MASB | STM32
Hello, World!

#stm32 #c #biomedical #microcontroladores #programacion

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

4 de febrero de 2020

Índice

1. Hello, World!

En esta segunda parte de la primera práctica, vamos a ver cómo lograr exactamente la misma funcionalidad que en la primera parte con el mismo microcontrolador, pero programando a nivel de registros. Programar a nivel de registros quiere decir entrar en las entrañas del microcontrolador y programarlo *manualmente*. Esto nos lo hace Arduino por detrás, sin que nos demos cuenta, facilitando la realización de pequeños proyectos o prototipos.

Pero… Entonces… ¿por qué no usar Arduino si me facilita la vida? Buena pregunta, mi querido Watson. Antes de ver causas, veamos la mayor consecuencia: **Arduino no tiene presencia (o es casi nula) en el ámbito industrial/profesional**. Hay varios motivos por los que Arduino no está presente en la industria o en el ámbito profesional, pero 2 son los más importantes: 1) Arduino tiene una licencia LGPL y 2) no optimiza los recursos de nuestro microcontrolador.

El primer motivo tiene como consecuencia la obligatoriedad de ofrecer bajo la misma licencia los archivos `.o` (archivos *object*) que genera Arduino durante el proceso de compilación del código. Esto quiere decir que el archivo *object* debe de estar disponible de manera abierta; y sobre estos archivos se puede realizar ingeniería inversa para poder reconstruir el código o generar una imagen del mismo… Esto, en el ámbito industrial/profesional, ya ofrece una primera barrera infranqueable: **no puedes proteger tu desarrollo**. Además, ligado con el segundo motivo, si modificas Arduino para adaptarlo a tu producto (para mejorar la eficiencia del uso de los recursos de tu microcontrolador, por ejemplo), debes de abrir directamente el código fuente. De aquí saltamos al segundo motivo.

Arduino funciona en mil y un microcontroladores disponibles. Consecuencia: **no está optimizado para ninguno en particular**. Trabajando con microcontroladores, los **recursos disponibles** son **limitados** y la optimización de los mismos es crucial. Trabajando a nivel de registros logramos esa optimización al permitirnos configurar hasta el más mínimo detalle de nuestro microcontrolador.

Y ahora viene el "Pero entonces… ¿¡Para que me has hecho hacer la práctica anterior en Arduino!?". Pues me remito al primer párrafo: **Arduino es una herramienta muy potente para realizar prototipos**. Es una herramienta que permite con **poco esfuerzo y tiempo realizar un prototipo** que luego ya se trasladará a una versión comercializable migrando el código en Arduino a código a nivel de registros. El ahorro en tiempo que ofrece Arduino durante el prototipado es un activo nada desdeñable cuyo impacto en el coste y el *time-to-market* es directo. Y este es un de los múltiples flujos de desarrollo utilizados en el ámbito profesional y es el que utilizamos en las prácticas: **primeramente realizamos un prototipo en Arduino para comprobar que logramos la funcionalidad deseada y luego trasladamos el código a nivel de registros y realizamos las optimizaciones pertinentes**.

En este caso, vamos a ver cómo trabajar con entradas/salidas digitales a nivel de registros con STM32CubeIDE. Para ello, utilizaremos las librerías **HAL (*Hardware Abstraction Layer*)** que ofrece STMicroelectronics. Las librerías HAL ofrecen una interfaz entre nuestro *software* y el *hardware* del

microcontrolador. Estas nos permiten un **mayor nivel de abstracción** de nuestro código (un código con un elevado nivel de abstracción se traduce en un código que está poco ligado al *hardware* utilizado y que, por lo tanto, puede migrarse-a/usarse-en otro microcontrolador distinto más fácilmente) y nos ofrece herramientas que **facilitan el desarrollo**. Aun así, estás librerías nos permiten, en caso que lo necesitemos, acceder directamente a las entrañas del microcontrolador para **optimizar nuestra aplicación**.

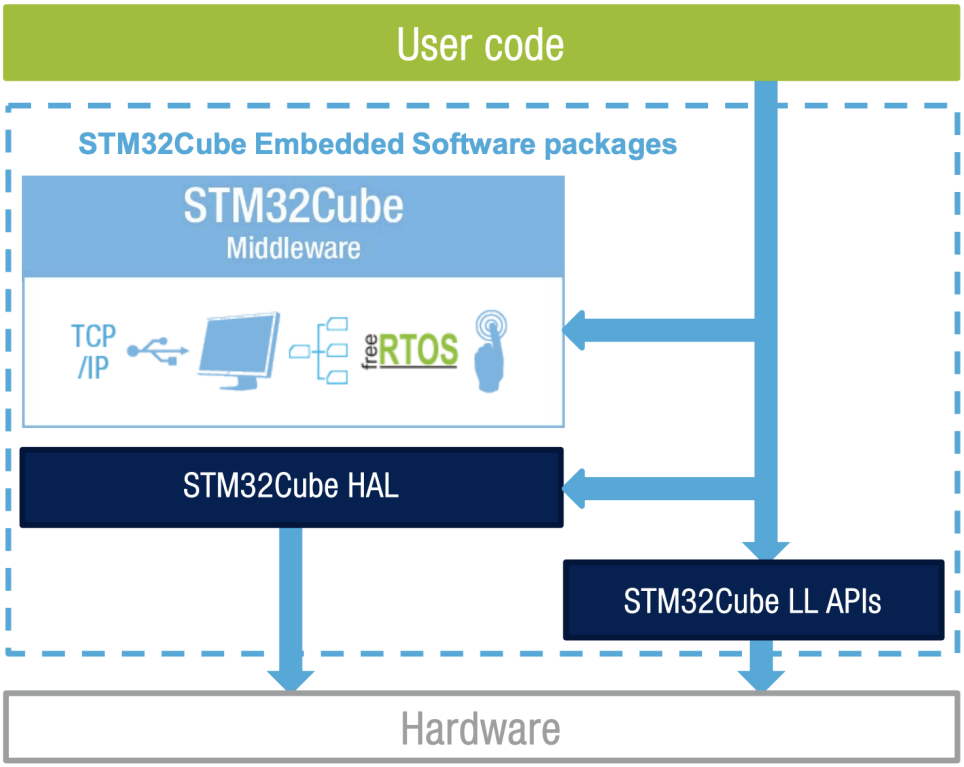

Figura 1: Esquema de HAL.

> Imagen de STMicroelectronics.

No todas las compañías ofrecen librerías HAL. El hecho que las ofrezcan o no **es un criterio más** a tener en cuenta a la hora de **escoger** desarrollar nuestro producto con un microcontrolador de una compañía u otra puesto que tiene un impacto directo en el tiempo de desarrollo (*time-to-market*) y, por lo tanto, en su coste. Por ello, las **principales compañías de desarrollo siempre suelen ofrecer HAL**. Las HAL de cada compañía son distintas, así que, como desarrolladores, nos tocará aprender las HAL del microcontrolador pertinente. Pero bien es cierto que una vez se ha aprendido a usar una HAL, dar el salto a la HAL de otra compañía no nos llevará mucho esfuerzo adaptarnos puesto que todas suelen seguir una filosofía de implementación parecida.

> Si no queremos usar las HAL, siempre podemos ir de chic@s dur@s y picar código al más bajo nivel (y así poder ser la típica persona que uno se encuentra en los foros de Internet gritando a los cuatro vientos que él/ella no usa HAL…), pero no suele ser la vía más común para la **mayoría** de desarrollos. Sin embargo, hay casos específicos que requieren de una gran optimización de los recursos y hará falta implementar nuestras propias HAL. Pero, repetimos, son **casos muy específicos**.

Ahora sí, ¡vamos a encender ese LED!

1.1. Objetivos

- Conocer STM32CubeIDE y su funcionamiento.
- Generar código mediante la herramienta STM32CubeMX.
- Conocer la estructura básica de `main.c` y su correcto uso.
- Utilizar los GPIOs tanto de entrada como de salida.
- Crear, compilar y cargar el primer programa en C/C++ a nivel de registros.
- Aprender nuevas funcionalidades y comandos de C y Git.

1.2. Procedimiento

1.2.1. Crear rama de desarrollo

Vamos a empezar asegurándonos que estamos en la rama `master` y que esta está actualizada con los últimos cambios que hayan podido haber en el repositorio remoto. Para cambiar a la rama `master` ejecutamos el siguiente comando:

```
1 git checkout master
```

Comprobamos que el terminal indica que estamos en la rama `master`. Si no nos ha dejado, seguramente sea debido a que tenemos cambios pendientes de hacer *commit*. Hacemos el *commit* pertinente si fuese el caso y cambiamos entonces de rama.

Cuando ya estemos en la rama `master`, para importar cualquier cambio que se haya hecho en el repositorio remoto, ejecutamos el comando:

```
1 git pull
```

Este comando hace la inversa del comando `git push`. Mientras que este último sube los cambios al repositorio remoto, el comando `git pull` los importa. El terminal nos dirá si hay cambios o no que se hayan incorporado a nuestro repositorio local.

Con todo correctamente sincronizado, vamos a crear nuestra rama de desarrollo para esta práctica:

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

Nos aseguramos que hemos podido cambiar correctamente de rama y ya podemos empezar con el desarrollo.

1.2.2. Preparación del IDE

1.2.2.1. Instalación de STM32CubeIDE Vamos a **instalar primeramente el IDE de STMicroelectronics: STM32CubeIDE**. Lo podéis encontrar en este enlace. Nos pedirá un correo al cual nos harán llegar un enlace con el que iniciar la descarga del instalador. Una vez descargado, lo iniciamos y hacemos un `Siguiente`, `Siguiente`, `Siguiente`, hasta finalizar la instalación. **Aseguraros de permitir instalar cualquier dispositivo que os pida**. Estos dispositivos son los controladores que nos permitirán cargar el programa en el microcontrolador y poder comunicarnos con él, en un futuro, mediante una comunicación serie.

1.2.2.2. Creación de un proyecto Una vez instalado STM32CubeIDE, lo iniciamos. Nos aparecerá una ventana en la que nos pedirá que seleccionemos la carpeta que hará de espacio de trabajo para la aplicación. En este espacio se nos creará una serie de archivos ligados a la configuración del IDE y a los proyectos disponibles. **Escogemos como directorio de trabajo o *workspace* la carpeta `stm32cube` dentro de nuestro repositorio local**.

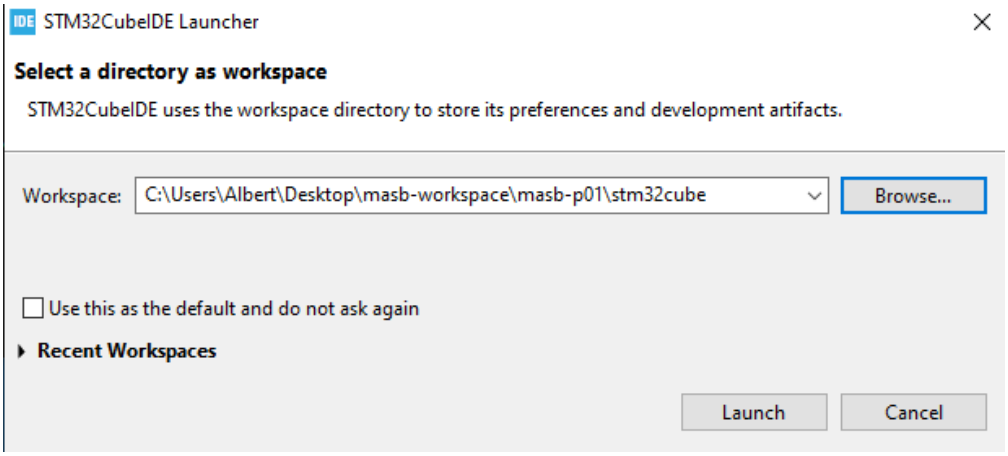

Figura 2: Selección del workspace.

> Los archivos generados en el *workspace* que no pertenecen al proyecto en si, suelen ser archivos auxiliares que no interesa guardar en el control de versiones. Por ello, estos archivos son añadidos al archivo `.gitignore` que hay dentro de la carpeta `stm32cube`. Te animo a echarle un vistazo para ver qué archivos no están sujetos al control de versiones y cómo se indica. Puedes abrir el archivo `.gitignore` con la aplicación Bloc de Notas.

Seleccionado el *workspace*, vamos a `File` > `New` > `STM32 Project`. Se nos abrirá una ventana en la que podemos escoger el microcontrolador para el cual vamos a desarrollar la aplicación. Pese a que podríamos escoger el microcontrolador STM32F401RET6U y empezar a partir de ahí a desarrollar, la herramienta permite escoger, en la pestaña `Board Selector`, nuestra EVB. De este modo, la herramienta nos preconfigurará el proyecto con los recursos disponibles en la EVB (LED, pulsador, comunicación UART, etc.). Nuestra EVB es la `NUCLEO-F401RE`. Podemos usar el buscador para encontrar más fácilmente nuestra EVB. La seleccionamos en la lista y clicamos en `Next`.

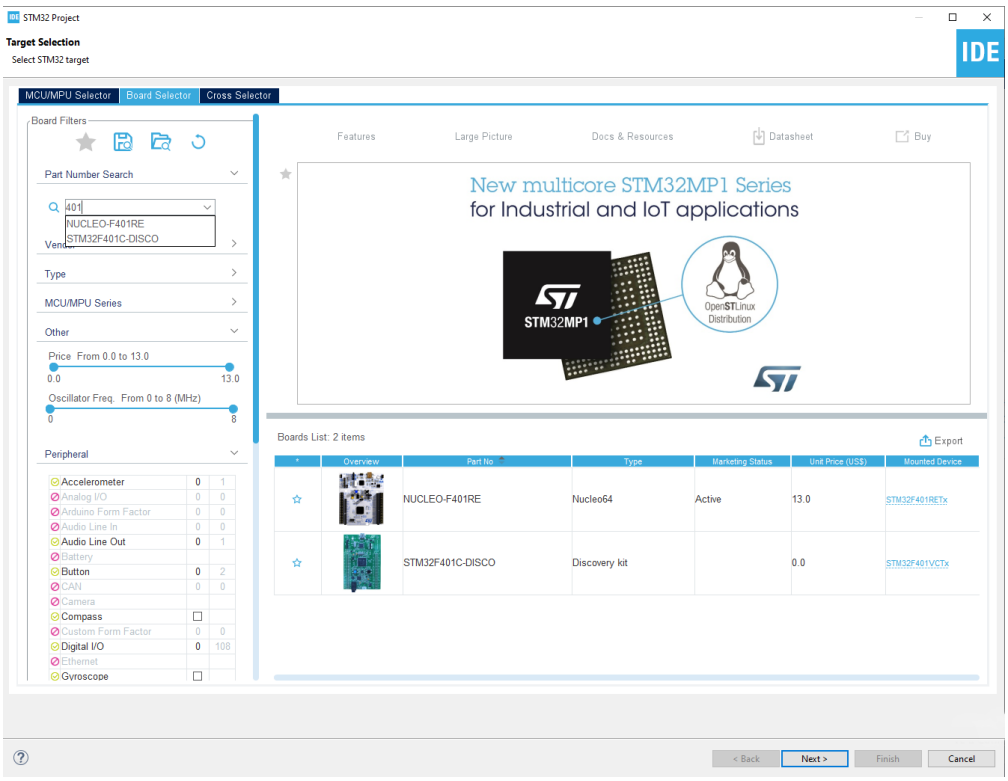

Figura 3: Selección de la EVB.

Una segunda ventana nos pedirá que indiquemos el nombre del proyecto. Le llamaremos `masb-p01`. También nos permite, si queremos, seleccionar un destino distinto al de por defecto donde almacenar el proyecto. Por defecto, se nos guardará en una carpeta, dentro del *workspace* que hemos indicado

al arrancar la aplicación, con el mismo nombre que el proyecto que hayamos indicado. **Dejamos la localización por defecto** y clicamos en `Finish`.

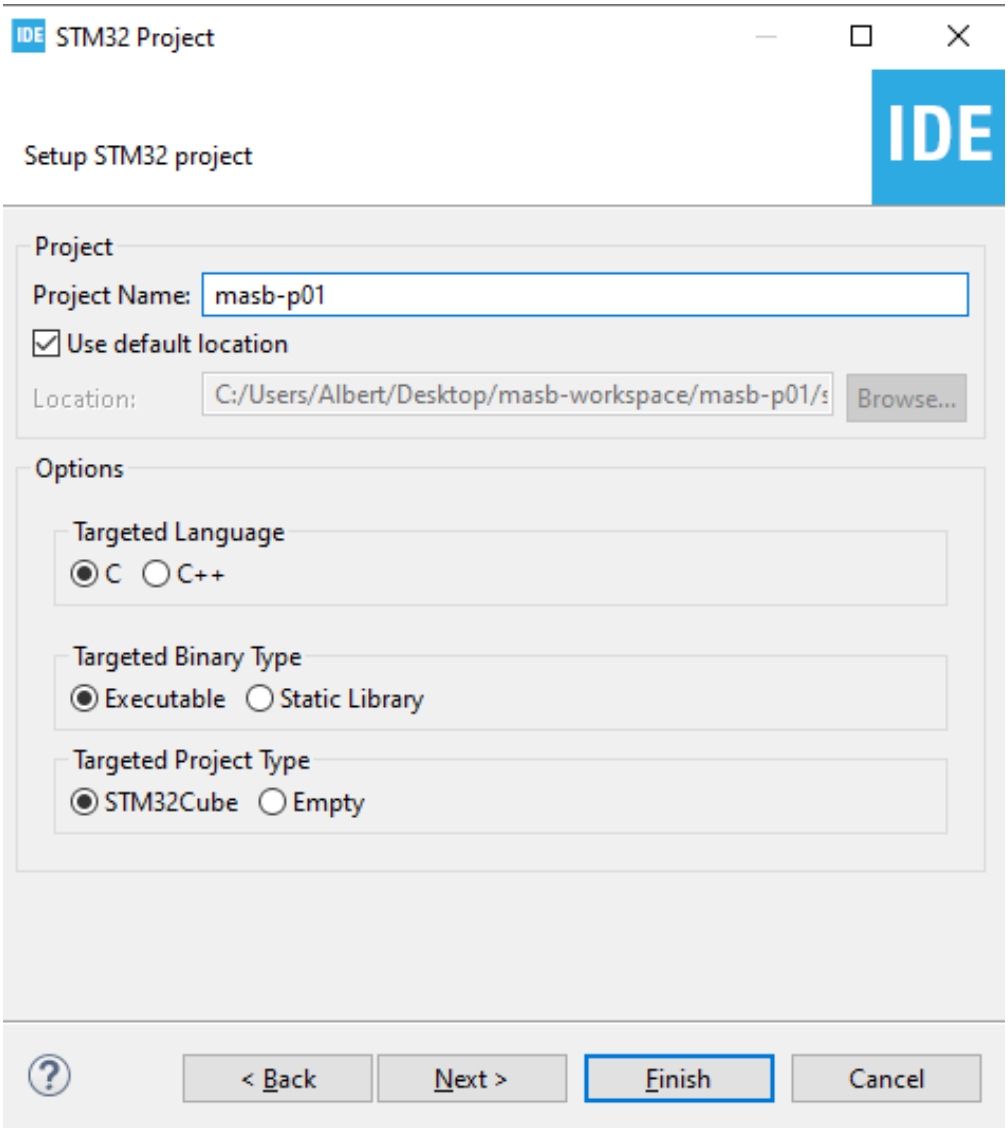

Figura 4: Configuración del proyecto.

Nos saldrá un mensaje preguntándonos si queremos inicializar los periféricos del microcontrolador a su modo por defecto. Le diremos que sí (es lo más recomendable siempre).

Seguidamente, nos comunica que el proyecto está asociado al uso de STM32CubeMX, la herramienta de STM que nos permite configurar las HAL gráficamente, y si queremos abrir dicha herramienta. Le decimos que por supuesto, por favor. Faltaría más.

> Marcad que recuerde esta selección si no queréis que os pregunte en futuros proyectos.

1.2.3. Ei, hola otra vez, Mundo

1.2.3.1. Configuración del pin conectado al LED En este punto, ya se nos ha creado el proyecto y todos sus archivos básicos. Lo primero que se nos ha abierto es la herramienta gráfica de configuración del microcontrolador: STM32CubeMX. Esta ventana/herramienta está asociada al archivo con extensión `.ioc` que podemos encontrar dentro del proyecto.

> Si no está desplegado ya, podéis desplegar el proyecto clicando en la flecha adjunta al proyecto en la ventana `Project Explorer` situada a la izquierda.

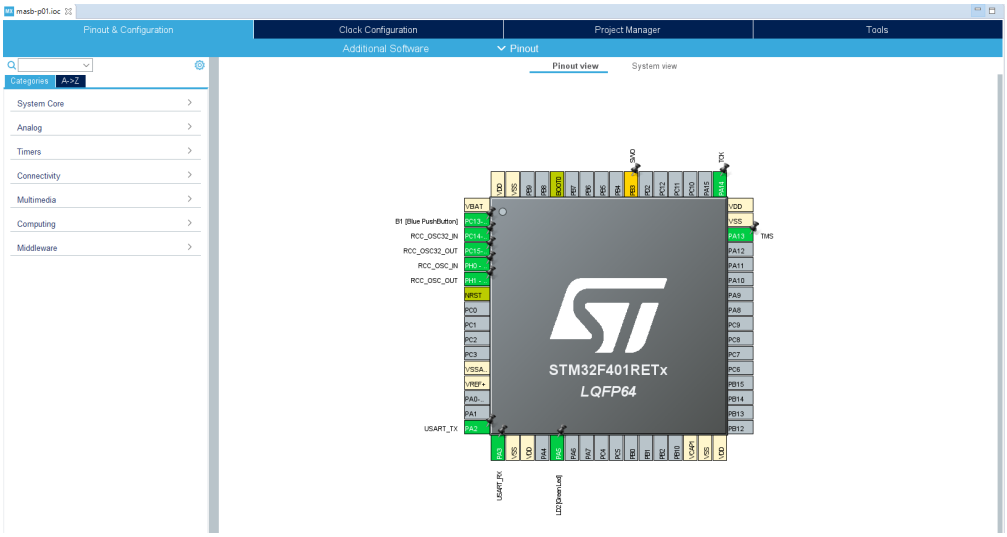

Figura 5: Herramienta de configuración del microcontrolador.

En la ventana del STM32CubeMX nos aparece una imagen interactiva del microcontrolador y todos sus pines. A través de esta imagen interactiva, podemos configurar la función de cada pin de manera individual (primera cosa que no podíamos hacer con Arduino Vanilla).

> "Vanilla" es un término que se utiliza para hacer referencia al hecho de usar un elemento sin modificarlo. Arduino Vanilla quiere decir "Arduino tal cual sale de la caja".

Vamos a configurar el pin al que está conectado el LED. Por defecto, ya viene configurado puesto que hemos seleccionado la EVB correcta durante la creación del proyecto, pero lo haremos igualmente para ver cómo hacerlo.

Para escoger la función que queremos que tenga un pin, hacemos clic izquierdo sobre el pin en cuestión y seleccionamos, entre las opciones disponibles, la función deseada. En este caso, hacemos clic izquierdo sobre el pin `PA5` y seleccionamos `GPIO_Output`. Puesto que ya venia configurado, lo que hemos hecho ahora es desconfigurarlo. Vuelve a hacer clic izquierdo en el pin y selecciona la función `GPIO_Output`.

Seguidamente, le daremos un alias o *label* a ese pin para que, cuando estemos programando, el **código sea mucho más legible**. El alias se lo ponemos haciendo clic derecho sobre el pin `PA5` y seleccionando `Enter User Label`. En el cuadro de texto emergente escribimos `LED`.

Si clicamos en `System view`, situado justo encima de la imagen interactiva, iremos a una vista en la que nos aparecen los diferentes periféricos actualmente configurados. Entre los diversos periféricos preconfigurados por STM32CubeMX, aparece el periférico `GPIO`. Hacemos clic sobre este periférico e iremos a una vista detallada de la configuración de los GPIOs. Aquí aparecen 2 pines configurados. Uno de ellos es el LED que acabamos de configurar, y el otro es el pulsador que preconfigura la herramienta. Si seleccionamos la fila del pin del LED, nos aparece abajo un formulario donde podemos configurar diferentes aspectos del pin: la salida por defecto, su tipo de salida, si utiliza pull-up/down o no, velocidad del pin y su alias. En este caso, no necesitáis modificar ninguna propiedad más allá de escoger qué valor de salida por defecto queréis que tenga el LED. **Por defecto**, si no queréis que sea al contrario, **la salida estará en nivel bajo y el LED estará apagado**.

Ya tenemos el microcontrolador correctamente configurado. Damos a `File` > `Save` y os preguntará si queréis generar el código a partir de esta configuración. Clicad en `Yes`.

> Si por lo que fuera no os propone generar el código al guardar, siempre podéis generar el código haciendo clic en `Project` > `Generate code` o clicando sobre el icono  .

1.2.3.2. El *main* Al generarse el código, STM32CubeMX nos crea diferentes archivos en el proyecto y nos edita/modifica el archivo `main.c` de nuestra aplicación. **Este archivo contiene la función del mismo nombre, `main`, la cual será la que el microcontrolador ejecute**. A través del `Project Explorer`, abrid el archivo `main.c` que se encuentra dentro del proyecto en `Core` > `Src` > `main.c`. Haced un *scroll* rápido de arriba a bajo. Todo lo que contiene el archivo, y muchas más líneas de código repartidas por diferentes archivos, nos lo ha generado STM32CubeMX por nosotros. ¡Gracias STM32CubeMX! ¡¡Vivan las HAL!! Veamos la estructura del archivo `main.c`.

Utilizando las HAL, **los comentarios toman un papel fundamental** puesto que nos indican dónde podemos, y dónde no, escribir nuestro código. Si lo hacemos en un lugar incorrecto, al volver a generar el código con STM32CubeMX (porque cambiamos alguna configuración o parámetro), este nos borrará nuestro código si está donde no debe de estar. Así que, ¡cuidado con esto!

¿Dónde podemos añadir código? **Podemos ingresar código entre los comentarios que empiecen con**

`USER CODE BEGIN` y **`USER CODE END`**. Por ejemplo, el siguiente código corresponde a las primeras líneas de código de `main.c` hasta la línea 32 (**no añadáis nada de esto a vuestro archivo `main.c`**).

```
1  /* USER CODE BEGIN Header */
2  --> AQUI PODEMOS ANADIR CODIGO
3  /**
4    ******************************************************************************
5    * @file           : main.c
6    * @brief          : Main program body
7    ******************************************************************************
8    * @attention
9    *
10   * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
11   * All rights reserved.</center></h2>
12   *
13   * This software component is licensed by ST under BSD 3-Clause
       license,
14   * the "License"; You may not use this file except in compliance with
       the
15   * License. You may obtain a copy of the License at:
16   *                        opensource.org/licenses/BSD-3-Clause
17   *
18   ******************************************************************************
19   */
20 /* USER CODE END Header */
21
22 /* Includes
     ------------------------------------------------------------------*/
23 #include "main.h"
24 --> AQUI NO PODEMOS ANADIR CODIGO
25
26 /* Private includes
     ----------------------------------------------------------*/
27 /* USER CODE BEGIN Includes */
28 --> AQUI PODEMOS ANADIR CODIGO
29 /* USER CODE END Includes */
30
31 /* Private typedef
     -----------------------------------------------------------*/
32 /* USER CODE BEGIN PTD */
33 --> AQUI PODEMOS ANADIR CODIGO
34 /* USER CODE END PTD */
35
36 ...
```

Sabiendo esto sobre los comentarios y que no podemos añadir código allí donde queramos, ahora sí, pasamos a ver qué contiene el archivo `main.c` ignorando/eliminando todo lo que esté comentado.

La primera instrucción es:

```
1 ...
2
3 /* Includes
     ------------------------------------------------------------------*/
4 #include "main.h"
5
6 ...
```

Esta instrucción, y **todas las que empiezan con una almohadilla #**, no la ejecuta el microcontrolador, sino que **es una instrucción para el preprocesador**. La instrucción `include` le pide al preprocesador que, como su nombre bien indica, incluya el archivo indicado antes de realizar la compilación del programa. En este caso, el archivo incluido es `main.h`, un archivo con extensión `.h` llamado de tipo `header`. Este tipo de archivo, que no utilizamos en Arduino (porque nos los oculta, no porque no estén) contiene la definición de constantes, macros, variables globales, prototipos de funciones, etc. De momento no sabemos que son todos estos elementos. Poco a poco los iremos viendo. Para ver el contenido de este `main.h`, podemos abrirlo desde el `Proyect Explorer` en `Core > Inc > main.h`. Si se da el caso que no sabemos la localización del archivo (¡que suele ser lo más común!) podemos situar el cursor encima del nombre del archivo indicado en la instrucción `#include` y pulsar `F3`. Esto nos abrirá directamente el archivo `main.h`.

> **¡Super útil el atajo `F3`!** Cuando no sepáis de dónde sale un archivo, una función o variable, situad el cursor encima de él y pulsad `F3` para que os muestre su origen.

El siguiente código es:

```
1 ...
2
3 /* Private variables
     ---------------------------------------------------------*/
4 UART_HandleTypeDef huart2;
5
6 ...
```

En esta instrucción se declara una variable global llamada `huart2`. Al ser global, esta variable puede ser usada desde cualquier función dentro del archivo en el que ha sido declarada. Recordemos, una variable es global cuando no está declarada dentro de ninguna función.

Es de tipo `UART_HandleTypeDef`, un tipo de variable no estándar definido mediante la palabra clave o *keyword* `typedef`. La *keyword* `typedef` sirve para asignar un nombre alternativo a un tipo de variable existente para hacer el código más legible.

A continuación, en `main.c` nos encontramos:

```
...

/* Private function prototypes
   -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

...
```

En esta sección del código se indican los prototipos de 3 funciones distintas. En Arduino esto no lo hacemos porque nos lo hace él automáticamente. Prototipar una función sirve para informar al compilador sobre qué funciones existen, qué tipo de valor devuelve (si devuelve) y qué tipo de parámetros tiene (si tiene). Estas 3 funciones se utilizan dentro de la función `main` y su código se encuentra después de esta.

Precisamente, después de los prototipos, aparece la función `main`, la función que ejecutará el microcontrolador.

```
...

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_USART2_UART_Init();

  while (1)
  {

  }

}

...
```

La función `main` está implementada de tal manera que **su contenido es altamente legible** (¡que es siempre lo que buscamos al desarrollar nuestro código!). Sin saber lo que hace en detalle, solo por el nombre de las funciones, podemos ver que en la función `main`, primeramente, se inicializan las librerías HAL. Luego se configura el reloj del sistema. Seguidamente, se inicializa el periférico GPIO. A continuación, se inicializa el periférico de comunicación UART y, finalmente, se entra en el *while loop*. Las 4 primeras instrucciones solo se ejecutarán una **sola vez al inicio del programa (equivalente a la**

función `setup`en Arduino). Lo que contenga el *while loop* se ejecutará continuamente (equivalente a la función `loop` de Arduino).

Por último, nos quedan una serie de funciones generadas por STM32CubeMX en cuyo detalle de momento no entraremos. Básicamente, hacen las tareas comentadas anteriormente: inicializar el sistema de reloj, el GPIO y la UART.

```
...

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{

...

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

...

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{

...

}

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
```

```
44
45  }
46
47  #ifdef  USE_FULL_ASSERT
48  /**
49    * @brief  Reports the name of the source file and the source line
          number
50    *         where the assert_param error has occurred.
51    * @param  file: pointer to the source file name
52    * @param  line: assert_param error line source number
53    * @retval None
54    */
55  void assert_failed(uint8_t *file, uint32_t line)
56  {
57
58  }
59  #endif /* USE_FULL_ASSERT */
```

1.2.3.3. “Madre mía... ¡Pero si solo quiero hacer parpadear el LED!” Vale, vale. Vamos a ello. Para hacer que parpadee el LED, únicamente tenemos que añadir la instrucción de encender y apagar. No hace falta configurar/inicializar nada ya que STM32CubeMX ya lo ha hecho por nosotros.

Puesto que es algo que queremos que se ejecute continuamente, lo añadiremos dentro del *while loop*. ¿Dónde exactamente? Donde os indico en el siguiente código (**no añadáis esto a vuestro `main.c`**):

```
1  ...
2
3    /* Infinite loop */
4    /* USER CODE BEGIN WHILE */
5    while (1)
6    {
7      PUEDES PONER CODIGO AQUI
8      /* USER CODE END WHILE */
9
10     NI SE TE OCURRA PONER NADA AQUI
11
12     /* USER CODE BEGIN 3 */
13     PUEDES PONER CODIGO AQUI
14   }
15   /* USER CODE END 3 */
16
17 ...
```

Más claro: el agua. **Si pones código donde no toca, STM32CubeMX te lo eliminará al regenerar el código.**

Y ¿qué funciones existen para hacer que la salida del pin conectado al LED `PA5` sea `0` o `1`? Aquí es de suma importancia la documentación que os pueda aportar la compañía propietaria del HAL. STMi-

croelectronics provee una guía de usuario con todas las instrucciones disponibles en el HAL. Podéis encontrar este documento aquí. **¡Guardad este documento como si fuera oro y tenedlo siempre a mano!**

En la documentación encontramos, entre un gran número de funciones, la siguiente función dentro de las disponibles para los GPIOS: `HAL_GPIO_WritePin`. Esta es la instrucción que debemos de utilizar para fijar el valor de salida de un pin GPIO.

> **Es sumamente importante que sepáis manejaros con la documentación**. Por ello, os recomiendo buscar todas las funciones que utilicemos en la guía de usuario de las HAL. Allí encontraréis su cometido y su uso.

Así pues, vamos a hacer que el LED se apague y se encienda añadiendo la función `HAL_GPIO_WritePin` dentro del *while loop*.

```
...

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET); //
          encendemos el LED
      HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET); //
          apagamos el LED
    /* USER CODE END WHILE */

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

...
```

> Un *tip* muy útil, **extremadamente útil**, es el atajo `CTRL`+`ESPACIO`. Si mientras escribís la función pulsáis ese atajo, os aparecerá una lista en la que podéis encontrar todas las funciones, variables y macros disponibles. A medida que vais escribiendo, se os irán filtrando los resultados de la lista hasta que podáis encontrar el elemento que buscáis. Lo mismo funciona para variables y macros.

La función `HAL_GPIO_WritePin` requiere tres parámetros: el puerto del pin, el número del pin y el estado de salida deseado. El pin del LED es el `PA5` que corresponde al puerto `A` (`GPIOA`) y al pin 5 (`GPIO_PIN_5`). Y queremos que esté en nivel alto (`GPIO_PIN_SET`) y bajo (`GPIO_PIN_RESET`). Todo esta información la encontramos en el documento de las HAL.

Puesto que al pin del LED le hemos puesto un *label* en STM32CubeMX, podemos utilizar las macros `LED_GPIO_Port` y `LED_Pin` haciendo el código más legible.

> "Ya has mencionado varias veces lo de las macros. ¿Qué es eso?" Una macro se genera a partir de una instrucción para el preprocesador (recordad, instrucciones que empiezan con almohadilla #). Esta instrucción es `#define`. Con esta instrucción le decimos al preprocesador que, antes de compilar, reemplace una determinada macro por un código dado. Por ejemplo, la macro `LED_Pin` se define:
>
> ```
> 1  #define LED_Pin                 GPIO_PIN_5
> ```
>
> Por lo que cada vez que usemos `LED_Pin` será sustituido por `GPIO_PIN_5` antes de compilar el programa. A su vez, `GPIO_PIN_5` es otra macro:
>
> ```
> 1  #define GPIO_PIN_5    ((uint16_t)0x0020)  /* Pin 5 selected    */
> ```
>
> El atajo `F3` también aplica a las macros, por lo que siempre podéis ver cual es la definición de una macro.
>
> Las macros son muy útiles para hacer el código más legible y para modificar un programa rápidamente frente a un cambio de diseño. Por ejemplo, si utilizamos la macro `LED_Pin` en todo el código y el LED pasa de estar del pin `5` al pin `2`, podemos cambiar la definición de la macro `LED_Pin`:
>
> ```
> 1  #define LED_Pin                 GPIO_PIN_2
> ```
>
> y todo nuestro código se actualiza automáticamente. Si hubiésemos usado `GPIO_PIN_5` directamente, tendríamos que ir buscando todos los `GPIO_PIN_5` uno a uno en el código y reemplazarlos por `GPIO_PIN_2`.
>
> Por último, suele ser una norma no escrita escribir las macros con todo mayúsculas para hacerlas fácilmente identificables.

Perfecto, vamos a compilar el proyecto. Esto lo podemos hacer desde `Project` > `Build project` o clicando en el icono del martillo  . Si no hay ningún error, en la consola situada abajo, veremos el mensaje:

```
03:11:59 Build Finished. 0 errors, 0 warnings. (took 2s.589ms)
```

Figura 6: No error en la consola.

Si es así, vamos a cargar el programa en el microcontrolador. Para ello, nos aseguramos que tenemos la EVB conectada al ordenador y vamos a `Run` > `Debug` o pulsamos en el icono del insecto  . La primera vez que depuremos o *debuggeemos* un proyecto nos saldrá la siguiente ventana:

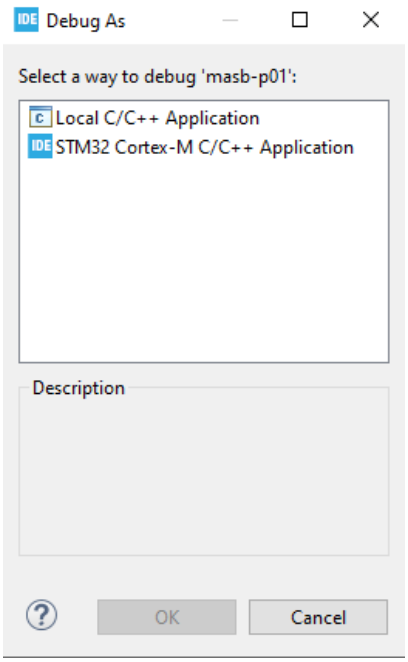

Figura 7: Selección de la configuración de depuración.

Seleccionamos `STM32 Cortex-M C/C++ Application` y clicamos `Ok`. En la siguiente ventana que aparece, volvemos a clicar `Ok`. Esto ya no lo tendremos que hacer más en este proyecto.

Al empezar a *debuggear* un programa, se nos pedirá cambiar la perspectiva del IDE. Clicamos en `Switch` habiendo marcado, previamente, la casilla que nos permite recordar esta elección. La perspectiva cambiará y un cursor se nos situará en la primera instrucción dentro de la función `main`.

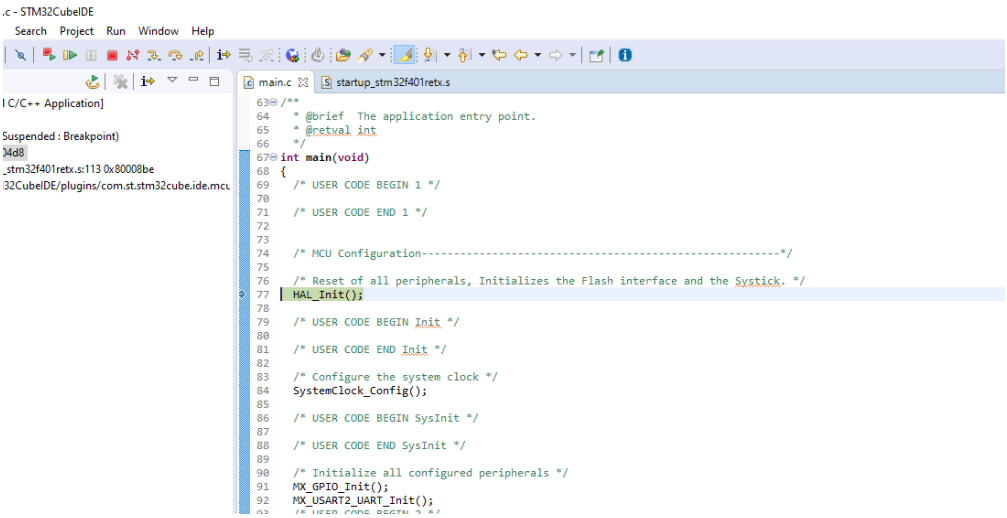

Figura 8: Inicio de la depuración.

La ejecución de nuestro programa queda pausado justo antes de la ejecución de la instrucción resaltada. Para iniciar la ejecución, clicamos `Run` > `Resume` o sobre el icono  .

Como ya vimos en la primera parte de la práctica en Arduino, el LED no parpadea. Faltan los *delays*.

Paramos la depuración del programa clicando en `Run` > `Terminate` o sobre el icono  y nos vamos a añadir los *delays*.

En la documentación de las HAL, buscamos cómo implementar un *delay*. Encontraremos que nuestra función es `HAL_Delay()` donde el tiempo se indica en milisegundos. Vamos a incorporar la instrucción al código.

```
...

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET); //
         encendemos el LED
      HAL_Delay(1000); // esperamos 1 segundo
      HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET); //
         apagamos el LED
      HAL_Delay(1000); // esperamos 1 segundo
    /* USER CODE END WHILE */

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

...
```

Probamos el programa y, esta vez sí, el LED parpadea cada 1 segundo. Paramos la depuración y vamos a guardar esta versión del código.

Puesto que al crear un proyecto nuevo se nos han creado muchos archivos nuevos, añadiendo uno a uno los archivos sería algo tedioso. Por ello utilizaremos la instrucción Git para añadir archivos al *stage* que ya conocemos, pero en lugar de indicar los nombres de los archivos, haremos que añada todos los archivos nuevos, editados o eliminados:

```
git add -A
```

"Guau… Haberme enseñado eso desde el principio…". No. En este caso es pertinente añadir todos los archivos de una sola vez porque hemos inicializado un proyecto nuevo y se nos han creado muchos archivos en una sola acción, pero lo recomendable es hacer *commits* en los que se modifiquen pocos archivos y añadir estos manualmente al *stage* como hemos hecho hasta ahora.

Hacemos el *commit* y, si queremos, un *push* para subir los cambios al repositorio remoto.

```
git commit -m "LED parpadeando cada 1 segundo."
git push
```

Acordaos que, como hemos creado una rama nueva, el comando `push` nos dará error a la vez que nos indicará el comando que debemos de utilizar. **Copiad ese comando y ejecutadlo**. Una vez la rama nueva se ha subido al repositorio remoto, podemos ejecutar el comando `push` normalmente.

1.2.4. Entradas digitales

Ahora vamos a tratar de apagar/encender el LED mediante el pulsador B1. Para ello deberemos de configurar el pin `PC13`. Abrimos STM32CubeMX haciendo doble clic sobre el archivo con extensión `.ioc` en nuestro proyecto. En la imagen interactiva configuramos el pin `PC13` como `GPIO_Input` y le ponemos como *label* `PULSADOR`. Hecho esto, regeneramos el código  .

Volvemos al archivo `main.c` y si hemos añadido nuestro código allí donde toca, este seguirá ahí. ¿No está? Tendrás que volver a añadir el código.

> En realidad, **aquí Git sería la herramienta a utilizar** ya que siempre haríamos un *commit* justo antes de cualquier regeneración de código que hagamos. Así, si se diera el caso que se nos borra el código, podemos volver a la versión inmediatamente anterior y restaurar nuestro programa. Pero como aún no hemos visto como volver a una versión anterior, si se te borra el código, deberás de añadirlo de nuevo a `main.c` (¡y esta vez en el sitio correcto!).

En `main.c` parece que no se ha modificado nada, pero sí. Un ejemplo de código modificado es la función `MX_GPIO_Init`, donde ahora se inicializa el pin `PC13` como `PULSADOR` y antes no.

Pues vamos a hacer que según el estado del pulsador se alterne entre encender y apagar el LED. Nuestra función, según la documentación de las HAL, es `HAL_GPIO_ReadPin`, quedando el código de la siguiente forma.

```
...

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      if (HAL_GPIO_ReadPin(PULSADOR_GPIO_Port, PULSADOR_Pin) ==
         GPIO_PIN_RESET) { // si B1 es pulsado
        if (HAL_GPIO_ReadPin(LED_GPIO_Port, LED_Pin) == GPIO_PIN_RESET)
            { // si el LED esta apagado
           HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET); //
               lo encendemos
```

```
10          } else { // si no
11              HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET);
                    // lo apagamos
12          }
13        }
14      /* USER CODE END WHILE */
15
16      /* USER CODE BEGIN 3 */
17    }
18    /* USER CODE END 3 */
19
20  ...
```

Podemos refactorizar el código utilizando una función disponible en las HAL: `HAL_GPIO_TogglePin`.

> Refactorizar código significa mejorarlo rehaciendo la misma funcionalidad con el objetivo de mejorar su eficiencia, legibilidad, nivel de abstracción, etc.

De este modo, el código quedaría:

```
1   ...
2
3     /* Infinite loop */
4     /* USER CODE BEGIN WHILE */
5     while (1)
6     {
7         if (HAL_GPIO_ReadPin(PULSADOR_GPIO_Port, PULSADOR_Pin) ==
              GPIO_PIN_RESET) { // si B1 es pulsado
8           HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin); // alternamos
                estado
9       }
10      /* USER CODE END WHILE */
11
12      /* USER CODE BEGIN 3 */
13    }
14    /* USER CODE END 3 */
15
16  ...
```

Compilamos y ejecutamos el programa y, como ya podíamos intuir, no acaba de funcionar bien puesto que la lectura del pulsador la hacemos por *level triggered*. Vamos hacerlo *edge triggered* con una variable booleana.

Si buscamos en los comentarios de `main.c`, veremos que STM32CubeMX nos indica que las variables globales debemos de añadirlas en la siguiente posición:

```
1   ...
2
```

```
3  /* USER CODE BEGIN PV */
4
5  /* USER CODE END PV */
6
7  ...
```

Podemos utilizar el buscador para encontrar estos comentarios. El buscador lo podéis abrir clicando en `Edit > Find/Replace...` o directamente con el atajo `CTRL+F`. Allí añadimos nuestra variable global booleana.

```
1  ...
2
3  /* USER CODE BEGIN PV */
4  bool highToLowTransition = false;
5  /* USER CODE END PV */
6
7  ...
```

Antes de seguir vamos a compilar el programa  ...

¡Pum!¡Explosión! El IDE nos indica error. Los podemos ver directamente en la ventana `Console`, abajo del IDE, o desde la ventana `Problems` situada en el mismo sitio. El compilador se queja de que el tipo de variable `bool` no existe, ni la palabra clave **`false`**. En el primer error nos propone una posible solución, que es sustituir `bool` por `_Bool`. En este entorno de desarrollo no existe por defecto las variables tipo `bool` y se utilizan las tipo `_Bool`. Hacemos la sustitución y logramos solucionar el primer error.

> Es **muy importante acostumbrarse a leer la información que nos provee el compilador por consola** puesto que nos da mucha información sobre el error para poder solucionarlo, así como posibles soluciones, como es el caso.

El segundo error nos dice que no existe **`false`**. **La lógica en un controlador es que un valor que es 0 será siempre `false`, mientras que todo lo diferente a 0 será `true` (lo mismo aplica en Arduino)**. Por ello, si no existe **`true`**/**`false`**, podemos usar `0` y `1` (o cualquier valor distinto de 0) o podemos crearnos nuestras propias macros para hacer el código más legible.

```
1  #define TRUE            1
2  #define FALSE           0
```

La última opción es la más adecuada. Vamos a incorporar las macros allá donde nos indiquen los comentarios de STM32CubeMX.

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

```
4  /* USER CODE BEGIN PM */
5  #define TRUE            1
6  #define FALSE           0
7  /* USER CODE END PM */
8
9  ...
```

Ahora sí, modificamos la declaración de la variable anterior:

```
1  ...
2
3  /* USER CODE BEGIN PV */
4  _Bool highToLowTransition = FALSE;
5  /* USER CODE END PV */
6
7  ...
```

Y ahora sí. Compilamos y *no problem my friend*.

> Es una muy buena práctica, **altamente recomendable**, ir compilando el código de vez en cuando mientras vayamos desarrollando de cara a detectar fallos lo antes posible. Si realizamos un desarrollo largo y luego tenemos errores, será más difícil saber de dónde vienen.

Ahora en la función `main` acabamos de desarrollar el código:

```
1  ...
2
3    /* Infinite loop */
4    /* USER CODE BEGIN WHILE */
5    while (1)
6    {
7        if (HAL_GPIO_ReadPin(PULSADOR_GPIO_Port, PULSADOR_Pin) ==
             GPIO_PIN_RESET) { // si B1 es pulsado
8            if (highToLowTransition == FALSE) {  // y no se habia pulsado
                 antes
9                highToLowTransition = TRUE;  // guardamos que se ha
                     pulsado
10               HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin); // y
                     alternamos estado del LED
11           }
12       } else { // si en cambio, el boton no esta pulsado
13           highToLowTransition = FALSE;  // reiniciamos la variable a
                 FALSE
14       }
15     /* USER CODE END WHILE */
16
17     /* USER CODE BEGIN 3 */
18   }
19   /* USER CODE END 3 */
20
```

```
21  ...
```

Compilamos/depuramos  y, ahora sí, todo funciona correctamente. Paramos la depuración y guardamos esta versión del código.

```
1  git add -A
2  git commit -m "Alternamos estado del LED con el pulsador B1."
3  git push
```

Una vez más, usamos `git add -A` porque hemos modificado muchos archivos al regenerar el código con STM32CubeMX. Si no, hubiésemos hecho:

```
1   git add stm32cube/masb-p01/Core/Src/main.c
```

Recordar que siempre podéis ver los archivos que se han creado, modificado o eliminado con la instrucción `git status`.

1.3. Reto

¿Os imagináis cuál es? Sí. El mismo que con Arduino: hacer que el LED alterne entre estar apagado o parpadeando cada 500 ms con el pulsador B1. Realizad el reto y, cuando lo tengáis, el pertinente *commit* y *push*.

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 3 *commits* realizados durante la práctica: LED parpadeando, LED encendido y apagado con el pulsador, y el reto.

☐ **Reto**

En el último *commit* antes citado, debe de haber vuestro código que solucione el reto propuesto.

☐ **Informe**

Informe con el mismo formato/indicaciones que el anterior.

El informe debe de contener:

- ☐ Comandos nuevos de Git que se hayan utilizado en esta práctica y para qué sirven.
- ☐ Qué es STM32CubeMX.
- ☐ Tabla con las funciones HAL utilizadas 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, como en la práctica anterior.

> **¡Importante!** Se me pasó comentároslo en la práctica anterior. Cuando hagáis el PR, añadidme como *reviewer* en el PR. Lo podéis hacer clicando sobre el engranaje al lado del apartado `Reviewers` situado a la derecha del PR. Ahí me seleccionáis en la lista. De este modo, GitHub me enviará una notificación pidiéndome que eche un vistazo a vuestra práctica.
>
> En un futuro, durante el proyecto, podéis utilizar esta funcionalidad de revisión para pedir a vuestros compañeros que os revisen vuestra propuesta de código antes de insertarla en una rama.
>
> Durante la revisión de los PR, puede que se os pida cambiar algo que pueda mejorar vuestro código o práctica. Si después de hacer un PR hacéis más *commits* para aplicar esos cambios requeridos, no tenéis que hacer otro PR. El PR se actualiza automáticamente con los últimos *commits*.

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 esta segunda parte, hemos finalizado la primera práctica de MASB, donde hemos visto cómo trabajar de manera básica con los GPIOs de un microcontrolador a nivel de registros con las HAL.

En Git hemos visto cómo saltar entre ramas y cómo sincronizar el repositorio local para incorporar modificaciones que se hayan podido hacer en el repositorio remoto. También hemos visto cómo añadir varios archivos a la vez al *stage*.

Hemos aprendido cómo crear un proyecto en STM32CubeIDE y cómo configurarlo gráficamente con la herramienta STM32CubeMX. También hemos aprovechado para echar un vistazo a la estructura del archivo `main.c` del proyecto y en qué lugares debemos de insertar nuestro código.

Por último, hemos aprendido otros aspectos de la programación en C, como el uso de macros o `typedef`.