MASB | Arduino Timers

#stm32 #c #biomedical #microcontroladores #programacion

Albert Álvarez Carulla
hello@thealbert.dev
https://thealbert.dev/

18 de febrero de 2020

Índice

1. *Timers*

Tic, tac, tic, tac, … ¡Aparece el *timer* por la puerta! Es el momento de saltar a otro periférico: los *timers* o contadores. Un periférico cuyo nombre ya nos indica su principal propósito: **medir/operar con el tiempo**.

Un *timer* no es más que un registro que actúa de contador. Este contador incrementa su valor por cada ciclo de reloj o por cada transición de una señal de entrada. Este segundo uso suele ser utilizado para la lectura de *encoders*. Nosotros veremos y nos centraremos en el primer caso, donde el contador incremente su valor por cada ciclo de reloj.

¿Qué veremos con los *timers*? Veremos dos de sus principales usos: ejecutar tareas en función del tiempo y generar una señal PWM.

1.1. Objetivos

- Introducción a punteros en C.
- Generación de una interrupción basada en un *timer* en Arduino.
- Generación y salida de una señal cuadrada en Arduino en *hardware*.
- Generación y salida de una señal PWM en Arduino.

1.2. Procedimiento

Venga, va. Una última vez: clonad el repositorio y cread vuestra rama de desarrollo.

Por cierto, ya **no os diré cuando hacer un *commit* ni que descripción/mensaje usar**. Después de cuatro proyectos, queda a vuestro criterio cuándo hacerlos y qué mensajes poner. **Se evaluará como operáis en git en función del uso que hagáis de él.** Tened en cuenta que una vez finalizada la práctica no se pueden añadir *commits* que no hayáis añadido ya o cambiar los mensajes de los *commits* que hayáis hecho. Así que **sobre el uso de git es materialmente imposible haceros una pre-evaluación para que luego lo modifiquéis**. Tenedlo en cuenta.

1.2.1. *Blink the LED* con *timers*

"~~J***r~~ *Recórcholis*, Albert. ¿Otra vez el *blink the LED*?" Sí. Otra vez el ***blink the LED***, pero esta vez vamos a hacerlo **mediante las interrupciones de los *timers***. De momento solo disponemos del LED como único elemento visual que puede indicarnos la correcta aplicación de los conocimientos de programación de microcontroladores que estamos aprendiendo. ¡No menospreciéis al pobre LED!

Vamos a crear un *sketch* de Arduino con el nombre `masb-p03` en la carpeta Arduino de nuestro repositorio local. Lo primero que vamos a hacer es configurar uno de los *timers* del microcontrolador.

El microcontrolador tiene hasta 11 *timers*. Arduino utiliza alguno de estos *timers* para implementar las funciones `millis` o `delay` que hemos visto en otras prácticas. Así que es posible que si utilizamos uno de esos *timers*, esas funciones dejen de funcionar correctamente. Mejor. Así no las usaremos.

1.2.1.1. Punteros en C

Antes de ver cómo configurar un *timer*, tenemos que ver qué son los punteros en C, puesto que los necesitaremos.

Un puntero es algo que apunta. Hala. Ya sabemos que son los punteros.

Puede sonar simplificado al absurdo (y así es), pero pese a que el uso de punteros puede ser confuso al empezar, lo que hacen no va más allá de lo que dice su nombre: apuntar.

Un puntero no es más que una variable que guarda la dirección de memoria de otra variable. Veámoslo rápidamente con un ejemplo en un archivo de programación inventado.

```
1 uint8_t VariableNum = 0;
```

Imaginaros que tenemos una variable númerica tipo `uint8_t` que hemos llamado `VariableNum` y la hemos inicializado a 0. Nada raro de momento. En el microcontrolador, esta variable se guardará en la memoria. Más específicamente, se guardará en una posición de esa memoria. Esa posición viene identificada por una dirección.

Si hiciéramos una tabla de la memoria del microcontrolador tendríamos:

Dirección	Valor
…	…
0x003A 58E0	0x3A
0x003A 58E1	**0x00**
0x003A 58E2	0x21
…	…

El valor de nuestra variable quedaría almacenado en la memoria en la posición `0x003A 58E1`.

> La notación `0x` indica que el valor está escrito en hexadecimal. La dirección indicada es totalmengte inventada para el ejemplo. El compilador será el encargado de escoger en que posición de memoria se guardan las variables.

Ahora vamos a crear un puntero:

```
uint8_t *punteroVariableNum;
uint8_t VariableNum = 0;

punteroVariableNum = &VariableNum;
```

Veamos qué hemos hecho. En la primera línea del código hemos creado un puntero. **El tipo que se indica en la declaración de un puntero no indica el tipo del puntero en sí, sino el tipo de variable a la que apunta el puntero.** Un puntero siempre tendrá el tipo/tamaño del bus de direcciones del microcontrolador. En este caso, el microcontrolador STM32 es de 32 bits, por lo que todos los punteros serán de 32 bits. Repito, el tipo que aparece en la declaración del puntero es el tipo de la variable a la que apunta. **Los punteros se indican con un asterisco delante del nombre.**

En la segunda línea, declaramos e inicializamos a 0 una variable de 8 bits sin signo llamada `VariableNum`. Aquí, nada nuevo.

En la tercera línea, **hemos utilizado el carácter & delante de la variable** `VariableNum`. De este modo, **extraemos**, no el valor de `VariableNum`, sino **su puntero o posición de memoria**. Esta posición de memoria lo guardamos en el puntero que hemos guardado.

Llegados aquí, la sinceridad debe de imperar: 1) los punteros son confusos la primera vez que se ven/usan, 2) los punteros son fáciles una vez se entiende lo que son/hacen. (+info)

1.2.1.2. Inicialicemos el *timer*

Vamos a nuestro archivo `masb-p03`. Allí ingresamos el siguiente código:

```
void setup() {
  // put your setup code here, to run once:

  // creamos un puntero a la configuracion del timer 3 en memoria
  HardwareTimer *MyTim = new HardwareTimer(TIM3);

  MyTim->setMode(2, TIMER_OUTPUT_COMPARE); // configuramos el modo del timer
  MyTim->setOverflow(1000000, MICROSEC_FORMAT); // configuramos el periodo del timer a 1 segundo
  MyTim->attachInterrupt(cambiarEstadoLED); // indicamos el nombre de la ISR a ejecutar
  MyTim->resume(); // iniciar el timer
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Lo que hemos hecho en el código es primeramente crear un puntero a la posición de memoria donde se encuentra la configuración del *timer* 3. Hemos cogido el 3 como podíamos haber cogido cualquiera de los otros 10.

El puntero a `MyTim` apunta a un tipo de variable `HardwareTimer` que es una estructura. **Una estructura no es más que una variable que agrupa variables de diferentes tipos.** En este caso, funciones.

> De hecho, el nombre de una función no es más que un puntero que lleva a la posición de memoria donde se encuentra el código de esa función.

Para acceder a los diferentes elementos en una estructura, utilizamos el operador `->`. Para configurar el *timer*, simplemente seguimos las indicaciones de la librería de Arduino:

- Configuramos el modo del *timer*.
- Configuramos el periodo del *timer*. Una vez el *timer* cuente hasta el periodo indicado, se reinicia el contador y vuelve a empezar.
- Configuramos el nombre la ISR que queremos que se ejecute cuando el contador llegue al periodo previamente configurado.
- Arrancamos el *timer* para que empiece a contar.

1.2.1.3. Hagamos que parpadee el LED

Vamos a utilizar la aproximación que ya hemos hecho en prácticas anteriores: utilizar una variable booleana para controlar el LED y conmutar el valor de esa variable en la interrupción. El código quedaría del siguiente modo:

```
1  #define LED     13
2
3  bool estadoLED = false;
4
5  void setup() {
6    // put your setup code here, to run once:
7
8    // creamos un puntero a la configuracion del timer 3 en memoria
9    HardwareTimer *MyTim = new HardwareTimer(TIM3);
10
11   MyTim->setMode(2, TIMER_OUTPUT_COMPARE); // configuramos el modo del
       timer sin salida por ningun pin
12   MyTim->setOverflow(1000000, MICROSEC_FORMAT); // configuramos el
       periodo del timer a 1 segundo
13   MyTim->attachInterrupt(cambiarEstadoLED); // indicamos el nombre de
       la ISR a ejecutar
14   MyTim->resume(); // iniciar el timer
15
16   pinMode(LED, OUTPUT); // pin del LED de salida
17   digitalWrite(LED, estadoLED); // apagado por defecto
18
```

```
19 }
20
21 void loop() {
22   // put your main code here, to run repeatedly:
23
24   digitalWrite(LED, estadoLED);
25
26 }
27
28 // ISR del timer
29 void cambiarEstadoLED(void) {
30
31   estadoLED = !estadoLED; // conmutamos el LED
32
33 }
```

Con esto ya tendríamos el LED parpadeando mediante interrupciones del *timer*. A diferencia de la práctica anterior (donde utilizabamos la función `millis`), esta aproximación ofrece una periodicidad mucho más exacta. El uso de `millis` para programar la conmutación, hacía que pudiera darse el caso que se demore la conmutación del LED si se demora la ejecución del código hasta que `millis` es consultado. En este caso no. Cada 1 segundo exacto salta la interrupción y el LED conmuta.

En principio, debemos de tener el programa corriendo sin problemas con el LED parpadeando cada 1 segundo. Pasamos al siguiente uso del *timer* donde ni siquiera utilizaremos la función `digitalWrite` para conmutar el LED. Lo haremos por *hardware* puro, sin *software*.

1.2.2. *Blink the LED like a Greek God*

Ahora vamos a hacer parpadear el LED de una manera muy especial: sin utilizar *software*. **La conmutación del LED la hará el periférico *timer* por su cuenta y la CPU no estará ejecutando ningún código para ello.** Esto hace que la CPU quede totalmente liberada para realizar otras tareas.

¿Cómo lo hacemos? Un *timer*, además del contador en sí, tiene disponible **canales**. Estos canales, que **pueden ser redirigidos hacia un pin del microcontrolador** para darles salida, se pueden configurar para realizar una cierta acción en función del *timer*.

Por ejemplo, podemos configurar un valor de comparación y un periodo para el *timer*. El *timer* incrementará su valor hasta llegar al periodo configurado y reiniciará su cuenta. Pues uno de los canales del *timer* se puede configurar para que sea nivel bajo cuando el contador tenga un valor inferior al valor de comparación configurado y sea nivel alto cuando el contador tenga un valor superior. De este modo, podemos generar una señal periódica rectangular cuyos tiempos en nivel bajo y alto son configurables con el valor de comparación del *timer*. Hemos hecho un PWM.

Otra opción es la que utilizaremos a continuación, no configurar un valor de comparación; simplemente,

un periodo. Y establecemos el **modo del *timer* en `toggle`** de tal modo que **un canal**, que será el que podamos redirigir al pin, **conmute cada vez que el *timer* llegue hasta el periodo configurado**.

Vamos a verlo en el código.

```
#define LED     13

void setup() {
  // put your setup code here, to run once:

  // creamos un puntero a la configuracion del timer 2 en memoria
  HardwareTimer *MyTim = new HardwareTimer(TIM2);

  MyTim->setMode(1, TIMER_OUTPUT_COMPARE_TOGGLE, LED); // modo conmutacion del canal 1 del timer 2
  MyTim->setOverflow(1000000, MICROSEC_FORMAT); // configuramos el periodo del timer a 1 segundo
  MyTim->resume(); // iniciar el timer
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Primeramente, hemos creado un puntero al *timer* 2. En este caso, **el *timer* a escoger sí es vital**. Si vamos a la hoja de especificaciones del microcontrolador (página 40, Tabla 8), el pin `PA5` del LED tiene como función alternativa `TIM2_CH1`, entre otras. Esto nos indica que este pin puede dar salida al canal 1 del *timer* 2. Así que debemos de utilizar el *timer* 2 para operar con el LED.

Seguidamente, fijamos el modo. De todos los modos que nos posibilita configurar la librería de Arduino, escogemos el modo `TIMER_OUTPUT_COMPARE_TOGGLE`, que es el que hará que nuestro LED cambie de estado cuando el *timer* complete una cuenta hasta el periodo configurado. En esa misma instrucción también debemos de indicar el canal que estamos configurando y el pin por el que saldrá el canal.

> Hay microcontroladores/periféricos que ofrecen más de un pin para un mismo canal para dar flexibilidad.

A continuación, fijamos el periodo del *timer* a 1 segundo y, por último, arrancamos el *timer*.

Compilamos/cargamos el programa y tenemos nuestro LED parpadeando cada 1 segundo. Fijaos en una cosa: ¿cuanto código tenemos en la función `loop`? ¡Ninguno! Actualmente, la CPU de nuestro microcontrolador no está haciendo absolutamente nada y es el *timer* quien se encarga de realizar la conmutación del LED.

Parece que tenemos algo funcionando. Vamos al siguiente apartado donde veremos cómo hacer variar la intensidad del LED con un PWM.

1.2.3. Regular la intensidad del LED

Podemos controlar la intensidad de un LED utilizando una señal PWM. Una señal PWM no es más que una señal periódica rectangular cuyo ratio entre el tiempo que la señal esta a nivel alto y el periodo de la señal, llamado *duty cycle* o ciclo de trabajo, es configurable. **El LED se iluminará más con un PWM con un *duty* elevado y se iluminará menos con un *duty* bajo.**

Vamos a dar una intensidad fija al LED con un PWM. Puesto que para ello configuraremos un canal de un *timer*, utilizaremos el *timer* que tiene acceso al pin del LED: el *timer* 2.

```
#define LED     13

void setup() {
  // put your setup code here, to run once:

  // creamos un puntero a la configuracion del timer 2 en memoria
  HardwareTimer *MyTim = new HardwareTimer(TIM2);

  // pwm de al 15% a 200Hz
  MyTim->setPWM(1, 13, 200, 10);
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Lo que hemos hecho ha sido crear nuestro puntero a la configuración del *timer*. Nada nuevo. Y después utilizamos una función que nos da Arduino para configurar un PWM automáticamente: `setPWM`. De la documentación de la librería, vemos que a esta función hay que hacerle llegar los siguientes parámetros: el canal, el pin, la frecuencia del PWM en hercios y el *duty* en tanto por ciento.

Probad el programa varias veces modificando el *duty* y veréis como podemos variar la intensidad del LED. Las variaciones de intensidad son fácilmente perceptibles para valores de entre 0 y 25 % para el *duty*.

1.2.3.1. Latidos en el LED

Es un poco … “aburrido” ir cambiando el *duty* a mano para ver que funciona. Vamos a variar el *duty* en el código. Además, lo haremos utilizando una variación sinusoide. De este modo podremos hacer un `#include` de una librería matemática a modo demostración.

Para generar una señal sinusoide necesitamos una amplitud, una frecuencia de oscilación y tiempo. Además, la señal sinusoide deberá de tener un *offset* para generar solo valores positivos. También eliminaremos el desfase. La fórmula sería la siguiente.

$$y(t) = A\sin(2\pi ft) + A$$

Figura 1: Señal sinusoide.

La amplitud deberá de ir de 0 a 25 para hacer perceptible el cambio de intensidad. La frecuencia podemos configurarla entre 200 Hz a 1 kHz. Nos faltaría el tiempo… ¿Se os ocurre algo? Eeeexacto. Vamos a usar la función `millis`.

La función `sin` para computar el seno de un valor forma parte de una librería llamada `math.h`. Incorporaremos la librería con un `#include`.

El código quedaría del siguiente modo.

```
#include <math.h>

#define LED     13

// constantes que definen nuestra senal sinusoidal
const double  pi = 3.14,      // constante pi
              amplitud = 25,  // amplitud de oscilacion
              periodo = 2;    // periodo de oscilacion

double duty = 0;

// creamos un puntero a la configuracion del timer 2 en memoria
// lo declaramos global para que este disponible en la funcion loop
HardwareTimer *MyTim = new HardwareTimer(TIM2);

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

  // calculamos el duty del pwm a partir de una senal sinusodal
  duty = amplitud*sin(2*pi/periodo*millis()/1000) + amplitud;

  // aplicamos el duty al pwm
  MyTim->setPWM(1, LED, 200, duty);
}
```

Probamos y tenemos nuestro LED parpadeando suavemente. ¡Madre mía las locuras que podremos hacer el año que viene con las luces de Navidad de nuestro árbol!

¡Tenemos un código que va perfecto!

1.3. Reto

Vamos a hacer los mismo que en la práctica anterior: conmutar el modo del LED de apagado a parpadeando y viceversa con el pulsador. Esta vez, todo mediante interrupciones. O lo que es lo mismo, **no puede haber nada de código en la función `loop`**.

Como veis, un mismo reto se puede afrontar de diversas maneras. De ello, podemos deducir que no existe una única manera o una manera correcta de realizar una misma tarea. Hay maneras que son más o menos óptimas en función del escenario en el que estemos y, como ingenieros, deberemos escoger la opción más óptima.

1.4. Evaluación

1.4.1. Entregables

Estos son los elementos que deberán de estar disponibles para el profesorado de cara a vuestra evaluación.

- ☐ **Commits**

Después de realizar cuatro proyecto en dos prácticas, en esta práctica empezamos a dejar a vuestro criterio cuándo realizar los *commits*. No hay número mínimo/máximo y se evaluará el uso adecuado del control de versiones (creación de ramas, *commits* en el momento adecuado, mensajes descriptivos de los cambios, …).

- ☐ **Reto**
- ☐ **Informe**

Informe con el mismo formato/indicaciones que en prácticas anteriores. Guardad el informe con el nombre `REPORT.md` en la misma carpeta que este documento.

El informe debe de contener:

- ☐ Qué son los punteros en C y cómo se utilizan.
- ☐ Tabla con las nuevas funciones de Arduino vistas en la práctica así como una explicación de qué hacen y cómo se utilizan.

1.4.2. Pull Request

Finalizados todos los entregables, acordaos de hacer el *push* pertinente y cread un *Pull Request* (PR) de vuestra rama a la master. Acordaos de ponerme como *Reviewer*.

1.4.3. Rúbrica

La rúbrica que utilizaremos para la evaluación la podéis encontrar en el CampusVirtual. Os recomendamos que le echéis un vistazo para que sepáis exactamente qué se evaluará y qué se os pide.

1.5. Conclusiones

En esta práctica hemos introducido un nuevo periférico: el *timer*. Este periférico nos servirá para todas esas acciones que estén relacionadas con una medición del tiempo.

Hemos visto cómo implementar una interrupción que salte cada vez que el *timer* haga una cuenta hasta un valor dado. También hemos visto que los *timers* tienen canales que podemos utilizar para generar señales rectangulares periódicas. Estos canales nos permiten realizar conmutaciones en pines de salida sin que participe la CPU. También nos permiten generar señales PWM de gran utilidad (controlar la intensidad de LEDs, controlar motores, controlar convertidores DC/DC, …).

También hemos visto en C el uso de punteros, estructuras e incluir librerías.

En la siguiente práctica implementaremos los *timers* a nivel de registros con las HAL.