MASB | STM32
Timers

#stm32 #c #biomedical #microcontroladores #programacion

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

18 de febrero de 2020

Índice

1. *Timers*

Vistos los *timers* en Arduino, ahora vamos a verlos con STM32CubeMX. No hay nada nuevo que introducir ya que **ya sabemos qué es un *timer***, **cómo funciona** y cuáles son **algunos de sus usos**. Ahora vamos a ver cómo hacerlo a nivel de registro. Una vez más, nos toparemos con un pequeño salto de complejidad (**que no dificultad, recordad**) que nos ofrecerá una mayor flexibilidad a la hora de utilizar los *timers* de nuestro microcontrolador.

También aprovecharemos para echar un **primer vistazo a herramientas profesionales de depuración** que no tenemos en Arduino y que nos ayudarán en nuestro día a día a lograr que nuestros programas funcionen correctamente. ¡Al ataque!

1.1. Objetivos

- Generación de una interrupción basada en un *timer* a nivel de registros.
- Generación y salida de una señal cuadrada a nivel de registros en *hardware*.
- Generación y salida de una señal PWM a nivel de registros.
- Depuración de un programa en STM32CubeIDE.

1.2. Procedimiento

De verdad. Ahora super en serio: **la última vez** que pego el chivatazo. Vamos al repositorio local donde trabajaremos, **saltamos a la rama `master`**, **importamos posibles cambios** que hayan podido haber en el repositorio remoto y **creamos/saltamos a una rama nueva** para esta práctica. Acordaros de ir haciendo los ***commits* pertinentes cuando creáis oportuno** utilizando un **mensaje descriptivo** de los cambios realizados.

Abrimos STM32CubeIDE configurando el *workspace* en la carpeta `stm32cube` de nuestro repositorio local y creamos un proyecto llamado `masb-p03`.

> Acordaos de inicializar los periféricos a su estado por defecto al crear el proyecto.

1.2.1. *Blink the LED* con interrupciones del *timer*

1.2.1.1. Configuración del *timer* Vamos a configurar el microcontrolador. Primeramente, configuramos el **pin `PA5` como `GPIO_Output` y con *label* `LED`**. Seguidamente, vamos a `Timers` en el menu lateral de la izquierda y, del desplegable, clicamos el ***timer* `TIM3`**. Se nos abrirá el formulario de configuración. **Para habilitar el *timer*, solo debemos de escoger una señal de reloj que lo controle**. En el campo `Clock Source` **escogemos `Internal Clock`**.

Al seleccionar la fuente del reloj, se nos mostrará abajo un formulario de configuración. Allí podemos toquetear cualquier cosa del *timer*: su **modo de contar** (ascendente, descendente, centrado,…), su **periodo** (el *timer* cuenta hasta llegar al periodo y se reinicia), su **preescalado** de frecuencia (a lo mejor la frecuencia del reloj interno usado no nos sirve y tampoco podemos modificarlo porque influye en otros periféricos, pues podemos preescalar la frecuencia del *clock* para obtener una frecuencia distinta), etc. Hay mil configuraciones distintas y si queréis ver todas ellas, ¿sabéis dónde tenemos que ir? Eeeexacto. A la documentación. (Ya os lo busco yo, va. Página 316.)

Para un *timer* con un periodo de 1 segundo, en el formulario, configuraremos un **preescalado de 8,399** en el campo `Prescaler` y un **periodo 10,000** en el campo `Counter Period`. ¿Porque esos valores? Empecemos por ir a la **pestaña `Clock Configuration`** en la parte superior de STM32CubeMX. Iremos a un formulario donde se muestra todo el **sistema de reloj del microcontrolador** que podemos configurar. No tocaremos nada, pero nos fijaremos (os lo he resaltado en rojo en la siguiente imagen) en que el reloj que va hacia los *timers*, el `APB2`, tiene una frecuencia de 84 MHz.

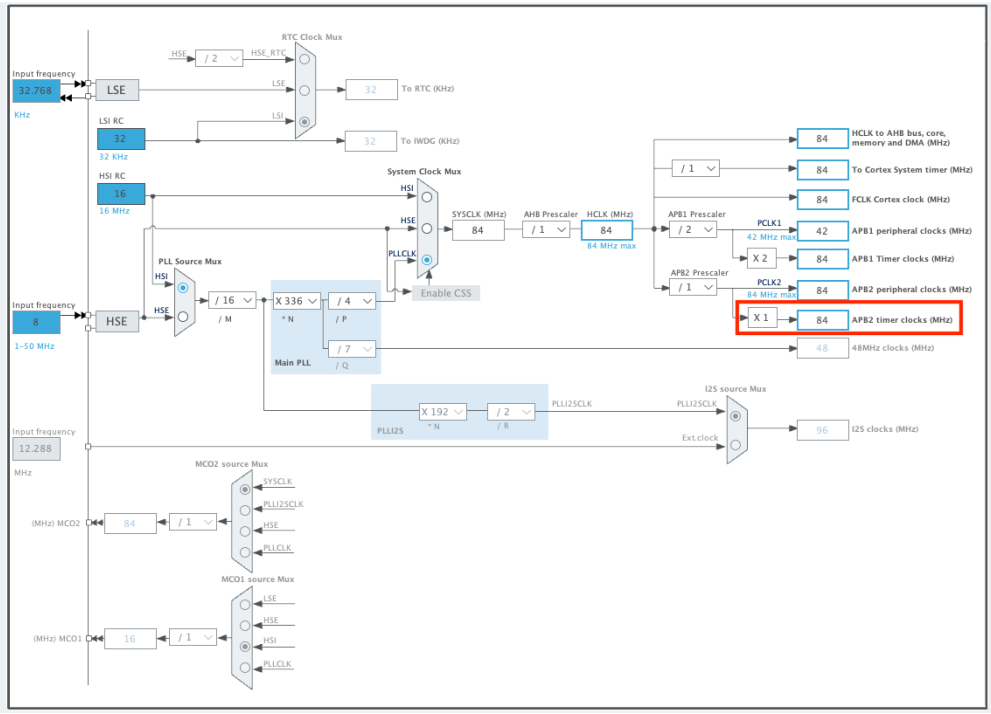

Figura 1: Configuración del clock.

Si queremos hacer que el *timer* tenga un periodo de 1 segundo, lo primero que uno haría es fijar un `Counter Period` de 84,000,000. El cálculo viene de:

$$1\,\mathrm{s}\,\frac{84\times10^{6}\,\mathrm{cycles}}{1\,\mathrm{s}}=84,000,000\,\mathrm{ticks}$$

Figura 2: Segundos a ciclos.

Si el *timer* opera con una frecuencia de 84 MHz, tenemos que dejar que el *timer* cuente hasta 84,000,000 para que pase 1 segundo. La cosa es que si ponemos 84,000,000 en el campo `Counter Period`... No nos deja. Y es que solo podemos poner un número de **máximo 16 bits** (lo que equivale a como máximo 65,535). ¿Alternativa? Bajar la frecuencia del *timer*. Para no tocar el reloj `APB2`, utilizaremos el preescalado. Mirando la documentación (página 368), vemos que la fórmula para preescalar la frecuencia es fCK*PSC / (PSC[15:0] + 1). Para hacer que la frecuencia pase de 84 MHz a, por ejemplo,* ***10 kHz****, fijaremos el preescalado a 8,399. Ahora sí. Aplicando el cálculo anterior, necesitamos un* ***periodo de 10,000*** *para lograr tener un _timer* con un periodo de 1 segundo.

$$1\,\mathrm{s}\,\frac{10\times10^{3}\,\mathrm{cycles}}{1\,\mathrm{s}}=10,000\,\mathrm{ticks}$$

Figura 3: Segundos a ciclos.

Configurado el periodo, vamos a la **pestaña `NVIC Settings`** del mismo formulario de configuración y **habilitamos la interrupción `TIM3 global interrupt`**.

Con esto, tenemos el microcontrolador totalmente configurado. Guardamos el archivo de configuración y generamos el código.

1.2.1.2. Implementación del *callback*

Vamos al `main.c` y creamos una **variable global llamada `estadoLED`** que inicializaremos a `FALSE`. También crearemos las macros pertinentes para `TRUE` y `FALSE`. Luego, dentro del **`while`**`(1)` de la función `main`, escribiremos:

```
...

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
     HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, estadoLED);
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */

...
```

Finalmente, lo que haremos será, en la ISR del *timer*, hacer que conmute el valor de la variable `estadoLED`. Lo implementaremos en el ***callback* pertinente** que podemos encontrar en la documentación de las HAL.

```
...

/* USER CODE BEGIN 4 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {

    estadoLED = !estadoLED;

}
/* USER CODE END 4 */

...
```

Compilamos y depuramos el programa y… **No parpadea nada.** Vamos a ver qué ocurre utilizando las herramientas de STM32CubeIDE para depurar código.

1.2.1.3. Los *breakpoints*

Lo que vamos a hacer es iniciar la depuración del programa . Como ya sabemos, siempre que iniciamos la depuración, el IDE pausa la ejecución del programa en la primera instrucción de la función `main`. Eso es un *breakpoint*. Un ***breakpoint*** no es más que un **punto en el programa configurado para que, cuando la ejecución del programa pase por ese punto, la ejecución del programa se pause**. El *breakpoint* inicial siempre viene por defecto, pero nosotros podemos configurar los que queramos dónde queramos.

Para el caso que nos ocupa, suele ser muy útil colocar un ***breakpoint* en el *callback***. Si la interrupción está bien implementada, al colocar un *breakpoint* allí, **cuando la interrupción salte, el programa debería de pausarse allí**. Si no se pausa, querrá decir que el *callback* no se está ejecutando y algo falla.

Para **añadir un *breakpoint***, solo debemos de hacer **doble clic en el número línea línia de código** donde queramos añadir el *breakpoint*. Saldrá una bolita al lado del número de línea indicando que allí se ha configurado un *breakpoint*.

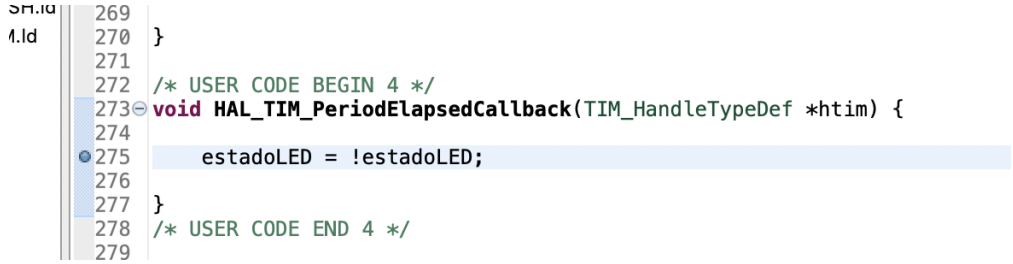

Figura 4: Breakpoint.

Una vez configurado el *breakpoint*, pulsamos el icono para iniciar/reanudar la ejecución del programa  . Vemos que el programa no se pausa en ese *breakpoint* por lo que no se está ejecutando la interrupción.

Otra comprobación puede ser ver si el *timer* está funcionando. Vamos a ver si el contador del *timer* está contando. Pulsamos el icono para pausar a mano la ejecución del código  . Luego, vamos arriba a la derecha, a la **pestaña SFRs** (*Special Function Registers*). Esta ventana muestra el valor de los registros del microcontrolador.

> Es **imprescindible** pausar la ejecución del código para que el microcontrolador pueda actualizar los valores al IDE.

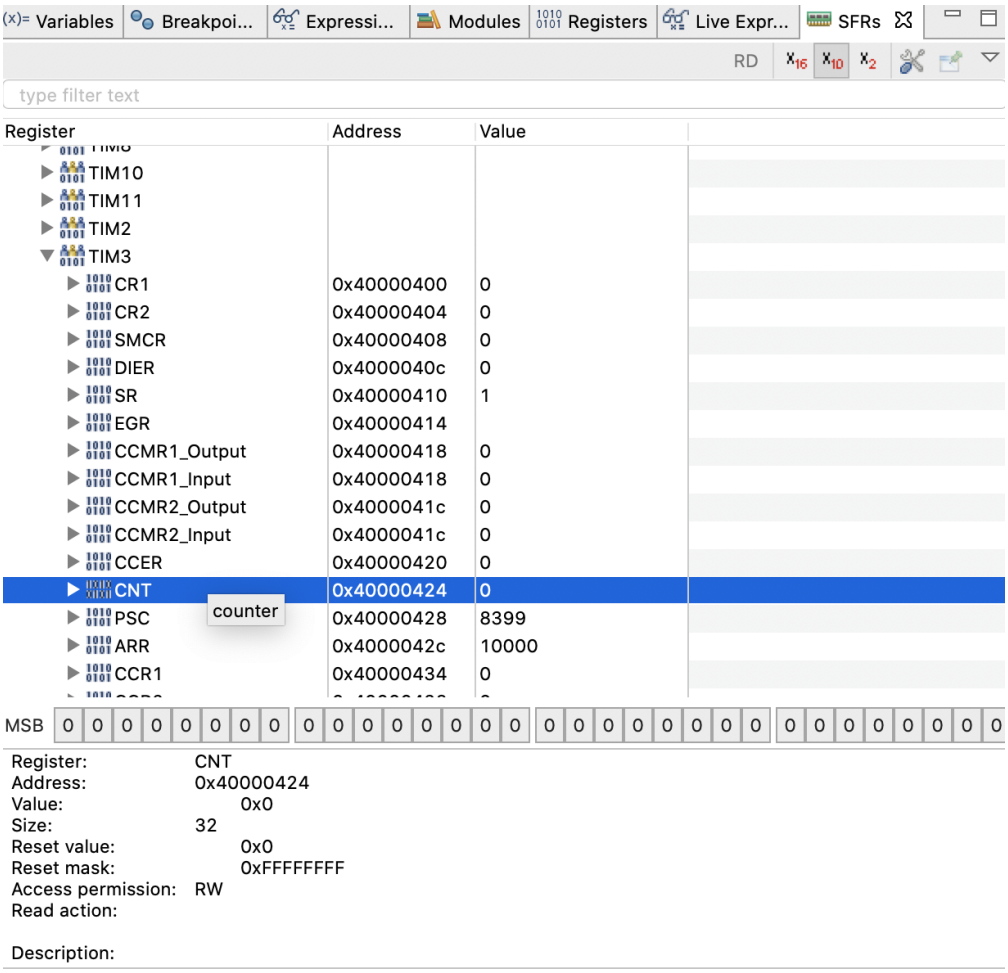

Figura 5: Registros.

Si comprobamos el registro `CNT` (el contador, como podemos comprobar en la documentación (página

353)) del *timer* 3 pausando y reanudando a mano el programa varias veces, vemos que siempre es 0. El *timer* no está funcionando.

> Para comprobar si el *timer* está funcionando, hay que reanudar y pausar el programa ya que **mientras el programa está en pausa nada funciona. Todo está pausado.**

Si miramos un poco la documentación de las HAL, vemos que hay que utilizar la función `HAL_TIM_Base_Start_IT` para iniciar el funcionamiento del *timer* con las interrupciones habilitadas. Añadimos esa función al `main`:

```
...

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM3_Init();
  /* USER CODE BEGIN 2 */

  HAL_TIM_Base_Start_IT(&htim3);

  /* USER CODE END 2 */

...
```

A la función hay que pasarle el puntero de la variable `htim3` con la configuración del *timer*. Esa variable con la configuración nos la crea STM32CubeMX por nosotros.

Ejecutamos el código y, **si no hemos quitado el *breakpoint* anterior**, el código se nos pausa dentro del *callback* indicando que este se ejecuta. También podemos comprobar como el LED no parpadea puesto que el programa está en pausa.

> Para deshabilitar un *breakpoint*, volvemos a hacer doble clic sobre el número de línea.

Si quitamos el *breakpoint*, podemos pausar/reanudar a mano la ejecución del código para ver como ahora sí está variando el valor del registro `CNT` del *timer* 3.

Ya tenemos nuestro LED parpadeando mediante interrupciones. Vamos a ver cómo hacerlo mediante *hardware*.

1.2.2. *Blink the LED* con solo *hardware*

Como hicimos con Arduino, vamos a configurar el *timer* para que, mediante uno de sus canales, se encargue el solo de hacer parpadear el LED.

Abrimos el archivo `.ioc` para configurar el *timer*. El *timer* con un canal en el pin `PA5` es el 2. Así que vamos, primeramente, a **deshabilitar el *timer* 3** previamente configurado. Esto lo hacemos yendo al formulario de configuración del *timer* 3 y seleccionando `Disable` como `Clock Source`.

Luego, vamos a **habilitar el *timer* 2** yendo a su formulario de configuración y escogiendo `Internal Clock` como `Clock Source`. Seguidamente configuramos el preescalado y el periodo. Este *timer* tiene un contador de **32 bits** en lugar de los 16 bits del *timer* 3. Por ello, podemos dejar el **preescalado a 0** y utilizar directamente un **periodo de 84,000,000** en el campo `Counter Period`. En este caso, **no hace falta habilitar interrupciones puesto que lo haremos todo con *hardware***.

Lo que vamos a hacer es **configurar el canal 1**. En el campo `Channel1` escogemos `Output Compare CH1`. Más abajo, en el formulario de configuración del *timer*, nos habrá aparecido un apartado para configurar el canal 1. Establecemos como modo `Toggle on match` en el campo `Mode`. En este modo, la salida del canal conmutará cada vez que el contador sea igual al valor configurado en el campo `Pulse`.

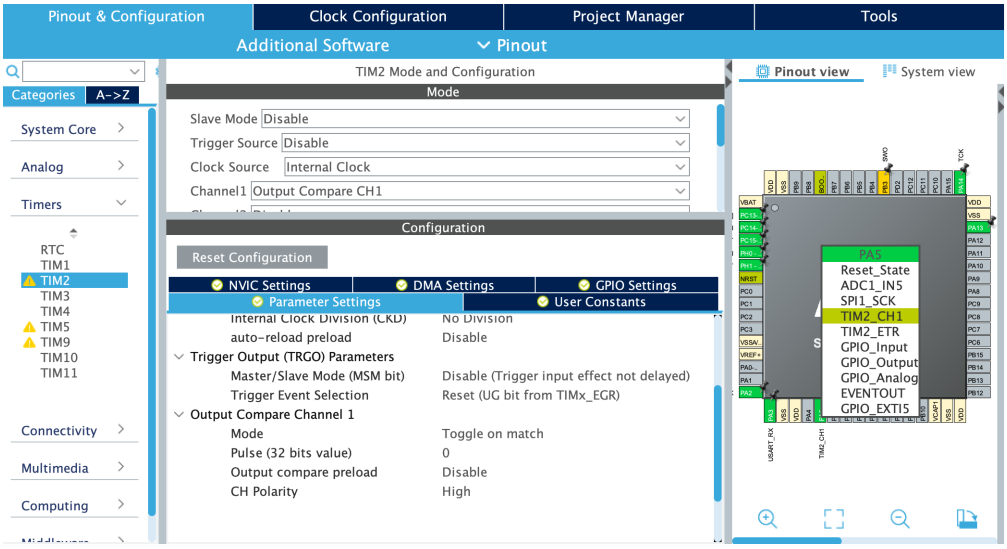

Figura 6: Configuración del canal 1 del timer.

Finalmente, en la pestaña `GPIO Settings`, STM32CubeMX, con toda su buena fe, nos ha configurado el pin `PA0` como el pin de salida del canal 1 puesto que el pin `PA5` lo tenemos configurado como `GPIO_Output`. **Esto no lo queremos.** Vamos al `Pinout view` y seleccionamos la función del pin `PA5` como `TIM2_CH1`. El pin `PA0` dejará de dar salida al canal 1 y la salida pasará a ser pin `PA5`.

Guardamos el archivo de configuración y generamos código.

Vamos al `main.c` y lo **limpiamos**:

- Eliminamos la instrucción `HAL_GPIO_WritePin` del *loop* **`while`**`(1)` de la función `main`.

- Eliminamos el *callback* `HAL_TIM_PeriodElapsedCallback`.
- Eliminamos la variable `estadoLED`.
- Eliminamos las macros `TRUE` y `FALSE`.
- Sustituimos `HAL_TIM_Base_Start_IT(&htim3);` por `HAL_TIM_OC_Start(&htim2, TIM_CHANNEL_1);`. Hemos cambiado de función (iniciamos el canal del *timer* 2). **La función no sale de la nada.** Recordad que estas funciones hay que buscarlas en la documentación de las HAL.

Compilamos, probamos el programa y el LED debería de estar parpadeando perfectamente sin tener ningún trozo de código en el *loop* `while(1)` que se encargue de ello.

1.2.3. Regular la intensidad del LED

En este último apartado, vamos a regular la intensidad del LED con un PWM. Vamos a empezar configurando el *timer* 2. **Abrimos el archivo `.ioc`** y nos dirigimos a la **configuración del *timer* 2**. En el formulario de configuración, vamos a empezar editando el campo `Counter Period` para configurar un **periodo para el *timer* de tal modo que su inversa (la frecuencia) sea de 200 Hz**. Esto lo logramos con un `Counter Period` de 420,000.

$$\frac{1\,\mathrm{s}}{200\,\mathrm{ticks}}\frac{84\times10^{6}\,\mathrm{ticks}}{1\,\mathrm{s}}=420{,}000$$

Figura 7: Conversión frecuencia.

Seguidamente, cambiamos el modo del canal 1 y **escogemos `PWM Generation CH1`**. En la configuración del PWM, abajo del todo, indicamos **10,000 en el campo `Pulse`**. Cuando el contador del *timer* esté **por debajo de 10,000, la salida del LED estará en nivel alto**; y cuando el valor del *timer* sea **mayor de 10,000, la salida del LED estará en nivel bajo**. Es la manera de fijar el *duty cycle*. Con un periodo de 420,000 y un `Pulse` de 10,000, tenemos un ***duty cycle* de 2.4 %**. Guardamos la configuración y generamos el código. **Sustituimos** la función `HAL_TIM_OC_Start(&htim2, TIM_CHANNEL_1);` por `HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);`, compilamos y depuramos el programa y vemos como **el LED brilla con una intensidad leve**. Podemos editar en la configuración del microcontrolador el valor del campo `Pulse` para ver cómo varía la intensidad del LED (os recomiendo hacerlo), pero vamos a hacerlo con una señal sinusoide como en el caso del Arduino.

Como en Arduino, vamos al `main.c` e incluimos la librería estandard `math.h`.

```
1 ...
2
3 /* Private includes
     ----------------------------------------------------------*/
```

```
4  /* USER CODE BEGIN Includes */
5  #include <math.h>
6  /* USER CODE END Includes */
7
8  ...
```

Creamos las constantes `pi`, `amplitud` y `periodo`, y la variable `duty`. Puesto que el valor del **`Pulse` es un entero en valor absoluto y no un porcentaje**, **creamos la constante** `TIM2Period` para almacenar el periodo del *timer*, y la **variable** `varPulse` para calcular el valor de `Pulse` en función del *duty*. Puesto que no deben de estar accesibles en ninguna función más allá de la `main`, **no hay que declararlas como constantes/variables globales**.

```
1  ...
2
3    /* USER CODE BEGIN 1 */
4      const double  pi = 3.14,      // constante pi
5                    amplitud = 25,  // amplitud de oscilacion
6                    periodo = 2;    // periodo de oscilacion
7      const uint32_t TIM2Period = 420000; // periodo de TIM2
8
9      double duty = 0; // duty cycle
10     uint32_t varPulse = 0; // valor para el Pulse
11   /* USER CODE END 1 */
12
13 ...
```

En el bucle **`while`**`(1)` de la función `main`, **calculamos el *duty* con la función `sin`**, calculamos el **valor para `Pulse` en función del *duty*** calculado y **utilizamos la macro** `__HAL_TIM_SET_COMPARE` de las HAL para modificar el valor de `Pulse` del *timer* 2.

```
1  ...
2
3    /* Infinite loop */
4    /* USER CODE BEGIN WHILE */
5    while (1)
6    {
7        duty = amplitud*sin(2*pi/periodo*HAL_GetTick()/1000) + amplitud;
8        varPulse = TIM2Period * duty/100;
9        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, varPulse);
10     /* USER CODE END WHILE */
11
12     /* USER CODE BEGIN 3 */
13   }
14   /* USER CODE END 3 */
15
16 ...
```

Guardamos, compilamos, cargamos el programa y tenemos nuestro LED parpadeando **¡con estilazo!**

1.3. Reto

El reto será conmutar el estado del LED entre apagado y encendido con el pulsador `B1` mediante interrupciones del pulsador y el *timer*. **No puede haber código en el bucle `while(1)` de la función `main`.**

Venga, va. Que seguramente ya no os haga hacer más este reto.

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**

Se os deja 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 *breakpoints*.
- ☐ Tabla con las nuevas funciones de las HAL 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

Con la finalización de esta práctica, podemos decir que **sabemos controlar a nivel de registros** aquellas **acciones más importantes desempeñadas por un *timer***. Esas acciones van desde la **ejecución periódica de tareas mediante interrupciones**, hasta la **generación de ondas rectangulares periódicas sin la participación de la CPU**, pasando por la modulación de un tren de pulsos para **generar un PWM**.

También hemos visto **cómo se utilizan los *breakspoints*** y **cómo ver los registros del microcontrolador** dentro del abanico de herramientas profesionales de depuración que ofrece STM32CubeIDE en comparación con Arduino.

En la siguiente práctica pasaremos a ver el **ADC** para poder leer/convertir **señales analógicas** con nuestro microcontrolador.