Trabajo práctico Integrador

Carga y descarga de condensadores

Laboratorio de microprocesadores
Primer Cuatrimestre del 2021

Becker, Gonzalo Agustín	104291

Docentes:
Gerardo Stola
Fernando Cofman
Guido Salaya

Índice

1. Objetivos

El objetivo del presente informe es exponer múltiples herramientas propias de los microcontroladores. Para ello, se mide con el microcontrolador Atmega328P la carga y descarga sobre un capacitor, y se consigue luego graficar los valores medidos en una computadora.

2. Descripción del proyecto

El desarrollo del proyecto puede subdividirse en tres etapas: la construcción del programador mediante la utilización de un Arduino nano, la programación del microcontrolador usando dicho programador y la utlilización del microcontrolador programado en un circuito independiente al arduino.

En este caso el microcontrolador será usado para medir la carga y descarga sobre un capacitor. En primer instancia, se genera mediante un PWM del timer 2 una señal cuadrada de $50Hz$ con un duty cycle del 50 %, y se verifica que la frecuencia es correcta a partir del modo de captura del Timer 1, usando una interrupción en Int0. Una vez verificada la señal, esta es aplicada sobre un circuito RC, en donde se verifica que el producto $RC \simeq 6,67ms$ (se ha escogido $R = 660\Omega$ y $C = 10uF$). Luego, mediante el ADC (Analogue to digital converter) integrado al microcontrolador, se logra medir la tensión sobre el capacitor, y los valores obtenidos son enviados hacia la computadora a través de una comunicación serie lograda por un conversor USB a TTL. Finalmente, desde la computadora se realiza un gráfico temporal mostrando la evolución de esta señal.

En el siguiente link se encuentra un video mostrando su funcionamiento.

3. Programación del microcontrolador

En primer lugar, es necesario cargar en el Arduino nano el código que le permitirá funcionar como ISP (in-circuit serial programmer). Para ello, en el IDE de Arduino, es necesario abrir el archivo en *Archivo ->Ejemplos ->ArduinoISP*. Allí, en este caso, se definió el valor de baud rate a 115200, como se puede observar en la Figura 1. El baud rate es la velocidad de comunicación serial, en bits por segundo.

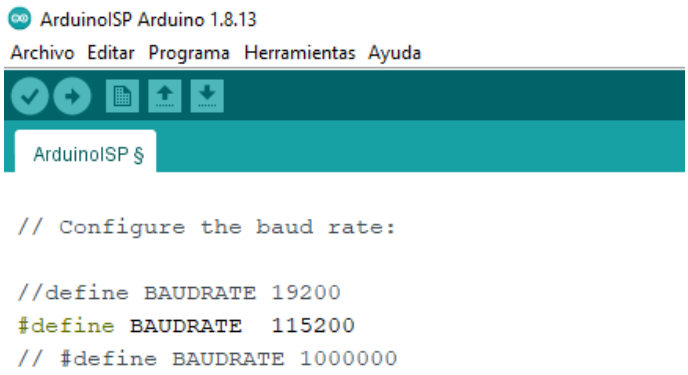

Figura 1: IDE de Arduino

Luego, es necesario construir un circuito externo al Arduino nano para lograr la correcta programación del Atmega328P. Este se puede observar en la Figura 2. El led D1 indica que el programador está listo para operar, D2 se enciende en caso de error y D3 indica que el proceso de programación está ocurriendo. Si bien no se puede ver, es necesario conectar el pin de 5v y de GND del arduino a los pines de VCC y GND del microcontrolador, respectivamente (pues el arduino debe estar en todo momento conectado por USB a una computadora)

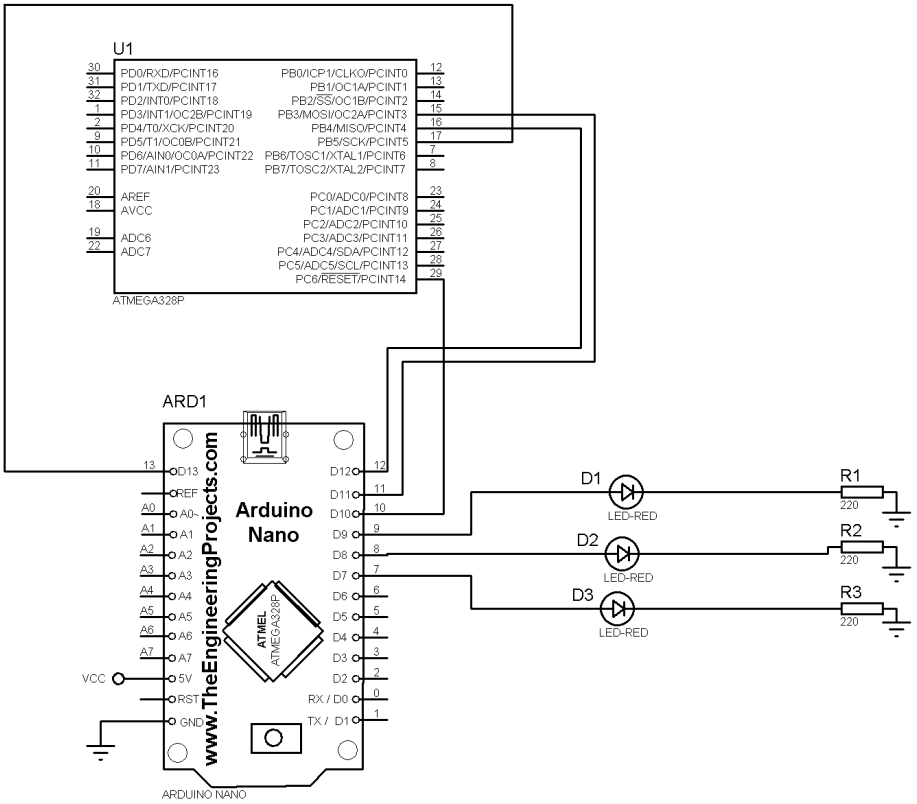

Figura 2: Esquemático del circuito Programador

Por último, en la figura 3, se puede ver la implementación de dicho programador. Es importante recordar los pines del microcontrolador Atmega328p, los cuales se encuentran indicados en la figura 4.

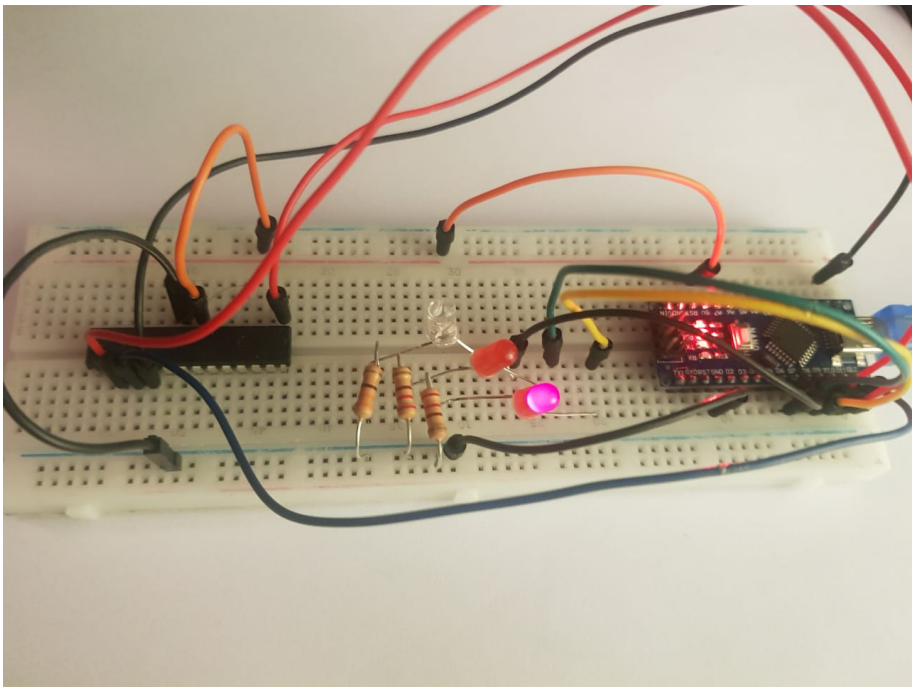

Figura 3: Circuito Programador

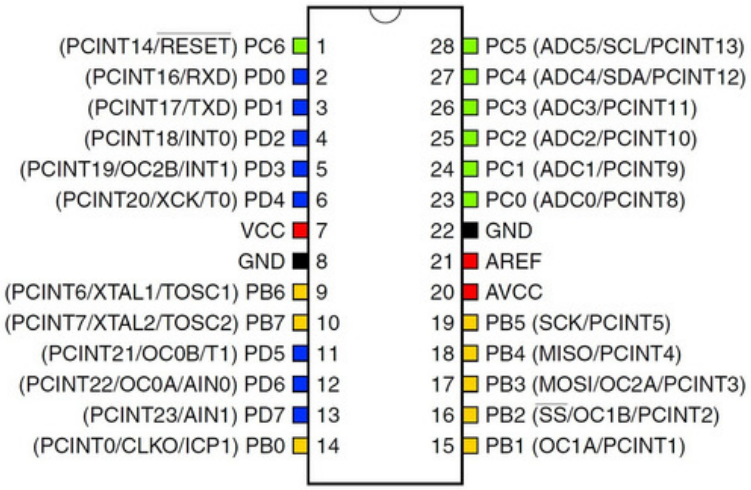

Figura 4: Pines del microcontrolador Atmega328

Para programar al microcontrolador usando al Arduino nano como ISP, es necesario disponer del archivo .hex generado después de la compilación. En este caso se utilizó la plataforma Microchip Studio, y se programó en c.

Una vez conocida la ubicación del archivo, es posible programar al microcontrolador haciendo uso del programa *avrdude*. En este caso, se ha instalado una interfaz que permite simplificar el proceso, llamada *Avrdudess*. Esta permite programar al microcontrolador a través de una ventana como la que se muestra en la Figura 5.

Como se puede ver, se ha definido el tipo de programador (arduino), el puerto de comunicación (COM3, depende de cada ordenador), el baud rate utilizado (115200), el MCU a programar (Atmega328P) y la ruta de acceso al archivo .hex que se programará en la memoria flash.

También se pueden programar los fuses (o fusibles), que permiten modificar una gran cantidad de características del microcontrolador, como se puede observar en la figura 6. En este trabajo se han cambiado sus valores con el fin de obtener una frecuencia de 8Mhz en lugar de la de 1MHz con la que estaba configurada incialmente el microcontrolador (anulando al bit CKDIV8). Sin embargo, esta es una acción que debe realizarse con suma precaución, ya que es posible bloquear al dispositivo de cometerse un error.

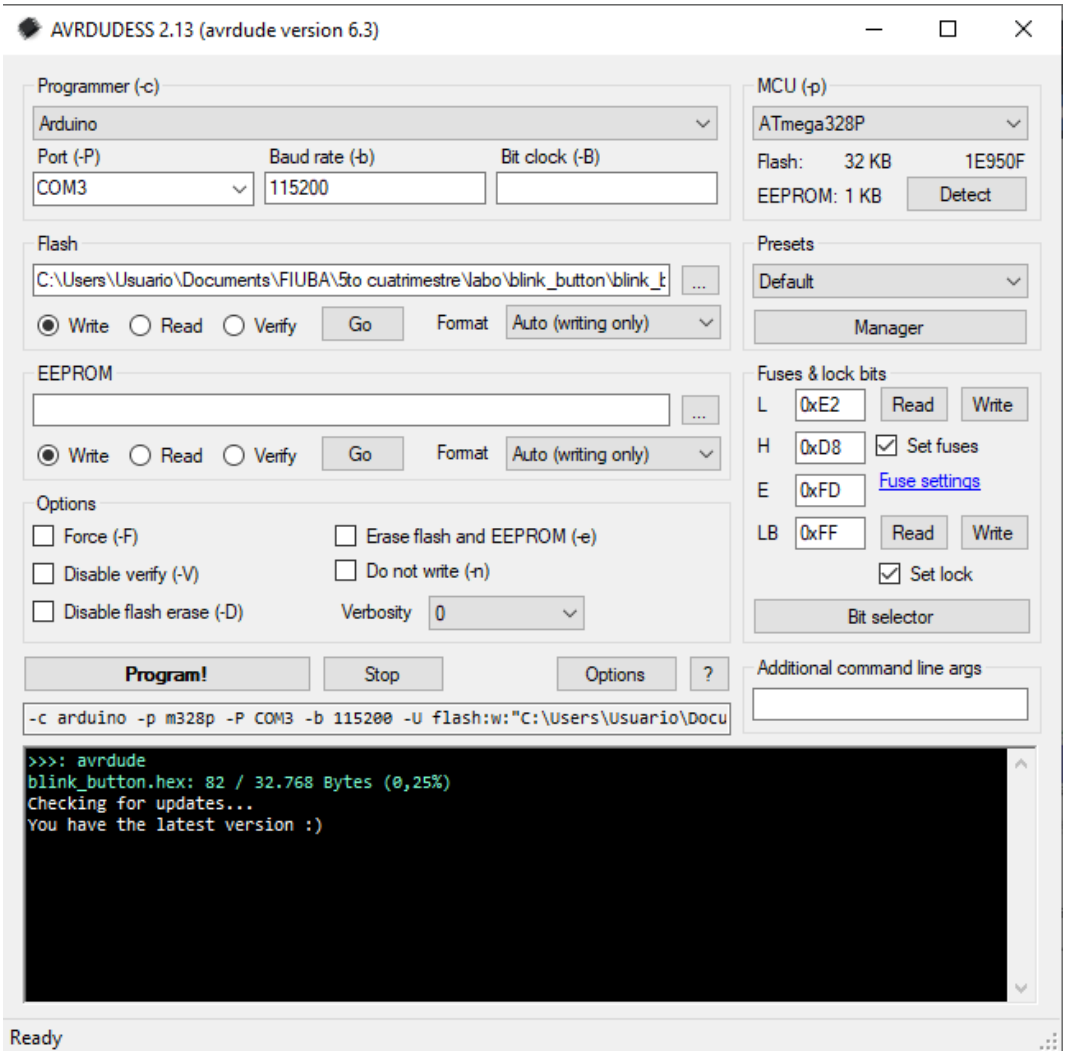

Figura 5: Interfaz de AVRDUDESS

Bit	Low	High	Extended
7	☐ **CKDIV8** *Divide clock by 8*	☐ **RSTDISBL** *External reset disable*	
6	☐ **CKOUT** *Clock output*	☐ **DWEN** *debugWIRE Enable*	
5	☐ **SUT1** *Select start-up time*	☑ **SPIEN** *Enable Serial programming and Data Downloading*	
4	☑ **SUT0** *Select start-up time*	☐ **WDTON** *Watchdog Timer Always On*	
3	☑ **CKSEL3** *Select Clock Source*	☐ **EESAVE** *EEPROM memory is preserved through chip erase*	
2	☑ **CKSEL2** *Select Clock Source*	☑ **BOOTSZ1** *Select boot size*	☐ **BODLEVEL2** *Brown-out Detector trigger level*
1	☐ **CKSEL1** *Select Clock Source*	☑ **BOOTSZ0** *Select boot size*	☐ **BODLEVEL1** *Brown-out Detector trigger level*
0	☑ **CKSEL0** *Select Clock Source*	☐ **BOOTRST** *Select reset vector*	☐ **BODLEVEL0** *Brown-out Detector trigger level*

Figura 6: Fusibles del Atmega328P

4. Señal cuadrada

Para obtener la señal cuadrada, se utiliza un PWM en el timer 2, en modo FAST PWM. Siendo el prescaler igual a 1024 y OCR2A igual a 154, la frecuencia puede calcularse como:

$$f = \frac{8MHz}{1024 \cdot 155} \simeq 50Hz$$

4.1. Esquemático

En la figura 7 se observa el circuito utilizado para conseguir generar y verificar una señal de $50Hz$ por PWM.

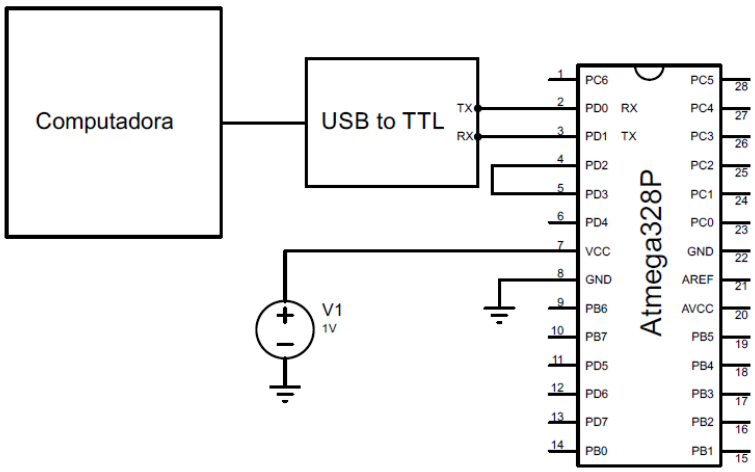

Figura 7: PWM y control de la frecuencia

4.2. Implementación

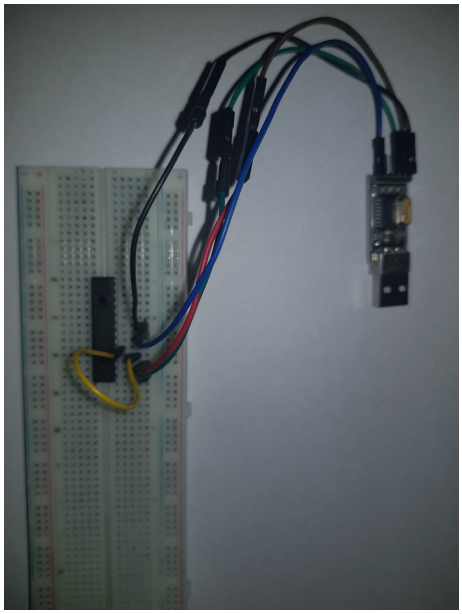

Figura 8: Implementación - PWM y control de la frecuencia

4.3. Diagrama de flujo

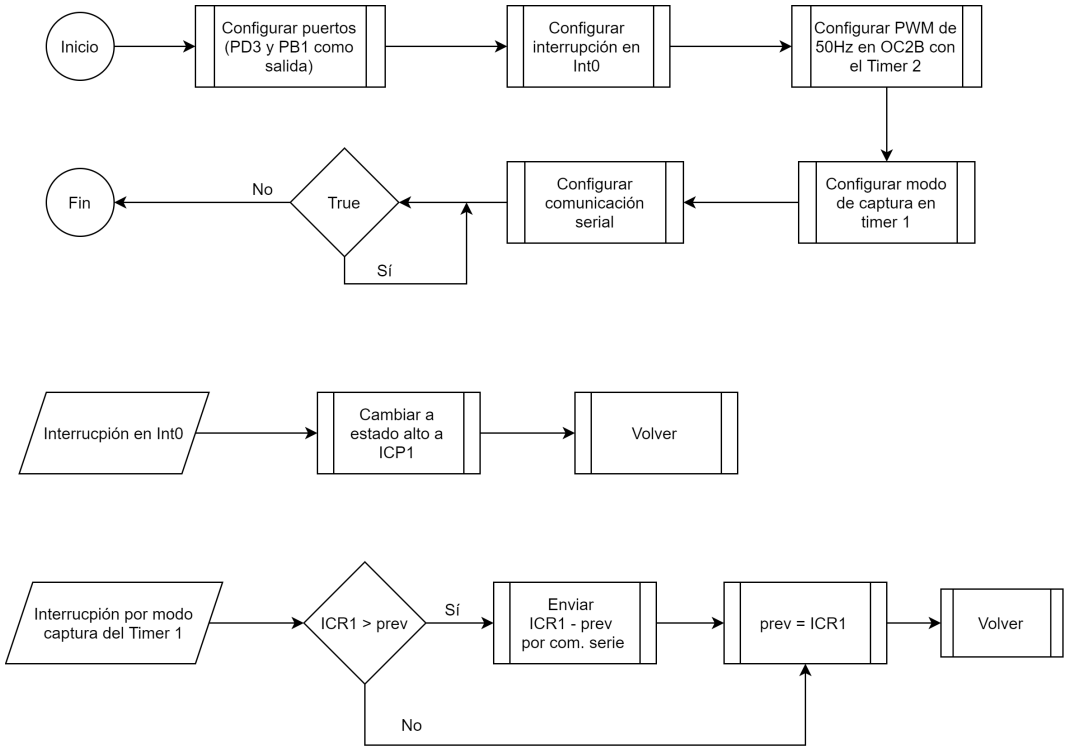

Figura 9: Diagrama de flujo - PWM y control de la frecuencia

4.4. Programación en C

En un todo de acuerdo con lo establecido en el diagrama de flujo, se explican a continuación los segmentos de código que permiten implementar la lógica utilizada.

4.4.1. Inicialización de puertos

Se define PD3 como salida ya que allí se encuentra el terminal OC2B, por donde saldrá la señal del PWM del Timer 2. PB0 también debe colocarse en alto pues allí se encuentra ICP1, lo que resulta necesario para utilizar el modo de captura del Timer 1.

```
DDRD = 0b00001000; //PD3 como salida
DDRB = 0b00000001; //PB0 como salida
```

4.4.2. Interrupción en INT0

Las interrupciones son un mecanismo que permite leer continuamente los cambios lógicos en una entrada e interrumpir el programa en caso de su ocurrencia. Resultan de gran relevancia al momento de tener que realizar operaciones frente a cambios lógicos de una señal digital. Los registros EICRA y EIMSK permiten controlar el comportamiento de la interrupción.

```
EICRA = (1 << ISC00); //Cualquier cambio lógico en INT0 genera una
↪  interrupción
EIMSK = 1 << INT0;
```

4.4.3. PWM en el Timer 2

A partir de los siguientes registros puede definirse una señal de PWM en el Timer 2.

- **TCCR2A y TCCR2B**: Permiten definir el comporatmiento del Timer 2. Siendo que WGM2, WGM1 y WGM0 están en estado alto, el modo de operación de este Timer es fast PWM, donde el tope viene dado por OCR2A y el cambio lógico de la señal se da en OCR2B, de forma no invertida. Al setear COM2B1 se habilita la salida de la señal por el pin OC2B. Los bits CS22, CS21 y CS20 permiten establecer el prescaler como 1024.
- **TCNT2**: En cada instante, contiene el valor del Timer 2.

```
TCCR2A = (1<<COM2B1) | (1<<WGM21) | (1<<WGM20);
TCCR2B = (1 << CS21 ) | ( 1 << CS20 ) | ( 1 << CS22 ) |  ( 1 << WGM22 );
TCNT2=0x00;
OCR2A = 154; //para obtener f = 50 Hz con prescaler 1024
OCR2B = 77; //Duty cycle del 50%
```

4.4.4. Modo captura en el Timer 1

A partir de los siguientes registros puede definirse el modo de captura en el Timer 1.

- **TCCR1A y TCCR1B**: Permiten definir el comporatmiento del Timer 1. Se define nuevamente el prescaler a 1024, mediante los bits CS12 y CS10, para que así tanto el Timer 1 como el Timer 2 tengan la misma escala temporal. El bit ICES1 en estado alto establece que la interrupción por captura sólo se dará cuando haya un flanco ascendente en ICP1.
- **TCNT2**: En cada instante, contiene el valor del Timer 2.
- **TIMSK1**: El bit ICIE1 en estado alto define a la interrupción por captura.

```
TCCR1A = 0;
TCCR1B = (1 << CS10 ) | (1 << CS12 ) | (1 << ICES1 );
TIMSK1 = 1<<ICIE1;
```

4.4.5. Comunicación serie

El código que permite configurar la transmisión en formato 8N1 es el siguiente:

```
UBRR0H = (BPS>>8);
UBRR0L = (BPS);
UCSR0C = (0<<UMSEL00) | (0<<UPM00) | (0<<USBS0) | (3<<UCSZ00);
UCSR0B = (1<<RXEN0) | (1<<TXEN0) | (0<<UCSZ02);
```

A continuación, se explica cada registro con más detalle:

- **UBRR0L** Observar que previo a la carga de este registro, el código carga el valor del *prescaler* en los registros r16 y r17.

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	UBRRn[11:8]				UBRRnH
	UBRRn[7:0]								UBRRnL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Figura 10

Los bits 15 a 12 están reservados para su uso en futuras versiones del microcontrolador, motivo por el cual deben ser incializados a cuatro siempre que se utilize este registro. Los restantes 12 bits definen el *baud rate*, como un número de 12 bits.

- **UCSR0C**: Este registro permite definir la configuración de la codificación que tendrán los bytes que se envían a través del puerto serie.

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

Figura 11

A partir del esquema en la figura 11, se pueden definir las funcionalidades de cada bit. Los bits 6 y 7 seleccionan el modo de operación del USARTn. Los bits 5 y 4 definen a los bits de paridad, mientras que el bit 3 seleccionan el número de bits de paro (*stop bits*). Los bits 2 y 1 definen el tamaño de los caracteres, y el bit 0 se pone en alto solamente en caso de que se use en modo sincrónico.

Así, se definen los valores de estos bits de forma correcta para obtener la configuración deseada.

- **UCSR0B**

  Nuevamente, se definen aquí los bits necesarios para obtener la configuración buscada.

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 12

En este caso, los bits 7 y 6 definen las interrupciones en Rx y Tx. El bit 5 habilita la interrupción en el falg UDREn. Los bits 4 y 3 habilitan al transmisor y receptor n. El bit 2 define la cantidad de bits que usan el transmisor y receptor por segemento. Los bits 1 y 0 son los novenos bits de los caracteres transmitidos, que deben leerse o escribirse antes de realizar cualquier operación de lectura/escritura del caracter.

4.5. Explicación

Con la configuración indicada del Timer 2, se logra una señal de PWM de $50Hz$ y un duty cycle del 50 %. La señal cuadrada es retroalimentada hacia el pin INT0, en donde se configura una interrupción tanto por flanco ascendente como descendente. Es evidente que entre cada dos interrupciones en INT0, debería transcurrir la mitad del periodo de la señal cuadrada. Esta interrupción genera en cada ocurrencia un flanco ascendente sobre el pin ICP1 del microcontrolador, lo que activa a la interrupción por captura en el Timer 1. Esto último se observa en el siguiente extracto del código:

```
ISR(INT0_vect){
  unsigned char sreg;
  sreg = SREG;
  PORTB = 0x01;
  SREG = sreg;
}
```

En cada interrupción por captura, se envía la diferencia entre el instante temporal actual y el instante correspondiente a la interrupción anterior. Esto se logra a partir de una variable auxiliar, *prev*, y del registro TCNT1, cuyo valor es guardado en el registro ICR1. El siguiente código implementa la lógica aquí descripta.

```
ISR(TIMER1_CAPT_vect){
  unsigned char sreg;
  sreg = SREG;
  if(ICR1 > prev)
```

```
    send_number(ICR1 - prev);
  prev = ICR1;
  PORTB = 0;
  SREG = sreg;
}
```

Por último, la comunicación serie se realiza a partir de dos funciones que envían los valores de los números o las cadenas de texto. Observar que se utiliza al bit UDRE0 para saber si el buffer de transmisión ya está vacío.

```
void send_string(char* str){
  for (int i = 0; i < strlen(str) && str[i]!=0; i++){
    while (( UCSR0A & (1<<UDRE0))  == 0){};
    UDR0 = str[i];
  }
  while (( UCSR0A & (1<<UDRE0))  == 0){};
  UDR0 = '\r';
  while (( UCSR0A & (1<<UDRE0))  == 0){};
  UDR0 = '\n';
  return;
}

void send_number(uint8_t n){

  while (( UCSR0A & (1<<UDRE0))  == 0){};
  UDR0 = n;
  return;
}
```

4.6. Resultados

Se observa a continuación que cada 77 o 78 ciclos hay un cambio lógico en la señal cuadrada. Dado que el período está asociado a 155 ciclos, se deduce que el duty cycle es efectivamente del 50 %. El valor de la frecuencia se puede calcular como se hizo previamente.

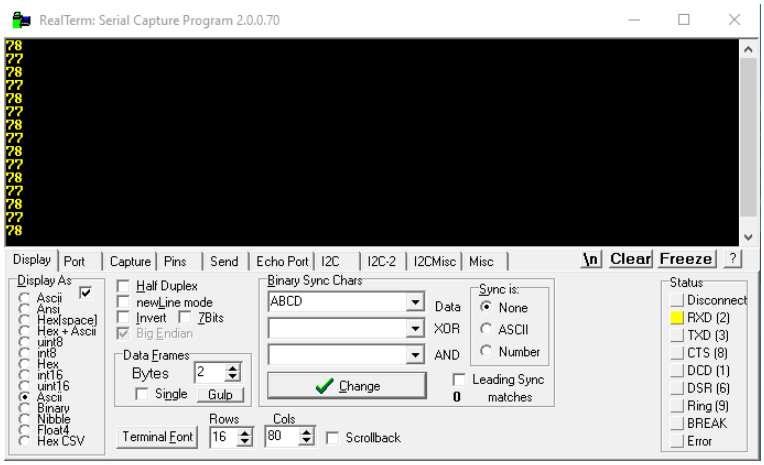

Figura 13: Cantidad de pulsos obtenidas entre cada

5. Carga y descarga sobre un condensador

Se escogieron los valores de R y C de modo que $\tau = RC \simeq 6.67ms$, la tercera parte del periodo de la señal cuadrada. Así, $R = 660\Omega$ y $C = 10uF$.

5.1. Esquemático

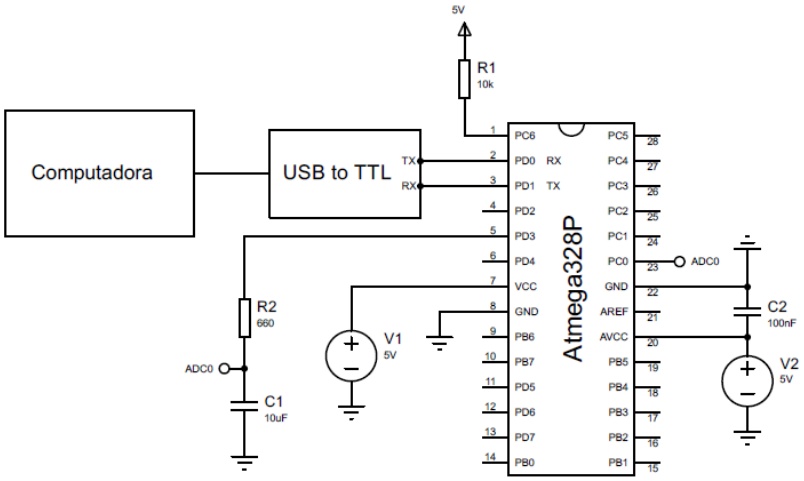

Figura 14: Medición de la tensión sobre un capacitor

5.2. Implementación

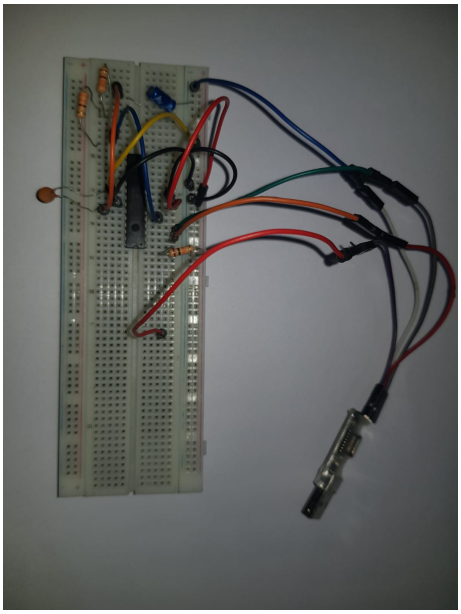

Figura 15: Implementación - Carga y descarga sobre un condensador

5.3. Diagrama de flujo

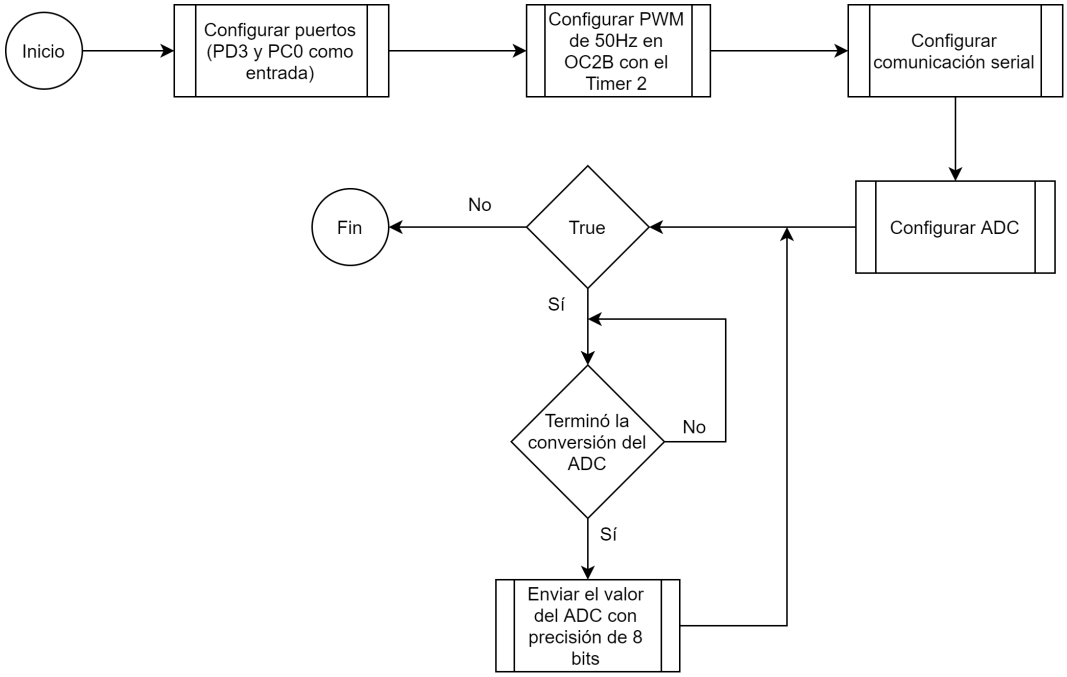

Figura 16: Diagrama de flujo - Medición de la carga y descarga de un capacitor

5.4. Programación en C

Las herramientas utilizadas son similares a las explicadas en la sección correspondiente al control de la frecuencia del PWM.

5.4.1. ADC (Analogue to Digital converter)

- **ADCSRA Y ADCSRB**: Permite definir el comportamiento del ADC. El bit ADPS2 define un prescaler de 64 para la frecuencia de muestreo del ADC, resultando $f_s = 8MHz/64 = 125kHz$. El bit ADEN habilita el ADC. El bit ADSC se encuentra en estado alto durante la conversión.
- **ADMUX**: Los bits REFS0 en alto y RES1 en estado bajo establecen que la referencia de tensión se da con respecto a AVCC, debiendo incluir un capacitor entre AVCC y AREF. El bit ADLAR define un corrimiento a la izquierda del registro de 10 bits *ADC*, conformado por ADCH y ADCL. De este modo, se puede enviar solamente ADCH, obteniendo así una precisión de 8 bits para el ADC (entre 0 y 255).

5.5. Explicación

El tiempo de transmisión para cada número de 8 bits viene dado por

$$t_{transmisin} = \frac{9bits}{9600bits/segundo} \simeq 0.9375ms$$

Son 9 bits porque el formato es 8N1.

A su vez, el tiempo de conversión del ADC es, de acuerdo a lo indicado por la hoja de datos del Atmega328P, de 13 ciclos del clock del ADC. Entonces,

$$t_{conversin} = \frac{13}{125kHz} \simeq 0.104ms$$

Siendo que ambas operaciones se realizan una a continuación de la otra, el tiempo asociado a la medición y tranmisión de cada muestra es

$$t_{muestra} = 0.9375ms + 0.104ms \simeq 1.0415ms$$

Dado que las muestras son enviadas de forma continua, se puede estimar que aproximadamente transcurren $1.0415ms$ entre el envío de dos bytes consecutivos. Entonces, en 20ms de tiempo la cantidad de valores enviados será, aproximadamente,

$$n = \frac{20ms}{1.0415ms} \simeq 19.2$$

A partir de los datos extraídos, se realiza un gráfico de la tensión sobre el capacitor.

5.6. Resultados

En la imagen 17, se evidencia que la tensión obtenida en el tiempo sobre el capacitor es la esperada, y se ajusta a las típicas fórmulas exponenciales desarrolladas de forma teórica sobre los circuitos RC.

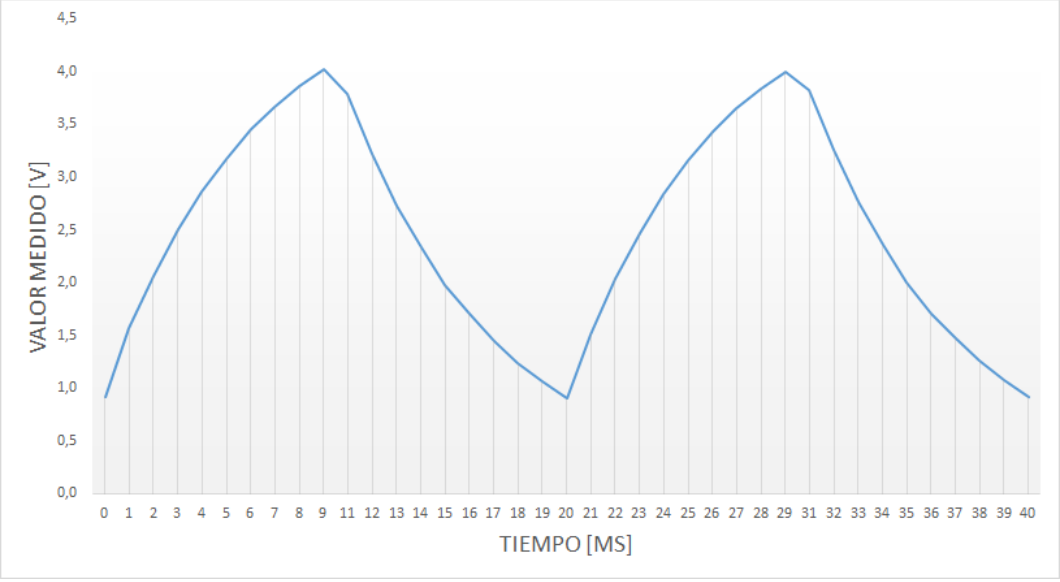

Figura 17: Tensión sobre el capacitor

6. Lista de materiales

Material	costo ($)
3x LED	17.8
Atmega328P	504.7
Arduino nano	539.0
Conversor USB a TTL	305.6
3 x Resistor 220 Ω	12.0
2 x Resistor 330 Ω	8.0
1 x Resistor 1k Ω	3.6
1 x Capacitor 10uF	5.9
1 x Capacitor 100nF	4.0
Total	1400.6

NOTA: Se incluyen los costos del programador. Los precios son de Abril de 2021.

7. Conclusión

El presente proyecto ha puesto en evidencia la utilidad que representan las numerosas herramientas que vienen integradas con los microcontroladores modernos, tales como las interrupciones, los timers, la comunicación serie y el ADC, entre otros. Gracias a su utilización, se ha logrado la medición de una señal analógica sin mayores problemas. En líneas generales, puede decirse que se han logrado los objetivos planteados en este proyecto.

8. Código fuente

8.1. Control de la frecuencia

```
#include <avr/io.h>
#define F_CPU 8000000UL
#define BAUD 9600
#define BPS (F_CPU/16/BAUD - 1)
#include <util/delay.h>
#include <avr/interrupt.h>
#include <string.h>
#include <stdlib.h>

volatile uint16_t actual = 0, prev = 0;

void send_string(char* str);
void send_number(int n);



int main(void){
  DDRD = 0b00001000; //PD3 como salida
  DDRB = 0b00000001;
```

```
  EICRA = (1 << ISC00); //Cualquier cambio lógico en INT0 genera una
  ↪  interrupción
  EIMSK = 1 << INT0;

  TCCR2A = (1<<COM2B1) | (1<<WGM21) | (1<<WGM20);
  TCCR2B = (1 << CS21 ) | ( 1 << CS20 ) | ( 1 << CS22 ) |  ( 1 << WGM22 );
  TCNT2=0x00;
  OCR2A = 154; //para obtener f = 50 Hz con prescaler 1024
  OCR2B = 77; //Duty cycle del 50%

  TCCR1A = 0;
  TCCR1B = (1 << CS10 ) | (1 << CS12 ) | (1 << ICES1 );
  TIMSK1 = 1<<ICIE1;

  UBRR0H = (BPS>>8);
  UBRR0L = (BPS);
  UCSR0C = (0<<UMSEL00) | (0<<UPM00) | (0<<USBS0) | (3<<UCSZ00);
  UCSR0B = (1<<RXEN0) | (1<<TXEN0) | (0<<UCSZ02);

  sei();

  while(1){
  }
}

ISR(INT0_vect){
  unsigned char sreg;
  sreg = SREG;
  PORTB = 0x01;
  SREG = sreg;
}

ISR(TIMER1_CAPT_vect){
  unsigned char sreg;
  sreg = SREG;
  if(ICR1 > prev)
    send_number(ICR1 - prev);
  prev = ICR1;
  PORTB = 0;
  SREG = sreg;
}

void send_string(char* str){
  for (int i = 0; i < strlen(str) && str[i]!=0; i++){
    while (( UCSR0A & (1<<UDRE0))  == 0){};
    UDR0 = str[i];
  }
  while (( UCSR0A & (1<<UDRE0))  == 0){};
  UDR0 = '\r';
  while (( UCSR0A & (1<<UDRE0))  == 0){};
  UDR0 = '\n';
  return;
}
```

```
void send_number(int n){
  char int_string[8];
  itoa(n, int_string, 10);
  send_string(int_string);
  return;
}
```

8.2. Carga y descarga sobre un condensador

```
#include <avr/io.h>
#define F_CPU 8000000UL
#define BAUD 9600
#define BPS (F_CPU/16/BAUD - 1)
#include <util/delay.h>
#include <avr/interrupt.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

void send_string(char* str);
void send_number(uint8_t n);

volatile bool new_data;

int main(void){

  cli();

  DDRD = 0b00001000; //PD3 como salida
  DDRC = 0; // PC0 como entrada


  TCCR2A = (1<<COM2B1) | (1<<WGM21) | (1<<WGM20);
  TCCR2B = (1 << CS21 ) | ( 1 << CS20 ) | ( 1 << CS22 ) |  ( 1 << WGM22 );
  TCNT2=0x00;
  OCR2A = 154; //para obtener f = 50 Hz con prescaler 1024
  OCR2B = 77; //Duty cycle del 50%

  UBRR0 = BPS;
  UCSR0C = (0 << UMSEL00) | (0 << UPM00) | (0 << USBS0) | (3 << UCSZ00);
  UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (0 << UCSZ02);

  ADCSRA  = (1 << ADPS2) | (1 <<ADEN) | (1 <<ADSC); //Para f=125kHz, prescaler =
  ↪  64
  ADMUX = (1 << REFS0) | (1 << ADLAR);


  while(1){

    ADCSRA |= (1 << ADSC);
```

```
    while(ADCSRA & (1 << ADSC)){};

    send_number(ADCH);
  };
}

void send_string(char* str){
  for (int i = 0; i < strlen(str) && str[i]!=0; i++){
    while (( UCSR0A & (1<<UDRE0))  == 0){};
    UDR0 = str[i];
  }
  while (( UCSR0A & (1<<UDRE0))  == 0){};
  UDR0 = '\r';
  while (( UCSR0A & (1<<UDRE0))  == 0){};
  UDR0 = '\n';
  return;
}

void send_number(uint8_t n){

  while (( UCSR0A & (1<<UDRE0))  == 0){};
  UDR0 = n;
  return;
}
```