Escuela Técnica Superior de
Ingenieros de Telecomunicación

Tutoriales
sobre los fundamentos teóricos de sistemas basados en microcontrolador STM32
-
Sistemas Digitales II
Sistemas Electrónicos

Escuela Técnica Superior de Ingenieros de Telecomunicación

Tutoriales sobre los fundamentos teóricos de sistemas basados en microcontrolador STM32

-

Sistemas Digitales II
Sistemas Electrónicos

Autores:
Román Cárdenas Rodríguez
Josué Pagán Ortiz
Alberto Boscá Mojena
Iván Martín Fernández
Sergio Esteban Romero
Departamento de Ingeniería Electrónica
Contacto:r.cardenas@upm.es, j.pagan@upm.es

Versión 2.0.1

Prefacio

Este documento contiene los tutoriales sobre el uso de la placa **Nucleo-STM32F446RE** explicada en el libro *"Fundamentos teóricos de sistemas basados en microcontrolador STM32"* [1], y el uso de Finite State Machine (Máquina de Estados Finitos) (FSM) para familiarizarse con el entorno y la metodología de trabajo presentados en el libro. Se divide en:

(I) un ejemplo básico guiado para hacer parpadear un Light-Emitting Diode (Diodo Emisor de Luz) (LED) (Capítulo 1),

(II) una introducción a las máquinas de estados (Capítulo 2),

(III) y un ejemplo avanzado de máquinas de estados combinadas (Capítulo 3).

Índice general

Índice de figuras

Lista de acrónimos

API Application Programming Interface (Interfaz de Programación de Aplicaciones) 7

FSM Finite State Machine (Máquina de Estados Finitos) v

FPU Floating Point Unit (Unidad de Punto Flotante) 3

GPIO General Purpose Input/Output (Entrada/Salida de Propósito General) . 7

ISR Interrupt Service Routine (Rutina de Servicio/Atención a la Interrupción) 4

LED Light-Emitting Diode (Diodo Emisor de Luz) v

Capítulo 1

Ejemplo de proyecto: Blink

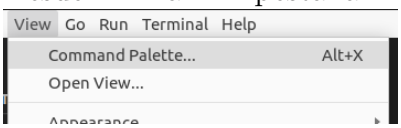
Josué Pagán, Román Cárdenas, Alberto Boscá, Iván Martín, Sergio Esteban

Este capítulo ilustra cómo comenzar un proyecto con el entorno MatrixMCU, asumiendo ya la instalación del entorno descrita en la *"Guía de instalación de herramientas para compilación cruzada en C"* [2] y la descarga de la *toolkit* descrito en el Capítulo *"Desarrollando para **Nucleo-STM32**"* [1].

Como primer ejemplo, se implementará el parpadeo periódico del LED integrado en nuestra placa. El primer paso será clonar el repositorio de GitHub del tutorial (`https://github.com/sdg2DieUpm/tutorial`).

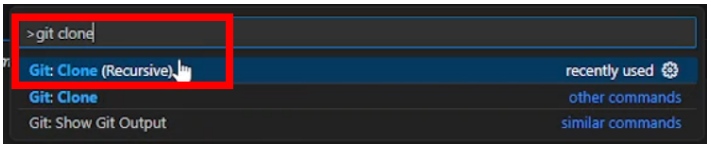
1. Abrimos VSCode y abrimos la paleta de comandos. Recuerde:

   - Desde la pestaña "View → Command Palette":

   - Presionando Alt+X
   - Presionando F1
   - Presionando Ctrl+Shift+P

   A continuación escribimos *"Git: Clone"*.

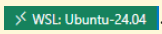
   

2. Inmediatamente, nos pide la URL del repositorio. Escribimos o pegamos `https://github.com/sdg2DieUpm/tutorial` y pulsamos *enter*.

3. Nos pedirá la carpeta donde queremos clonar el repositorio. Seleccionamos la carpeta `MatrixMCU/projects`, a la misma altura que `project_template`.

Una vez descargado el repositorio, abriremos la carpeta `MatrixMCU/projects/tutorial` como un nuevo proyecto en Visual Studio Code, el editor con el que trabajaremos.

ATENCIÓN

Si está utilizando **Windows**, asegúrese de que VSCode se está ejecutando en WSL y en la distribución correcta de Ubuntu (`Ubuntu-24.04`), ya que es donde tenemos las herramientas de desarrollo. Puede comprobarlo fijándose en la esquina inferior izquierda de su ventana de VSCode, donde debería ver algo como `WSL:Ubuntu-24.04` 

. De no ser así, active la *"Command Palette"* presionando la tecla "F1" o utilizando el atajo de teclado "Ctrl + Shift + P". Busque y seleccione la opción ``WSL: Reopen folder in WSL'' y seleccione ``Ubuntu-24.04''.

NOTA

Para que la función de autocompletado *Intellisense* de Visual Studio Code reconozca las rutas de los ficheros y funcione correctamente, seleccione el entorno en el que está trabajando en la parte inferior derecha del editor (`C/C++ Configuration`). Por ejemplo, si está en Ubuntu y está programando para el microcontrolador **STM32F446RE** seleccione `linux-stm32f4`. Si estuviese programando en MacOS con procesador x86, eligiría `macos-intel-stm32f4`. Como caso particular, **si está utilizando Windows con WSL** debe seleccionar `linux-stm32f4`, ya que el entorno de programación y las herramientas de compilación se encuentran en el subsistema Ubuntu.

1.1. Funciones de inicialización del sistema

Como se comenta en la Sección de "Arranque del sistema" [1], antes de ir a la función `main` el microcontrolador debe realizar el *boot* del sistema. Si abrimos el fichero de cabecera `port/include/port_system.h`, veremos lo siguiente:

```
1 /* Initializes the system */
2 uint32_t port_system_init(void);
```

Esto es una declaración de una función. Normalmente, los ficheros cabecera se componen de declaraciones de funciones. Una declaración de función puede considerarse

un *contrato* entre la persona que implementa la función declarada y la persona que usa la función. En este caso, si atendemos a la documentación que acompaña a la declaración, el usuario **asume** que existe una función llamada `port_system_init` y que, cuando quiera inicializar el sistema, bastará con ejecutarla. Del mismo modo, la persona que implemente la función **debe garantizar** que la función hace **exactamente lo que promete**: ni más ni menos.

NOTA

Si echamos un vistazo a todas las funciones declaradas en `port_system.h`, veremos que ninguna da detalles sobre la plataforma donde se ejecutará nuestro código. A priori, todas ellas valen para cualquier dispositivo.

Los detalles de implementación de la función `port_system_init` dependen de la plataforma sobre la que estemos desarrollando: no es lo mismo un PC con el sistema operativo Windows que una placa **STM32F446RE** sin ningún sistema operativo. En el caso de nuestra placa **STM32F446RE**, las funciones de inicialización no son evidentes y realizan muchas configuraciones de bajo nivel. Por tanto, se le proporcionan ya codificadas. Todas se encuentran en el fichero `port/stm32f4/src/stm32f4_system.c`. Estas son:

- `SystemInit()`: es llamada directamente por `Reset_Handler` del fichero `startup_stm32f446xx.s`. En nuestra implementación, inicializa la Floating Point Unit (Unidad de Punto Flotante) (FPU) (si se usa), configura la memoria externa (si la hay) y recoloca la tabla de vectores de interrupción (si es que la modificamos). En nuestro caso, ninguna de estas tres configuraciones se da.

- `port_system_init()`: esta función inicializa los periféricos, la memoria *flash* y llama a la función de configuración del reloj `system_clock_config()`. **IMPORTANTE: esta llamada debemos hacerla nosotros al inicio del programa antes de configurar cualquier periférico. Si no la hacemos, no funcionará nada que tenga que ver con el HW**.

- `system_clock_config()`: Esta función inicializa el oscilador interno `HSI` a 16 MHz (valor puesto en el `#define HSI_VALUE`). Esta función también gestiona la alimentación y configura **el temporizador de sistema `SysTick` a** 1 ms**. IMPORTANTE: esta función, por seguridad, no puede ser accedida desde el exterior** por lo que tiene el modificador `static`, el cual previene que la función se use en otro fichero.

Pongamos atención en la función `port_system_init`. Tiene exactamente el mismo nombre y argumentos que la función que habíamos declarado en `port/include/port_system.h`. Sin embargo, ahora estamos **definiendo** la lógica exacta que hay que ejecutar si queremos inicializar una placa **STM32F446RE**. Como esta función hace referencia a registros específicos de esta familia de placas, la hemos ubicado dentro de la carpeta `port/stm32f4`. Dentro de esta carpeta guardaremos el código específico de las placas STM32F4.

NOTA

En este caso, los profesores de SDG2 somos *implementadores* de la función `port_system_init`, y **garantizamos** que la función existe y, tras llamar a esta función, la placa estará preparada para ejecutar cualquier aplicación. En cambio, los alumnos actuáis de *usuarios* de esta función, la cual está a vuestra disposición para que la **uséis** cuando queráis inicializar el sistema.

1.1.1. Referencia temporal del sistema: SysTick

El `SysTick` es el temporizador de referencia del sistema, ya configurado para generar una interrupción interna cada 1 ms. Nosotros debemos decidir qué hacer cada vez que se genere dicha interrupción. En nuestro caso, decidimos actualizar **una variable que lleve la cuenta de las veces que se ha interrumpido cada** 1 ms, `msTicks`.

Conforme está el proyecto `tutorial_1`, si intenta depurarlo (`Ejecución y depuración (panel izquierdo)` → `Clean and Debug (stm32f446re)`→ `target: main`), le dará el error que se muestra en la Figura 1.1.

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS  MEMORY  XRTOS  MONITOR SERIE  COMENTARIOS

'msTicks' defined but not used [-Werror=unused-variable]
   24 | static  uint32_t msTicks = 0; /*!< Variable to store millisecond ticks. @warning **It must be declared
volatile!** Just because it is modified in an ISR. **Add it to the definition** after *static*. */
      |                 ^~~~~~~
cc1.exe: all warnings being treated as errors
```

Figura 1.1: Error de variable en desuso.

El error indica que está en la línea 33 del fichero `smt32f4_system.c`, y nos dice que la variable `msTicks` está definida, pero no se usa. Hemos de comprender el acceso externo a variables locales (funciones *set* y *get*) y el funcionamiento de las Interrupt Service Routine (Rutina de Servicio/Atención a la Interrupción) (ISR) para poder usarla y resolver este error.

Queremos que la variable `msTicks` lleve la cuenta (ticks) en milisegundos del tiempo del sistema. Su comportamiento esperado es que, en cada interrupción del reloj del sistema `SysTick`, aumente su valor en 1. Además, se ha de añadir el modificador `volatile`, dado que se va a utilizar en una ISR, quedando así:

```
1 static volatile uint32_t msTicks = 0;
```

El modificador `volatile` previene al compilador de realizar optimizaciones sobre el acceso a esta variable (por ejemplo, reordenación de instrucciones) para evitar comportamientos anómalos.

Puesto que las interrupciones podemos deshabilitarlas —como en los modos de bajo consumo—, **el valor almacenado no será un valor absoluto desde que se inició el sistema, sino un valor que podremos tener en cuenta de forma relativa para contar lapsos de tiempo.**

La función `SysTick_Handler`, ISR del reloj `SysTick`, se encuentra en el fichero `port/stm32f4/src/interr.c`. Este fichero alojará todas las funciones de interrupción de nuestro sistema, ya sea de pines u otros temporizadores. Pongamos directamente el siguiente código (erróneo) y depuremos:

```
1 void SysTick_Handler(void)
2 {
3     msTicks += 1; /* Equivalente a: msTicks = msTicks + 1;*/
4 }
```

En la terminal debe aparecer el error *'msTicks' undeclared*, distinto al anterior. Se debe a que la variable está declarada en el fichero `stm32f4_system.c`, pero no es accesible desde `interr.c`. Para acceder a la variable modificaremos las funciones *set* y *get* del recurso, localizadas al final del fichero `stm32f4_system.c`. Sabemos que la función *get* ha de devolver un entero de 32 bits sin signo (`uint32_t`), como `msTicks`, y que no recibe ningún parámetro (`void`), y que la función *set* recibe un parámetro del tipo de `msTicks` pero no devuelve nada, solo modifica la variable `msTicks`. Se puede poner de la siguiente manera:

```
1 uint32_t port_system_get_millis()
2 {
3   return msTicks;
4 }
5
6 void port_system_set_millis(uint32_t ms)
7 {
8   msTicks = ms;
9 }
```

Este tipo de funciones se han de crear cada vez que se deba leer o modificar un recurso compartido del sistema, como en este caso es `msTicks`.

De forma sencilla, ya podemos modificar la función ISR `SysTick_Handler` haciendo uso de las dos funciones anteriores, usándolas para aumentar la cuenta en 1 ms:

```
1 void SysTick_Handler(void)
```

```
4   #include "port_system.h"
5   #include "port_led.h"
6   #include <fsm.h>
7
8   #define BLINK_T_MS 2000
9
10  /**
11   * @brief main routine
12   *
13   *  * > **TO-DO alumnos:**
14   * >
15   * > ✅ 1. Initialize the system \n
16   * > ✅ 2. Initialize the LED GPIO \n
17   * > ✅ 3. Infinite loop that toggles the LED value. The toggling period must be T_LED_MS \n
18   * > ✅ 4. Even thought it never returns, it is good practice to return 0 at the end of the
     function.
19   *
20   * @return this function never returns.
21   */
22
23  int main()
24  {
25      return 0;
26  }
27
```

Figura 1.2: Primera ejecución en modo depuración.

```
2 {
3   port_system_set_millis(port_system_get_millis() + 1);
4 }
```

Si volvemos a depurar (`Ejecución y depuración (panel izquierdo)` → `Debug (stm32f446re)` → `target: main`), vemos que ya no aparecen problemas durante la compilación y se crea el ejecutable `main.elf` en la carpeta `/bin/stm32f446re/Debug`.

NOTA

Si estás usando Windows con WSL, tienes que hacer *Attach* de la placa antes de depurar. Si no, el subsistema Linux no va a ser capaz de conectarse al depurador de tu placa.

El entorno nos llevará a la primera línea de la función `main` del archivo `main.c`, en el directorio raíz del proyecto.

Para conseguir la funcionalidad más básica del proyecto (parpadeo del LED), todavía tendremos que inicializar el pin del led y rellenar la función `main()` con las inicializaciones y el bucle que enciende y apaga el LED.

NOTA

En este punto se habrá dado cuenta de que tanto las funciones como los archivos se nombran siguiendo ciertas convenciones de nomenclatura. Esto se hace así para que sea más fácil identificar de dónde vienen las funciones y qué hacen solo con su nombre.

En general, el directorio `common` incluye el código que controla el sistema central, independiente del HW, mientras que el directorio `port` aloja el código "portable" o dependiente del HW. En el directorio `port/include` se incluyen las cabeceras de todas las funciones de control de HW que serán necesarias, independientemente de la placa que se utilice. Sin embargo, la implementación de estas funciones se debe realizar para cada sistema en carpetas y ficheros distintos, identificadas con el nombre de la placa para la que se programa. Así, si queremos que nuestro sistema pueda encender un LED de la `stm32f4`, tendremos que declarar la función `port_led_on` en la cabecera `port/include/port_led.h`, pero escribiremos el código que implementa esta funcionalidad en `port/stm32f4/src/stm32f4_led.c`

Se ha de ceñir a un estilo a la hora de programar. Se recomienda ojear el libro de estilo "Embedded C Coding Standard" [3][a].

[a] El libro es distribuido gratuitamente por los autores en `https://barrgroup.com/sites/default/files/barr_c_coding_standard_2018.pdf`. Accedido: 2024-01-15.

1.2. Interfaz para interactuar con el LED

A continuación, debemos implementar todas las funciones relacionadas con la interacción con el LED, que dependerá considerablemente de la plataforma en uso (en este caso, **STM32F446RE**). Por lo tanto, todas las funciones estarán definidas e implementadas en la carpeta `/port/stm32f4/`. En este tutorial, se proporciona la Application Programming Interface (Interfaz de Programación de Aplicaciones) (API) que debemos implementar para interactuar con el LED. Puede consultar esta API en la cabecera `/port/stm32f4/include/port_led.h`. Nos fijaremos en las funciones de este fichero:

- `void port_led_gpio_setup(void)`: Esta función debe configurar el pin General Purpose Input/Output (Entrada/Salida de Propósito General) (GPIO) correspondiente al LED (*i.e.*, puerto de GPIO *A*, pin 5) para que actúe como una salida sin resistencias de *pull up* ni *pull down*.
- `bool port_led_get(void)`: Se lee el estado del LED (encendido/apagado) en el registro del pin y se devuelve como variable tipo `bool`.

- `port_led_on(void)`: Se enciende el LED.
- `port_led_off(void)`: Se apaga el LED.
- `void port_led_toggle(void)`: Esta función debe modificar el estado del LED. Si está apagado, lo encenderá. En caso contrario, lo apagará.

Como es costumbre, la implementación de estas funciones deberá realizarse en el fichero fuente `/port/stm32f4/src/stm32f4_led.c`. Este fichero contiene lo siguiente:

```
1  /* Standard C includes */
2  #include <stdio.h>
3
4  /* HW independent includes */
5  #include "port_system.h"
6  #include "port_led.h"
7
8  /* HW dependent includes */
9  #include "stm32f4xx.h"
10 #include "stm32f4_system.h"
11
12 // HW Nucleo-STM32F446RE:
13 #define LD2_PIN 5           /*!< GPIO pin of the LED2 in the Nucleo board */
14 #define LD2_GPIO_PORT GPIOA /*!< GPIO port of the LED2 in the Nucleo board */
15
16 #define MODER5_MASK (GPIO_MODER_MODE0 << LD2_PIN * 2) /*!< Mask for the LED2 in the
       MODE Register */
17 #define PUPDR5_MASK (GPIO_PUPDR_PUPD0 << LD2_PIN * 2) /*!< Mask for the LED2 in the
       PUPD Register */
18
19 #define MODER5_AS_OUTPUT (STM32F4_GPIO_MODE_OUT << LD2_PIN * 2) /*!< Output mode
      for the LED2 in the MODE Register */
20 #define PUPDR5_AS_NOPUPD (STM32F4_GPIO_PUPDR_NOPULL << LD2_PIN * 2) /*!< No push/
      pull configuration for the LED2 in the MODE Register */
21
22 #define IDR5_MASK (GPIO_IDR_ID0 << LD2_PIN) /*!< Mask for the LED2 in the Input
      Data Register */
23 #define ODR5_MASK (GPIO_ODR_OD0 << LD2_PIN) /*!< Mask for the LED2 in the Output
      Data Register */
```

Primero, se incluyen cabeceras de utilidad para la implementación de las funciones. A continuación, definimos el pin y el puerto específicos de nuestra placa en el que se ubica el LED 2, que es el que queremos controlar (`LD2_PIN` y `LD2_GPIO_PORT`). Después, especificamos las máscaras para configurar los registros `MODER` y `PUPDR` solo para el pin del LED (`MODER5_MASK` y `PUPDR5_MASK`, respectivamente). A continuación, definimos qué valores debemos escribir en estos registros para que el pin asociado al LED actúe como salida (`MODER5_AS_OUTPUT`) sin resistencias de *pull up* o *pull down* (`PUPDR5_AS_NOPUPD`). Tiene más información sobre estos registros y sus valores de configuración en las secciones correspondientes del libro [1]. Además, se proporcionan las máscaras para leer el estado del pin GPIO del registro `IDR` y para modificar el estado de este pin en el registro `ODR` (`IDR5_MASK` y

`ODR5_MASK`, respectivamente). Puede encontrar más información sobre estos registros en las secciones correspondientes del libro [1].

IMPORTANTE

Es VITAL para con este entorno aprender a manejarse con registros, máscaras y desplazamientos a nivel de bit. Analice las definiciones proporcionadas y comprenda sus valores y por qué son útiles para el desarrollo de la interfaz del LED. En caso de cualquier duda, transmítasela a su profesor.

En primer lugar, implementaremos la función para configurar el pin GPIO del LED, `port_led_gpio_setup()`. A continuación se proporciona el código necesario para que el LED actúe como una salida sin resistencia de *pull up* o *pull down*:

```
1  void port_led_gpio_setup(void)
2  {
3      /* Primero , habilitamos siempre el reloj de los perifericos */
4      RCC -> AHB1ENR |= RCC_AHB1ENR_GPIOAEN ;
5      /* Luego , limpiamos los registros MODER y PUPDR correspondientes */
6      LD2_GPIO_PORT -> MODER &= ~ MODER5_MASK ;
7      LD2_GPIO_PORT -> PUPDR &= ~ PUPDR5_MASK ;
8      /* Finalmente , configuramos el LED como salida sin pull up/ pull down */
9      LD2_GPIO_PORT -> MODER |= MODER5_AS_OUTPUT ;
10     LD2_GPIO_PORT -> PUPDR |= PUPDR5_AS_NOPUPD ;
11 }
```

Lo primero que se hace es, **activar el reloj del periférico** `GPIOA` (línea 4). Si no lo hacemos, el puerto A del GPIO del sistema permanecerá desactivado. A continuación, ponemos todos los bits relacionados con el pin 5 del puerto A de GPIO a 0 para los registros `MODER` y `PUPDR` (líneas 6 y 7). Hacer esto es importante para evitar que quede activo algún bit indeseado debido a una configuración previa. Finalmente, configuramos estos mismos registros para que el pin actúe como deseamos (líneas 9 y 10).

IMPORTANTE

Solo debemos modificar los bits correspondientes al pin 5. En caso contrario, podríamos desconfigurar pines GPIO que se estén usando en otras partes del código. Estudie las operaciones a nivel de bit que se emplean e intente comprenderlas. De nuevo, si tiene cualquier duda, transmítasela a su profesor.

Seguiremos con la función que lee el estado del LED, `port_led_get`. Esta función accede al registro que almacena este estado, el Input Data Register (IDR). A continuación, haciendo uso de la máscara del LED, nos quedamos con la información relativa al estado del LED, desechando el estado del resto de pines GPIO.

Finalmente, devolvemos el estado del LED convertido al tipo `bool`. A continuación se detalla su implementación:

```
1 bool port_led_get(void)
2 {
3     return (LD2_GPIO_PORT->IDR & IDR5_MASK) != 0; // Return the state of the LED,
          which is the value of the IDR register
4 }
```

Para poder encender o apagar el LED podemos implementar las funciones `port_led_on` y `port_led_off`. Éstas se encargarán de escribir en el bit correspondiente al LED del registro de salida, Output Data Register (ODR). Encenderemos y apagaremos escribiendo un 1 y un 0 lógicos respectivamente, como se puede ver en la implementación:

```
1 void port_led_on(void)
2 {
3     LD2_GPIO_PORT->ODR |= ODR5_MASK; // Set the corresponding bit in the ODR
          register by writing a logic 1
4 }
5 void port_led_off(void)
6 {
7     LD2_GPIO_PORT->ODR &= ~ODR5_MASK; // Clear the corresponding bit in the ODR
          register by writing a logic 0
8 }
```

De nuevo, hacemos uso extensivo de operaciones lógicas a nivel de bit para interactuar con los registros `IDR` y `ODR`. Repase la estructura de estos registros en las secciones correspondientes del libro [1] para tener una mayor comprensión de la implementación presentada.

1.2.1. Ejercicio: implementar la función `port_led_toggle`

Por último, será común que nuestra aplicación quiera alternar el estado del LED en vez de forzar una posición concreta. Para ello, podemos escribir una función que lea el valor actual del pin del IDR y escriba el contrario en el ODR y llamarla `port_led_toggle`. Como ya hemos implementado funciones que realizan estas operaciones, podemos hacer uso de ellas. Por lo tanto, se deja como ejercicio escribir el código de esta función `toggle`.

1.3. Función principal: main

En un proyecto de C, escribiremos la lógica de nuestro programa en la función `main` que se encuentra en la raíz del proyecto `main.c`. Como puede observar, esta función únicamente devuelve 0. Por lo tanto, nuestro programa no está ejecutando nada. La documentación Doxygen nos guía sobre los próximos pasos a seguir para hacer parpaderar el LED:

- Primero debemos inicializar el sistema mediante la función `port_system_init()`.
- A continuación, configuraremos debidamente el pin GPIO del LED mediante la función `port_led_gpio_setup()`.
- Finalmente, abriremos un bucle infinito que haga parpadear el LED llamando a la función `port_led_toggle()`. Como queremos que el periodo del parpadeo sea `BLINK_T_MS` milisegundos, haremos una espera activa de `BLINK_T_MS / 2` milisegundos antes de volver a cambiar el estado del LED.

A continuación, ilustramos cómo se implementaría esta función:

```
1  int main()
2  {
3      port_system_init();     // inicializamos el sistema
4      port_led_gpio_setup(); // Configuramos el GPIO para el LED
5
6      uint32_t t = port_system_get_millis(); // cuenta del tiempo actual
7      while (1)
8      {
9          port_led_toggle();                              // Hacemos parpadear el LED
10         port_system_delay_until_ms(&t, BLINK_T_MS / 2); // Y esperamos el periodo
11     }
12     return 0;
13 }
```

El programa ya no solo compila correctamente, sino que su comportamiento es el que deseamos. Ya podemos ejecutar la aplicación en nuestra placa **Nucleo-STM32**. (`Ejecución y depuración (panel izquierdo)` → `Clean and Debug (stm32f446re)` → `target: main`)

Recuerde que el depurador siempre detiene la ejecución al principio de la función `main`)). La línea marcada en ámbar todavía no ha sido ejecutada. Deberá hacer click en `Continue` para que el programa siga ejecutándose. Si todo ha ido bien, verá que el LED de su placa parpadea con un periodo de, aproximadamente, 2 segundos.

Capítulo 2

Introducción a las máquinas de estados en C

Román Cárdenas, Iván Martín, Sergio Esteban

2.1. Introducción a las máquinas de estados

Las **máquinas de estados** o **autómatas** son formalismos que permiten modelar (e incluso verificar u optimizar automáticamente) la solución a numerosos problemas de ingeniería. Una máquina de estados se puede implementar por medio de hardware específico o por medio de un programa que se ejecute en un sistema basado en un microprocesador.

Las máquinas de estados definen el conjunto de estados en los que se puede encontrar el sistema, uno de los cuales actúa como estado inicial. Periódicamente se comprueba el valor de las entradas del sistema, y en función del estado actual y del valor de dichas entradas, se calcula cuál debe ser el siguiente estado del sistema. En este tutorial emplearemos **máquinas de estados de Mealy síncronas**[1], que se caracterizan porque las salidas que produce la máquina dependen del estado actual y de las entradas recibidas (es decir, dependen de las transiciones entre estados), y **solo se comprueban las entradas y se actualizan el estado y las salidas cuando lo indica el reloj de la máquina**. Una máquina de estados, por tanto, se define como:

1. Un conjunto de entradas al sistema.

2. Un conjunto de salidas del sistema.

3. Un conjunto de estados del sistema.

[1]Como contrapartida existen las máquinas de estados de Moore, en las que las salidas dependen únicamente del estado actual del sistema.

4. Un conjunto de transiciones entre estados.

Las transiciones se representan mediante un diagrama de bolas que sigue la estructura de la Figura 2.1.

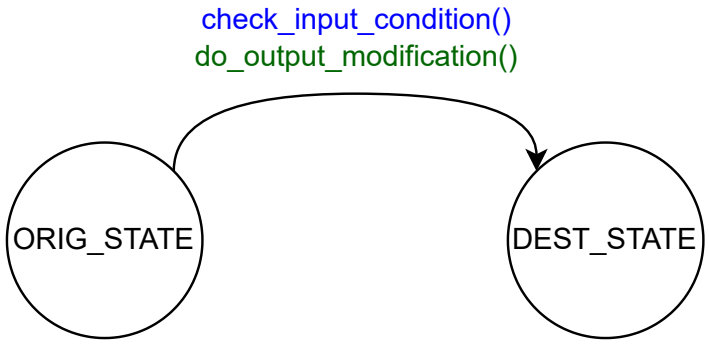


Figura 2.1: Estructura de un diagrama de bolas de una máquina de Mealy.

Los círculos representan estados del sistema y las líneas que unen estados representan una transición de un estado a otro. En cada transición hay una condición (llamada `check_input_condition()` en el diagrama) que deben cumplir las entradas del sistema para que provoque que el autómata modifique las salidas del sistema (modificación llamada `do_output_modification()` en el diagrama) y que además transite de un estado `ORIG_STATE` a otro `DEST_STATE`.

2.2. Blink usando la librería FSM

IMPORTANTE

Este tutorial se plantea a modo de *extensión* del tutorial anterior (Capítulo 1). **Antes de continuar, complete dicho tutorial.**

La *toolkit* MatrixMCU proporciona una librería software que nos facilitará la implementación de nuestra máquina de estados. Se trata de la librería FSM, que se encuentra en la carpeta `/MatrixMCU/lib/fsm`.

A modo de ejemplo, modificaremos el ejemplo de parpadeo LED del Capítulo 1 para que el comportamiento esté definido por un autómata sencillo. Este autómata de Mealy tiene un único estado *IDLE* y presenta una transición con su correspondiente condición de entrada y modificación de salidas: si estando en el estado *IDLE* ha pasado suficiente tiempo (*i.e.*, la función `check_timeout()` devuelve `true`), la máquina de estados ejecutará la función de modificación de salidas `do_toggle()` y se moverá al estado *IDLE*. La Figura 2.2 ilustra el diagrama de bolas de esta máquina de estados.

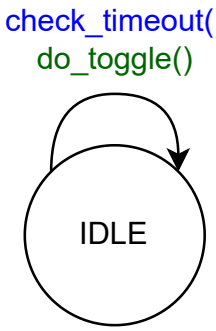


Figura 2.2: Ejemplo de autómata que hace parpadear un LED.

El primer paso será crear un nuevo proyecto en la carpeta `/MatrixMCU/projects`. Para ello duplique el `tutorial` y renómbrelo como `tutorial_2`. **IMPORTANTE: Recuerde utilizar los ficheros `stm32f4_system.c` y `stm32f4_led.c` del tutorial anterior ya completados**. Abra la carpeta `/MatrixMCU/projects/tutorial_2` en VSCode para empezar este segundo tutorial.

NOTA

Si estás usando Windows con WSL, Recuerda abrir el proyecto **en WSL**.

Descargue los ficheros `fsm_blink.h` y `fsm_blink.c` del repositorio del segundo tutorial en `https://github.com/sdg2DieUpm/tutorial/tree/tutorial_2`. Coloque estos ficheros en las carpetas `common/include` y `common/src` del nuevo proyecto, respectivamente. Estos ficheros son la cabecera y fuente de la máquina de estados que queremos implementar. Obvie el fichero `README.md`.

Puede eliminar los ficheros `source.c` y `header.h` que se crearon para poder mantener las carpetas en el repositorio.

2.2.1. Implementación de la máquina de estados

Si abrimos la cabecera de nuestra máquina de estados (*i.e.*, el fichero `/common/include/fsm_blink.h`), veremos su contenido (se han eliminado los comentarios para reducir el espacio):

```
1 #ifndef FSM_BLINK_H_
2 #define FSM_BLINK_H_
3
4 #include <stdint.h>  // para usar uint32_t
5 #include <fsm.h>     // para usar fsm_t
6
7 enum FSM_BLINK_STATES
8 {
9     IDLE, // Estado unico de la FSM
```

```
10 };
11
12 // Definimos una estructura opaca:
13 // sabemos que existe, pero no de que esta hecha
14 typedef struct fsm_blink_t fsm_blink_t;
15
16 // Funcion para crear una nueva maquina de estados
17 fsm_blink_t * fsm_blink_new(uint32_t period_ms);
18
19 // Funcion para eliminar una maquina de estados
20 void fsm_blink_destroy(fsm_blink_t *p_fsm_blink);
21
22 // Funcion que recorre tabla de transiciones de la FSM
23 void fsm_blink_fire(fsm_blink_t *p_fsm_blink);
24
25 #endif // FSM_BLINK_H_
```

Como en cualquier fichero de cabecera, todo su contenido estará protegido por directivas de preprocesador para evitar doble inclusión (líneas 1, 2 y 25; ver apartado en el vídeo *"Directivas de precompilador en C"*).

Primero debemos incluir todas las cabeceras **estrictamente necesarias** para declarar las funciones públicas del fichero (líneas 4 y 5).

Para modelar los estados de nuestra máquina de estados, crearemos una enumeración (`enum`) e incluiremos una entrada por cada estado. En este tutorial, ya lo proporcionamos, es el `enum FSM_BLINK_STATES` (líneas 7 a 10) que contiene un único estado: `IDLE`. Este `enum` nos permitirá referirnos a cada estado por su nombre (lo cual será muy cómodo y legible), y que cada nombre tenga asociado un número **positivo** (que es lo que realmente necesita la librería).

A continuación, definiremos **la existencia** de la estructura `fsm_blink_t` (línea 14). Esto se llama una **estructura opaca**. Cualquier archivo fuente que incluya la cabecera `fsm_blink.h` sabrá de la existencia de una estructura de datos llamada `fsm_blink_t`. Sin embargo, **no sabrá de qué elementos se compone**. Este patrón de programación es una manera de hacer estructuras *privadas* en C.

A continuación, expondremos el prototipo de las funciones para crear nuestra nueva máquina de estados (línea 17) y eliminarla una vez hayamos acabado de usarla (línea 20).

Finalmente, en la línea 23, declaramos la función `fsm_blink_fire`. Esta función es la que hace que la máquina de estados funcione. Internamente, dependiendo del estado actual, comprobará si alguna condición de la máquina de estados se cumple y, de ser así, ejecutará la función de transición y cambiará de estado.

Cualquier fichero que incluya la cabecera `fsm_blink.h` podrá llamar a la función `fsm_blink_new(uint32_t period_ms)` para crear una nueva máquina de estados encargada de hacer parpadear el LED. El periodo de parpadeo se puede configurar mediante el argumento de entrada `period_ms`.

ATENCIÓN

Recomendamos reducir el contenido de las cabeceras a lo estrictamente necesario. De este modo, tendremos un mejor control sobre qué partes de nuestro proyecto pueden usarse desde otros ficheros.
El entorno MatrixMCU permite testear los módulos de nuestro proyecto. Para ello, es necesario que estos elementos sean públicos. No obstante, no sería lo ideal que lo fueran, y todas las partes necesarias para implementar la lógica del autómata que faltan deberían implementarse en el fichero fuente `fsm_blink.c`.
Las funciones, estructuras, o cualquier elemento auxiliar que aparezca en `fsm_blink.c` pero no esté en `fsm_blink.h` permanecerá inaccesible para otras partes de nuestros proyectos.

Implementación de la máquina de estados

Comenzaremos abriendo el fichero `fsm_blink.c`. Primero, incluiremos todas las cabeceras necesarias para la implementación de la máquina de estados:

```
1 #include <stdlib.h>       // para usar NULL
2 #include <stdbool.h>      // para usar booleanos
3 #include "fsm_blink.h"    // para libreria FSM
4 #include "port_system.h" // para consultar el tiempo del sistema
5 #include "port_led.h"     // para interactuar con LED
```

A continuación, encontramos la definición de la estructura `fsm_blink_t`, esta vez sí, indicando todos los elementos que componen esta estructura:

```
1 struct fsm_blink_t
2 {
3     fsm_t fsm;           // FSM insterna. SIEMPRE PRIMER ELEMENTO (composicion)
4     uint32_t period_ms; // Periodo de parpadeo
5     uint32_t last_time; // Tiempo de la ultima vez que el LED cambio de estado
6 };
```

Esta estructura contiene todos los elementos necesarios para implementar nuestra máquina de estados. En este caso, contiene la FSM interna (línea 3), el período de parpadeo del LED (línea 4) y el último instante de tiempo en el que el LED cambió de estado (línea 5).

NOTA

Note que, aunque habíamos declarado la existencia de la estructura `fsm_blink_t` en el fichero de cabecera `fsm_blink.h`, este fichero no especificaba todos los elementos de los que se componía. Estos elementos los estamos definiendo en el fichero fuente `fsm_blink.c`. Esto es lo que se conoce como **estructuras opacas**. Aunque podemos incluir la cabecera `fsm_blink.h` en otros ficheros y hacer uso de datos de tipo `fsm_blink_t`, **no podemos acceder a ninguno de sus elementos internos**. La manipulación de estos elementos se hará exclusivamente dentro del fichero fuente `fsm_blink.c`.

Note que la primera entrada de la estructura `fsm_blink_t` es del tipo `fsm_t` (línea 3). **Esto es OBLIGATORIO PONERLO LO PRIMERO para poder usar la librería FSM.**

Al contrario que en Java o Python, C no implementa un mecanismo de programación orientado a objetos. Por lo tanto, no tenemos ni clases, ni objetos, ni herencia. Sin embargo, podemos usar el **patrón de diseño por composición para emular la herencia**. Para ello, el primer elemento de la estructura que "hereda" será del tipo de la estructura "padre". En este caso nuestra estructura `fsm_blink_t` heredaría de la estructura `fsm_t`, la cual está definida en la librería FSM. Como el primer elemento de nuestra estructura es del tipo `fsm_t`, podemos tratar punteros a estructuras `fsm_blink_t` como si fuesen punteros a estructuras `fsm_t`. De este modo, podemos hacer uso de las funciones de `fsm_t` con un puntero del tipo `fsm_blink_t`, consiguiendo algo parecido a la herencia.

A continuación, implementaremos las funciones de entrada de nuestra máquina de estados. Es este caso, solo necesitamos una función de entrada para comprobar si ha pasado el tiempo suficiente como para cambiar el estado del LED. Se incluye aquí la implementación de la función:

```
1 static bool check_timeout(fsm_t *p_fsm)
2 {
3     fsm_blink_t *p_blink = (fsm_blink_t *) p_fsm;
4     return port_system_get_millis() >= p_blink->last_time + p_blink->period_ms / 2;
5 }
```

Primero, transformamos el puntero `p_fsm` para que, en vez de apuntar a una estructura del tipo `fsm_t`, apunte a una del tipo `fsm_blink_t` (línea 3). Esta transformación del tipo de puntero es conocida como *casteo* (del inglés *to cast*). A continuación, comprobamos si el instante de tiempo actual es mayor o igual que la suma de la última vez que el LED cambió de estado y la mitad del período seleccionado para el autómata. Si la condición de entrada se cumple, la función devuelve `true`. En caso contrario, devolverá `false`. Nótese que la función está definida como `static`. Por tanto, no podremos hacer uso de esta función fuera del fichero `fsm_blink.c`.

Después, implementaremos la función de modificación de salidas de nuestra máquina de estados: `do_toggle()`:

```
1 static void do_toggle(fsm_t *p_fsm)
2 {
3     fsm_blink_t *p_blink = (fsm_blink_t *) p_fsm;
4     p_blink->last_time = port_system_get_millis();
5     port_led_toggle();
6 }
```

Como en el caso anterior, primero *casteamos* el puntero `p_fsm` para que apunte a una estructura del tipo `fsm_blink_t` (línea 3). A continuación, actualizamos el último instante de tiempo en el que el LED cambió de estado y llamamos a la función `port_led_toggle()` para que el LED parpadee.

El siguiente paso es implementar la tabla de transiciones de estados de nuestro autómata, que constará de 1 transición real y 1 transición nula. **Todas las tablas de transición deben acabar con una transición nula, pues este es el modo que usa nuestra librería FSM para detectar que ha llegado al final de la tabla.** Cada entrada o transición es del tipo `fsm_trans_t` (definido en `fsm.h`): una estructura (`struct`) que contiene el estado origen, la función de entrada, el estado destino y la función de modificación de salidas asociada a la transición. En nuestro caso la tabla se llamará `fsm_blink_tt`:

```
1 // {EstadoOrigen,FuncionDeEntrada,EstadoDestino,FuncionDeSalida}
2 static fsm_trans_t fsm_blink_tt[] = {
3     {IDLE, check_timeout, IDLE, do_toggle},
4     {-1, NULL, -1, NULL},
5 };
```

IMPORTANTE

- **El estado origen de la primera transición de la tabla será el estado inicial de la máquina de estados** (en este caso, `IDLE`). Debemos tener esto en cuenta cuando definamos la tabla de transiciones.
- La librería `FSM` comprueba las transiciones de arriba abajo. Si hay dos transiciones cuyas funciones de entrada se cumplen, siempre se ejecutará la que esté primero en la tabla de transiciones. Sabiendo esto, podemos dar más prioridad a determinadas transiciones situándolas más arriba en la tabla.
- Todas nuestras máquinas de estados **DEBEN** acabar con una transición nula `{-1, NULL, -1, NULL}`. Este es el mecanismo que usa la librería `FSM` para identificar cuándo ha llegado al final de la tabla de transiciones.

En la función `fsm_blink_new` (la función declarada en `fsm_blink.h` para crear una nueva máquina de estados) veremos lo siguiente:

```
1 fsm_t *fsm_blink_new(uint32_t period_ms)
```

```
2 {
3     fsm_t *p_fsm = (fsm_t *) malloc(sizeof(fsm_blink_t));
4     if (p_fsm)
5     {
6         fsm_blink_init(p_fsm, period_ms);
7     }
8     return p_fsm;
9 }
```

En la línea 3, con `malloc`, intentamos reservar memoria suficiente como para alojar una estructura del tipo `fsm_blink_t`. Si lo conseguimos (*i.e.*, `malloc` no nos devuelve un puntero `NULL`), llamaremos a la función `fsm_blink_init()` para inicializar todos los parámetros de la máquina de estados:

```
1 void fsm_blink_init(fsm_t *p_fsm_blink, uint32_t period_ms)
2 {
3     fsm_init(&p_fsm_blink->fsm, fsm_blink_tt); // inicializo la FSM interna
4     p_fsm_blink->last_time = port_system_get_millis();
5     p_fsm_blink->period_ms = period_ms;
6     port_led_gpio_setup(); // configuro el pin GPIO del LED
7 }
```

Primero, inicializamos la FSM interna y guardamos el tiempo actual como el tiempo del último cambio en el LED y el periodo de parpadeo deseado en esta estructura. Finalmente, configuraremos el pin GPIO del LED para que actúe como una salida (esta función fue implementada en el tutorial anterior).

En la función `fsm_blink_destroy` (la función declarada en `fsm_blink.h` para eliminar una nueva máquina de estados) veremos lo siguiente:

```
1 void fsm_blink_destroy(fsm_blink_t *p_fsm_blink)
2 {
3     free(p_fsm_blink);
4 }
```

Siempre que reservamos memoria con `malloc`, debemos liberarla con `free` cuando hayamos acabado.

En la función `fsm_blink_fire` (la función declarada en `fsm_blink.h` para que la FSM funcione) veremos lo siguiente:

```
1 void fsm_blink_fire(fsm_blink_t *p_fsm_blink)
2 {
3     fsm_fire(&p_fsm_blink->fsm);
4 }
```

Aunque la función sea una única línea de código, tiene bastante complejidad si no se domina el manejo de punteros. Podemos dividirlo en los siguientes pasos:

```
1 void fsm_blink_fire(fsm_blink_t *p_fsm_blink)
2 {
3    fsm_t *p_fsm = &p_fsm_blink->fsm;
4    fsm_fire(p_fsm);
5 }
```

Internamente, se llama a `fsm_fire`, una función que nos proporciona la librería FSM. Esta función espera como parámetro de entrada un puntero a una estructura de

tipo `fsm_t` (`fsm_t *`. Sin embargo, `p_fsm_blink` es un puntero de tipo `fsm_blink_t` (`fsm_blink_t *`).

Si echamos un vistazo a los elementos que componen la estructura de datos `fsm_blink_t`, veremos que el primer elemento, `fsm`, es una `fsm_t`. **¡Justo lo que queremos!**

Para acceder a elementos de una estructura desde un puntero, usamos el operador `->`. Por lo tanto, si queremos acceder a la `fsm` interna de `p_fsm_blink`, haremos `p_fsm_blink->fsm`. Sin embargo, el resultado de esta operación es un `fsm_t`, **no un puntero a** `fsm_t` (`fsm_t *`).

Un puntero no es más que un número indicando la dirección de memoria en la que se encuentra un dato de un determinado tipo. Por ejemplo un puntero `fsm_t *` indica la dirección de memoria en la que se encuentra un dato de tipo `fsm_t`. En C, el operador para crear un puntero a una variable es `&`. Por lo tanto, `&p_fsm_blink->fsm` es un puntero de tipo `fsm_t *` que apunta al elemento `fsm` interno de la máquina de estados de tipo `fsm_blink_t` a la que apunta el puntero `fsm_blink_t * p_fsm_blink`

NOTA

...
Si... El puntero al elemento apuntado por el puntero del elemento que apunta a... Bienvenid@s al lenguaje de programación C :)

2.2.2. Función principal main

Es hora de implementar la función principal `main()` que ejecute la máquina de estados. Borre el contenido del `main()` del tutorial anterior y agregue el siguiente código. Como siempre, la lógica de esta función debe ir en el fichero de la carpeta raíz. `main.c`:

```
1  #include <stdlib.h>
2  #include <stdint.h>
3  #include <stdio.h>
4  #include "port_system.h"
5  #include "fsm_blink.h"
6
7  #define T_LED_MS 1000 // Period of the LED blink
8  #define T_FSM_MS 10   // Trigger period of the FSM
9
10 int main() {
11   port_system_init();
12   fsm_blink_t *p_fsm_blink = fsm_blink_new(T_LED_MS);
13   uint32_t t = port_system_get_millis();
14   while (1)
15   {
```

```
16      fsm_blink_fire(p_fsm_blink);
17      port_system_delay_until_ms(&t, T_FSM_MS);
18    }
19    fsm_blink_destroy(p_fsm_blink);
20    return 0;
21  }
```

Tras añadir todas las cabeceras necesarias (líneas 1 a 5), definimos el periodo de parpadeo del LED deseado ($1s$) y el periodo de ejecución la máquina de estados ($10ms$).

Finalmente, implementamos la función `main()`. Esta función:

- Primero inicializa el sistema y crea la máquina de estados haciendo uso de la función `fsm_blink_new()`.
- A continuación, guarda el tiempo actual en la variable `t`. Esta variable nos ayudará a implementar un comportamiento síncrono en nuestro sistema.
- Como en casi cualquier sistema empotrado, lo que nos encontramos a continuación es un bucle infinito que ejecuta una tarea de forma periódica: `while(1){}`, es decir, mientras la condición `1` sea cierta, que es siempre cierta, ejecuta el código que se encuentra entre las llaves.
- Al entrar en el bucle infinito, en cada iteración intenta ejecutar alguna transición (función `fsm_blink_fire`), y a continuación el programa se queda dormido durante un tiempo hasta que pasan 10 milisegundos **desde el comienzo de esta iteración** (llamada a función `port_system_delay_until_ms()`). Esto nos permitirá ejecutar la máquina de estados de forma síncrona.

  En otros ejemplos podría no ser necesario hacer este retardo de comprobación, o el tiempo podría ser otro, mayor, o menor, dependiendo de la aplicación.
- Finalmente se debería llamar a la función `fsm_blink_destroy()`, que destruye el autómata una vez hemos acabado de usarlo y devolver 0. En realidad, al ejecutar el autómata en bucle infinito, esta última parte nunca se produciría, por lo que no sería necesaria. Sin embargo, se considera buena práctica incluir esto siempre al final de nuestra función principal.

Ya podemos ejecutar la aplicación en nuestra placa **Nucleo-STM32F446RE** (`Ejecución y depuración (panel izquierdo)` → `Clean and Debug (stm32f446re)` → `target: main`). Recuerde que el depurador siempre detiene la ejecución al principio de la función `main()`. Deberá hacer click en `Continue` para que el programa siga ejecutándose. Si todo ha ido bien, verá que el LED de su placa parpadea con un periodo de, aproximadamente, 1 segundo.

Capítulo 3

Máquinas de Estados Combinadas

Román Cárdenas, Iván Martín, Sergio Esteban

En este capítulo ilustramos cómo combinar varias máquinas de estados para que trabajen de forma coordinada. El sistema propuesto consta de 2 máquinas de estados diferentes. La primera monitoriza el estado del botón de la placa y mide durante cuánto tiempo el usuario ha mantenido pulsado el botón. La segunda máquina de estados consulta la duración medida por la primera y, si esta es superior a cierto límite, cambia el estado del LED.

IMPORTANTE

Este tutorial se plantea a modo de *extensión* de los tutoriales anteriores (Capítulo 1 y Capítulo 2). **Antes de continuar, complete dichos tutoriales.**

El primer paso será crear un nuevo proyecto en la carpeta `/MatrixMCU/projects`. Para ello duplique el `tutorial_2` y renómbrelo como `tutorial_3`. **IMPORTANTE: Recuerde utilizar los ficheros de los tutoriales anteriores ya completados**. Abra la carpeta `/MatrixMCU/projects/tutorial_3` en VSCode para empezar este segundo tutorial.

Descargue los ficheros `fsm_button.h` y `fsm_button.c` del repositorio del segundo tutorial en `https://github.com/sdg2DieUpm/tutorial/tree/tutorial_3`. Coloque estos ficheros en las carpetas `common/include` y `common/src` del nuevo proyecto, respectivamente. Estos ficheros son la cabecera y fuente de la máquina de estados encargada de medir las pulsaciones en el botón y cambiar de estado el LED.

A continuación, descargue el fichero `port_button.h` en `port/include`, `stm32f4_button.h` en `port/stm32f4/include` y `stm32f4_button.c` en `port/stm32f4/src`. En `port_button.h` se declara la interfaz con la que interactuar con el botón de una placa arbitraria, mientras que `stm32f4_button.h` y `stm32f4_button.c` implementan la cabecera y fuente específicas para interactuar con

el botón incorporado en la **STM32F446RE**. Obvie el fichero `README.md`.

3.1. Interfaz para interactuar con el botón

Antes de empezar a definir las máquinas de estados, debemos implementar una interfaz para interactuar con el botón similar a la interfaz para interactuar con el LED presentada en la Sección 1.2. La filosofía de diseño es que la lógica del sistema (el `common`) use funciones para comunicarse con el HW que sean transparentes a la placa que se utilice. Para eso, declararemos las funciones que serán usadas por la lógica del sistema en `port_button.h`, ubicado en `port/include`, mientras que `stm32f4_button.h`, ubicada en `port/stm32f4/include`, incluirá definiciones específicas del microcontrolador en cuestión. Por último, usaremos el fichero `stm32f4_button.c`, ubicado en `port/stm32f4/src`, para escribir el código específico de nuestra placa que realiza estas funciones.

Empezando por el fichero `port_button.h` —el cual se incluye en la plantilla de este tutorial—, este define la API que debemos implementar. En particular, son las siguientes dos funciones:

- `void port_button_gpio_setup(void)`: función que configura el pin GPIO correspondiente al botón (*i.e.*, puerto de GPIO *C*, pin 13) para que actúe como una entrada sin resistencias de *pull up* ni *pull down*.
- `bool port_button_read(void)`: función que lee el estado actual del pin GPIO correspondiente al botón a través del registro `IDR` apropiado. Esta función devuelve `true` si el botón **no está siendo pulsado**.

A su vez, la cabecera `stm32f4_button.h` contiene diferentes definiciones para operar con los registros involucrados:

```
1  /* HW dependent includes */
2  #include "stm32f4xx.h"
3  #include "stm32f4_system.h"
4
5  /* Defines ------------------------------------------------------------------*/
6  #define BUTTON_PORT GPIOC /*!< Button port */
7  #define BUTTON_PIN 13     /*!< Button pin */
8
9  #define MODER13_MASK (0x03 << BUTTON_PIN * 2) /*<! Mask for BUTTON_PIN in MODER
       register */
10 #define PUDR13_MASK (0x03 << BUTTON_PIN * 2)  /*<! Mask for BUTTON_PIN in PUPDR
       register */
11
12 #define MODER13_AS_INPUT (STM32F4_GPIO_MODE_IN << BUTTON_PIN * 2)       /*<! Input
       mode for BUTTON_PIN in MODER register */
13 #define PUPDR13_AS_NOPUPD (STM32F4_GPIO_PUPDR_NOPULL << BUTTON_PIN * 2) /*<! No
       pull up/down for BUTTON_PIN in PUPDR register */
14 #define IDR13_MASK (0x01 << BUTTON_PIN)                          /*<! Mask for
       BUTTON_PIN in IDR register */
```

Primero se incluyen cabeceras de utilidad para la implementación de las funciones. A continuación, se definen los registros y pines asociados al botón de la placa (`BUTTON_PORT` y `BUTTON_PIN`, respectivamente). A continuación, definimos las máscaras para configurar los registros `MODER` y `PUPDR` solo para el pin del botón (`MODER13_MASK` y `PUPDR13_MASK`, respectivamente). Estas máscaras son necesarias para evitar que se modifiquen otros pines del puerto C del GPIO involuntariamente. A continuación, definimos qué valores debemos escribir en estos registros para que el pin asociado al botón actúe como entrada (`MODER13_AS_INPUT`) sin resistencias de *pull up* o *pull down* (`PUPDR13_AS_NOPUPD`). Además, se proporciona la máscara para leer el estado del pin GPIO del registro `IDR` (`IDR13_MASK`).

Tiene más información sobre estos registros y sus valores de configuración en la sección *"Entradas y salidas de propósito general (GPIOs)"* del libro de fundamentos [1].

Vaya al fichero `stm32f4_button.c` y empiece a implementar las funciones de la interfaz. En primer lugar, implementaremos la función para configurar el pin GPIO del botón:

```
1  void port_button_gpio_setup()
2  {
3      /* Primero, habilitamos siempre el reloj de los perifericos */
4      RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN;
5      /* Luego, limpiamos los registros MODER y PUPDR correspondientes */
6      BUTTON_PORT->MODER &= ~MODER13_MASK;
7      BUTTON_PORT->PUPDR &= ~PUDR13_MASK;
8      /* Finalmente, configuramos el boton como entrada sin pull up/pull down*/
9      BUTTON_PORT->MODER |= MODER13_AS_INPUT;
10     BUTTON_PORT->PUPDR |= PUPDR13_AS_NOPUPD;
11 }
```

En primer lugar, **tenemos que activar el reloj del periférico** `GPIOC` (línea 4). Si no lo hacemos, el puerto C del GPIO del sistema permanecerá desactivado. A continuación, ponemos todos los bits relacionados con el pin 13 del puerto C de GPIO a 0 para los registros `MODER` y `PUPDR` (líneas 6 y 7). Hacer esto es importante para evitar que quede activo algún bit indeseado debido a una configuración previa. Finalmente, configuramos estos mismos registros para que el pin actúe como deseamos (líneas 9 y 10).

Para finalizar con la interfaz del botón, implementaremos la función `port_button_read()`. Esta función debe leer el estado actual del botón desde el registro `IDR` y devolver su valor:

```
1  bool port_button_read()
2  {
3      return (bool)(BUTTON_PORT->IDR & IDR13_MASK);
4  }
```

3.2. Implementación del autómata `fsm_button`

La Figura 3.1 representa el diagrama de bolas para la máquina de estados encargada de medir la duración de las pulsaciones al botón de la placa.

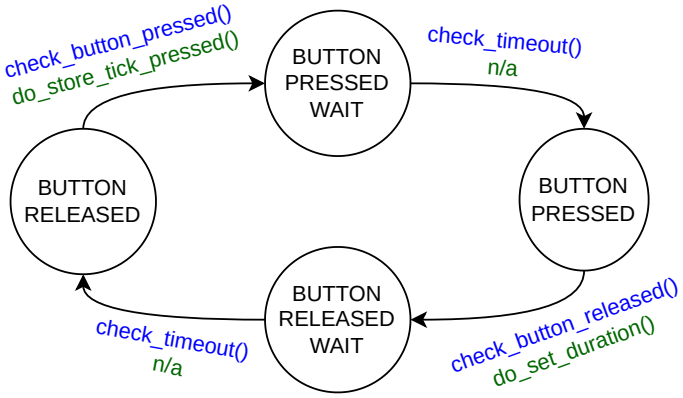


Figura 3.1: Diagrama de la máquina de estados del botón.

Inicialmente, la máquina de estados se encuentra en el estado `BUTTON_RELEASED`. El autómata consulta de forma periódica el estado del botón mediante la función de comprobación de entrada `check_button_pressed()`. En cuanto detecta que el botón está siendo pulsado, la máquina de estados ejecuta la función de modificación de salida `do_store_tick_pressed()`, la cual simplemente marca el instante de tiempo en el que se detectó la pulsación, y transita al estado `BUTTON_PRESSED_WAIT`. Este estado funciona como un mecanismo de anti-rebotes software.

Los botones son un elemento mecánico que, al pulsarlos, generan una serie de oscilaciones antes de que la señal se estabilice. Durante este tiempo, nuestro sistema detectará múltiples pulsaciones que podrían considerarse *glitches*. Para evitar este comportamiento anómalo, nuestra máquina de estados permanecerá en el estado `BUTTON_PRESSED_WAIT` por un tiempo fijo. Una vez pase este tiempo (*i.e.*, `check_timeout()` devuelva `true`), el autómata irá al estado `BUTTON_PRESSED` sin ejecutar ninguna función de modificación de salida. Una vez en este estado, volverá a monitorizar el botón periódicamente para, en este caso, detectar cuándo el usuario deja de pulsar el botón. En cuanto esto ocurra, nuestro autómata calculará el tiempo total que ha pasado el botón pulsado (`do_set_duration()`) y pasará al estado `BUTTON_RELEASED_WAIT`. De nuevo, este estado actúa como un sistema anti-rebotes. Tras esperar el tiempo necesario, el autómata vuelve al estado `BUTTON_RELEASED`.

Implementaremos este autómata en el fichero `fsm_button.c`. Si abrimos la cabecera de nuestra máquina de estados (*i.e.*, el fichero `/common/include/fsm_button.h`), veremos la API que debemos implementar:

```
1 #ifndef FSM_BUTTON_H_
2 #define FSM_BUTTON_H_
```

```
3
4  #include <stdint.h>
5
6  enum FSM_BUTTON_STATES
7  {
8      // TODO: define the FSM's states
9  };
10
11 typedef struct fsm_button_t fsm_button_t;
12
13 uint32_t fsm_button_get_duration(fsm_button_t *p_fsm_button);
14 void fsm_button_reset_duration(fsm_button_t *p_fsm_button);
15
16 fsm_button_t * fsm_button_new(uint32_t debounce_time);
17 void fsm_button_destroy(fsm_button_t *p_fsm_button);
18 void fsm_button_fire(fsm_button_t *p_fsm_button);
19
20 #endif // FSM_BUTTON_H_
```

Cualquier fichero que incluya la cabecera `fsm_button.h` podrá llamar a la función `fsm_button_new(uint32_t debounce_time)` para crear una nueva máquina de estados encargada de medir las pulsaciones del botón. El tiempo de anti-rebotes se puede configurar mediante el argumento de entrada `debounce_time`. Además, la API expone funciones para consultar detalles de la máquina de estados. Por ejemplo, se puede consultar la duración de la última pulsación mediante la función `fsm_button_get_duration()`. Alternativamente, podemos borrar la última duración medida usando la función `fsm_button_reset_duration()`.

Antes de continuar, para modelar los estados de nuestra máquina de estados, completamos el `enum FSM_BUTTON_STATES` e incluiremos una entrada por cada estado:

```
1  enum FSM_BUTTON_STATES
2  {
3      BUTTON_RELEASED,
4      BUTTON_RELEASED_WAIT,
5      BUTTON_PRESSED,
6      BUTTON_PRESSED_WAIT,
7  };
```

Es hora de implementar la máquina de estados en el fichero `fsm_button.c`. Primero, incluiremos todas las cabeceras necesarias:

```
1  #include <stddef.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include "port_button.h"
5  #include "port_system.h"
6  #include "fsm_button.h"
```

Después, definimos el tipo de estructura `fsm_button_t`. Esta vez, sí definiremos todos los elementos que necesita nuestra máquina de estados:

```
1  struct fsm_button_t {
2      fsm_t fsm;
3      uint32_t debounce_time;
```

```
4       uint32_t next_timeout;
5       uint32_t tick_pressed;
6       uint32_t duration;
7   };
```

Recuerde: La estructura `fsm_button_t` es **opaca** fuera del fichero `fsm_button.c`. Solo podremos manipular sus elementos internos directamente desde `fsm_button.c`.

Nuestra máquina de estados tiene una máquina de estados genérica (`fsm`) en primer lugar para poder hacer uso de composición. El tiempo de anti-rebotes seleccionado por el usuario se guarda en `debounce_time`. En cambio, `next_timeout` y `tick_pressed` guardan el instante de tiempo en el que el periodo de anti-rebotes acaba y en el que se pulsó el botón por última vez, respectivamente. Finalmente, `duration` es el contador de duración de pulsaciones.

A continuación, implementaremos las funciones auxiliares para leer/borrar las mediciones de pulsación. Hágalo en la sección *`/* FSM public functions */`* del fichero `fsm_button.c`:

```
1   uint32_t fsm_button_get_duration(fsm_button_t *p_fsm_button)
2   {
3       return p_fsm_button->duration;
4   }
5
6   void fsm_button_reset_duration(fsm_button_t *p_fsm_button)
7   {
8       p_fsm_button->duration = 0;
9   }
```

En este caso, devolveremos el valor del contador interno (línea 3) en la función `fsm_button_get_duration()`. En cambio, en la función `fsm_button_reset_duration()` borraremos el valor del contador interno (línea 8).

Ahora implementaremos las funciones de entrada de nuestra máquina de estados. Hágalo en la sección *`/* State machine input or transition functions */`* del fichero `fsm_button.c`:

```
1   static bool check_button_released(fsm_t *p_fsm)
2   {
3       (void)p_fsm; // Cast to void to avoid unused parameter warning
4       return port_button_read();
5   }
6
7   static bool check_button_pressed(fsm_t *p_fsm)
8   {
9       return !check_button_released(p_fsm);
10  }
11
12  static bool check_timeout(fsm_t *p_fsm)
13  {
14      fsm_button_t *p_fsm_button = (fsm_button_t *) p_fsm;
15      uint32_t now = port_system_get_millis();
16      return now > p_fsm_button->next_timeout;
17  }
```

No olvide definir todas estas funciones como `static` para evitar que se usen fuera del fichero `fsm_button.c`. Las funciones `check_button_released()` y `check_button_pressed()` no necesitan acceder a ningún campo de la máquina de estados, pues simplemente monitorizan el estado del botón. En cambio, la función `check_timeout()` deberá comprobar que el instante de tiempo actual es mayor que el tiempo de anti-rebotes guardado en nuestra máquina de estados (línea 15).

NOTA

La primera línea de la función `check_button_released` es un *cast* del puntero `p_fsm` a `void`. Esto se hace así para indicar explícitamente que no haremos uso de este parámetro en la función y que no salte un aviso de parámetro no usado (*Warning: unused parameter*) que, según las reglas de compilación definidas, impediría que el código se compilase en la placa. El lector curioso podrá observar que al eliminar esa línea el programa ya no compila. Podríamos cambiar la función a `check_button_released(void)` y que no recibiera ningún parámetro, pero la librería FSM **exige** que todas las funciones de comprobación `check_*` obtengan un puntero a una fsm como parámetro.

A continuación implementaremos las funciones de salida o actualización de nuestra máquina de estados, `do_store_tick_pressed()` y `do_set_duration()` en la sección */* State machine output or action functions */* del fichero `fsm_button.c`:

```
1  static void do_store_tick_pressed(fsm_t *p_fsm)
2  {
3      fsm_button_t *p_fsm_button = (fsm_button_t *) p_fsm;
4      uint32_t now = port_system_get_millis();
5
6      p_fsm_button->tick_pressed = now;
7      p_fsm_button->next_timeout = now + p_fsm_button->debounce_time;
8  }
9
10 static void do_set_duration(fsm_t *p_fsm)
11 {
12     fsm_button_t *p_fsm_button = (fsm_button_t *) p_fsm;
13     uint32_t now = port_system_get_millis();
14
15     p_fsm_button->duration = now - p_fsm_button->tick_pressed;
16     p_fsm_button->next_timeout = now + p_fsm_button->debounce_time;
17 }
```

En `do_store_tick_pressed()`, guardaremos el tiempo actual en el campo `tick_pressed` de nuestro autómata (línea 6). En cambio, `do_set_duration` calcula el tiempo que ha pasado desde que el botón se pulsó hasta que se soltó y lo guarda en el campo `duration` del autómata. En ambas funciones, debemos actualizar el tiempo de anti-rebotes para evitar *glitches* (líneas 7 y 16).

El siguiente paso es implementar la tabla de transiciones de estados de nuestro autómata, que constará de 4 transiciones reales y 1 transición nula (que pondremos

al final de la tabla y que es siempre igual para todos los autómatas que creemos). En nuestro caso, la tabla se llamará `fsm_trans_button`:

```
static fsm_trans_t fsm_trans_button[] = {
  {BUTTON_RELEASED, check_button_pressed, BUTTON_PRESSED_WAIT,
      do_store_tick_pressed},
  {BUTTON_PRESSED_WAIT, check_timeout, BUTTON_PRESSED, NULL},
  {BUTTON_PRESSED, check_button_released, BUTTON_RELEASED_WAIT, do_set_duration},
  {BUTTON_RELEASED_WAIT, check_timeout, BUTTON_RELEASED, NULL},
  {-1, NULL, -1, NULL}
};
```

NOTA

Compare el diagrama de bolas de la Figura 3.1 con la tabla presentada. Como puede comprobar, esta tabla ilustra muy bien el comportamiento de la máquina de estados. Una comprobación rápida del diagrama de bolas con la tabla de transiciones es un método sencillo para detectar errores en la tabla.

Como en el tutorial anterior, debemos completar la función `fsm_button_init()` para inicializar todos los parámetros de la máquina de estados:

```
static void fsm_button_init(fsm_button_t *p_fsm_button, uint32_t debounce_time)
{
    fsm_init(&p_fsm_button->fsm, fsm_trans_button);
    p_fsm_button->debounce_time = debounce_time;
    p_fsm_button->tick_pressed = 0;
    p_fsm_button->duration = 0;
    port_button_gpio_setup(); /*  Inicializa el HW del boton */
}
```

Primero, inicializamos la FSM interna, guardamos el tiempo de anti-rebotes, y escribimos un valor inicial para el resto de campos de nuestra estructura. Finalmente, configuraremos el pin GPIO del botón para que actúe como una entrada (esta función fue implementada en la Sección 3.1).

3.3. Implementación del autómata `fsm_led`

La Figura 3.2 ilustra el diagrama de bolas de esta máquina de estados encargada de controlar el LED.

Aunque el diagrama se parece mucho al autómata del Capítulo 2, la función de comprobación de entrada es diferente. En este caso, la máquina de estados monitoriza la duración medida por el autómata del botón y, si ésta es superior a un límite puesto por el usuario, ejecutará la función de modificación de salidas `do_toggle()`.

La API de este autómata se define en `/common/include/fsm_led.h`. Cree dicho fichero en ese directorio y añada el siguiente contenido:

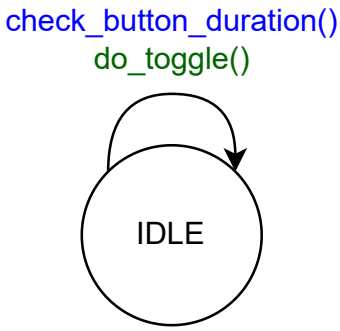


Figura 3.2: Diagrama de la máquina de estados del botón.

```
#ifndef FSM_LED_H_
#define FSM_LED_H_

#include <stdint.h>
#include <fsm_button.h>

enum FSM_LED_STATES {
    IDLE,
};

typedef struct fsm_led_t fsm_led_t;

fsm_led_t * fsm_led_new(fsm_button_t *p_fsm_button, uint32_t min_duration);

void fsm_led_destroy(fsm_led_t *p_fsm_led);
void fsm_led_fire(fsm_led_t *p_fsm_led);

#endif // FSM_LED_H_
```

En este caso, la función para crear esta máquina de estados, `fsm_led_new(fsm_button_t *p_fsm_button, uint32_t min_duration)`, necesita un puntero a la máquina de estados del botón (`p_fsm_button`) para obtener las mediciones de duración. Además, permite que configuremos el tiempo mínimo de pulsación del botón para cambiar el estado del LED mediante el argumento de entrada `min_duration`.

Hemos definido la estructura opaca `fsm_led_t`, que contendrá todos los parámetros que necesita nuestra máquina de estados.

Cree el fichero `/common/src/fsm_led.c`. En él incluiremos todas las cabeceras necesarias para la implementación de la máquina de estados.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdbool.h>
#include <fsm.h>
#include "fsm_button.h"
#include "fsm_led.h"
#include "port_led.h"
#include "port_system.h"
```

A continuación, definimos la estructura `fsm_led_t`, que contiene todos los

parámetros que necesita nuestra máquina de estados:

```
1 struct fsm_led_t {
2     fsm_t fsm;
3     fsm_button_t *p_fsm_button;
4     uint32_t min_duration;
5 };
```

Nuestra máquina de estados contiene una máquina de estados genérica (`fsm`) en primer lugar para poder hacer uso de composición. A continuación, guardamos un puntero a la máquina de estados del botón (`p_fsm_button`) y el tiempo mínimo de pulsación para modificar el valor del LED (`min_duration`).

Después, implementaremos la única función de entrada de nuestra máquina de estados:

```
1 static bool check_button_duration(fsm_t *p_fsm)
2 {
3     fsm_led_t *p_fsm_led = (fsm_led_t *) p_fsm;
4     uint32_t duration = fsm_button_get_duration(p_fsm_led->p_fsm_button);
5     return duration >= p_fsm_led->min_duration;
6 }
```

Primero, transformamos el puntero `p_fsm` para que, en vez de apuntar a una estructura del tipo `fsm_t`, apunte a una del tipo `fsm_led_t` (línea 3). Seguidamente, comprobamos si la última pulsación medida por el autómata del botón es mayor que `min_duration`. Si la condición de entrada se cumple, la función devuelve `true`. En caso contrario, devolverá `false`.

Note que en la línea 4 hacemos uso de la función `fsm_button_get_duration()` para poder leer la duración de la última pulsación medida por el autómata del botón. Esta es la primera vez que las dos máquinas de estados trabajan de manera combinada.

Después, implementaremos la función de modificación de salidas de nuestra máquina de estados: `do_toggle()`:

```
1 static void do_toggle(fsm_t *p_fsm)
2 {
3     fsm_led_t *p_fsm_led = (fsm_led_t *)p_fsm;
4     fsm_button_reset_duration(p_fsm_led->p_fsm_button);
5     port_led_toggle();
6 }
```

Como en el caso anterior, primero *casteamos* el puntero `p_fsm` para que apunte a una estructura del tipo `fsm_led_t` (línea 3). A continuación, borramos la última medición de la máquina de estados del botón y llamamos a la función `port_led_toggle()` para que el LED cambie su estado.

Debemos borrar la última medición porque, en caso contrario, la condición de entrada `check_button_duration()` se cumpliría indefinidamente, por lo que el LED cambiaría su estado constantemente.

Note que en la línea 4 hacemos uso de la función `fsm_button_reset_duration()` para borrar la última duración. Esta es la

segunda vez que las dos máquinas de estados trabajan de manera combinada.

El siguiente paso es implementar la tabla de transiciones de estados de nuestro autómata, que constará de 1 transición real y 1 transición nula (que pondremos al final de la tabla y que es siempre igual para todos los autómatas que creemos). En este caso la tabla se llamará `fsm_trans_led`:

```
1 static fsm_trans_t fsm_trans_led[] = {
2     {IDLE, check_button_duration, IDLE, do_toggle},
3     {-1, NULL, -1, NULL}
4 };
```

Para terminar, completaremos las funciones `fsm_led_init()`, `fsm_led_new()`, `fsm_led_destroy()` y `fsm_led_fire()`:

```
1  static void fsm_led_init(fsm_led_t *p_fsm_led, fsm_button_t *p_fsm_button, uint32_t
       min_duration)
2  {
3      fsm_init(&p_fsm_led->fsm, fsm_trans_led);
4      p_fsm_led->p_fsm_button = p_fsm_button;
5      p_fsm_led->min_duration = min_duration;
6      port_led_gpio_setup(); /*  Inicializa el HW del LED */
7  }
8
9  fsm_led_t *fsm_led_new(fsm_button_t *p_fsm_button, uint32_t min_duration)
10 {
11     fsm_led_t *p_fsm_led = malloc(sizeof(fsm_led_t));
12     if (p_fsm_led)
13     {
14         fsm_led_init(p_fsm_led, p_fsm_button, min_duration);
15     }
16     return p_fsm_led;
17 }
18
19 void fsm_led_destroy(fsm_led_t *p_fsm_led)
20 {
21     free(p_fsm_led);
22 }
23
24 void fsm_led_fire(fsm_led_t *p_fsm_led)
25 {
26     fsm_fire(&p_fsm_led->fsm);
27 }
```

En la función `fsm_led_init()`, primero inicializamos la FSM interna y guardamos el puntero al autómata del botón y la duración mínima para modificar el LED en esta estructura. Finalmente, configuraremos el pin GPIO del LED para que actúe como una salida.

El resto de funciones son muy similares a las anteriores máquinas de estados.

3.4. Implementación de la función principal `main`

Como en los tutoriales anteriores, es hora de implementar la función principal `main()` que ejecute las máquinas de estados. Sustituya el contenido de `main.c` por lo siguiente:

```
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include "port_system.h"
#include "fsm_led.h"
#include "fsm_button.h"

#define T_DEBOUNCE_MS 200 // Button debounce time
#define MIN_DURATION_MS 1000 // Minimum button duration time
#define T_FSM_MS 10   // Trigger period of the FSM

int main() {
    port_system_init();
    fsm_button_t *p_fsm_button = fsm_button_new(T_DEBOUNCE_MS);
    fsm_led_t *p_fsm_led = fsm_led_new(p_fsm_button, MIN_DURATION_MS);
    uint32_t t = port_system_get_millis();
    while (1)
    {
        fsm_button_fire(p_fsm_button);
        fsm_led_fire(p_fsm_led);
        port_system_delay_until_ms(&t, T_FSM_MS);
    }
    fsm_button_destroy(p_fsm_button);    // limpiamos memoria...
    fsm_led_destroy(p_fsm_led);    // limpiamos memoria...
    return 0;          // Y devolvemos 0 para indicar que todo ha ido bien
}
```

Primero, incluimos las cabeceras que vayamos a utilizar. **Preste atención a que no hay que incluir `fsm_blink.h`.**

A continuación, definimos el tiempo de anti-rebotes deseado (200 milisegundos), la duración mínima de pulsación para modificar el LED (1 segundo) y el periodo de ejecución de la máquina de estados (10 milisegundos).

Finalmente, implementamos la función `main()`. Donde primero inicializamos el sistema y creamos las máquinas de estados. Nótese que debemos crear primero la máquina de estados del botón, pues la máquina de estados del LED necesita un puntero a ésta para consultar las mediciones de las pulsaciones.

El bucle de ejecución es similar al del Capítulo 2. Sin embargo, en este caso debemos llamar a la función `fsm_X_fire()` para cada una de las dos máquinas de estados.

Finalmente se debería llamar a las funciones `fsm_X_destroy()` y devolver el estado de finalización de la función `main()` (`0`, típicamente, para indicar que todo ha ido bien).

Ya podemos ejecutar la aplicación en nuestra placa **Nucleo-STM32F446RE** (`Ejecución y depuración (panel izquierdo)` → `Clean and Debug (stm32f446re)` → `target: main`). Recuerde que el depurador siempre detiene la ejecución al principio de la función `main`. Deberá hacer

click en `Continue` para que el programa siga ejecutándose.

Si todo ha ido bien, verá que el LED de su placa se enciende y apaga al pulsar el botón de usuario azul (B1) por un tiempo superior a 1 segundo.

IMPORTANTE

Como habrá podido comprobar a lo largo de los tutoriales anteriores, la implementación de una máquina de estados es siempre igual, es un proceso muy metódico. Podemos resumirlo en los siguientes pasos:

- (`.h`) Definimos los estados del autómata en un `enum`.
- (`.h`) Declaramos una estructura opaca para representar el estado de la FSM.
- (`.h`) Declaramos las funciones públicas que necesitemos para interactuar con el autómata. Esto **siempre** incluye una `fsm_X_new`, `fsm_X_destroy` y `fsm_X_fire`.
- (`.c`) Definimos la estructura del autómata que contiene todos los parámetros necesarios para su funcionamiento y cuyo primer campo es siempre una máquina de estados genérica.
- (`.c`) Implementamos las funciones de entrada del autómata.
- (`.c`) Implementamos las funciones de salida del autómata.
- (`.c`) Implementamos la tabla de transiciones de estados del autómata.
- (`.c`) Implementamos la función (**estática**) de inicialización del autómata.
- (`.c`) Implementamos las funciones `fsm_X_new`, `fsm_X_destroy` y `fsm_X_fire` del autómata.
- (`.c`) Implementamos otras funciones públicas que puedan ser necesarias.
- (`main.c`) Creamos las máquinas de estados y las ejecutamos en el bucle principal constantemente.

Bibliografía

[1] J. Pagán Ortiz, P. J. Malagón Marzo, R. Cárdenas Rodríguez, and J. J. Gómez Valverde, *Fundamentos teóricos de sistemas basados en microcontrolador STM32*. Universidad Politécnica de Madrid, 2025.

[2] J. Pagán Ortiz, P. J. Malagón Marzo, D. Capellán Martín, R. Cárdenas Rodríguez, and A. de Gracia Herranz, *Guía de instalación de herramientas para compilación multiplataforma en C*. Universidad Politécnica de Madrid, 2024.

[3] M. Barr, *Embedded C Coding Standard*. Netrino, 2009.