MASB | Arduino
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!

Nadie empieza el aprendizaje de un nuevo lenguaje de programación (o en este caso, en un nuevo *target*) sin empezar con un **"Hello, World!"**. Normalmente, los "Hello, World" son pequeños programas que se ejecutan en el *host* o Ordenador Personal (PC) y muestra por pantalla la frase "Hello, World". En nuestro caso, ¡no tenemos pantalla! El equivalente en microcontroladores es hacer parpadear un LED. Nosotros lo llevaremos un poco más allá haciéndo que el LED se encienda a nuestra voluntad usando el botón o *push button* de la Placa de Evaluación (EVB).

En esta práctica aprenderemos a crear un programa con Arduino. En este programa usaremos el pilar fundamental de los microcontroladores: los GPIOs digitales. El uso de los GPIOs digitales está presente en infinidad de dispositivos y aplicaciones que van desde detectar la pulsación de botones y generación de indicadores en interfaces de usuario, hasta detectar o generar eventos en un dispositivo biomédico, como la detección de presencia de la tira sensora de un glucómetro o detectar el estado de la alimentación en un *Point-of-Care* (PoC) alimentado con baterías, entre muchos otros usos.

1.1. Objetivos

- Conocer la EVB STM32 Nucleo-F401RE.
- Conocer Arduino IDE y su funcionamiento.
- Crear, compilar y cargar el primer programa en Arduino.
- Utilizar los GPIOs tanto de entrada como de salida.
- Primera toma de contacto con VCS (Git).

1.2. Procedimiento

1.2.1. Preparación del IDE

1.2.1.1. Instalación de Arduino IDE Lo primero que haremos es instalar la aplicación para desarrollar programas basados en Arduino: Arduino IDE. El acrónimo IDE responde a *Integrated Development Environment* o, en el idioma de Cervantes, Entorno de Desarrollo Integrado. Un IDE es una aplicación que proporciona las herramientas para desarrollar, compilar, implementar y depurar *software*. Para instalarlo, empezaremos descargándonos la aplicación del siguiente enlace. La versión de Arduino IDE utilizada para la preparación de estas prácticas ha sido la 1.8.10.

> La aplicación está disponible para múltiples Sistemas Operativos (OS). Las prácticas se pueden realizar en todos los OSs compatibles con la aplicación, pero **el detalle de su desarrollo, así como la garantía de su correcto funcionamiento, se basará en Windows**. Específicamente,

> Windows 10.

La instalación de Arduino IDE es tan fácil como ejecutar el archivo `.exe` descargado y utilizar el algoritmo matemático `Siguiente`, `Siguiente`, `Siguiente`, ..., y `Finalizar`. Las instrucciones detalladas de cómo hacer clic en los diferentes `Siguente` las podéis encontrar en este enlace.

1.2.1.2. Configuración de la EVB

Una vez instalado, Arduino IDE es capaz de programar sus propias EVB: Arduino UNO, Arduino Zero, Arduino Due, entre otras. Sin embargo, la EVB que utilizaremos es una placa de STMicroelectronics (STM): la STM32 Nucleo-F401RE. Esta EVB es compatible con Arduino; pero, puesto que no es comercializada por la compañía Arduino, los archivos de configuración de la EVB deben de ser importados al IDE. Por suerte para nosotros, es muy fácil importar los archivos de configuración en Arduino IDE y STM provee los archivos de configuración oficiales para sus EVB.

El procedimiento para importar los archivos de configuración se encuentra en uno de los repositorios de la cuenta oficial de STM en GitHub y lo podéis encontrar en el siguiente enlace.

1.2.2. La EVB STM32 Nucleo-F401RE

Como hemos visto anteriormente, la EVB que utilizaremos es la STM32 Nucelo-F401RE de STMicroelectronics. Esta EVB utiliza el microcontrolador STM32F401RET6U del mismo fabricante. Lo interesante de las EVB es que ofrecen una manera rápida de prototipar dispositivos basados en microcontroladores a un bajo coste. Por ejemplo, la STM32 Nucleo-F401RE tiene un precio de venta al público actual de unos 15 € y ya integra en la propia EVB el *debugger* (circuito electrónico necesario para programar el microcontrolador). Solo un *debugger* oficial por separado ya puede superar los 100 €. Obviamente, los fabricantes crean estas EVB para facilitarnos nuestra entrada a su ecosistema de desarrollo a un bajo coste y así lograr introducir sus microcontroladores en nuestros desarrollos. Otra ventaja es que la EVB expone todos los pines del microcontrolador de manera que facilita las conexiones con elementos externos durante la fase de prototipaje y nos ahorra tener que fabricar nuestras propias placas de prototipaje (con los que nos ahorramos un coste importante en diseño, componentes, fabricación y test).

El esquemático de la EVB lo podéis encontrar aquí. Otros tres documentos importantes, pero que aún no utilizaremos en esta práctica con Arduino, son: el *datasheet* del microcontrolador, el manual de referencia de la familia del microcontrolador y el manual de usuario de las librerías HAL (*Hardware Abstraction Layer*).

1.2.3. Ahora sí: ¡Hola, Mundo!

En estas primeras prácticas, para que todos podamos aprender las tareas más básicas, todos debéis de hacer lo indicado. Dejamos para prácticas venideras el interactuar con el trabajo de nuestro compañero de prácticas y trabajar en equipo. Así que: ¡todos debéis de hacer lo que aquí se os pide!

1.2.3.1. Clonar el repositorio En esta y todas las prácticas de la asignatura se utiliza un Sistema de Control de Versiones (VCS) para desarrollar, colaborar y entregar las prácticas y el proyecto. El detalle de cómo funcionan los VCS lo iremos viendo poco a poco en cada práctica y en más detalle en una unidad especial dedicado a ellos. Por ahora, os pedimos un pequeño salto de fe y que sigáis al pie de la letra las indicaciones a realizar con Git. Os iremos explicando los comandos y términos a medida que estos vayan apareciendo, dejando los detalles para la unidad dedicada a ellos.

Como esta es la primera práctica de todas, si no lo habéis hecho ya, veremos cómo clonar el repositorio remoto en GitHub en nuestro ordenador. Esto es, tener una copia de todos los contenidos en GitHub en nuestro ordenador y establecer un enlace al repositorio remoto en GitHub que nos permitirá rescatar versiones y ramas alojadas en el servidor y poder subir nuestros cambios. Para hacer esto, vamos a la página principal del repositorio `Biomedical-Electronics\masb-p01-<tu-grupo>` y copiamos el enlace que tenemos disponible en el botón verde `Clone or Download`. Aseguraros que copiáis el enlace mientras pone `Clone with HTTPS` y no `Clone with SSH`. Con el enlace copiado, vamos a la carpeta donde queramos que se guarde nuestro repositorio local. Allí hacemos clic derecho sobre ningún archivo ni carpeta y en el menú desplegable clicamos `Git Bash here`. Si no aparece este elemento en el menú desplegable, muy probablemente no tenéis instalado Git. Podeís instalarlo descargándoslo desde este enlace y usando el algoritmo previamente descrito de `Siguiente`, `Siguiente`, `Siguiente`, ... , y `Finalizar`.

> Es importante tener presente que **Git y GitHub son cosas diferentes**. Git es el VCS propiamente dicho y es *open source*. En cambio, GitHub es una compañia (recientemente comprada por Microsoft) que ofrece una página web que utiliza Git para posibilitar guardar el histórico de Git en un servidor remoto y poder compartir o trabajar en equipo (entre otras mil funcionalidades adicionales que iremos viéndo poco a poco).

Al hacer clic en `Git Bash here` se nos abrirá un terminal *bash*. Ahí ejecutamos el siguiente comando:

```
1 git clone <direccion-copiada>
```

Si es nuestro primer `git clone` de un repositorio de GitHub, nos pedirá nuestras credenciales. Las ingresamos (correo y contraseña de GitHub). Con todo esto, el repositorio remoto en GitHub se nos ha clonado en nuestro ordenador y podemos empezar a trabajar.

> **Importante:** al clonar o inicializar un repositorio se crea una carpeta `.git` normalmente oculta. **No eliminéis esa carpeta** o se perderá el control de versiones en vuestro repositorio local y tampoco podréis subir las prácticas.

Un último paso es ingresar nuestro nombre y correo en la configuración de Git para que estos aparezcan cuando realicemos *commits* de nuestro repositorio local. El nombre y correo se pueden indicar para solo el repositorio en cuestión o para todos. Esto último es lo que haremos. Primeramentre entramos en la carpeta que se nos habrá creado:

```
cd masb-p01-<tu-grupo>
```

Y allí introducimos:

```
git config --global user.name "<tu-nombre-y-apellidos>"
git config --global user.email "<tu-correo-electronico-en-github>"
```

Si quitásemos el `--global` del comando, estaríamos indicando el nombre y el correo solo para el repositorio en cuestión. (Ergo, no pongáis `--global` si estáis en un ordenador compartido con diferentes repositorios).

Aquí viene el primer salto de fe. Ejecutad el siguiente comando:

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

Con este comando creáis una rama con el nombre `develop/A-<tu-nombre-sin-espacios-ni-caracteres-raros>` y vais a ella. Una rama es una ramificación del repositorio donde uno puede desarrollar en paralelo lo que quiera sin que esto interfiera en el desarrollo principal o en el de otros miembros del equipo. Al final de la práctica habrán más de una rama: una para cada uno de los miembros de vuestro equipo. A partir de aquí, no cerréis el terminal ya que lo iremos usando para realizar los diferentes *commits*. (Si cerráis el terminal: *no problem*. Simplemente volved a abrirlo como antes desde dentro de la carpeta `masb-p01-<tu-grupo>`.)

1.2.3.2. Crear un *sketch*

Bien. Ya tenemos nuestro repositorio local en nuestro ordenador. Ahora vamos a crear el archivo en el que desarrollaremos nuestro programa con Arduino. Estos archivos usan el formato `.ino` y se llaman *sketches*. Para crear el archivo, simplemente, inicamos Arduino IDE y vamos a `Archivo` > `Nuevo`. Seguidamente, seleccionamos para qué EVB va a ser el programa que desarrollemos. Esto lo hacemos yendo a `Herramientas` > `Placa` y seleccionando `Nucleo-64`. Con ello hemos seleccionado la familia a la que pertenece la EVB. Ahora escogeremos el modelo de esta yendo a `Herramientas` > `Board part number` y seleccionando `Nucleo F401RE`. Lo último que debemos de hacer es seleccionar el método con el que cargaremos el programa en el microcontrolador. Esto lo hacemos yendo a `Herramientas` > `Upload method` y seleccionando `Mass Storage`. Le

damos finalmente a `Guardar como...` desde `Archivo` y guardamos el *sketch* en `./arduino` con el nombre `masb-p01`. Se nos creará una carpeta llamada `masb-p01` dentro de la carpeta `arduino` y dentro de esta un archivo `masb-p01.ino`, el archivo de programación de Arduino en sí. Un *sketch*.

> Es **muy importante** que guardéis todos los archivos en los *paths* o directorios en el que se os indica utilizando el nombre también indicado.

Delante nuestro tenemos el siguiente código:

```
1  void setup() {
2    // put your setup code here, to run once:
3
4  }
5
6  void loop() {
7    // put your main code here, to run repeatedly:
8
9  }
```

Tenemos dos funciones: `setup` y `loop`. Arduino IDE nos ha añadido comentarios que ya nos indican para qué sirve cada función. La función `setup` se ejecuta una única vez al principio del programa y es normalmente utilizado para inicializar variables, periféricos, etc. Por otro lado, la función `loop`, una vez ejecutada la función `setup`, se ejecutará contínuamente. Aquí se codifica el comportamiento deseado para el microcontrolador. Ahí vamos a crear un programa que haga parpadear el LED de la EVB cada 1 s.

1.2.3.3. Salidas digitales: *Blink the LED*

El LED está conectado al pin D13 del conector de Arduino. La `D` denota que ese pin es un GPIO digital que nosotros usaremos como salida. GPIO significa *General Purpose Input/Output* y es un pin del microcontrolador que puede realizar diferentes funciones (capturar una señal analógica, generar un PWM, programar el microcontrolador, etc.) tanto de entrada (lectura) como salida (generar una señal). En Arduino, la función del GPIO no puede ser modificada de manera directa y el pin D13 en cuestión solo puede usarse como GPIO digital. En este caso, al LED le enviaremos una señal digital que encienda y apague el LED. Cuando la señal digital de salida sea `0`, el LED estará apagado. Cuando sea `1`, el LED estará encendido.

> Hay que tener presente que en el mundo digital nos movemos en un sistema binario. Es decir, unos y ceros. Pero esto es a nivel lógico. A nivel físico, estos valores binarios de `0` y `1` se traducen a unos niveles de tensión definidos. Estos niveles de tensión suelen ser 0 V para un `0` binario y un nivel tensión igual a la tensión de la alimentación del microcontrolador para un `1` binario. En este caso, la EVB alimenta el microcontrolador con 3.3 V, por lo que un `1` binario se traduce en un nivel de tensión de 3.3 V.

Así pues, lo primero que necesitamos hacer es configurar el pin D13 como un GPIO digital de salida. Puesto que la funcionalidad/uso del pin no se modificará a lo largo del programa, solo debemos de configurarlo una vez. Por ello, la configuración la realizaremos mediante la instrucción `pinMode` dentro de la función `setup`.

```
1  void setup() {
2    // put your setup code here, to run once:
3    pinMode(13, OUTPUT); // configuramos el pin 13 como pin digital de
         salida
4    digitalWrite(13, LOW); // configuramos que al iniciarse el programa
         el LED este apagado
5  }
6
7  void loop() {
8    // put your main code here, to run repeatedly:
9
10 }
```

> Es importante (**y obligatorio en la asignatura**) comentar el código para que una tercera persona, e incluso a veces nosotros mismos, entendamos qué se está haciéndo en el programa al leerlo. No utilicéis eñes, acentos o caracteres raros en los comentarios. Si los hacéis en inglés, mucho mejor.

En la función `pinMode` hemos configurado el pin 13 como `OUTPUT` o salida. Si quisieramos configurarlo como entrada (lectura de un valor digital) indicaríamos `INPUT`. También hemos añadido una segunda instrucción, `digitalWrite`, que sirve para decir qué valor queremos que salga por el pin indicado. Si indicamos `LOW`, estaremos diciendo que genere un `0`. Si indicamos `HIGH`, estaremos diciendo que genere un `1`.

> Aviso a navegantes. Es muy tentador copiar y pegar el código del documento en lugar de escribirlo uno mismo. Avisamos de antemano que ese sería un craso error. No porque no podáis hacerlo. Sino porque no comprenderéis ni integraréis los conceptos que se estudian en las prácticas y será hartamente difícil que podáis afrontar el proyecto final por vosotros mismos.

Y ahora lo que haremos será hacer parpadear el LED. Esto se realizará continuamente durante la ejecución del programa, por lo que lo codificaremos en la función `loop`. Justo acabamos de ver la función `digitalWrite`para fijar un valor de salida. Podemos intuir que para hacer parpadear el LED solo necesitamos utilizar esta función alternando entre `LOW` y `HIGH`.

```
1  void setup() {
2    // put your setup code here, to run once:
3    pinMode(13, OUTPUT); // configuramos el pin 13 como pin digital de
         salida
4    digitalWrite(13, LOW); // configuramos que al iniciarse el programa
         el LED este apagado
5  }
```

```
6
7  void loop() {
8    // put your main code here, to run repeatedly:
9    digitalWrite(13, HIGH); // LED encendido
10   digitalWrite(13, LOW); // LED apagado
11 }
```

Compilamos dándole al icono para verificar  y, si todo está correcto, el terminal de Arduino IDE no nos debe de indicar ningún error y sí una serie de información adicional, como el espacio que ocupa el programa dentro de la memoria del microcontrolador, el máximo que podría ocupar, etc.:

```
1 El Sketch usa 12484 bytes (2%) del espacio de almacenamiento de
      programa. El máximo es 524288 bytes.
2 Las variables Globales usan 892 bytes (0%) de la memoria dinámica,
      dejando 97412 bytes para las variables locales. El máximo es 98304
      bytes.
```

Conectamos, si no lo habíamos hecho ya, la EVB al ordenador utilizando el cable USB y clicamos en el icono para subir el programa  .

Una vez cargado… ¿el LED parpadea?

A simple vista, parece que el LED no parpadea. En realidad sí lo hace, pero a tal velocidad que no podemos apreciarlo con nuestro ojo. Vamos a decirle al microcontrolador que deje pasar 1 s entre el encendido y el apagado. Para ello utilizaremos la función `delay` que nos proporciona Arduino. Esta función tiene como argumento el número de milisegundos que queremos que el microcontrolador se quede esperando. Mientras el microcontrolador espera dentro de esta función `delay`, este no hace ninguna otra cosa más. Así quedaría el código añadiendo el *delay*.

```
1  void setup() {
2    // put your setup code here, to run once:
3    pinMode(13, OUTPUT); // configuramos el pin 13 como pin digital de
         salida
4    digitalWrite(13, LOW); // configuramos que al iniciarse el programa
         el LED este apagado
5  }
6
7  void loop() {
8    // put your main code here, to run repeatedly:
9    digitalWrite(13, HIGH); // LED encendido
10   delay(1000); // esperamos 1 segundo con el LED encendido
11   digitalWrite(13, LOW); // LED apagado
12   delay(1000); // esperamos 1 segundo con el LED apagado
13 }
```

Ahora sí, compilamos, cargamos el programa y podemos comprobar como el LED parpadea cada 1 s.

1.2.3.3.1. *Commit* de la versión Tenemos un programa que funciona. Vamos a guardar esta versión para asegurarnos que siempre podremos volver a esta versión del código si tocamos algo que no deberíamos. Vamos a ejecutar la siguiente instrucción:

```
1 git add arduino/masb-p01/masb-p01.ino
```

Con esta instrucción añadimos el archivo `mas-p01.ino` al *stage*. El *stage* simplemente es una área donde se colocan los archivos cuya versión queremos guardar. Una vez estén añadidos en el *stage* los archivos cuya versión queramos guardar, haremos el *commit*. Y eso vamos a hacer ahora:

```
1 git commit -m "LEDs parpadeando cada 1 segundo."
```

El comando es sencillo y simplemente hay que añadir un mensaje descriptivo de los cambios realizados respecto el último *commit*. ¡Ya hemos guardado nuestra primera versión de código!

1.2.3.4. Entradas digitales Sabemos cómo hacer salir *ceros* y *unos* a través de los GPIOs. Ahora vamos a ver cómo leerlos. Para ello utilizaremos el pulsador/botón B1 de la EVB. De este pulsador, como podemos ver en el esquemático de la EVB, sale un `1` cuando no lo pulsamos y un `0` cuando lo hacemos. Este comportamiento se suele describir como *active-low* porque indica una acción (en este caso, el pulsado de un botón) generando un `0`. Si fuera al revés, sería (oh, sorpresa) *active-high*.

La salida del pulsador B1 está conectada al pin 23. Este pin no se encuentra por defecto en las placas originales de Arduino UNO puesto que estas no proveen de un pulsador. Por eso, si buscáis en el conector de Arduino, no encontraréis nada. Pero yo os lo chivo, va. Vamos a configurarlo como GPIO digital de entrada para hacer que el LED alterne entre apagado y encendido cuando pulsemos el botón B1. Para configurar un pin como de entrada usamos la misma función `pinMode` que hemos usado con el LED, pero esta vez indicando `INPUT`.

> Hay un tercer valor disponible para el `pinMode` que es `INPUT_PULLUP`. Con esta tercera opción, además de configurar el pin como de entrada, estaremos indicando que se habilite la resistencia de *pullup* interna del microcontrolador.

Para alternar entre un estado u otro, utilizaremos la función `digitalRead` para leer el valor de entrada del pin y utilizaremos un estamento **`if`** para establecer la lógica deseada. También podemos utilizar la función `digitalRead` para saber el estado actual del LED. Este sería el código:

```
1 void setup() {
2   // put your setup code here, to run once:
3   pinMode(13, OUTPUT); // configuramos el pin 13 como pin digital de
        salida
4   digitalWrite(13, LOW); // configuramos que al iniciarse el programa
        el LED este apagado
```

```
5   pinMode(23, INPUT); // configuramos el pin 23 como pin digital de
        entrada sin pullup
6 }
7
8 void loop() {
9   // put your main code here, to run repeatedly:
10  if (digitalRead(23) == LOW) { // si el pulsador es pulsado
11    if (digitalRead(13) == HIGH) { // si el LED esta encendido
12      digitalWrite(13, LOW); // lo apagamos
13    } else { // si no esta encendido
14      digitalWrite(13, HIGH); // lo encendemos
15    }
16  }
17 }
```

Compilamos y cargamos el programa. Pulsamos el botón y… ¿Funciona? Pulsad unas cuantas veces… Parece que sí, pero hay veces que no…, a ratos…, … , algo falla…

Una vez más, nuestros ojos y nuestra *velocidad* nos limitan frente el microcontrolador. Realmente funciona el programa que hemos escrito. ¿Dónde está el problema? Que el microcontrolador es tan rápido que durante una pulsacióin nuestra, la función `loop` se ha ejecutado ~~infinidad de veces~~ múltiples veces y el LED parpadea rápidamente sin nosotros percibirlo dejando a la aleatoriedad el que la última iteración de la función `loop`deje encendido o apagado el LED. Esto hay que solucionarlo. ¿Cómo? Haciendo que el microcontrolador no controle el LED en función del valor de entrada `0` o `1`, sino mediante la transición de `0` a `1` o de `1` a `0`. Aunque el microcontrolador sea muy rápido, solo tendrá lugar una transición cada vez que pulsemos o soltemos el botón independientemente de la duración de la pulsación. La activación por transición se conoce como *edge triggered*, mientras que, si es por nivel, se conoce como *level triggered*.

> El sistema de reloj de un microcontrolador controla la ejecución de las instrucciones del mismo. El sistema de reloj genera una señal cuadrada. El microcontrolador ejecuta una instrucción cada vez que tiene lugar una transición de `0` a `1`. Es decir, un microcontrolador trabaja internamente en modo *edge triggered*.

Para hacer una detección de transición (también se suele llamar *flanco*) en *software*, utilizaremos una variable tipo `bool`. Veamos la explicación de cómo hacerlo directamente en el código:

```
1 /*
2  * creamos una variable booleana fuera de las funciones setup y loop
3  * de este modo, la variable pasa a ser global
4  * una función global esta disponible tanto en la funcion setup como en
       la loop
5  * si hubiesemos definido la variable dentro de la funcion setup, esta
       solo podria
6  * estar disponible en la funcion setup
```

```
 * en cambio, si hubiesemos definido la variable dentro de la funcion loop,
 * esta solo podria estar disponible en la funcion loop
 */
bool highToLowTransition = false; // por defecto, su valor sera false

void setup() {
  // put your setup code here, to run once:
  pinMode(13, OUTPUT); // configuramos el pin 13 como pin digital de salida
  digitalWrite(13, LOW); // configuramos que al iniciarse el programa el LED este apagado
  pinMode(23, INPUT); // configuramos el pin 23 como pin digital de entrada sin pullup
}

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

  if (digitalRead(23) == LOW) { // si el boton es pulsado
    if (highToLowTransition == false) { // y no se habia pulsado antes

      highToLowTransition = true; // guardamos que se ha pulsado
      // y commutamos el estado del LED
      if (digitalRead(13) == HIGH) { // si el LED esta encendido
        digitalWrite(13, LOW); // lo apagamos
      } else { // si no esta encendido
        digitalWrite(13, HIGH); // lo encendemos
      }
    }

  } else { // si en cambio, el boton no esta pulsado
     highToLowTransition = false; // reiniciamos la variable a false
  }
}
```

Probamos esta vez y… ¡bingo! El programa funciona tal y como queríamos. Del código anterior cabe destacar la propiedad *scope* de las variables. En este caso, de la variable `highToLowTransition`. El *scope* de una variable define dónde se encuentra disponible para ser leída o modificada. Una variable está disponible dentro de la función donde esta es definida. Si la variable es definida fuera de cualquier función, se le conoce como una variable global y esta queda disponible para todas las funciones.

1.2.3.4.1. *Commit* de la versión

Tenemos un programa funcionando correctamente. Guardemos esta versión:

```
git add arduino/masb-p01/masb-p01.ino
git commit -m "El LED se apaga y enciende mediante el pulsador B1."
```

1.3. Reto

Ahora vamos a proponer un paso más allá que deberéis de realizar por vosotros mismos, un reto. En este caso, el reto consiste en hacer que el LED parpadee o deje de parpadear cada vez que pulsemos el botón B1. El LED debe de empezar inicialmente apagado. Cuando pulsemos el botón B1 debe de empezar a parpadear cada 500 ms. Cuando volvamos a pulsar el botón B1 el LED debe de volver a apagarse. Con lo que sabemos hacer hasta ahora, para que el programa os funcione correctamente, debéis de mantener pulsado el botón hasta 1 segundo para que el LED se apague siempre (y no de manera aleatoria). **¿Por qué?** (Veremos como solucionarlo en la próxima práctica.)

Habeís visto cómo apagar y encender un LED, cómo hacerlo parpadear, cómo utilizar las entradas digitales mediante *edge triggered*,… ¡Si ya casi lo tenéis!

1.3.1. *Commit* de la versión y *push* de la rama

¡No olvidéis hacer el pertinente `git add` y `git commit` para guardar la versión del código con el reto realizado! Cuando ya tengáis el *commit* realizado, ejecutaremos esta instrucción:

```
1 git push
```

Esta instrucción lo que hace es subir los cambios del repositorio local al repositorio remoto en GitHub. Así podéis compartir los cambios realizados con el resto de los miembros del equipo (y con el profesorado… ¡así que es un paso vital!). Si se crea en el repositorio local una rama que aún no existe en el repositorio remoto, al hacer *push* nos informará de ello y nos pedirá que usemos una instrucción especial. Copiad/pegad esa instrucción que el propio Git os da cuando hacéis el *push* y ejecutadla en el terminal:

```
1  git push --set-upstream origin develop/A-<tu-nombre-sin-espacios-ni-
      caracteres-raros>
```

Una vez ejecutado, puedes comprobar que se han subido bien los cambios yendo a la página web de tu repositorio en GitHub. Por defecto, la página muestra la rama `master`. Acuérdate de hacer que se muestre la rama `develop/A-<tu-nombre-sin-espacios-ni-caracteres-raros>` desde el desplegable `Branch` arriba a la izquierda.

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 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**

Un informe escrito en la misma carpeta o directorio que este documento `README.md` con el nombre `REPORT.md`. El documento debe de escribirse utilizando el lenguaje de marcado *markdown*. En este enlace tenéis información sobre los marcajes básicos de *markdown* en GitHub. Aquí podéis encontrar diferentes editores que os dejarán ver qué forma va cogiendo vuestro informe a medida que váis escribiendo. Este documento lo hemos escrito con este editor. La estructura del informe es libre (pero usando el sentido común…:unamused:), siendo solo obligatorio la introducción de una tabla de contenidos al principio del informe, como la del presente documento. El formato a seguir es el de una página web, como el presente documento (imágenes, encabezados, enlaces a videos, a infografías, a referencias, a recursos multimedia, etc.), pero con un **uso formal y técnico del lenguaje** (lo sentimos profundamente… no os dejaremos usar emoticonos en los informes :stuck*out_tongue_winking_eye:). Si queréis añadir imágenes, guardadlas en la carpeta `assets`. Por cierto, ¡acuérdate de hacer un 4rto _commit* y un *push* una vez finalices el informe para que esté disponible en GitHub!

El informe debe de contener:

- ☐ Qué es Git y para qué sirve.
- ☐ Los comandos básicos de Git utilizados en esta práctica y para qué sirven.
- ☐ Qué es GitHub.
- ☐ Qué es Arduino.
- ☐ Tabla con las funciones de Arduino vistas en la práctica así como una explicación de qué hacen y cómo se utilizan.
- ☐ Pregunta del reto: ¿Por qué hace falta mantener pulsado el botón hasta 1 segundo para que siempre que queramos apagar el LED?

> A estas alturas de la película no habría que decirlo, pero al redactar/realizar cualquier tipo de documento (ya sea texto, imagenes, vídeo, etc.), aseguraros de no cruzar la **gruesa** línea que separa la copia/plagio de la citación/referencia. Lo mismo aplica en lo referente a la copia/préstamo en informes o código entre vosotros. Si se detectase uno de estos casos, se evaluaría la práctica con un 0. Si se detectara un segundo caso, ya hablaríamos de un suspenso de la asignatura. A lo indicado en el Plan Docente nos remitimos.

1.4.2. Pull request

Una manera que ofrece GitHub para incorporar el código de una rama a otra (*merge*) es mediante un *Pull Request*. La rama `master` es la rama que contiene el código de producción, el código final, el código que siempre debe de funcionar. El resto de ramas que se puedan crear se utilizan para desarrollar código que luego acabará incorporándose a la rama `master`. Por ello, una vez hayáis hecho todo lo indicado en el apartado de Entregables, haremos un *Pull Request* para incorporar el trabajo hecho a la rama `master`.

Para hacer un *Pull Request* (PR) vamos a la página web en GitHub de nuestro repositorio y vamos al apartado *Pull Requests*. Allí clicamos sobre el botón verde `New pull request`. En el primer recuadro nos aseguramos que esté seleccionada la rama `master`(rama de destino) y en el segundo recuadro nuestra rama `develop`/`A-<tu-nombre-sin-espacios-ni-caracteres-raros>` (rama de donde cogeremos los cambios a incorporar). Échale un vistazo a toda la información que te da la página (todos los cambios que se incorporarán en la rama `master`). Finalmente, clica sobre el botón `Create pull request`.

En un escenario real, un tercero (o a veces incluso nosotros mismos) revisaria que los cambios son los pertinentes y los aceptaría para que fueran incorporados a la rama `master`. Nosotros lo dejaremos así, con los cambios pendientes de ser aceptados.

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 primera práctica hemos tenido un primer contacto con la programación de microcontroladores.

Hemos visto que cualquier desarrollo debe de acompañarse de un sistema de control de versiones que garantice que siempre se puede disponer de código validado y funcional. En particular, hemos visto el uso de Git. Un VCS *open source* con el que hemos clonado un repositorio remoto de GitHub en nuestro ordenador, hemos guardado diferentes versiones del código con *commits* y hemos compartido el código agregando los cambios realizados al repositorio remoto mediante un *push*.

En este código que hemos creado, hemos utilizado entradas digitales para generar y leer señales digitales a través de los GPIOs digitales. También hemos visto cuál es la estructura básica de un

programa de Arduino (*sketch*) y cómo debemos de configurar Arduino IDE para poder trabajar con la EVB de STM.

Finalmente, hemos visto algunas técnicas basadas en *software* para generar tiempos de espera en el programa o para solucionar problemas con la lectura constante de una señal digital generada por un pulsador.