

CIRCUITOS DIGITALES Y MICROCONTROLADORES (E0305)
AGOSTO 2020

TRABAJO PRACTICO NRO. 4

Autor:
Nehuen Pereyra (878/6)

1. Interpretación del problema 1

Se nos pide poder controlar la intensidad de un LED utilizando PWM, la intensidad va ser proporcional al valor de tensión medido de una entrada analógica.

1.1. Detalle del problema a resolver

Se utilizara el modulo ADC para medir una entrada analógica en el terminal PTA0 (este valor de tensión estará entre 0 y 3.3 V) con el valor medido se generará una onda PWM cuyo ciclo de trabajo (de 0 % a 100 %) sea proporcional al rango de tensión medido.

Para que el usuario no detecte el parpadeo del led se debe elegir una frecuencia mayor a los 50 Hz para la señal de PWM, además se debe generar con el TPM1, utilizando el canal 1 es decir el terminal PTC1.

2. Implementación problema 1

Para resolver este problema utilizaremos una arquitectura del tipo Background/Foreground por lo cual las tareas se ejecutan siempre en el super loop en forma de ciclo, en función de los eventos asociados a las interrupciones. Para este problema con este tipo de arquitectura es suficiente para poder resolver el problema.

En el programa principal lo que haremos es medir la entrada analógica para luego configurar el módulo del canal del TPM1, para generar la señal PWM con el ciclo de trabajo correspondiente, para esto analizaremos estos dos módulos y sus configuraciones (Módulo ADC y TPM).

2.1. Módulo TPM

El microcontrolador posee dos módulos exactamente iguales de TPM (Timer Pulse-Width Modulation). El TPM sirve para temporizar interrupciones, generar ondas PWM o hacer mediciones de tiempo entre eventos proveniente de periféricos.

Cada TPM está formado por un contador de 16 bits (TPMxCNT) que se incrementa continuamente a una frecuencia que puede ser: la frecuencia del clock de bus predividida, clock fijo de sistema o un clock externo.

Además el TPM tiene dos canales, cada uno asociado a un pin, que permiten, entre otras cosas, hacer un cambio en el nivel lógico de su pin cuando el contador del TPM (TPMxCNT) coincide con un valor propio del canal (TPMxCnV), disparando una interrupción manejable por el software.

2.1.1. Generación de la onda PWM

La modulación del pulso (PWM) es una técnica de modulación digital donde la información útil de la señal se encuentra en el ancho del pulso. La señal de PWM se genera de forma automática una vez tenemos configurados los registros necesarios del TPM. Analizaremos la generación de una onda PWM alineada por flanco (edge aligned) en este modo el valor del registro de módulo (TPMMOD) establece el periodo de la señal PWM y el registro de canal (TPMCxV) establece el ciclo de trabajo. Además se puede seleccionar la polaridad del pulso es high true o low true. Este modo se llama alineado al flanco porque al comienzo del periodo las señales de todos los canales están alineadas y todas tienen el mismo periodo. En la figura 1 podemos ver cómo se genera la señal de PWM cuando tenemos los registros configurados y como podemos ir cambiando el ancho de pulso al modificar el valor del registro de canal.

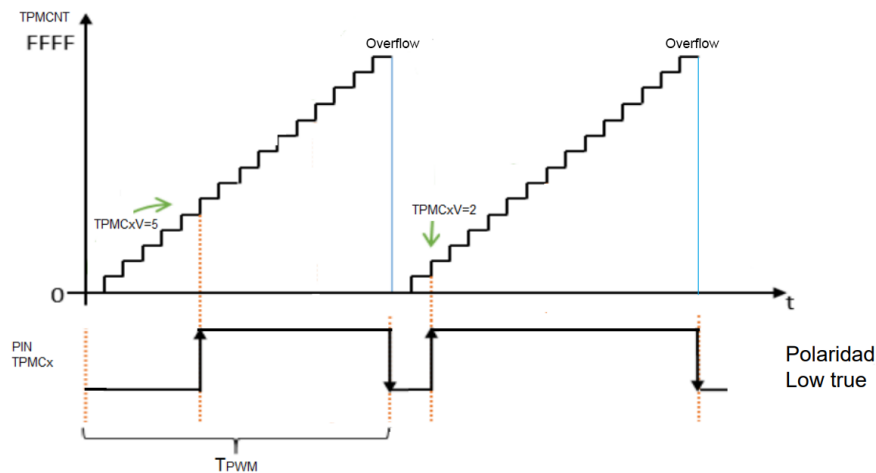


Figura 1: Generación de la señal PWM.

2.1.2. Configuración utilizada

La configuración que utilizaremos para resolver el problema planteado se puede ver en la figura 2. Podemos ver en el Device Initialization del Codewarrior que seleccionamos como fuente de reloj a Bus clock, con un preescalador de 1 y un módulo de contador de 255 para tener una resolución de PWM de 8 bits más que suficiente para controlar la intensidad de un LED. Como podemos ver la frecuencia es mayor a 50Hz que es la frecuencia mínima exigida en el enunciado. Luego seleccionamos el canal por el cual se genera la señal PWM en esta caso el canal 1 y establecemos la polaridad en este caso Clear output on compare para que sea low true. Por último habilitamos las interrupciones de canal y de overflow.

2.2. Módulo ADC

El ADC es un módulo conversor analógico a digital integrado del tipo de aproximaciones sucesivas. El módulo puede realizar conversiones con precisión configurable de 8 o 10 bits. El tiempo de muestreo y la tasa son configurables también tiene dos modos para realizar la conversión de modo continuo o discontinuo. Al finalizar la conversión puede activar un flag con interrupción. Se puede elegir como fuente de reloj:

- El clock de bus.
- La mitad del clock de bus.
- El reloj alternativo derivado del sistema de reloj del microcontrolador.
- Oscilador interno del ADC.

Cualquiera de estas fuentes de reloj pueden ser divididas por: 1, 2, 4 y 8.

Se puede ver en el diagrama de bloques de este módulo en la figura 3.

Una vez configurado el módulo ADC para utilizarlo, cuando se termina de hacer una conversión

Device Initialization - Init_TPM

Component Parameters

Name	Value	Details
Component name	TPM3	
Device	TPM1	TPM1
Settings		
Clock settings		
Clock source select	Bus rate clock	
Prescaler	1	
Modulo counter	255	
Period	32 us	
Channels	1	
Channel		
Capture/compare device	TPM11	TPM11
Settings		
Mode	PWM	
PWM output action	Clear output on compare	
Channel compare value	64	
Channel duty	8 us	
Pin		
Channel pin	PTC1_TPM1CH1_ADP9	PTC1_TPM1CH1_ADP9
Channel pin signal		
Pull resistor	autoselected pull	no pull resistor
Interrupt		
Channel Interrupt		
Interrupt	Vtpm1ch1	Vtpm1ch1
Channel Interrupt	Enabled	
ISR name	isrVtpm1ch1	
> Pins		
Interrupts		
Overflow Interrupt		
Interrupt	Vtpm1ovf	Vtpm1ovf
Overflow Interrupt	Enabled	
ISR name	isrVtpm1ovf	
Initialization		
Enable module	yes	

Figura 2: Configuración del TPM.

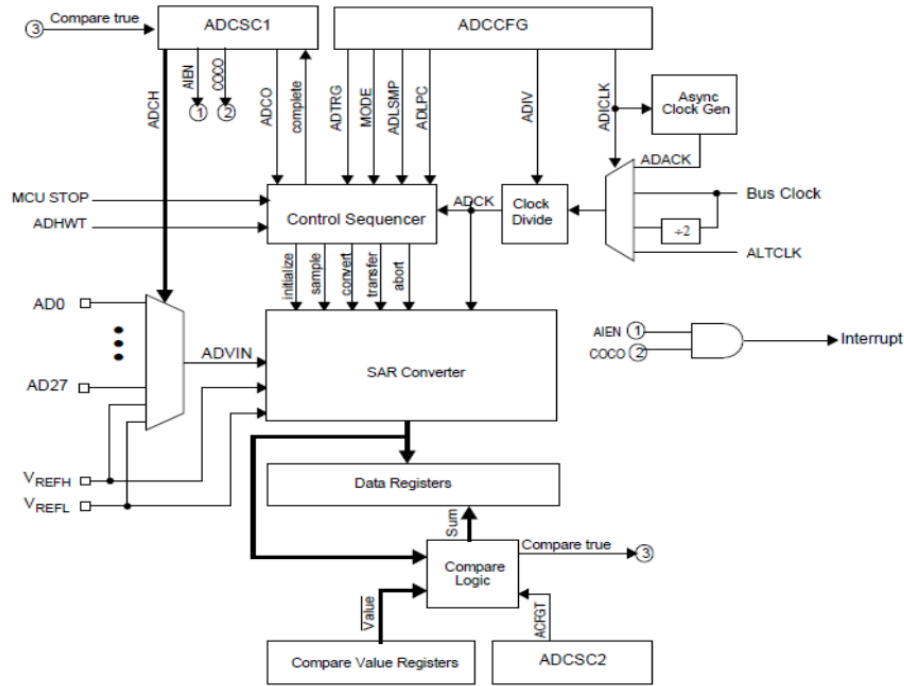


Figure 9-2. ADC Block Diagram

Figura 3: Diagrama en bloques del módulo ADC.

se activa el flag COCO poniéndose en alto, entonces podemos leer el valor del registro ADCR para obtener el resultado de la conversión (al leerse el resultado el flag de COCO se borra).

2.2.1. Configuración utilizada

La configuración que utilizaremos para resolver el problema planteado se puede ver en la figura 4. Podemos ver en el Device Initialization del Codewarrior que seleccionamos como fuente de reloj a Bus clock como preescalador 8 (cuanto más lento funciona es mejor el resultado de la conversión con respecto a los errores internos) y en modo de conversión elegimos: Continuous conversion, además de seleccionar 8 bits para el formato de la conversión. Añadimos un pin para configurar el canal a utilizar, elegimos el el pin PTA0 y habilitamos ese terminal como analógico. Por último seleccionamos el canal 0.

2.3. Relación entre la señal PWM y el conversor ADC

El ciclo de trabajo (D) varía linealmente entre 0 y 1 en una señal PWM. Teniendo en cuenta que nuestra tensión de referencia es de 3.3 y el ADC configurado a 10 bits, la relación que existe entre el valor leído por el ADC es:

$$V_{in} = (ADCR * 3,3)/1024 \quad (1)$$

$$D = V_{in}/3,3 \quad (2)$$

Device Initialization - Init_ADC

Component Parameters

Name	Value	Details
Component name	ADC1	
Device	ADC	ADC
Settings		
Clock settings		
Input clock select	BusClk	
External ref. clock for periph	Disabled	
Prescaler	8	
Long sample time	yes	
Frequency	1000 kHz	
ADC timing details		
Conversion time	Short	
Conversion time	37.00 us	27.027 kHz
Conversion mode	Continuous conversion	
Result data format	8-bit right	
Low power mode	Disabled	
Conversion trigger	Software	
Hardware trigger select	RTC	
Compare Function		
Compare function	Disabled	
Compare type	less than	
Compare value	0	D
Internal bandgap buffer	Disabled	
Pins		
ADC Input Pins	1	
Input Pin0		
Pin	PTA0_PIA0_TPM1CH0_ADP0_ACM...	PTA0_PIA0_TPM1CH0_ADP0_ACM...
Pin Signal		
Pin I/O control disable	yes	
Interrupts/DMA		
Initialization		
ADC type	ADC12V1 or ADC10V1 or ADC8V1	
Initial channel select	Channel 0	

Figura 4: Configuración del ADC.

Sabiendo que el ciclo de trabajo para high true es:

$$D = TPMCxV/TPMMOD + 1 \quad (3)$$

Reemplazando V_{in} , luego igualando el ciclo de trabajo y por último despejando el registro de módulo podemos obtener la fórmula que vincula al PWM con el conversor ADC:

$$D = ((ADCR * 3,3)/1024)/3,3 = TPMCxV/TPMMOD + 1 \quad (4)$$

$$TPMCxV = (ADCR * (TPMMOD + 1))/1024 \quad (5)$$

Si se configuran ambos periféricos con 8 bits de precisión, podemos observar que existen 256 ciclos de trabajos distintos que se pueden representar. Despejando de la ecuación (2) V_{in} podemos ver que: $V_{in} = D * 3,3$. La resolución en mV será el cambio de la diferencia de potencial al variar el D en un incremento mínimo esto quiere decir $1/256=0.004$ entonces $V_{dif} = 0,004 * 3,3 = 0,013 = 13mV$. Por lo tanto la resolución del PWM es de 13 mV mientras que la resolución del ADC es $3,3/1024 = 0,0032mV$.

3. Validación

Para verificar la solución de este problema analizaremos si se generan correctamente las señales de PWM en base a la tensión de entrada, analizamos los ciclos en los que está en alto con respecto al periodo. Cómo configuramos el módulo del TPM1 en 255 dicho de otro modo con una resolución de 8 bits como también el módulo ADC no es necesario realizar una conversión a la hora de asignar el valor de la conversión de la tensión a un valor binario de 8 bits que se lo asigna al canal 1 de TPM1.

La figura 5 muestra para un ciclo de trabajo de 25 % se logra la cantidad de ciclos en alto requerido para generar esa onda PWM con un margen de error de 2 ciclos ya que deberían ser 64 ciclos. La figura 6 muestra para un ciclo de trabajo del 50 % existe un margen de error un poco mayor pero todavía es aceptable para el problema que estamos solucionando qué es aumentar o disminuir la intensidad de iluminación de un LED. Por último en la figura 7 se muestra para un ciclo de trabajo del 75 % se tiene un error de 1 ciclo ya que debería ser 192 ciclos en alto pero es un error despreciable.

4. Interpretación del problema 2

Se pide diseñar un sistema que mantenga la temperatura de una habitación entre 17 y 24 grados utilizando el microcontrolador. Si el sistema detecta que la temperatura excede los 24 grados el sistema debe activar un ventilador, si es menor a 17 grados debe activar un calefactor, mientras que si se encuentra entre 17 y 24 debe apagar ambos actuadores. Se debe informar a través de un display LCD la temperatura actual.

4.1. Detalle del problema a resolver

Para sensar la temperatura se usa un sensor lineal LM35 el cual entrega 10mV/C. Dado que el sistema trabaja a la temperatura ambiente, el voltaje proporcionado por el sensor es amplificado por un factor de 10, de manera que para temperaturas entre 0 y 33 C, el sensor entrega una tensión entre

Para un ciclo de trabajo del 25 %

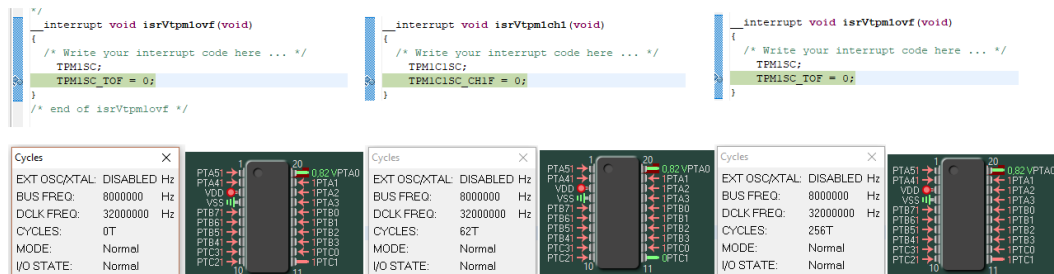


Figura 5: Generación de señal PWM con un ciclo de trabajo del 25 %.

Para un ciclo de trabajo del 50 %

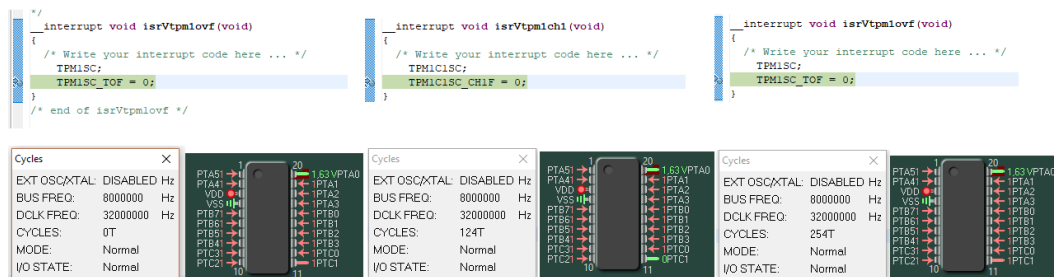


Figura 6: Generación de señal PWM con un ciclo de trabajo del 50 %.

0 y 3.3V. Se utiliza el módulo ADC para medir la tensión entregada por el sensor, como entrada se utilizará el puerto PTA0 configurado de la misma forma que el ejercicio anterior pero cambiando a modo de 10 bits y sabiendo que tenemos una tensión de referencia de 3.3V.

Luego de obtener el valor de temperatura actual se decidirá si activar el ventilador (conectado al PTC0), el calefactor (conectado al PTC1) o no activar ninguno. Cada 0.5 segundos se tiene que actualizar un LCD que muestra la temperatura actual, con el siguiente formato: "TEMP: 25.7 C".

El diagrama de conexión de lo descrito anteriormente se puede ver en la figura 8.

Además nos piden resolver el problema utilizando variables enteras y luego usando variables flotantes para comparar la exactitud de las medidas y la cantidad de recursos consumidos por el

Para un ciclo de trabajo del 75 %

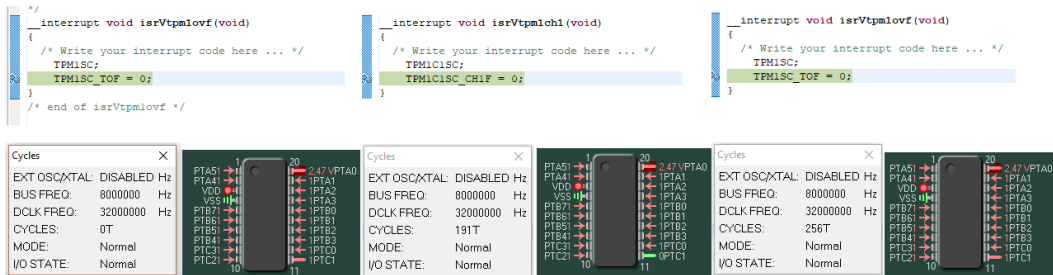


Figura 7: Generación de señal PWM con un ciclo de trabajo del 75 %.

programa (Memoria RAM , ROM y cantidad de ciclos de CPU).

5. Implementación problema 2

Para resolver este problema utilizaremos una arquitectura del tipo Background/Foreground por lo cual las tareas se ejecutan siempre en el super loop en forma de ciclo en función de los eventos asociados a las interrupciones. Para este problema con este tipo de arquitectura es suficiente para poder resolver el problema.

Lo primero que se realiza es configurar el ADC de la misma manera que lo hicimos para el ejercicio 1 salvo que en el Device Initialization elegimos el modo de 10 bits, se configura además el RTC para que genere una interrupción cada 0.5 segundos el cual permite actualizar la temperatura en el LCD (este será implementado con el módulo Terminal el cual permite presentar en el formato especificado la salida deseada) la última configuración es la de los puertos PTC0 y PTC1 como puertos de salida e inicializados en cero.

En el funcionamiento normal del programa se realiza la conversión de la temperatura, luego se identifica dentro de qué rango está para activar los correspondientes actuadores y se visualiza a través del módulo Terminal, con el formato especificado la temperatura actual.

Podemos ver el pseudocódigo del programa principal:

```

1  Función main()
2      Se inicializan los modulos del microcontrolador
3      Configurar como salida los pines PTC0 y PTC1
4      Inicializo en bajo las salidas de los pines PTC0 y PTC1
5      Repetir indefinidamente
6          Si se activo el flag de COCO
7              Leo el valor de la conversión
8              Convierto el valor en byte a grados celsius (para luego
                mostrarlo en el LCD)
9              Si la temperatura es mayor a 24 grados
10                 Se pone en alto el pin PTC1

```

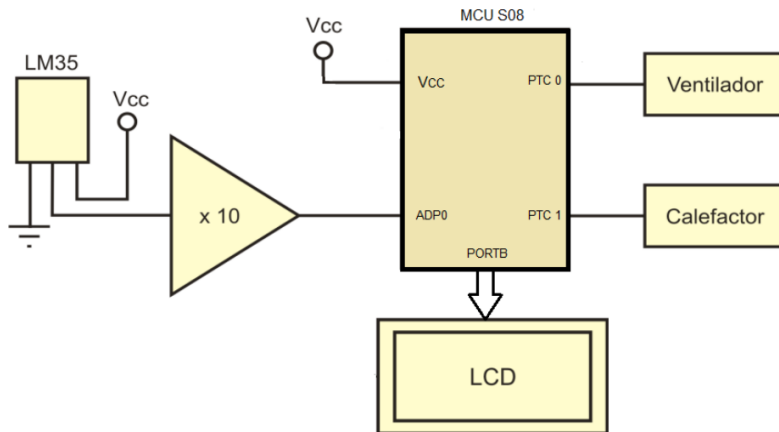


Figura 8: Diagrama de conexión entre el microcontrolador y los demás elementos.

11	Se pone en bajo el pin PTC0
12	Si la temperatura es menor a 17 grados
13	Se pone en alto el pin PTC0
14	Se pone en bajo el pin PTC1
15	Sino
16	Se pone en bajo el pin PTC0
17	Se pone en bajo el pin PTC1
18	Fin main()

Pseudocódigo del programa principal.

En la figura 9 se puede ver cómo se configuró el RTC.

La descripción del módulo ADC se realizó en el ejercicio 1 y su configuración también solo se cambia al modo de funcionamiento de 10 bits.

5.0.1. Ecuaciones para el cálculo de la temperatura en variables enteras y flotantes

Como la tensión de entrada puede valer de 0 a 3.3 V el módulo ADC permite convertir la entrada en bytes (por cómo lo configuramos en 10 bits) para el caso entero utilizó la siguiente fórmula (para el caso de que ADCR no valga cero):

$$temperaturaGrados = ((ADCR * 330) / 1024) + 1 \quad (6)$$

La resolución la podemos hallar de restar dos valores consecutivos del ADCR y reemplazandolos en la fórmula anterior, obteniendo una resolución de 0.32 C. Por lo tanto por cada bit menos significativo que cambia el ADCR el resultado pega saltos de 0.32 C.

Para el caso de variables flotantes se puede utilizar la siguiente fórmula:

$$temperaturaGrados = ((ADCR * 3,3) / 1024) \quad (7)$$

Device Initialization - Init_RTC

Component Parameters

Name	Value	Details
Component name	RTC1	
Device	RTC	RTC
▼ Settings		
▼ Clock settings		
Clock select	1-kHz low power oscillator	
Internal ref. clock for peripherals	Enabled	
External ref. clock for peripherals	Disabled	
Prescaler	100	
RTC modulo value	4	D
Period	500 ms	
▼ Interrupts		
Interrupt	Vrtc	Vrtc
RTC Interrupt	Enabled	
ISR name	isrVrtc	



?
 ☒ Disable Peripheral Initialization
 


Figura 9: Configuración del RTC.

Si calculamos como se dijo anteriormente la resolución para este caso se tiene que por cada bit menos significativo que cambie se pegan saltos de 0.0032 C.

Se puede observar que se tiene la misma resolución, utilizando variables enteras o flotantes.

5.0.2. Comparación de recursos

Para comparar los recursos que utilizan las variables flotantes a comparación de las enteras podemos ver que en ROM las variables flotantes utilizan más ROM (las enteras utilizan 1140 bytes

mientras que las flotantes 1833 bytes) y si analizamos los ciclos necesarios para el cálculo de la temperatura podemos ver que se requieren muchos más ciclos operar con variables flotantes (para variables enteras en 1068 ciclos se realiza la operación mientras para flotantes 6919 ciclos).

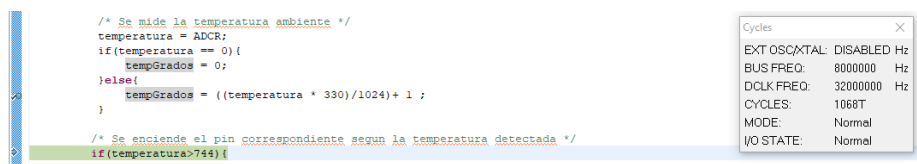
La conclusión que puedo sacar es que, utilizar variables flotantes consume más recursos y para este caso no vale la pena ese gasto de recursos, que con variables enteras se puede resolver de manera aceptable.

```
Summary of section sizes per section type:
READ_ONLY (R):          474 (dec:    1140)
READ_WRITE (R/W):       53 (dec:     83)
NO_INIT (N/I):          6C (dec:    108)
```

Figura 10: Recursos consumidos con variables enteras.

```
Summary of section sizes per section type:
READ_ONLY (R):          729 (dec:    1833)
READ_WRITE (R/W):       53 (dec:     83)
NO_INIT (N/I):          6C (dec:    108)
```

Figura 11: Recursos consumidos con variables flotantes.

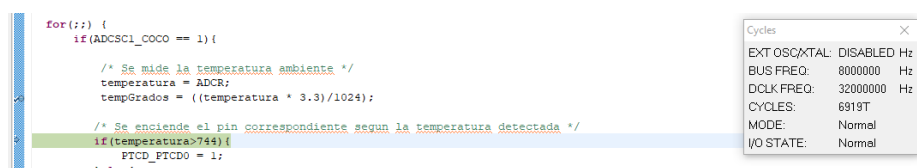


```
/* Se mide la temperatura ambiente */
temperatura = ADCR;
if(temperatura == 0){
    tempGrados = 0;
}else{
    tempGrados = ((temperatura * 330)/1024)+ 1 ;
}

/* Se enciende el pin correspondiente segun la temperatura detectada */
if(temperatura>744){
    PTCD_PTCD0 = 1;
}
```

Cycles window:
EXT OSC/XTAL: DISABLED Hz
BUS FREQ: 8000000 Hz
DCLK FREQ: 32000000 Hz
CYCLES: 1068T
MODE: Normal
I/O STATE: Normal

Figura 12: Cantidad de ciclos con variables enteras.



```
for(;;) {
    if(ADCS1_COCO == 1){
        /* Se mide la temperatura ambiente */
        temperatura = ADCR;
        tempGrados = ((temperatura * 3.3)/1024);

        /* Se enciende el pin correspondiente segun la temperatura detectada */
        if(temperatura>744){
            PTCD_PTCD0 = 1;
        }
    }
}
```

Cycles window:
EXT OSC/XTAL: DISABLED Hz
BUS FREQ: 8000000 Hz
DCLK FREQ: 32000000 Hz
CYCLES: 6919T
MODE: Normal
I/O STATE: Normal

Figura 13: Cantidad de ciclos con variables flotantes.

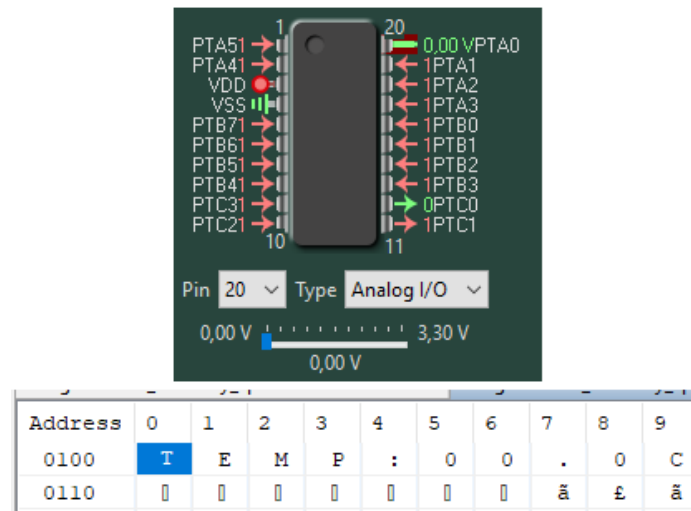
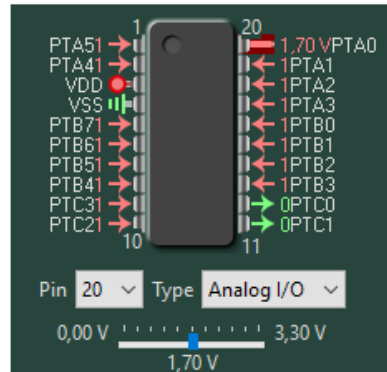


Figura 14: Medición de una temperatura a 0 grados celsius y su procesamiento.

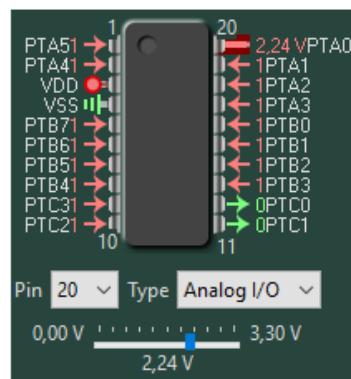
6. Validación

Para validar el funcionamiento del programa desarrollado, analizaremos 5 regiones de funcionamiento. En la figura 14 podemos ver cuando el valor de temperatura es menor a 17 grados celsius comprobando que prende correctamente el calefactor con el pin PTC1 mientras tiene apagado el ventilador, mostrando la temperatura por el LCD en el formato pedido. En la figura 15 y 17 podemos ver los valores límites que son 17 y 24 grados celsius, se tiene apagado tanto el ventilador como al calefactor. Cuando la temperatura es mayor a 24 grados celsius se debe prender el ventilador, utilizando el pin PTC0, esto se puede observar en la figura 16. Por último analizamos un valor intermedio que este entre 17 y 24 que tenga un dígito decimal, para ver si se muestra correctamente en el LCD y eso se muestra en la figura 18.



Address	0	1	2	3	4	5	6	7	8	9
0100	T	E	M	P	:	1	7	.	0	C
0110	0	0	0	0	0	0	:	0	-	0

Figura 15: Medición de una temperatura a 17 grados celsius y su procesamiento.



Address	0	1	2	3	4	5	6	7	8	9
0100	T	E	M	P	:	2	2	.	4	C
0110	0	0	0	0	0	0	:	0	-	0

Figura 16: Medición de una temperatura a 22.4 grados celsius y su procesamiento.

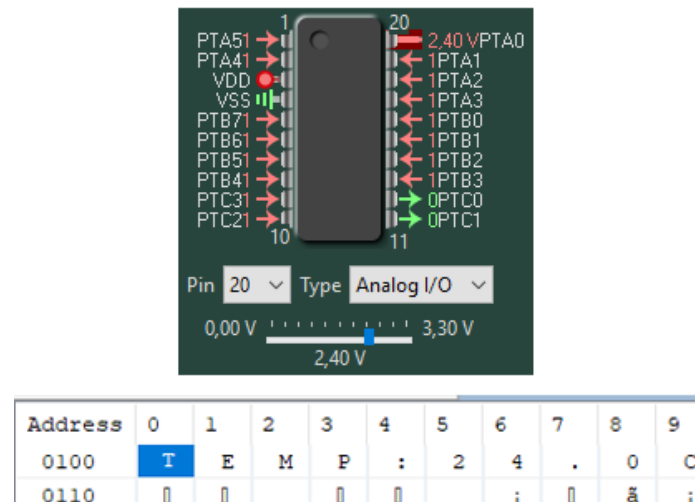


Figura 17: Medición de una temperatura a 24 grados celsius y su procesamiento.

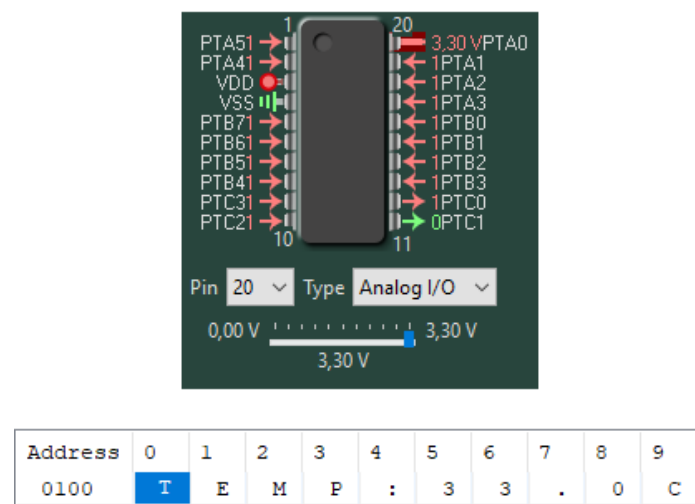


Figura 18: Medición de una temperatura a 33 grados celsius y su procesamiento.

7. Apéndice: Código fuente

Código fuente del problema 1

7.1. main.c

```
1  #include <hidef.h> /* for EnableInterrupts macro */
2  #include "derivative.h" /* include peripheral declarations */
3  #include "types.h"
4
5
6  void MCU_init(void);
7
8  volatile uint16_t tension;
9
10 void main(void) {
11     MCU_init();
12
13     for(;;) {
14         /* Cuando se active el flag de COCO
15          * indica que ya se ha realizado la conversión
16          * (de un valor analogico a un valor digital, que
17          * por como esta configurado es de 1 byte)
18          * establecemos el valor del canal del TPM para
19          * setear el nuevo ciclo de trabajo dependiendo
20          * de la tensión medida.
21          * Como la conversión es de 1 byte y dado que el TPM
22          * se configuro con un modulo de 256 no se requiere calculo
23          * extra para establecer el ciclo de trabajo.
24          */
25         if(ADCSC1.COCO == 1){
26             tension = ADCR;
27             TPM1C1V = tension;
28         }
29     }
30 }
31 }
```

7.2. MCUinit.c

```
1  __interrupt void isrVtpm1ovf(void)
2  {
3      TPM1SC;
4      TPM1SC.TOF = 0;
5  }
6
7  __interrupt void isrVtpm1ch1(void)
8  {
```



```

9      TPM1C1SC;
10     TPM1C1SC.CH1F = 0;
11 }

```

Codigo fuente del problema 2

7.3. main.c (variable enteras)

```

1  #include <hidef.h> /* for EnableInterrupts macro */
2  #include "derivative.h" /* include peripheral declarations */
3  #include "types.h"
4
5  void MCU_init(void); /* Device initialization function declaration */
6
7  volatile uint8_t string[11];
8  volatile uint32_t temperatura;
9  volatile uint32_t tempGrados;
10
11 void main(void) {
12     MCU_init();
13
14     /* Configuro los puertos PTC0 y PTC1 como salida */
15     PTCDD_PTCDD0 = 1;
16     PTCDD_PTCDD1 = 1;
17
18     /* Configuro los puertos PTC0 y PTC1 con valor bajo*/
19     PTCDD_PTCDD0 = 0;
20     PTCDD_PTCDD1 = 0;
21
22     for(;;) {
23
24         if(ADCSC1.COCO == 1){
25
26             /* Se mide la temperatura ambiente */
27             temperatura = ADCR;
28             if(temperatura == 0){
29                 tempGrados = 0;
30             }else{
31                 tempGrados = ((temperatura * 330)/1024)+ 1 ;
32             }
33
34             /* Se enciende el pin correspondiente segun la temperatura detectada */
35             if(temperatura>744){
36                 PTCDD_PTCDD0 = 1;
37                 PTCDD_PTCDD1 = 0;
38             }else{
39                 if(temperatura<527){
40                     PTCDD_PTCDD0 = 0;
41                     PTCDD_PTCDD1 = 1;
42                 }else{
43                     PTCDD_PTCDD0 = 0;

```

```

44                                     PTC_D_PTC_D1 = 0;
45                                     }
46                                 }
47                            }
48                        }
49                    }
50    }

```

7.4. terminal.c (variable enteras)

```

1  #include "types.h"
2
3  extern volatile uint8_t string[11];
4  extern volatile uint32_t tempGrados;
5
6  void temp_get_string(){
7      uint8_t dig1, dig2, dig3;
8
9      dig3 = tempGrados % 10;
10     tempGrados = tempGrados / 10;
11     dig2 = tempGrados % 10;
12     tempGrados = tempGrados / 10;
13     dig1 = tempGrados % 10;
14
15     string[0] = 'T';
16     string[1] = 'E';
17     string[2] = 'M';
18     string[3] = 'P';
19     string[4] = '.';
20     string[5] = dig1+'0';
21     string[6] = dig2+'0';
22     string[7] = '.';
23     string[8] = dig3+'0';
24     string[9] = 'C';
25     string[10] = '\0';
26 }

```

7.5. main.c (variable flotante)

```

1  #include <hidef.h> /* for EnableInterrupts macro */
2  #include "derivative.h" /* include peripheral declarations */
3  #include "types.h"
4
5  #ifdef __cplusplus
6  extern "C"
7  #endif
8  void MCU_init(void); /* Device initialization function declaration */

```

```

9
10 volatile uint8_t string[11];
11 volatile uint32_t temperatura;
12 volatile float tempGrados;
13
14 void main(void) {
15     MCU_init();
16
17     /* Configuro los puertos PTC0 y PTC1 como salida */
18     PTCDD_PTCDD0 = 1;
19     PTCDD_PTCDD1 = 1;
20
21     /* Configuro los puertos PTC0 y PTC1 con valor bajo*/
22     PTCD_PTCD0 = 0;
23     PTCD_PTCD1 = 0;
24
25     for(;;) {
26         if(ADCSC1_COCO == 1){
27
28             /* Se mide la temperatura ambiente */
29             temperatura = ADCR;
30             tempGrados = ((temperatura * 3.3)/1024);
31
32             /* Se enciende el pin correspondiente segun la temperatura detectada */
33             if(temperatura>744){
34                 PTCD_PTCD0 = 1;
35                 PTCD_PTCD1 = 0;
36             }else{
37                 if(temperatura<527){
38                     PTCD_PTCD0 = 0;
39                     PTCD_PTCD1 = 1;
40                 }else{
41                     PTCD_PTCD0 = 0;
42                     PTCD_PTCD1 = 0;
43                 }
44             }
45         }
46     }
47
48 }

```

7.6. terminal.c (variable flotante)

```

1 #include "types.h"
2
3 extern volatile uint8_t string[11];
4 extern volatile float tempGrados;
5
6 void temp_get_string(){
7     uint8_t dig1, dig2, dig3;

```

```

8      float valor = tempGrados*100;
9
10     dig3 = (uint8_t)valor % 10;
11     valor = valor / 10;
12     dig2 = (uint8_t)valor % 10;
13     valor = valor / 10;
14     dig1 = (uint8_t)valor % 10;
15
16     string[0] = 'T';
17     string[1] = 'E';
18     string[2] = 'M';
19     string[3] = 'P';
20     string[4] = '.';
21     string[5] = dig1+'0';
22     string[6] = dig2+'0';
23     string[7] = '.';
24     string[8] = dig3+'0';
25     string[9] = 'C';
26     string[10] = '\0';
27 }

```

Común para el programa de variables enteras y flotantes

7.7. terminal.h

```

1  #ifndef TERMINAL_H_
2  #define TERMINAL_H_
3
4  void temp_get_string(void);
5
6  #endif

```

7.8. types.h

```

1  #ifndef TYPES_H
2  #define TYPES_H
3
4  typedef unsigned char uint8_t;
5  typedef unsigned int uint16_t;
6  typedef unsigned long uint32_t;
7
8  typedef signed char int8_t;
9  typedef signed int int16_t;
10 typedef signed long int32_t;
11
12 #endif

```

7.9. MCUinit.c

```
1  __interrupt void isrVrtc(void)
2  {
3      /* Se ejecuta cada 0.5 segundos */
4      temp_get_string();
5      RTCSC_RTIF = 1;
6  }
```