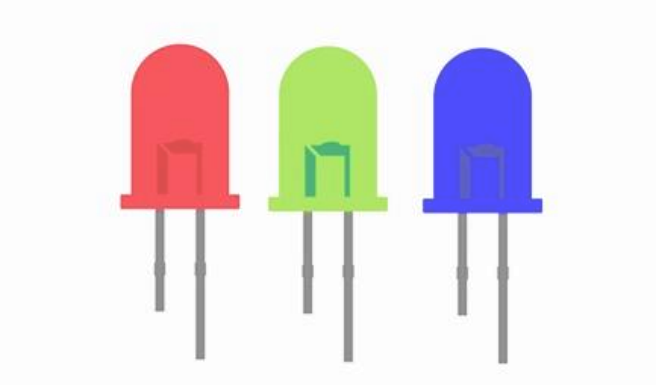
Trabajo Práctico N°04

Control de un LED RGB



Circuitos Digitales y Microcontroladores – UNLP

NOTA 10

CALDERÓN Sergio Leandro

8 de agosto de 2022

Índice

1 – Conexión de LED

1.1 – Enunciado

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 220ohms y en forma ánodo común. Para la simulación del modelo en Proteus puede utilizar RGBLED-CA.

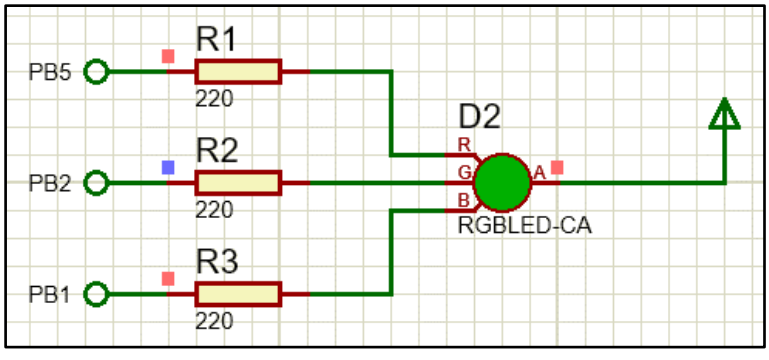
1.2 – Interpretación

Se cuenta con un LED RGB en el kit de desarrollo proporcionado por la cátedra. Para poder simular el programa y posteriormente validarlo en clase, se deberá realizar la conexión entre el microcontrolador y el componente RGBLED-CA en el simulador Proteus.

Se deberán conectar los terminales PB5, PB2 y PB1 del MCU a los terminales del diodo rojo, verde y azul, respectivamente, para poder encenderlos o apagarlos de forma independiente en un instante determinado. En cada rama, se deberá conectar una resistencia de $220\ \Omega$ para limitar la corriente. Además, se deberá utilizar la forma ánodo común, es decir, todos los pines positivos del RGB LED estarán conectados a un mismo punto VCC.

1.3 – Resolución

En el simulador Proteus, se insertó el componente RGBLED-CA de la librería *Display* y se conectó su terminal "A" (ánodo) a la fuente de alimentación V_{CC}, representado con el terminal "POWER". Por otro lado, los terminales R, G y B se conectaron a los pines PB5, PB2 y PB1 del microcontrolador, respectivamente. En cada rama se insertaron los resistores mediante el componente "RESISTOR", configurado para tener resistencias de $220\ \Omega$.



ok

Fig. 1.1. Conexión en forma ánodo común del LED RGB en Proteus.

2 – Corriente máxima

2.1 – Requerimiento

Determine la corriente máxima de cada LED.

2.2 – Datos

Se proporcionaron las tensiones requeridas para el encendido adecuado de cada LED que forma parte del componente RGB. Los valores se muestran a continuación:

Tabla 2.1. Tensión de activación para cada LED.

	LED Rojo	LED Verde	LED Azul
V_{LED} $[V]$	2,049	2,81	2,871

2.3 – Interpretación

Se deberá calcular la mayor intensidad de corriente que circula por cada rama del componente LED RGB, teniendo en cuenta los datos proporcionados de tensión para cada LED y la tensión de alimentación, así como también la tensión en los pines involucrados del MCU.

También se deberán actualizar las tensiones de activación en el simulador Proteus para poder realizar la simulación y verificar los resultados calculados.

2.4 – Resolución

2.4.1 – Caso ideal

En la forma ánodo común, la tensión de alimentación V_{CC} es la misma para todos los LED, que es 5V si consideramos el caso ideal. Luego, cada LED se encuentra conectado a un pin del microcontrolador. Para que se encuentre encendido, dicho pin debe encontrarse en un nivel bajo (0V en caso ideal), de modo que la corriente circule en sentido $V_{CC} \rightarrow PBn$.

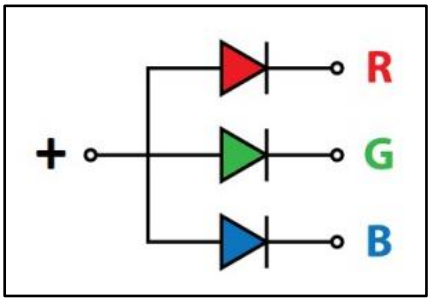


Fig. 2.1. Descomposición del componente LED RGB en forma ánodo común.

Aplicando la 2° Ley de Kirchoff, o ley de las tensiones, se determina la siguiente expresión para cada rama, donde V_{LED} es la tensión de activación del LED correspondiente, V_{RES} es la caída de tensión producida por la resistencia conectada en dicha rama y V_{OL} es la tensión de nivel bajo en el terminal del microcontrolador.

$$V_{CC} - V_{LED} - V_{RES} - V_{OL} = 0$$

Despejando se obtiene la tensión V_{RES}, para luego calcular I, conociendo R

$$V_{RES} = V_{CC} - V_{LED} - V_{OL} = I_n * R \quad \Leftrightarrow \quad I_n = \frac{V_{RES}}{R} = \frac{V_{CC} - V_{LED} - V_{OL}}{R}$$

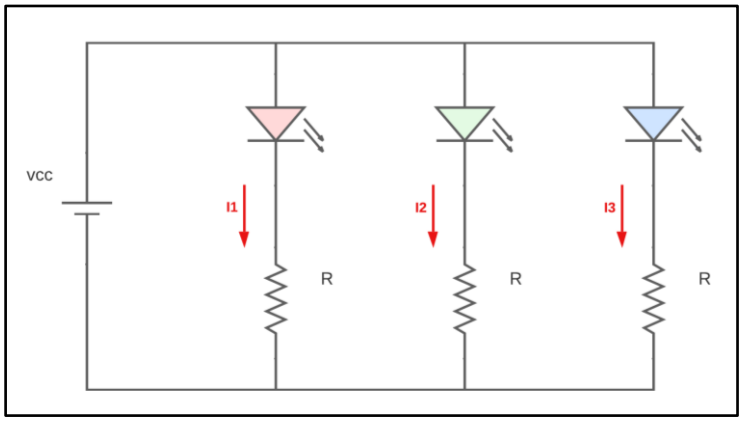


Fig. 2.2. Circuito equivalente para el componente RGB y las resistencias.

muy bien

Reemplazando los valores correspondientes para cada rama se obtiene:

$$I_1 = \frac{5V - 2{,}049\ V - 0V}{220\ \Omega} = \frac{2{,}951\ V}{220\ \Omega} = 0{,}0134\ A = 13{,}4\ mA$$

$$I_2 = \frac{5V - 2{,}81\ V - 0V}{220\ \Omega} = \frac{2{,}19\ V}{220\ \Omega} = 0{,}00995\ A = 9{,}95\ mA$$

$$I_3 = \frac{5V - 2{,}871\ V - 0V}{220\ \Omega} = \frac{2{,}129\ V}{220\ \Omega} = 0{,}00968\ A = 9{,}68\ mA$$

La corriente máxima que circula por el LED rojo es 13,4 mA, la máxima por el LED verde es 9,95 mA, y por el LED azul es 9,68 mA.

2.4.2 – Caso real

De acuerdo a la documentación del microcontrolador, la tensión de nivel bajo V_{OL} en los terminales no es $0V$, sino un valor menor a $0{,}7V$ que depende de la corriente que circula hacia dicho terminal (I_{OL}) y la temperatura. Los valores típicos se muestran a continuación.

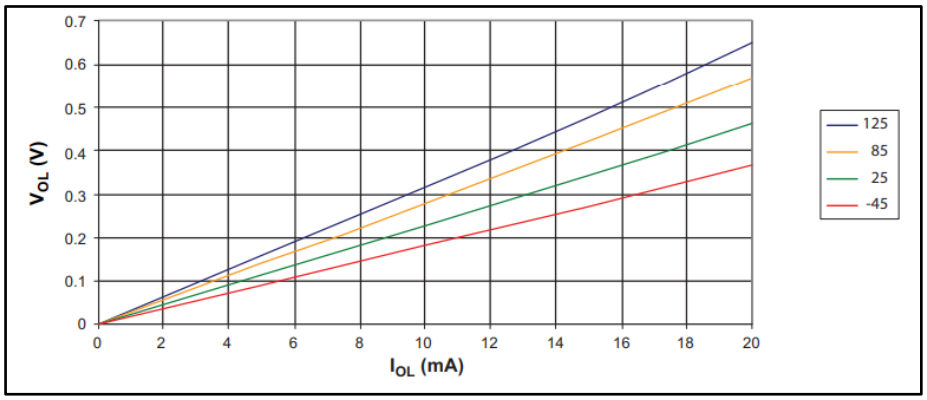


Fig. 2.3. Tensión de nivel bajo para $V_{CC} = 5V$ en los terminales del ATmega328P.

Sin embargo, las corrientes calculadas con estos valores serán menores a los resultados del caso ideal, ya que la diferencia de tensión $V_{CC} - V_{LED} - V_{OL}$ es mayor en dicho caso ideal.

Tomando I_1, I_2 e I_3 y temperatura de 85°C para obtener V_{OL} de cada rama se obtiene:

$$I_1{}' = \frac{5V - 2{,}049\ V - 0.25V}{220\ \Omega} = \frac{2{,}701\ V}{220\ \Omega} = 0{,}0123\ A = 12{,}3\ mA$$

$$I_2{}' = \frac{5V - 2{,}81\ V - 0.18V}{220\ \Omega} = \frac{2{,}01\ V}{220\ \Omega} = 0{,}00914\ A = 9{,}14\ mA$$

$$I_3{}' = \frac{5V - 2{,}871\ V - 0.17V}{220\ \Omega} = \frac{2{,}129\ V}{220\ \Omega} = 0{,}0089\ A = 8{,}9\ mA$$

Por lo tanto, las corrientes máximas son las calculadas en la sección 2.4.1.

Tabla 2.2. Tensión de activación y corriente para cada LED.

	LED Rojo	LED Verde	LED Azul
$V_{LED}\ [V]$	2,049	2,81	2,871
$I_{LED,max}\ [mA]$	13,4	9,95	9,68
$I_{LED}\ [mA]$	12,3	9,14	8,9

Muy buena la verificación con las especificaciones del fabricante

2.5 – Simulación

En primer lugar, se actualizaron los parámetros “Forward Voltage” de cada LED del componente RGBLED-CA en el simulador Proteus, como se muestra a continuación.

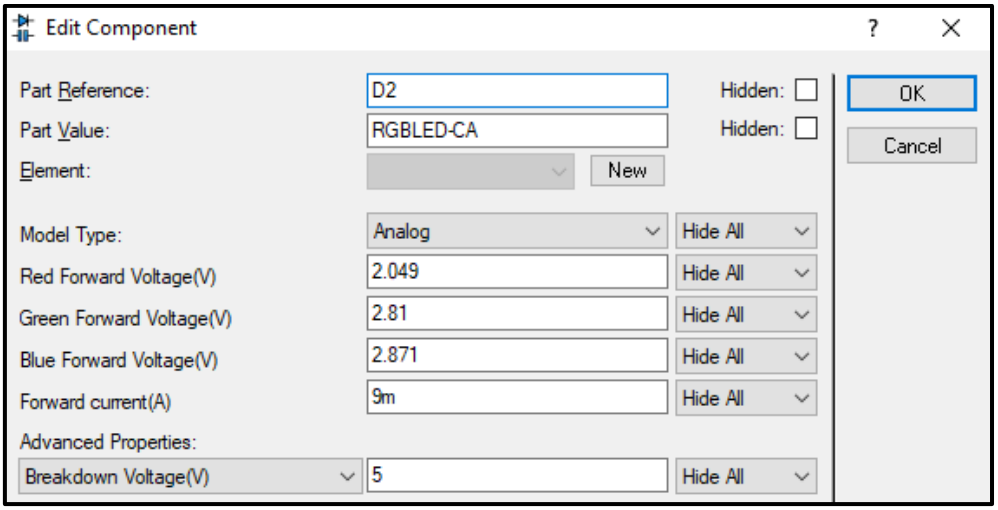
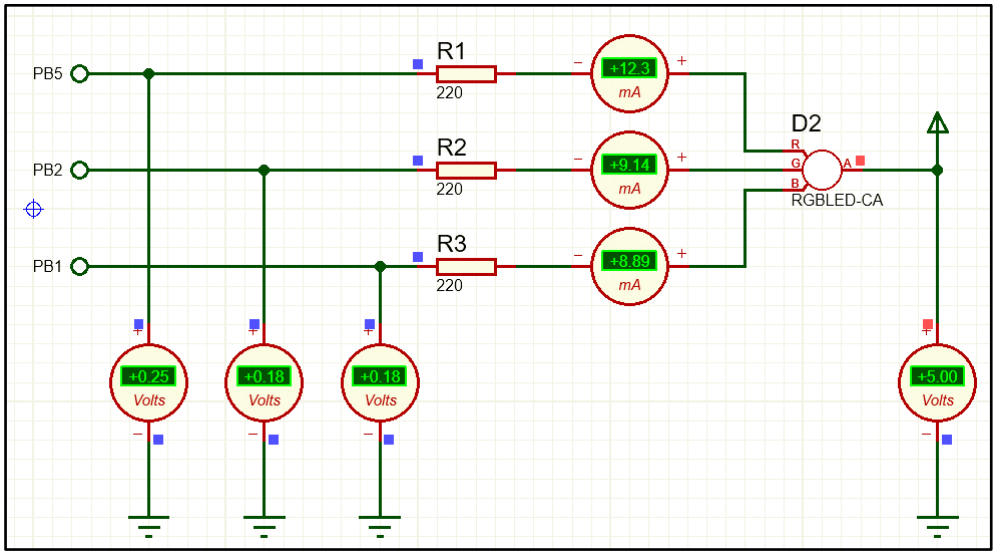


Fig. 2.4. Edición final del componente RGBLED-CA en Proteus.

Luego, se agregaron amperímetros y voltímetros para cada rama. También fue necesario establecer los terminales PB1, PB2 y PB5 como salida y escribir un “0” en los mismos para garantizar que la corriente circule en el sentido esperado. El resultado fue el siguiente:



muy bien

Fig. 2.5. Simulación del componente LED RGB en brillo máximo.

Se puede observar que, efectivamente, las medidas obtenidas en los amperímetros coinciden con los resultados del caso real, y son menores a las corrientes máximas calculadas en el caso ideal. Los voltímetros, por su parte, evidencian que $V_{OL} > 0$ en las tres ramas.

3 – Señales PWM

3.1 – Requerimiento

Genere en los tres terminales de conexión, tres señales PWM de frecuencia 50Hz o mayor con una resolución de 8 bits cada una.

3.2 – Interpretación

Se deberá generar una señal PWM para cada diodo del LED RGB mediante alguno de los Timer disponibles en el ATmega328P. La frecuencia de las señales generadas deberá ser de 50 Hz como mínimo, es decir, mayor a la del ojo humano. Además, deberán contar con 8 bits de resolución, de modo que se puedan identificar 256 valores distintos por señal.

3.3 – Resolución

3.3.1 – Análisis de situación

Cada uno de los Timer de ATmega328P permite la generación de señales PWM a través de los terminales OCnA y OCnB. En el kit de desarrollo provisto, el pin PB1, que está conectado al LED azul, coincide con la salida del generador A de Timer 1 (OC1A); y PB2, conectado al LED verde, coincide con la salida del generador B de Timer 1 (OC1B).

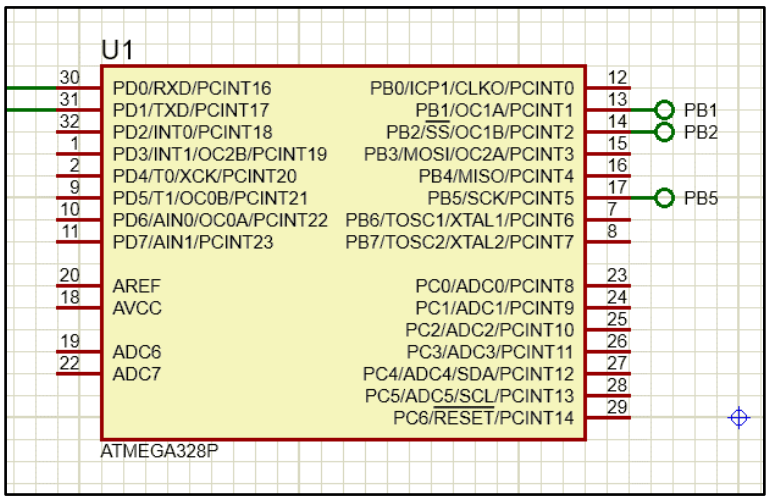


Fig. 3.1. Terminales del microcontrolador ATmega328P.

Sin embargo, como se observa en la Figura 3.1, el terminal conectado al LED rojo, PB5, no se corresponde con ninguna salida de generador PWM. Para este caso, se debe generar la señal PWM vía software, modificando el nivel de la línea manualmente cada cierto tiempo.

3.3.2 – Parámetros de Timer 1

En primer lugar, para generar señales con resolución de 8 bits, se debe limitar el contador TCNT1 a 8 bits, estableciendo como límite TOP al valor 0x00FF (255). Por lo tanto, se asigna el modo 5 al Timer 1 (Fast PWM de 8 bits), de acuerdo a la siguiente tabla:

Tabla 3.1. Modos disponibles del Waveform Generator en Timer 1.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, phase correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, phase correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, phase correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, phase and frequency correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, phase and frequency correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, phase correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, phase correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Además, se debe comprobar que existe un divisor de prescaler N que satisface la frecuencia mínima requerida de 50 Hz. Para una frecuencia del sistema de 16 MHz:

$$f_{OC1x} = \frac{f_{clk\ IO}}{256 * N} \geq 50\ Hz \quad \Rightarrow \quad N \leq \frac{16\ MHz}{256 * 50\ Hz} = 1250 \quad \Rightarrow \quad N = 1024$$

Tabla 3.2. Divisores de prescalers disponibles para Timer 1.

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (no prescaling)
0	1	0	$clk_{I/O}$/8 (from prescaler)
0	1	1	$clk_{I/O}$/64 (from prescaler)
1	0	0	$clk_{I/O}$/256 (from prescaler)
1	0	1	$clk_{I/O}$/1024 (from prescaler)

Luego, para N = 1024, se obtiene la siguiente frecuencia de señal PWM:

$$f_{OC1x} = \frac{f_{clk\ IO}}{256 * N} = \frac{16\ MHz}{256 * 1024} = \frac{2^4 * 10^6\ Hz}{2^{18}} \cong 61{,}03\ Hz$$

ok

PWM es una técnica de modulación digital donde la información útil de la señal se encuentra en el ancho del pulso. La modificación de este último se consigue mediante los registros OCR1x en Timer 1, manteniéndose la frecuencia constante. En el modo Fast PWM, se producen flancos en la señal generada OC1x cuando el contador autoincrementado TCNT1 toma el valor establecido en OCR1x o luego de alcanzar el límite TOP.

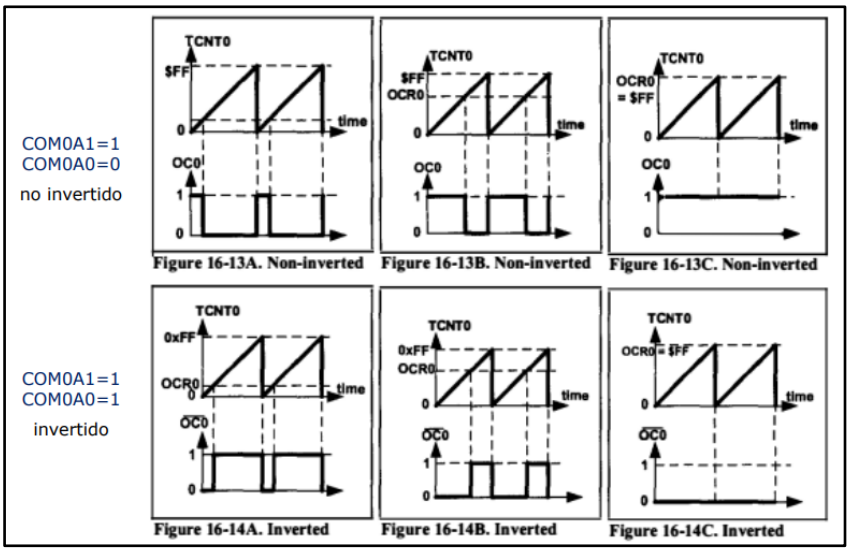


Fig. 3.2. Ejemplos de señales generadas en modo Fast PWM (similar en Timer 0 y 1)

Para una intensidad máxima en el LED, se debe generar un 0 permanente. En caso de elegir el modo invertido, se consigue asignando el valor TOP al registro OCR1x.

$$\text{Duty Cycle} = \frac{\text{Top} - \text{OCR1x}}{\text{Top} + 1} \times 100$$

Ecuación 3.1. Ciclo de trabajo en modo invertido de Timer 1.

Se decidió entonces elegir dicho modo, ya que se puede interpretar como que el registro OCR1x será proporcional al brillo del color indicado. Se indican algunas características (mínimo, máximo y resolución) del ciclo de trabajo, indicado como "p", entre 0 y 1:

$$p_{min} = \frac{255 - 255}{255 + 1} = 0 \qquad p_{max} = \frac{255 - 0}{255 + 1} \cong 0.996$$

$$\Delta p_{min} = \pm\frac{255 - 254}{255 + 1} = \pm\frac{1}{256} \cong \pm 0.0039$$

Se deduce entonces que un incremento o decremento unitario en la escala de 0 a 255 de produce una variación del 0,39% del valor medio de I_{LED}, o bien, de 19,5 mV en el pin. ok

3.3.3 – Parámetros de Timer 0

Para la generación de la señal PWM del LED rojo se decidió utilizar Timer 0. En este caso, los flancos de la señal generada no son producidos por un "Waveform Generator", sino por interrupciones. Se debe imitar el comportamiento establecido para Timer 1, entonces se configura un mismo prescaler y el modo Normal, para que TCNT0 tome valores entre 0 y 255.

Tabla 3.4. Modos de funcionamiento disponibles en Timer 0.

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on(1)(2)
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, phase correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, phase correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

De forma análoga a Fast PWM, se producirán flancos cuando el contador TCNT0 tome el valor OCR0A, por interrupción Compare Match A, y luego de alcanzar el valor MAX, por interrupción de Overflow. Dichas interrupciones deben ser habilitadas en la inicialización, para que luego en la rutina de servicio de interrupción asociada se realice la acción indicada.

3.3.4 – Módulo "Reloj"

En el proyecto se creó un archivo o módulo dedicado a la inicialización de los Timer y la manipulación directa sobre los registros de los mismos. Respecto a Timer 1 son:

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 3.3. Registro de Control "A" de Timer 1.

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 3.4. Registro de Control "B" de Timer 1.

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 3.5. Registro Máscara de Interrupciones de Timer 1.

Se implementó una función *RELOJ_T1_Init* que sigue el siguiente algoritmo:

Deshabilitar interrupciones en TIMSK1

Asignar modo 5 mediante bits WGM1[3:0]

Detener reloj interno asignando prescaler 0

Reiniciar registros de comparación OCR1x

Reiniciar contador TCNT1

Pseudocódigo 3.1. Función pública de inicialización de Timer 1.

La asignación del prescaler se realiza en una función *RELOJ_Start_Both* que lo realiza también para el Timer 0, activando los bits CSn2 y CSn0 para ambos casos (N = 1024).

Respecto a la asignación del modo invertido, existen dos funciones para actuar sobre alguna de las señales generadas: OC1A o OC1B, sin producir interferencias entre sí. Las funciones son *RELOJ_T1_PWM_x_Enable*, donde "x" es A o B. Sus contrapartes son *RELOJ_T1_PWM_x_Disable*, que desconectan la señal, según la siguiente tabla:

Tabla 3.3. Configuración del modo de OC1x mediante bits COM1x1:0 de Timer 1.

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 14 or 15: Toggle OC1A on compare match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on compare match, set OC1A/OC1B at BOTTOM (non-inverting mode)
1	1	Set OC1A/OC1B on compare match, clear OC1A/OC1B at BOTTOM (inverting mode)

Para Timer 0 se trabaja sobre los mismos registros análogos que en Timer 1, es decir, sobre TCCR0A, TCCR0B y TIMSK0, aunque la ubicación de los bits es distinta:

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 3.6. Registro de Control "A" de Timer 0.

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 3.7. Registro de Control "B" de Timer 0.

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 3.8. Registro Máscara de Interrupciones de Timer 0.

La inicialización se realiza en *RELOJ_T0_Init*, que sigue el siguiente algoritmo:

Deshabilitar interrupciones en TIMSK0

Asignar modo normal en TCCR0A

Detener reloj interno asignando prescaler 0

Reiniciar registro de comparación OCR0A

Reiniciar contador TCNT0

Pseudocódigo 3.2. Función pública de inicialización de Timer 0.

Por ambas interrupciones de Timer 0 existe una función que la habilita y deshabilita. Las mismas siguen la siguiente denominación: *RELOJ_T0_Nombre_Interrupt_Enable* y *RELOJ_T0_Nombre_Interrupt_Disable*, respectivamente, donde Nombre es OVF para el caso de la interrupción de Overflow (bit TOIE0) y COMPA para Compare Match A (bit OCIE0A).

ok

4 – Escalas de colores

4.1 – Requerimiento

Seleccione la escala de color de cada LED (de 0 a 255) mediante la interfaz serie (UART0).

4.2 – Interpretación

Mediante la terminal serie, se deberán ingresar los 3 valores de intensidad de brillo en el orden RGB, es decir, primero el correspondiente al LED rojo, luego el verde y por último el azul. Dichos números podrán ingresarse en una misma línea separados por espacios.

Cuando el usuario presione Enter, se deberá aplicar la modificación, para lo cual se actualizarán las señales asociadas a cada LED. Además, si el usuario ingresa una cantidad menor de parámetros o valores no numéricos, se deberá informar el error por la terminal.

4.3 – Resolución

4.3.1 – Biblioteca SerialPort

Para el manejo de la terminal USART, se utilizó la biblioteca SerialPort provista por la cátedra, previamente utilizada en el TP 3. Se cuenta con las siguientes funciones:

- SerialPort_Init: configura el periférico para su funcionamiento a 9600 baudios (acorde a la frecuencia del MCU), 8 bits como tamaño de dato y 1 bit de parada.
- SerialPort_RX_Enable: habilita el uso del receptor.
- SerialPort_TX_Enable: habilita el uso del transmisor.
- SerialPort_RX_Interrupt_Enable: habilita la interrupción de carácter recibido.
- SerialPort_TX_Interrupt_Enable: habilita la interrupción de transmisor libre.
- SerialPort_TX_Interrupt_Disable: desactiva la interrupción de transmisor libre.
- SerialPort_Receive_Data: retorna el carácter recibido en el registro UDR0.
- SerialPort_Send_Data(dato): escribe el carácter “dato” en el registro UDR0.

4.3.2 – Biblioteca Buffer

El almacenamiento de los parámetros ingresados y los mensajes a transmitir se realiza mediante el Buffer RX y Buffer TX, respectivamente. Ambos son estructuras que contienen un vector de un tamaño predeterminado, un índice de lectura y un índice de escritura. El acceso y manipulación de los mismos se realiza mediante las funciones de la biblioteca Buffer:

- Buffer_RX_Push(dato): agrega el carácter “dato” en el índice de escritura de RX.
- Buffer_TX_Push(dato): agrega el carácter “dato” en el índice de escritura de TX.
- Buffer_TX_PushString(texto): agrega cada carácter de “texto” al Buffer TX.
- Buffer_TX_PushLine(texto): ídem anterior, pero además agrega un salto de línea.
- Buffer_RX_Pop: retorna el carácter apuntado por el índice de lectura de RX.
- Buffer_RX_PopString(texto, len): retorna todos los caracteres entre el índice de lectura y el índice de escritura de RX a través de “texto”, de longitud máxima “len”.
- Buffer_TX_Pop: retorna el carácter apuntado por el índice de lectura de TX.
- Buffer_TX_Empty: permite verificar si hay caracteres pendientes en Buffer TX.

Ambos búferes son de comportamiento circular, y difieren en el tamaño del vector. En el caso de Buffer RX, se decidió utilizar 32 caracteres, y para Buffer TX, 196 caracteres.

4.3.3 – Arquitectura del programa

Se diseñó una arquitectura Background-Foreground, junto con el modelo de Productor-Consumidor. Para ello, se utilizan Flags, activados por las rutinas de servicio de interrupción correspondientes (parte Foreground) para que luego en el Loop principal se consulte por los mismos para realizar tareas determinadas (parte Background) y se suspenda el uso de CPU hasta que se produzca una nueva interrupción.

El modelo Productor-Consumidor se encuentra implementado mediante el uso de los búferes. En este caso, al contar con 2 búferes, se tienen 2 productores y 2 consumidores:

- Productor de comandos: es la ISR de carácter recibido (USART_RX), que agrega cada carácter al Buffer RX (función Push) hasta recibir un salto de línea.
- Consumidor de comandos: es el intérprete de comandos, que recibe el texto ingresado desde el Buffer RX (función PopString) para evaluarlo y realizar la acción indicada.
- Productor de mensajes: es el intérprete de comandos, que puede generar avisos o bien, el menú de opciones, insertándolos en el Buffer TX (función PushLine).
- Consumidor de mensajes: es la ISR de transmisor libre (USART_UDRE), que quita cada carácter del Buffer TX (función Pop) para transmitirlo a la terminal serie.

4.3.4 – Listado de comandos

Se implementó un módulo “CLI” y una utilidad “Parser” para comparar Strings, separar subcadenas mediante delimitadores y extraer números. Se decidió crear 4 comandos:

- RGB: asigna la escala de color indicada para cada LED. Requiere 3 valores enteros y sin signo, que representan la escala de color de rojo, verde y azul respectivamente.
- SEL: selecciona un LED para poder regular su brillo mediante el potenciómetro del kit de desarrollo. Requiere 1 parámetro, que puede ser “R”, “G”, “B” o “*” (color actual).
- ADC: activa o desactiva el conversor analógico digital conectado al potenciómetro del kit. Requiere 1 parámetro, que puede ser “ON” (activar) u “OFF” (desactivar).
- CLR: apaga los 3 LEDs y borra la selección realizada. No requiere parámetros.

En este apartado, se explicará el funcionamiento del comando RGB.

4.3.5 – Detección de comando

En la función “CLI_LeerComando” se debe detectar que la primera palabra del texto ingresado corresponde a alguno de los comandos mencionados. Para ello, primero se invoca a la función auxiliar “Parser_Split”, que recorre el texto y lo separa en 2 partes a partir del delimitador ‘ ’ (espacio en blanco). El pseudocódigo se muestra a continuación:

```
Inicializar índice de lectura “i” en 0
Inicializar índice de escritura “j” en 0
Obtener longitud del texto con strlen
Mientras “i” sea menor a la longitud del texto y no sea un espacio
    Copiar carácter de texto a la primera parte
    Incrementar índices
Agregar final de cadena a la primera parte
Ignorar espacio incrementando índice de lectura
Reiniciar índice de escritura a 0
Mientras “i” sea menor a la longitud del texto
    Copiar carácter de texto a la segunda parte
    Incrementar índices
Agregar final de cadena a la segunda parte
```

Pseudocódigo 4.1. Función pública Parser_Split de la utilidad Parser.

Luego, se compara la primera parte con los posibles comandos mediante otra función del Parser, llamada “Parser_Equals”, que compara los dos textos usando *strcmp*. En caso de coincidir con algún comando, se continúa con la evaluación de la segunda parte, que se trata de manera diferente de acuerdo al comando identificado. En el caso de RGB, se deben extraer los 3 números; en SEL se compara con las 4 opciones disponibles; en ADC se compara con dos opciones; y en CLR la segunda parte directamente es ignorada.

4.3.6 – Comando RGB

Para extraer los valores RGB, se invoca la función auxiliar “Parser_Convert”, que recibe la segunda parte del texto ingresado, y devuelve los 3 números por referencia. El éxito de la operación se conoce a través del código de retorno. A su vez, se utilizan funciones privadas para separar en palabras y convertir a números. El pseudocódigo es el siguiente:

```
Inicializar 3 strings auxiliares
Separar el texto en 3 palabras
Si hay menos de 3 palabras
    Devolver error de parámetros insuficientes
Convertir primera palabra a número “n1”
Si no es un número
    Devolver error de conversión
Convertir segunda palabra a número “n2”
Si no es número
    Devolver error de conversión
Convertir tercera palabra a número “n3”
Retornar estado de la última conversión
```

Pseudocódigo 4.2. Función pública Parser_Convert de la utilidad Parser.

Si la operación es exitosa se comunican los valores de brillo a un módulo “Controlador”, a través de una función “Controlador_Set_RGB”, caso contrario, se informa el error.

El controlador opera sobre el módulo “LED” y el módulo “ADC”. Para este comando, se limita a comunicar los valores al LED para que los actualice, pudiendo generar un mensaje de advertencia si el ADC se encuentra encendido, debido a que puede sobrescribirlos.

El módulo “LED”, además de actualizar los registros OCR, considera los casos en que la escala de un color es 0 o 255, pudiendo deshabilitar interrupciones o cambiar modos.

A continuación, se presentan los pseudocódigos de las funciones encargadas de asignar el valor de brillo a un LED en particular. Nótese que para Timer 0 se consideran 3 casos.

```
Si la escala es 0
        Desactivar interrupción de Overflow en T0
        Forzar apagado del LED rojo
Sino si la escala es 255
        Desactivar interrupción de Compare Match A en T0
        Forzar encendido del LED rojo
Sino
        Habilitar interrupciones de Timer 0
        Actualizar escala en OCR de Timer 0
Si fue establecido por el usuario
        Guardar escala de rojo
```

Pseudocódigo 4.3. Función “LED_UpdateRed” del módulo “LED”.

```
Si la escala es 0
        Desconectar señal OC1B
        Forzar apagado del LED verde
Sino
        Habilitar señal OC1B
        Actualizar escala en OCR1B de Timer 1
Si fue establecido por el usuario
        Guardar escala de verde
```

Pseudocódigo 4.4. Función “LED_UpdateGreen” del módulo “LED”.

```
Si la escala es 0
        Desconectar señal OC1A
        Forzar apagado del LED azul
Sino
        Habilitar señal OC1A
        Actualizar escala en OCR1A de Timer 1
Si fue establecido por el usuario
        Guardar escala de azul
```

Pseudocódigo 4.5. Función “LED_UpdateBlue” del módulo “LED”.

Como se puede observar, en todos los casos se consulta si es valor establecido por el usuario. Para ello también se recibe un parámetro “save”, que indica si se debería guardar dicha escala para una futura regulación del brillo con ADC. Para el comando RGB, es verdadero.

Respecto a valores mínimos y máximos, en ningún caso se obtiene un brillo nulo con solamente escribir el valor 0 al registro OCRnx. En Timer 0, se deshabilita la interrupción de Overflow debido a que la ISR asociada enciende el LED. En Timer 1, se desconecta la señal OC1x ya que en modo invertido el ciclo de trabajo máximo es menor al 100%.

$$\text{Duty Cycle} = \frac{\text{Top} - \text{OCR1x}}{\text{Top} + 1} \times 100$$

Ecuación 3.1. Ciclo de trabajo en modo invertido de Timer 1 (extraído de sección 3.3.2).

El brillo máximo si es posible en Timer 1 con un ciclo de trabajo del 0% al establecer el valor TOP a OCR1x, como se observa en la ecuación 3.1. Sin embargo, en Timer 0 se debe deshabilitar la interrupción de Compare Match A, ya que la ISR asociada apaga el LED.

Como última aclaración de este apartado, en los casos mencionados anteriormente se fuerza a encender o apagar el LED, según corresponda. Por ejemplo, al asignar brillo 0 de rojo, puede ocurrir que la interrupción TIMER0_COMPA está deshabilitada, por ejemplo, si el brillo anterior era 255, entonces, el LED podría permanecer encendido si no se fuerza su apagado.

4.4 – Simulación

4.4.1 – Previo a las mejoras

Para evidenciar los problemas vistos en la sección anterior, primero se realizó una simulación con un módulo “LED” simplificado, que únicamente actualiza el registro OCRnx, es decir, que para los máximos o mínimos no se utilizan interrupciones o cambios de modo.

Se conectó un osciloscopio a cada rama del LED para poder observar los valores de las señales en simultáneo. A continuación, se muestran los resultados para ambos casos.

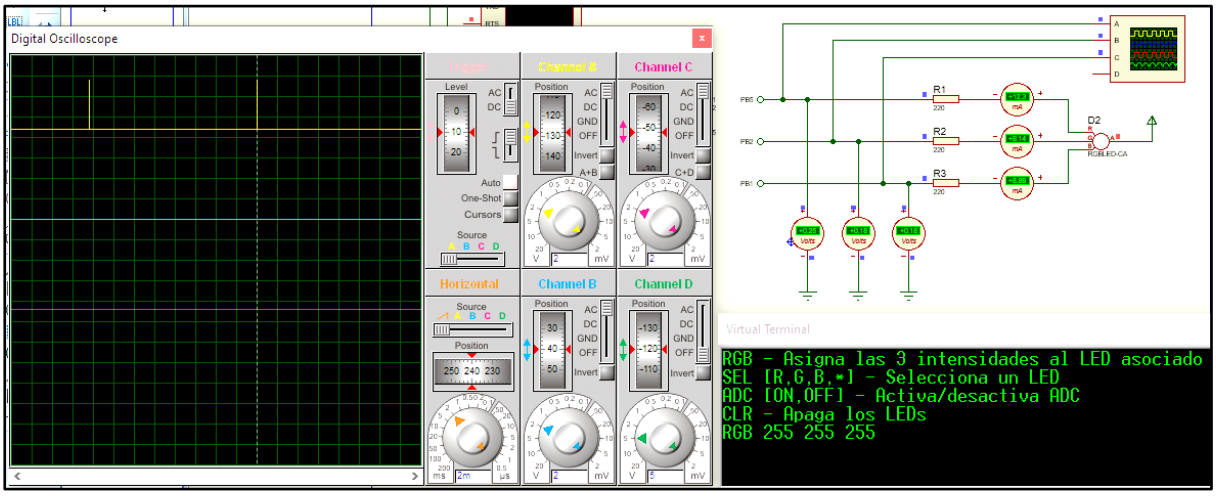


Fig. 4.1. Brillo máximo a todos los LEDs. Nótese los picos de amplitud en la señal superior.

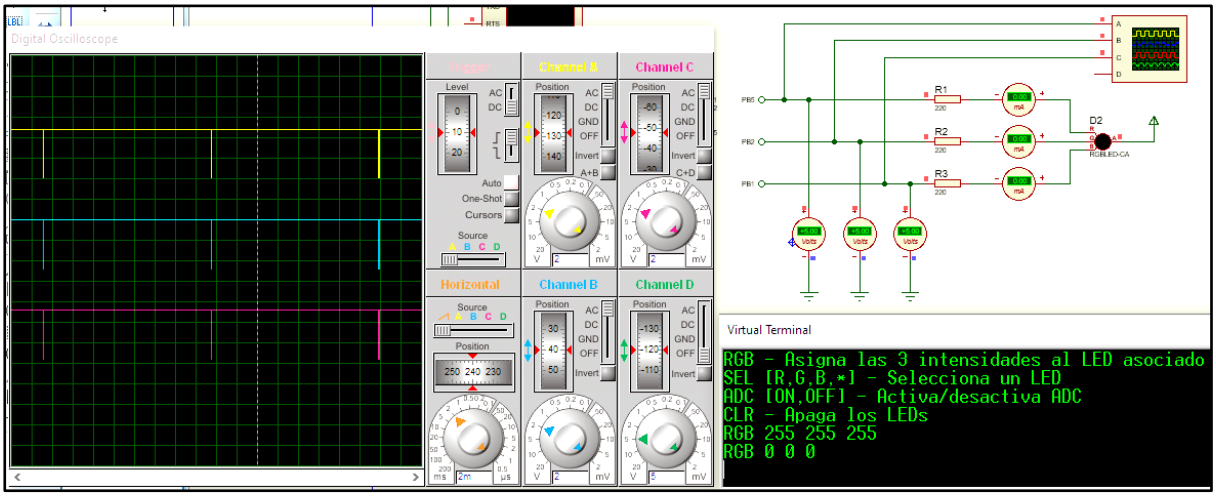


Fig. 4.2. Brillo nulo a todos los LEDs. Nótese que las 3 señales no se mantienen en nivel alto.

Aunque en la simulación puede parecer que el problema es poco relevante, ya que el LED se puede ver de color negro cuando todos están apagados, al momento de validarlo en el kit por primera vez se observó que el leve brillo de los 3 LEDs en realidad es muy notable.

muy bien

4.4.2 – Programa final

Se presentan los resultados del correcto funcionamiento del comando RGB.

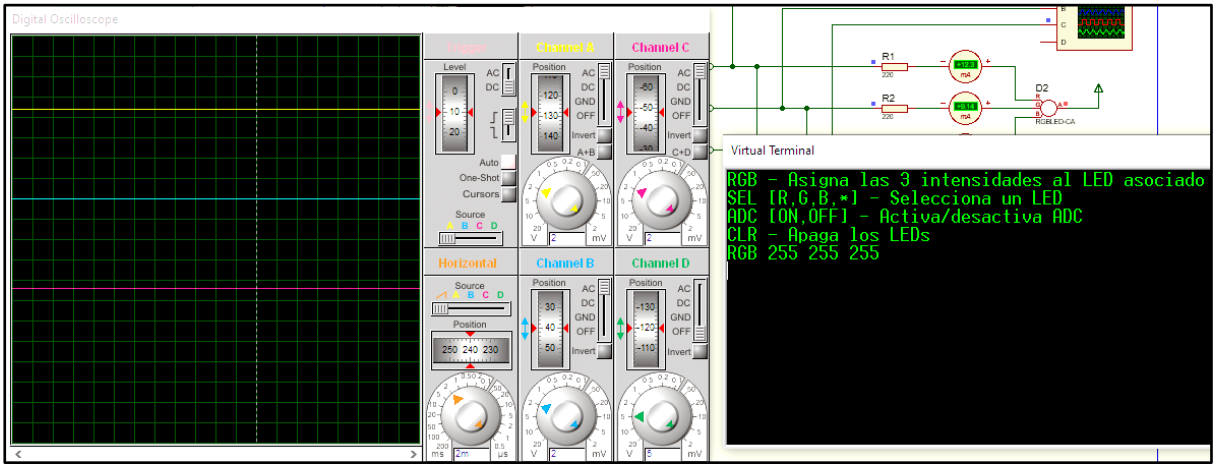


Fig. 4.3. Brillo máximo para todos los LEDs. Se observa que están siempre encendidos.

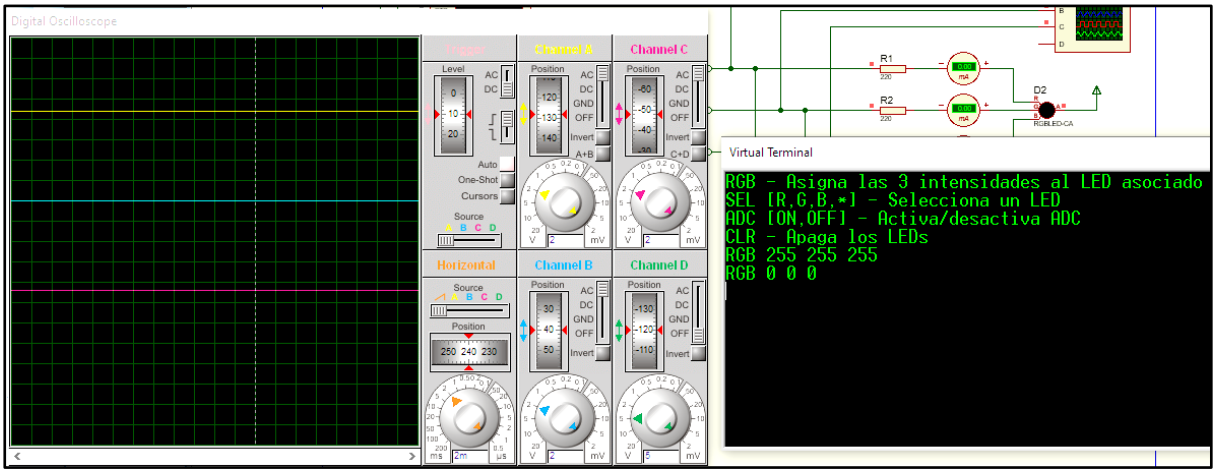


Fig. 4.4. Brillo nulo para todos los LEDs. Se observa que están siempre apagados.

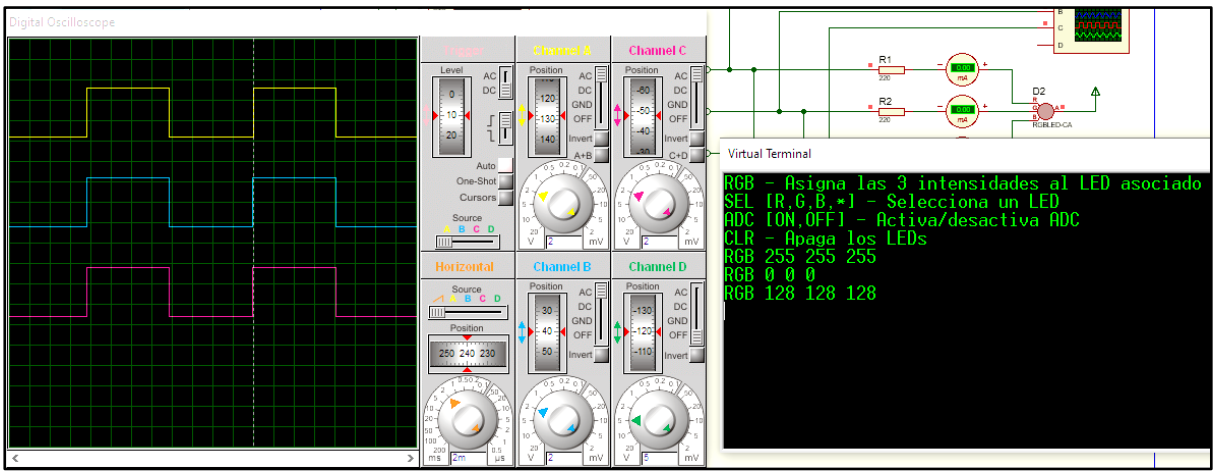


Fig. 4.5. Brillo intermedio para todos los LEDs. Se observa que se encuentran en fase.

También se verificó que el brillo nulo o máximo se aplique de forma correcta cuando solo uno o dos LEDs se encuentren en dicha condición. Se muestran ejemplos a continuación.

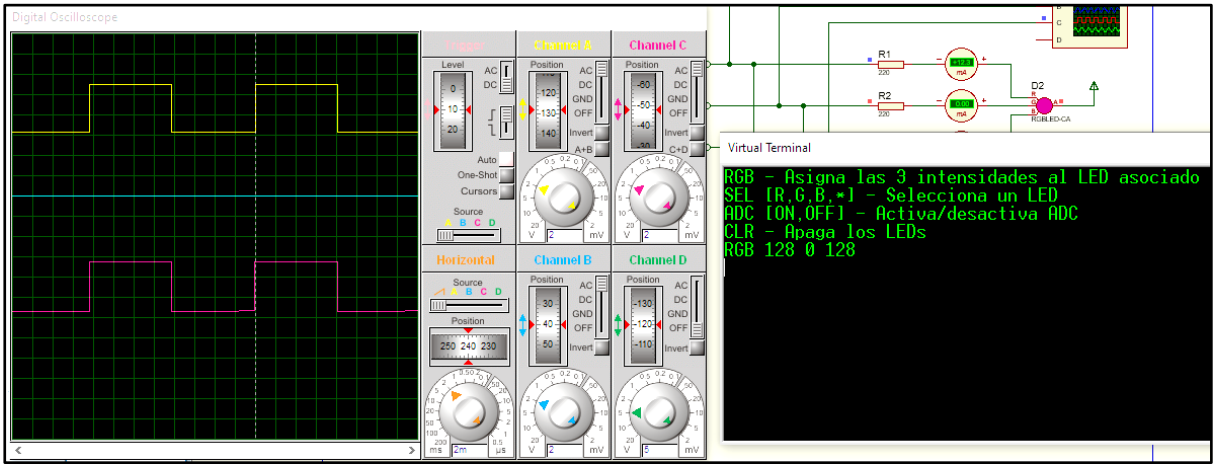


Fig. 4.6. Señales PWM para el color violeta. El LED verde se encuentra apagado.

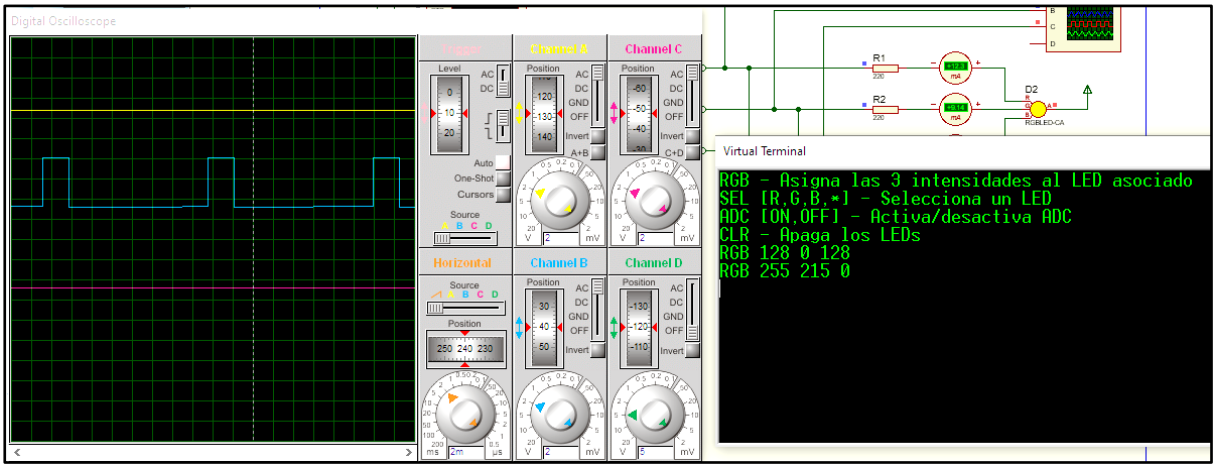


Fig. 4.7. Señales PWM para el color dorado. El LED rojo está siempre encendido y el azul apagado.

Luego se continuó con la verificación de la detección de parámetros incorrectos, primero indicando menos de 3 parámetros, y luego ingresando letras en lugar de números.

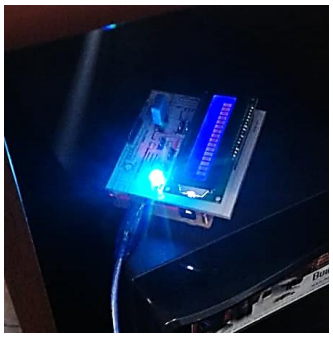
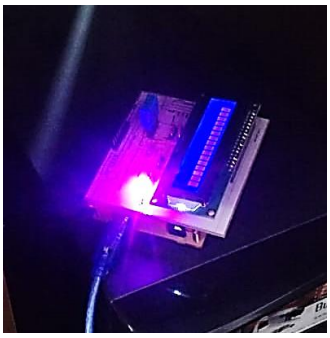
Figs. 4.8 y 4.9. Errores mostrados en la terminal para el comando RGB.

4.5 – Validación

A través de la terminal de Bray, se especificaron 8 colores mediante el comando RGB para comprobar su correcto funcionamiento en el kit de desarrollo.



Figs. 4.10, 4.11 y 4.12. Resultados de un único LED encendido al brillo máximo a la vez en el kit.

Figs. 4.13, 4.14 y 4.15. Combinaciones de 2 LEDs encendidos al brillo máximo en el kit.


Figs. 4.16 y 4.17. Resultados para brillo máximo y nulo de los 3 LEDs en el kit.

excelente

En la figura 4.17 se muestra que los 3 LEDs se apagaron completamente, ya que no se emite ninguna fracción de luz roja, verde o azul de los mismos (nota: la luz roja que se observa detrás del RGB proviene del LED de encendido de la placa Arduino).

5 – Regulación con potenciómetro

5.1 – Requerimiento

Utilice el potenciómetro (resistencia variable) del kit conectado al terminal ADC3 para controlar el brillo o intensidad del color seleccionado vía interfaz serie.

5.2 – Interpretación

Se deberá configurar el conversor analógico-digital que posee el ATmega328P de modo que se tenga como entrada analógica al terminal ADC3 (pin PC3). También se tendrá que elegir una frecuencia de prescaler del conversor para conseguir una performance aceptable.

La habilitación del ADC se podrá realizar con un comando, de modo que previamente se pueda asignar un color específico y luego se pueda regular el brillo del mismo mediante el potenciómetro conectado al terminal ADC3. Opcionalmente, el usuario podrá indicar si desea trabajar con únicamente con el LED rojo, verde o azul (no es parte del requerimiento).

5.3 – Resolución

5.3.1 – Biblioteca ADC

De igual manera que para el resto de periféricos, se diseñó una biblioteca “ADC” que se encarga de inicializar, activar, desactivar y obtener resultados de conversión. Esto evita que el módulo Controlador opere directamente sobre los registros del conversor.

Las funciones públicas implementadas se describen a continuación:

- ADC_Init: establece el terminal ADC3 como entrada analógica, selecciona el prescaler adecuado y habilita la interrupción de conversión finalizada, activando el bit ADIE.
- ADC_Enable: habilita el conversor activando el bit ADEN del registro ADCSRA.
- ADC_Disable: deshabilita el conversor desactivando el bit ADEN.
- ADC_Run: da inicio a una conversión, activando el bit ADSC de ADCSRA.
- ADC_GetResult: retorna los 8 bits más significativos del resultado de la conversión.

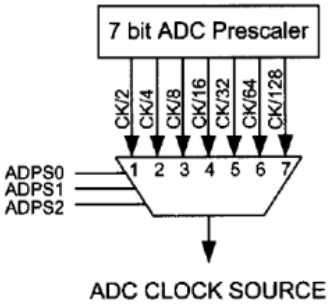
Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	

Fig. 5.1. Bits del registro ADCSRA del conversor analógico-digital.

5.3.2 – Frecuencia de conversión

El prescaler elegido para el conversor es N = 128, debido a que se garantiza una buena *performance* hasta una frecuencia ADCLK de 200 kHz, y la frecuencia del MCU es 16 MHz.

$$f_{ADCLK} = \frac{f_{IO\_CLK}}{N} \leq 200\ kHz \quad \Rightarrow \quad N \geq \frac{f_{IO\_CLK}}{200\ kHz} = \frac{16 * 10^6\ Hz}{2 * 10^5\ Hz} = 80 \quad \Rightarrow \quad N = 128$$



ADPS2	ADPS1	ADPS0	ADC Clock
0	0	0	Reserved
0	0	1	CK/2
0	1	0	CK/4
0	1	1	CK/8
1	0	0	CK/16
1	0	1	CK/32
1	1	0	CK/64
1	1	1	CK/128

Fig. 5.2. Prescalador de 7 bits del conversor analógico-digital.

Como se observa en la figura 5.2, se elige el prescaler mediante los bits ADPS2:0, que se encuentran en el registro ADCSRA mostrado previamente en la figura 5.1.

5.3.3 – Justificación de bits

Mediante el bit ADLAR del registro ADMUX, se puede elegir si los bits del resultado de conversión se encontrarán justificados a la izquierda o derecha del registro ADC. Esto se debe a que el resultado tiene 10 bits (1024 niveles), y el registro ADC es de 16 bits.

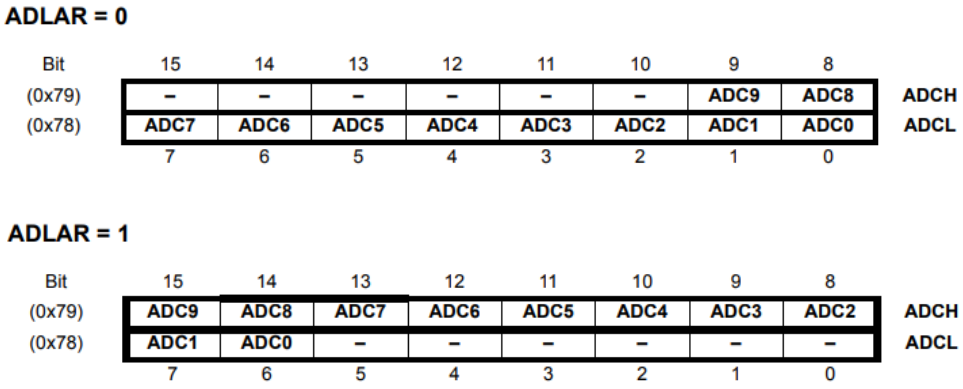


Fig. 5.3. Justificación del resultado de acuerdo al bit ADLAR en el conversor.

En este proyecto se utiliza el resultado para regular el brillo entre 0 y 255, por ende, solo se requieren 8 bits. Mediante ADLAR = 1, un LSB representa 5V / 256 = 19,53 mV.

5.3.4 – Comando SEL

Para la selección de un LED especifico o del color actual, se implementó el comando SEL, que acepta 4 opciones posibles: R, G, B y *. En los 3 primeros, al presionar Enter se guarda la selección en el módulo Controlador, y se apagan los otros dos LEDs. En la última opción, en cambio, se realiza la selección del color previamente asignado vía comando RGB.

Estas opciones se encuentran representadas en un enumerativo, con prefijo “OP_”, y la opción seleccionada se guarda en una variable “selectedLed” del controlador. Por defecto, la opción “*” (OP_ALL) se encuentra seleccionada desde el inicio del programa.

5.3.5 – Comando ADC

Se implementó un comando simple con la palabra reservada “ADC”, que acepta 2 opciones posibles: “ON” y “OFF”, que habilita y deshabilita el conversor, respectivamente.

La habilitación del conversor se realiza invocando a la función *ADC_Enable*, seguido del llamado a la función *ADC_Run*, que desencadena el comienzo de la primera conversión. Al completarse una conversión, se activa la interrupción que activa el Flag ADC.

```
Si FLAG_ADC está activo
        Leer resultado de la conversión
        Actualizar brillo mediante Controlador_ADC_Update
        Iniciar nueva conversión
        Reiniciar FLAG_ADC
Si FLAG_DATOS está activo
        Retirar texto ingresado de Buffer RX
        Enviar texto al intérprete de comandos
        Desactivar FLAG_DATOS
Suspender uso de CPU hasta próxima interrupción
```

Pseudocódigo 5.1. Algoritmo final del Loop principal de Background.

La función *Controlador_ADC_Update* invoca a la función *LED_UpdateCustom*, que actualiza el brillo únicamente de un LED indicado por el controlador (selectedLed). A su vez, en *LED_UpdateCustom* invoca a otras funciones, como se muestra a continuación.

```
Si la opción es…
    LED rojo…
        Actualizar brillo del LED rojo, sin guardar
    LED verde…
        Actualizar brillo del LED verde, sin guardar
    LED azul…
        Actualizar brillo del LED azul, sin guardar
    Color actual…
        Actualizar brillo de LEDs según lectura y guardados
```

Pseudocódigo 5.2. Función *LED_UpdateCustom* del módulo "LED".

"Sin guardar" refiere a que las funciones *LED_UpdateRed*, *LED_UpdateGreen* o *LED_UpdateBlue*, vistas en la sección 4.3.6, se invocan con parámetro "save" asignado en 0.

La función implementada para la opción de color actual es *LED_UpdateBright*, que tampoco sobrescribe los valores guardados, e invoca a las 3 funciones de actualizar color con un valor de brillo entre 0 y el valor guardado de cada una, calculado de la siguiente forma:

$$Valor\ de\ brillo = Valor\ Guardado * \frac{Lectura\ de\ ADC}{255}$$

Por otra parte, la desactivación del conversor analógico-digital (ADC OFF) se realiza mediante la invocación a *ADC_Disable*, que detiene también una posible conversión en curso.

5.4 – Simulación

Se adjunta un enlace a los videos de las validaciones realizadas con el kit de desarrollo, como así también de la simulación en Proteus: https://bit.ly/cdym-tp4-videos

Se complementa con algunas capturas realizadas durante la depuración del programa, mostrando que el tiempo de conversión cambia en la primera respecto a las siguientes.

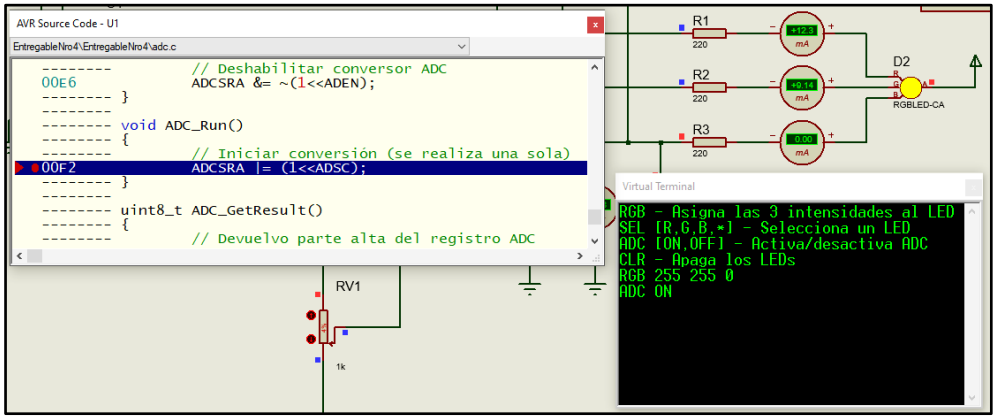


Fig. 5.4. Inicio de la primera conversión, con un color asignado previamente.

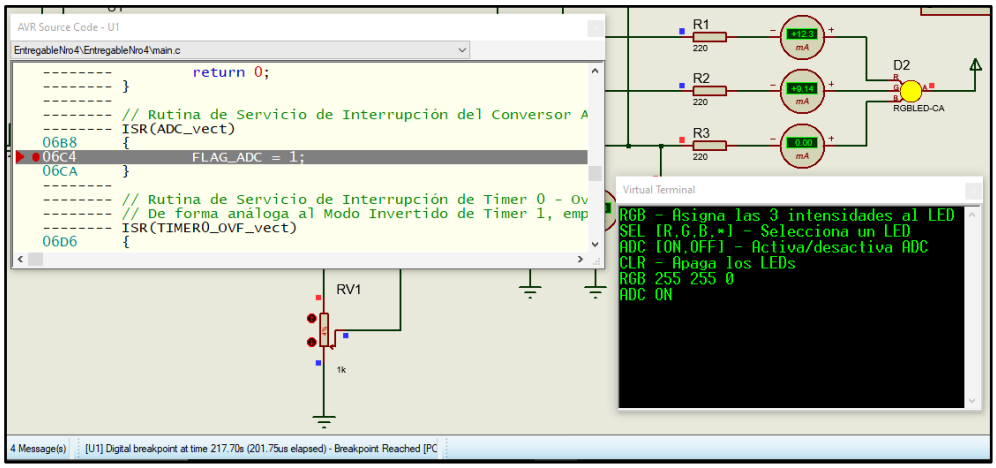


Fig. 5.5. Medición del tiempo de la primera conversión: 201,75 μs = 25 ciclos.

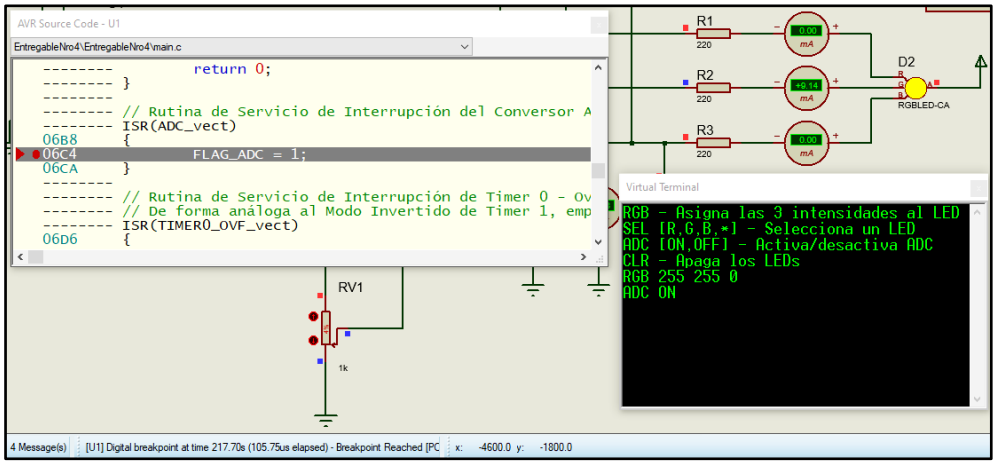


Fig. 5.6. Medición del tiempo de la segunda conversión: 105,75 μs = 13 ciclos.

Esto se debe a que en la primera conversión se inicializa el circuito analógico y se puede esperar a que la tensión se estabilice para evitar una lectura errónea. 1 ciclo = 1 / 125 kHz.

6 – Conclusiones

Se debieron implementar soluciones a dos problemáticas no contempladas previamente en los enunciados. En primer lugar, la generación de la señal PWM para el LED rojo, ya que no se encuentra conectado a la salida de un Waveform Generator que se encargue de subir y bajar el nivel del pin sin intervención de CPU. En segundo lugar, la imposibilidad de generar un brillo nulo al haber elegido el modo invertido de Fast PWM en Timer 1.

6.1 – Ventajas de la solución

Se incorporaron bibliotecas ya creadas para un manejo eficiente del periférico USART mediante los búferes realizados en el TP3, para la recepción y envío de mensajes.

Se implementaron diversos comandos para que el usuario pueda realizar las acciones en el orden que desee. El comando ADC permite que el usuario pueda visualizar el color elegido con el comando RGB en lugar de que el conversor funcione desde el comienzo. Por otra parte, los comandos SEL y CLR son características adicionales que no estaban solicitadas para el trabajo pero que fueron implementadas para ofrecer más opciones y "atajos" al usuario.

Además de los chequeos de comandos válidos, se agregaron funciones tipo Parser, lo que posibilitó el uso de múltiples parámetros y la detección de tipos correctos.

6.2 – Desventajas de la solución

La solución al primer problema mediante interrupciones de Timer 0 involucra un costo de CPU por cada vez que se requiera modificar el nivel de tensión del pin PB5. Una mejor solución sería que el LED rojo esté conectado a PD5 o PD6 (OC0x), sin embargo, se restringe a seguir el conexionado ya presente en el kit. Respecto al brillo nulo, para evitar el problema se podría optar por modo no invertido, pero no sería posible establecer el brillo máximo.

Como desventaja a la adición de comandos y parámetros, el tiempo de procesamiento resulta mayor debido a que se deben realizar más comparaciones de Strings.

7 – Bibliografía

- Atmel Corporation (2015). *ATmega 328P Datasheet*. Capítulos 14, 15, 19, 23 y 29. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

Apéndice A – Archivos de cabecera

A continuación, se muestran los archivos de cabecera del proyecto realizado.

adc.h

```
/*
 * adc.h
 *
 * Created: 10/7/2022 16:19:55
 *  Author: Calderón Sergio
 */


#ifndef ADC_H_
	#define ADC_H_

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

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

	// ----------------- Prototipos de funciones públicas ---------

	// Establece y selecciona el terminal ADC3 como entrada analógica, configura
	// el ADC para su funcionamiento a 125 kHz con justificación a la izquierda
	// y habilita la interrupción de conversión finalizada
	void ADC_Init();

	// Habilita el uso del conversor
	void ADC_Enable();

	// Deshabilita el uso del conversor
	void ADC_Disable();

	// Inicia una nueva conversión
	void ADC_Run();

	// Retorna el resultado de la conversión en 8 bits (parte alta de ADC)
	uint8_t ADC_GetResult();


#endif /* ADC_H_ */
```

buffer.h

```
/*
 * Created: 24/6/2022 22:40:20
 * Author: Calderón Sergio
 */

#ifndef BUFFER_H_
	#define BUFFER_H_

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

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


	// ------------------- Constantes ----------------------------

	// Tamaño del buffer RX en bytes
	#define RX_BUFFER_SIZE 32

	// Tamaño del buffer TX en bytes
	#define TX_BUFFER_SIZE 196


	// -------- Prototipos de funciones públicas ---------------

	// --------- Operaciones sobre Buffer RX -------------------

	// Agrega un dato al Buffer RX
	void Buffer_RX_Push(uint8_t);

	// Extrae los datos del Buffer RX en el texto de longitud maxLength
	void Buffer_RX_PopString(char* texto, uint8_t maxLength);

	// Retira un dato del Buffer RX
	uint8_t Buffer_RX_Pop();

	// --------- Operaciones sobre Buffer TX -----------------

	// Agrega un dato al Buffer TX
	void Buffer_TX_Push(uint8_t);

	// Agrega una cadena de texto al Buffer TX
	void Buffer_TX_PushString(char*);

	// Agrega una cadena de texto + un salto de línea al Buffer TX
	void Buffer_TX_PushLine(char*);

	// Retira un dato del Buffer TX
	uint8_t Buffer_TX_Pop();

	// Retorna 1 si no hay datos sin retirar en Buffer TX, 0 en caso contrario
	uint8_t Buffer_TX_Empty();

#endif /* BUFFER_H_ */
```

controlador.h

```
/*
 * controlador.h
 *
 * Created: 30/7/2022 04:31:12
 *  Author: Calderón Sergio
 */


#ifndef CONTROLADOR_H_
	#define CONTROLADOR_H_

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

	// Archivo de cabecera del Módulo LED
	#include "led.h"

	// Archivo de cabecera del Conversor Analógico-Digital
	#include "adc.h"

	// Archivo de cabecera para manejo de mensajes
	#include "CLI.h"

	// ----------------- Prototipos de funciones públicas ----------

	// Selecciona un LED para su posterior regulación con ADC
	void Controlador_Select(LED_Option option);

	// Actualiza y guarda los valores RGB de los LEDs
	void Controlador_Set_RGB(uint8_t r, uint8_t g, uint8_t b);

	// Apaga todos los LEDs y borra la selección actual
	void Controlador_Clear();

	// Actualiza el brillo del LED o color seleccionado, sin guardarlo
	void Controlador_ADC_Update(uint8_t bright);

	// Enciende el conversor analógico-digital
	void Controlador_ADC_ON();

	// Apaga el conversor analógico-digital
	void Controlador_ADC_OFF();

#endif /* CONTROLADOR_H_ */
```

CLI.h

```
/*
 * CLI.h
 *
 * Created: 7/7/2022 03:13:15
 *  Author: Calderón Sergio
 */


#ifndef CLI_H_
	#define CLI_H_

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

	// Tipos de datos enteros estandarizados
	#include <stdint.h>

	// Biblioteca Parser de utilidades
	#include "parser.h"

	// Biblioteca Buffer
	#include "buffer.h"

	// Biblioteca del USART
	#include "serialPort.h"

	// Archivo de cabecera del Controlador
	#include "controlador.h"

	// ------------------- Prototipos de funciones públicas ---------

	// Muestra el menú de comandos disponibles
	void CLI_MostrarMenu();

	// Interpreta el texto pasado por parámetro como un comando y lo ejecuta
	void CLI_LeerComando(char* texto);

	// Agrega el texto pasado por parámetro al Buffer TX y activa la transmisión
	void CLI_EscribirMensaje(char*);

#endif /* CLI_H_ */
```

led.h

```
/*
 * led.h
 *
 * Created: 4/7/2022 14:34:38
 * Author: Calderón Sergio
 */

#ifndef LED_H_
	#define LED_H_

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

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

```
	// Archivo de cabecera de Timer 0 y 1
	#include "reloj.h"

	// ------------------------ Tipos de Datos ------------------------

	// LEDs disponibles para la modificación de su brillo
	typedef enum {OP_RED, OP_GREEN, OP_BLUE, OP_ALL} LED_Option;

	// -------------------------- Constantes ----------------------------

	// Valor a escribir en la salida
	// Si es ánodo común (VCC), se escribe un 0 para encender
	// Si es cátodo común (GND), se escribe un 1 para encender
	#define ANODO_COMUN 1

	#if ANODO_COMUN
		#define NIVEL_ON 0
		#define NIVEL_OFF 1
	#else
		#define NIVEL_ON 1
		#define NIVEL_OFF 0
	#endif

	// Pin conectado a cada LED
	#define PIN_RED PORTB5
	#define PIN_GREEN PORTB2
	#define PIN_BLUE PORTB1

	// ------------ Prototipos de funciones públicas -----------------

	// Configura los pines conectados al LED como salida
	void LED_Init();

	// Mantiene el brillo únicamente del LED indicado
	void LED_KeepOnly(LED_Option);

	// Actualiza los valores de intensidad para cada color
	void LED_Update(uint8_t red, uint8_t green, uint8_t blue);

	// Actualiza el brillo del LED especificado, sin guardarlo
	void LED_UpdateCustom(LED_Option option, uint8_t valor);

	// Actualiza el valor de intensidad del LED rojo
	void LED_UpdateRed(uint8_t bright, uint8_t save);

	// Actualiza el valor de intensidad del LED verde
	void LED_UpdateGreen(uint8_t bright, uint8_t save);

	// Actualiza el valor de intensidad del LED azul
	void LED_UpdateBlue(uint8_t bright, uint8_t save);

	// Modifica el brillo del color actual
	void LED_UpdateBright(uint8_t);

	// Apaga el LED ubicado en el PIN de Port B especificado
	void LED_ON(uint8_t);

	// Enciende el LED ubicado en el PIN de Port B especificado
	void LED_OFF(uint8_t);

#endif /* LED_H_ */
```

main.h

```
/*
 * main.h
 *
 * Created: 4/7/2022 15:03:28
 * Author: Calderón Sergio
 */


#ifndef MAIN_H_
	#define MAIN_H_

	// ----------------------- Constantes ---------------

	// Frecuencia del Microcontrolador (16 MHz)
	#define F_CPU 16000000UL

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

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

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

	// Modos de ahorro de energía
	#include <avr/sleep.h>

	// Archivos del Proyecto
	// Se incluyen módulos a inicializar, utilizados en ISR y en Loop principal
	#include "adc.h"
	#include "buffer.h"
	#include "controlador.h"
	#include "CLI.h"
	#include "led.h"
	#include "reloj.h"
	#include "serialPort.h"

#endif /* MAIN_H_ */
```

parser.h

```
/*
 * parser.h
 *
 * Created: 30/7/2022 03:19:31
 *  Author: Calderón Sergio
 */


#ifndef PARSER_H_
	#define PARSER_H_

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

	// Utilidades con Strings
	#include <string.h>
```

```
	// Tipos de datos enteros estandarizados
	#include <stdint.h>

	// ------------------------- Constantes ------------------------

	#define OK 1
	#define ERROR_CAST 2
	#define ERROR_PARAM 3

	// -------------------- Prototipos de funciones públicas --------------

	// Compara dos textos y retorna 1 si coinciden, sensible a mayúsculas y
minúsculas
	uint8_t PARSER_Equals(const char* s1, const char* s2);

	// Separa el texto pasado por parámetro de acuerdo al primer blanco
	void PARSER_Split(const char* texto, char* s1, char* s2);

	// Extrae 3 números enteros y positivos de un texto pasado por parámetro
	// Si contiene menos de 3 palabras retorna un error de parámetros
insuficientes.
	// Si el contenido no son números enteros y positivos retorna un error de cast
	uint8_t PARSER_Convert(char* texto, uint8_t* n1, uint8_t* n2, uint8_t* n3);

#endif /* PARSER_H_ */
```

reloj.h

```
/*
 * reloj.h
 *
 * Created: 4/7/2022 14:50:40
 *  Author: Calderón Sergio
 */

#ifndef RELOJ_H_
	#define RELOJ_H_

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

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

	// -------------------- Prototipos de funciones públicas ------------

	// Inicializa el Timer 0
	// En este proyecto, es el encargado de generar interrupciones
	void RELOJ_T0_Init();

	// Inicializa el Timer 1
	// En este proyecto, es el encargado de generar las señales PWM
	void RELOJ_T1_Init();

	// Asigna el prescaler adecuado para dar inicio a los relojes
	void RELOJ_Start_Both();

	// Timer 0 - Interrupción de Overflow
	void RELOJ_T0_OVF_Interrupt_Enable();
	void RELOJ_T0_OVF_Interrupt_Disable();
```

```
	// Timer 0 - Interrupción de Compare Match A
	void RELOJ_T0_COMPA_Interrupt_Enable();
	void RELOJ_T0_COMPA_Interrupt_Disable();

	// Timer 0 - Actualización de brillo
	void RELOJ_T0_UpdateOCR(uint8_t);

	// Timer 1 - Señal PWM de OC1A
	void RELOJ_T1_PWM_A_Enable();
	void RELOJ_T1_PWM_A_Disable();
	void RELOJ_T1_PWM_A_UpdateOCR(uint8_t);

	// Timer 1 - Señal PWM de OC1B
	void RELOJ_T1_PWM_B_Enable();
	void RELOJ_T1_PWM_B_Disable();
	void RELOJ_T1_PWM_B_UpdateOCR(uint8_t);

#endif /* RELOJ_H_ */
```

serialPort.h

```
/*
 * Created: 07/10/2020 03:02:42
 * Author: vfperri, Calderón Sergio
 */

#ifndef SERIALPORT_H_
	#define SERIALPORT_H_

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

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

	// -------- Prototipos de funciones públicas ---------------

	// Inicializa el puerto serie para la frecuencia de CPU especificada
	void SerialPort_Init(uint32_t);

	// Inicializa y habilita el receptor
	void SerialPort_RX_Enable(void);

	// Inicializa y habilita el transmisor
	void SerialPort_TX_Enable(void);

	// Habilitación y deshabilitación de interrupciones
	void SerialPort_RX_Interrupt_Enable(void);
	void SerialPort_TX_Interrupt_Enable(void);
	void SerialPort_TX_Interrupt_Disable(void);

	// Transmite el dato pasado por parámetro
	void SerialPort_Send_Data(uint8_t);

	// Recibe y retorna el dato actual
	uint8_t SerialPort_Receive_Data(void);

#endif /* SERIALPORT_H_ */
```

Apéndice B – Archivos .C

A continuación, se muestran los archivos .C que conforman el programa completo, los cuales se compilan y utilizan para la simulación y la validación en el kit de desarrollo.

adc.c

```
/*
 * adc.c
 * Author: Calderón Sergio
 */

#include "adc.h"

// ---------------- Implementación de funciones públicas ---------------

void ADC_Init()
{
	// Configurar pin de ADC3 como entrada analógica (PC3)
	DDRC &= ~(1<<PORTC3);
	DIDR0 |= (1<<PORTC3);

	// Establecer VCC como referencia (REFS1:0 = 1)
	ADMUX = (1<<REFS0);

	// Seleccionar ADC3 como entrada analógica (MUX3:0 = 3)
	ADMUX |= (1<<MUX1)|(1<<MUX0);

	// Ubicar los 10 bits desde la izquierda (ADLAR = 1)
	ADMUX |= (1<<ADLAR);

	// Establecer prescaler de 128, mediante ADPS2:0 = 7
	ADCSRA = (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);

	// Habilitar interrupción de conversión finalizada
	ADCSRA |= (1<<ADIE);
}

void ADC_Enable()
{
	ADCSRA |= (1<<ADEN);
}

void ADC_Disable()
{
	ADCSRA &= ~(1<<ADEN);
}

void ADC_Run()
{
	ADCSRA |= (1<<ADSC);
}

uint8_t ADC_GetResult()
{
	return ADCH;
}
```

buffer.c

```
/*
 * buffer.c
 * Author: Calderon Sergio
 */

#include "buffer.h"

// Búfferes de tamaño asignado según su función
static uint8_t RX_Data[RX_BUFFER_SIZE];
static uint8_t TX_Data[TX_BUFFER_SIZE];

// Estructura tipo Buffer
typedef struct {
	uint8_t* data;
	uint8_t readIndex;
	uint8_t writeIndex;
} TipoBuffer;

static TipoBuffer RX_Buffer = {RX_Data, 0, 0};
static TipoBuffer TX_Buffer = {TX_Data, 0, 0};

// Funciones públicas

void Buffer_RX_Push(uint8_t dato)
{
	// Asignar dato en la posición de escritura
	RX_Buffer.data[RX_Buffer.writeIndex] = dato;

	// Incrementar indice de escritura
	RX_Buffer.writeIndex = (RX_Buffer.writeIndex + 1) % RX_BUFFER_SIZE;
}

uint8_t Buffer_RX_Pop()
{
	// Recuperar dato en la posición de lectura
	uint8_t dato = RX_Buffer.data[RX_Buffer.readIndex];

	// Incrementar indice de lectura
	RX_Buffer.readIndex = (RX_Buffer.readIndex + 1) % RX_BUFFER_SIZE;

	// Devolver dato
	return dato;
}

void Buffer_RX_PopString(char* texto, uint8_t maxLength)
{
	// Inicializar indice de String en cero
	uint8_t dato, i = 0;

	// Hasta alcanzar un fin de cadena o la longitud máxima...
	do
	{
		// Retirar dato del Buffer RX
		dato = Buffer_RX_Pop();

		// Colocar dato en la posición "i" del String
		texto[i] = dato;

		// Incrementar indice
		i++;
```

```
	} while (dato != '\0' && i < maxLength);

	// Adelantar indice de lectura hasta el de escritura
	// Se ignoran el resto de datos que no se retiraron
	RX_Buffer.readIndex = RX_Buffer.writeIndex;
}

void Buffer_TX_Push(uint8_t dato)
{
	// Asignar dato en la posición de escritura
	TX_Buffer.data[TX_Buffer.writeIndex] = dato;

	// Incrementar indice de escritura
	TX_Buffer.writeIndex = (TX_Buffer.writeIndex + 1) % TX_BUFFER_SIZE;
}

void Buffer_TX_PushString(char* texto)
{
	// Por cada caracter del String
	for(uint8_t i=0; texto[i] != '\0'; i++)
	{
		// Agregar caracter al Buffer TX
		Buffer_TX_Push(texto[i]);
	}
}

void Buffer_TX_PushLine(char* texto)
{
	// Por cada caracter del String
	for(uint8_t i=0; texto[i] != '\0'; i++)
	{
		// Agregar caracter al Buffer TX
		Buffer_TX_Push(texto[i]);
	}

	// Agregar salto de linea
	Buffer_TX_Push('\r');
	Buffer_TX_Push('\n');
}

uint8_t Buffer_TX_Pop()
{
	// Obtener dato en la posición de lectura
	uint8_t dato = TX_Buffer.data[TX_Buffer.readIndex];

	// Incrementar indice de lectura
	TX_Buffer.readIndex = (TX_Buffer.readIndex + 1) % TX_BUFFER_SIZE;

	// Devolver dato
	return dato;
}

uint8_t Buffer_TX_Empty()
{
	// Devolver si los indices de lectura y escritura coinciden
	return TX_Buffer.readIndex == TX_Buffer.writeIndex;
}
```

controlador.c

```
/*
 * controlador.c
 *
 * Created: 30/7/2022 04:31:26
 *  Author: Calderón Sergio
 */

// Archivo de cabecera
#include "controlador.h"

// Variables privadas
static LED_Option selectedLed = OP_ALL;
static uint8_t adcEnabled = 0;

// ---------------------- Implementación de funciones públicas ----------

void Controlador_Select(LED_Option option)
{
	// Actualizar LED seleccionado
	selectedLed = option;

	// Apagar los otros LEDs
	LED_KeepOnly(option);
}

void Controlador_Set_RGB(uint8_t r, uint8_t g, uint8_t b)
{
	if (adcEnabled) CLI_EscribirMensaje("--> Advertencia: ADC está activo");
	LED_Update(r,g,b);
}

void Controlador_Clear()
{
	// Asignar un brillo de 0 a todos los LEDs
	LED_Update(0,0,0);

	// Borrar el LED seleccionado
	selectedLed = OP_ALL;
}

// ------------------------------ Manejo de ADC -----------------------

void Controlador_ADC_Update(uint8_t bright)
{
	LED_UpdateCustom(selectedLed, bright);
}

void Controlador_ADC_ON()
{
	// Habilitar ADC e iniciar nueva conversión
	ADC_Enable();
	ADC_Run();
	adcEnabled = 1;
}

void Controlador_ADC_OFF()
{
	ADC_Disable();
	adcEnabled = 0;
}
```

CLI.c

```
/*
 * CLI.c
 *
 * Created: 7/7/2022 03:09:19
 *  Author: Calderón Sergio
 */

// Archivo de cabecera
#include "CLI.h"


// ------------------ Funciones públicas --------------

void CLI_LeerComando(char* texto)
{
	char comando[10];
	char param[10];

	// Obtener comando y parámetro escrito a continuación
	PARSER_Split(texto, comando, param);

	// Comprobar si el comando existe y actuar de acuerdo al parámetro
	if (PARSER_Equals(comando, "SEL"))
	{
		if (PARSER_Equals(param, "R")) Controlador_Select(OP_RED);
		else if (PARSER_Equals(param, "G")) Controlador_Select(OP_GREEN);
		else if (PARSER_Equals(param, "B")) Controlador_Select(OP_BLUE);
		else if (PARSER_Equals(param, "*")) Controlador_Select(OP_ALL);
		else CLI_EscribirMensaje("--> Opciones validas: R, G, B, *");
	} else if (PARSER_Equals(comando, "RGB")) {
		uint8_t r,g,b,status;
		status = PARSER_Convert(param, &r, &g, &b);
		if (status == OK) Controlador_Set_RGB(r,g,b);
		else if (status == ERROR_PARAM)
			CLI_EscribirMensaje("--> Se esperaban 3 parametros");
		else if (status == ERROR_CAST)
			CLI_EscribirMensaje("--> Los parametros deben ser numeros");
	} else if (PARSER_Equals(comando, "ADC")) {
		if (PARSER_Equals(param, "ON")) Controlador_ADC_ON();
		else if (PARSER_Equals(param, "OFF")) Controlador_ADC_OFF();
		else CLI_EscribirMensaje("--> Opciones validas: ON, OFF");
	} else if (PARSER_Equals(comando, "CLR")) Controlador_Clear();
	else CLI_EscribirMensaje("--> Comandos validos: RGB, SEL, ADC, CLR");
}

void CLI_MostrarMenu()
{
	Buffer_TX_PushLine("RGB - Asigna las 3 intensidades al LED asociado");
	Buffer_TX_PushLine("SEL [R,G,B,*] - Selecciona un LED");
	Buffer_TX_PushLine("ADC [ON,OFF] - Activa/desactiva ADC");
	Buffer_TX_PushLine("CLR - Apaga los LEDs");
	SerialPort_TX_Interrupt_Enable();
}

void CLI_EscribirMensaje(char* texto)
{
	// Agregar línea al Buffer TX y habilitar interrupciones de transmisión
	Buffer_TX_PushLine(texto);
	SerialPort_TX_Interrupt_Enable();
}
```

led.c

```
/*
 * led.c
 *
 * Created: 4/7/2022 14:34:24
 *  Author: Calderón Sergio
 */

// Archivo de cabecera
#include "led.h"

// Declaración de tipos: estructura de valores guardados
typedef struct {
	uint8_t red;
	uint8_t green;
	uint8_t blue;
} LED_UserValues;

// Prototipos de funciones privadas
static void setAlto(uint8_t);
static void setBajo(uint8_t);

// Variables privadas
static LED_UserValues userValues = {0,0,0};

// ----------------- Implementación de funciones públicas ------------------

void LED_Init()
{
	// Configurar cada pin conectado a un LED como salida
	DDRB |= (1<<PIN_RED)|(1<<PIN_GREEN)|(1<<PIN_BLUE);

	// Apagar todos los LEDs por defecto
	LED_OFF(PIN_RED);
	LED_OFF(PIN_GREEN);
	LED_OFF(PIN_BLUE);
}

void LED_KeepOnly(LED_Option option)
{
	// Apagar los LEDs que no correspondan a la opción
	switch(option)
	{
		case OP_RED:
			LED_UpdateGreen(0,1);
			LED_UpdateBlue(0,1);
			break;
		case OP_GREEN:
			LED_UpdateRed(0,1);
			LED_UpdateBlue(0,1);
			break;
		case OP_BLUE:
			LED_UpdateRed(0,1);
			LED_UpdateGreen(0,1);
			break;
		case OP_ALL:
			break;
	}
}
```

```
void LED_Update(uint8_t red, uint8_t green, uint8_t blue)
{
	// Actualizar el brillo de rojo, verde y azul
	// Se indica que los mismos deben guardarse
	LED_UpdateRed(red, 1);
	LED_UpdateGreen(green, 1);
	LED_UpdateBlue(blue, 1);
}

void LED_UpdateCustom(LED_Option option, uint8_t valor)
{
	switch(option)
	{
		case OP_RED:
			LED_UpdateRed(valor, 0);
			break;
		case OP_GREEN:
			LED_UpdateGreen(valor, 0);
			break;
		case OP_BLUE:
			LED_UpdateBlue(valor, 0);
			break;
		case OP_ALL:
			LED_UpdateBright(valor);
			break;
		default:
			break;
	}
}

void LED_UpdateRed(uint8_t bright, uint8_t save)
{
	// El LED rojo se controla mediante interrupciones de T0
	// La ISR de TIMER0_OVF enciende el LED, y de TIMER0_COMPA lo apaga

	// Si la intensidad es 0, se debe mantener apagado el LED
	if (bright == 0)
	{
		// Desactivar interrupción de Overflow
		RELOJ_T0_OVF_Interrupt_Disable();

		// Forzar apagado del LED
		LED_OFF(PIN_RED);
	}
	else if (bright == 255)
	{
		// Sino, si la intensidad es máxima, se debe mantener encendido
		// Desactivar interrupción de Compare A
		RELOJ_T0_COMPA_Interrupt_Disable();

		// Forzar encendido del LED
		LED_ON(PIN_RED);
	} else {
		// Sino...
		// Activar interrupciones de T0
		RELOJ_T0_OVF_Interrupt_Enable();
		RELOJ_T0_COMPA_Interrupt_Enable();

		// Actualizar valor de brillo
		RELOJ_T0_UpdateOCR(bright);
	}
```

```
	// Guardar intensidad personalizada
	if (save) userValues.red = bright;
}

void LED_UpdateGreen(uint8_t bright, uint8_t save)
{
	// El LED verde se controla mediante señal PWM de OC1B
	// Se trabaja en Fast PWM, modo invertido.

	// Si la intensidad es 0, se debe mantener apagado el LED
	if (bright == 0)
	{
		// Desconectar señal OC1B
		RELOJ_T1_PWM_B_Disable();

		// Forzar apagado del LED
		LED_OFF(PIN_GREEN);
	} else {
		// Sino...
		// Habilitar señal OC1B
		RELOJ_T1_PWM_B_Enable();

		// Actualizar valor de brillo
		RELOJ_T1_PWM_B_UpdateOCR(bright);
	}

	// Guardar intensidad personalizada
	if (save) userValues.green = bright;
}

void LED_UpdateBlue(uint8_t bright, uint8_t save)
{
	// El LED azul se controla mediante señal PWM de OC1A
	// Se trabaja en Fast PWM, modo invertido.

	// Si la intensidad es 0, se debe mantener apagado el LED
	if (bright == 0)
	{
		// Desconectar señal OC1A
		RELOJ_T1_PWM_A_Disable();

		// Forzar apagado del LED
		LED_OFF(PIN_BLUE);
	} else {
		// Sino...
		// Habilitar señal OC1A
		RELOJ_T1_PWM_A_Enable();

		// Actualizar valor de brillo
		RELOJ_T1_PWM_A_UpdateOCR(bright);
	}

	// Guardar intensidad personalizada
	if (save) userValues.blue = bright;
}

void LED_UpdateBright(uint8_t intensity)
{
	// Calcular valores de brillo a asignar
	uint8_t red = userValues.red * intensity / 255;
	uint8_t green = userValues.green * intensity / 255;
	uint8_t blue = userValues.blue * intensity / 255;
```

```
	// Asignar nuevo brillo de rojo, sin guardar
	LED_UpdateRed(red, 0);

	// Asignar nuevo brillo de verde, sin guardar
	LED_UpdateGreen(green, 0);

	// Asignar nuevo brillo de azul, sin guardar
	LED_UpdateBlue(blue, 0);
}

void LED_ON(uint8_t pin)
{
	// Poner un nivel alto o bajo según macro NIVEL_ON

	#if NIVEL_ON
		setAlto(pin);
	#else
		setBajo(pin);
	#endif
}

void LED_OFF(uint8_t pin)
{
	// Poner un nivel alto o bajo según macro NIVEL_OFF

	#if NIVEL_OFF
		setAlto(pin);
	#else
		setBajo(pin);
	#endif
}

static void setAlto(uint8_t pin)
{
	PORTB |= (1<<pin);
}

static void setBajo(uint8_t pin)
{
	PORTB &= ~(1<<pin);
}
```

main.c

```
/*
 * main.c
 *
 * Created: 4/7/2022 14:31:36
 * Author : Calderón Sergio
 */

// Archivo de cabecera
#include "main.h"


// Flags de tipo volátil
static volatile uint8_t FLAG_DATOS = 0;
static volatile uint8_t FLAG_ADC = 0;
```

```c
int main(void)
{
	/* Setup */

	// Inicialización de USART
	SerialPort_Init(F_CPU);
	SerialPort_TX_Enable();
	SerialPort_RX_Enable();
	SerialPort_RX_Interrupt_Enable();

	// Inicialización de Timer
	RELOJ_T1_Init();
	RELOJ_T0_Init();
	RELOJ_Start_Both();

	// Inicialización de LEDs
	LED_Init();

	// Inicialización de Conversor ADC
	ADC_Init();

	// Visualización de Menú
	CLI_MostrarMenu();

	// Habilitar interrupciones globales
	sei();


	/* Loop */
	while (1)
	{
		// Si se completó la conversión... (Flag ADC activo)
		if (FLAG_ADC)
		{
			// Leer valor
			uint8_t valor = ADC_GetResult();

			// Actualizar brillo
			Controlador_ADC_Update(valor);

			// Iniciar nueva conversión
			ADC_Run();

			// Desactivar flag
			FLAG_ADC = 0;
		}


		// Si se ingresaron datos... (Flag de Datos activo)
		if (FLAG_DATOS)
		{
			char texto[20];

			// Obtener texto ingresado (máx 20 caracteres)
			Buffer_RX_PopString(texto, 20);

			// Interpretar y ejecutar comando
			CLI_LeerComando(texto);

			// Desactivar flag
			FLAG_DATOS = 0;
		}
```

```
			// Suspender uso de CPU hasta próxima interrupción
			sleep_mode();
		}

		return 0;
}


// Rutina de Servicio de Interrupción del Conversor Analógico-Digital
ISR(ADC_vect)
{
		FLAG_ADC = 1;
}


// Rutina de Servicio de Interrupción de Timer 0 - Overflow
// De forma análoga al Modo Invertido de Timer 1, empieza en nivel bajo
ISR(TIMER0_OVF_vect)
{
		LED_ON(PIN_RED);
}

// Rutina de Servicio de Interrupción de Timer 0 - Compare Match A
// Luego de haberse generado un nivel bajo, se pasa a nivel alto
ISR(TIMER0_COMPA_vect)
{
		LED_OFF(PIN_RED);
}


// Rutina de Servicio de Interrupción de Byte Recibido
// Tarea en Foreground - Productor de Comandos
ISR(USART_RX_vect)
{
		// Leer dato de UDR (USART)
		volatile uint8_t aux = SerialPort_Receive_Data();

		// Si se presionó Enter
		if (aux == '\r')
		{
			// Agregar un fin de cadena al Buffer RX
			Buffer_RX_Push('\0');

			// Activar Flag "Hay Datos"
			FLAG_DATOS = 1;
		} else {
			// Sino, agregar dato al Buffer RX
			Buffer_RX_Push(aux);
		}
}

// Rutina de Servicio de Interrupción "Libre para Transmitir"
// Tarea en Foreground - Consumidor de Mensajes
ISR(USART_UDRE_vect)
{
		// Retirar dato del Buffer TX
		volatile uint8_t dato = Buffer_TX_Pop();

		// Escribir dato en UDR (USART)
		SerialPort_Send_Data(dato);
```

```
	// Si no hay más datos...
	if (Buffer_TX_Empty())
	{
		// Deshabilitar interrupciones de transmisión
		SerialPort_TX_Interrupt_Disable();
	}
}
```

parser.c

```
/*
 * Created: 30/7/2022 03:30:39
 * Author: Calderón Sergio
 */

// Archivo de cabecera
#include "parser.h"

// Prototipos de funciones privadas
static uint8_t split(char* origen, char* pal1, char* pal2, char* pal3);
static uint8_t getWord(char* origen, char* dest, uint8_t startIndex);
static uint8_t toNumber(char* palabra, uint8_t* numero);
static uint8_t toDigit(char);

// -------------------- Implementación de funciones públicas ----------------

uint8_t PARSER_Equals(const char* s1, const char* s2)
{
	return strcmp(s1, s2) == 0;
}

void PARSER_Split(const char* texto, char* s1, char* s2)
{
	uint8_t i=0, j=0, max=strlen(texto);

	// Extraer primera parte
	while (i < max && texto[i] != ' ')
	{
		s1[j] = texto[i];
		i++; j++;
	}

	// Agregar final de cadena a la primera parte
	s1[j] = '\0';

	// Ignorar espacio
	i++;

	// Reiniciar indice para la segunda parte
	j = 0;

	// Extraer segunda parte
	while (i < max)
	{
		s2[j] = texto[i];
		i++; j++;
	}

	// Agregar final de cadena a la segunda parte
	s2[j] = '\0';
}
```

```
uint8_t PARSER_Convert(char* texto, uint8_t* n1, uint8_t* n2, uint8_t* n3)
{
	char pal1[32], pal2[32], pal3[32];

	// Separar el texto en 3 palabras (delimitador ' ')
	uint8_t estado = split(texto, pal1, pal2, pal3);
	if (estado != OK) return estado;

	// Convertir primer palabra a numero
	estado = toNumber(pal1, n1);
	if (estado != OK) return estado;

	// Convertir segunda palabra a numero
	estado = toNumber(pal2, n2);
	if (estado != OK) return estado;

	// Convertir tercera palabra a numero
	estado = toNumber(pal3, n3);

	// Retornar estado (operación exitosa o error)
	return estado;
}


// ------------------- Implementación de funciones privadas ----------------

// Separa un texto de origen en 3 palabras (en reemplazo a strtok)
uint8_t split(char* origen, char* pal1, char* pal2, char* pal3)
{
	uint8_t i, length;

	length = strlen(origen);
	if (length == 0) return ERROR_PARAM;

	i = getWord(origen, pal1, 0);
	if (i >= length) return ERROR_PARAM;

	i = getWord(origen, pal2, i+1);
	if (i >= length) return ERROR_PARAM;

	getWord(origen, pal3, i+1);
	return OK;
}

// Obtiene una palabra a partir de un texto e índice de origen
uint8_t getWord(char* origen, char* dest, uint8_t startIndex)
{
	uint8_t j=0, i=startIndex;

	while (origen[i] != '\0' && origen[i] != ' ')
	{
		dest[j] = origen[i];
		i++;
		j++;
	}

	dest[j] = '\0';
	return i;
}
```

```
// Convierte un número en formato texto a entero positivo de 8 bits
static uint8_t toNumber(char* palabra, uint8_t* numero)
{
	uint8_t digito, factor = 1;
	*numero = 0;

	// Se comienza a armar el número desde la unidad
	for (int8_t i=strlen(palabra)-1; i>=0; i--)
	{
		// Obtener digito
		digito = toDigit(palabra[i]);

		// Si el caracter no es valido, retornar que no es número
		if (digito > 9) return ERROR_CAST;

		// Sumar al resultado
		(*numero) += digito * factor;

		// Actualizar factor (x10)
		factor *= 10;
	}

	// Retornar conversión exitosa
	return OK;
}

// Convierte un caracter a su digito correspondiente
static uint8_t toDigit(char c)
{
	return c - '0';
}
```

reloj.c

```
/*
 * Created: 4/7/2022 14:49:56
 *  Author: Calderón Sergio
 */

// Archivo de cabecera
#include "reloj.h"

// ---------------- Implementación de funciones públicas -------------

void RELOJ_T0_Init()
{
	// Deshabilitar todas las interrupciones de Timer 0
	TIMSK0 &= ~((1<<TOIE0)|(1<<OCIE0A)|(1<<OCIE0B));

	// Asignar Modo Normal, se cuenta hasta 0xFF (255)
	TCCR0A = 0;

	// Detener reloj interno (CS0[2:0] = 0)
	TCCR0B = 0;

	// Al inicio, el LED está apagado (brillo 0)
	OCR0A = 0;

	// Reiniciar contador a cero
	TCNT0 = 0;
}
```

```
void RELOJ_T1_Init()
{
	// Deshabilitar interrupciones de Timer 1
	TIMSK1 &= ~((1<<TOIE1)|(1<<OCIE1A)|(1<<OCIE1B));

	// Asignar Modo Fast PWM de 8 bits (WGM1[3:0] = 5)
	// Detener reloj interno (CS1[2:0] = 0)
	TCCR1B = (1<<WGM12);
	TCCR1A = (1<<WGM10);

	// Reiniciar valores de comparación
	OCR1A = 0;
	OCR1B = 0;

	// Reiniciar contador a cero
	TCNT1 = 0;
}

void RELOJ_Start_Both()
{
	// Asignar prescaler de N = 1024 (CS0[2:0] = 5)
	// La frecuencia generada es 61 Hz, mayor a 50 Hz
	TCCR0B |= (1<<CS02)|(1<<CS00);
	TCCR1B |= (1<<CS12)|(1<<CS10);
}


// ------------------------------- TIMER 1 --------------------------

void RELOJ_T1_PWM_A_Enable()
{
	// Establecer modo invertido para OC1A
	TCCR1A |= (1<<COM1A1)|(1<<COM1A0);
}

void RELOJ_T1_PWM_B_Enable()
{
	// Establecer modo invertido para OC1B
	TCCR1A |= (1<<COM1B1)|(1<<COM1B0);
}

void RELOJ_T1_PWM_A_Disable()
{
	// Desconectar salida OC1A
	TCCR1A &= ~((1<<COM1A1)|(1<<COM1A0));
}

void RELOJ_T1_PWM_B_Disable()
{
	// Desconectar salida OC1B
	TCCR1A &= ~((1<<COM1B1)|(1<<COM1B0));
}

// Modifica el valor del registro OCR1A
// En este proyecto se traduce como escala de azul
void RELOJ_T1_PWM_A_UpdateOCR(uint8_t valor)
{
	OCR1A = valor;
}
```

```
// Modifica el valor del registro OCR1B
// En este proyecto se traduce como escala de verde
void RELOJ_T1_PWM_B_UpdateOCR(uint8_t valor)
{
	OCR1B = valor;
}

// ------------------------------ TIMER 0 ---------------------------------

void RELOJ_T0_OVF_Interrupt_Enable()
{
	TIMSK0 |= (1<<TOIE0);
}

void RELOJ_T0_OVF_Interrupt_Disable()
{
	TIMSK0 &= ~(1<<TOIE0);
}

void RELOJ_T0_COMPA_Interrupt_Enable()
{
	TIMSK0 |= (1<<OCIE0A);
}

void RELOJ_T0_COMPA_Interrupt_Disable()
{
	TIMSK0 &= ~(1<<OCIE0A);
}

void RELOJ_T0_UpdateOCR(uint8_t valor)
{
	OCR0A = valor;
}
```

serialPort.c

```
/*
 * Created: 07/10/2020 03:02:18 p. m.
 *  Author: Calderón Sergio
 */

// Archivo de cabecera
#include "SerialPort.h"

// Inicialización de Puerto Serie
void SerialPort_Init(uint32_t f_cpu){

	// Deshabilitar receptor, transmisor e interrupciones
	UCSR0B = 0;

	// Definir tamaño de dato de 8 bits (8 bit data)
	UCSR0C = (1<<UCSZ01) | (1<<UCSZ00);

	// Para la frecuencia de CPU actual, configurar a 9600 bps, 1 stop
	if (f_cpu == 16000000UL) UBRR0L = 103;
	else if (f_cpu == 8000000UL) UBRR0L = 51;
	else if (f_cpu == 4000000UL) UBRR0L = 25;
}
```

```
// 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

void SerialPort_Send_Data(uint8_t data){
	UDR0 = data;
}

// Recepción

uint8_t SerialPort_Receive_Data(void){
	return UDR0;
}
```