Práctica 3

Programación de Microcontroladores

Abrante Delgado, Rubén[1], y Barroso Alonso, Andrea[1]

[1]Informática Industrial (339394103), Grado en Ingeniería Electrónica Industrial y Automática (2020 - 2021).

Resumen

El principal objetivo de este informe es el de explicar las suposiciones de montaje llevadas a cabo para abordar un problema específico mediante la programación de un microcontrolador AT90S8515, conforme a la práctica 3 de la asignatura de Informática Industrial (39394103 - GIEIA 20/21). Esta experiencia plantea elaborar un programa que genere una señal PWM en función de un número de *switches* a uno. Dicha señal deberá variar su ancho de pulso en una progresión geométrica, dependiendo del número de *switches* en alta. La salida se verá afectada solamente cuando se accione un pulsador.

Índice

Índice de figuras

1. Planteamiento del problema

Elaborar un programa en el microcontrolador AT90S8515 que genere una señal PWM. Para ello, se dispondrá de 10 *switches* y un pulsador como entradas. El problema consistirá en que el ancho de pulso de la señal deberá tener una progresión geométrica de 2 dependiendo del número de *switches* que se encuentren a uno, aunque la señal solo se verá afectada a la salida tras la activación del pulsador.

2. Suposiciones del montaje

Para comenzar, tendremos que determinar algunos puntos según el planteamiento del problema.

2.1. Interrupciones

Solo emplearemos 2 interrupciones: con el **RESET** indicaremos el comienzo del programa y el **INT0** se producirá por la activación del pulsador. De esta forma la salida solo cambiará cuando se produzca dicha interrupción.

```
.include "8515def.inc"

.org $000       rjmp RESET      ; Reset Handler
.org INT0addr   rjmp EXT_INT0   ; IRQ0 Handler
```

Figura 1. Vectores de interrupción.

Las direcciones de los vectores de interrupción se encuentran definidas previamente al incluir el archivo de definiciones **"8515def.inc"**. Estas direcciones se encuentran en las posiciones más bajas de memoria.

```
.equ    INT0addr=$001   ;External Interrupt0 Vector Address
.equ    INT1addr=$002   ;External Interrupt1 Vector Address
.equ    ICP1addr=$003   ;Input Capture1 Interrupt Vector Address
.equ    OC1Aaddr=$004   ;Output Compare1A Interrupt Vector Address
.equ    OC1Baddr=$005   ;Output Compare1B Interrupt Vector Address
.equ    OVF1addr=$006   ;Overflow1 Interrupt Vector Address
.equ    OVF0addr=$007   ;Overflow0 Interrupt Vector Address
.equ    SPIaddr =$008   ;SPI Interrupt Vector Address
.equ    URXCaddr=$009   ;UART Receive Complete Interrupt Vector Address
.equ    UDREaddr=$00a   ;UART Data Register Empty Interrupt Vector Address
.equ    UTXCaddr=$00b   ;UART Transmit Complete Interrupt Vector Address
.equ    ACIaddr =$00c   ;Analog Comparator Interrupt Vector Address
```

Figura 2. Direcciones de los vectores de interrupción.

Durante la ejecución del programa principal, debemos habilitar y configurar la **INT0**. Existen tres modos posibles para disparar la interrupción: **a bajo nivel, en flanco de subida y en flanco de bajada**. Lo haremos para que cuando se produzca un **flanco de subida** se produzca la interrupción, es decir, en el momento en que se active el pulsador (Figura 3).

```
in r16, GIMSK
sbr r16, (1 << INT0)
out GIMSK, r16                          ; Habilitamos INT0

in r16, MCUCR                           ; Configuramos INT0 para que
sbr r16, (1 << ISC00) + (1 << ISC01)    ; dispare una interrupción
out MCUCR, r16                          ; en el flanco de subida
```

Figura 3. Configuración de la interrupción INT0.

También habría sido posible elegir el disparo en flanco de bajada, provocando el disparo en cuanto soltamos el pulsador. La activación a bajo nivel es inviable, ya que podrían ejecutarse cientos de interrupciones involuntarias. Finalmente, para habilitar o deshabilitar las interrupciones, no debemos olvidar las siguientes instrucciones:

```
sei     ; Habilita las interrupciones
cli     ; Deshabilita las interrupciones
```

Figura 4. Instrucciones SEI y CLI para habilitar y deshabilitar las interrupciones.

2.2. Pila

Inicializaremos la pila (Stack Pointer) a la dirección más alejada en memoria. Aunque no la utilicemos durante la ejecución de nuestro programa, será necesaria para guardar las direcciones de memoria a la vuelta de las rutinas de interrupción

```
ldi r16, high(RAMEND)
out SPH, r16
ldi r16, low(RAMEND)      ; RAMEND contiene la posición más alejada en memoria
out SPL, r16
```

Figura 5. Inicialización de la pila.

2.3. Entradas

Para poder procesar el estado de los *switches* y el pulsador debemos configurar los puertos como entradas del microcontrolador. Asignaremos los puertos A y B como los *switches* y el PD2 como el pulsador. Esta asignación se hace al poner a cero los bits adecuados en los puertos DDRx (Figura 6).

En principio, solo necesitaremos 8 entradas del puerto A (**PORTA**) y 2 entradas del puerto B (**PORTB**) para conectar nuestros 10 switches ideales. No pasa nada por declarar el resto de pines no usados en **PORTB** como entradas, siempre que se encuentren en un estado definido de cero lógico para nuestro programa.

```
clr r16             ; 0b00000000 en r16
out DDRA, r16
out DDRB, r16       ; PORTA y PORTB son entradas
cbi DDRD, DDD2      ; PD2 es una entrada
```

Figura 6. Declaración de pines de entrada.

El programa de simulación *AVR Studio* es capaz de leer **archivos de estímulo**. Son un tipo de archivos muy simple que consiste en líneas de texto que contienen dos parámetros: **ciclos de reloj y un número hexadecimal**. Dentro del programa, una vez seleccionado el puerto (PORTA, PORTB, PORTC, PORTD) al que queremos redirigir el fichero de estímulos, el número hexadecimal corresponderá con los bits del puerto que se activarán (si son entradas) en dicho ciclo de reloj.

```
000100000:01    ;0b0000_0001
000200000:03    ;0b0000_0011
000300000:07    ;0b0000_0111
000400000:0F    ;0b0000_1111
```

Figura 7. Ejemplo de archivo de estímulos.

Es decir, si redirigimos el fichero del ejemplo (Figura 7) a PORTA, y todos los pines de este puerto son entradas, entonces:

- En el ciclo de reloj 100000, se pondrá a uno el PINA0
- En el ciclo de reloj 200000, se pondrán a uno los pines PINA0 y PINA1
- En el ciclo de reloj 300000, se pondrán a uno los pines PINA0, PINA1 y PINA2
- En el ciclo de reloj 400000, se pondrán a uno los pines PINA0, PINA1, PINA2 y PINA3

Y así, sucesivamente. Resulta conveniente para comprobar que el programa se comporta de forma adecuada. De la misma manera, podríamos crear un archivo de estímulos para activar y desactivar el pulsador, redirigiendo el fichero hacia PORTD y habilitando el bit correspondiente.

2.4. Salidas

La única salida que tenemos que definir es la del pin PD5, que es en la que se va a encontrar nuestra señal PWM. Debemos habilitarla como salida poniendo un uno lógico en el bit DDxn correspondiente al DDRx:

```
sbi DDRD, DDD5      ; PD5 es una salida
```

Figura 8. Declaración de pines de salida.

AVR Studio permite generar archivos (también conocidos como *logs*) que contienen información sobre algunos registros durante la ejecución de una simulación. Para poder visualizar la señal PWM, debemos redirigir la información que aparece en el pin de salida a un puerto que no estemos usando (por ejemplo, PORTC). Además, tendremos que estar continuamente leyendo el pin de salida para saber que valor toma en cada momento. Por ello, debemos crear un bucle infinito en el que se realice esta acción (Figura 9).

```
LOOP:                       ; Definimos un bucle infinito
    in r16, PIND
    out PORTC, r16          ; Obtenemos la PWM en PORTC
    rjmp LOOP
```

Figura 9. Lectura de la señal PWM en un bucle infinito.

De hecho, no solo vamos a obtener la información de la PWM, sino también cuando se activa o desactiva el pulsador, ya que al redirigir todo el registro de PIND a PORTC, tendríamos acceso también a este dato. El *log* generado tras una simulación es exactamente de la misma naturaleza que el archivo de estímulos.

2.5. Timer/Counter1

Para crear la señal PWM es necesario usar un contador. Primero debemos configurarlo para que sea de la forma deseada mediante los registros TCCR1A y TCCR1B. En el TCCR1A pondremos a 1 el COM1A1, el cual indicará la forma en la que se comparará el *timer* con el valor de comparación OCR1A, y los bits PWM11 y PWM10 harán que la señal PWM sea de 10 bits.

```
clr r16
out OCR1AH, r16             ; Ciclo de trabajo al 0%
out OCR1AL, r16

ldi r16, 0b1000_0011        ; PWM no invertida, 10 bits de resolución
out TCCR1A, r16

in r16, TCCR1B
sbr r16, (1 << CS10)        ; Ponemos en marcha la PWM a una frecuencia de CK
out TCCR1B, r16             ; (CK = 8 MHz)

clr r16
out TCNT1H, r16             ; Reset del counter
out TCNT1L, r16
```

Figura 10. Configuración del Timer/Counter1.

Por otro lado, en el TCCR1B los bits CS12, CS11 y CS10 marcarán la frecuencia de la señal, es necesario poner alguno de ellos a 1 para elegir un preescalado, puesto que si están a 0 el contador no avanzará.

Es importante comprender que, al necesitar que el ancho de pulso se comporte en una progresión geométrica de dos, **debemos elegir un *timer/counter* que permita una PWM con una resolución igual al número de *switches***. El *Timer/Counter1* es un contador de 16 bits con estas capacidades.

2.6. Contar el número de entradas a uno

Durante la interrupción, necesitaremos contar el número de entradas que estén a uno. Para ello, lo primero que haremos será leer el puerto de entrada en un registro de propósito general, y luego lo pasaremos a través de un bucle contando el número de bits que están a uno. Un truco para hacer esto sería **desplazar** dicho registro **a la derecha**: si el bit que ha desaparecido era un uno, se activará el bit de acarreo en SREG:

```
in r16, PINA                    ; Leemos las entradas del puerto A
clr r17                         ; Ponemos a cero los registros r17 y r0
clr r0
clc                             ; Limpiamos el bit de acarreo en SREG
COUNT_PINA_BITS:
    lsr r16                     ; Desplaza el registro a la derecha
    breq TERMINATE_PINA         ; Si no quedan 1, salimos del bucle
    adc r17, r0                 ; Suma el bit de acarreo si era un 1
    rjmp COUNT_PINA_BITS
TERMINATE_PINA:
    adc r17, r0                 ; Suma el bit de acarreo si procede
```

Figura 11. Contador de bits (Puerto A).

Si al desplazar el registro a la derecha eliminamos un cero, el bit de acarreo no se activa y, por lo tanto, al ejecutar la instrucción suma con acarreo, sumamos un cero. El bucle se repetirá hasta que el registro en r16 sea $0b00000000$. Después haremos lo mismo con las entradas del puerto que nos queda:

```
in r16, PINB                    ; Leemos las entradas del puerto B
clr r0                          ; Hacemos lo mismo que en el caso anterior
clc
COUNT_PINB_BITS:
    lsr r16
    breq TERMINATE_PINB
    adc r17, r0
    rjmp COUNT_PINB_BITS
TERMINATE_PINB:
    adc r17, r0                 ; r17 contiene el número de switches a uno
```

Figura 12. Contador de bits (Puerto B).

Tras haber pasado por estos dos bucles, tendremos en r17 el número total de *switches* a uno. Este método tiene la ventaja de que no tenemos que recorrer todo el registro para contar todos los *switches*. Desde el momento en el que uno de ellos se haga cero, termina el bucle.

2.7. Progresión geométrica

Para la progresión geométrica de 2 variaremos el valor de comparación OCR1A en función del número de *switches* que estén a 1. Para ello, haremos en un registro un desplazamiento de bits a la izquierda en función del recuento de los puertos A y B, de esta forma cada desplazamiento implicará una multiplicación por 2 con respecto al valor anterior (Figura 13). Dicho registro lo iniciaremos a 2 cuando exista al menos 1 *switch* a 1. Así, este será el ancho de pulso más pequeño.

```
clr r16
out OCR1AH, r16          ; Desde $0000 (BOTTOM) a $03FF (TOP)

cpi r17, $00
breq SET_PWM
                         ; Empezamos por el valor más bajo que puede tomar OCR1A
ldi r16, $02             ; Ponemos a 1 el bit 1 de r16
cpi r17, $01
breq SET_PWM

dec r17
SHIFT_LOW:
    lsl r16              ; Lo multiplicamos por 2 tantas veces como switches a uno
    breq OVF             ; Overflow cuando hay 8 bits (128 << 1 = 256 -> $00)
    dec r17
    brne SHIFT_LOW

SET_PWM:                 ; Si r17 <= 8 switches
    out OCR1AL, r16
    sei
    reti
```

Figura 13. Calculamos la nueva PWM.

Quizás, esta es la parte más complicada de entender del algoritmo de interrupción. Los registros OCR1AH y OCR1AL conforman, esencialmente, el ciclo de trabajo de la PWM. Como hemos configurado la señal PWM para 10 bits, podremos variar este registro entre $0000 y $03FF (es decir, entre 0 y 1023 valores posibles). Al ser un contador lineal, los incrementos lineales en este registro siempre van a implicar un cambio en el ancho de pulso en la misma proporción.

- Si el número de switches es cero, no hacemos nada. Escribimos un $0000 en OCR1A
- Si el número de switches es uno, escribimos un $0002 en OCR1A (El mínimo ancho de pulso que podemos registrar en la PWM).
- Si el número de switches es más de uno, desplazamos el $0002 a la izquierda tantas veces como *switches* a uno tengamos.

Un desplazamiento de 1 bit a la izquierda equivale a multiplicar por 2, por lo que estamos avanzando continuamente en potencias de 2 hasta el valor máximo (Ej. 2, 4, 8, 16, 32, 64, 128, 256, 512 y 1024).

2.8. Overflow

Es importante tener en cuenta cuando se puede producir un overflow. Al desplazar los bits del registro a la izquierda cuando haya 8 *switches* a 1, el contador volverá a 0. Por lo tanto es necesario tener en cuenta el registro SREG, en concreto el bit Z (Zero flag) ya que indicará si la operación realizada en ese momento es igual a 0. Así, empleando un *breq* (Branch if Equal) tras el desplazamiento de bits sabremos si se produjo un overflow.

```
OVF:                                    ; si r17 > 8 switches
    out OCR1AL, r16

    ldi r16, $01
    SHIFT_HIGH:
        lsl r16
        dec r17
        brne SHIFT_HIGH
    lsr r16
    cpi r16, $04                        ; Nos hemos pasado del TOP
    breq SET_MAX_PWM

    out OCR1AH, r16
    sei
    reti

    SET_MAX_PWM:
        ldi r20, $03
        ldi r21, $FF
        out OCR1AL, r21                 ; Ciclo de trabajo al 100%
        out OCR1AH, r20
        sei
        reti
```

Figura 14. Cálculo de la PWM en caso de overflow.

Cuando pasamos de $0080 a $0100, se produce un overflow, por lo que el salto condicional nos lleva a una etiqueta (OVF) que se ocupa de limpiar la parte baja de OCR1A y vuelve a realizar el mismo proceso con la parte alta.

Cuando tenemos más de 8 *switches*, como sólo es posible que estén activados 2 *switches* de más, el método es ligeramente diferente (no es necesario tener en cuenta el overflow en esta porción de código). En el caso de llegar a los 10 *switches*, como el valor $0400 está fuera del rango de la PWM, nos encargamos de escribir exclusivamente el valor máximo posible ($03FF), lo cual corresponde a un valor fijo en alta en la señal de salida.