10)|(1<<COM1B1)|(1<<COM1B0);
	TCCR1B=(1<<CS12)|(1<<CS10)|(1<<WGM12);//prescaler 1024 = 61hz
}

void initTimer0(){
		TCCR0A = (1 << WGM01)|(1<<WGM00);
		TCCR0B = (1 << CS02) | (1 << CS00); // prescaler 1024 = 61hz
		OCR0B=255;
		TIMSK0 = (1 << OCIE0A)|(1 << OCIE0B);
}

void setPWMazul (uint8_t valor){
	if(valor==0){
		TCCR1A &= ~(1<<COM1A1)|(1<<COM1A0);
		PORTB |=(1<<PINB1);
	}
	else {
		TCCR1A |= (1<<COM1A1)|(1<<COM1A0);
	}
	OCR1A=valor;
}

void setPWMverde (uint8_t valor){
	if(valor==0){
		TCCR1A &= ~(1<<COM1B1)|(1<<COM1B0);
		PORTB |=(1<<PINB2);
	}
```

```
	else {
		TCCR1A |= (1<<COM1B1)|(1<<COM1B0);

	}
	OCR1B=valor;

}

void setPWMrojo (uint8_t valor){
	if(valor==0){
		TIMSK0 = (0 << OCIE0A)|(0 << OCIE0B);
		PORTB |=(1<<PINB5);
	}
	else{
		TIMSK0 = (1 << OCIE0A)|(1 << OCIE0B);
	}
	OCR0A=valor;

}

ISR(TIMER0_COMPB_vect){
		PORTB &= ~(1<<PINB5);
}

ISR(TIMER0_COMPA_vect){
		PORTB |= (1<<PINB5);
}
```

ADC.c

```
#include "ADC.h"

static uint8_t vec[3];

void initADC (){
	ADCSRA |= (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)|(1<<ADEN); //preescaler
a 128 clk 16Mhz/128 = 125Khz y habilito adc
	ADMUX |= (1<<REFS0)|(1<<MUX1)|(1<<MUX0); //ADC3  y REFS0 en 1
	ADMUX |=(1<<ADLAR);
	DIDR0= (1<<ADC3D);

}
uint8_t getData(){
	ADCSRA|= (1<<ADSC);//start conversion
```

```
	while((ADCSRA&(1<<ADIF))==0);//wait for conversion to finish
	ADCSRA|= (1<<ADIF);// borro flag
	return  ADCH;
}

void setValor(int valor, int a){
	vec[a]=valor;
}

uint8_t  getValor(int a){
	return vec[a];
}

void brillo(){
	uint8_t aux,i,intensidad;
	intensidad=(getData()*100)/255;
	for(i=0;i<3;i++){
		aux=(getValor(i)*intensidad)/100;
		switch(i){
			case 0:
			OCR0A=aux;
			break;
			case 1:
			OCR1B=aux;
			break;
			case 2:
			OCR1A=aux;
			break;
		}
	}
}
```

UART.c

```
#include "UART.h"
static char msg1[] = "Luminosidad del led: \r";
static char RX_Buffer[32];
static char * mensaje;
void init_periferico(){
	UCSR0B = (1<<RXEN0) | (1<<TXEN0);
	UCSR0C = (1<<UCSZ01) | (1<<UCSZ00);
	UBRR0L = 103;//9600bp
	mensaje=msg1;
	SerialPort_TX_Interrupt_Enable();
```

```
	setPWMrojo(0);
	setPWMverde(0);
	setPWMazul(0);
}

void definirColor(){
	if(verificador()){
		mensaje="valores seteados correctamente \r";
		SerialPort_TX_Interrupt_Enable();
	}
}

uint8_t verificador(void){
		char * token = strtok(RX_Buffer, " ");
		int valor;
		if(esChar(RX_Buffer)==0){
			mensaje="No es un numero \r";
			SerialPort_TX_Interrupt_Enable();
			return 0;
		}
		valor=atoi(token);
		if((valor>-1)&&(valor<256)){
			setPWMrojo(valor);
			setValor(valor,0);
		}
		else {
			mensaje="Valor rojo fuera de rango \r";
			SerialPort_TX_Interrupt_Enable();
			return 0;
		}
		token = strtok(NULL, " ");
		valor=atoi(token);
		if((valor>-1)&&(valor<256)){
			setPWMverde(valor);
			setValor(valor,1);
		}
		else {
			mensaje="Valor verde fuera de rango \r";
			SerialPort_TX_Interrupt_Enable();
			return 0;
		}
		token = strtok(NULL, " ");
		valor=atoi(token);
		if((valor>-1)&&(valor<256)){
			setPWMazul(valor);
			setValor(valor,2);
```

```
            }
            else {
                  mensaje="Valor azul fuera de rango \r";
                  SerialPort_TX_Interrupt_Enable();
                  return 0;
            }
            return 1;
}

uint8_t esChar (char *token ){
      int i=0;
      while( token[i] != '\0'){
            if((token[i]<48)||(token[i]>57)){
                  return 0;
            }
            i++;
      }
      return 1;
}
```

SerialPort.c

```
#include "SerialPort.h"


#define TX_BUFFER_LENGTH 32
#define RX_BUFFER_LENGTH 32

volatile static unsigned char TXindice_lectura=0, TXindice_escritura=0;
volatile static unsigned char RXindice_lectura=0, RXindice_escritura=0;

static char TX_Buffer [TX_BUFFER_LENGTH];
static char RX_Buffer [RX_BUFFER_LENGTH];

// Inicialización de Puerto Serie
void SerialPort_Init(uint8_t config){
      // config = 0x67 ==> Configuro UART 9600bps, 8 bit data, 1 stop @
F_CPU = 16MHz.
      // config = 0x33 ==> Configuro UART 9600bps, 8 bit data, 1 stop @
F_CPU = 8MHz.
      // config = 0x25 ==> Configuro UART 9600bps, 8 bit data, 1 stop @
F_CPU = 4MHz.
      UCSR0B = 0;
      UCSR0C = (1<<UCSZ01) | (1<<UCSZ00);
```

```
	//UBRR0H = (unsigned char)(config>>8);
	UBRR0L = (unsigned char)config;
}


// Inicialización de Transmisor

void SerialPort_TX_Enable(void){
	UCSR0B |= (1<<TXEN0);
}

void SerialPort_TX_Interrupt_Enable(void){
	UCSR0B |= (1<<UDRIE0);
}

void SerialPort_TX_Interrupt_Disable(void)
{
	UCSR0B &=~(1<<UDRIE0);
}


// Inicialización de Receptor

void SerialPort_RX_Enable(void){
	UCSR0B |= (1<<RXEN0);
}

void SerialPort_RX_Interrupt_Enable(void){
	UCSR0B |= (1<<RXCIE0);
}


// Transmisión

// Espera hasta que el buffer de TX este libre.
void SerialPort_Wait_For_TX_Buffer_Free(void){
	// Pooling - Bloqueante hasta que termine de transmitir.
	while(!(UCSR0A & (1<<UDRE0)));
}

void SerialPort_Send_Data(char data){
	UDR0 = data;
}

void SerialPort_Send_String(char * msg){ //msg -> "Hola como andan hoy?"
20 ASCII+findecadena, tardo=20ms
```

```
	uint8_t i = 0;
	//'\0' = 0x00
	while(msg[i]){ // *(msg+i)
		SerialPort_Wait_For_TX_Buffer_Free(); //9600bps formato 8N1,
10bits, 10.Tbit=10/9600=1ms
		SerialPort_Send_Data(msg[i]);
		i++;
	}
}


// Recepción

// Espera hasta que el buffer de RX este completo.
void SerialPort_Wait_Until_New_Data(void){
	// Pooling - Bloqueante, puede durar indefinidamente!
	while(!(UCSR0A & (1<<RXC0)));
}


char SerialPort_Recive_Data(void){
	return UDR0;
}


void SerialPort_Send_uint8_t(uint8_t num){

	SerialPort_Wait_For_TX_Buffer_Free();
	SerialPort_Send_Data('0'+num/100);

	num-=100;

	SerialPort_Wait_For_TX_Buffer_Free();
	SerialPort_Send_Data('0'+num/10);

	SerialPort_Wait_For_TX_Buffer_Free();
	SerialPort_Send_Data('0'+ num%10);
}

/************************************************************
	This function writes a integer type value to UART
	Arguments:
	1)int val   : Value to print
	2)unsigned int field_length :total length of field in which the
value is printed
	must be between 1-5 if it is -1 the field length is no of digits
```

```
in the val
*************************************************************/
void SerialPort_send_int16_t(int val,unsigned int field_length)
{
	char str[5]={0,0,0,0,0};
	int i=4,j=0;
	while(val)
	{
	str[i]=val%10;
	val=val/10;
	i--;
	}
	if(field_length==-1)
		while(str[j]==0) j++;
	else
		j=5-field_length;

	if(val<0) {
		SerialPort_Wait_For_TX_Buffer_Free();
		SerialPort_Send_Data('-');
		}
	for(i=j;i<5;i++)
	{
	SerialPort_Wait_For_TX_Buffer_Free();
	SerialPort_Send_Data('0'+str[i]);
	}
}
//*********


void SerialPort_Write_Char_To_Buffer ( char Data )
{
	// Write to the buffer *only* if there is space
	if (TXindice_escritura < TX_BUFFER_LENGTH){
		TX_Buffer[TXindice_escritura] = Data;
		TXindice_escritura++;
	}
	else {
		// Write buffer is full
		//Error_code = ERROR_UART_FULL_BUFF;
	}
}

void SerialPort_Write_String_To_Buffer( char *  STR_PTR )
{
	unsigned char i = 0;
```

```
	while ( STR_PTR [ i ] != '\0')
	{
		SerialPort_Write_Char_To_Buffer ( STR_PTR [ i ] );
		i++;
	}
}

void SerialPort_Send_Char (char dato)
{
	SerialPort_Wait_For_TX_Buffer_Free(); // Espero a que el canal de
transmisión este libre (bloqueante)
	SerialPort_Send_Data(dato);
}

void SerialPort_Update(void)
{
	static char key;

	if ( UCSR0A & (1<<RXC0) ) { // Byte recibido. Escribir byte en
buffer de entrada
		if (RXindice_escritura < RX_BUFFER_LENGTH) {
			RX_Buffer [RXindice_escritura] =UDR0; // Guardar dato
en buffer
			RXindice_escritura++; // Inc sin desbordar buffer
		}
		//else
		//Error_code = ERROR_UART_FULL_BUFF;
	}
	// Hay byte en el buffer Tx para transmitir?
	if (TXindice_lectura < TXindice_escritura){
		SerialPort_Send_Char ( TX_Buffer [TXindice_lectura] );
		TXindice_lectura++;
	}
	else {// No hay datos disponibles para enviar
		TXindice_lectura = 0;
		TXindice_escritura = 0;
	}
}

char SerialPort_Get_Char_From_Buffer (char * ch)
{
	// Hay nuevo dato en el buffer?
	if (RXindice_lectura < RXindice_escritura){
		*ch = RX_Buffer [RXindice_lectura];
		RXindice_lectura++;
		return 1; // Hay nuevo dato
```

```
	}
	else {
		RXindice_lectura=0;
		RXindice_escritura=0;
		return 0; // No Hay
	}
}

char SerialPort_Get_String_From_Buffer (char * string)
{
	char rxchar=0;

	do{
		if(SerialPort_Get_Char_From_Buffer (&rxchar)){
			*string=rxchar;
			string++;
		}
		else{
			rxchar='\n'; //empty string
		}
	}while(rxchar!='\n');
	*string='\0'; //End of String
	return 1;
}

char SerialPort_Receive_data (char * dato)
{
	if ( (UCSR0A & (1<<RXC0))==1) {
		*dato=UDR0;
		return 1;
	}
	return 0; //no data
}
```