MASB | STM32 Interrupciones

#stm32 #c #biomedical #microcontroladores #programacion

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

10 de febrero de 2020

Índice

1. Interrupciones

En esta segunda parte de la segunda práctica, vamos a **migrar el código de Arduino a STM32CubeIDE**. Esta vez, haremos que el **LED parpadee sin utilizar la función `HAL_Delay`** y veremos cómo **implementar una interrupción para un GPIO** con la herramienta gráfica STM32CubeMX. La gran flexibilidad que ofrece la programación a nivel de registros hace que implementar una interrupción sea más complicado que en Arduino. **Más complicado no quiere decir difícil**, como veremos a continuación. Además, también veremos que esa mayor libertad nos posibilita un mayor número de opciones a la hora de desarrollar nuestras aplicaciones.

1.1. Objetivos

- Implementación de interrupciones en pines digitales con STM32CubeIDE.
- Alternativas a implementaciones basadas en la función `HAL_Delay`.

1.2. Procedimiento

1.2.1. Crear rama de desarrollo

Empezamos creando nuestra rama de desarrollo. Para ello, tal y como hicimos en la segunda parte de la primera práctica, **vamos primero a la rama máster**, sincronizamos la rama para **importar posibles cambios** que hayan podido haber en el repositorio remoto y, finalmente, **creamos y saltamos a una nueva rama de desarrollo** que llamaremos `develop/B-<tu-nombre-sin-espacios-ni-caracteres-raros>`. A continuación, os dejo los comandos por última vez.

```
git checkout master
git pull
git checkout -b develop/B-<tu-nombre-sin-espacios-ni-caracteres-raros>
```

1.2.2. Configuración del microcontrolador

Iniciamos STM32CubeIDE, establecemos como *workspace* la carpeta `stm32cube` de nuestro repositorio local y **creamos un nuevo proyecto** llamado `masb-p02`. Ya sabéis cómo hacerlo.

Vamos a configurar el pin `PA5` como un GPIO de salida y con el *label* `LED`. Guardad el archivo de **configuración del microcontrolador** y **generad el código**.

1.2.3. "*Blink the LED, HAL*"

Ya sabemos cuál será la estrategia de implementación puesto que ya la hemos utilizado en Arduino: en lugar de implementar *delays* entre conmutaciones del LED, vamos a **programar esas conmutaciones**.

Creamos las tres variables que utilizaremos: `periodMillis`, `currentMillis` y `previousMillis`. Las declaramos como variables globales. ¡Siempre fijaros de **añadir código allí donde los comentarios del generador de código os indica**!

```
...

/* USER CODE BEGIN PV */
const uint32_t periodMillis = 1000;  // periodo entre conmutaciones del LED
uint32_t currentMillis = 0;  // ms actuales
uint32_t previousMillis = 0;  // ms transcurridos en la conmutacion anterior
/* USER CODE END PV */

...
```

Seguidamente, implementamos la programación de la conmutación del LED dentro del *while loop* de la función `main`. Utilizaremos la función `HAL_GetTick`, que es el equivalente de `millis` en Arduino.

> Aunque os ponga las funciones HAL a utilizar, no perdáis la oportunidad de buscarla en la documentación sobre las HAL del fabricante. El día de mañana en vuestra vida profesional y, sobre todo, durante el proyecto, no se os indicarán las funciones HAL a utilizar y deberéis de saber encontrarlas por vosotros mismos. **¡No desaprovechéis la oportunidad de practicar la búsqueda a través del documento!**

El código quedaría de la siguiente manera. Veréis "algo" que os chirriará en el siguiente código. Hacedlo como se indica y luego ya veremos el qué.

```
...

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
      currentMillis = HAL_GetTick();  // ms actuales

      if (currentMillis - previousMillis >= periodMillis) { // si ha pasado el periodo deseado

```

```
12            previousMillis = currentMillis; // guardamos los ms
                  transcurridos desde el inicio del programa
13
14            // y conmutamos el LED
15            HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
16
17          }
18        /* USER CODE BEGIN 3 */
19      }
20      /* USER CODE END 3 */
21
22  ...
```

Por si alguno no confía, lo que os debe de chirriar es que **hemos puesto código en un sitio no permitido**. Lo hemos hecho adrede para, más adelante, ver sus consecuencias y cómo solucionarlas.

Compilamos el código y depuramos. Si todo está correcto, no deberíamos de haber obtenido ningún error de compilación y el LED debe de estar parpadeando cada 1 segundo.

1.2.3.1. *Scope* de las variables

Antes de pasar a las interrupciones con el pulsador, veamos un aspecto ligado al *scope* de las variables que hemos creado. Ahora mismo tenemos todas las variables declaradas como variables globales. Esto es, están declaradas fuera de cualquier función. Esto hace que estén disponibles para cualquier función del archivo `main.c`. La pregunta es: **¿es necesario que las variables estén disponibles para todas las funciones?**

Ya os lo respondo yo: **no**. Todas las variables que tenemos solo se utilizan dentro de la función `main`. **Intentaremos evitar dar a las variables un *scope* mayor al necesario. O lo que es lo mismo, intentaremos evitar declarar variables como globales si no necesitamos que lo sean.**

En este caso, declararemos las variables dentro de la función `main`: eliminamos las declaraciones de las variables donde están ahora mismo y las **movemos al principio de la función `main`**. Quedaría de la siguiente manera.

```
1  ...
2
3    /* USER CODE BEGIN 2 */
4
5    const uint32_t periodMillis = 1000;  // periodo entre conmutaciones
         del LED
6    uint32_t currentMillis = 0;  // ms actuales
7    uint32_t previousMillis = 0;  // ms transcurridos en la conmutacion
         anterior
8
9    /* USER CODE END 2 */
10
11
12
```

```
13    /* Infinite loop */
14    /* USER CODE BEGIN WHILE */
15    while (1)
16    {
17
18  ...
```

Volvemos a compilar y a comprobar que todo funciona correctamente.

Importante, tenemos código funcionando. Guardamos una versión de este código en Git.

1.2.4. Interrupciones con STM32CubeMX

Vamos a crear una interrupción para el pulsador de la EVB de tal modo que se conmute el estado del LED cada vez que lo pulsemos. Primeramente, **abrimos STM32CubeMX** haciendo doble clic al archivo `.ioc` de nuestro proyecto. Seguidamente, indicamos `PULSADOR` como *label* para el pin `PC13`. Y ahora, como función del pin, **no escogemos `GPIO_Input`** como en la práctica anterior, **sino `GPIO_EXTI13`**.

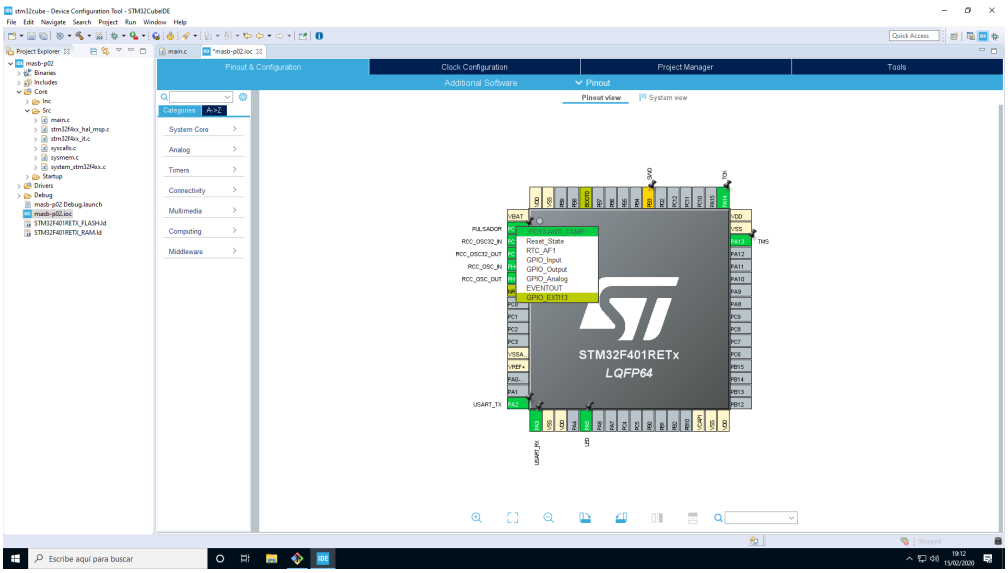

Figura 1: Selección del modo del pin PC13.

Seguidamente, vamos a habilitar la interrupción del pin. Esto lo hacemos yendo a `System view`, en la parte superior de la imagen interactiva del microcontrolador, y entrando en `NVIC` en la columna `System Core`. En esta ventana tenemos disponible todas las interrupciones disponibles según la actual configuración del microcontrolador.

> Solo se muestran las interrupciones disponibles para una configuración dada. No aparecen interrupciones de periféricos no habilitados/utilizados. Si los habilitásemos, aparecerían interrupciones nuevas.

Marcamos la casilla `Enabled` de la interrupción `EXTI line[15:10] interrupts`. **Esta interrupción es común a los pines del 10 al 15**. Hacemos un punto y aparte para ver cómo funcionan las interrupciones de los GPIOs: EXTI.

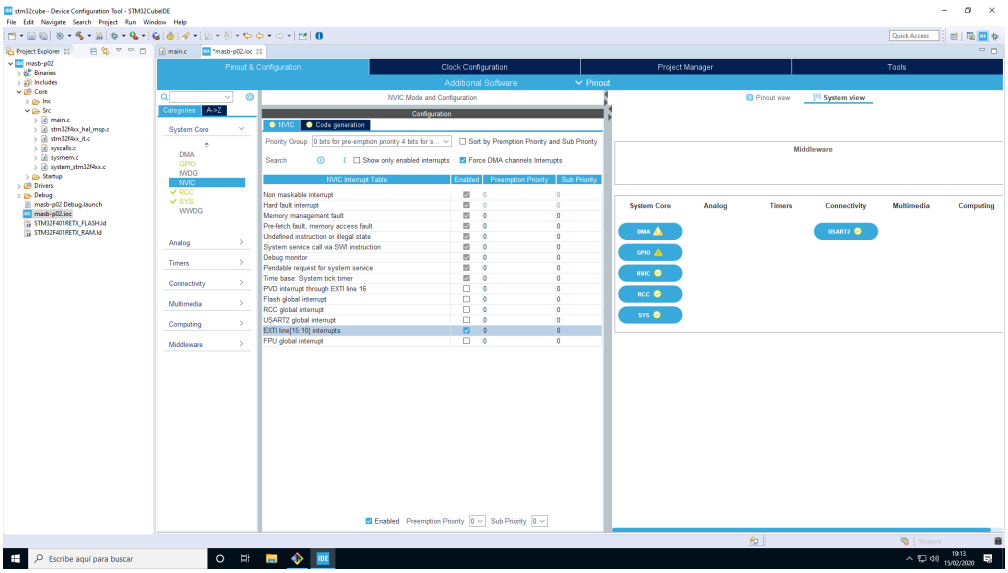

Figura 2: Habilitar interrupciones.

1.2.4.1. Interrupciones EXTI La documentación del microcontrolador nos describe cómo opera el módulo de interrupciones para los GPIOs: EXTI. Por si queréis (deberíais) echarle un vistazo, la descripción del módulo la tenéis en la página 202 del manual de referencia.

El módulo genera hasta 23 señales de interrupción o IRQs: `EXTI[0:22]`. **Todos los pines de un mismo número, indiferentemente del puerto, comparten el mismo IRQ**. Por ejemplo, todos los pines 0 (`PA0`, `PB0`, `PC0`, etc.) comparten un mismo IRQ: `EXTI0`. También operan así el resto de IRQs del módulo EXTI. Conlusión: **no podemos operar una interrupción en dos pines distintos si estos comparten numeración**. Ejemplos de combinación de interrupciones **sin problemas**: `PA1` y `PB4`, `PC4` y `PC3`, o `PA2` y `PD1`. Ejemplos de combinación de interrupciones **con problemas**: `PA1` y `PB1`, `PC5` y `PD5`, o `PA7` y `PE7`. Podemos habilitar interrupciones en 1 o más pines, pero no debe de haber más de una interrupción para un número de pin dado, indiferentemente del puerto.

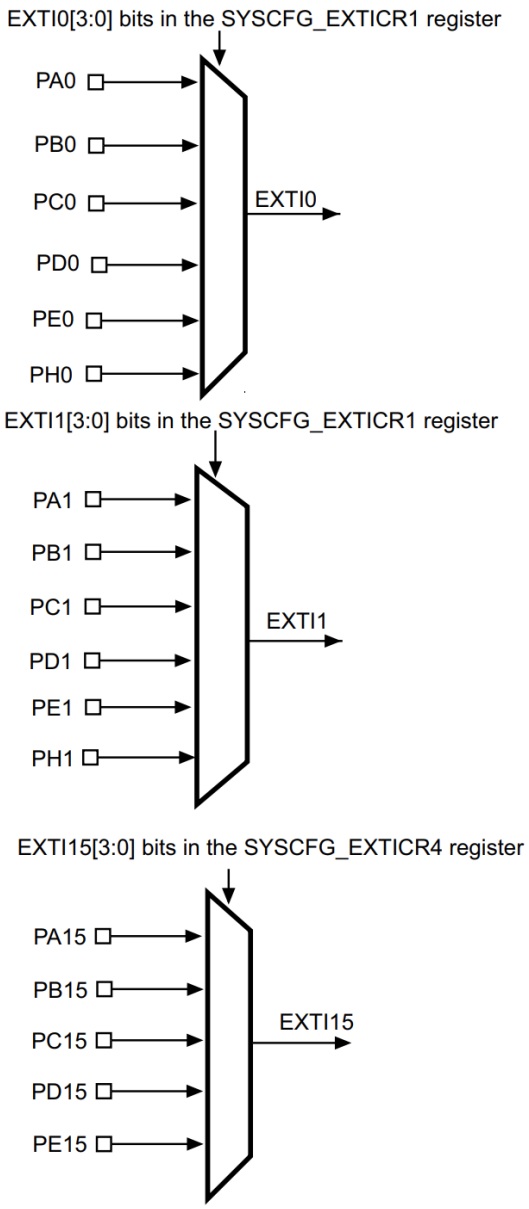

Figura 3: Módulo EXTI.

> Imagen de STMicroelectronics.

Por último, podemos echar un vistazo a la Tabla 38 del documento donde aparecen los vectores de interrupción. Vemos que `EXTI0`, `EXTI1` y `EXTI2`, por ejemplo, tienen su propio vector de interrupción. En cambio, **las señales `EXTI` de la 10 a la 15 comparten vector de interrupción**. Esto quiere decir que en los primeros casos tendremos una ISR para un solo pin, mientras que en el segundo caso habrá una ISR para seis pines. Será **tarea nuestra**, dentro de esta última ISR común a seis pines, **discernir**

que IRQ/EXTI ha generado la interrupción en el código del programa y actuar en consecuencia.

1.2.4.2. Generar código y recuperar versión

Bien. Siguiendo donde lo habíamos dejado, guardamos la configuración del microcontrolador y generamos el código. Vamos al archivo `main.c` y… tacháááán! Se nos fué el código porque lo pusimos en el sitio que no toca. Vamos a ver rápidamente cómo recuperarlo.

Hay mil y un escenarios en que puede darse que queramos deshacer algo: hemos modificado algo y queremos revertirlo a la versión de justo el último *commit*, ya hemos hecho un *commit* y queremos deshacerlo, queremos deshacer cambios en muchos archivos, queremos deshacer cambios en solo un archivo, … Esto hay que tenerlo en cuenta porque ahora veremos **como gestionar uno de esos infinitos escenarios: reestablecer la versión del último *commit* de un solo archivo** (sin haber hecho ningún *commit* de esos cambios a descartar). Este es nuestro escenario actual. Hemos hecho un *commit*, hemos regenerado el código, se nos ha modificado el archivo `main.c` y queremos descartar los cambios y dejarlo tal y como estaba en el último *commit* (el que hemos hecho antes de la regeneración del código).

Pues el comando es sencillo:

```
1 git checkout -- stm32cube/masb-p02/Core/Src/main.c
```

El comando a utilizar es `git checkout`. Como veis es un comando muy versátil. Simplemente debemos de indicar, junto al comando, el nombre del archivo cuya versión queremos reestablecer. Los dos guiones `--`son opcionales y sirven para evitar ambigüedades. Una ambigüedad del tipo “hay una rama que se llama justo como el archivo que queremos reestablecer”. Pocas veces se da el caso, pero si añadís los guiones por sistema os evitaréis futuros problemas. También podríamos indicar, en lugar de un archivo, todos los archivos del repositorio local mediante un punto `.`.

```
1 git checkout -- .
```

Pero es mucho mejor solo restablecer aquello que necesitamos. Si en nuestro caso restableciésemos todo, perderíamos la configuración de microcontrolador `.ioc` que acabamos de crear y tendríamos que hacerlo de nuevo.

Después ejecutar el comando `git checkout`, vamos a `main.c` y vemos que el archivo se nos ha restaurado a la última versión “*commiteada*”. Corregimos el error y movemos el código allí donde toca. Volvemos a regenerar el código con STM32CubeMX para que nos añada las modificaciones pertinentes al `main.c` y ¡*solved*!

¡Haced un *commit* para que pueda ver que habéis seguido estos pasos!

Esto era una demostración para ver cómo recuperar una versión. En realidad, no necesitamos el código del `main` puesto que ahora haremos el apagado/encendido del LED con el pulsador.

1.2.4.3. En busca del *Callback*... Empezamos por ver que hay que implementar en la función `main`. Como en Arduino, simplemente crearemos una variable booleana `estadoLED` que utilizaremos para controlar el LED mediante la función `HAL_GPIO_WritePin`.

> Acordaros de crear las macros para `TRUE` y `FALSE`.

Posteriormente, en una interrupción del pulsador, invertiremos el valor de la variable booleana. Esta será la pinta del código dentro de la función `main` del archivo `main.c`.

> Archivo: `main.c`

```
...

  /* USER CODE BEGIN 2 */
  _Bool estadoLED = FALSE;
  /* USER CODE END 2 */



  /* 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 */

...
```

Ahora toca ver cómo/dónde implementa las interrupciones STM32CubeMX... Agarraros a la silla... Y tened el dedo preparado sobre el `F3`...

> En la documentación de las HAL aparecen directamente las funciones o *callbacks* que hay que utilizar para cada interrupción y módulo. Aquí haremos el ejercicio de buscar manualmente esos *callbacks*, pero, en un futuro, iremos directamente a la documentación a buscarlas.
>
> Un *callback* no es más que una función llamada desde una ISR.

Vamos a empezar yendo al archivo `stm32f4xx_it.c` que tenéis en la misma carpeta que el archivo `main.c`. En este archivo, justo abajo del todo, tenemos la función `EXTI15_10_IRQHandler`. **Los comentarios del código son los que nos indican que es la función que buscamos.**

> Archivo: `stm32f4xx_it.c`

```
...

/*
    ******************************************************************************
    */
/* STM32F4xx Peripheral Interrupt Handlers                                    */
/* Add here the Interrupt Handlers for the used peripherals.                  */
/* For the available peripheral interrupt handler names,                      */
/* please refer to the startup file (startup_stm32f4xx.s).                    */
/*
    ******************************************************************************
    */

/**
  * @brief This function handles EXTI line[15:10] interrupts.
  */
void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */

  /* USER CODE END EXTI15_10_IRQn 1 */
}

...
```

Esta función se ejecuta cuando se da una interrupción en alguno de los pines numerados del 10 al 15 y tengan, obviamente, las interrupciones habilitadas. Dentro tenemos la función `HAL_GPIO_EXTI_IRQHandler`. Si ponemos el cursor encima y pulsamos `F3`, vemos el contenido de esta función.

> Archivo: `stm32f4xx_hal_gpio.c`

```
...

/**
  * @brief  This function handles EXTI interrupt request.
  * @param  GPIO_Pin Specifies the pins connected EXTI line
  * @retval None
  */
```

```
 8  void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
 9  {
10    /* EXTI line interrupt detected */
11    if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)
12    {
13      __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
14      HAL_GPIO_EXTI_Callback(GPIO_Pin);
15    }
16  }
17
18  ...
```

Allí ya encontramos la función `HAL_GPIO_EXTI_Callback`. Si ponemos una vez más el cursor encima y hacemos `F3`, nos lleva a su contenido (que está justo debajo). Está función `HAL_GPIO_EXTI_Callback` no hace nada y tiene **el calificador `__weak`**. Este calificador **nos permite escribir otra función en otro lugar con el mismo nombre y que sobreescriba a esta**. Está es la función que implementaremos para nuestra interrupción.

> Archivo: `stm32f4xx_hal_gpio.c`

```
 1  ...
 2
 3  /**
 4    * @brief  EXTI line detection callbacks.
 5    * @param  GPIO_Pin Specifies the pins connected EXTI line
 6    * @retval None
 7    */
 8  __weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
 9  {
10    /* Prevent unused argument(s) compilation warning */
11    UNUSED(GPIO_Pin);
12    /* NOTE: This function Should not be modified, when the callback is
         needed,
13             the HAL_GPIO_EXTI_Callback could be implemented in the user
                file
14     */
15  }
16
17  ...
```

"¡Pero bueeeno! ¡Pero si estamos saltando de función en función sin ver nada!" Lo se… Y es que realmente… ¡No hay nada que ver! STM32CubeMX nos genera una serie de funciones que se llaman unas a otras y ya está. Uno podría decir: "¿Y porque no llamamos directamente a `HAL_GPIO_EXTI_Callback` ?" STM32CubeMX nos hace esta estrategia de llamar varias funciones para darle un mayor nivel de abstracción al código. (¿Os acordáis que era el nivel de abstracción de un código?) Es un tema avanzado que cuesta ver sin experiencia previa en la programación de microcontroladores y por lo tanto no entraremos a verlo. Pero sí que debemos de reservar 2 segundos de nuestro tiempo para darle a

STM32CubeMX las gracias por hacer este trabajo por nosotros.

Como acabamos decir, vamos a crear la función `HAL_GPIO_EXTI_Callback`. La crearemos en el archivo `main.c`. Allí conmutamos el valor de la variable `estadoLED`. A continuación, tenéis el código.

> Archivo: `main.c`

```
...

/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    // funcion para la interrupcion del GPIO

    // conmutamos el estado del LED
    estadoLED = !estadoLED;
}
/* USER CODE END 4 */
...
```

Puesto que utilizamos la variable `estadoLED` en diferentes funciones, **haremos que sea una variable global**. Además, si **modificásemos** la variable `estadoLED` desde **diferentes puntos del programa**, añadiríamos el calificador **`volatile`** a la variable.

> Archivo: `main.c`

```
...

/* USER CODE BEGIN PV */
volatile _Bool estadoLED = FALSE;
/* USER CODE END PV */

...
```

¡Perfecto! Compilamos, depuramos y probamos, y todo debería de ir ok.

Como tenemos una versión de código funcionando… Ya sabéis…

1.3. Reto

Pues el mismo que con Arduino: hacer que el LED conmute entre apagado y parpadeando (cada 500 ms) mediante el pulsador B1 utilizando las interrupciones de los GPIOs y sin utilizar la función `HAL_Delay`. No debe de haber ninguna restricción de cuándo y con qué duración se pulsa B1. Debe de ir fino, fino.

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**

En el repositorio remoto en GitHub, debe de haber vuestra rama con, como mínimo, los 4 *commits* realizados durante la práctica: LED parpadeando, corrección del `main` después de regenerar el código, LED encendido y apagado con el pulsador, y el reto.

☐ **Reto**

Cómo mínimo, un *commit* que contenga vuestro código que solucione el reto propuesto.

☐ ~~**Informe**~~

No hay informe en esta práctica.

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, como en la práctica anterior (¡pero no hagáis el *merge*!). Acordaros 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 visto cómo programar eventos a nivel de registros utilizando la función `HAL_GetTick`. De este modo, logramos no bloquear la ejecución del programa durante los tiempos de espera creados con `HAL_Delay`.

También hemos visto cómo crear una interrupción para un GPIO con STM32CubeMX y cómo este las implementa. Mediante la interrupción, evitamos tener que hacer *polling* al GPIO del pulsador,

economizando el uso de los recursos del microcontrolador y logrando que no se pierdan pulsaciones indiferentemente de cómo y cuándo se realicen.

¡En la próxima práctica saltaremos de periférico y veremos el uso de *timers* o contadores!