Sistema de Medición de Carga de Capacitores
Rev.8
Laboratorio de Microprocesadores 86.07 - FIUBA

José F. González
jfgonzalez@fi.uba.ar,

Atento a: Ing. Stola, Ing. Salaya, Ing. Cofman

2021
Febrero

Índice

1. Introducción

Este informe describe el diseño de un circuito para la medición de los procesos de carga y descarga de capacitores, implementado con un microcontrolador AVR ATmega328P programado en Assembly/C.

En la Figura 1 se muestra el circuito propuesto donde la tensión de carga del capacitor C en el circuito de la Figura 1 es una función del tiempo dada por

$$v_C(t) = V_0\dot{(}1 - e^{-t/RC}) \tag{1}$$

donde V_0 indica la tensión final y se define $\tau = RC$ como la Constante de Tiempo y se cumple que en $t = 5\tau$ la tensión alcanza el 99 % de su valor final.

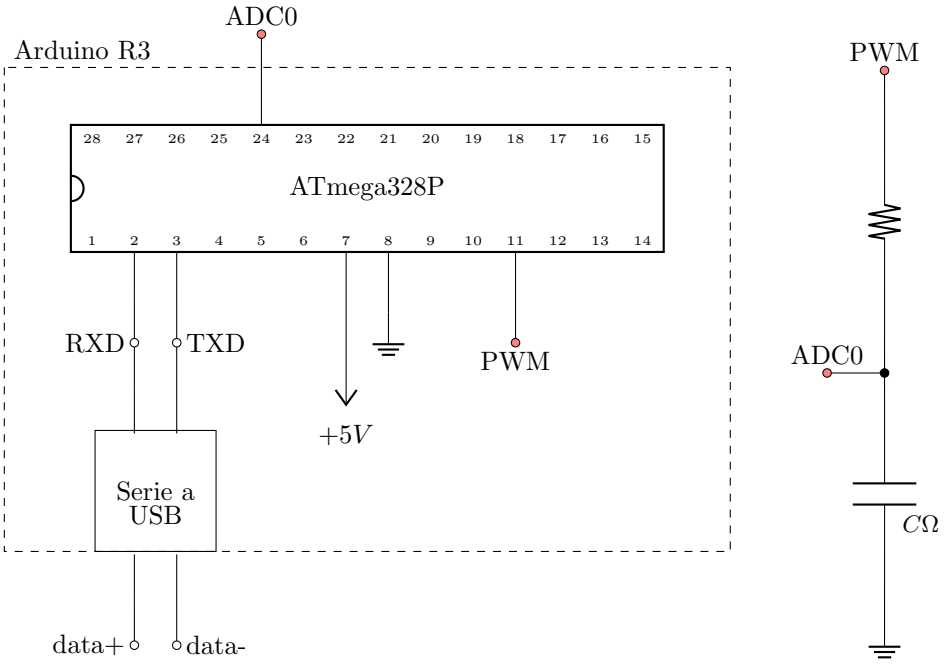

Figura 1: Circuito a implementar.

El circuito RC es alimentado con una señal cuadrada de 62 Hz con ciclo de trabajo del 50 % generada por el Timer 0 del microcontrolador. Se eligen los valores $R = 68\ k\Omega$ y $C = 100\ nF$ tal que $RC = 6{,}8\ ms$ sea aproximadamente tres veces menos que un período de la señal cuadrada - $T = 16ms$ - y permita apreciar las cargas y descargas.

La tensión entre los nodos del capacitor es medida por el conversor analógico-digital del microcontrolador ADC0, procesada y transmitida por el puerto serie hacía un conversor Serie/USB hacía la PC.

2. Diagrama de Bloques

En la Figura 2 se muestra el diagrama de bloques para el sistema propuesto.

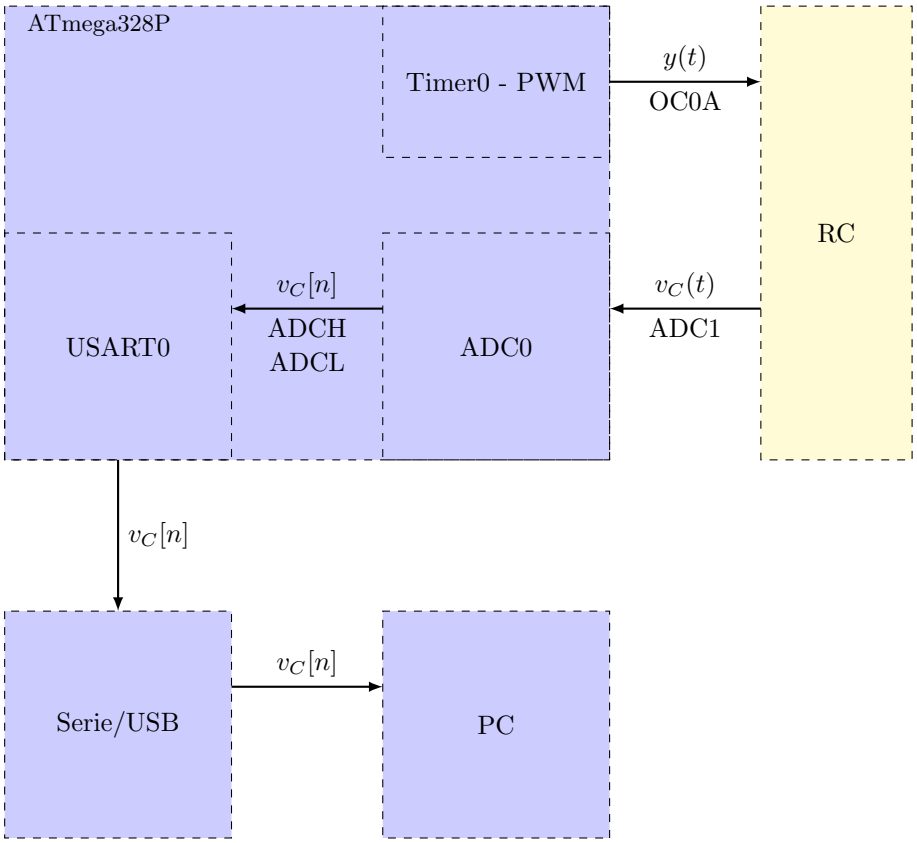

Figura 2: **Diagrama de Bloques.** En Azul: Circuito Digital. En Amarillo: Circuito Analógico.

2.1. Fast PWM

Se utiliza el módulo 8-bit Timer0 para la generación de la señal cuadrada $y(t)$, configurado en modo Fast PWM (COM0A0:1 = 2) no invertido con cuenta hasta TOP = 0xFF (WGM00:2 = 3). La señal se invierte en el puerto OC0A cuando el contador llega a TCNT0 = OCR0A = 126 (El 50 % del TOP = 255)

La frecuencia PWM para un valor de prescaler $N = 1024$ (CS00:2 = 5) es de aproximadamente

$$f_{PWM} = \frac{f_{CLK}}{N \times 256} \approx 61\ Hz \tag{2}$$

Los registros de control se muestran en la Figura 3.

2.2. ADC0

De acuerdo al Teorema de Nyquist el proceso de muestreo, para poder reconstruir la señal cuadrada, debe tener un perído que cumpla

$$w_S > 2W_{y(t)} \rightarrow f_S > 100Hz \tag{3}$$

COM0A1=1	COM0A0=0	COM0B1	COM0B0	TCCR0A
-	-	WGM01=1	WGM00=1	
F0C0A=0	FC0B=0	-	-	TCCR0B
WGM02=0	CS02=1	CS01=0	CS00=1	

Figura 3: **Registros del Timer 0.** En Azul: los bits más importantes.

En rigor, para ver si podemos reconstruir la señal de la tensión $v_C(t)$ debemos analizar la transformada de Fourier de su expresión teórica - no se realiza este análisis en este informe.

Entonces se elige la configuración de la Figura **??** para el conversor analógico digital, donde el prescaler vale $N = 128$ (ADPS0:2=5). Para habilitar cada conversión se usa el Bit ADSC, este método demora 25 cliclos del reloj del ADC - en lugar de 13 ciclos en modo libre - entonces se asegura una frecuencia de muestreo

$$f_S = \frac{16\ MHz}{128} \times \frac{1}{25} = 5\ kHz > \ min(f_S) \tag{4}$$

ADMUX	REFS1=0	REFS0=0	ADLAR=0	-
	MUX3=0	MUX2=0	MUX1=1	MUX0=1
ADCSRA	ADEN=1	ADSC=1	ADATE=0	ADIF
	ADIE	ADPS2=1	ADPS1=1	ADPS0=1

Figura 4: **Registros del ADC** En Azul: Los bits más importantes.

2.3. USART0

Se utiliza el modelo USART0 en modo de comunicación asincrónica de 8 bits con un bit de STOP como se muestra en la Figura 6.

Si el ADC está generando muestras a 9,6 kHz - cada muestra de 16 bits. Para enviar las muestras se mandan los caracteres correspondientes a su representación decimal (0 a 1024) seguidos por un salto de línea para indicar el fin del número. En caso de tener que enviar los cuatro dígitos más el salto de línea el puerto debe poder mandar

$$9{,}6\ kHz \times (8bits + 1\ bit\ stop + 1\ bit\ start) \times 5char = 480000\ bps \tag{5}$$

La tasa normalizada que supera este valor es de 500000 bps que es el valor elegido para la USART, tal que el registro de baud rate tiene un valor de prescaler

$$UBRR0 = \frac{f_{OSC}}{16 \cdot BAUDRATE} - 1 \approx 1 \tag{6}$$

ST	0	1	2	3	4	5	6	7	8	SP1	} FRAME

Figura 5: Formato de Trama

RXCIE0=0	TXCIE0=0	UDRIE0=0	RXEN0=0	} UCSR0B
TXEN0=1	UCSZ02=0	RXB80=0	TXB80=0	

Figura 6: **Registros de la USART** En Azul: Los bits más importantes.

2.4. PC - Gráfico

Para gráficar se muestreo durante un segundo obteniendose 5184 muestras útiles que se graficaron a razón de $1s/5184$. En la Figura 1 se muestra la señal capturada en un segundo donde se recuperó la tensión como

$$v_C = \frac{ADC \cdot V_{REF}}{1024} \tag{7}$$

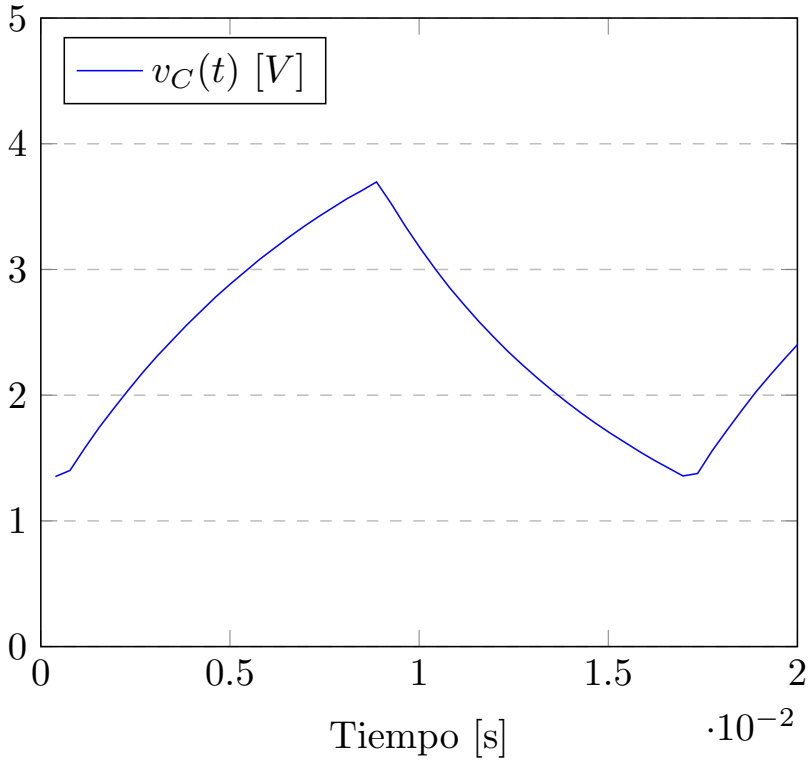

Figura 7: **Señal Reconstruida**. Reconstrucción de la tensión del capacitor.

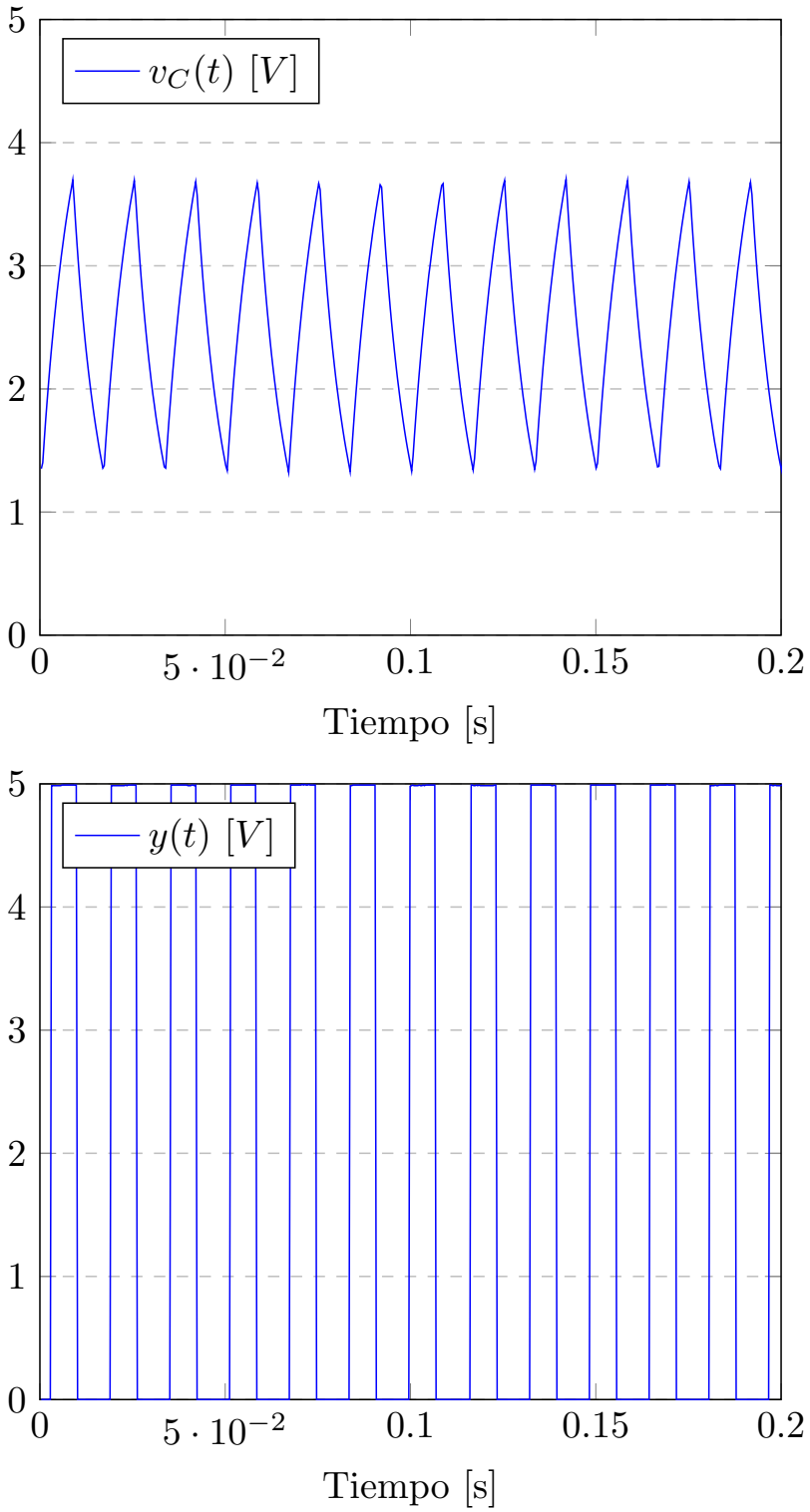

Figura 8: **Arriba:** Reconstrucción de la tensión del capacitor. **Abajo:** Reconstrucción de la señal cuadrada.

3. Programa

En la Figura 9 se muestra el algoritmo del programa implementado donde se distinguen dos partes: una etapa de funciones de incialización y un ciclo infinito de lectura y envío de información.

El programa principal está implementado en lenguaje C con algunos submodulos en Assembly AVR. Se lo compila utilizando AVR/GNU Compiler 5.4.0.

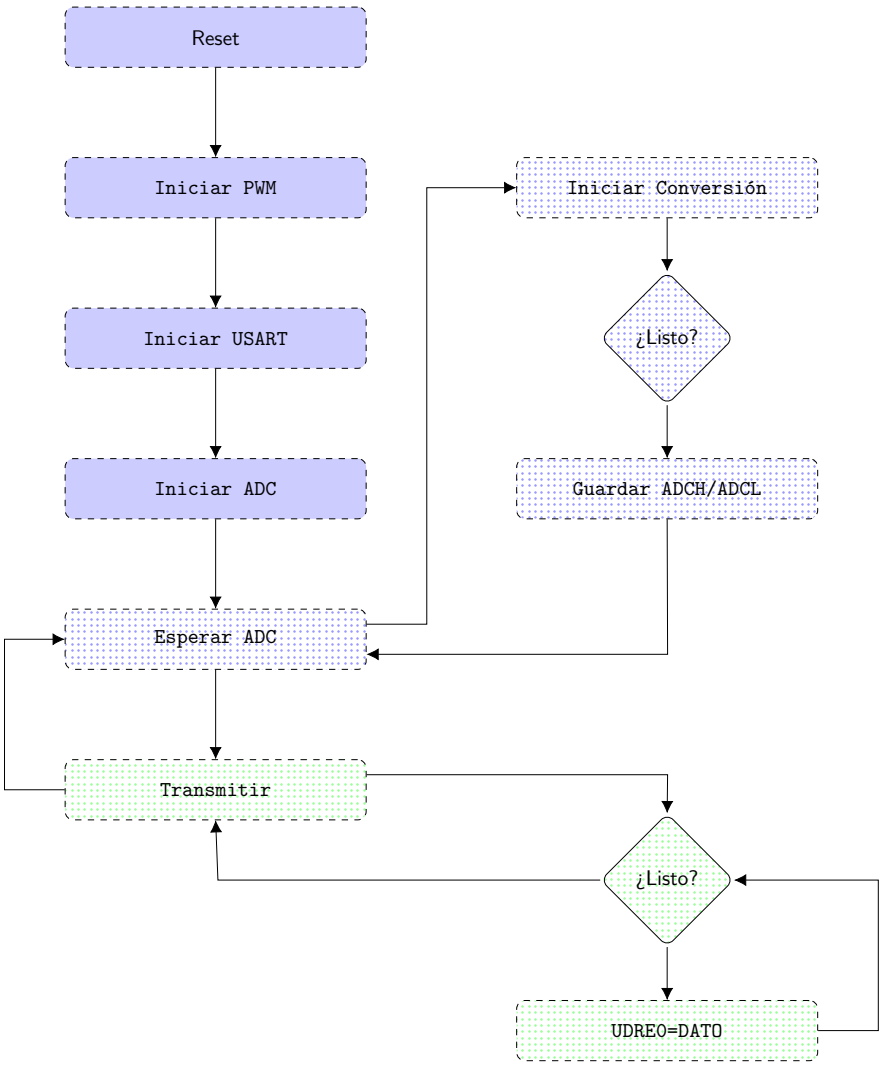

Figura 9: **Algoritmo.**

3.1. usart.h

El modulo más importante del programa consiste en la transmisión serie. La configuración del puerto se realiza una sola vez con los parámetros indicados en el Capítulo anterior. Las siguientes funciones lo convierten en una cadena de caracteres usando una función de C y se envía la cadena con un puntero a cada caracter agregando un salto de línea al final.

En el Cuadro 1 se muestra el código de este segmento del programa.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void uartInit(unsigned int ps)
6  {
7          UCSR0C = (1<<UCSZ01) | (1<<UCSZ00);
8          UCSR0B = (1<<TXEN0);
9          UBRR0 = ps;
10 }
11
12 void uartTX(unsigned char to_send)
13 {
14     while(!(UCSR0A & (1<<UDRE0)));
15     UDR0 = to_send;
16 }
17
18 void sendString(char * str)
19 {
20     while(*str != '\0')
21     {
22         uartTX(*str);
23         str++;
24     }
25 }
26
27 void send(uint16_t data)
28 {
29         char string[2];
30         sprintf(string, "%u", data);
31         sendString(string);
32         uartTX('\n');
33 }
```

Cuadro 1: usart.h

3.2. adc.s

El código del conversor analógico digital está escrito en Assembly AVR. La directiva .comm reserva memoria para las variables que guardaran la lectura del ADC mientras se llama al puerto serie.

En el Cuadro 2 se muestra el código de este segmento del programa.

```
1  .global adcInit
2  .global adcGet
3
4  .comm LOW, 1
5  .comm HIGH, 1
6
7  .global LOW
8  .global HIGH
9
10 adcInit:
11         ldi R16, (1<<MUX0)
12         sts ADMUX, R16
13         ldi R16, (1<<ADEN)|(1<<ADSC)|(1<<ADPS2)|(1<<ADPS1)
14         |(0<<ADPS1)
15         sts ADCSRA, R16
16         ret
17
18 adcGet:
19         wait:
20                 lds R16, ADCSRA
21                 sbrs R16, ADIF
22                 rjmp wait
23
24                 lds R16, ADCL
25                 sts LOW, R16
26
27                 lds R16, ADCH
28                 sts HIGH, R16
29
30         lds R16, ADCSRA
31         ori R16, (1<<ADSC)
32         sts ADCSRA, R16
33         ret
```

Cuadro 2: adc.s

3.3. main.c

El programa principal realiza los llamados a las rutinas anteriores. Las variables y funciones definidas como *extern* son buscadas por el compilador entre los otros archivos del proyecto. La variable definida como *uint16_t* es el resultado del ADC concatenado en una sola variable que se pasa a la función send() que lo procesa.

En el Cuadro 3 se muestra el código de este segmento del programa.

```
1  #include <avr/io.h>
2  #include "usart.h"
3
4  #define F_CPU 16000000UL
5  #define BAUD 500000
6  #define PRESCALER 1
7
8  extern void pwmInit();
9  extern void adcInit();
10 extern void adcGet();
11
12 extern uint8_t HIGH, LOW;
13
14 uint16_t adc_data;
15
16 int main(void)
17 {
18         pwmInit();
19         uartInit(PRESCALER);
20         adcInit();
21
22         while(1)
23         {
24                 adcGet();
25                 adc_data =  LOW;
26                 adc_data |=  HIGH << 8;
27                 send(adc_data);
28         }
29
30         return 0;
31 }
```

Cuadro 3: main.c

3.4. pwm.s

El código que ejecuta el PWM está escrito en Assembly AVR y se ejecuta una sola vez al inicio del programa. La directiva de compilador GNU .global permite que la función pueda ser llamada desde el programa principal.

En el Cuadro 4 se muestra el código de este segmento del programa.

```
.global pwmInit

pwmInit:
        sbi DDRD, 6


        ldi R20, (1<<CS02)|(1<<CS00)
        out TCCR0B, R20

        ldi R20, (1<<COM0A1)|(1<<WGM01)|(1<<WGM00)
        out TCCR0A, R20


        ldi R20, 127
        out OCR0A, R20
        ret
```

Cuadro 4: pwm.s

4. Conclusiones

Se implementó el circuito deseado y se verificó su funcionamiento quedando registrado en el siguiente video (Ver Video). No se encontraron problemas de implementación.

5. Bibliografía

- M. Mazidi - The AVR Microcontroller and Embedded System using Assembly and C - Prentice Hall - Primera Edición.
- ATmega328P Datasheet - 2015 Atmel Corporation. / Rev.: 7810
- AVR Instruction Set Manual - 2020 Microchip Technology Inc.
- GNU Documentation - http://web.mit.edu/gnu/doc/html/as_7.html