

SIMONE: Manual de laboratorio

Sistemas Digitales II - UPM

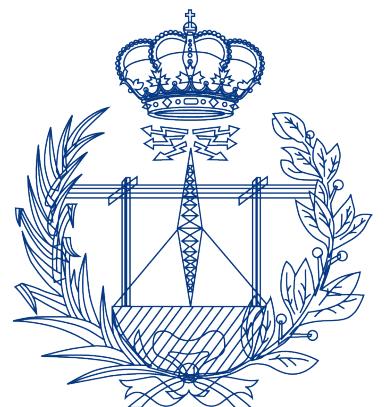
Profesores de SDG2

2026 - Universidad Politécnica de Madrid

Table of contents

1. Portada	3
2. Introducción general	6
2.1 Introducción	6
2.2 Sesión de introducción	18
3. Proyecto	20
3.1 Versión 0 y funciones de sistema	20
3.2 Versión 1: botón de usuario	35
3.3 Versión 2: teclado matricial	52
3.4 Versión 3: <i>RGB light</i>	68
3.5 Versión 4: integración final y modos de bajo consumo	80
3.6 Versión 5: funcionalidades de libre elección	98
4. Apéndices	103
4.1 Lista de materiales (BOM)	103
5. Acrónimos	106
5.1 Acrónimos	107

1. Portada



Simone

Un juego de memoria visual y velocidad

Reinventando el clásico juego Simon



Sistemas Digitales II
Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación
Curso 2025-2026

Profesores:

Josué Pagán Ortiz (coordinador docente) j.pagan@upm.es,
Manuel Gil Martín (coordinador administrativo) manuel.gilmartin@upm.es,
Román Cárdenas Rodríguez, Amadeo de Gracia Herranz,
Raúl Izquierdo López, Juan Antonio López Martín

Colaboradores docentes:

Álvaro Basterra García, Sergio Esteban Romero, Ignacio Hernández Abad, Iván Martín Fernández

Departamento de Ingeniería Electrónica

En memoria de Mario Garrido Gálvez (1981-2025), un excelente profesor y compañero. DEP.

2. Introducción general

2.1 Introducción

En este capítulo se presenta una introducción a la asignatura de laboratorio **SDG2**, y la descripción general del proyecto de la asignatura y la organización de este. Aquí se podrá encontrar información sobre la evaluación y el calendario que se seguirá.

2.1.1 Sistemas Digitales II

La asignatura se imparte con una **metodología de Aprendizaje Basado en Proyectos**². Los alumnos tienen como objetivo realizar el **desarrollo de un sistema electrónico empotrado complejo basado en un microcontrolador de bajas prestaciones partiendo de una descripción y unas especificaciones básicas**. Este aprendizaje requiere por tu parte el **manejo de distintas fuentes de información** y disciplinas necesarias para resolver problemas. **Tu papel ha de ser activo, con un compromiso y responsabilidad por tu propio aprendizaje**. Trabajarás en parejas desde el principio y desarrollarás un proyecto planificando la actuación, distribuyendo las tareas, evaluando las posibles consecuencias, y previendo los éxitos.

Como indica la **Guía de Aprendizaje Basado en Proyectos** del Servicio de Innovación Educativa-UPM², y a modo de resumen, las características más relevantes a las que debe atender son (recuperado de la Guía de Aprendizaje de la asignatura):

- **No existe una única solución correcta.**
- Se presenta la situación y **los alumnos tienen que ampliar la información** para avanzar en el proyecto.
- **El papel del profesor es supervisar y revisar el plan de trabajo** y evolución de cada pareja, utilizando las clases de laboratorio y las tutorías para ayudar a resolver dudas. El profesor tiene también el papel de **evaluar** el progreso y trabajo final.
- La interacción con el alumno se da en las **sesiones de laboratorio: orientación a las dudas y problemas encontrados para seguir el progreso de los estudiantes para evitar equivocaciones, corregir errores conceptuales y orientar el aprendizaje**. Se realiza un seguimiento de cada pareja. También en las tutorías.
- El lugar de trabajo es **el laboratorio y fuera de este**.



El proyecto no se puede completar solo con las sesiones de laboratorio sin un trabajo de búsqueda de información previa y de trabajo fuera del horario de clase. 3 ECTS son unas 81 horas de esfuerzo. El laboratorio son solo \((28)\) (un \((35\%)\)), por lo que en casa ha de trabajar tanto o más que las horas de laboratorio³.

Las sesiones de laboratorio plantean al alumno ir consiguiendo hitos que corresponderán a ciertos niveles de desarrollo (funcionalidad) del proyecto. **El proyecto propuesto se divide en 4 versiones básicas más una versión final de libre elección. Las versiones básicas son requisitos de funcionalidad obligatorios, pero para completar el proyecto y conseguir la máxima nota las parejas deberán implementar la versión final de libre elección.**



IMPORTANTE

Las sesiones de laboratorio NO SON COMO LAS DE OTRAS ASIGNATURAS DE LABORATORIO en las que cada día es una práctica temática. En SDG2 tú **gestionas tu tiempo** dentro y fuera del laboratorio. Existe un objetivo final organizado por fases y, siendo proactivo, debes prever y prepararte para el trabajo que va a hacer cada día en el laboratorio. Aunque existe un calendario orientativo, cada pareja avanzará a un ritmo. Los profesores estarán para ayudar a resolver dudas que la pareja plantee sobre el problema al que se están enfrentando.

Harás frente a un caso real de **diseño e implementación de un sistema electrónico basado en microcontrolador** de bajas prestaciones, empleando los medios disponibles en el **laboratorio B-043 y componentes electrónicos —alguno de los cuales deberá adquirir—** (ver materiales en).

Como has venido trabajando en **CELT**, el **laboratorio estará abierto durante \ (2h30')**. Si se quiere hacer una analogía con las clases teóricas, **las horas con profesor son \ (2h) de supervisión puramente dicha**. El resto del tiempo es para que inicies las máquinas, prepares los montajes, y recojas ordenadamente al finalizar. El acceso al laboratorio está controlado por los maestros de laboratorio.

Por las mañanas habrá acceso libre siempre que haya un maestro de laboratorio o persona responsable que abra el laboratorio y vigile. Se recomienda que **lleves tu propio material**, pues no se garantiza que la persona responsable pueda hacerse cargo de los préstamos.

2.1.2 El proyecto Simone

El objetivo es desarrollar un sistema básico completamente funcional de **una versión del clásico juego de memoria visual, Simón** utilizando un teclado matricial y un LED **RGB** que permita al jugador identificar cada color que se represente por el LED con una tecla del teclado. A este sistema le vamos a llamar **Simone**. El sistema se basa en (i) una placa **Nucleo-STM32F446RE** que hará las veces de sistema central, (ii) un teclado matricial alfanumérico de membrana, y (iii) un LED **RGB** que cambia de color de forma aleatoria con cada secuencia de colores que crea el juego. Puede ver el video demostrativo en [Demostración del sistema Simone](#). Al sistema básico posteriormente el alumno debe añadir más funcionalidades de su elección.

Para entender lo que vamos a construir, debemos viajar atrás en el tiempo, hasta 1978, cuando el ingeniero **Ralph Baer** (conocido como el *padre de los videojuegos*) y Howard J. Morrison presentaron al mundo el juego **Simon**. Este dispositivo con forma de platillo volante se convirtió en un ícono de la cultura pop de los 80. Su funcionamiento era simple: un juego de memoria visual y auditiva donde la máquina generaba una secuencia progresiva de luces y sonidos aleatorios que el jugador debía repetir.



Juego Simon original.

La mecánica original consistía en cuatro grandes botones de colores (**rojo**, **verde**, **azul** y **amarillo**), cada uno asociado a un tono musical específico. El juego comenzaba con una secuencia de un solo paso. Si el usuario acertaba, la máquina repetía la secuencia y añadía un nuevo paso al final. El juego terminaba cuando el usuario fallaba o tardaba demasiado en responder.

En esta práctica vamos a rendir homenaje a este clásico, pero elevando la complejidad técnica para adaptarla a un sistema embebido moderno basado en el **STM32F446RE**. Nuestro sistema, bautizado como **Simone**, mantiene la esencia del juego de memoria visual pero introduce diferencias significativas respecto al juguete original:

6. **Interfaz de entrada:** En lugar de cuatro botones grandes dedicados, utilizaremos un **teclado matricial** (desarrollado en la [Versión 2](#)). Esto nos obliga a mapear teclas específicas (['1'](#), ['2'](#), ...) a colores.
7. **Interfaz de salida:** Sustituimos las cuatro bombillas independientes por un único **LED RGB** (desarrollado en la [Versión 3](#)). Esto implica que la secuencia no es espacial (luces en distintas posiciones), sino puramente cromática (el mismo punto de luz cambia de color).
8. **Complejidad cromática:** El juego original usaba 4 colores. Simone gestiona una paleta de **6 colores** (**rojo**, **verde**, **azul** y **amarillo**, **turquesa** y **blanco**), aumentando la dificultad de memorización.
9. **Niveles de dificultad dinámicos:** Implementaremos un sistema de niveles ([EASY](#), [MEDIUM](#), [HARD](#)) que no solo afecta a la velocidad de reproducción, sino también a la **intensidad lumínica (PWM)**. En los niveles difíciles, los colores se mostrarán más tenues y rápidos, poniendo a prueba tanto la memoria como la agudeza visual del jugador.
10. **Feedback textual:** Al carecer de altavoz en esta versión, utilizaremos la terminal de VSCode para enviar mensajes de estado, victoria o derrota, lo que nos servirá como herramienta de realimentación visual en tiempo real.

El reto de este proyecto no es solo programar la lógica del juego, sino orquestar los periféricos que construiréis a lo largo de las sesiones (botón, teclado, LED RGB) para que funcionen al unísono bajo el control de una **máquina de estados finitos (FSM)** central.

El alumno tiene la libertad de imaginar e implementar a través de la **Versión 5 del proyecto** cualquier sistema que desee. Por ejemplo: añadir un sistema de sonido, más teclados y LEDs, botones, un *buzzer* para generar sonidos, un **LCD** u otro **RGB light**, ... o incluso añadir un sistema de comunicación inalámbrica para enviar la información a un dispositivo móvil.

El sistema correrá sobre la plataforma **Nucleo-STM32F446RE** (ver libro de fundamentos teórico-prácticos de la asignatura ⁴). Sería posible emplear otros modelos de placa **Nucleo-STM32** si el modelo basado en el microcontrolador **STM32F446RE** no está disponible (*e.g.* el modelo **STM32F411RE**). En dicho caso deberá realizar los pasos para adaptarse a la asignatura. **Se deberá comprar una placa por pareja o una por cada miembro**, si lo desea (les dará más posibilidades de hacer pruebas y nuevas implementaciones más interesantes). **Los componentes electrónicos básicos se les prestan**, pero se aconseja que consigáis los vuestros, sobre todo para la Versión 5 del proyecto. Para los primeros días o en caso de olvido, habrá placas de préstamo durante las sesiones de laboratorio.

Para poder trabajar en tu ordenador personal deberás tener instalado el entorno de compilación cruzada siguiendo los pasos de instalación del capítulo de la “Guía de instalación de herramientas para compilación cruzada en C” ⁵. **Durante las sesiones de laboratorio no se resolverán problemas de instalación del entorno para no entorpecer al resto de compañeros que tengan dudas del proyecto. Si se tienen problemas en la instalación, se concertará una tutoría.**

Materiales

La lista de materiales que se usan se muestra en la [BOM del anexo](#). En la puede verlos agrupados por cada una de las "partidas" del proyecto: los básicos, los de la sensorización, y los de actuación.

Lo más urgente de adquirir es la placa Nucleo-STM32F446RE y el cable USB necesarios para la V1 y siguientes. El resto de HW se usará en las siguientes versiones. **Durante las primeras sesiones podrás trabajar con las placas que se prestan** en el laboratorio, pero no podrás llevártela. Cuando adquieras la tuya para trabajar en casa, será la que debes traer.

En el anexo se proporciona información de los componentes y un enlace para que tengas una orientación de cuál es. Puedes conseguirlos tú mismo. Muchos de dichos componentes quizás ya los tengas, como la *protoboard*, o los cables para la misma. El resto puedes conseguirlo en alguna tienda física de la ciudad, u online. Fíjate que **algunos de los componentes no se venden por**

unidad en las tiendas online (quizás le convenga comprar con otros compañeros o ir a una tienda física y comprarlos unitarios). **Un miembro de la pareja se responsabilizará del material prestado, que ha de devolver en la fecha del examen.**

2.1.3 Organización (versiones)

El desarrollo del sistema *Simone* está guiado como un tutorial, por puntos. El proyecto básico son **los requisitos mínimos obligatorios, y suponen el (50%) del total de la calificación de la asignatura. Un (20%) de la nota es la V5** (ver [Evaluación y calendario](#)).

El proyecto básico lo podemos dividir en versiones o fases, a modo de guía. **Todos los códigos deberán estar documentados con Doxygen.** Cada versión tiene un código de test unitarios HW y SW que se les proporcionará. Los códigos corren sobre la placa y nos da una indicación de si el sistema funciona correctamente.

Las características más importantes de cada versión son:

- **Versión 1:** desarrollo de la base del sistema con (i) **FSM**, (ii) interrupción de botón para interactuar con el juego y (iii) temporización de la pulsación con **SysTick** para poder iniciar y detener el juego. (iv) Por último, test unitarios y la documentación del código. (*Estimado 2 semanas*)
- **Versión 2:** desarrollo del subsistema de detección de teclas del teclado matricial con (i) FSM, (ii) excitación de las filas de manera alterna mediante interrupción de un temporizador, (iii) captura de entrada (*input capture*) de las columnas para detección de tecla, (iii) montaje HW del teclado, y (iv) test unitarios y documentación del código. (*Estimado 3 semanas*)
- **Versión 3:** desarrollo del subsistema de visualización (i) FSM, (ii) interrupciones de temporizadores para generación de **PWM**, (iii) montaje HW del LED **RGB**, y (iv) test unitarios y documentación del código. (*Estimado 2 semanas*)
- **Versión 4:** (i) implementación de modos de bajo consumo, (ii) integración de la FSM final del sistema y (iii) prueba y documentación del código. (*Estimado 1-2 semanas*)
- **Versión 5:** funcionalidades de libre elección del alumno. (*Estimado 3 semanas*)

2.1.4 Profesorado y turnos de laboratorio

El turno que elijas para asistir al laboratorio no tiene por qué coincidir con el grupo de matriculación. Puedes elegir el que mejor se ajuste a tu horario. El turno se elige **por orden de inscripción** en la [página de Moodle de la asignatura](#). Típicamente habrá 2-3 profesores por turno y también podemos contar con la ayuda inestimable de los colaboradores docentes que te ayudarán en todo lo posible, aunque no tienen competencias de evaluación, i.e., para consultas de gestión de la asignatura, mejor pregunta a un profesor del turno.

Tradicionalmente un profesor se encarga de "un pasillo" del laboratorio y tiene seguimiento detallado de los alumnos de esos puestos. No obstante, siempre que sea posible se le atenderá por cualquier profesor que esté libre.

IMPORTANTES

/ironic mode on/

Ya se ha mencionado que la metodología es "de Aprendizaje Basado en Proyectos" y que el profesor está para resolver dudas y guiar. **Sabemos que este mensaje a estas alturas de la carrera y de la vida está de más, y que tú no lo haces nunca, pero: levantar la mano y quedarse de brazos cruzados cual cliente en un bar esperando ser servido —en el mejor de los casos, sino es que no se está viendo el último viral de TikTok— no es la actitud que se espera. Se lee, se prueba, se escribe, se hacen dibujos..., se sacan conclusiones y, cuando se ha cavilado, si no se entiende, se pregunta con fundamento.**

/ironic mode off/

Turnos**Turnos de laboratorio**

LT	MT	XT	JT	VC
AGH	JPO	AGH	JPO	MGM
JLM	MGM	RIL	JLM	RCR
-	-	-	RIL	-
ABG	SER	IMF	IHA	-

Lunes-jueves (16:30–18:30, apertura 16:00)

Viernes (10:30-12:30, apertura 10:00)

Algunos profesores pueden usar la pizarra para llevar un orden de atención a los alumnos que se vayan apuntando. Se podrá asistir al laboratorio si no es tu turno y ocupas las **bancadas (pasillos) libres**, si las hay. **No se garantiza que haya profesores o colaboradores docentes para atenderte. Los alumnos del turno tienen preferencia.**

Profesores y colaboradores docentes**Profesores**

- AGH, Amadeo de Gracia Herranz (C-229) *e-mail:* amadeo.degracia@upm.es
- JPO, Josué Pagán Ortiz (coordinador) (C-221) *e-mail:* j.pagan@upm.es
- MGM, Manuel Gil Martín (coord. administrativo) (B-111) *e-mail:* manuel.gilmartin@upm.es
- JALM, Juan Antonio López Martín (B-111) *e-mail:* juanantonio.lopez@upm.es
- RCR, Román Cárdenas Rodríguez (B-305) *e-mail:* r.cardenas@upm.es
- RIL, Raúl Izquierdo López (C-206) *e-mail:* raul.izquierdo@upm.es

Colaboradores docentes

- ABG, Álvaro Basterra García
- SER, Sergio Esteban Romero
- IHA, Ignacio Hernández Abad
- IMF, Iván Martín Fernández

 **Note**

El laboratorio se organiza por parejas por temas de espacio y gestión de recursos. Que pueda elegir un turno distinto del grupo de matriculación es lo que permite que pueda elegir pareja. **Si no tiene pareja, se le asignará una y un turno. No obstante, si el trabajo con la pareja asignada no funciona por x motivo, hágaselo saber antes de la evaluación parcial a un profesor.** No queremos que el no-trabajo de un alumno afecte al rendimiento de otro. **No se permiten tríos. Podrá hacer el trabajo de forma individual.** Le requerirá más disciplina, pero es perfectamente abordable. **Los requisitos del proyecto seguirán siendo los mismos.**

2.1.5 Evaluación y calendario

La asignatura tiene una componente práctica muy importante. No obstante, en esta asignatura no solo se aplican conceptos aprendidos, sino que también se adquieren otros nuevos. La asignatura tiene 2 grandes bloques de evaluación **que se pueden liberar hasta la evaluación extraordinaria del mismo curso:**

5. Un **proyecto por parejas** que tiene 2 partes:

6. requisitos básicos obligatorios: versiones V1-V4. Es guiada. Supone un $\backslash(50\%)$ de la nota total. **Las especificaciones básicas del proyecto son requisitos impuestos por la asignatura**

7. funcionalidades de libre elección: V5. Desarrollo libre. Supone un $\backslash(20\%)$ de la nota total.

Las dos entregas de código (y documentación de la versión V5) se realizarán en buzones de Moodle que se abrirán cerca de la fecha de entrega. **Solo un miembro de la pareja tiene que subir los ficheros fuente del proyecto.** Si realiza implementaciones en V5 que incorporen HW nuevo, deberá mostrar su funcionamiento el día del examen práctico individual o cuando se convenga con su profesor asignado. **En todo caso, se grabará un vídeo breve demostrativo de las funcionalidades implementadas en V5.**

Dispone de una rúbrica orientativa de evaluación del proyecto en la .

8. Un **examen práctico individual** que tiene el fin de diferenciar el trabajo de cada uno de los miembros de la pareja. Supone el $\backslash(30\%)$ restante de la nota total.

El examen consta típicamente de 2 ejercicios. Se proporciona un proyecto adaptado del trabajado durante el curso pero más sencillo. Este proyecto no compila y se pide que (i) se depure y corrija, y (ii) se hagan modificaciones. Se realiza en el laboratorio con los ordenadores del mismo. Se dispone de 1 hora para realizarlo. Se puede consultar la documentación que se proporciona. No se puede consultar internet. Se evalúa la capacidad de depuración de código, la capacidad de análisis y resolución de problemas, y la capacidad de implementación de soluciones. Puede ver cómo abordar esta prueba en estos vídeos de [corrección de examen práctico de 2023](#), [de 2024](#), y [de 2025](#).

Se recomienda el uso de repositorios **privados** de código como [GitHub](#) o [BitBucket](#) para guardar el avance del proyecto cada semana. También puedes usar una memoria USB. **Es responsabilidad exclusiva del alumno conservar copias de las distintas versiones del proyecto y de tus avances parciales. Los ordenadores del laboratorio se borran diariamente.** No obstante, estará abierto continuamente un buzón en Moodle para que puedas subir copias de sus códigos cuando quieras. **Este buzón no tiene copia de seguridad.**

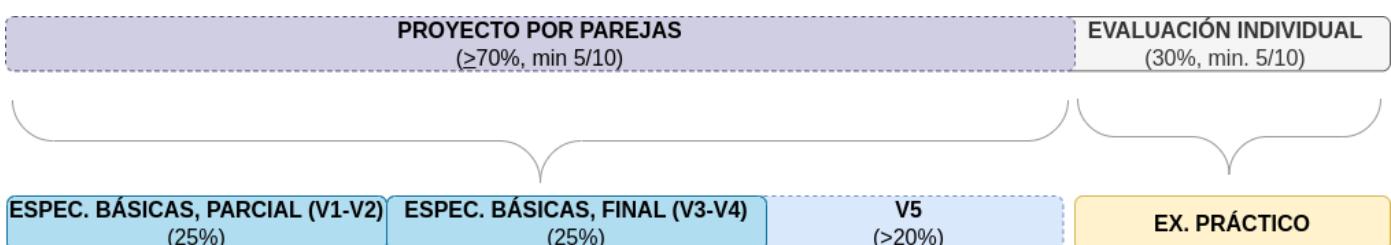
Es responsabilidad de la pareja protegerse de copiar o ser copiados. Los códigos pasan el anti-copy. Si se detecta una copia, la nota de los proyectos copia y copiado será 0. Sean honestos, insíprense, pregúnten, pero nunca copien.

Conviene recordar que, como indica la Guía de Aprendizaje, cualquier evaluación o entrega realizada podrá requerir una evaluación oral complementaria por parte del profesor para validar que se ha realizado por el alumno sin ayuda de sistemas de Inteligencia Artificial.

Como sabrá, la normativa de la UPM⁶ distingue entre evaluación progresiva y global. A continuación se presentan los criterios de evaluación en ambas modalidades.

Evaluación progresiva

En esta modalidad hay dos períodos de evaluación: uno a mitad de semestre y otro al final.



Distribución de las calificaciones de la asignatura.

Puede ver el desglose de las calificaciones en la y el detalle de fechas de evaluación en el calendario de la . En particular:

- **La semana de exámenes del GITST: LÍMITE de entrega por parejas versiones V1 y V2 ((25%)).** Solo se entrega un proyecto QUE COMPILE, hasta donde haya llegado. **SI NO COMPILA, NO SE CORRIGE, Y LA NOTA ES 0.** El profesor corrige con un HW idéntico al tuyo y en la siguiente sesión habrá realimentación de los problemas encontrados. **El código debe compilar y cargar en la placa Nucleo-STM32F446RE.** Esta fecha es el plazo límite de entrega en Moodle, y establecida dentro del periodo de evaluaciones intermedias de la ETSIT. .

ATENCIÓN

El montaje HW no es en sí mismo objeto de evaluación, pero es indispensable que monte bien para que el sistema funcione según las especificaciones. Si sigues las pautas dadas no debería existir ningún problema con el montaje usado por el profesor para la evaluación. **En caso de que detectes algún problema o hagas alguna modificación relevante, comunícaselo a tu profesor durante las sesiones de laboratorio** (Los días próximos a la entrega no son una sesión de laboratorio. Salvo causas de fuerza mayor, claro 😊).

- **Último día lectivo: LÍMITE de entrega por parejas versiones V3 y V4 ((25%)), y V5 ((geq20%)).** El profesor corrige con un HW idéntico al tuyo. **El código debe compilar y cargar en la placa Nucleo-STM32F446RE.** Esta fecha es el plazo límite de entrega en Moodle.

Hay que hacer 2 entregas separadas. Una para la parte obligatoria de requisitos mínimos V1-V4, y otra para V1-V5.

Las funcionalidades de la V5 son de libre elección por el alumno, pueden ser SW, FW, o HW. Las parejas que realicen alguna implementación deberán documentarlas y explicarlas **con Doxygen y Markdown utilizando una plantilla proporcionada.** Con estas modificaciones SW o montajes alternativos añadidos al proyecto básico —y dependiendo de su dificultad y realización — se podrán sumar puntos hasta alcanzar la máxima nota: 10 puntos (ver).

Note

Los alumnos pueden hacer tantas modificaciones como deseen y la suma de puntos podría exceder el 10. No obstante, la calificación máxima en actas es 10, y el exceso se considera a efectos de poder otorgar Matrículas de Honor.

ATENCIÓN

La evaluación de las versiones V1-V2 no tiene nota mínima. Tampoco la tienen V3-V4, ni V5. No obstante, **la suma de la nota de ambas evaluaciones del proyecto V1-V2 + V3-V4-V5 ha de ser mayor de 5/10 para liberar el bloque “proyecto”,** sino, quedará pendiente esta parte para extraordinaria.

- **En la fecha estipulada de exámenes de la ETSIT: examen práctico individual en el laboratorio ((30%)).** De 1 hora de duración aproximada. Se dividirá a los alumnos en varios horarios de evaluación que se harán disponibles cuando se acerque la fecha.

El examen consiste en depurar un código dado, hacer que compile, y hacer modificaciones o pequeños desarrollos de código. Se hará con los ordenadores del laboratorio y no tendrá acceso a internet. Se le proporcionará la documentación de consulta necesaria. Puede ver cómo abordar esta prueba en estos vídeos de **corrección de examen práctico** de 2023, de 2024, y de 2025.

La nota del examen ha de ser mayor de 5/10 y liberar el bloque “examen práctico”, sino, quedará pendiente esta parte para la convocatoria extraordinaria.

En algún momento de la jornada de evaluación, este día también (o cuando se convenga con el profesor), se realizará la **demonstración de las funcionalidades HW añadidas en V5, si las hay.**

La gestión de los plazos también es parte del proyecto. No se aceptan proyectos después de la fecha límite. La fecha es no es la única fecha de entrega, sino la fecha límite. Si no es entrega en plazo, se va a convocatoria extraordinaria.

Evaluación mediante proyecto innovador

Dentro de la evaluación progresiva, aquellos alumnos que deseen realizar un proyecto innovador alternativo que se base en un problema o diseño propio (o propuesto por un profesor), pueden hacerlo. Deberán hablar con alguno de los profesores de la asignatura y presentarle **durante la primera y segunda semana de curso** una propuesta de proyecto donde describan, en 2 o 3 páginas:

- Objetivos del sistema propuesto.
- Recursos necesarios para llevarlo a cabo.
- Arquitecturas HW y SW propuestas para abordar el problema.

Para poder abordar el proyecto será necesario contar con la aprobación de dicho profesor. No será admitido ningún proyecto (por muy complejo o perfecto que sea) que no se ajuste a estas normas. En el canal de YouTube¹ de la asignatura puede ver vídeos de proyectos de alumnos en cursos pasados.

Solo se aceptarán 10 proyectos innovadores. Para poder evaluar las propuestas y que las parejas puedan comenzar a trabajar lo antes posible, disponen de no más de la segunda semana de clase para presentar la propuesta.

El proyecto es libre, no obstante ha de cubrir los conceptos básicos de la asignatura con los que se evalúa al resto de compañeros:

- Deberá trabajar con
- una base del sistema programada en C,
- donde se demuestre el manejo de registros básicos (parte del programa escrito a bajo nivel: *bare-metal*),
- que contenga una o varias **FSM**, interrupciones, y temporización.
- Metodología de proyecto como la propuesta (división de código en **COMMON** y **PORT**).
- Documentación del código con Doxygen.
- El *lenguaje tipo Arduino* no está permitido como base del sistema, aunque sí sobre elementos añadidos (otros microcontroladores). Cubierto lo básico en C, puede trabajar sobre el lenguaje que quiera, sea de alto nivel, o no.
- Cubierto lo anterior, sí puede hacer uso de la **HAL** del fabricante.



Quién realice un **proyecto innovador** NO hace la evaluación individual. En cambio, ha de hacer **una entrega intermedia y final** del código y documentación (mismas fechas que el proyecto estándar), y **una demostración** del proyecto en una fecha a convenir a los profesores de la asignatura. En cualquier momento se pueden abandonar y engancharse a la evaluación global.

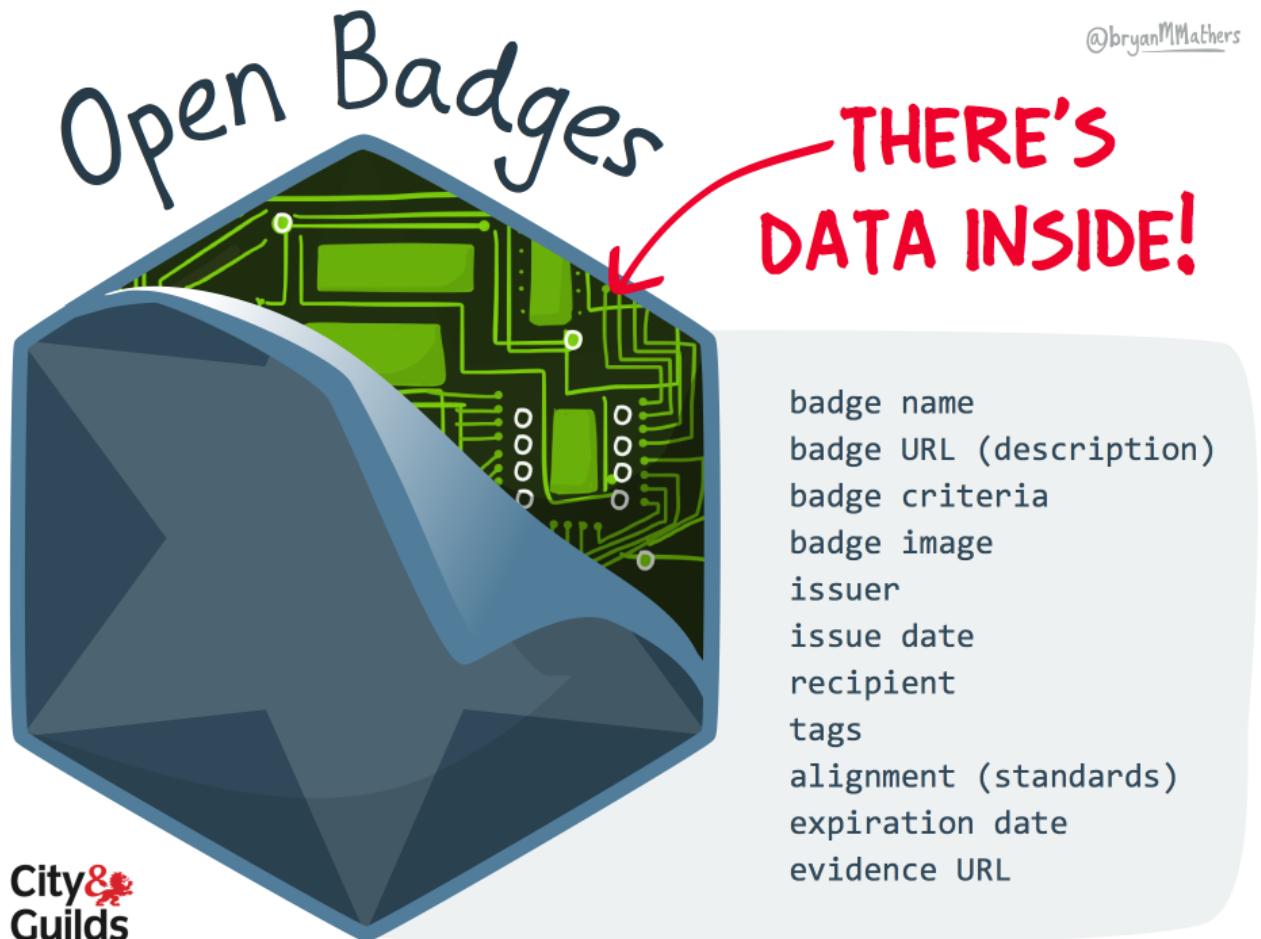
Evaluación global

En la modalidad de evaluación global se han de superar las **mismas actividades que en la evaluación progresiva y tienen los mismos pesos y criterios**.

El estudiante que no han hecho la entrega de V1-V2 puede optar a un \{7.5\}, incluyendo V5, siempre que se cumplan los criterios de mínimos mencionados anteriormente.

Insignias

En general, las insignias, o *badges* en inglés, se usan como reconocimiento digital cuando se realiza un curso o una actividad. Constan de una imagen y una descripción con el nombre y los criterios que hay que cumplir para obtenerla, como se muestra en la figura. Sirven para reconocer logros a lo largo del curso. Están firmadas por un profesor y permite al alumno anexarla a su *Curriculum Vitae*, o almacenarlas en *mochilas virtuales* como [Badgr](#), u [OpenBadges](#). Puede conocer más sobre las insignias de Moodle UPM en el [CanalTIC: Badges o insignias digitales](#). Puede leer más sobre insignias digitales en el manual “[Insignias digitales como acreditación de competencias en la Universidad](#)”⁷.



Contenido de una insignia (“Open Badges Peeled” de Bryan Mathers, con licencia CC-BY-ND License).

Los profesores, bajo criterio consensuado, podrán otorgar insignias para reconocer algún mérito destacable en la asignatura. En concreto, se contemplan las siguientes insignias (ver figura de insignias):

- Para la evaluación progresiva y global:
- **Mejor diseño HW:** electrónica adicional, diseño de **PCB**, integración con otras placas, etc.
- **Mejor diseño SW:** funcionalidades **FW** o **SW** extras que sean significativas.
- **Mejor diseño de producto:** entendido como un todo, se valorará la experiencia de usuario, acabado con impresión 3D, etc.
- **Proyecto destacado:** proyecto que cumple funcionalidades, tiene mejoras considerables, etc. y a juicio de los profesores es un proyecto destacable.
- Para los proyectos innovadores:

- Todo proyecto innovador que lo merezca.
- El mejor proyecto innovador.



(a) Mejor HW



(b) Mejor SW



(c) Mejor diseño



(d) Proyecto destacado



(e) Innovador



(f) Mejor innovador

Insignias del curso.

Rúbrica de evaluación

Una rúbrica “es una herramienta de puntuaciones en la que se valora la calidad de un producto (proyecto, tarea, etc.) en base a los criterios establecidos, iguales para todos los estudiantes. Dichos criterios se presentan en distintos grados y se completan según sea el producto evaluado”². No espere ver una nota detallada de cada aspecto, sino un grado de consecución.

Para cada “Nivel de calidad” de la rúbrica existen procesos intermedios que han de ser superados aunque no queden descritos como tal en la misma. No obstante es una guía para que tenga siempre presente los objetivos de grano grueso del proyecto. El

profesor puede tener mayor granularidad para evaluar parcialmente un criterio de la rúbrica que no se haya logrado en su completitud. El muestra la rúbrica del proyecto *Simone*.

Nivel de calidad			
Criterio	Malo	Regular	Excelente
V1 - Botón	Solo FSM. El sistema no hace lo requerido.	No funciona completamente. No cambia de modo. No gestiona bien los rebotes. No gestiona bien el SysTick .	Pasa los test, funciona correctamente y está bien documentado.
V2 - Teclado	Solo FSM. El sistema no hace lo requerido.	No funciona completamente. No gestiona bien la excitación de filas. No gestiona bien el <i>input capture</i> de columnas. No recoge bien las teclas.	Pasa los test, funciona correctamente y está bien documentado.
V3 - LED RGB	Solo FSM. El sistema no hace lo requerido.	No funciona correctamente. No gestiona bien el PWM para mostrar los colores. No se detiene cuando se le indica.	Pasa los test, funciona correctamente y está bien documentado.
V4 - Bajo consumo	Solo FSM completa. El sistema no hace lo requerido.	No se gestionan comandos ON/OFF. No detiene SysTick . Solo duerme una vez o no duerme en todas las situaciones.	Funciona correctamente y está bien documentado.

Cuadro 1. Rúbrica de evaluación.

Calendario

En la se muestra el calendario de la asignatura. A la izquierda, los días de la semana, señalando los eventos más importantes, como las evaluaciones. A la derecha un gráfico de colores para ver fácilmente qué días tiene clase cada turno. Orientativamente, se indica por qué versión del proyecto se debería ir cada semana. Ajústate lo más que puedas, no te confíes.

	semana	leyenda turnos:					Sesión				
		L	MT	XT	JT	VM	L	MT	XT	JT	VM
FEBRERO	1	2	3	4	5	6	I	I	I	I	I
	2	9	10	11	12	13	V1	V1	V1	V1	V1
	3	16	17	18	19	20					
	4	23	24	25	26	27	V2	V2	V2	V2	V2
MARZO	5	2	3	4	5	6					
	6	9	10	11	12	13					
	7	16	17	18	19	20	X18: día de ajuste. Clase de VC				
	8	23	24	25	26	27	V27 límite entrega intermedia				
	9	30	31	1	2	3	Vacaciones SS				
ABRIL	10	6	7	8	9	10					
	11	13	14	15	16	17					
	12	20	21	22	23	24					
MAYO	13	27	28	29	30	1	M22: día de ajuste. Clase de VC				
	14	4	5	6	7	8					
	15	11	12	13	14	15					
	16	18	19	20	21	22	V22: límite entrega final. Fin per. lectivo				
	17	25	26	27	28	29					
JUNIO	18	1	2	3	4	5	X3: ordinario @ 8:00				
	19	8	9	10	11	12					
	20	15	16	17	18	19					
	21	22	23	24	25	26					
JULIO	22	29	30	1	2	3	M7: extraordinario @ 8:00				
	23	6	7	8	9	10					
	24	13	14	15	16	17					

Fechas importantes

Turnos por grupo y versiones (orientativo)

Calendario de la asignatura.

Fíjate que, debido a cómo es el calendario académico, hay alteraciones de los turnos que, en algunos casos nos llevan a estar varias semanas sin clase, no te distraigas durante ese tiempo para que no te cueste retomar las sesiones de laboratorio. **Todos los turnos tienen el mismo número de horas de laboratorio.** Las fechas de evaluación son iguales para todos los turnos.

En el curso 2025-26 se ha reducido una semana el semestre. Para mantener el número de horas de laboratorio, las 2 sesiones previas a la entrega intermedia y final serán de (2h30') en lugar de (2h).

9. Canal Youtube de la asignatura: https://www.youtube.com/channel/UCYIw_gl745WMJ1n0MamDzQw/ ↵
10. Servicio de Innovación Educativa de la UPM. Aprendizaje orientado a proyectos. Technical Report, Servicio de Innovación Educativa de la UPM, 2008. ↵ ↵ ↵
11. Comisión del Plan de Estudios. Memoria del título de graduado en ingeniería de tecnologías y servicios de telecomunicación. Technical Report, Escuela Técnica Superior de Ingenieros de Telecomunicación, 2014. ↵
12. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, and Juan José Gómez Valverde. *Fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/88460/>. ↵
13. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, Amadeo de Gracia Herranz, Sergio Esteban Romero, and Daniel Capellán Martín. *Guía de instalación de herramientas para compilación multiplataforma en C. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/92376/>. ↵
14. Consejo de Gobierno. Normativa de evaluación del aprendizaje en las titulaciones oficiales de grado y máster universitario de la universidad politécnica de madrid. Technical Report, Universidad Politécnica de Madrid, 2022. ↵
15. Oriol Borrás Gené. Insignias digitales como acreditación de competencias en la universidad. \url <https://oa.upm.es/47460/1/Insignias%20digitales%20como%20acreditacion%20de%20competencias%20en%20la%20Universidad.pdf>, 2017. ↵

2.2 Sesión de introducción

Bibliografía

- "Fundamentos teóricos de sistemas basados en microcontrolador STM32"¹
- "Tutoriales sobre los fundamentos teóricos de sistemas basados en microcontrolador STM32"²
- "Guía de instalación de herramientas para compilación multiplataforma en C"³
- Datasheet "STM32F446xC/E"⁴
- Reference manual "RM0390. STM32F446xx advanced Arm-based 32-bit MCUs"⁵
- "SISTEMAS DIGITALES I: Práctica de programación en C"

Vídeos del canal de SDG1

- Clase inicial de parpadeo de un LED y manejo de proyecto con MatrixMCU
- Conceptos básicos de C (canal SDG1)

En tu primera sesión de laboratorio vas a trabajar con los capítulos de ejemplo de los tutoriales del libro de la asignatura  "Tutoriales sobre los fundamentos teóricos de sistemas basados en microcontrolador STM32"².

Empezarás por el **proyecto Blink** donde aprenderás a configurar un proyecto para compilación cruzada sobre la placa **Nucleo-STM32F446RE**. Luego continúa con los otros **tutoriales para familiarizarte con las máquinas de estado**. Si no te da tiempo, acábalos en casa, porque gran parte del código que ahí realices te servirá para el desarrollo del proyecto. De hecho, constituyen casi en su completitud (aunque no en forma), la versión 1 del sistema. Además, te servirá para familiarizarte con el entorno.

Posteriormente, en casa, te vendrá bien ver los vídeos recomendados de SDG1. Tanto los documentos como los vídeos es importante que los tengas siempre a mano, pues tratan de conceptos fundamentales.

IMPORTANTE

Ve este vídeo-tutorial donde se explica cómo manejar el LED de la placa y añadir uno nuevo. En él se explica en detalle cómo elegir y configurar las GPIO basándose en la documentación disponible [MatrixMCU Blink LED y manejo de proyecto](#).

El libro de *Fundamentos teóricos*¹ no es precisamente un libro de cabecera, **pero lea los capítulos**, y aquellas partes más densas en las que se habla de especificidades de registros, ojéalas, para tenerlas ubicadas cuando las necesites. Esos tiempos muertos en el transporte público son buenos momentos para leerlo y entender qué partes forman la placa **Nucleo-STM32**, repasar de SDG1 qué son los modos de bajo consumo, qué es un reloj... 😊

Para trabajar en casa sigue el Capítulo "Guía de instalación de herramientas para compilación cruzada en C" de la guía³ o ve el vídeo [\[MatrixMCU\] Guía de instalación toolkit MatrixMCU en YouTube](#). No obstante, alguno de los pasos del vídeo puede que los tengas completados de la instalación de SDG1. Repásalo igualmente.

Esperemos que disfrutes y aprendas.

¡Adelante!

-
1. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, and Juan José Gómez Valverde. *Fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/88460/>. ↵ ↵
 2. Román Cárdenas Rodríguez, Josué Pagán Ortiz, Alberto Boscá Mojena, Iván Martín Fernández, and Sergio Esteban Romero. *Tutoriales sobre los fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Román Cárdenas Rodriguez, Madrid, March 2025. URL: <https://oa.upm.es/88470/>. ↵ ↵
 3. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, Amadeo de Gracia Herranz, Sergio Esteban Romero, and Daniel Capellán Martín. *Guía de instalación de herramientas para compilación multiplataforma en C. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/92376/>. ↵ ↵
 4. STMicroelectronics. Stm32f446xc/e. Technical Report, STMicroelectronics, 2021. URL: <https://www.st.com/resource/en/datasheet/stm32f446re.pdf>. ↵
 5. STMicroelectronics. Rm0390 reference manual. stm32f446xx advanced arm-based 32-bit mcus. Technical Report, STMicroelectronics, 2021. URL: https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. ↵

3. Proyecto

3.1 Versión 0 y funciones de sistema

Antes de arrancar con la versión [V1](#) del sistema, tenemos que manejarnos con el versión 0. En esta configuración inicial vamos a programar una serie de funciones genéricas que se usarán a lo largo de todo el proyecto. Esta parte, junto con la versión [V1](#) tiene un desarrollo estimado en unas 2-3 semanas. El inicio del proyecto tiene una curva de aprendizaje mayor; por eso, se te va a guiar en los pasos y la explicación será más extensa. Al acabar este capítulo tendrás listas algunas de las funciones base del proyecto *Simone*.

Debes tener a mano en todo momento **los documentos referenciados y ver los videos sugeridos** a fin de entender mejor cómo tiene que escribir el código o realizar montajes.

Bibliografía

1. *Fundamentos teóricos de sistemas basados en microcontrolador STM32* ⁴
2. *Datasheet “STM32F446xC/E”* ⁵
3. *Reference manual “RM0390. STM32F446xx advanced Arm-based 32-bit MCUs”* ⁶
4. *SISTEMAS DIGITALES I: Práctica de programación en C*

Videos del canal de SDGII

- [\[MatrixMCU - examples\] Blink LED y manejo de proyecto](#)
- [\[MatrixMCU\] Documentación de código con Doxygen](#)

3.1.1 Versión 0: Simone

En esta sección se muestra el esquema general del sistema, y se presenta cómo has de configurar el proyecto en *VSCode*.

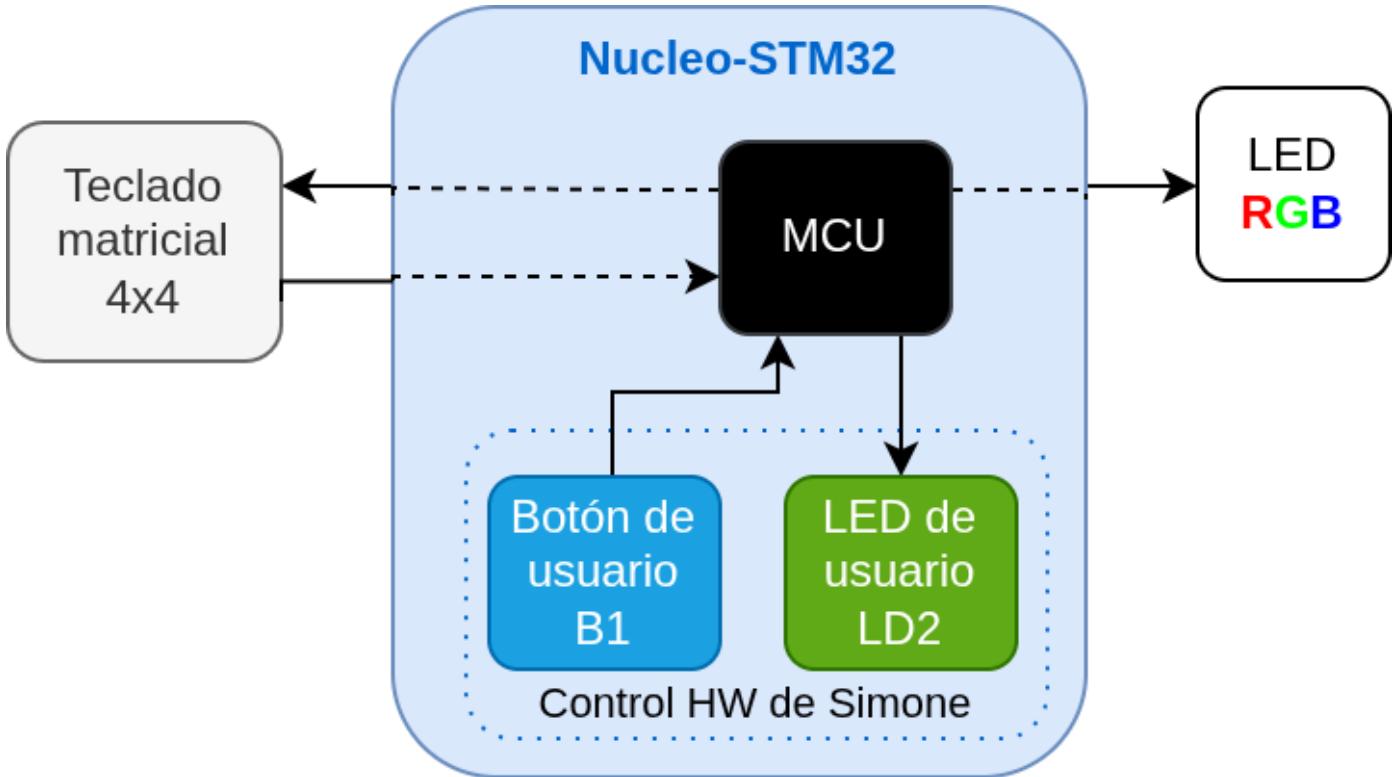


Diagrama de elementos del sistema.

Nuestro sistema base versiona el clásico juego de los 80's, Simon. Cuando se presiona el botón de usuario (el azul) durante un tiempo indica que se quiere encender, o apagar, el juego. En ese instante el sistema central encenderá el *RGB light* (LED RGB) con la primera secuencia del juego en en nivel fácil (LEVEL_EASY). Cuando acaba de reproducirse (*playback*) la secuencia, se activa la excitación de las filas del teclado matricial (4x4) de forma iterativa, y se duerme para esperar una interrupción en alguna de las columnas. Cuando se detecta una pulsación, se comprueba si es la correcta. Si lo es, se vuelve a reproducir la secuencia con un nuevo paso añadido. Si no lo es, se enciende el *RGB light* en rojo para indicar el fallo, y se vuelve a empezar desde el principio. Si se pulsa el botón de usuario durante un tiempo largo, el sistema se apaga. Ve el video demostrativo en [Demostración del sistema Simone](#). La [figura del diagrama](#) muestra los 4 bloques que conforman el sistema:

1. **El sistema central con la placa Nucleo-STM32** aloja el microcontrolador STM32F446RE. Se encarga de gestionar, el encendido y apagado del sistema *Simone*, de interpretar las pulsaciones del usuario.
2. La placa **Nucleo-STM32** también tiene el botón de usuario B1, y el LED de usuario LD2. Estos conforman **el sub-sistema de control básico**. El botón se usará para detectar pulsaciones que el sistema central interpretará para cambiar encender/ apagar, u otras implementaciones que el quieras hacer en [Versión 5](#). El LED de la placa podrá usarse para saber si hemos realizado correctamente una operación, a modo de *feedback*.
3. **El teclado matricial**. Representa a un dispositivo HW que permite al usuario interactuar con el sistema. En este caso, el teclado matricial es de (4x4) teclas, y se usará para que el usuario introduzca la secuencia que el sistema le va indicando. Se trata de un **teclado matricial de membrana** en el que se excitan las filas y se irán leyendo las columnas para identificar qué tecla se ha pulsado.
4. **El módulo de actuación**. El *RGB light* está compuesto por un LED RGB que se encenderá en función de la secuencia que toque reproducir en el juego, y para dar *feedback* al jugador de la tecla que ha pulsado.

Proyecto Simone en VSCode

Vamos a ir construyendo el proyecto poco a poco. En esta primera fase/versión V1 construiremos parte del sistema central y el sub-sistema de control de encendido/ apagado. Se proporciona la estructura del proyecto sobre el que: desarrollaremos (i) las

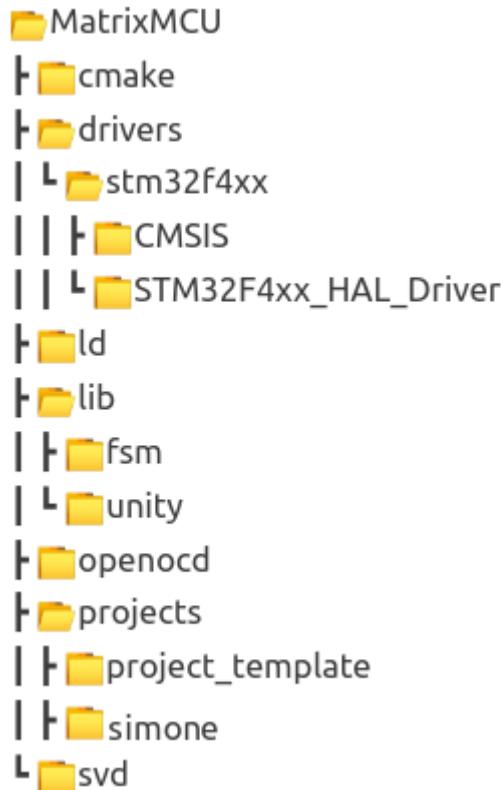
funciones básicas del sistema para gestión de las **GPIO**, (ii) la **FSM** del botón y para el cambio de encendido/ apagado, (iii) *testearmos* su funcionamiento y, en todo momento documentaremos el código. Para empezar a trabajar con el proyecto:

1. Se asume que has hecho la primera sesión de introducción y tienes **ya montada la estructura para programación multiplataforma** con el proyecto *project_template*.

Si estás trabajando en su ordenador, sigue los pasos para tener el entorno como indica la Guía de instalación ⁷.

2. Si ya tenemos todo configurado es suficiente con **descargar** —o clonar, si vas a trabajar con Git— del repositorio GitHub de la asignatura **el versión 0 Simone**: <https://github.com/sdg2DieUpm/simone>, y **dejarlo en la carpeta “projects” de la estructura de directorios MatrixMCU**.

Deberá tener una estructura de directorios como la de la [figura de árbol de directorios](#):

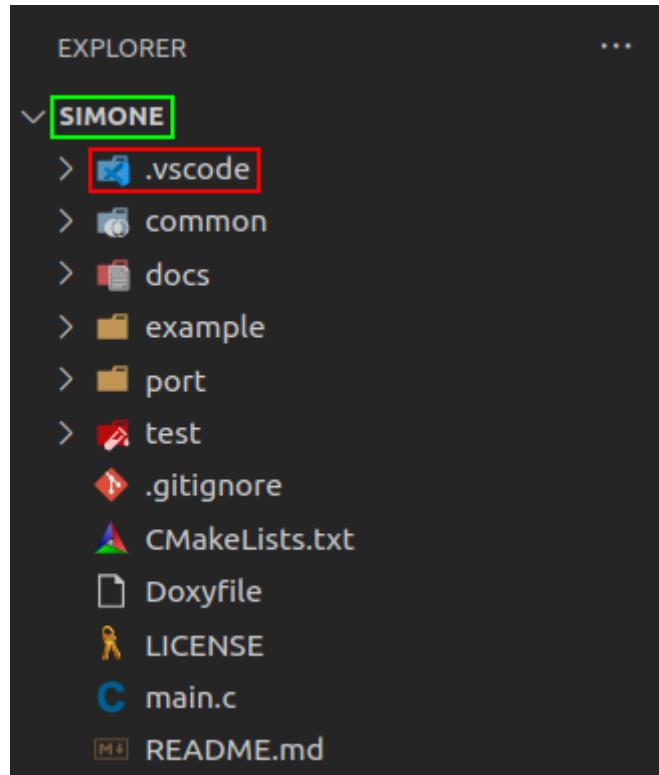


Árbol de directorios tras descargar el proyecto *Simone*.

Note

Los puestos del laboratorio utilizan Windows 10. A priori no debería haber conflicto con la configuración del proyecto en VSCode entre máquinas. No obstante, puedes trabajar con tu portátil en el laboratorio si te es más cómodo que llevar y traer el código cada día. **Sí es obligatorio que traigas tu placa Nucleo-STM32 y el HW necesario.**

1. Abre [VSCode](#) y abre el proyecto [Simone](#) que está en la carpeta [projects](#). **Es muy importante que solo abras esa carpeta, ¡no toda la carpeta `projects`!** Tiene que ver en [VSCode](#) algo como lo que se muestra en la [figura del proyecto en VSCode](#).
2. Conviene que, si no lo has hecho ya, **leas el capítulo “Desarrollando para Nucleo-STM32”**⁴ y te familiarices con la estructura de proyecto y los conceptos.



Proyecto Simone en VSCode.

Verás que el proyecto ya tiene algunos ficheros de partida. Los ficheros de cabecera (`.h`) definen nuevos tipos de variable, contienen `#define`, hacen `#include` de otros ficheros, y declaran variables. Si abres los ficheros `port_system.h` (ver figura) o `stm32f4_system.h` verás que son muy verbosos; casi todo son líneas de comentario. La gran mayoría son líneas de comentario interpretables por el generador de documentación Doxygen. Esto se ha hecho así para que puedas desarrollar el código basándote en la API disponible en la página de GitHub Pages: <https://sdg2dieupm.github.io/simone/>. **Tendrás que tener esta página siempre a mano¹**. Solo tiene algunos ficheros con ejemplo de documentación Doxygen; tú tendrás que completar el de cada función que desarrolles fijándote en los `TODO` alumnos de las descripciones de función. Más adelante se te indicará cómo utilizarla.

En la carpeta `common` verás que hay dos ficheros `header.h` y `source.c` que indican que han de ser borrados. Están ahí para que veas cómo se estructura del proyecto completo. Bórralos cuando añadas los ficheros de la Versión V1.

```
port > include > h port_system.h > ...
14 #ifndef PORT_SYSTEM_H_
38
39 /**
40 * @brief Delays the program execution for the specified number of milliseconds.
41 *
42 * @param ms Number of milliseconds to delay.
43 */
44 void port_system_delay_ms(uint32_t ms);
15
```

Documentación de una función con Doxygen para generar la API.

Podemos empezar ahora con la gestión de las funciones básicas del sistema y las de acceso a las GPIO. En las siguientes secciones vamos a ver (i) un conjunto de funciones genéricas de sistema, (ii) a desarrollar alguna función relacionada con la temporización del sistema, y (iii) funciones de configuración y manejo de las GPIO. **Fíjate que dividimos por puntos enumerados las acciones a realizar. Ten esto en cuenta para llevar un orden y no perderse.**

3.1.2 Funciones de inicialización del sistema

Como se comenta en el capítulo “*Arranque del sistema*” del libro de “*Fundamentos prácticos*”⁴, antes de ir a la función `main()` el microcontrolador debe realizar el *boot* del sistema. En dicho capítulo se indica que una de las cosas que hace antes es llamar a la función `SystemInit()`. Lea dicho capítulo del libro.

Las funciones de inicialización no son evidentes y realizan muchas configuraciones de bajo nivel. Se te proporcionan ya codificadas. Todas se encuentran en el fichero `stm32f4_system.c`. Estas son:

- `SystemInit()`: es llamada directamente por `Reset_Handler` del fichero `startup_stm32f446xx.s`. En nuestra implementación, inicializa la **FPU** (si se usa), configura la memoria externa (si la hay), re-coloca la tabla de vectores de interrupción (si es que la modificamos). En nuestro caso, ninguna de estas tres configuraciones se da.
- `port_system_init()`: **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. Esta función inicializa los periféricos, la memoria *flash* y llama a la función de configuración del reloj `system_clock_config()`.
- `system_clock_config()`: **IMPORTANTE** esta función, por seguridad, no puede ser accedida desde el exterior por lo que su declaración no está en `port_system.h`, para que no la pueda encontrar nadie. Esta función inicializa el oscilador interno **HSI** a $\backslash(16\text{ MHz})$ (valor puesto en el `#define HSI_VALUE`). Esta función también gestiona la alimentación y configura el **temporizador de sistema** `SysTick` a $\backslash(1\text{ ms})$.

3.1.3 Referencia temporal del sistema: SysTick

En esta sección vamos a hablar de las funciones:

- `port_system_delay_ms()`
- `port_system_delay_until_ms()`
- `port_system_get_millis()`
- `port_system_set_millis()`

Y de la **ISR** del temporizador del sistema o `SysTick`:

- `SysTick_Handler()`

El `SysTick` es el temporizador de referencia del sistema. Como se ha mencionado, **está ya configurado para que genere una interrupción interna cada $\backslash(1\text{ ms})$** . Nosotros debemos decidir qué hacer cada vez que se genere dicha interrupción. En nuestro caso queremos tener **una variable que lleve la cuenta de las veces que se ha interrumpido cada $\backslash(1\text{ ms})$** . Es lo que vamos a ver en esta sección.

Conforme está el proyecto *Simone* descargado, si lo compilamos, no te dará errores, pero no funciona. Posteriormente, completaremos algunos detalles que faltan, aunque intercalaremos con la explicación del código en este capítulo.

2. Compila el código. Lo podemos hacer de varias maneras. Típicamente, lo haremos en depuración, **con la placa conectada**, pero por ahora, podemos construir los binarios sin necesidad de la placa. En este orden:

3. **Primero generamos las reglas de compilación** con `CMake`: Menú `Terminal → Run Task... → Run CMake → stm32f446re (Default) → Release (Default)`

Esto generará las carpetas `build/Release/` con todas las reglas de compilación.

Si no vas a trabajar en modo depuración, es necesario que generes las reglas de compilación en modo `Release` cada vez que añadas un fichero al proyecto. Si trabajas en modo depuración, no es necesario, se hace automáticamente tras el Clean.

4. **Seguidamente,** **compilamos** **el** **código:** Menú `Terminal → Run Task... → Build → stm32f446re (Default) → Release (Default) → main`

Note

El modo `Release` se usa cuando queremos generar los ficheros binarios que se ejecutarán en el dispositivo, pero no queremos depurar. Se usa típicamente cuando no tenemos la placa o cuando se genera el código final de un proyecto que pasa a producción.

El modo `Debug` se usa cuando queremos depurar el código, y necesitamos tener el dispositivo conectado. En este caso, el compilador añade información extra al código para que el depurador pueda seguir el código línea a línea. Esto hace que el código sea más lento y ocupe más memoria. En el modo `Release` no se añade esta información extra. Normalmente trabajarás en modo `Debug`.

En el fichero `stm32f4_system.c`, tenemos la variable global `msTicks`. `msTicks` **es una variable muy importante que hemos definido para llevar la cuenta (ticks) en milisegundos del sistema, y está controlado por `SysTick`** (ver capítulo de “Circuito de reloj” del libro de Fundamentos Teóricos ⁴). El valor guardado en `msTicks` se actualiza cada vez que el reloj de sistema `SysTick` genera una **interrupción**. 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**. Trabajaremos con esta variable en las siguientes secciones.

Función `port_system_delay_ms`

La función `port_system_delay_ms` nos podría ser de ayuda en algún momento. Como dice la API, esta función hace una espera activa durante `ms` milisegundos, *i.e.*, el programa se bloquea en el `while{}` que no hace nada durante un tiempo dado. Esta función, para saber cuántos milisegundos han pasado, necesita hacer la resta entre el instante actual y la referencia que coge al inicio `tickstart`.

¿Cómo sabemos cuál es el instante actual? Pues con el valor de la variable global `msTicks`. Esta variable es global y estática (`static uint32_t msTicks`), por lo que es accesible por todas las funciones de este fichero solo². Esta “librería” de sistema `stm32f4_system.c` ha de proporcionar al resto del código diversas funcionalidades, y una de ellas es la de dar información del tiempo de sistema `msTicks`. Para ello —y puesto que la variable no es accesible desde el exterior por ser estática—, nos proporciona la función `port_system_get_millis()`.

Función `port_system_delay_until_ms`

La función `port_system_delay_until_ms()` recibe la referencia del tiempo actual y hace una espera activa hasta `ms` después. Nos puede ser también de utilidad durante el proyecto. Esta función toma la referencia de tiempos llamando a `port_system_get_millis()`.

Función `port_system_get_millis()`

La función `port_system_get_millis()` está declarada en `port_system.h` y, aunque está implementada, no hace nada. Según la API, dicha función simplemente devuelve la cuenta del sistema en milisegundos. Sabemos que ha de devolver un entero de 32 bits sin signo (`uint32_t`), es decir, tenemos que devolver el valor de la variable `msTicks`. Así pues, modifique la función para que quede como:

```
uint32_t port_system_get_millis()
{
    return msTicks; /* ms */
}
```

y como ya está documentada la función en el `.h`, no tenemos que poner la documentación Doxygen aquí.

Función `port_system_set_millis()`

En casi todos los lenguajes de programación, cuando una librería nos proporcione una función `get` que devuelve un parámetro encontraremos, típicamente, un `set`, para poder modificar dicho parámetro. Así pues, encontramos la función `port_system_set_millis()`.

Esta función también está declarada en `port_system.h` e implementada, pero sin desarrollar. Según la API, dicha función simplemente recibe la cuenta del sistema en milisegundos que queremos poner. ¿Por qué ibamos a querer modificar la cuenta de tiempos del sistema? Bueno, nuestra aplicación podría tener algún criterio relacionado con ello, pero esta función existe porque es la que la **ISR** usa para modificar el contador `msTicks`. Ya que la **ISR** está en otro fichero (el fichero `interr.c`) y no tiene acceso al contador, ha de hacerlo llamado a una función de `stm32f4_system.c` que sí pueda acceder; esta es nuestra función `port_system_set_millis()`.

Vamos a completarla:

```
void port_system_set_millis(uint32_t ms)
{
    msTicks = ms;
}
```

 Note

En este punto te habrás dado cuenta que hay funciones que empiezan por "`port_system_`". Esto es una convención de nomenclatura, un estilo. Si seguimos un criterio, nos puede ser fácil identificar de dónde vienen las funciones y qué hacen solo con su nombre. Estas en particular, nos indican que las funciones son "portables" (`port_`) —por lo que estarán en el directorio `port`, y no en el `common`—, y que son funciones centrales del sistema (`system`, por abreviar).

Como reciben y/o devuelven variables estándar (`uint32_t`, `void`, ...), no tienen nada que ver con el microcontrolador en particular, están en declaradas en los ficheros `.h` de `port/include`, pero su implementación está en `stm32f4_system.c` en `stm32f4/src`. Si tuviésemos otro microcontrolador, podríamos reutilizar estas funciones sin modificar el prototipo y solo cambiando la implementación.

Se seguirá un cierto estilo a la hora de programar. Se recomienda ojear el libro de estilo “Embedded C Coding Standard”⁸. El libro es distribuido gratuitamente por los autores en https://barrgroup.com/sites/default/files/barr_c_coding_standard_2018.pdf. Accedido: 2026-01-18.

Función `SysTick_Handler`

`msTicks` está definida y se usa cuando se llama a las funciones `port_system_set_millis()` para darle un valor, y `port_system_get_millis()` para leerlo, pero ¿quién llama a estas funciones para actualizar el valor de `msTicks`? Bueno, estrictamente nadie las llama, saltan ahí por hardware, pues es la **ISR** del `SysTick`. Vamos a ver cómo funciona y a completarla.

En el documento `interr.c` encontrarás la **ISR** del `SysTick`, `SysTick_Handler`. Recuerda que este nombre no lo elegimos nosotros, sino que está definido en el fichero de ensamblador de cada dispositivo; `startup_stm32f446xx.s` para el microcontrolador **STM32F446RE**. Dentro de la **ISR** vamos a usar las funciones `port_system_get_millis()` y `port_system_set_millis()` para leer y escribir el valor de `msTicks`.

5. Llama a la función `port_system_get_millis()` en la **ISR** `SysTick_Handler()` para leer el valor actual de `msTicks` y guárdalo en una variable local. **No olvides declarar la variable**.
6. Llama a la función `port_system_set_millis()` en la **ISR** `SysTick_Handler()` para escribir el valor de `msTicks` con **el valor anterior incrementado en 1**.

Ya tenemos hecho el punto 1 de la **ISR** `SysTick_Handler()` (ver [figura de la API](#)), y que estaba pendiente de hacer ([TODO alumnos](#)).

◆ `SysTick_Handler()`

```
void SysTick_Handler ( void )
```

Interrupt service routine for the System tick timer (SysTick).

Note

This ISR is called when the SysTick timer generates an interrupt. The program flow jumps to this ISR and increments the tick counter by one millisecond.

1. Increment the System tick counter `msTicks` in 1 count. To do so, use the function `port_system_get_millis()` and `port_system_set_millis()`.

Warning

The variable `msTicks` **must be declared volatile!** Just because it is modified by a call of an ISR, in order to avoid [race conditions](#). **Added to the definition after static.**

Vista de la API para la ISR 'SysTick_Handler'.

Podemos comprobar que **el programa sigue compilando correctamente** y viendo que se ha generado el fichero ejecutable `.elf` en la carpeta `bin`, donde se guardan los binarios generados a ejecutar por el microcontrolador. Tendremos un mensaje en la terminal similar al que se muestra en la [figura de compilación](#).

```
[100%] Linking C executable /home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1
/bin/stm32f446re/Release/main.elf
memory region           used Size Region Size %age Used
          RAM:      1576 B     128 KB    1.20%
          FLASH:     1172 B     512 KB    0.22%
```

Compilación exitosa generando el ejecutable `.elf`.

3.1.4 Configuración y manejo de GPIOs

En esta sección vamos a comprender el código de las funciones de configuración y manejo de las GPIOs y a completar algunas de ellas. Con estas funciones conseguiremos inicializar el modo (*i.e.*, entrada, salida, o alternativo) de cada pin de la GPIO, configurar interrupciones, y leer y escribir en los puertos.

Es muy importante que tengas a mano el capítulo “Configuración de GPIOs” del libro de “Fundamentos teóricos”⁴. En esta sección vamos a trabajar con registros. Todo lo que vamos a ver es un repaso de lo que tiene ya en los ejemplos del libro, pero

debés asegurarte de entender lo que hace y saber leer las tablas que en él se muestran. No se te va a pedir manejar el *reference manual* ni el *datasheet* del microcontrolador, aunque también puedes mirar en ellos si quieres más detalle sobre los registros.

A lo largo del proyecto vamos a configurar **varias GPIOs, como las del botón, el teclado, o los LED**. Así, para no hacer el código repetitivo, se proporcionan funciones genéricas que reciben como entrada, al menos, el puerto y el pin que quieren configurar. Para ver las funciones sitúate en el fichero [stm32f4_system.c](#), bajo el comentario [// GPIO RELATED FUNCTIONS](#). La lista de funciones que se proporcionan y que vamos a ver es:

- [stm32f4_system_gpio_config\(\)](#)
- [stm32f4_system_gpio_config_alternate\(\)](#)
- [stm32f4_system_gpio_config_exti\(\)](#)
- [stm32f4_system_gpio_exti_enable\(\)](#)
- [stm32f4_system_gpio_exti_disable\(\)](#)

Además, por otro lado **vas a tener que implementar** las funciones de lectura, escritura y toggle de las GPIOs**. Son:

- [stm32f4_system_gpio_read\(\)](#)
- [stm32f4_system_gpio_write\(\)](#)
- [stm32f4_system_gpio_toggle\(\)](#)

Note

¿Por qué las funciones ahora se llaman [stm32f4_system_gpio\....](#) y no [port_system_gpio\....](#)? Porque estas funciones reciben y devuelven tipos de datos (estructuras, punteros, ...) que son específicas del microcontrolador **STM32F446RE**, y no son portables a otros microcontroladores. Por eso, están declaradas en el fichero [stm32f4/include/stm32f4_system.h](#) y no en [port/include/port_system.h](#). Si tuviésemos otro microcontrolador, tendríamos que incluirla en su carpeta correspondiente y adaptar el prototipo de la función.

Función [stm32f4_system_gpio_config](#)

Esta función configura el modo de la GPIO: **entrada, salida o función alternativa**, y el tipo de conexión a la que está el pin (conectada a resistencias de *pull-up*, o de *pull-down*). Sirva este primer punto para establecer las bases de cómo se codifican las funciones. Tengamos a mano la entrada de la API. La [figura](#) muestra la descripción de la anatomía de una entrada de la API.

◆ `stm32f4_system_gpio_config()`

```
void port_system_gpio_config ( GPIO_TypeDef * p_port,
                               uint8_t      pin,
                               uint8_t      mode,
                               uint8_t      pupd
)
```

Configure the mode and pull of a GPIO.

Prototipo de la función:
tipo que devuelve nombre (tipos que recibe)

This is what function does:

1. Enables GPIOx clock in AHB1ENR
2. Sets mode in MODER
3. Sets pull up/down configuration

Qué hace la función.

Indicativos de cómo lo hace o hay que hacerlo.

Puede haber alguna nota aclarativa.

Note

This function performs the GPIO Port Clock Enable. It may occur that a port clock is re-enabled, it does not matter if it was already enabled.*

This function enables the AHB1 peripheral clock. After reset, the peripheral clock (used for registers read/write access) is disabled and the application software has to enable this clock before using it.

Parameters

`p_port` Port of the GPIO (CMSIS struct like)
`pin` Pin/line of the GPIO (index from 0 to 15)
`mode` Input, output, alternate, or analog
`pupd` Pull-up, pull-down, or no-pull

Descripción argumentos recibidos.

Return values

`None`

Descripción argumentos devueltos.

Definition at line 196 of file `stm32f4_system.c`.

Vista de la API para la función `stm32f4_system_gpio_config`.

Como dice el punto 1 de la API, lo primero que se hace es habilitar el reloj de las GPIO. Esto se indica en los ejemplos del capítulo “Configuración de GPIOs” del libro como:

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; /* GPIOA clock enable */
```

Esta línea de código activa el bit `RCC_AHB1ENR_GPIOAEN` (GPIOA) del registro `RCC_AHB1ENR`. Vamos por partes:

- El `RCC` es el controlador de relojes. En el fichero `stm32f4xx.h`³, CMSIS lo define como un puntero a una estructura de tipo `RCC_TypeDef`: `#define RCC ((RCC_TypeDef *) RCC_BASE)`. Esta estructura también la define CMSIS en el mismo fichero. Tiene muchos campos. Cada campo es un registro del controlador. Bien, pues uno de esos registros es el `RCC_AHB1ENR`.
- Recuerda de los videos de SDG1 que para acceder a los campos de una estructura tenemos el punto (.), y la flecha (→). Usamos punto (.) cuando tenemos acceso directo a la estructura, y usamos flecha (→) cuando lo que tenemos es un acceso indirecto, un puntero. Esta es nuestra situación, porque `RCC` está definido como puntero, y por eso, para acceder al campo que representa al registro hacemos: `RCC->AHB1ENR`.

Aplica toda esta lógica para cualquier registro. Todos están definidos por CMSIS en el fichero `stm32f4xx.h`, y todos son punteros, por lo que siempre accederemos a los registros de los periféricos con → (flecha).

- El fabricante nos indica en el manual de referencia que el acceso a los registros se hace, generalmente, de 32 en 32 bits por lo que no se pueden seleccionar bits particulares de un registro —como indica el libro en un ejemplo—, por eso usamos máscaras, en este caso con el operador `|` (`OR`) sobre todo el registro para activar el/los bit/s deseado/s.
- El `RCC_AHB1ENR_GPIOAEN` es una máscara que está definida en `stm32f4xx.h`, y vale `0x01`. Gran parte de los bits de los registros, si no todos, están definidos en este fichero. Para cualquier registro, tú puedes decidir usarlos, o definirte tus propias máscaras si te es más cómodo.

Ya estamos preparados para ver qué hacen las funciones. Conviene que prestes atención al código y a las explicaciones aquí dadas, pues te ayudará en las próximas versiones del proyecto. De igual modo, no dejes de leer el capítulo “**Configuración de GPIOs**” del libro de “Fundamentos teóricos”⁴, ni de ver el vídeo [MatrixMCU - examples] Blink LED y manejo de proyecto.

5. En el fichero `stm32f4_system.h` encontramos el prototipo de la función. En el fichero `stm32f4_system.c` encontramos su implementación.

6. Dentro de la función vemos que el código activa el bit `RCC_AHB1ENR_GPIOAEN`, `RCC_AHB1ENR_GPIOBEN`, o `RCC_AHB1ENR_GPIOCEN`, en función de si el valor de `port` es `GPIOA`, `GPIOB`, o `GPIOC`, respectivamente. Esto completa el primer punto de la API.

El *pinout* de Arduino no da acceso a ningún pin que no sea de estos tres puertos. El *pinout* de Morpho sí da acceso a algunos del `GPIOH`, y uno del `GPIOD`. Si en tus implementaciones de la Versión 5 vas a utilizar alguno de estos pines, necesitarás incluirlos en esta función de configuración. Mira la sección “*Conectores y zócalos de la placa*” para confirmarlo.

7. Se configura el modo del puerto (registro `MODER`) como se hace en la sección “**GPIO port mode register (GPIOx_MODER)**” del libro.

- Limpia lo que hubiese escrito en las posiciones del registro `MODER` con la máscara creada.

Se usa la máscara base `GPIO_MODE_RMODERO` definida por CMSIS en el fichero `stm32f446xx.h`. Esta máscara vale `0x03UL`, i.e., `0b11` en binario, *unsigned* (`U`, no tiene signo), *long* (`L`, de 32 bits). Es muy útil usar estas máscaras *base* para limpiar los bits de los registros.

Para limpiar los bits de un registro o variable, se hace una `AND` (&) con la máscara negada.

- Escribe en el registro de modo el valor de la variable `mode` —que es el argumento recibido y podrá tomar valores `STM32F4_GPIO_MODE_IN` — `STM32F4_GPIO_MODE_OUT`, `STM32F4_GPIO_MODE_AF`, o `STM32F4_GPIO_MODE_AN`. Estos son los valores que se han definido en el fichero `stm32f4_system.h`, y que valen `0x00`, `0x01`, `0x02`, y `0x03`, respectivamente.

Para escribir en las posiciones correctas del registro, se usa el desplazamiento con la variable `pin`. La variable `pin` se multiplica por `2U` para asegurar que el desplazamiento es de 2 en 2 bits.

Para poner a 1 los bits de un registro o variable, se hace una `OR` (`|`) con el valor deseado.

Ya se ha hecho el segundo punto de la API de esta función.

8. De la misma forma, se configura el tipo de conexión *pull-up/pull-down* del pin. El código es más genérico que el que se explica en la sección “**GPIO port pull-up/pull-down register (GPIOx_PUPDR)**” del libro. Simplemente, se repite los pasos anteriores pero esta vez sobre el registro `PUPDR`: crea la máscara, limpia y escribe en el registro.

Ya se ha hecho el último punto de la API de esta función.

Función `stm32f4_system_gpio_config_alternate`

Esta función configura el modo de la GPIO como función alternativa. Para ello sigue las indicaciones de la API. A saber, lo que hace es:

- Define una máscara base de 4 bits (`0x0F`) para seleccionar los bits del registro `AFR` que se van a modificar.

Esta máscara se desplaza *4 por el valor de la variable* `pin` módulo 8. 4 es el ancho de cada campo del registro `AFR`. El *módulo 8* hace que el valor de `pin` se mantenga entre 0 y 7, y así, no nos salgamos del ancho del registro, que tiene 32 bits.

- Se limpian los bits del registro `AFR` con la máscara creada.

Se selecciona el registro `AFRL` (*low*) o `AFRH` (*high*) en función de: *el valor de la variable* `pin` dividido entre 8. Como `pin` solo puede tomar valores entre 0 y 15, la división entre 8 nos da 0 o 1. Esto hace que el valor resultante seleccione el registro `AFRL` o `AFRH`, respectivamente.

- Se escribe en el registro `AFR` el valor de la variable `alternate`.

Función `stm32f4_system_gpio_config_exti`

Vamos a ver ahora esta función que configura el pin dado para generar interrupciones externas. Lee y ten a mano **el capítulo “Interrupciones” del libro**. El ejemplo te servirá de mucho. Esta función nos servirá para enterarnos cuando pulsemos el botón y realizar alguna acción en nuestro sistema. Vamos a seguir los cuatro puntos de la API de esta función.

4. Lo primero que hace es habilitar el controlador de configuración del sistema que, como nos indica la API, sirve para controlar la línea de interrupciones externas hacia las GPIO. Para ello, se activa el bit `SYSCFGEN` del registro `APB2ENR` del `RCC`. De nuevo, podemos definir nuestra propia máscara para dicho bit mirando el registro, o usar la `RCC_APB2ENR_SYSCFGEN` de `CMSIS` que nos indica la API.

5. Lo siguiente que hace es asociar la interrupción externa al puerto. Este código puede hacerse de muchas maneras, unas más eficientes que otras, aunque no nos preocupamos de eso ahora.

Como se indica en el libro, dependiendo del pin y el puerto, tendremos que elegir un registro u otro para realizar la asociación. Como se trata de una función genérica y no sabemos qué valores nos van a llegar, se hace la implementación para cada pin y puerto según sea el valor de la variable `pin` y `port`. Así pues:

- si el valor de `pin` está entre `0` y `3`, se trabaja sobre el registro `SYSCFG_EXTICR1` (`SYSCFG->EXTICR[0]`),
- si el valor de `pin` está entre `4` y `7`, se trabaja sobre el registro `SYSCFG_EXTICR2` (`SYSCFG->EXTICR[1]`),
- si el valor de `pin` está entre `8` y `11`, se trabaja sobre el registro `SYSCFG_EXTICR3` (`SYSCFG->EXTICR[2]`), y
- si el valor de `pin` está entre `12` y `15`, se trabaja sobre el registro `SYSCFG_EXTICR4` (`SYSCFG->EXTICR[3]`).

Se define una máscara base de 4 bits (`0x0F`) para seleccionar los bits del registro `EXTICRx` que se van a modificar.

Esta máscara se desplaza *4 por el valor de la variable* `pin` módulo 4. 4 es el ancho de cada campo del registro `EXTICRx`. El *módulo 4* hace que el valor de `pin` se mantenga entre 0 y 3 y, así, no nos salgamos del ancho del registro, del que se usan 16 bits.

Se limpian los bits del registro `EXTICRx` con la máscara creada.

Se selecciona el registro `EXTICR1`, `EXTICR2`, `EXTICR3`, o `EXTICR4` en función de *el valor de la variable* `pin` dividido entre 4. Como `pin` solo puede tomar valores entre 0 y 15, la división entre 4 nos da 0, 1, 2 o 3. Esto hace que el valor resultante seleccione el registro `EXTICR1`, `EXTICR2`, `EXTICR3` o `EXTICR4`, respectivamente.

Se escribe en el registro `EXTICRx` el valor que corresponda para cada puerto. Si `port` es `GPIOA`, escribimos el valor `0`, si es `GPIOB`, escribimos el valor `1`..., como indica la “Figura” del libro.

6. A continuación se selecciona la dirección de disparo de la interrupción: en flanco de subida, de bajada, o ambos.

- Se activa el bit del pin correspondiente en el registro `RTSR` (*rising*) si el valor de la variable `mode` es `0x01`, indicando que la interrupción ha de notificarse cuando ocurra un flanco de subida.

El número del pin, `pin`, actúa como índice de una máscara de 1 solo bit, para lo que se usa la macro dada `BIT_POS_TO_MASK(pin)` para actuar sobre los registros.

- Se hace lo propio con el registro `FTSR` (*falling*) si el valor de la variable `mode` es `0x02`, indicando que la interrupción ha de notificarse cuando ocurra un flanco de bajada.
- Si el valor es `0x03` escribe en ambos registros: subida y bajada.

¿Por qué estos valores? Porque así lo hemos decidido, podrían ser otros cualquiera. ¿Y cómo sabemos que son dichos valores? Porque la API nos dice los valores que podemos darle a la variable `mode`:

`mode` Trigger mode can be a combination (OR) of: (i) direction: rising edge (0x01), falling edge (0x02), (ii) event request (0x04), or (iii) interrupt request (0x08).

Distintos valores que podemos darle a la variable `mode`.

Es un consenso con nosotros mismos. Se ha añadido en `stm32f4_system.h` un conjunto de `#define` para estos valores a fin de que el código sea más inteligible y no haya valores “*a pincho*”. Se usan nombres representativos como:

- `#define STM32F4_TRIGGER_RISING_EDGE`
- `#define STM32F4_TRIGGER_FALLING_EDGE`
- `#define STM32F4_TRIGGER_BOTH_EDGE`

7. Por último, de nuevo codificado en el valor de `mode`, se actúa sobre los registros `EXTI_EMR` o `EXTI_IMR` activando el bit correspondiente según si el valor de `mode` es `0x04` o `0x08`, respectivamente. También puede darse el caso de que se quieran activar ambos a la vez. No son excluyentes. En este caso, como en el anterior, activaríamos ambos (se puede añadir un tercer `#define` híbrido, o realizar dos llamadas a la función).

Se añade en `stm32f4_system.h` un conjunto de `#define` con nombres representativos como:

- `#define STM32F4_TRIGGER_ENABLE_EVENT_REQ`
- `#define STM32F4_TRIGGER_ENABLE_INTERR_REQ`

Ya hemos visto cómo está programada la función. Esta ha sido más compleja, pero esperemos que ya vaya cogiendo soltura con los registros.

Función `stm32f4_system_gpio_exti_enable`

El objetivo de esta función es **establecer el nivel de prioridad y subprioridad, y a la vez habilitar la interrupción de una línea externa dada**. La llamaremos cuando queramos configurar un pin como entrada y que genere interrupciones. Internamente llama a una macro de **CMSIS** que controla los registros de interrupción. Estos registros no aparecen en el *reference manual* porque son del *core* de Cortex-M4. **Es muy importante habilitar la interrupción o nunca saltará la ISR.**

Función `stm32f4_system_gpio_exti_disable`

El objetivo de esta función es **deshabilitar la interrupción de una línea externa dada**. La usaremos cuando no queramos que nos interrumpa un pin/ línea configurado como entrada y previamente habilitado. **Mientras esté desactivada, no saltará la ISR asociada a dicha línea.**

Función `stm32f4_system_gpio_read`

Esta función será llamada cuando queramos leer el valor digital de un pin. Leeremos un `'1'` o un `'0'` lógico, por eso, la función devuelve `bool`. Vamos a seguir los dos puntos de la API de esta función para que la implementes tú mismo/a.

8. En `stm32f4_system.h` declara el prototipo de la función según indica la API. De esta forma la función se hace pública y podrá ser llamada desde cualquier otro fichero.

9. En `stm32f4_system.c` escribe el prototipo de la función y abre llaves. Vamos a seguir los pasos que se indican en la API.

10. Lee el valor del registro `IDR` de la GPIO como se muestra en la sección “*GPIO port input data register (GPIOx_IDR)*” del libro.

- El puerto es el `port` dado.
- Usa la macro `BIT_POS_TO_MASK(pin)` para crear la máscara según el pin dado.
- Tenemos que devolver una variable de tipo `bool`, por lo que tendremos que hacer un *cast* del resultado leído del registro, ya que el registro es una variable de 32 bits. Puedes utilizar una variable intermedia para cargar el valor del registro, o hacerlo como en el ejemplo.

Para leer el valor de un bit en un registro hacemos el producto bit a bit (`&`).

11. Devuelve el valor leído.

12. En `stm32f4_system.h` documenta la función con `Doxygen` ayudándose con la API. Recuerda que tienes el vídeo “[MatrixMCU] Documentación de código con Doxygen” con las bases para documentar y generar tu propia API.

Ya tenemos la función que nos permite leer valores digitales del exterior dado un puerto y un pin. Comprueba que compila sin errores: Menú `Terminal → Run Task... → Build → stm32f446re (Default) → Release (Default) → main`.

Función `stm32f4_system_gpio_write`

A continuación vamos a desarrollar la función que será llamada cuando queramos escribir un valor digital en un pin. Escribiremos `'1'` o `'0'` lógicos. Vamos a seguir los dos puntos de la API de esta función.

13. En `stm32f4_system.h` declara el prototipo de la función según indica la API.
14. En `stm32f4_system.c` escribe el prototipo de la función y abre llaves.
15. Escribe el valor correspondiente en el registro `BSRR` de la GPIO como se muestra en la sección “*GPIO port bit set/reset register (GPIOx_BSRR)*” del libro.
 - El puerto es el `port` dado.
 - Si el valor de la variable `value` es `true` activamos el bit correspondiente. Puedes usar la macro `BIT_POS_TO_MASK(pin)` para crear la máscara según el pin dado. **Para evitar posibles condiciones de carrera, utilizamos el registro `GPIOx_BSRR` y no el `GPIOx_ODR`.**
 - Si el valor de la variable `value` es `false` limpiamos el bit correspondiente. Lo limpiamos poniendo a uno el bit correspondiente en el registro `GPIOx_BSRR`; así funciona este registro .
16. En `stm32f4_system.h` documenta la función con `Doxygen` ayudándote con la API.

Ya tenemos la función que nos permite escribir valores digitales dado un puerto y un pin. Comprueba que compila sin errores.

Función `stm32f4_system_gpio_toggle`

Esta función es muy cómoda cuando queremos invertir el valor de un pin. Si estaba en alto (`'1'` lógico), que pase a bajo (`'0'` lógico), y viceversa. Esta función usa las dos anteriores; lee el valor del pin y escribe el opuesto. Seguimos la API.

17. En `stm32f4_system.h` declara el prototipo de la función según indica la API.
 18. En `stm32f4_system.c` escribe el prototipo de la función y abre llaves.
 19. Lee el valor del pin y el puerto. Puedes servirte de la función anteriormente implementada.
 20. Escribe en el pin y el puerto el valor contrario al leído. Puedes servirte de la función anteriormente implementada.
- Puedes definir utilizar las macros `HIGH` y `LOW` si te resulta más cómodo de interpretar que `true` y `false`.
21. En `stm32f4_system.h` documenta la función con `Doxygen` ayudándote con la API.

Ya hemos tenemos la función que nos permite alternar el valor de un pin. Comprueba que compila sin errores. **Podrás probar estas funciones junto con el test unitario del PORT de la versión 1.**

También hemos acabado con las funciones básicas del sistema. Estas funciones se usarán en las distintas versiones del proyecto, por eso es tan importante que las hiciésemos lo primero y que las entienda. Además, nos ha servido para calentar en esto de la programación de bajo nivel (*baremetal*). A continuación seguiremos con la implementación HW del botón y su máquina de estados.

-
22. Aunque tú mismo puedes re-generar dicha API localmente como se explica en la guía ⁷. ↪
 23. La función `port_system_delay_ms()` sí podría acceder a la variable `msTicks` directamente, aunque con llamada a `port_system_get_millis()` es más ortodoxo y así es como lo harían, inevitablemente, funciones de otros ficheros. ↪
 24. El fichero `stm32f4xx.h` se encuentra en el árbol de directorios de la **toolkit MatrixMCU** en: *MatrixMCU/drivers/stm32f4xx/CMSIS/Device/ST/STM32F4xx/Include/stm32f446xx.h* ↪

25. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, and Juan José Gómez Valverde. *Fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/88460/>. ↪ ↫ ↬ ↭ ↮ ↯
26. STMicroelectronics. Stm32f446xc/e. Technical Report, STMicroelectronics, 2021. URL: <https://www.st.com/resource/en/datasheet/stm32f446re.pdf>. ↪
27. STMicroelectronics. Rm0390 reference manual. stm32f446xx advanced arm-based 32-bit mcus. Technical Report, STMicroelectronics, 2021. URL: https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. ↪
28. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, Amadeo de Gracia Herranz, Sergio Esteban Romero, and Daniel Capellán Martín. *Guía de instalación de herramientas para compilación multiplataforma en C. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/92376/>. ↪ ↫
29. Michael Barr. *Embedded C Coding Standard*. Netrino, 2009. ↪

3.2 Versión 1: botón de usuario

Ya estamos en disposición de programar el bloque que controla la pulsación del botón para encender y apagar el sistema *Simone*, y para pausarlo. Debes tener a mano en todo momento **los documentos referenciados y ver los videos sugeridos** a fin de entender mejor cómo tienes que escribir el código o realizar montajes. tt

Bibliografía

1. “Fundamentos teóricos de sistemas basados en microcontrolador STM32”³
2. Datasheet “STM32F446xC/E”⁴
3. Reference manual “RM0390. STM32F446xx advanced Arm-based 32-bit MCUs”⁵

Videos del canal de SDGII

- Demostración Simone
- Blink LED y manejo de proyecto
- Conceptos básicos de C (canal SDG1)
- [MatrixMCU] Documentación de código con Doxygen

Vamos a desarrollar el *botón* de control del sistema *Simone*. Vamos a trabajar con el *timer* del sistema y máquinas de estado. En esta sección explicaremos los fundamentos de esta librería y posteriormente seguiremos los pasos para desarrollarla. Para desarrollar esta librería vamos a contar con la [API](#).

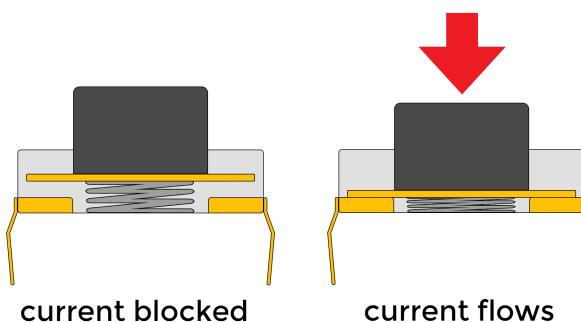
En esta versión 1, el sistema solo trabaja con el botón de usuario. Las características a destacar de este montaje se muestran en el . El botón de usuario está conectado al pin [PC13](#), por lo que usaremos la interrupción externa [EXTI13](#) para detectar cuando pulsamos y soltamos el botón. La prioridad de la interrupción será 1, la más alta, para poder parar el sistema en cualquier momento.

Parámetro	Valor
Pin	PC13
Modo	Entrada
Pull up/ down	No push no pull
EXTI	EXTI13
ISR	EXTI15_10_IRQHandler()
Prioridad	1
Subprioridad	0
Tiempo anti-rebotes	\(100-200 ms\)

3.2.1 Características del botón de usuario en Versión 1

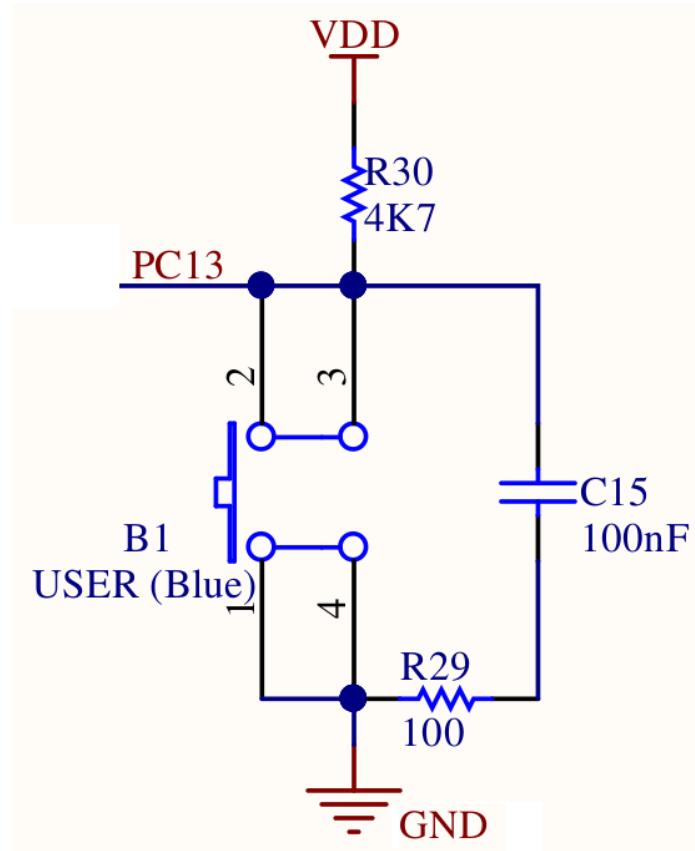
Estudia y practica con las máquinas de estado. Entiende cómo se definen los estados, las tablas de transiciones, y qué son las funciones de entrada/ comprobación y salida/ actuación. Para ello vuelve a leer  el capítulo “2. Introducción a las máquinas de estados en C”⁶.

Si tienes la oportunidad de abrir un botón pulsador¹, seguramente encuentres un diseño mecánico como el de la figura. Se trata de una chapa metálica pegada a un aislante (donde toca el usuario) y colocada sobre un muelle. En la parte inferior, los pines del botón están también en contacto con una chapa metálica. Al presionar, la chapa superior entra en contacto con la inferior y cierra el circuito (cortocircuito), circulando la corriente a través del botón. Cuando se suelta, la corriente deja de pasar (circuito abierto).



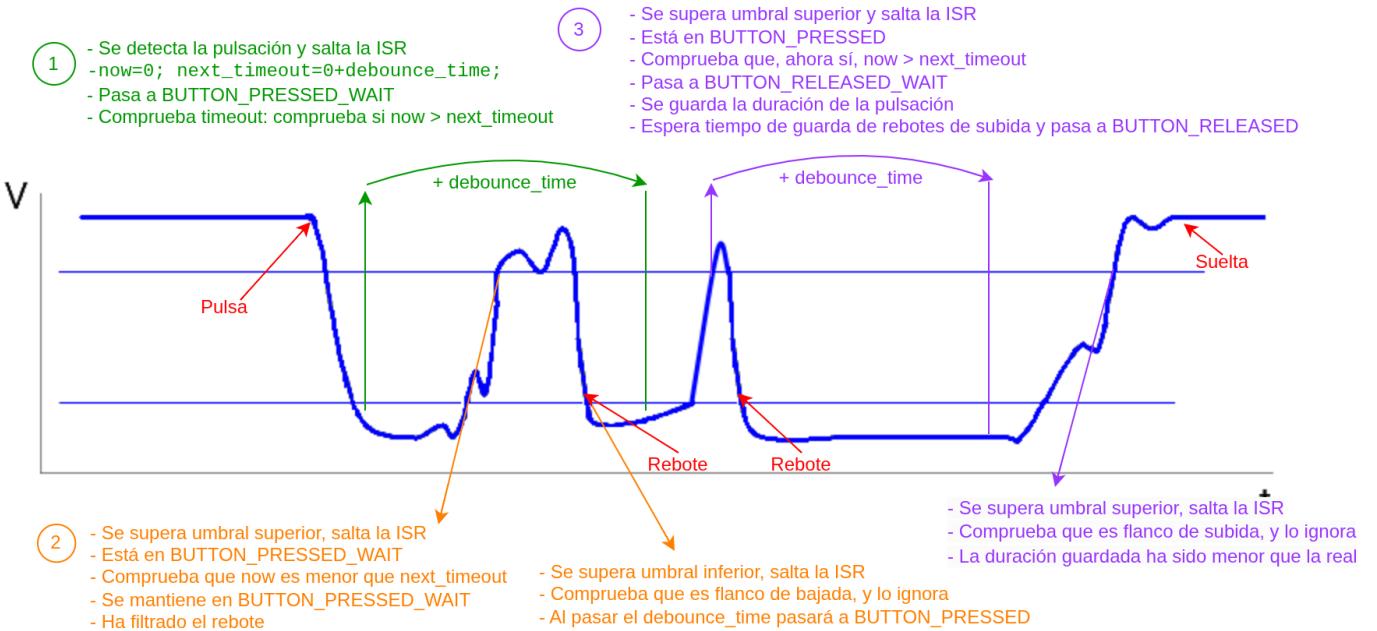
Representación mecánica de un botón.

En la [figura del esquema](#) vemos el esquemático del botón de usuario [B1](#) en la placa [Nucleo-STM32](#). Cuando pulsamos el botón, este hace cortocircuito con [GND](#), por lo que la tensión en el puerto [PC13](#) del microcontrolador pasa de alto a bajo (flanco de bajada).

Esquema del circuito del botón en la placa [Nucleo-STM32F446RE](#).

Como habrás podido intuir, el muelle genera inestabilidades cuando se pulsa, esto es a lo que llamamos **rebotes** (*bounces* en inglés). Estos rebotes son molestos porque se pueden interpretar como múltiples pulsaciones del usuario, por lo que hay que poner algún mecanismo para filtrarlos. Se pueden hacer mecanismos HW como un filtro paso bajo o, como en nuestro caso, se pueden implementar mecanismos SW.

En SW hay varias formas de implementar dicho mecanismo. En nuestra propuesta lo haremos a través de la definición de un tiempo de guarda que obliga a ignorar todo lo que pasa en ese intervalo. Fíjate en el supuesto de la [figura de rebotes](#). Para el proyecto vamos a considerar que la pulsación empieza cuando la ISR detecta el flanco de bajada, y hasta que se detecta un flanco de subida válido. En la figura, hay una pulsación de usuario, y la mecánica del botón hace que haya 2 rebotes bien definidos. El primer rebote no se detecta porque está dentro del tiempo de guarda. No obstante, este tiempo no es lo suficiente grande y se nos cuela el segundo rebote. Esta figura nos servirá como ejemplo para entender el desarrollo de la FSM.



Ejemplo de 2 rebotes en una pulsación con tiempo de anti-rebotes bajo.

Salvo que tengamos mucha experiencia jugando a videojuegos 😊, no seremos muy rápidos pulsando el botón. Dejando un tiempo de guarda entre \((100-200\text{ ms})\) debería ser suficiente, pero depende del deterioro de nuestro botón y quizás debamos ajustarlo haciendo pruebas en el laboratorio con el osciloscopio. Ya estamos preparados para implementar la FSM.

Tenemos que entender que **estamos desarrollando una librería**. Así lo vamos a hacer con todos los bloques del sistema. ¿Qué quiere decir esto? Tenemos que pensar que una librería es una entidad superior que me proporciona "cosas". Pueden ser funciones, o elementos (estructuras). Me puede proporcionar tantos como necesite. El desarrollo de la librería es ajeno al botón que vamos a usar (irá en el **COMMON**, es una lógica **común**, genérica, independiente del HW). Puedo tener **(1)** botón, o **(N)** botones. Cada vez que quiera usar un botón, a este le asociaré una **FSM** de botón. Las particularidades de dónde está conectado este nuevo botón, sus características físicas (tiempo de rebote), etc., son cosas específicas del HW, por lo que estarán en **PORT**. La lógica la vamos a hacer, siempre que se pueda, con máquinas de estado. El **botón** implementa su propia máquina de estados. Esta filosofía es la que mantendremos a lo largo de todo el proyecto.

Note

Vamos a empezar por la parte portable `PORT` de control y acceso al HW. Empezaremos con las cabeceras `.h` y seguiremos con la implementación de las funciones de los ficheros fuente `.c`. Para comprobar que todo funciona correctamente, pasaremos el *test* unitario de la parte `PORT`. Luego, pasaremos a la parte `COMMON` de la librería, que es la lógica común a todos los sistemas. Empezaremos con las cabeceras y seguiremos con la implementación de las funciones de los ficheros fuente. Para comprobar que todo funciona correctamente, pasaremos el *test* unitario de la parte `COMMON`, y finalizaremos con el ejemplo de uso de la librería.

Las cabeceras `.h` van a ser nuestro “**contrato con el usuario**”, y en los ficheros fuente `.c` implementaremos las funciones. **Este es el ciclo de desarrollo que vamos a seguir en todo el proyecto.**

<code>stm32f4_button_hw_t</code>	
<code>GPIO_TypeDef</code>	<code>*p_port;</code>
<code>uint8_t</code>	<code>pin;</code>
<code>uint8_t</code>	<code>pupd_mode;</code>
<code>bool</code>	<code>flag_pressed</code>

<code>fsm_button_t</code>	
<code>fsm_t</code>	<code>f;</code>
<code>uint32_t</code>	<code>debounce_time_ms;</code>
<code>uint32_t</code>	<code>next_timeout;</code>
<code>uint32_t</code>	<code>tick_pressed;</code>
<code>uint32_t</code>	<code>duration;</code>
<code>uint32_t</code>	<code>button_id;</code>

(a) Estructura del HW del botón en `PORT`, (b) Estructura de la FSM del botón en `COMMON`.

Las **figuras de estructuras HW y SW** muestran las estructuras que vamos a necesitar para el botón. La estructura del HW del botón en `PORT`. El `PORT` de otro microcontrolador podría implementar internamente una estructura diferente, por eso está dentro de la carpeta `stm32f4`. Por ejemplo, al *portar* el código para PC no tendría sentido definir la estructura de una GPIO. La estructura de la **FSM** del botón en `COMMON` se muestra en la [figura de la FSM](#).

Ahora sí, comencemos. Preparamos el proyecto para poder añadir el *botón*:

1. Descarga del repositorio de la asignatura los ficheros correspondientes a la parte `PORT` de la librería del *botón* correspondientes a la versión `v1`: https://github.com/sdg2DieUpm/Simone/tree/simone_v1. Solo descarga por ahora: `port_button.h`, `stm32f4_button.h` y `stm32f4_button.c` y colócalos en las carpetas correspondientes de tu proyecto. **No añadas los ficheros de la parte COMMON.**
2. Coloca cada uno donde corresponde: `include`, o `src`. **Ten en cuenta que algunos ficheros de `PORT` están en la carpeta `stm32f4` porque sus funciones reciben o devuelven estructuras específicas de la Nucleo-STM32F446RE.**

Verás que no compila, y es que solo se te proporciona cierta parte del código. Los prototipos de gran parte de las funciones públicas no están definidos.

3.2.2 `PORT`: cabeceras de la librería del botón

Vamos a implementar el *contrato con el usuario* de la parte dependiente del HW de librería del botón. Esto es, qué interfaz vamos a proporcionar al usuario para que pueda usar la librería y crear tantos botones como necesite. Lo haremos, cómo no, para la placa **Nucleo-STM32F446RE**.

Durante todo el desarrollo del proyecto, **si detectas que falta algún `#define`, o `#include`, o declaración de variable que sea necesaria o que necesite, hazlo. Lo que aquí se expone no es algo inmutable, aunque sigue unas buenas prácticas.**

La [figura de plantilla](#) representa las secciones de una plantilla genérica de cabecera que puede utilizar a lo largo del proyecto. El orden no es un estándar, ni las secciones que ahí aparecen. No obstante, sí es muy conveniente ser ordenado y metódico en programación. Sí es importante el orden en los siguientes casos:

- La inclusión de cabeceras ha de ser lo primero. Es importante el orden en caso de existir dependencias entre ellas.
- Es aconsejable definir las etiquetas, macros, enumerados... justo después para que puedan ser utilizados en la declaración y definición de variables. **Solo pondremos en el `.h` aquéllas que queramos que sean visibles y utilizadas por otros ficheros. En caso contrario, lo colocaremos en el `.c`.**
- Si se declara algún tipo nuevo de variable, hay que hacerlo antes de que se use en el prototipo de alguna función. **Las funciones que no queramos que sean accesibles por otros ficheros no tendrán prototipo, y escribiremos y documentaremos directamente en el `.c`, además, se definirán como `static`.**

```

#ifndef NOMBRE_H_
#define NOMBRE_H_                                     #define para prevenir la inclusión múltiple.
                                                       Se usa un nombre representativo cualquiera.

/* Includes -----*/
/* Standard C includes */
#include <string.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>                                Cabeceras <externas> y "propias"

/* HW dependent includes */

/* Other includes */

/* Defines and enums -----*/
/* Defines */
/* Enums */                                         Etiquetas, macros y enumerados públicos al resto de ficheros

/* Typedefs -----*/                                 Estructuras públicas al resto de ficheros

/* Variables -----*/
/* Extern variables */                            Declaración de variables externas, const, etc., si los hay

/* Function prototypes -----*/
/* Variable initialization functions */

/* State machine input or transition functions */
/* State machine output or action functions */
/* Other auxiliary functions */                  Prototipos de las funciones de la librería/ código que se quieran hacer públicas y accesibles

#endif /*NOMBRE_H_*/                                Cierre para prevenir la inclusión múltiple

```

Sugerencia de plantilla genérica de una cabecera.

Nuestro botón es el botón de usuario **B1** de la placa (el azul) será el botón para arranque y parada del juego. Está conectado a la GPIO PC13. Si lo queremos ver en el osciloscopio, tendremos que pinchar en el pin indicado en el header-Morpho izquierdo, como marca la figura “Pinout y funciones header -Morpho izquierdo.” del libro de fundamentos teóricos ³. El esquemático del circuito del botón es el que se mostró en la figura del esquemático.

Cabecera port_button.h

Esta cabecera depende del HW pero no de las particularidades del microcontrolador **STM32F446RE**. En ella vamos a definir las funciones que el usuario podrá usar para gestionar el botón. **Fíjate que todas las funciones reciben el identificador del botón** y solo pueden recibir o devolver variables que **NO dependan del microcontrolador**. Vamos a seguir los siguientes pasos:

- Vamos a definir (`#define`) dos valores que nombraremos en un consenso con nosotros mismos. Puede cambiarlos si lo desea, pero deberán ser los mismos que luego use en el resto del proyecto **¡y en los test, si los cambias, tendrás que tocar los ficheros de test también!** Los valores son:
- `PORT_USER_BUTTON_ID`: valor numérico natural que será el identificador del botón para indicar arranque y parada del juego Simone. Si es nuestro primer y/o único botón del sistema, le asignaremos el `0`.
- `PORT_USER_BUTTON_DEBOUNCE_TIME_MS`: tiempo del anti-rebotes del botón en \ms\.
- Escribe los prototipos de las funciones públicas que aparecen en la API del fichero `port_button.h`.
- Puede ser buen momento ahora para documentar las funciones con Doxygen. Para que aparezca en la API el campo y su definición, hay que poner `/*!< aquí_la_definicion */` junto al nombre del campo como muestra la [figura de ejemplo de Doxygen en enumerados](#).

```
enum FSM_BUTTON
{
    BUTTON_RELEASED = 0, /*!< Starting state. Also co
    BUTTON_RELEASED_WAIT, /*!< State to perform the ar
    BUTTON_PRESSED /*!< State while the button is pressed
};
```

Comentario de Doxygen en las claves de un enumerado.

Puedes partir de los ejemplos dados en el fichero `port_system.h` y ayudarse con extensa documentación online. [Puede hacerlo en español o inglés²](#).

Note

Podemos tener ayuda para autocomentar parcialmente el código con Doxygen. Comprueba si tienes instalada la extensión llamada *Doxygen Documentation Generator* —a veces se instala junto con otras extensiones—; si no la tienes, puedes instalarla. Con esta extensión, si te colocas justo en la línea encima del nombre de la función y escribes: *barra, asterisco, asterisco, enter* `/***/`, colocando el cursor en la tercera posición, como en la imagen, para darle a enter, se genera automáticamente un esqueleto para poder completarlo. También se puede hacer si, colocados justo encima de la función, damos botón derecho → Generate Doxygen Comment.

Ya hemos acabado con el encabezado que interactúa con el HW del botón y que no depende del microcontrolador. Todavía dará errores al compilar. Vamos ahora a programar la cabecera que sí depende del microcontrolador `stm32f4_button.h`.

Cabecera stm32f4_button.h

Esta cabecera depende del HW y del microcontrolador **STM32F446RE**. En ella vamos a definir la función que permitirá al usuario asignar una GPIO al botón. En la versión V5, si lo necesitas, puedes añadir más funciones que dependan del microcontrolador. Vamos a seguir los siguientes pasos:

- Como vamos a hacer uso de funciones de control de las GPIO, vamos a incluir `stm32f4xx.h`, que nos da acceso a los registros.
- Vamos a definir (`#define`) los valores de la GPIO y el pin al que está conectado el botón, y que se indican en la [tabla resumen del botón](#):

- `STM32F4_USER_BUTTON_GPIO`: GPIO a la que está conectada el botón de usuario en la placa. Pon el nombre de la GPIO. Por ejemplo: `GPIOH`, si estuviese conectado a la GPIO H. Este nombre, es el de la estructura de `CMSIS` y se define en `stm32f4xx.h`. Puedes hacer `CTRL + click` sobre el nombre de la GPIO para ir a su definición.
 - `STM32F4_USER_BUTTON_PIN`: pin/ línea de la GPIO del botón. Es un número entero que va de 0 a 15.
 - En el fichero tenemos que declarar también una estructura llamada `stm32f4_button_hw_t` (ver la [figura del struct](#)). Esta estructura se pone en el `.h` y se hace pública para que pueda ser usada por la ISR del botón en el fichero `interr.c`. Esto es así porque a las ISR no se les puede pasar argumentos. En ella se definen los campos que se muestran en la figura.
- Esta estructura es genérica para cualquier botón que vayamos a usar, no solo el de usuario `B1`, sino cualquiera que desee añadir más tarde.
- El campo `flag_pressed` nos indica si el botón se ha pulsado, o no. Ya vimos que cuando pulsamos nuestro botón se produce un flanko de bajada por cómo están conectados sus pines (ver [figura de rebotes](#)). Son las ISR las que hacen esta interpretación.
- Vamos declarar un array de estructuras de tipo `stm32f4_button_hw_t` como `extern` que se definirá en el fichero fuente `stm32f4_button.c`. Este array contendrá las características de todos los botones que tengamos en el sistema.

```
extern stm32f4_button_hw_t buttons_arr[];
```

- Puede ser buen momento ahora para documentar la función, la estructura y sus campos, y los `#define` con Doxygen.

Ya hemos acabado con el encabezado (*header*) que interactúa con el HW del botón y depende del microcontrolador. Todavía dará errores al compilar. Vamos ahora a implementar todas las funciones prototipadas aquí y en `port_button.h`.

3.2.3 `PORT`: fuente de la librería del botón

Si observas la API, verás que la FSM del *botón* hace llamadas a funciones que empiezan por `port`. Estas son funciones *portables*, y el usuario que quiera usar la librería de la FSM del *botón* debe programarlas y adaptarlas a su HW.

Vamos a *portar* las funciones necesarias para usar la librería botón y comprobar que la parte HW está bien programada. Lo haremos, cómo no, para la placa `Nucleo-STM32F446RE`. Ya tenemos las cabeceras HW del botón: las que no dependen del microcontrolador (`port_button.h`) y las que sí (`stm32f4_button.h`). Vamos a programar los ficheros fuente de la parte `PORT`, que **todos estarán en el fichero** `stm32f4_button.c`. Deberás implementar o completar todas las funciones públicas de las que ya has declarado el prototipo en el encabezado. Posteriormente completarás la ISR asociada al pin del botón que aparece en la API del fichero `interr.c`.

Fuentes `stm32f4_button.c`

2. Incluye las librerías necesarias, si falta alguna, según indique la API.
3. Vamos a definir la **variable global** `stm32f4_button_hw_t buttons_arr[]` que se declaró en el `.h`. Se trata de un array que no especifica el número de elementos que tiene, pero cada uno será de tipo `stm32f4_button_hw_t`, que representa al HW de cada botón que tengamos en nuestro sistema.

La declaración e inicialización de este array podemos hacerla a la vez, que será lo más aconsejable. También podríamos hacer solo la declaración e inicializar los valores de cada botón en una función aparte —que ahora mismo no tenemos definida—. Convendría recordar las secciones correspondientes de los vídeos de SDG1: [inicialización de arrays](#) y [arrays de estructuras](#).

Asigna los valores del botón `PORT_USER_BUTTON_ID` utilizando los `#define` de `stm32f4_button.h`. Deberá quedarte algo como lo de la [figura de arrays](#). Si tienes más botones en el sistema, simplemente añadirás una fila para cada botón con sus características correspondientes.

```

/* Global variables */
/**
 * @brief Array of elements that represents the HW characteristics of the buttons connected to the STM32F4
 * platform.
 *
 * This is an **extern** variable that is declared in `stm32f4_button.h` . It represents an array of hardware
 * buttons.
 *
 * @hideinitializer
 */
stm32f4_button_hw_t buttons_arr[] = {
    [PORT_USER_BUTTON_ID] = {.p_port = STM32F4_USER_BUTTON_GPIO, .pin = STM32F4_USER_BUTTON_PIN, .pupd_mode =
        STM32F4_GPIO_NOPULL},
};

```

Array de botones con las características del botón de usuario.

- Se os proporciona ya codificada la función `stm32f4_button_get()`. Se trata de una función privada que devuelve un puntero a la estructura del botón que se le pasa como argumento. Esta función, aunque prescindible, es útil para poder hacer un código más legible y acceder a los campos de la estructura del botón de forma más sencilla desde otras funciones del fichero `stm32f4_button.c`. ¡Recuerda que esta función es privada y por tanto debe aparecer codificada antes de cualquier función que la utilice!

Si el botón no existe, la función devuelve `NULL`. Esto es habitual en funciones que devuelven punteros y es muy útil para detectar errores en la programación.

- Completa la función `port_button_init()` como se indica en la API. Fíjate cómo la parte de código proporcionada define la variable local `*p_button` que nos permite acceder al botón. También podría haberse hecho con acceso directo al elemento del array de botones `buttons_arr[button_id]`, pero es más elegante, seguro, y legible hacerlo con la función `stm32f4_button_get()`.

Recuerda que es muy importante indicar en el modo de la interrupción del botón que, además de detectar ambos flancos (subida y bajada), debe habilitar la petición de interrupción (registro `EXTI_IMR`).

- Codifica la función pública declarada en el fichero `port_button.h` siguiendo la API: `port_button_get_pressed()`.

¡Ya hemos acabado con la implementación de la parte HW `stm32f_button.c`! Ahora solo queda la ISR asociada al pin del botón para poder probarlo. Vamos a ello.

interr.c

Abre el fichero `interr.c` e implementa la ISR `EXTI15_10_IRQHandler`. Hay que implementar la parte correspondiente a la **Versión 1** que indica la API solo.

- Copia este esqueleto en el documento:

```

void EXTI15_10_IRQHandler(void)
{
    /* ISR user button */
    if ((EXTI->PR & BIT_POS_TO_MASK(buttons_arr[PORT_USER_BUTTON_ID].pin))
    {
        ...
    }
}

```

- Completa la ISR `EXTI15_10_IRQHandler` como se indica en la API.

Esta ISR es la misma para cualquier elemento que se conecte en las líneas `10-15`, y por ello debe identificar cuál de ellas ha sido. Es por eso que tenemos la línea `EXTI->PR & BIT_POS_TO_MASK(buttons_arr[PORT_USER_BUTTON_ID].pin)`, para asegurar que es el botón de usuario.

Esta línea comprueba si el bit de la posición del pin del botón está activo en el registro Pending Register `EXTI_PR`. Si es así, es porque se ha producido una interrupción en el pin del botón. Es muy importante que borremos el flag de interrupción escribiendo un 1 en el bit correspondiente del registro `EXTI_PR`.

Debemos identificar si el flanco que ha producido la interrupción es de subida o bajada, pues el montaje HW de cada botón puede ser distinto.

9. Si queda algo por documentar puede ser buen momento ahora.

Si ahora compila, el código no debería tener ningún error. ¡Ya hemos acabado con la implementación de **portado del botón!**. Vamos a probarlo con el *test* unitario de la parte [PORT](#).

3.2.4 [PORT](#): Test unitario del botón

Veremos que la lógica de las máquinas de estado hacen uso de las funciones portables que acceden al HW del dispositivo. Es por ello que es importante comprobar primero que la parte [PORT](#) funciona correctamente. Vamos a hacer el test de HW del código que hemos desarrollado de la librería del botón y probar que funciona antes de continuar con la implementación de la lógica de la parte de la FSM.

Importante! Los test que se proporcionan comprueban solo algunos aspectos esenciales, pero no son exhaustivos. Es responsabilidad del alumno comprobar que el sistema final funciona correctamente.  Ten a mano y revisa el capítulo “Test unitarios y ejemplos de integración” del libro de fundamentos teóricos ³.

En esta sección vamos a practicar con el concepto de **depuración (debugging)** ( recordar ejercicio de clase en SDG1 y las pruebas de la Guía de instalación con [blink](#)⁷).

La herramienta más importante y cómoda que tenemos para depurar es el **IDE** que usemos. En nuestro caso, como **VSCode** no es un **IDE** en sí mismo, sino un editor de texto *vitaminado*, tenemos que usar extensiones. La extensión *Cortex-Debug* para **VSCode** es la que nos ayudará a depurar.

A veces no nos queda otra forma para depurar que imprimir texto por pantalla (cuando esta existe). Recuerde de la *Guía de instalación*⁷ que algunos dispositivos basados en ARM proporcionan una consola de **ITM**. ITM es una aplicación de ARM que permite, entre otras cosas, el uso de la función [printf\(\)](#) en depuración con la placa. También existen alternativas como el *semihosting*, que envía mensajes a través de la pestaña de *Debug Console* de **VSCode**. Esta es la opción que tenemos configurada en nuestro proyecto. **No hay que abusar de ellas, porque no vale para todo. Tendremos que acudir inevitablemente a ver los valores de los registros en algún momento.** Bien es cierto que, una vez el proyecto está funcionando en producción, puede ser útil tener trazas (impresas, LEDs, ficheros de *log*...) para saber qué está pasando en caso de fallo.

Descarga el fichero de test HW del botón [test_port_button.c](#) de https://github.com/sdg2DieUpm/simone/tree/simone_v1_test. Ponlo en la carpeta [test/stm32f4](#) de tu proyecto. Puedes eliminar el fichero plantilla [test_template.c](#)

10. Conecta la placa **Nucleo-STM32** al ordenador.

11. Pulsa sobre el **ícono de depuración**  y selecciona  **Clean and Debug** sobre la plataforma que queramos depurar ([stm32f446re](#)).

12. En el desplegable que se abre, selecciona el test [test_port_button](#). Se compilará y se cargará en la placa.

13. Inmediatamente se habrá parado en la primera línea del test. Continúa la depuración () para ejecutar el test por completo, o pon puntos de parada si deseas ir paso a paso.

14. Se habrá impreso por la terminal del [gdb-server](#) el resultado de las pruebas de los tests. Debería haber pasado todos los tests. Si no, lee el mensaje de error y corrige tu código hasta que pasen todas las pruebas. **Si no pasan las pruebas, no continúes.**

15. La depuración se queda en bucle en la instrucción [exit\(UNITY_END\(\)\)](#). Para terminar la depuración pulsa () y repite el proceso hasta que pasen todos los test.

Si algún test genera una situación de comportamiento inesperado, puede ser que no termine de ejecutarse. Comprueba dónde se queda pausando la depuración, y lee los mensajes de error que te proporciona el test.

Un ejemplo de ejecución del test se muestra en la [ejecución de test de la figura](#). En este caso, la segunda comprobación ha fallado indicándonos que la GPIO elegida para el botón no es la correcta. Luego aparecen más errores, pero son derivados de este. Corrigiéndolo, pasan todos los test correctamente. Por eso, se recomienda **arreglar los errores en el orden en que aparecen**.

```
[stm32f4x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080056d msp: 0x20020000
/home/josueportiz/entornos/MatrixMCU/projects/simone_private/test/stm32f4/test_port_button.c:319:test_identifiers:PASS
/home/josueportiz/entornos/MatrixMCU/projects/simone_private/test/stm32f4/test_port_button.c:53:test_button_wiring:FAIL: Expected 1073
874944 Was 1073873920. ERROR: USER BUTTON GPIO is incorrect
/home/josueportiz/entornos/MatrixMCU/projects/simone_private/test/stm32f4/test_port_button.c:85:test_regs_config_unchanged:FAIL: Expected 640 Was 256. ERROR: GPIO PUPD has been modified for other pins than the button
/home/josueportiz/entornos/MatrixMCU/projects/simone_private/test/stm32f4/test_port_button.c:322:test_write_gpio:PASS
/home/josueportiz/entornos/MatrixMCU/projects/simone_private/test/stm32f4/test_port_button.c:112:test_exti:FAIL: Expected 2 Was 1. ERROR: Button EXTI CR is not configured correctly
/home/josueportiz/entornos/MatrixMCU/projects/simone_private/test/stm32f4/test_port_button.c:324:test_exti_enabled_priority:PASS
/home/josueportiz/entornos/MatrixMCU/projects/simone_private/test/stm32f4/test_port_button.c:325:test_button_port_generalization:PASS

-----
7 Tests 3 Failures 0 Ignored
FAIL
semihosting: *** application exited normally ***
[stm32f4x.cpu] halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x0800511e msp: 0x2001ffd8, semihosting
```

Ejecución de los test unitarios de la parte 'PORT' del botón.

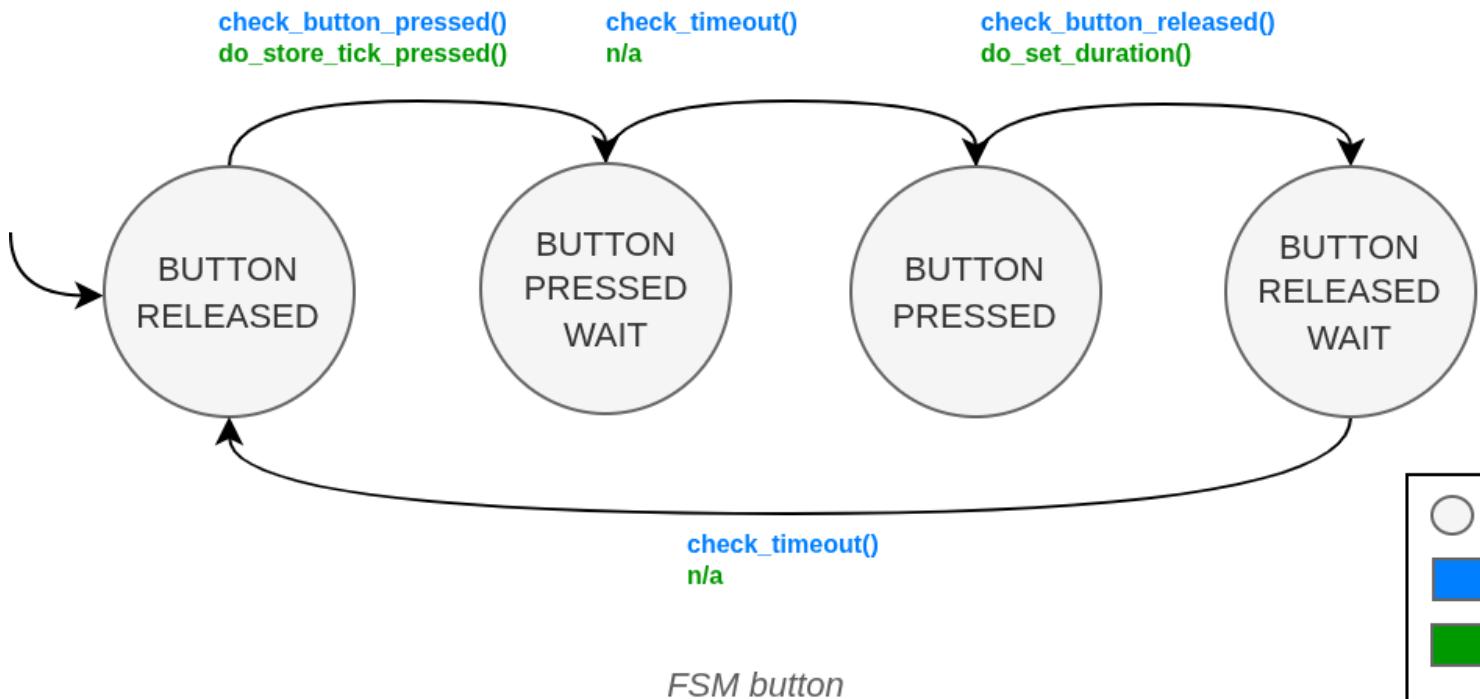
¡Ya hemos acabado con la parte `PORT` del botón! Vamos ahora a implementar la parte `COMMON` de la librería del botón.

3.2.5 `COMMON`: cabecera de la FSM del botón

Consideraciones de la FSM del botón

Antes de empezar vamos a partir de una serie de consideraciones.

- La FSM **almacena la duración de la última pulsación de botón**.
- El usuario debe solicitar/ comprobar la duración mediante la función `fsm_button_get_duration()`.
- El valor de inicio de duración al arrancar la FSM, y el de reinicio, debe ser `\(0 ms)`.
- **Un valor de `\(0 ms)` significa que no ha habido una nueva pulsación del botón**.
- **El usuario debe reiniciar el valor de duración una vez leído**, de lo contrario, este valor puede ser malinterpretado por el usuario si se realizan sucesivas comprobaciones sin haber pulsado el botón. En tal caso estaríamos leyendo información del pasado. Para reiniciar el valor se debe llamar a la función `fsm_button_reset_duration()`.
- Visto de otro modo, **el "flag" de estado de esta FSM es la variable duración**. Una duración de 0 significa que no ha habido ninguna nueva pulsación de botón. Un valor distinto de 0 representa que ha sido pulsado y el valor es su duración. Es por tanto responsabilidad del usuario borrar este "flag" de estado.
- La FSM contiene información del identificador (`ID`) del botón. Este `ID` es único y gestionado por el usuario en el `PORT`. Ahí es donde el usuario proporciona identificadores e información HW (GPIO a la que está conectado y tiempo de anti-rebotes) para todos los botones de su sistema.



Máquina de estados del botón.

Nuestra librería implementa la lógica de la **FSM** mostrada en la **FSM de la figura** y que llamaremos `fsm_button` (en los ficheros `.c` y `.h`). Tiene 4 estados porque **implementa un mecanismo anti-rebotes SW**. Los rebotes de botón o pulsaciones muy rápidas que duren menos que el *tiempo de anti-rebote* (`debounce_time`), se filtran. Los estados, como muestran la figura, son:

- `BUTTON_RELEASED`: es el estado inicial de la FSM. En este estado la FSM está comprobando constantemente si se ha producido un flanco de bajada, *i.e.*, si se ha pulsado el botón. Cuando este se produce, guarda el instante actual y calcula el *timeout* del tiempo de guarda.
- `BUTTON_PRESSED_WAIT`: se queda esperando en este estado hasta que ha pasado el tiempo de guarda de anti-rebotes, *i.e.*, hasta que el instante actual es mayor que el *timeout*. Este es el anti-rebote de bajada. Al salir, no hace nada (función, `NULL`).
- `BUTTON_PRESSED`: en este estado se queda mientras no se suelte el botón. Saldrá de él cuando se haya detectado un flanco de subida. Ya ha pasado el tiempo de guarda, por lo que si es un rebote, se detectará como falso (ver). Cuando el flanco de subida se produce, se calcula la duración de la pulsación.
- `BUTTON_RELEASED_WAIT`: se queda esperando en este estado hasta que ha pasado el tiempo de guarda de anti-rebotes, *i.e.*, hasta que el instante actual es mayor que el *timeout*. Este es el anti-rebote de subida. Al salir, no hace nada (función, `NULL`).

La parte `COMMON` de nuestra librería trabaja con la estructura (`struct`) **pública** que se muestra en la **figura (b) de estructuras**. Con **pública** queremos decir que está declarada en el fichero `.h` y no en el `.c`, por lo que otros ficheros pueden declarar variables de este tipo.

5. Lo primero, descarga del repositorio de la asignatura los ficheros correspondientes a la **parte COMMON** de la librería del *botón* correspondientes a la versión `v1`: https://github.com/sdg2DieUpm/Simone/tree/simone_v1. Solo descarga lo que faltaba por implementar, es decir, los ficheros `fsm_button.h` y `fsm_button.c` y ponlos en las carpetas correspondientes de tu proyecto.

Ahora, vamos a completar la cabecera de la **FSM** del botón, `fsm_button.h`.

6. Incluye las librerías necesarias, si falta alguna, según indique la API.

7. Ahora vamos a definir el enumerado con los nombres de los 4 estados de la FSM. Escribe un `enum` `FSM_BUTTON` con los nombres de los estados del diagrama de la separados por `,`. No olvides poner un `;` al final del `enum`.

8. Seguidamente **declararemos** la estructura `fsm_button_t` para hacerla pública como indica la figura de la estructura de la FSM.
9. Es buen momento para aprovechar a documentar la estructura, del mismo modo que se hizo anteriormente y se muestra en [la figura de comentario](#).

```
typedef struct
{
    fsm_t f;                                /*!< Button FSM */
    uint32_t debounce_time; /*!< Button debounce time in ms */
```

Comentario de Doxygen en un campo de una estructura.

Continuamos con las declaraciones de funciones públicas de la librería. **Procedamos**:

10. Escribe los prototipos de las **funciones públicas** que aparecen en la API del fichero `fsm_button.h`.

Como puedes intuir, estas no son todas las funciones de la librería, sino solo aquellas que podrán ser llamadas desde el exterior. Hemos establecido un criterio general por el que diremos que, si una función va a ser accesible desde el exterior, el nombre de la función debe empezar por `fsm`.

11. Puede ser buen momento ahora para documentar las funciones con Doxygen. En este caso, la documentación va encima del nombre de cada función.

Ya hemos acabado con el encabezado. Quizás de errores al compilar. Vamos ahora a programar el fichero fuente `fsm_button.c`.

3.2.6 **COMMON**: fuente de la FSM del botón

Vamos a proceder con la implementación de las funciones del *botón*. Deberás implementar todas las **funciones públicas** de las que ya has declarado el prototipo en el encabezado, y el resto de **funciones privadas** que aparecen en la API del fichero `fsm_button.c`. También definiremos las variables globales y estructuras que sean necesarias. ¡Recuerda que las **funciones privadas no se declaran en el `.h`!**

12. Lo primero que debe aparecer es la inclusión de cabeceras; en nuestro caso `fsm_button.h`, `port_button.h`, y `port_system.h` como indica la API.

Ahora empezamos a codificar las **funciones privadas de la FSM**. Empezaremos con las **funciones de entrada o comprobación de la FSM**. Hemos establecido un criterio general por el que, **si una función es de entrada o comprobación, será privada y estática, y el nombre de la función va empezar por `check`** (porque comprueba la condición de salto de la máquina de estados). Es muy importante aquí que hayas entendido bien los ejemplos con FSM de los tutoriales  “Capítulo 2. Introducción a las máquinas de estados en C” y “Capítulo 3. Máquinas de Estados Combinadas”.

IMPORTANTES

Si te fijas en la API, todas las funciones de la máquina de estados reciben el mismo argumento: `fsm_t*`, un puntero a una máquina de estados.

```
fsm_button_t *p_fsm = (fsm_button_t *)(p_this);
```

Esto es así porque la librería `fsm.c` no sabe qué tipo de máquina de estados es. Nosotros sabemos que esta máquina de estados es una máquina con asteroides porque incluye, además, la estructura del botón. En realidad es tipo `fsm_button_t*`, un puntero a una máquina de estados de la estructura del botón, que por tener en su primer campo una `fsm_t*`, podemos hacer un cast y convertirla. Así pues, **en las funciones de la máquina de estados, siempre tendremos que recuperar nuestro tipo haciendo este cast** como nos dice la API y hace, por ejemplo, la función `fsm_button_init()`. **En el resto de funciones que no se pasen a la librería `fsm.c`, trabajaremos directamente con el tipo `fsm_button_t*` que hemos creado.**

13. Codifica la función `check_button_pressed()` como se indica en la API.

14. Codifica la función `check_button_released()` como se indica en la API.

15. Codifica la función `check_timeout()` como se indica en la API.

16. Puede ser buen momento ahora para documentar las funciones con Doxygen. En este caso, como las funciones no están declaradas en el encabezado, la documentación irá en el `.c`, encima del nombre de cada función.

Seguiremos con las funciones de salida o actualización de la FSM. Hemos establecido el criterio de que, **una función de salida o actualización será privada y estática, y el nombre de la función empezará por `do_`** (porque va a hacer algo). Hay veces que no hay que hacer nada, por lo que para la librería `fsm.c` será suficiente que apunte a `NULL`. Esto en el diagrama de la se ha representado como `n/a`.

17. Codifica la función `do_store_tick_pressed()` como se indica en la API.

18. Codifica la función `do_set_duration()` como se indica en la API.

19. Documenta las funciones con Doxygen. En este caso, igual que antes, la documentación irá en el `.c`, encima del nombre de cada función.

```
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c: In function 'fsm_button_init':
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c:166:22: error: 'fsm_trans_button' undeclared (first use in this function); did you mean
trans_t?
  166 |     fsm_init(p_this, fsm_trans_button);
      |             ^~~~~~
      |             fsm_trans_t
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c:166:22: note: each undeclared identifier is reported only once for each function it appears on
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c:165:19: error: unused variable 'p_fsm' [-Werror=unused-variable]
  165 |     fsm_button_t *p_fsm = (fsm_button_t *)p_this;
      |             ^~~~~~
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c: At top level:
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c:104:13: error: 'do_set_duration' defined but not used [-Werror=unused-function]
  104 | static void do_set_duration(fsm_t *p_this)
      |             ^~~~~~
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c:84:13: error: 'do_store_tick_pressed' defined but not used [-Werror=unused-function]
  84 | static void do_store_tick_pressed(fsm_t *p_this)
      |             ^~~~~~
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c:62:13: error: 'check_timeout' defined but not used [-Werror=unused-function]
  62 | static bool check_timeout(fsm_t *p_this)
      |             ^~~~~~
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c:43:13: error: 'check_button_released' defined but not used [-Werror=unused-function]
  43 | static bool check_button_released(fsm_t *p_this)
      |             ^~~~~~
/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/common/src/fsm_button.c:25:13: error: 'check_button_pressed' defined but not used [-Werror=unused-function]
  25 | static bool check_button_pressed(fsm_t *p_this)
      |             ^~~~~~
ccl: all warnings being treated as errors
make[3]: *** [CMakeFiles/project.dir/build.make:76: CMakeFiles/project.dir/common/src/fsm_button.c.obj] Error 1
make[3]: Leaving directory '/home/josueportiz/entornos/MatrixMCU/projects/jukebox_v1/build/stm32f446re/Release'
make[2]: *** [CMakeFiles/Makefile2:301: CMakeFiles/project.dir/all] Error 2
```

Error de compilación durante el desarrollo de `fsm_button.c`.

Todavía no hemos acabado con el desarrollo, pero vamos a compilar para ir depurando errores. Compile el programa. Verá errores parecidos a los que se muestran en la [figura de error](#). El compilador a veces nos da alguna sugerencia de corrección, pero no tienen por qué ser correctas. Nos dice que:

- Hay una variable que no está declarada y que se llama `fsm_trans_button`. Se trata de la tabla de transiciones. Aún tenemos que escribirla.
- Todas nuestras funciones `check_` y `do_` están declaradas pero no se usan.

¿Por qué no dice lo mismo de las funciones públicas que están declaradas en el `.h` y tampoco se usan? Pues porque son públicas. El compilador no sabe quién las podrá usar y ahí están declaradas para quién la pueda llamar. ¿Por qué en el `.c` da error? Porque en el [Makefile](#) —donde están las reglas de compilación—, igual que tenemos el *flag* para que nos dé error si la variable no se usa, lo mismo pasa con las funciones. Esto es útil para evitar generar códigos sucios con funciones que no sirven.

¿Quién va a usar las funciones de la máquina de estados? Pues las va a usar, mediante indirección, la librería `fsm.c`. ¿Cómo? Porque le vamos a pasar a la función `fsm_fire()` la tabla de transiciones. Y es ahí, donde vamos a “usar” estas funciones. ¿Y dónde se llama a `fsm_fire`? Ya lo haremos, pero se hace en `fsm_button_fire()`, que a su vez es llamada por el `main`.

- Si las usásemos, aparecería el error de que hay funciones que no están declaradas (*implicit declaration*). Son las llamadas a funciones del `PORT`, que más tarde codificaremos.

Mucha información hasta ahora, pero verá cómo se aclara todo enseguida. **Vamos a codificar la tabla (array) de transiciones de la FSM del botón.**

20. Definimos `static fsm_trans_t fsm_trans_button[] = ...` justo después de la función `do_set_duration()`.

Recuerda que cada fila de la tabla de transiciones tiene la forma: `EstadoIni, FuncCompruebaCondicion, EstadoSig, FuncAccionesSiTransicion`. No olvides añadir la fila `-1, NULL, -1, NULL` que sirve a la librería `fsm.c` para detectar el fin de la tabla.

¿Por qué es importante haber colocado la tabla en este punto? Bueno, puede ser en cualquier punto después de las funciones `check_` y `do_`, porque de lo contrario, de estar más arriba, al compilar, la tabla de transiciones estaría haciendo referencia a funciones que todavía no se sabe que existen (¡porque son privadas y no están en el `.h`!).

Si ahora compilas, verás que han desaparecido muchos errores. Ya queda menos para acabar la implementación de la librería de la máquina de estados. Prosigamos:

21. Codifica la función `fsm_button_get_duration()` como se indica en la API. Esta función nos servirá en el programa principal para preguntar cuánto tiempo ha durado la pulsación.

22. Codifica la función `fsm_button_reset_duration()` como se indica en la API. Con esta, reiniciaremos el valor de la duración a \0 ms) tras leerlo.

23. Codifica la función `fsm_button_get_debounce_time_ms()` como se indica en la API. Esta función será usada en los test unitarios.

24. Completa la función `fsm_button_init()` como se indica en la API.

Fíjate en las funciones `fsm_button_fire()` dada, que implementaremos también sucesivas versiones. Esta función sirve para lanzar la *máquina de estados* (`f`). Será usadas en el test unitario de la máquina de estados, en el `main.c`, y son útiles para depurar.

Ya hemos acabado con la programación de la librería del botón. Toda esta lógica `COMMON` **puede ser usada en cualquier sistema**, esté basado en microcontrolador, o sea un PC. Hemos hecho una librería de un botón que tiene un anti-rebotes y nos devuelve la duración de la última pulsación. Así pues, si compilas, no deberán aparecer errores.

25. Documenta el código que esté sin comentar.

3.2.7 `COMMON` Test unitario de la FSM del botón

Vamos a probar el test del código que hemos desarrollado de la librería de la máquina de estados del botón y probar que funciona antes de continuar con la siguiente versión. **¡Importante! Recuerda que los test que se proporcionan comprueban solo algunos aspectos esenciales, pero no son exhaustivos. Es responsabilidad del alumno comprobar que el sistema final funciona correctamente.**

Descarga el fichero de test de la FSM del botón `test_fsm_button.c` de https://github.com/sdg2DieUpm/simone/tree/simone_v1_test. Ponlo en la carpeta `test/` de tu proyecto. **¡No lo metas en stm32f4!, pues no es un test específico del microcontrolador!**

26. Con la placa **Nucleo-STM32** conectada al ordenador.

27. Pulsa sobre el **ícono Clean and Debug** sobre la plataforma que queramos depurar (`stm32f446re`).

28. En el desplegable que se abre, selecciona el test `test_fsm_button`. Se compilará y se cargará en la placa.

29. Ejecuta el test por completo, o pon puntos de parada si deseas ir paso a paso.

30. Se habrá impreso por la terminal del `gdb-server` el resultado de las pruebas de los tests. Debería haber pasado todos los tests. Si no, lee el mensaje de error y corrige tu código hasta que pasen todas las pruebas. **Si no pasan las pruebas, no continúes.**

31. Termina la depuración pulsando (`Q`) y repite el proceso hasta que pasen todos los test.

3.2.8 Ejemplo de uso de la Versión 1

Comúnmente llamado test de integración, es un código de ejemplo que consiste en probar la librería en el sistema final, o en una versión particular, con el resto de librerías y módulos. El test de integración no hace uso de la librería `unity`, sino que es como un pequeño programa de prueba sobre las funciones que hemos implementado. Tiene su propio `main`.

En los test de integración es responsabilidad del alumno comprobar que la funcionalidad es la esperada, porque aquí no hay test unitarios que nos ayuden.

Nuestra librería de botón devuelve la duración de la última pulsación. Así pues, algunas de las comprobaciones que podemos hacer son: que la duración de la pulsación es la que esperamos, que se reinicia adecuadamente el valor, que funciona el anti-rebotes...

Descarga el fichero de ejemplo [example_v1.c](#) de https://github.com/sdg2DieUpm/simone/tree/simone_v1_test. Ponlo en la carpeta [example/](#) de tu proyecto.

Procedamos:

32. Con la placa **Nucleo-STM32** conectada al ordenador.
33. Pulsa sobre el **ícono de depuración** y selecciona **Clean and Debug** sobre la plataforma que queramos depurar ([stm32f446re](#)).
34. En el desplegable que se abre, selecciona el test [example_v1](#). Se compilará y se cargará en la placa.
35. Se parará en la primera línea del [main\(\)](#). Ejecuta el test por completo, o pon puntos de parada si deseas ir paso a paso. Este código no termina, pues es un bucle [while](#) infinito.
36. Abre la terminal del [gdb-server](#) para ver los mensajes que se van imprimiendo.
37. Pulsa el botón de usuario [B1](#) de la placa. Deberás ver que se imprime por pantalla la duración de la pulsación. Si no es así, revisa tu código.
38. Haz distintas pruebas y asegúrate de que el comportamiento es el adecuado.

¡Hemos creado nuestra primera librería! Fíjate que es *portable* a cualquier plataforma solo con adaptar las funciones del [PORT](#).

No dejes de documentar el código. **Comprueba que la documentación del código se ha generado correctamente como se explica en la “Guía de instalación de herramientas para compilación multiplataforma en C”**⁷, o en el video “[MatrixMCU] Documentación de código con Doxygen”.

Guarda una copia de su proyecto como [simone_v1](#) para tener un punto de partida para la siguiente versión, y una copia de seguridad por si algo falla.

39. No nos hacemos responsables de posibles daños que puedas ocasionar . ↩
40. Se recomienda documentar sus códigos siempre en inglés, aunque su nivel no sea muy bueno, porque si trabajamos en una empresa, no sabemos quién lo tendrá que leer. ↩
41. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, and Juan José Gómez Valverde. *Fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/88460/>. ↩ ↩ ↩
42. STMicroelectronics. Stm32f446xc/e. Technical Report, STMicroelectronics, 2021. URL: <https://www.st.com/resource/en/datasheet/stm32f446re.pdf>. ↩
43. STMicroelectronics. Rm0390 reference manual. stm32f446xx advanced arm-based 32-bit mcus. Technical Report, STMicroelectronics, 2021. URL: https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. ↩
44. Román Cárdenas Rodríguez, Josué Pagán Ortiz, Alberto Boscá Mojena, Iván Martín Fernández, and Sergio Esteban Romero. *Tutoriales sobre los fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Román Cárdenas Rodríguez, Madrid, March 2025. URL: <https://oa.upm.es/88470/>. ↩
45. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, Amadeo de Gracia Herranz, Sergio Esteban Romero, and Daniel Capellán Martín. *Guía de instalación de herramientas para compilación multiplataforma en C. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/92376/>. ↩ ↩ ↩

3.3 Versión 2: teclado matricial

En la Versión 1 aprendimos a gestionar un único botón mediante interrupciones. Sin embargo, ¿qué ocurre si nuestro sistema necesita 12, 16 o más botones? Si utilizáramos la estrategia anterior, necesitaríamos una línea de interrupción y un pin GPIO por cada botón, agotando rápidamente los recursos del microcontrolador.

Para solucionar esto, utilizamos la técnica del **barrido** o *scanning* en un teclado matricial. Esta técnica aprovecha la persistencia temporal para leer muchos pulsadores utilizando pocos pines, organizándolos en filas y columnas.

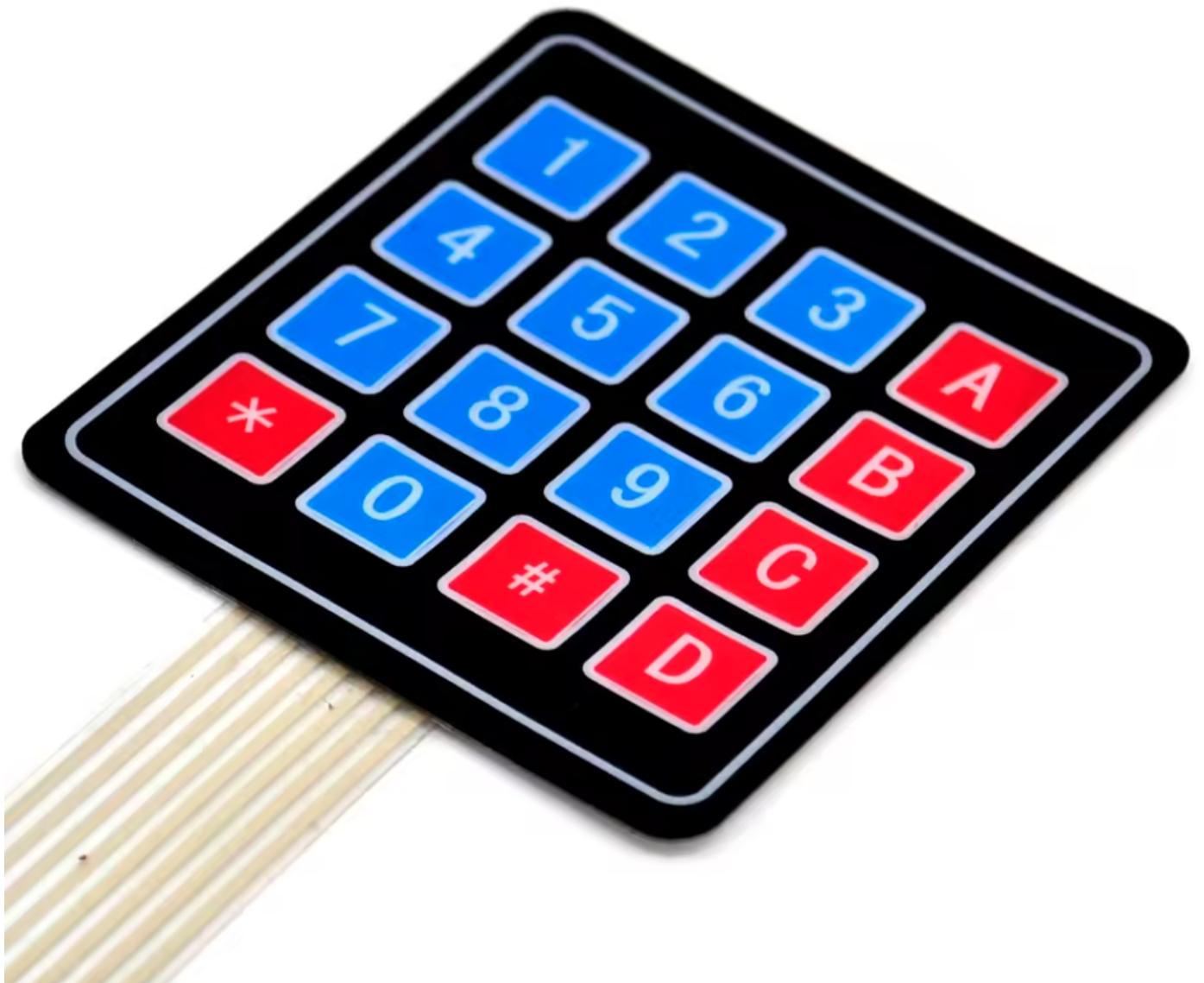
Bibliografía

1. “Fundamentos teóricos de sistemas basados en microcontrolador STM32”³
2. Datasheet “STM32F446xC/E”⁴
3. Reference manual “RM0390. STM32F446xx advanced Arm-based 32-bit MCUs”⁵

Videos del canal de SDGII

- Demostración Simone
- Blink LED y manejo de proyecto
- Conceptos básicos de C (canal SDG1)
- [MatrixMCU] Documentación de código con Doxygen

En este capítulo vamos a crear una librería que nos permita gestionar un teclado matricial 4x4 (16 teclas) [como el de la figura](#). A diferencia del botón, que funcionaba solo teníamos una interrupción, aquí **utilizaremos un temporizador que nos interrumpe para realizar una excitación periódica de las filas y esperaremos interrupciones de las columnas**.



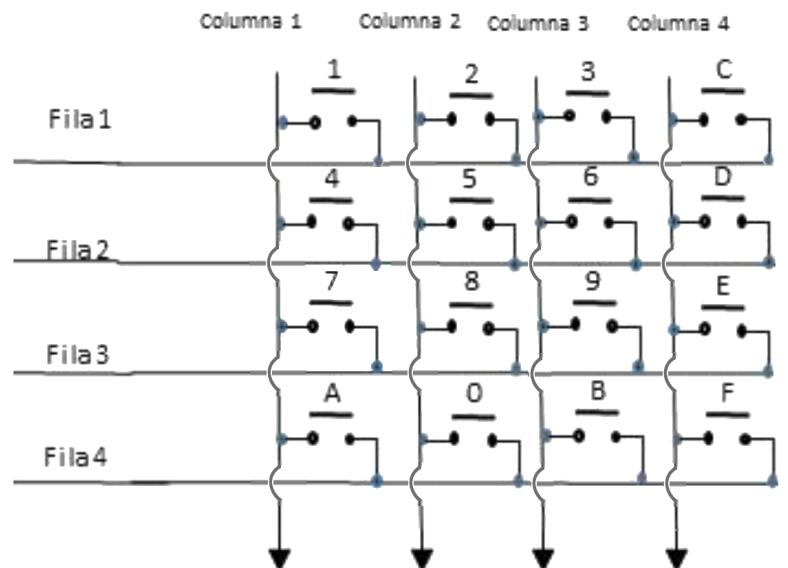
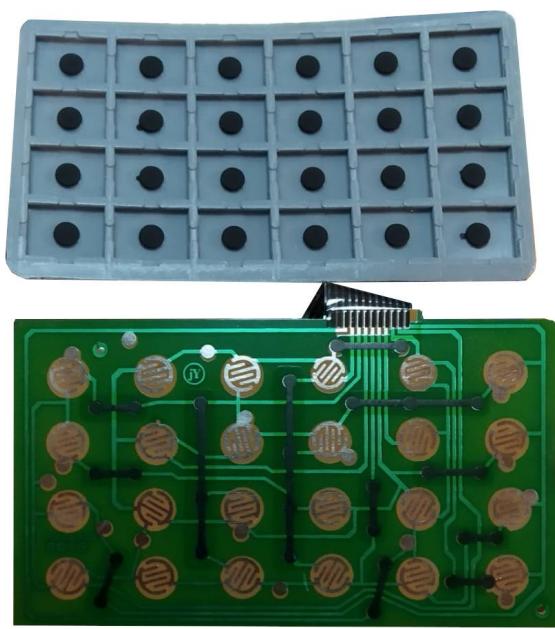
Teclado matricial de membrana 4x4 usado en el proyecto Simone.

Como ya hicimos en las versiones anteriores, (i) vamos a implementar la parte `portable` (`PORT`) dependiente del **HW** para gestionar los niveles lógicos de las filas y la lectura de las columnas, y lo probaremos con un test unitario. (ii) Despues, vamos a crear la lógica de la **FSM** para gestionar el barrido y el *anti-rebotes* (la parte `COMMON`), y lo probaremos con un test unitario. (iii) Por ultimo, montaremos el **HW** y probaremos el funcionamiento con un programa de ejemplo.

El teclado matricial es un array de pulsadores conectados en intersecciones de filas y columnas. Cuando no se pulsa ninguna tecla, no hay conexión entre filas y columnas. Al pulsar una tecla, se cortocircuita una fila con una columna específica. Las características a destacar del sistema de la Versión 2 se muestran en la [siguiente tabla](#).

Parámetro	Valor
Pin fila 1	PA0
Pin fila 2	PA1
Pin fila 3	PA4
Pin fila 4	PB0
Modo filas	Salida
Pull up/ down filas	No push, no pull
Timeout de excitación de filas	\(25 ms\)
Pin/ EXTI/ ISR columna 1	PA8 / EXTI8 / EXTI9_5_IRQHandler
Pin/ EXTI/ ISR columna 2	PB10 / EXTI10 / EXTI15_10_IRQHandler
Pin/ EXTI/ ISR columna 3	PB4 / EXTI4 / EXTI4_IRQHandler
Pin/ EXTI/ ISR columna 4	PB5 / EXTI5 / EXTI9_5_IRQHandler
Modo columnas	Entrada
Pull up/ down columnas	Pull down
Prioridad todas las columnas	1
Subprioridad todas las columnas	1
Tiempo anti-rebotes todas las columnas	\(100-200 ms\)

3.3.1 Características del teclado matricial en Versión 2



(a) Circuito de un teclado matricial de membrana, (b) Esquema de conexiones.

Si tiene la oportunidad de abrir en casa cualquier sistema que tenga un teclado o botonera¹, seguramente encuentre una circuitería como la de la figura (a). La goma gris es la cara interna de los botones, que está sobre la PCB verde. Las almohadillas negras que ve son contactos metálicos que, cuando se pulsa el botón, se cortocircuitan con el metal de la PCB y cierran el circuito².

La idea de colocar los botones así, haciendo una rejilla, es muy inteligente. Haciendo un enrejillado no es necesario tener un cable para cada botón, porque un cable por cada botón implicaría tener un pin de entrada en nuestro microcontrolador por cada uno (si no se usan multiplexores, claro), y los pines no es algo que sobre, generalmente, en los encapsulados de los chips. Por ejemplo, en el teclado de la figura (a) se usan 10 cables para 24 botones (ahorro del \((58.3\%)\) de conexiones/pines), y en el del teclado del laboratorio se usarán 8 conexiones para 16 botones (ahorro del \((50.0\%)\)).

Pero esta idea no sale gratis. A cambio de reducir el hardware necesario, tenemos que complicar el software un poquito. Debemos ir excitando —poniendo tensión— las filas o columnas de forma cíclica para poder detectar qué botón se ha pulsado (lo hacemos leyendo las columnas o filas respectivamente). Si alguna columna detecta esa tensión, sabemos qué tecla exacta (intersección fila-columna) se ha pulsado. Este proceso se repite para todas las filas rápidamente. **En nuestro caso excitaremos filas**. Fíjate que solo podemos tener una fila con tensión a la vez porque, de otro modo, no seríamos capaces de distinguir entre los botones de una misma columna. Si haces un dibujo, lo verás fácilmente.

No te preocunes, la gestión de filas y columnas tiene fácil solución si trabajamos con las máquinas de estado; pues para controlar la excitación de las filas del teclado matricial tendremos la [FSM del teclado](#).

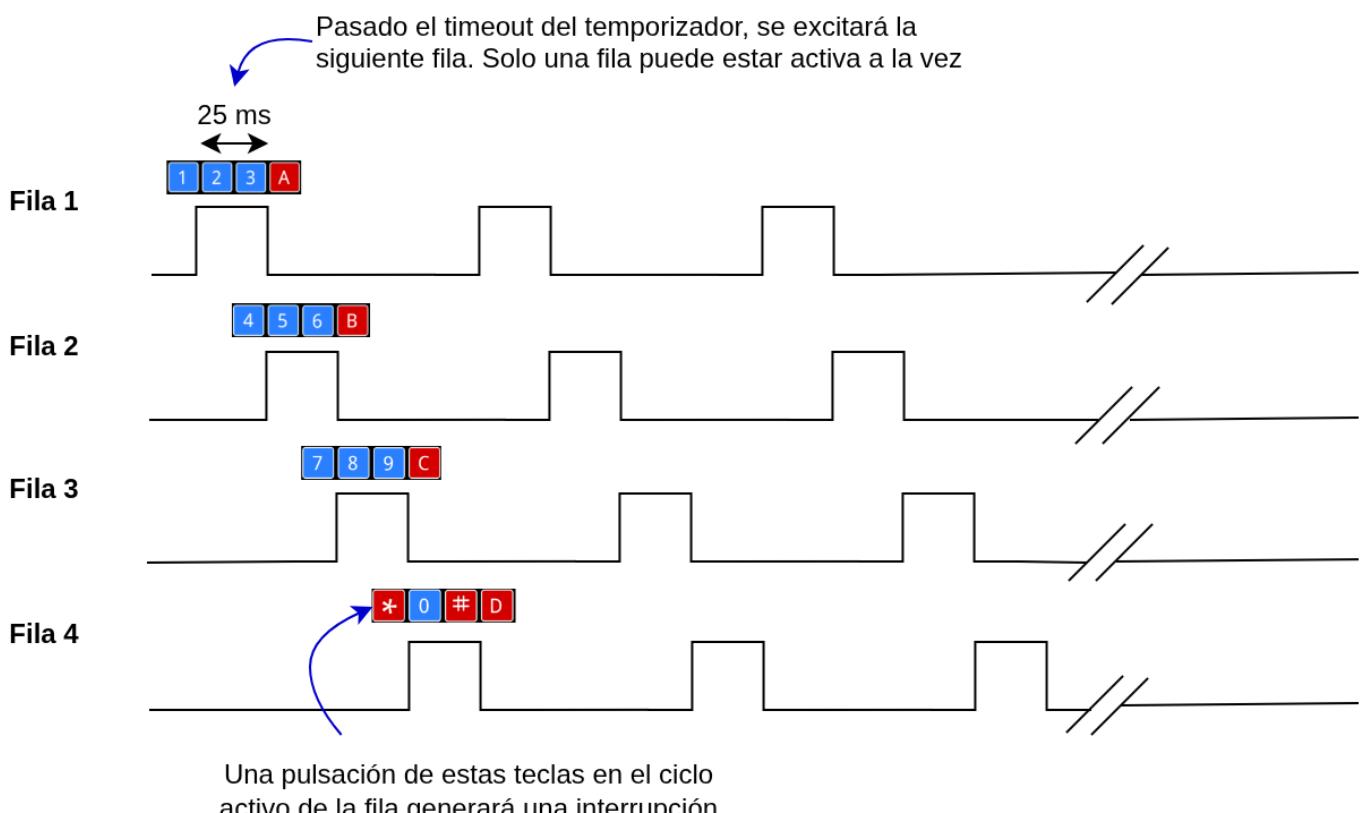
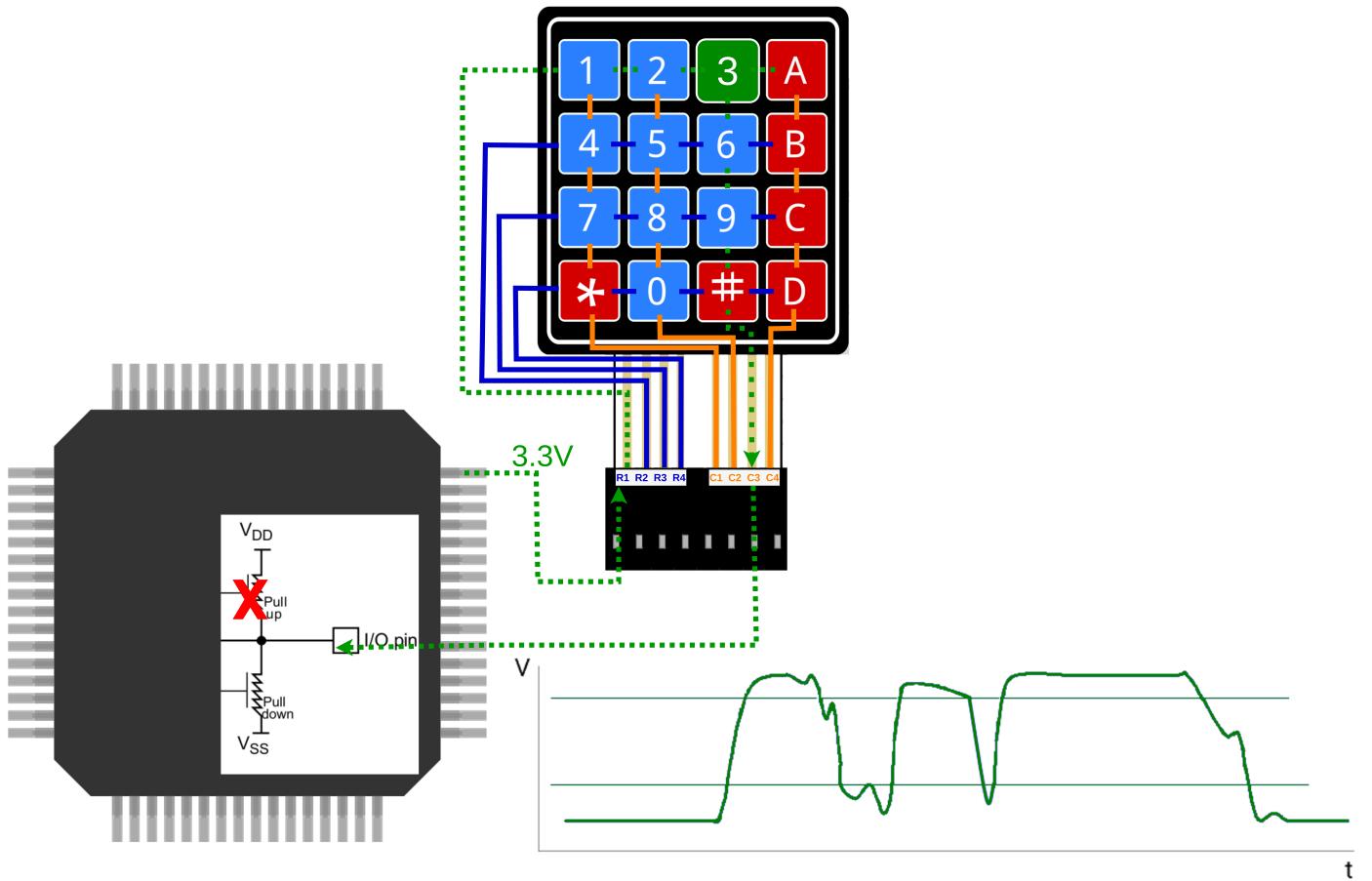


Diagrama de temporización de excitación de filas.

El [diagrama de temporización](#) muestra cómo se excitan las filas del teclado. Cada \((25\text{ ms})\) se excita una fila diferente poniendo un nivel lógico alto en el pin correspondiente. Durante ese tiempo que está la fila en alto, se puede producir interrupción si se pulsa alguna tecla de esa fila. Luego, se desactiva la fila (nivel lógico bajo) y se excita la siguiente fila. Así, en un período de \((100\text{ ms})\) se excitan todas las filas del teclado. Este tiempo es suficientemente rápido para que el usuario no note que las filas se excitan de forma secuencial.



Ejemplo de excitación de fila R1, pulsando tecla 3, generando interrupción en pin conectado a columna C3.

La figura de ejemplo muestra un ejemplo de cómo funcionará el sistema. En este caso, la FSM del teclado está excitando la fila R1 poniendo un nivel lógico alto en el pin correspondiente (cada 25 ms) excita una fila). Cuando se pulsa la tecla '3', se cierra el circuito entre la fila R1 y la columna C3, lo que hace que el pin conectado a C3 detecte un nivel lógico alto (3.3 V).

En este circuito vamos a activar las resistencias de pull-down ¡internas! de los pines del microcontrolador conectados a las columnas, por lo que cuando no se pulsa ninguna tecla, las columnas estarán a nivel lógico bajo (0 V). Al pulsar la tecla '3', la columna C3 pasa a nivel lógico alto debido a la conexión con la fila R1 que está excitada. **Pasa lo contrario que con el circuito del botón, que tenía una resistencia de pull-up ¡externa!** Esto genera una interrupción en el microcontrolador, que puede entonces identificar qué tecla se ha pulsado basándose en la fila actualmente excitada y la columna que ha generado la interrupción.

Recuerda que, como hicimos con el botón, **estamos desarrollando una librería**. El teclado matricial no tiene por qué saber nada de las acciones que hace el sistema cuando se pulsa una tecla. Es por eso que en nuestro proyecto, el teclado se encargará solo de guardar la tecla pulsada y de avisar de que ha habido una pulsación. El sistema que use esta librería —sea en este proyecto u otro— deberá comprobar dicho valor guardado con un *get*. La idea es exactamente la misma que la que implementó en el botón. Así,

cada vez que se quiera añadir un teclado, le asociaremos una **FSM**. Las particularidades de dónde están conectadas las filas y columnas son cosas específicas del HW, por lo que estarán en **PORT**.

stm32f4_keyboard_hw_t	
const keyboard_t	*p_layout;
GPIO_TypeDef	**p_row_ports;
uint8_t	*p_row_pins;
GPIO_TypeDef	**p_col_ports;
uint8_t	*p_col_pins;
bool	flag_key_pressed;
bool	flag_row_timeout;
uint8_t	col_idx_interrupt;
uint8_t	current_excited_row;

fsm_keyboard_t	
fsm_t	f;
uint32_t	debounce_time_ms;
uint32_t	next_timeout;
uint32_t	tick_pressed;
char	key_value;
uint32_t	keyboard_id;

(a) Estructura del HW del teclado en **PORT**, (b) Estructura de la **FSM** del teclado en **COMMON**.

Las **figuras de estructuras HW y SW** muestran las estructuras que vamos a necesitar para el teclado. La estructura del HW del teclado en **PORT**. El **PORT** de otro microcontrolador podría implementar internamente una estructura diferente, por eso está dentro de la carpeta **stm32f4**. Por ejemplo, al *portar* el código para PC no tendría sentido definir la estructura de una GPIO. La estructura de la **FSM** del teclado en **COMMON** se muestra en la [figura de la FSM](#).

Como en el caso del botón aquí, aunque no hay muelles, puede haber igualmente inestabilidades en la pulsación, **rebotes**, por lo que vamos a implementar, igualmente, un mecanismo antirebotes. Dejaremos unos tiempos de guarda de antirebotes también entre \((100-200 ms)\).

En el caso del teclado matricial no tenemos circuito preestablecido, por lo que podemos elegir libremente si queremos usar resistencias de *pull-up* o *pull-down* en las columnas. **En este caso, usaremos resistencias de *pull-down* en las columnas y excitaremos las filas poniendo un nivel lógico alto.** Así, cuando se pulsa una tecla, la columna correspondiente pasa a nivel **alto**. Esto es al revés de como pasaba en el botón.

Ahora sí, comencemos. Preparamos el proyecto para poder añadir el *teclado matricial*:

1. Descarga del repositorio de la asignatura los ficheros correspondientes a la parte **PORT** de la librería del *keyboard* correspondientes a la versión **v2**: https://github.com/sdg2DieUpm/simone/tree/simone_v2. Solo descarga por ahora: **port_keyboard.h**, **stm32f4_keyboard.h**, y **stm32f4_keyboard.c** y colócalos en las carpetas correspondientes. **De la parte COMMON descarga solo keyboards.h**, y **keyboards.c**, que incluyen los *layouts* de los teclados matriciales.
2. Coloca cada uno donde corresponde: **PORT** o **COMMON**, en **include**, o **src**. **Ten en cuenta que algunos ficheros de PORT están en la carpeta stm32f4 porque sus funciones reciben o devuelven estructuras específicas de la Nucleo-STM32F446RE.**

Verás que no compila, y es que solo se te proporciona cierta parte del código. Los prototipos de gran parte de las funciones públicas no están definidos.

3.3.2 Layouts de teclados matriciales

Antes de ponernos a programar, conviene explicar qué son los ficheros `keyboards.h` y `keyboards.c`. Estos ficheros se han de colocar en la carpeta `common/include` y `common/src` respectivamente. Estos ficheros contienen los *layouts* de los teclados matriciales que queramos usar en nuestro proyecto. Un *layout* es una matriz que define qué carácter representa cada tecla del teclado. Por ejemplo, en un teclado 4x4 típico, la primera fila podría representar los caracteres '1', '2', '3', 'A'; la segunda fila '4', '5', '6', 'B'; y así sucesivamente, pero otro de 4x4 también podría tener una distribución diferente (solo letras, o solo números), ¡o tener otro de 1x3 de colores!...

Estos ficheros permiten definir múltiples *layouts* para diferentes teclados matriciales, facilitando su uso en la librería del teclado. En la [Versión 5](#) podrías querer añadir un nuevo *layout* de algún teclado extra, o modificar el existente.

La estructura `keyboard_t` definida en `keyboards.h` contiene:

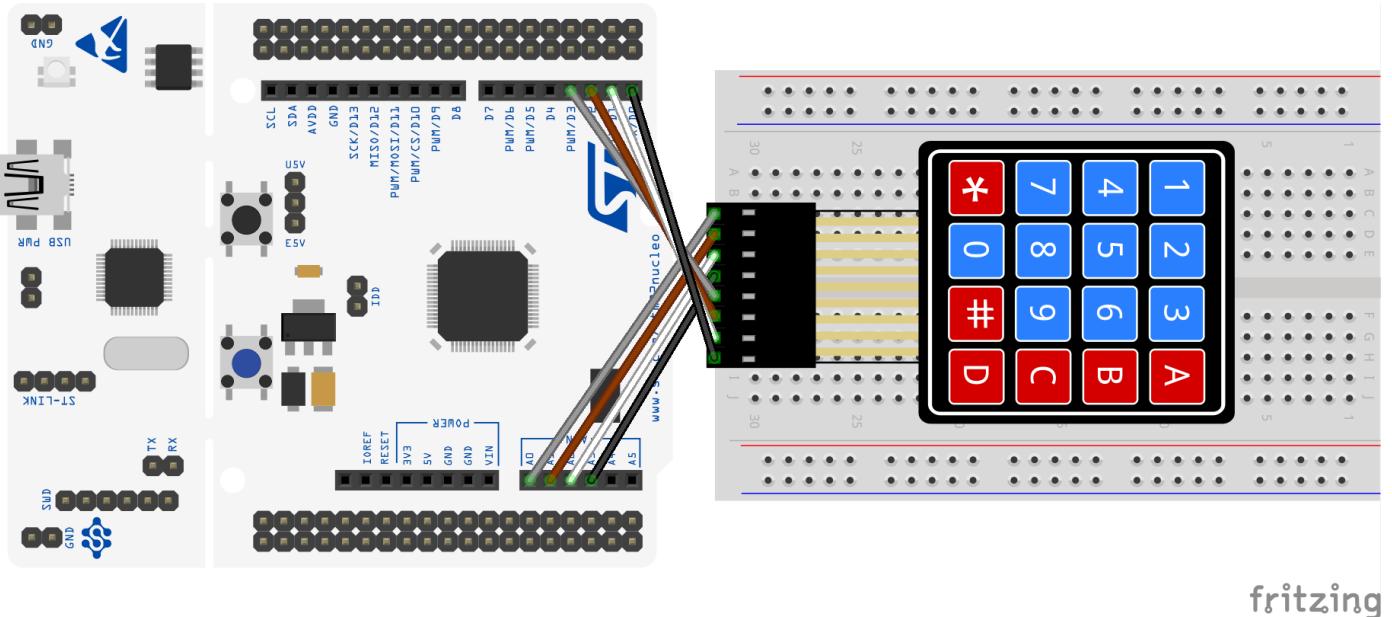
- Un puntero a una matriz de caracteres (`const char *keys`), que representa el *layout* del teclado.
- El número de filas (`uint8_t rows`) y columnas (`uint8_t cols`) del teclado.
- Un carácter especial (`char null_key`) que indica que no se ha pulsado ninguna tecla. Por ejemplo el carácter ASCII nulo `'\0'`.

En el mismo fichero se declara `standard_keyboard` como un ejemplo de *layout* para un teclado matricial 4x4. El nombre es algo genérico y representativo. Este *layout* se define en `keyboards.c` como una matriz de caracteres que representa las teclas del teclado. Aquí se hace público para que pueda ser usado en otras partes del código; sus particularidades se definen en el `.c`.

3.3.3 `PORT`: cabeceras de la librería del teclado

Vamos a implementar el *contrato con el usuario* de la parte dependiente del HW de librería del teclado. Esta interfaz permitirá configurar las filas y columnas de nuestro teclado y realizar el barrido de excitación de filas y lectura de columnas para identificar la tecla pulsada.

El montaje de nuestro módulo teclado matricial tendrá un aspecto como el mostrado en la [figura](#).



Montaje del teclado matricial con la [Nucleo-STM32F446RE](#)

Más adelante lo implementaremos.

Cabecera port_keyboard.h

Esta cabecera depende del HW pero no de las particularidades del microcontrolador **STM32F446RE**. Vamos a seguir los siguientes pasos:

4. Incluye todas las cabeceras necesarias según indica la API.
5. Incluye los (`#define`) necesarios para el teclado: el identificador `PORT_KEYBOARD_MAIN_ID` que usaremos en el proyecto Simone, el *timeout* de excitación de filas, y el tiempo de antirebotes de las teclas (**mismo tiempo para todas**).

Hemos decidido darle el nombre `PORT_KEYBOARD_MAIN_ID` al teclado que usaremos en el juego. En la [Versión 5](#) podríamos querer añadir más teclados, y entonces habría que definir más identificadores con otros nombres representativos. 3. Define el enumerado que identifica los índices de las columnas del teclado, y que será de utilidad para hacer el código más legible en el manejo de las interrupciones. 4. Escribe los prototipos de las funciones públicas que aparecen en la API del fichero `port_keyboard.h`. 5. Puede ser buen momento ahora para documentar con Doxygen.

Cabecera stm32f4_keyboard.h

Esta cabecera define los pines físicos a los que están conectadas las filas y columnas de los teclados que usemos con nuestra placa [Nucleo-STM32F446RE](#).

6. Incluye todas las cabeceras necesarias.
7. Define (`#define`) los valores de las GPIO y pines para las 4 filas y las 4 columnas según indica la [tabla de características del teclado matricial](#).
8. Declara la estructura `stm32f4_keyboard_hw_t` que contendrá la configuración física del teclado.

Presta atención a la documentación que explica en cada campo qué deberá contener. Recuerda que las filas son salidas y las columnas entradas.

Especial mención merecen los campos `p_row_ports` y `p_col_ports`, que **son dobles punteros: punteros a arrays de punteros a estructuras** `GPIO_TypeDef`. Estos campos permiten almacenar las referencias a los puertos GPIO de cada fila y columna del teclado. Esta es la forma de poder definir una estructura genérica sin saber el tamaño del teclado que va a gestionar. Nos da lo mismo que definamos un teclado de 2x2, 4x4 o 5x3; por eso estos campos son punteros que apuntarán a arrays de elementos de tipo `GPIO_TypeDef*` (por ejemplo, `GPIOA`, `GPIOB`, etc.). Así, podemos tener una lista dinámica de puertos para las filas y columnas del teclado, adaptándonos a cualquier configuración física que necesitemos.

El campo `p_layout` es un **puntero a una estructura** `keyboard_t`, que contiene el **layout del teclado**. Esto nos permite asociar el diseño lógico del teclado con su configuración física. Se declara `const` porque el *layout* no debe modificarse en tiempo de ejecución.

Fíjate también que **la estructura NO guarda la tecla pulsada**, sino que guarda el índice de la fila que se está excitando y la columna que genera la interrupción. La gestión de la tecla pulsada se hará en la FSM del teclado, que es independiente del HW.

9. Declara el array de estructuras de tipo `stm32f4_keyboard_hw_t` como se hizo con el botón. Este array contendrá las características de todos los teclados que tengamos en el sistema.
10. Documenta todo con Doxygen.

Ya hemos acabado con el encabezado (*header*) que interactúa con el HW. Todavía dará errores al compilar. Vamos ahora a implementar todas las funciones prototipadas en `port_keyboard.h`.

3.3.4 `PORT`: fuente de la librería del teclado

Vamos a *portar* las funciones necesarias para controlar los pines del teclado. Programaremos los ficheros fuente de la parte `PORT`, que **todos estarán en el fichero** `stm32f4_keyboard.c`.

Fuentes stm32f4_keyboard.c

La complejidad aquí reside en la gestión de múltiples pines y en la lógica de configuración de entrada/salida.

11. Incluye las librerías necesarias.
 12. Verás que en la plantilla proporcionada se han definido 4 arrays que corresponden con los puertos y pines de las filas y columnas del teclado. Estos arrays se usan para inicializar la estructura del teclado. Estos son los arrays de elementos `GPIO_TypeDef*` (y `uint8_t`) de los que hablábamos anteriormente y a los que apuntará la estructura `stm32f4_keyboard_hw_t` del teclado principal `KEYBOARD_MAIN`.
 13. Define la **variable global privada** `stm32f4_keyboard_hw_t keyboards_arr` con la configuración física de nuestro teclado tal y como hicimos con el botón. Los campos `p_row_ports`, `p_row_pins`, `p_col_ports`, y `p_col_pins` deben apuntar a los arrays definidos en el paso anterior. El campo `p_layout` **debe apuntar a la dirección de memoria** del *layout* del teclado estándar definido en `keyboards.c`. El resto de campos se inicializan en la función `port_keyboard_init()`.
 14. Codifica la función `_stm32f4_keyboard_get()` para recuperar la configuración del hardware, de modo análogo a como se hizo para el botón.
 15. Codifica la función `port_keyboard_init()` como se indica en la API.
- Recuerda que es muy importante indicar en el modo de las interrupciones de las columnas del teclado que, además de detectar ambos flancos (subida y bajada), debe habilitar la petición de interrupción (registro `EXTI_IMR`).**
- Se recomienda usar bucles para configurar las filas y columnas, y evitar el *spaghetti code*. Esto hace que el código sea más compacto y fácil de leer y mantener.
16. Codifica la función `port_keyboard_excite_row()` como se indica en la API. Esta función debe **activar la fila indicada y desactivar las restantes!**
 17. Codifica la función `port_keyboard_get_cols()` como se indica en la API. Esta función llama a la anterior pero antes actualiza `current_excited_row`, que es el índice de la fila a ser excitada.
 18. Codifica todos los *getters* y *setters* que aparecen en el `.h` como indica la API. Especial atención a la función `port_keyboard_get_key_value()`, que debe devolver el carácter ASCII correspondiente a la tecla pulsada, usando el *layout* del teclado.

Como en nuestra estructura **no tenemos definida de manera fija el array del layout del teclado**, sino que tenemos un puntero a una estructura `keyboard_t`, no podemos acceder directamente al array de teclas tratado como matriz bidimensional con los índices de la fila y columna. En su lugar, **debemos tratar el array como unidimensional** -que es, por otro lado, como está almacenado en memoria- y calcular la posición del carácter de la tecla pulsada usando la fórmula:

```
key index = (excited row * num columns) + column interrupting
```

Vamos a codificar ahora las funciones que gestionan el temporizador que controla la excitación de las filas.

Parámetro	Valor
Temporizador	<code>TIM5</code>
Prescaler	(a calcular para <code>PORT_KEYBOARDS_TIMEOUT_MS</code>)
Periodo	(a calcular para <code>PORT_KEYBOARDS_TIMEOUT_MS</code>)
ISR	<code>TIM5_IRQHandler</code>
Prioridad	2
Subprioridad	0

19. Codifica la función `_timer_scan_column_config()` como indica la API. Esta función configura el temporizador que controla el tiempo de excitación de las filas de cualquier teclado que se monte en el juego; si hubiese más de uno, todos se excitarían a la vez. Para ello, apóyate en el ejemplo "timer para interrupción periódica" del libro de fundamentos teóricos³.

Esta función configura un temporizador para que genere una interrupción de `PORT_KEYBOARDS_TIMEOUT_MS` milisegundos desde que se habilita el mismo. El temporizador elegido se muestra en la tabla de características del teclado.

Lo vamos a usar para que genere interrupciones periódicas. Lo activaremos cuando durante el juego sea turno del jugador que use el teclado matricial, y lo desactivaremos cuando se esté reproduciendo la secuencia de colores.

Para saber qué fuente de reloj habilitar para el temporizador, consulta la tabla "Figure 3. STM32F446xC/E block diagram" del datasheet "STM32F446xC/E"⁴. Allí podrás ver si nuestro temporizador está conectado al APB1 o al APB2, y tenemos que habilitar el reloj en el registro `RCC->APB1ENR` o `RCC->APB2ENR` respectivamente.

Es importante que no pongas los valores de los registros de configuración del temporizador a mano, sino que uses las ecuaciones que se proporcionan en el libro de fundamentos teóricos³ para calcular los valores de los registros `TIMx->PSC` y `TIMx->ARR`. En cualquier momento podríamos querer cambiar el periodo de excitación de filas y, si lo hacemos a mano, podríamos cometer errores, además de que es menos legible.

**Es muy importante que la función `_timer_scan_column_config()` se llame desde la función `port_keyboard_init()`. Si no, no se podrán generar interrupciones para excitar filas y leer columnas.

20. Codifica la función `port_keyboard_start_scan()` que se encarga de habilitar las interrupciones del temporizador y activar la cuenta (resetear el contador). En esta función se resetea el flag `flag_row_timeout`, y se excita la primera fila del teclado.

21. Codifica la función `port_keyboard_stop_scan()` que deshabilita las interrupciones del temporizador y detiene la cuenta. Del mismo modo, apaga todas las filas del teclado.

**¡Ya hemos acabado con la implementación de la parte HW `stm32f\keyboard.c` del teclado. Ahora solo queda la ISR asociada a dicho temporizador en el fichero `interr.c`. Vamos a ello.

interr.c

Abre el fichero `interr.c`. Tenemos que codificar las **ISR** para gestionar las interrupciones de cada una de las columnas y del temporizador de excitación de filas.

Es muy importante que notes que algunas están compartidas **ISR** por distintas líneas. Esto ya lo vimos en la Versión 1 con el botón. Por ejemplo la ISR `EXTI15_10_IRQHandler()` gestiona las interrupciones de las líneas 10 a la 15 de cualquier **GPIO**, y es por ello que habíamos puesto un `if` para identificar la fuente. Es ahora cuando le vas a encontrar más sentido a ese bloque condicional.

Fíjate en las **ISR** de las interrupciones de las columnas del teclado matricial en la [tabla de características del teclado](#): 2 de las columnas comparten **ISR**, y otra de ellas la comparte con el botón. Vamos a codificar dichas **ISR**:

22. Completa la ISR `EXTI15_10_IRQHandler` para gestionar la interrupción de la columna `PORT_KEYBOARD_COL_1` (la segunda) del teclado matricial. Recuerda que esta ISR también gestiona la interrupción del botón, por lo que debes mantener el bloque condicional `if` que ya estaba implementado.

Fíjate, en la API, en el *TODO* para Versión 2.

Note

Tanto esta ISR como las siguientes hacen llamadas la función privada `_check_column_interrupt()`, que se encargará de gestionar el flag de pulsación de tecla y guardar el índice de la columna que ha generado la interrupción. **Esta función auxiliar es opcional implementarla, pero hace el código más legible y evita repetir código.** Si no quieras implementarla, asegúrate de hacer en cada ISR de cada columna lo que la API indica para esta función.

Si quieres implementar `_check_column_interrupt()`, hazlo ahora. Debes colocarla antes de cualquier función que la use.

Esta ISR, cuando salta, se encarga de llamar a la función correspondiente para *settear* el estado del flag de pulsación de tecla y guardar el índice de la columna que ha generado la interrupción.

23. Codifica la ISR `EXTI9_5_IRQHandler` para gestionar las interrupciones de las columnas `PORT_KEYBOARD_COL_0` (la primera) y `PORT_KEYBOARD_COL_3` (la cuarta) del teclado matricial. Esta ISR gestiona las interrupciones de dos líneas del teclado matricial, por lo que debes mantener un bloque condicional `if` como en de la ISR anterior.

24. Codifica la ISR `EXTI4_IRQHandler` para gestionar la interrupción de la columna `PORT_KEYBOARD_COL_2` (la tercera) del teclado matricial. Como esta ISR no está compartida entre líneas del `EXTI`, no es necesario un bloque condicional.

25. Por último, codifica la ISR `TIM5_IRQHandler` como se indica en la API. Recuerda que las ISR no reciben ni devuelven nada.

Esta ISR, cuando salta, se encarga de llamar a la función correspondiente para *settear* el estado del flag de *timeout* que permitirá excitar la siguiente fila del teclado.

26. Si queda algo por documentar puede ser buen momento ahora.

Si ahora compilas, el código no debería tener ningún error. **¡Ya hemos acabado con la implementación de portado de excitación de filas para lectura de teclas en un teclado matricial!** Vamos a probarlo con el *test* unitario de la parte `PORT` para esta parte.

3.3.5 `PORT`: Test unitario del teclado matricial

Vamos a comprobar que la parte `PORT` funciona correctamente pasando los test HW del código que hemos desarrollado antes de continuar.

¡Importante! Los test que se proporcionan comprueban solo algunos aspectos esenciales, pero no son exhaustivos. Es responsabilidad del alumno comprobar que el sistema final funciona correctamente.  Ten a mano y revisa el capítulo “Test unitarios y ejemplos de integración” del libro de fundamentos teóricos ³.

Descarga el fichero de test HW del teclado `test_port_keyboard.c` de https://github.com/sdg2DieUpm/simone/tree/simone_v2_test. Ponlo en la carpeta `test/stm32f4` de tu proyecto.

27. Conecta la placa **Nucleo-STM32** al ordenador.
28. Pulsa sobre el **ícono de depuración**  y selecciona **Clean and Debug** sobre la plataforma que queramos depurar (`stm32f446re`).
29. En el desplegable que se abre, selecciona el test `test_port_keyboard`. Se compilará y se cargará en la placa.

30. Comprueba que todos los test pasan correctamente en el texto mostrado en la terminal de depuración. Si no es así, lee los mensajes de error y corrige tu código hasta que pase todas las pruebas. **Si no pasa las pruebas, no continúes programando, corrigelas.**

31. Termina la depuración pulsando (`□`) y repite el proceso hasta que pase todos los test.

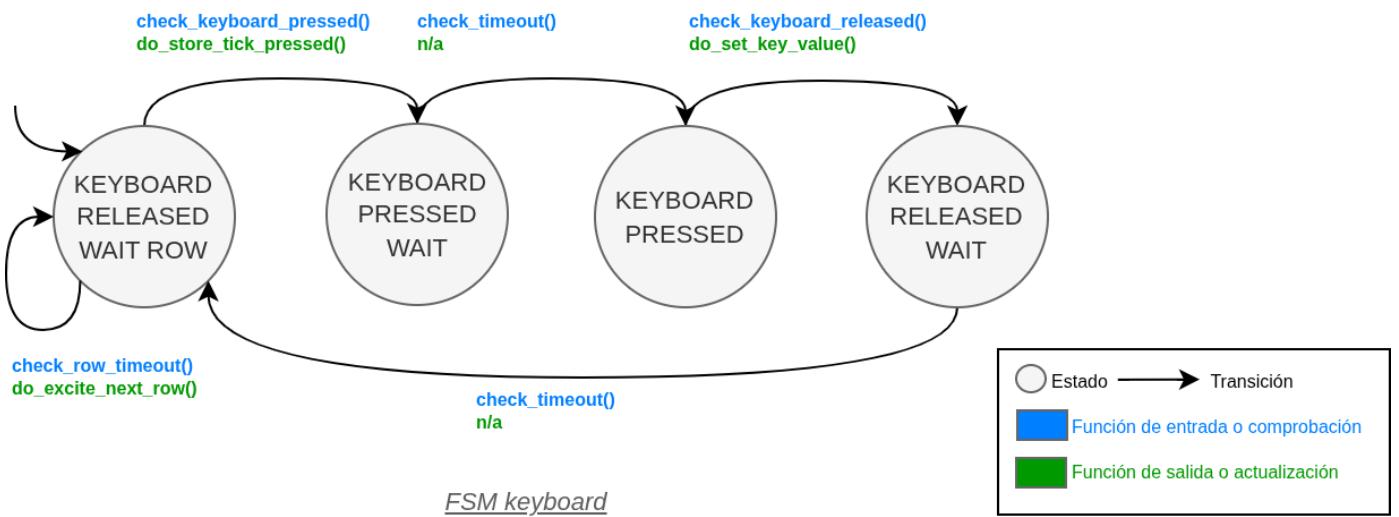
¡Ya hemos acabado con la parte `PORT` del teclado! Vamos ahora a implementar la parte `COMMON` de la librería del teclado.

3.3.6 `COMMON`: cabecera de la FSM del teclado

Consideraciones de la FSM del teclado

Antes de ponernos a programar, conviene explicar algunos aspectos importantes de la FSM del teclado matricial.

- La FSM **almacena el carácter** `char` **de la última tecla pulsada**.
- El usuario debe solicitar/ comprobar el carácter mediante la función `fsm_keyboard_get_key_value()`.
- El valor de inicio del carácter al arrancar la FSM, y el valor de reinicio, debe ser el valor de tecla inválida que haya definido el *layout* del teclado. En nuestro caso, el valor de tecla inválida es el carácter nulo `'\0'`, que está definido en la estructura `keyboard_t` del *layout* del teclado. En otro teclado podría ser otro, por lo que no hay que poner este valor *a pincho*, sino el que nos devuelva `port_keyboard_get_invalid_key_value()`.
- ¡Ojo! ☺ ¡Al inicializar la FSM, hay que inicializar el `port_keyboard_init()`, porque de lo contrario, no podremos leer qué tecla es inválida según el *layout*!
- Un valor de `invalid_key` significa que no ha habido una nueva pulsación del teclado.
- El usuario debe reiniciar el valor de tecla pulsada una vez leído, de lo contrario, este valor puede ser malinterpretado por el usuario si se realizan sucesivas comprobaciones sin haber pulsado el teclado. Es análogo a lo que hacíamos con el botón. Para reiniciar el valor se debe llamar a la función `fsm_keyboard_reset_key_value()`.
- La FSM contiene información del identificador (`ID`) del teclado que maneja. Este `ID` es único y gestionado por el usuario en el `PORT`. Ahí es donde el usuario proporciona identificadores e información HW (GPIOs a la que está conectado y tiempo de anti-rebotes) para todos los teclados de su sistema.



Máquina de estados del teclado matricial.

Nuestra librería implementa la lógica de la **FSM** mostrada en el [diagrama de la figura](#) y que llamaremos `fsm_keyboard` (en los ficheros `.c` y `.h`). Es análoga a la del botón, salvo por una **autotransición** en el primer estado. Tiene 4 estados porque

implementa también un mecanismo anti-rebotes SW. Nos centramos en la descripción de ese primer estado y su autotransición, para el resto, vaya la descripción hecha en las consideraciones de la FSM del botón:

- `KEYBOARD_RELEASED_WAIT_ROW`: es el estado inicial de la máquina de estados, y es al estado al que vuelve cuando se pulsa ¡y se suelta! una tecla. En este estado, además, se comprueba si ha pasado el *timeout* de excitación de filas (flag `flag_row_timeout`), y si es así, se excita la siguiente fila del teclado, **permaneciendo en este estado (autotransición)**. Esta excitación se hace llamando a la función `port_keyboard_get_cols()`, que devuelve el índice de la fila que se está excitando. Si, durante la excitación de una fila, se detecta una interrupción en alguna columna en `do_set_key_value()` se guardará dicho valor tras pedírselo al `PORT`.

Será el usuario en su programa principal quien deba reiniciar el valor de tecla pulsada una vez leído, llamando a la función `fsm_keyboard_reset_key_value()`, como se hacía con el botón.

La parte `COMMON` de nuestra librería trabaja con la estructura (`struct`) **pública** que se muestra en la [figura \(b\) de estructuras](#).

2. Descarga del repositorio los ficheros correspondientes a la parte **COMMON** de la librería del *teclado* correspondientes a la versión `v2`: https://github.com/sdg2DieUpm/simone/tree/simone_v2. Solo descarga lo que faltaba por implementar, es decir, los ficheros `fsm_keyboard.h` y `fsm_keyboard.c` y ponlos en las carpetas correspondientes de tu proyecto.

Ahora, vamos a completar la cabecera de la **FSM** del teclado, `fsm_keyboard.h`.

3. Incluye las librerías necesarias, si falta alguna, según indique la API.

4. Escribe el `enum` `FSM_KEYBOARD` con los nombres de los estados del diagrama de la separados por `,`. **No olvides poner un `;` al final del `enum`.**

5. **Declara** la estructura `fsm_keyboard_t` para hacerla pública como indica la [figura de la estructura de la FSM](#), y documenta cada campo con Doxygen.

Continuamos con las declaraciones de funciones públicas de la librería. **Procedamos**:

6. Escribe los prototipos de las **funciones públicas** que aparecen en la API del fichero `fsm_keyboard.h` y documenta cada función con Doxygen. Recuerda que la documentación va encima del nombre de cada función.

Ya hemos acabado con el encabezado. Quizás de errores al compilar. Vamos ahora a programar el fichero fuente `fsm_keyboard.c`.

3.3.7 `COMMON`: fuente de la FSM del teclado

Vamos a proceder con la implementación de las funciones del *teclado*. Deberás implementar todas las **funciones públicas** de las que ya has declarado el prototipo en el encabezado, y el resto de **funciones privadas** que aparecen en la API del fichero `fsm_keyboard.c`. También definiremos las variables globales y estructuras que sean necesarias. **¡Recuerda que las funciones privadas no se declaran en el `.h`!**

7. Incluye las cabeceras que indica la API.

Ahora empezamos a codificar las **funciones privadas de entrada o comprobación de la FSM** `check_`.

8. Codifica las funciones `check_row_timeout()`, `check_keyboard_pressed()`, `check_keyboard_released()`, y `check_timeout()` como se indica en la API.

Puede ser buen momento ahora para documentar las funciones con Doxygen. En este caso, como las funciones no están declaradas en el encabezado, la documentación irá en el `.c`, encima del nombre de cada función.

9. Codifica las funciones `do_excite_next_row()`, `do_store_tick_pressed()`, `do_clear_key_value()`, y `do_set_key_value()` como se indica en la API.

Documenta las funciones con Doxygen igual que antes.

10. Definimos `static fsm_trans_t fsm_trans_keyboard[] = ...` justo después de la función `do_set_key_value()` siguiendo el [diagrama de la FSM](#).

Recuerda que debe haber una fila en la tabla por cada flecha de transición entre estados de la forma: `EstadoIni, FuncCompruebaCondicion, EstadoSig, FuncAccionesSiTransicion`. No olvides añadir la fila `-1, NULL, -1, NULL`. No olvides que el `EstadoIni` de la primera transición es el estado inicial de la FSM.

11. Codifica las funciones `fsm_keyboard_start_scan()`, `fsm_keyboard_stop_scan()`, `fsm_keyboard_get_key_value()`, y `fsm_keyboard_get_is_valid_key()` como se indica en la API. Con esta última función, el usuario podrá comprobar si la última tecla pulsada es válida o no.

Codifica también la función `fsm_keyboard_reset_key_value()` con la que el usuario podrá reiniciar el valor de la tecla pulsada a `invalid_key` tras leerla.

12. Completa la función `fsm_keyboard_init()` como se indica en la API.

13. Codifica la función `fsm_keyboard_fire()` de manera análoga a [como se hizo en la FSM del botón](#).

14. Documenta el código que esté sin comentar.

Ya hemos acabado con la programación de la librería del teclado. Toda esta lógica `COMMON` **puede ser usada en cualquier sistema**. Hemos hecho una librería de un teclado que tiene un anti-rebotes y nos devuelve el valor de la última tecla pulsada. Así pues, si compilas, no deberían aparecer errores.

3.3.8 `COMMON` Test unitario de la FSM del teclado

Vamos a probar el test del código que hemos desarrollado de la librería de la máquina de estados del teclado y probar que funciona antes de continuar con la siguiente versión. **¡Importante! Recuerda que los test que se proporcionan comprueban solo algunos aspectos esenciales, pero no son exhaustivos. Es responsabilidad del alumno comprobar que el sistema final funciona correctamente.**

Descarga el fichero de test de la FSM del teclado `test_fsm_keyboard.c` de https://github.com/sdg2DieUpm/simone/tree/simone_v2_test. Ponlo en la carpeta `test/` de tu proyecto. **¡No lo metas en stm32f4!, pues no es un test específico del microcontrolador!**

15. Con la placa **Nucleo-STM32** conectada al ordenador.

16. Pulsa sobre el **ícono Clean and Debug** sobre la plataforma que queramos depurar (`stm32f446re`).

17. En el desplegable que se abre, selecciona el test `test_fsm_keyboard`. Se compilará y se cargará en la placa.

18. Ejecuta el test por completo, o pon puntos de parada si deseas ir paso a paso.

19. Se habrá impreso por la terminal del `gdb-server` el resultado de las pruebas de los tests. Debería haber pasado todos los tests. Si no, lee el mensaje de error y corrige tu código hasta que pasen todas las pruebas. **Si no pasan las pruebas, no continúes.**

20. Termina la depuración pulsando (`Q`) y repite el proceso hasta que pasen todos los test.

3.3.9 Ejemplo de uso de la Versión 2

El test de integración no hace uso de la librería `unity`, sino que es como un pequeño programa de prueba sobre las funciones que hemos implementado y tiene su propio `main`.

En los test de integración es responsabilidad del alumno comprobar que la funcionalidad es la esperada, porque aquí no hay test unitarios que nos ayuden.

Nuestra librería de teclado devuelve el valor de la última tecla pulsada. Así pues, algunas de las comprobaciones que podemos hacer son: que todas las teclas de todas las columnas aparecen impresas por pantalla, que se reinicia adecuadamente el valor tras leerlo, que funciona el anti-rebotes...

Descarga el fichero de ejemplo `example_v2.c` de https://github.com/sdg2DieUpm/simone/tree/simone_v2_test. Ponlo en la carpeta `example/` de tu proyecto.

Procedamos:

Para poder hacer el ejemplo del teclado matricial, necesitamos conectarlo como se muestra en el montaje de la [figura](#). **Fíjate que las filas son los pines de la izquierda si miramos el teclado de frente.**

21. Monta el circuito del teclado matricial como se muestra en la [figura](#).
22. Pulsa sobre el **ícono de depuración**  y selecciona **Clean and Debug** sobre la plataforma que queramos depurar ([stm32f446re](#)).
23. En el desplegable que se abre, selecciona el test [example_v2](#). Se compilará y se cargará en la placa.
24. Se parará en la primera línea del [main\(\)](#). Ejecuta el test por completo, o pon puntos de parada si deseas ir paso a paso. Este código no termina, pues es un bucle [while](#) infinito.
25. Abre la terminal del [gdb-server](#) para ver los mensajes que se van imprimiendo.
26. Pulsa una a una todas las teclas del teclado. Deberías ver que se imprime por pantalla el carácter de la pulsación. Si no es así, revisa tu código.
27. Haz distintas pruebas y asegúrate de que el comportamiento es el adecuado.

¡Hemos creado nuestra primera librería! Fíjate que es *portable* a cualquier plataforma solo con adaptar las funciones del [PORT](#).

No dejes de documentar el código. **Comprueba que la documentación del código se ha generado correctamente como se explica en la “Guía de instalación de herramientas para compilación multiplataforma en C”**⁶, o en el vídeo “[MatrixMCU] Documentación de código con Doxygen”.

Guarda una copia de su proyecto como [simone_v2](#) para tener un punto de partida para la siguiente versión, y una copia de seguridad por si algo falla.

28. Por ejemplo, un teclado de ordenador, una calculadora, un cajero automático, etc. No nos hacemos responsables de posibles daños que pueda ocasionar 😊. ↩
29. Para un mejor contacto, el circuito de la imagen son dos pistas de cobre en zig-zag que se cortocircuitan al pulsar el botón. Lo más simple sería una cruz que no se toca, pero su contacto es menos fiable. En teclados más avanzados, puede haber circuitos adicionales para mejorar la durabilidad o la respuesta táctil. ↩
30. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, and Juan José Gómez Valverde. *Fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/88460/>. ↩ ↩ ↩ ↩
31. STMicroelectronics. Stm32f446xc/e. Technical Report, STMicroelectronics, 2021. URL: <https://www.st.com/resource/en/datasheet/stm32f446re.pdf>. ↩ ↩
32. STMicroelectronics. Rm0390 reference manual. stm32f446xx advanced arm-based 32-bit mcus. Technical Report, STMicroelectronics, 2021. URL: https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. ↩
33. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, Amadeo de Gracia Herranz, Sergio Esteban Romero, and Daniel Capellán Martín. *Guía de instalación de herramientas para compilación multiplataforma en C. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/92376/>. ↩

3.4 Versión 3: *RGB light*

Ya tenemos una librería que nos permite tener botones que funcionan autónomamente mediante **FSM**. De ella hemos creado un botón que nos permitirá encender y apagar el juego Simone. Además, hemos creado una librería que nos permite crear teclados matriciales, también gestionados mediante **FSM**. De esta última, hemos creado un teclado que permitirá al jugador introducir la secuencia de colores en cada turno.

Bibliografía

1. "Fundamentos teóricos de sistemas basados en microcontrolador STM32"¹
2. *Datasheet "STM32F446xC/E"*²
3. *Reference manual "RM0390. STM32F446xx advanced Arm-based 32-bit MCUs"*³

Vídeos del canal de SDGII

- Demostración Simone
- Conceptos básicos de C (canal SDG1)
- "[MatrixMCU] Documentación de código con Doxygen"

En este capítulo vamos a crear una librería que nos permita mostrar un **LED RGB** donde cada color estará controlado por una señal **PWM**. Este elemento que llamaremos *RGB light* nos permitirá mostrar (i) los colores de la secuencia que tiene que replicar el usuario, y (ii) dar *feedback* visual al jugador cada vez que pulse una tecla.

Como ya hicimos en las versiones anteriores, (i) vamos a implementar la parte **portable** (**PORT**) dependiente del **HW** para comunicarnos con el LED **RGB**, y lo probaremos con un test unitario. (ii) Despues, vamos a crear la lógica de la **FSM** para gestionar las secuencias del juego con los distintos colores (la parte **COMMON**), y lo probaremos con un test unitario. (iii) Por último, montaremos el **HW** y probaremos el funcionamiento del LED **RGB** con un programa de ejemplo.

Cuando lea esta introducción conviene que lea y entienda el capítulo "*Circuito de reloj*" del libro ¹, prestando especial atención al ejemplo proporcionado sobre PWM.

Cada uno de los LED (**R**, **G**, y **B**) del LED estará conectado a una GPIO del **STM32F446RE**. Los tres LED están controlados por el mismo temporizador en modo **PWM**, pero cada uno con un canal de dicho temporizador. La señal **PWM** es una señal cuadrada que tiene un periodo fijo y un ciclo de trabajo variable. Si el ciclo de trabajo es del 100%, el LED estará a máxima intensidad, y si es del 0%, el LED estará apagado. Las características a destacar del sistema de la Versión 3 se muestran en la [siguiente tabla](#).

Parámetro	Valor
Pin LED rojo	PB6
Pin LED verde	PB8
Pin LED azul	PB9
Modo	Alternativo
Pull up/ down	Sin pull
Temporizador (para todos los colores)	TIM4
Canal LED rojo	(ver la tabla de Función Alternativa en el datasheet ²)
Canal LED verde	(ver la tabla de Función Alternativa en el datasheet ²)
Canal LED azul	(ver la tabla de Función Alternativa en el datasheet ²)
Modo PWM	Modo PWM 1
Prescaler	(Calcular para frecuencia de 50 Hz)
Periodo	(Calcular para frecuencia de 50 Hz)
Ciclo de trabajo LED rojo	(variable, depende del color a mostrar)
Ciclo de trabajo LED verde	(variable, depende del color a mostrar)
Ciclo de trabajo LED azul	(variable, depende del color a mostrar)

Un color se representa en el **LED RGB** mediante la combinación de los tres colores básicos: rojo, verde y azul. La combinación de los tres colores básicos en diferentes proporciones nos permite obtener una amplia gama de colores. La intensidad la controla el ciclo de trabajo (canal del temporizador), pero la frecuencia para los tres colores será la misma. La frecuencia de la señal PWM será de $\backslash(50\text{ Hz}\backslash)$, valor lo suficientemente alto para que no se perciba el parpadeo de los LED. Nuestro ojo integrará esos trenes de pulsos y lo veremos como un color determinado. **Si se cambia el ciclo de trabajo, estaremos controlando la intensidad de cada LED, y por tanto el color mostrado.**

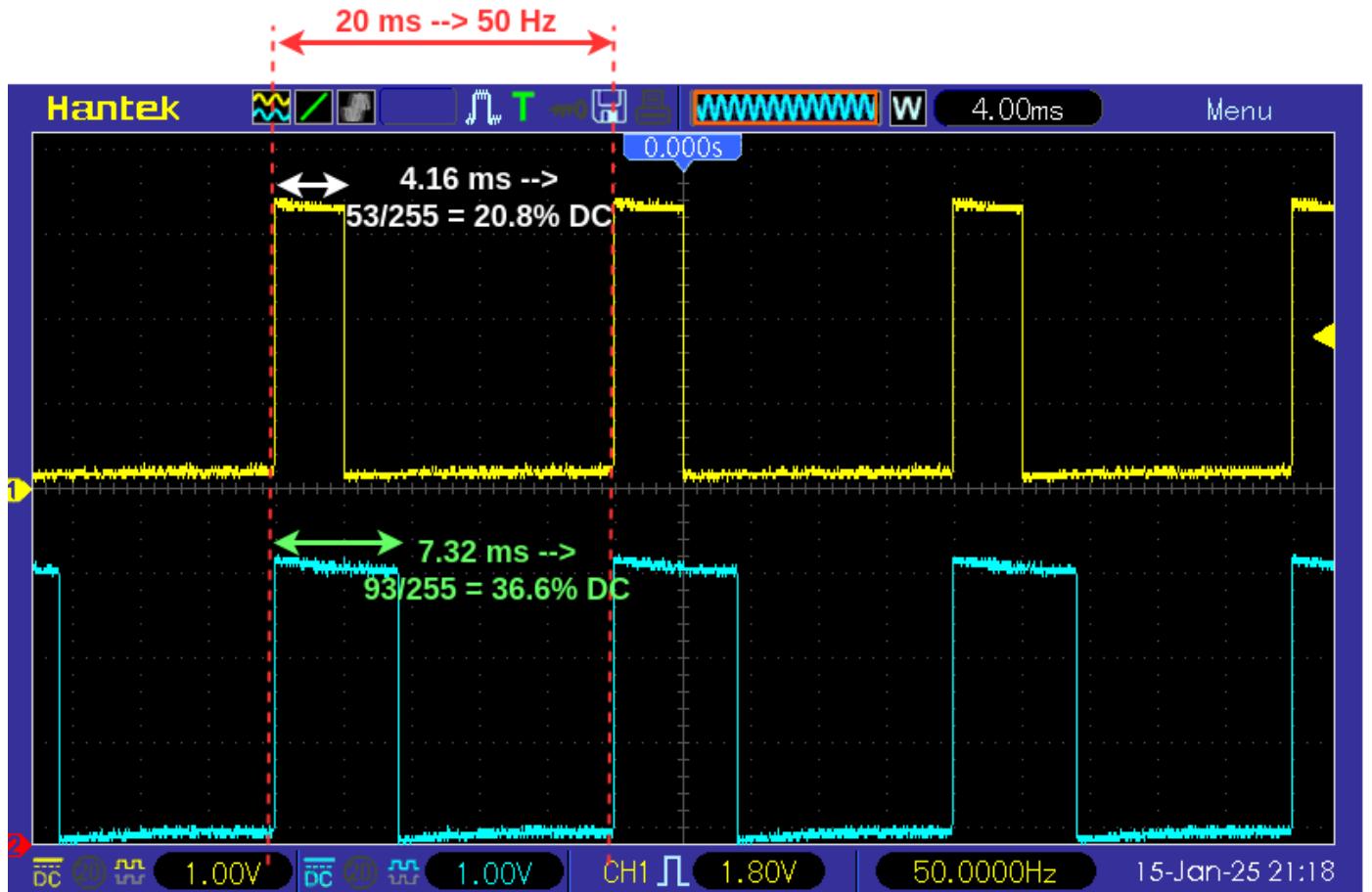
¿Cuándo cambiaremos los valores del ciclo de trabajo? Pues depende de la intensidad calculada de forma aleatoria según el nivel de dificultad del juego. Cuanto más difícil sea el nivel, más parecidos serán los colores (menor intensidad), y por tanto más difícil será para el usuario distinguirlos.

La [tabla de colores](#) muestra los colores que se usarán en el juego y los valores de ciclo de trabajo asociados a cada color **para máxima intensidad en tanto por ciento**. Los colores están definidos en la librería que se proporciona [`rgb_colors.c`](#) de la parte [`COMMON`](#), y que mencionaremos más adelante.

Color	LED rojo	LED verde	LED azul
Rojo	\(100~\%)	\(0~\%)	\(0~\%)
Verde	\(0~\%)	\(100~\%)	\(0~\%)
Azul	\(0~\%)	\(0~\%)	\(100~\%)
Amarillo	\(37~\%)	\(37~\%)	\(0~\%)
Turquesa	\(10~\%)	\(35~\%)	\(32~\%)
Blanco	\(100~\%)	\(100~\%)	\(100~\%)
Apagado	\(0~\%)	\(0~\%)	\(0~\%)

En la [Versión 5](#) puedes implementar otros colores. Tienes una lista de colores en la www.downtownuplighting.com—aunque no todos se pueden mostrar en un LED **RGB**.

La [figura](#) muestra un ejemplo de señal PWM para un color amarillento (que no es el amarillo de [la tabla](#)). Se muestran en el osciloscopio dos de los LED del **RGB**. En la parte superior se muestra el canal de osciloscopio para el LED rojo, y en la parte inferior el canal de osciloscopio para el LED verde. Fíjate que el periodo de la señal PWM es el mismo para los dos LED (\(20 ms\)), pero el ciclo de trabajo es distinto: \((20.8\%)\) para el LED rojo y \((36.6\%)\) para el LED verde.



Ejemplo de señales PWM para un color amarillento.

Igual que hemos hecho hasta ahora, **estamos desarrollando una librería**. Así, cada vez que se quiera añadir un LED **RGB** le asociaremos una **FSM**. Las particularidades de dónde está conectado cada nuevo LED **RGB**, sus características físicas, *etc.*, son cosas específicas del HW, por lo que estarán en [PORT](#).

rgb_color_t	
uint8_t	r
uint8_t	g
uint8_t	b

stm32f4_rgb_light_hw_t	
GPIO_TypeDef	*p_port_red;
uint8_t	pin_red;
GPIO_TypeDef	*p_port_green;
uint8_t	pin_green;
GPIO_TypeDef	*p_port_blue;
uint8_t	pin_blue

fsm_rgb_light_t	
fsm_t	f;
rgb_color_t	color;
uint8_t	intensity_perc
bool	new_color;
bool	status;
bool	idle;
uint32_t	rgb_light_id;

(a) Estructura de un color RGB, (b) Estructura del HW del LED RGB en PORT, (c) Estructura de la FSM del *RGB light* en COMMON.

Las figuras de estructuras de un **color RGB**, **del HW y del SW** muestran las estructuras que vamos a necesitar para el LED **RGB**.

Un color **RGB** se define por la estructura de la **figura (a)** con tres valores `uint8_t` que indican la intensidad de cada componente en un rango de [0, `COLOR_RGB_MAX_VALUE`]. Donde la etiqueta `COLOR_RGB_MAX_VALUE` se ha definido con un valor de `255`, por lo que este será el número de niveles de intensidad que podremos representar para cada componente.

La estructura del HW del LED **RGB** (en `PORT`) se muestra en la **figura (b)**. La estructura de la **FSM** (en `COMMON`) se muestra en la **figura (c)**.

Preparemos el proyecto para poder añadir el LED **RGB**:

1. Descarga del repositorio de la asignatura los ficheros correspondientes **a la parte PORT** de la librería del *RGB light* correspondientes a la versión `v3`: https://github.com/sdg2DieUpm/simone/tree/simone_v3. Solo descarga por ahora: `port_rgb_light.h`, `stm32f4_rgb_light.h`, y `stm32f4_rgb_light.c` y colócalos en las carpetas correspondientes. **De la parte COMMON descarga solo** `rgb_color.h` y `rgb_color.c`, que incluyen la definición de los colores que usaremos en el juego.
2. Coloca cada uno donde corresponde: `PORT` o `COMMON`, en `include`, o `src`, como se explicó en los capítulos anteriores.
3. Verás que no compila, y es que solo se proporciona un esqueleto del código.

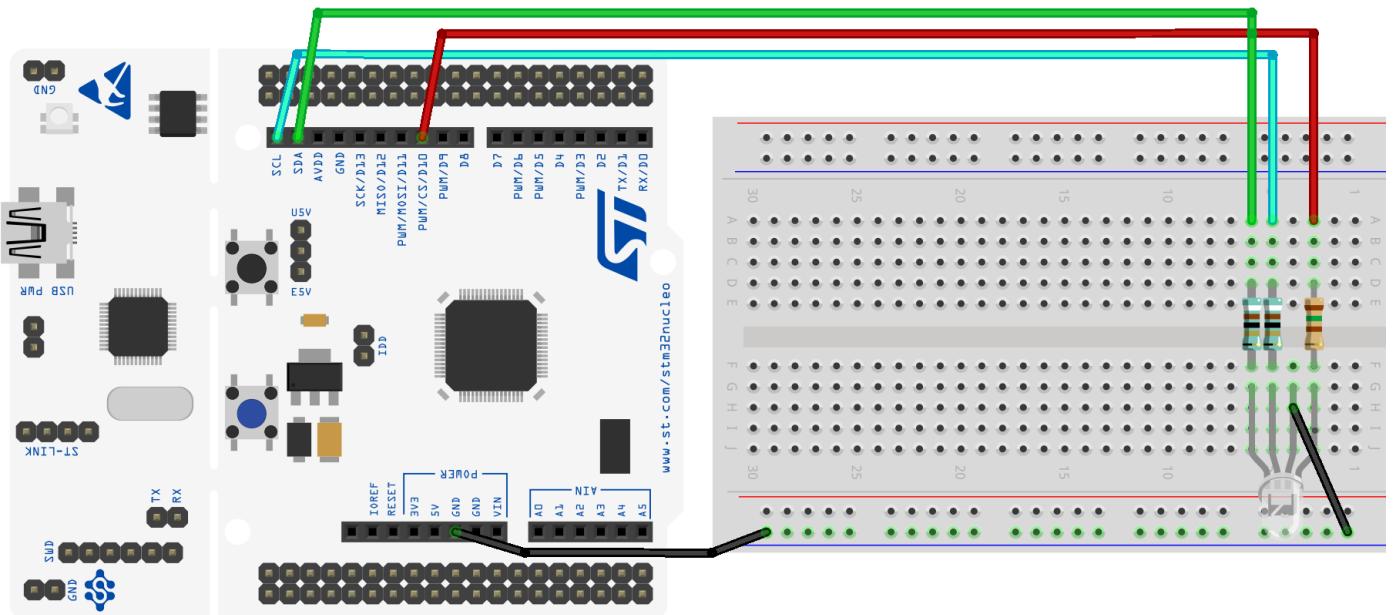
3.4.1 Colores RGB predefinidos

Antes de ponernos a programar, conviene explicar qué son los ficheros `rgb_colors.h` y `rg_colors.c`. Estos ficheros se han de colocar en la carpeta `common/include` y `common/src` respectivamente. Estos ficheros contienen los colores que se usarán en el juego. Cada color está representado por una estructura de tipo `rgb_color_t` que contiene tres campos: `r`, `g`, y `b`, que representan la intensidad de cada componente del color en un rango de [0, `COLOR_RGB_MAX_VALUE`]. Estos colores predefinidos son los que se muestran en la **tabla**.

Estos ficheros permiten definir múltiples colores de manera sencilla y reutilizable. Si deseas añadir más colores, puedes hacerlo en la **Versión 5** definiendo nuevas variables de tipo `rgb_color_t` en el fichero `rgb_colors.c` y declarándolas en el fichero `rgb_colors.h`. También podrías cambiar el rango de intensidad modificando el valor de `COLOR_RGB_MAX_VALUE` para tener mayor o menor resolución en la representación de los colores.

3.4.2 `PORT`: cabeceras de la librería del RGB light

Vamos a implementar el *contrato con el usuario* de la parte dependiente del HW de librería del *RGB light*. Esta es la interfaz que vamos a proporcionar al usuario para que pueda usar la librería y añadir LEDs **RGB** como necesite. Vamos a *portar* las funciones necesarias para usar la librería, cómo no, para la placa **Nucleo-STM32F446RE**. Comenzaremos de nuevo por la cabecera y luego los códigos fuente. El montaje de nuestro LED **RGB** con la **Nucleo-STM32F446RE** se muestra en la **figura**.



Montaje del LED RGB de cátodo común con la Nucleo-STM32F446RE.

En el mercado existen numerosos LEDs **RGB**. En este proyecto utilizaremos un LED de cátodo común. En este tipo de LEDs, el ánodo de cada LED está conectado a un pin del microcontrolador a través de una resistencia, y los cátodos de los tres LED están internamente conectados a un pin común que irá a tierra. Para adquirir uno, ve a [la BOM](#).

Cabecera port_rgb_light.h

Esta cabecera depende del HW pero no de las particularidades del microcontrolador **STM32F446RE**. Vamos a seguir los siguientes pasos:

1. Incluye todas las cabeceras necesarias según indica la API.
2. Define (`#define`) el `PORT_RGB_LIGHT_ID`, que es un valor numérico natural que será el identificador del *RGB light* trasero. Si es el único *RGB light* del sistema, le asignaremos el 0.
3. Escribe los prototipos de las funciones públicas que aparecen en la API del fichero `port_rgb_light.h`.
4. Puede ser buen momento ahora para documentar con Doxygen.

Ya hemos acabado con el encabezado que interactúa con el HW del *RGB light* y que no depende del microcontrolador. Todavía dará errores al compilar. Vamos ahora a programar la cabecera que sí depende del microcontrolador `stm32f4_rgb_light.h`.

Cabecera stm32f4_rgb_light.h

Esta cabecera define los pines a los que está conectado el LED **RGB** de cada *RGB light* asociado.

1. Incluye todas las cabeceras necesarias según indica la API.
2. Define (`#define`) los valores de las GPIO de los pines de los LED rojo, verde y azul.
3. Declara la estructura `stm32f4_rgb_light_hw_t`, que contiene en sus campos los puertos y pines de cada GPIO. Presta atención a la API.
4. Declara el array de estructuras de tipo `stm32f4_rgb_light_hw_t` como se hizo en versiones anteriores. Este array contendrá las características de todos los LED RGB que tengamos en el sistema.
5. Documenta todo con Doxygen.

Ya hemos acabado con el encabezado (*header*) que interactúa con el HW del *RGB light*. Todavía dará errores al compilar. Vamos ahora a implementar todas las funciones prototipadas en `port_rgb_light.h`.

3.4.3 PORT: fuente de la librería del RGB light

Vamos a *portar* las funciones necesarias para usar la librería del *RGB light* y comprobar que la parte HW está bien programada. Vamos a programar los ficheros fuente de la parte `PORT`, que **todos estarán en el fichero** `stm32f4_rgb_light.c`.

Fuentes `stm32f4_rgb_light.c`

Este fichero es mucho menos extenso que el de la versión anterior. La mayor complejidad está en la configuración de los canales del temporizador, pero si has leído el capítulo del libro relativo a PWM y lo has entendido, no deberías tener problema ¹. Vamos a ello.

1. Incluye las librerías necesarias, si falta alguna, según indique la API.
2. Igual que hicimos anteriormente, define la **variable global privada** `stm32f4_rgb_light_hw_t rgb_lights_arr[]` con la configuración física de nuestro LED RGB. Se trata del array de estructuras de tipo `stm32f4_rgb_light_hw_t`, que representa al HW de cada *RGB light* que tengamos en nuestro juego.
Asigna los valores HW del LED **RGB** del LED `PORT_RGB_LIGHT_ID` utilizando los `#define` de `stm32f4_rgb_light.h`.
3. Codifica la función `_stm32f4_rgb_light_get()` de manera análoga a como se ha hecho en las versiones anteriores.

Ahora vamos a codificar las funciones más importantes de la parte `PORT` del *RGB light*, y son las que configuran el temporizador asociado al LED **RGB**.

1. Codifica la función `_timer_pwm_config()` como indica la API. Esta función configura el temporizador que controla los ciclos de trabajo de los LED rojo, verde y azul. Para ello, apóyate en el ejemplo "timer para PWM" del libro de Fundamentos ¹.

Esta función configura un temporizador para que genere una señal **PWM** con una frecuencia fija y un ciclo de trabajo variable. El temporizador elegido y la frecuencia se muestra en la [tabla de HW para la versión 3](#).

Esta función recibe el identificador del LED **RGB**. Cada *RGB light* tendrá su propio temporizador, pero esta función será llamada para configurar todos ellos. Asegúrate de que el código se ejecuta dentro de un bloque condicional que compruebe el identificador del *RGB light*.

Para saber qué fuente de reloj habilitar para el temporizador, consulta la tabla "Figure 3. STM32F446xC/E block diagram" del [datasheet](#) ². Allí podrás ver si nuestro temporizador está conectado al **APB1** o al **APB2**, y tenemos que habilitar el reloj en el registro `RCC->APB1ENR` o `RCC->APB2ENR` respectivamente.

Puedes poner los valores de los registros `TIMx->PSC` y `TIMx->ARR` a mano, o usando las ecuaciones. Puedes crear un `#define` para la frecuencia de la señal PWM, si te es más cómodo. Si lo haces a mano, asegúrate de que los valores son correctos.

Asegúrate de que los registro *Capture Compare Enable Register* (`CCER`) y *Capture Compare Mode Register* (`CCMRx`) están configurados correctamente. En el *CCER* se habilitan los canales de salida y en el *CCMRx* se configura el modo PWM. La *x* es el registro 1 o 2, dependiendo del canal que estés configurando.

Es muy importante que la función `_timer_pwm_config()` se llame desde la función `port_rgb_light_init()`. Si no, no se podrán generar las señales PWM.

Vamos a continuar con las funciones públicas de la parte `PORT` del *RGB light*.

1. Completa la función `port_rgb_light_init()` como se indica en la API.

Configura las GPIO y el modo alternativo de los tres LED **RGB**. Para ello, consulta la tabla de *Función Alternativa* del [datasheet](#) ².

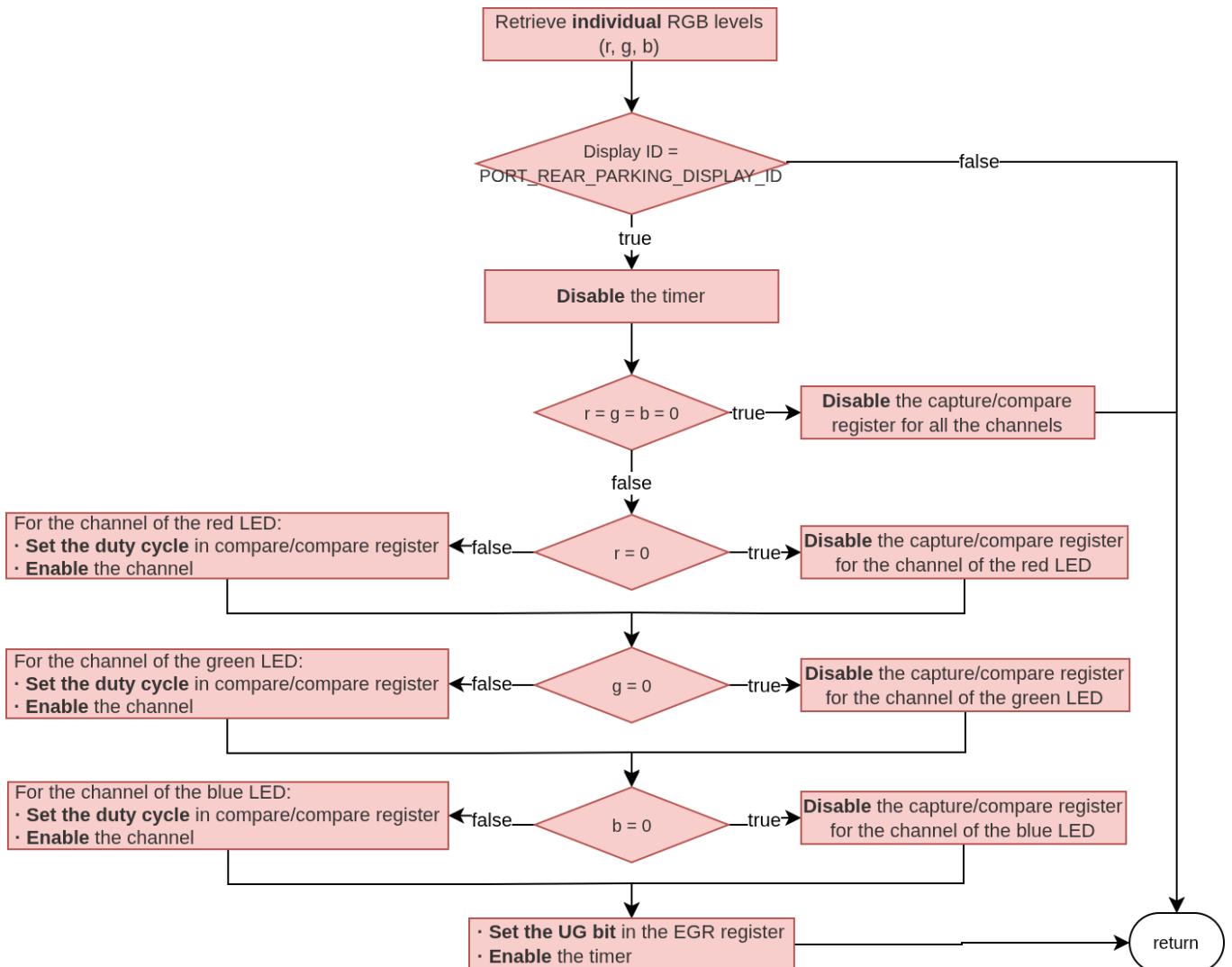
Asegúrate de que la función `_timer_pwm_config()` se llama desde esta función y que el LED **RGB** comienza apagado.

2. Codifica la función `port_rgb_light_set_rgb` siguiendo la API.

Esta función se encarga de configurar el ciclo de trabajo de los LED rojo, verde y azul. Lo hace con la proporción de los valores recibidos en la estructura `rgb_color_t` sobre el máximo valor definido en `COLOR_RGB_MAX_VALUE`. Si el valor recibido es 0, el LED estará apagado, deshabilitando el canal correspondiente.

Fíjate que la función recibe el identificador del LED **RGB**. Asegúrate de que el código se ejecuta dentro de un bloque condicional que compruebe el identificador del *RGB light*.

Sigue el [siguiente flujograma](#) para implementar la función `port_rgb_light_set_rgb()`:



3. Si queda algo por documentar puede ser buen momento ahora.

Ahora, compila, el código no debería tener ningún error. **¡Ya hemos acabado con la implementación de `portado` del LED **RGB**!** Vamos a probarlo con el *test* unitario de la parte `PORT`.

3.4.4 `PORT`: Test unitario del RGB light

Vamos a comprobar que la parte `PORT` funciona correctamente pasando los test HW del código que hemos desarrollado de la librería del LED **RGB** antes de continuar con la FSM.

¡Importante! Los test que se proporcionan comprueban solo algunos aspectos esenciales, pero no son exhaustivos. Es responsabilidad del alumno comprobar que el sistema final funciona correctamente.  Ten a mano y revisa el capítulo "Test unitarios y ejemplos de integración" del libro de fundamentos teóricos¹.

Descarga el fichero de test HW del *RGB light* `test_port_rgb_light.c` de https://github.com/sdg2DieUpm/simone/tree/simone_v3_test. Ponlo en la carpeta `test/stm32f4` de tu proyecto.

1. Conecta la placa **Nucleo-STM32** al ordenador.
2. Pulsa sobre el **ícono de depuración**  y selecciona  **Clean and Debug** sobre la plataforma que queramos depurar (`stm32f446re`).
3. En el desplegable que se abre, selecciona el test `test_port_rgb_light`. Se compilará y se cargará en la placa.
4. Comprueba que todos los test pasan correctamente en el texto mostrado en la terminal de depuración. Si no es así, lee los mensajes de error y corrige tu código hasta que pase todas las pruebas. **Si no pasa las pruebas, no continúes con el siguiente test.**
5. Termina la depuración pulsando () y repite el proceso hasta que pase todos los test.

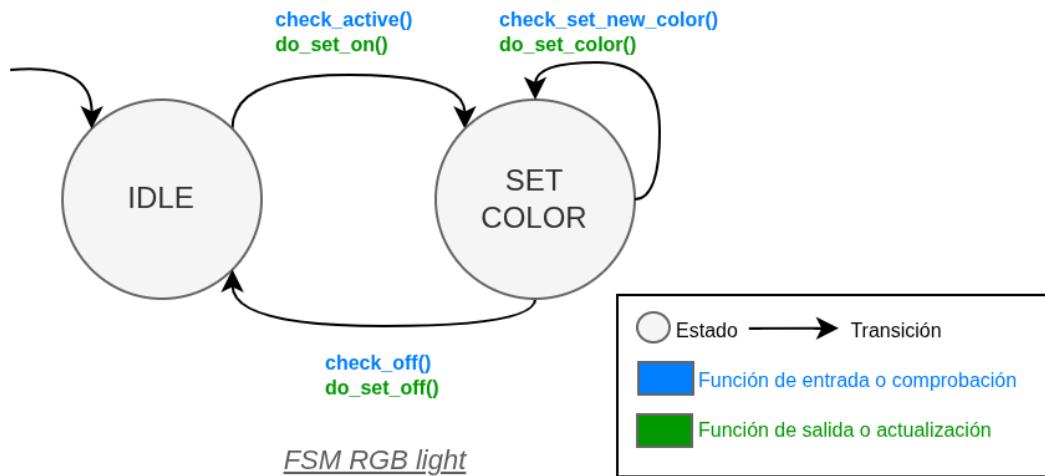
¡Ya hemos acabado con la parte `PORT` del *RGB light*! Vamos ahora a implementar la parte `COMMON`.

3.4.5 `COMMON`: cabecera de la FSM del *RGB light*

Consideraciones de la FSM del *RGB light*

Antes de empezar vamos a partir de una serie de consideraciones.

- La FSM **almacena el último color que se le ha indicado que tiene que representar y su intensidad**. La intensidad se almacena en porcentaje como un `uint8_t` pero ha de estar limitada a `MAX_LEVEL_INTENSITY` igual a \|(100\|) cuando se opere con ella. El color se almacena en una estructura de tipo `rgb_color_t`.
- La FSM contiene un campo de estado (`status`) que indica si el *RGB light* está activo. Si el *RGB light* está desactivado, no se mostrará ningún color. Esto puede ser útil si, por ejemplo, tenemos más de un *RGB light* en el sistema y queremos desactivar alguno de ellos.
- La máquina de estados superior, la del sistema Simone, será la que indique si el *RGB light* está activo o no. La FSM del *RGB light* solo se encargará de representar los colores si está activo.
- La FSM contiene otro *flag* que indica si el *RGB light* está en modo ocioso (`idle`). En este estado, el *RGB light* ya ha puesto un color, no se ha cambiado éste ni la intensidad, y podemos entrar en un modo de bajo consumo *Sleep Mode Versión 4*. En *Sleep Mode*, el núcleo del procesador se detiene, pero los periféricos y el sistema de reloj continúan funcionando, por eso el LED permanecerá encendido en el color que se haya quedado.
- El *flag* `new_color` lo activa la máquina de estados de Simone cuando se ha cambiado el color a representar. La FSM del *RGB light* comprobará este *flag* para saber si tiene que cambiar el color del LED.
- El valor de inicio de intensidad al arrancar debe ser el máximo permitido. **El intensidad se da en porcentaje**.
- La FSM contiene información del identificador (`ID`) del *RGB light*. Este `ID` es único y gestionado por el usuario en el `PORT`. Ahí es donde el usuario proporciona identificadores e información HW (GPIO a la que está conectado) para todos los LED **RGB** del sistema.



Máquina de estados del RGB light.

Nuestra librería implementa la lógica de la **FSM** mostrada en la [figura](#) y que llamaremos `fsm_rgb_light` (en los ficheros `.c` y `.h`). Tiene 2 estados en los que:

- `IDLE_RGB`: estado inicial de la FSM en el que espera a ser activada. Si se le activa se cambia al estado `SET_COLOR`. También se vuelve a este estado cuando el sistema no está activo (`status` es `false`).
- `SET_COLOR`: en este estado la FSM espera a que se le haya pasado un nuevo color base y/o una intensidad nueva. Tras poner el color, se activa el *flag* `idle` y se queda en este estado. El nuevo color se corrige modificando la intensidad del color base recibido según el porcentaje de intensidad indicado; esto se hace con la función privada `_correct_rgb_light_levels()` que deberá implementar. Si se le desactiva el sistema, vuelve al estado `IDLE_RGB`.

La parte `COMMON` de nuestra librería trabaja con la estructura (`struct`) **pública** que se muestra en la [figura \(c\)](#) `fsm_rgb_light_t`. El tipo de esta estructura está declarada en el fichero `fsm_rgb_light.h`.

3. Lo primero, descarga del repositorio de la asignatura los ficheros correspondientes a la **parte COMMON** de la librería del *RGB light* correspondientes a la versión `v3`: https://github.com/sdg2DieUpm/simone/tree/simone_v3. Solo descarga lo que faltaba por implementar, es decir, los ficheros `fsm_rgb_light.h` y `fsm_rgb_light.c`. Ponlos en las carpetas correspondientes de tu proyecto.

Ahora, vamos a completar la cabecera de la **FSM** del *RGB light*, `fsm_rgb_light.h`.

4. Incluye las librerías necesarias, si falta alguna, según indique la API.
5. Ahora vamos a definir el enumerado con los nombres de los estados de la FSM. Escribe el `enum` `FSM_RGB_LIGHT_SYSTEM` con los nombres de los estados del diagrama de la .
6. Defina los `#define` que se indican en la API para el máximo nivel de intensidad porcentual que se pueden manejar en el *RGB light*.
7. Es buena práctica ir documentando el código a la vez que se programa.
8. Seguidamente **declararemos** la estructura `fsm_rgb_light_t` y definimos sus campos. **Es muy importante que la máquina de estados del *RGB light* `fsm_t` sea el primer campo**.

Continuamos con las declaraciones de funciones públicas de la librería. **Procedamos**:

9. Escribe los prototipos de las **funciones públicas** que aparecen en la API del fichero `fsm_rgb_light.h`. Añada la función `fsm_rgb_light_check_activity()` que, aunque la usaremos en la siguiente versión, también será necesaria para el test de la FSM.
10. Puede ser buen momento ahora para documentar las funciones con Doxygen.

Vamos ahora a programar el fichero fuente `fsm_rgb_light.c`.

3.4.6 **COMMON**: fuente de la FSM del RGB light

Vamos a proceder con la implementación de las funciones del *RGB light*. Deberás implementar todas las **funciones públicas** de las que ya has declarado el prototipo en el encabezado, y la **función privada** `_correct_rgb_light_levels()` que aparece en la API del fichero `fsm_rgb_light.c`. ¡Recuerda que las funciones privadas no se declaran en el `.h`!

11. Lo primero que debe aparecer es la inclusión de cabeceras como indica la API.

Ahora empezamos a codificar las **funciones privadas de la FSM**. Bajo la línea de `/* Private functions */` vamos a codificar la función `_correct_rgb_light_levels()`.

12. Codifica la función `_correct_rgb_light_levels()` que se encarga de calcular los niveles de ciclo de trabajo de cada LED reo, verde y azul. Esta función simplemente multiplica por un valor entre [0, 1] cada uno de los componentes RGB. Esta función **Ten en cuenta que el valor que almacenan no es el ciclo de trabajo entre 0 y 100, sino que quedará entre 0 y 255. ¡El ciclo de trabajo se calcula en `port_rgb_light_set_rgb()`!**

13. Documenta la función con Doxygen.

Continuamos con las funciones de entrada o comprobación `check_` **de la FSM** bajo la línea de `/* State machine input or transition functions */`.

14. Codifica las funciones `check_active()`, `check_set_new_color()` y, `check_off()` como se indica en la API.

15. Documenta las funciones con Doxygen, encima del nombre de cada función.

Seguiremos con las funciones de salida o actualización de la FSM `do_`.

16. Codifica las funciones `do_set_on()`, `do_set_color()` y, `do_set_off()` como se indica en la API.

17. Documenta las funciones con Doxygen. En este caso, igual que antes, la documentación irá en el `.c`, encima del nombre de cada función.

Todavía no hemos acabado con el desarrollo, pero puedes compilar para ir depurando errores.

18. Declara la tabla de transiciones de la FSM `fsm_trans_rgb_light`. Esto eliminará muchos errores de compilación.

19. Completa la función de inicialización de la FSM `fsm_rgb_light_init()` como se indica en la API. En esta función se inicializan los campos de la estructura de la FSM, y se llama a la función `fsm_init()` para inicializar la máquina de estados. También se llama a la función `port_rgb_light_init()` para inicializar el HW del LED **RGB**.

20. Codifica las funciones `fsm_rgb_light_fire()` y `fsm_rgb_light_destroy()` igual que hiciste en las versiones anteriores. Estas funciones lanzan la FSM y liberan la memoria respectivamente.

21. Codifica el resto de las funciones `fsm_rgb_light_xxx()` públicas que quedan declaradas en el `.h` como se indica en la API.

22. Documenta el código que esté sin comentar todavía.

Ya hemos acabado con la programación de la librería del *RGB light*. Ahora, si compilas, no deberán aparecer errores.

3.4.7 **COMMON**: Test unitario de la FSM del RGB light

Vamos a hacer el test del código que hemos desarrollado de la librería de la máquina de estados del *RGB light* y probar que funciona antes de continuar con la siguiente versión.

¡Importante! Recuerda que los test que se proporcionan comprueban solo algunos aspectos esenciales, pero no son exhaustivos. Es responsabilidad del alumno comprobar que el sistema final funciona correctamente.

Descarga el fichero de test de la FSM del *RGB light* `test_fsm_rgb_light.c` de https://github.com/sdg2DieUpm/simone/tree/simone_v3_test. Ponlo en la carpeta `test/` de tu proyecto. ¡No lo metas en `stm32f4`!, pues no es un test específico del microcontrolador!

23. Con la placa **Nucleo-STM32** conectada al ordenador.

24. Pulsa sobre el **ícono Clean and Debug** sobre la plataforma que queramos depurar (`stm32f446re`).

25. En el desplegable que se abre, selecciona el test `test_fsm_rgb_light`. Se compilará y se cargará en la placa.
26. Ejecuta el test por completo, o pon puntos de parada si deseas ir paso a paso.
27. Comprueba los mensajes del `gdb-server` para ver el resultado de las pruebas de los tests. Debería haber pasado todos los tests. Si no, lee el mensaje de error y corrige tu código hasta que pasen todas las pruebas. **Si no pasan las pruebas, no continúes.**
28. Termina la depuración pulsando () y repite el proceso hasta que pasen todos los test.

3.4.8 Ejemplo de uso de la Versión 3

En este test de integración del *RGB light* es responsabilidad del alumno comprobar que la funcionalidad es la esperada. El ejemplo que se da no contempla todas las situaciones.

Descarga el fichero de ejemplo `example_v3.c` de https://github.com/sdg2DieUpm/simone/tree/simone_v3_test. Ponlo en la carpeta `example/` de tu proyecto.

Procedamos:

Para poder hacer el ejemplo del *RGB light*, necesitamos montar el LED **RGB**. En el [esquema de la figura](#) se muestra un ejemplo de montaje.

29. Monte el HW como se muestra en la .
30. Con la placa **Nucleo-STM32** conectada al ordenador.
31. Pulsa sobre el **ícono de depuración** y **selecciona** **Clean and Debug** sobre la plataforma que queramos depurar (`stm32f446re`).
32. En el desplegable que se abre, selecciona el test `example_v3`. Se compilará y se cargará en la placa.
33. Se parará en la primera línea del `main()`. Ejecuta el test por completo, o pon puntos de parada si deseas ir paso a paso. Este código no termina, pues es un bucle `while` infinito.
34. Abre la terminal del `gdb-server` para ver los mensajes que se van imprimiendo.
35. Comprueba que los colores del LED **RGB** cambian en función de la intensidad que se le indica en los mensajes de la terminal.
36. Haz distintas pruebas y asegúrate de que el comportamiento es el adecuado.

¡Hemos creado nuestra tercera librería! Fíjate que es *portable* a cualquier plataforma solo con adaptar las funciones del `PORT`.

No dejes de documentar el código. **Comprueba que la documentación del código se ha generado correctamente como se explica en la "Guía de instalación de herramientas para compilación multiplataforma en C"**⁴., o en el vídeo "[MatrixMCU] Documentación de código con Doxygen".

Guarda una copia de su proyecto como `simone_v3` para tener un punto de partida para la siguiente versión, y una copia de seguridad por si algo falla.

-
37. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, and Juan José Gómez Valverde. *Fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/88460/>. ↪ ↫ ↬ ↭ ↮
 38. STMicroelectronics. Stm32f446xc/e. Technical Report, STMicroelectronics, 2021. URL: <https://www.st.com/resource/en/datasheet/stm32f446re.pdf>. ↪ ↫ ↬ ↭ ↮
 39. STMicroelectronics. Rm0390 reference manual. stm32f446xx advanced arm-based 32-bit mcus. Technical Report, STMicroelectronics, 2021. URL: https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. ↪
 40. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, Amadeo de Gracia Herranz, Sergio Esteban Romero, and Daniel Capellán Martín. *Guía de instalación de herramientas para compilación multiplataforma en C. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/92376/>. ↪

3.5 Versión 4: integración final y modos de bajo consumo

3.5.1 Modos de bajo consumo

Ya tenemos todos los elementos del sistema para que sea funcional. Quizás quisiéramos —si se hace el diseño correspondiente— diseñar una **PCB** para desplegarlo en algún sitio. Si hiciésemos esto, muy seguramente alimentaríamos el dispositivo con una batería. Si midiésemos el consumo con un amperímetro, podríamos calcular la autonomía de nuestro sistema. Te habrás fijado que en los dispositivos comerciales como relojes inteligentes, mandos de TV, dispositivos IoT...la autonomía puede superar de largo varios meses con un uso normal del mismo. Para conseguir esto contamos con los **modos de bajo consumo**. Buena parte de los microcontroladores que hoy en día se precien cuentan con distintos modos de bajo consumo. **Lea la sección “Modos de bajo consumo” del libro de Fundamentos Teóricos¹**.

En esta versión, antes de hacer la integración final, vamos a implementar unas pocas funciones para gestionar el **modo sleep** de bajo consumo en nuestro sistema. Esto se destaca en 2 estados de la FSM de *Simone* que veremos más adelante. Estos estados comprueban si alguna de las FSM de los elementos está, o no, activa, y en caso de que todas estén inactivas, *se va a dormir*. El sistema *se despertará* ante alguna interrupción de un *timer* o interrupción externa (pulsación de botón o teclado).

Antes de empezar a implementar las funciones de bajo consumo vamos a partir de una serie de consideraciones de la FSM. En la siguiente sección se detallan mucho más los estados, pero por ahora, nos fijamos en lo relativo al bajo consumo:

- En bajo consumo desactivaremos el **SysTick** para que no despierte al sistema cada \((1 \text{ ms})\). Así pues, el contador del sistema no aumenta mientras se está *dormido*.
- Las ISR que generan interrupciones externas —*botón* y *teclado matricial*— son las encargadas de reactivar el **SysTick**.
- La FSM del *botón* está inactiva en el estado `BUTTON_RELEASED`.
- La FSM del *teclado matricial* está inactiva en el estado `KEYBOARD_RELEASED_WAIT_ROW`.
- La FSM del *RGB light* está activa si el *status* indica que está *funcionando*, y no está ocioso (*idle*).
- Las autotransiciones de los estados de la FSM *Simone* (`SLEEP_WHILE_IDLE` y `SLEEP_WHILE_PLAYBACK`) están pensadas para cuando esté trabajando en depuración. El depurador genera interrupciones en la ejecución del código que despiertan a nuestro sistema. Como no se trata de interrupciones de nuestros elementos, no pasaremos a los estados, pero debemos dormirnos mientras no se detenga el depurador de nuevo en otro *breakpoint*. Este es el cometido de dichas autotransiciones.

Procedamos. Como siempre, tenga abierta la página web de la **API** <https://sdg2dieupm.github.io/simone/>, ahí están todos los detalles de implementación. Ahora vamos a tocar varios ficheros pero no crearemos ninguno nuevo.

Vamos a añadir las funciones de comprobación específicas de cada máquina de estados.

7. En `fsm_button.c`: Añade la función `fsm_button_check_activity()` y su prototipo y documentación del código en `fsm_button.h`.
8. En `fsm_keyboard.c` añade la función `fsm_keyboard_check_activity()` y su prototipo y documentación del código en `fsm_keyboard.h`.
9. En `fsm_rgb_light.c` añade la función `fsm_rgb_light_check_activity()` y su prototipo y documentación en `fsm_rgb_light.h`.

Para terminar, vamos a añadir las funciones HW específicas de manejo del modo *stop* y *sleep* en nuestro **STM32F446RE**. Primero añadiremos algunas funciones generales del sistema en `stm32f4_system.c`; luego las modificaciones necesarias para restablecer el reloj de sistema **SysTick** tras una interrupción del *botón* o del *teclado matricial*, o de temporizador.

En `stm32f4_system.c`:

10. Copia el código de `port_system_power_stop()` y `port_system_power_sleep()` de la API. Por tener un orden, puedes hacerlo en una parte dedicada a *POWER RELATED FUNCTIONS*.

11. Copia el código de `(port_system_systick_suspend())` de la API. Por tener un orden, puedes hacerlo en la parte dedicada a *TIMER RELATED FUNCTIONS*.
12. Copia el código de `(port_system_systick_resume())` de la API. Por tener un orden, puedes hacerlo también en la parte dedicada a *TIMER RELATED FUNCTIONS*.
13. Implementa la función `(port_system_sleep())` como indica la API. Por tener un orden, puedes hacerlo junto con las anteriores en la parte dedicada a *POWER RELATED FUNCTIONS*.

En `(port_system.h)`:

14. Añade los prototipos de las funciones anteriores y su documentación.

En el fichero en el que se encuentran nuestras ISR, `(interr.c)`, añade al principio de todas las ISR de todos los GPIO la llamada a `(port_system_systick_resume())` para reactivar el contador del sistema **SysTick** inmediatamente tras la interrupción de pulsación de cualquier tecla o del botón de usuario: `(EXTI15_10_IRQHandler())`, `(EXTI4_IRQHandler())`, y `(EXTI9_5_IRQHandler())`.

¡Ya tenemos un sistema eficiente energéticamente! En el futuro ten siempre en consideración la importancia de estos modos de bajo consumo en cualquier sistema embebido que se alimente con baterías. Vamos a unir todas las piezas.

3.5.2 Integración final

Ya tenemos todos los módulos de las versiones V1-V3 de *Simone* desarrollados y probados: botón, teclado matricial y RGB light (LED) **RGB**. Ahora vamos a integrarlos en el sistema central, y llenar el `(main.c)` del programa. **Procedamos:**

La máquina de estados del sistema *Simone* involucra a todos los elementos del mismo y la gestión del bajo consumo. Será una implementación principalmente de la lógica de control del juego en `(fsm_simone.c)` y `(fsm_simone.h)`. La parte dependiente del hardware (`(PORT)`) relacionada con la temporización del juego (`(port_simone)`) se os proporciona parcialmente implementada para facilitar la integración.

3.5.3 Mecánica del juego y reglas

El sistema debe gestionar la lógica del juego, tiempos de espera, niveles de dificultad y la interacción con los drivers de hardware (botón, teclado matricial y LED **RGB**).

Inicio

El sistema **arranca** en reposo en el estado `(IDLE)`. Al pulsar el **botón de usuario**, el sistema arranca poniendo al estado que gestiona las secuencias de colores.

El juego tiene 3 niveles de dificultad predefinidos: fácil, medio y difícil. Al arrancar el sistema, empieza en modo fácil por defecto.

Los niveles predefinidos son: `(LEVEL_EASY)`, `(LEVEL_MEDIUM)` y `(LEVEL_HARD)`.

Consejo

Utiliza un `(enum)` para definir los niveles y `(#defines)` para definir las teclas de cada nivel. La FSM tendrá un campo `(level)` en su estructura para guardar el nivel del juego.

Generación de secuencia de colores y teclas asociadas

El juego básico usa 6 colores para mostrar al usuario de manera aleatoria en una secuencia, y se define también el *color* apagado para parpadear entre colores. Cada color estará asociado a una tecla. Los colores están definidos en `(rgb_colors.h)` como estructuras de tipo `(rgb_color_t)`. Las asociaciones son las siguientes:

Tecla	Color
'0'	<code>color_white</code>
'1'	<code>color_red</code>
'2'	<code>color_green</code>
'3'	<code>color_blue</code>
'5'	<code>color_yellow</code>
'8'	<code>color_turquoise</code>

El número de colores **se debe definir** en el fichero de cabecera con la etiqueta `NUMBER_OF_COLORS_GAME`.

Cada color de la secuencia se mostrará a una **velocidad** (tiempo que tarda en apagarse), y a una **intensidad** lumínica. La dificultad rige estos dos parámetros elegidos al inicio del juego.

Nivel	Velocidad por color	Intensidad mínima
<code>LEVEL_EASY</code>	<code>SIMONE_TIME_ON_LEVEL_EASY_MS</code> : 3000 ms	<code>LEVEL_EASY_MIN_INTENSITY</code> : 80%
<code>LEVEL_MEDIUM</code>	<code>SIMONE_TIME_ON_LEVEL_MEDIUM_MS</code> : 2000 ms	<code>LEVEL_MEDIUM_MIN_INTENSITY</code> : 50%
<code>LEVEL_HARD</code>	<code>SIMONE_TIME_ON_LEVEL_HARD_MS</code> : 1000 ms	<code>LEVEL_HARD_MIN_INTENSITY</code> : 20%

En todos los casos el tiempo de apagado entre colores es fijo: `SIMONE_TIME_OFF_BETWEEN_COLORS_MS`: 300 ms, que **se debe definir** en el fichero de cabecera. De igual modo se debe definir el tiempo de espera máximo para la entrada del usuario entre pulsaciones: `SIMONE_TIME_WAIT_INPUT_MS`: 5000 ms, y el tiempo de feedback visual al usuario tras cada pulsación: `SIMONE_TIME_VISUAL_FEEDBACK_MS`: 300 ms.

La longitud de la secuencia máxima es fija para todos los niveles `SEQUENCE_LENGTH`: 5 colores. **Cuando el jugador complete la secuencia máxima en un nivel, el sistema subirá automáticamente al siguiente nivel (si no está ya en el máximo) y reiniciará la secuencia.**

Consejo

Declara los `#define` de velocidad e intensidad en `fsm_simone.h` para mayor legibilidad.

El juego debe comportarse de forma determinista siguiendo las siguientes reglas:

Flujo del juego

15. Ronda:

- El sistema añade un color aleatorio a la secuencia.
- El sistema reproduce la secuencia completa usando el LED (respetando la velocidad del nivel actual). **Nota:** Debe haber un breve instante de apagado entre colores consecutivos para distinguirlos si son el mismo.
- El sistema espera a que el usuario repita la secuencia.

16. Turno del jugador:

- El usuario debe pulsar las teclas en el orden correcto.
- **Timeout de usuario:** Si el usuario tarda más de `SIMONE_TIME_WAIT_INPUT_MS` milisegundos en pulsar una tecla entre paso y paso, pierde la partida. Se ha establecido que este valor sea de **5000 ms**.
- Si se pulsa una tecla incorrecta, pierde la partida.
- Si se pulsa una tecla correcta, se reinicia el temporizador y el jugador tiene otros `SIMONE_TIME_WAIT_INPUT_MS` milisegundos para pulsar la siguiente tecla.

17. Victoria de ronda y juego:

- Si el usuario completa la secuencia actual correctamente, el sistema añade un nuevo color y repite el proceso (Ronda + 1).
- Si el usuario completa la secuencia de máxima longitud `SEQUENCE_LENGTH` correctamente, **aumenta de nivel**.
- Si el usuario ya estaba en el nivel máximo y completa la secuencia, gana la partida y la FSM va al estado `IDLE`.

18. Game over

- Si el usuario pierde (por error o por timeout), el sistema debe mostrar un mensaje de resultado y volver al estado de reposo para permitir empezar una nueva partida.

3.5.4 Instrucciones de implementación

Para el desarrollo de la FSM, se os proporcionan la parte portable `PORT`, algunos `#define` de `fsm_simone.h`, y algún código en `fsm_simone.c`, como dos funciones auxiliares que facilitan la conversión entre los tipos de datos:

- `_get_key_from_color()`: devuelve el carácter asociado a un color (ej. '1' para Rojo).
- `_get_color_from_key()`: devuelve el color asociado a un carácter.

Debéis implementar el resto de la lógica siguiendo la tabla de transiciones que diseñéis basándoos en [la especificación de la FSM](#). **Debéis completar el fichero de cabecera con los prototipos de función y añadir cualquier `#include`, `#define`, o función auxiliar que consideréis necesaria para el correcto funcionamiento.**

Descarga del repositorio de la asignatura los ficheros correspondientes a la parte `PORT` y `COMMON` de la librería de *Simone* correspondientes a la versión [V4](#): https://github.com/sdg2DieUpm/simone/tree/simone_v4 y colócalos en las carpetas correspondientes de tu proyecto.

3.5.5 FSM Simone. Especificación detallada

En esta ocasión no se proporciona la máquina de estados, ni funciones, ni API. Se dará el detalle de la lógica de control del juego y algún detalle de implementación más crítico, así como las restricciones a implementar.

La lógica del juego es más compleja que la de un simple periférico. El sistema debe ser capaz de generar secuencias aleatorias, reproducirlas respetando tiempos, esperar la entrada del usuario, validar dicha entrada en tiempo real y gestionar la victoria o la derrota. Para gobernar todo esto, utilizaremos una FSM central que orquestará el funcionamiento del juego.

Objetivo

El objetivo de esta versión es implementar la lógica de control del juego en `fsm_simone.c` y `fsm_simone.h`. La parte dependiente del hardware (`PORT`) relacionada con la temporización del juego (`port_simone`) se os proporciona parcialmente implementada para facilitar la integración.

Definición de la estructura de datos

Antes de dibujar estados y transiciones, es fundamental entender qué datos necesita manejar nuestra máquina para funcionar. La estructura de datos (`fsm_simone_t`) actúa como la memoria del juego. Aparte de los punteros a las otras FSM (teclado, botón, luces), necesitamos variables para gestionar la secuencia. **Complétala con los detalles que se indican a continuación.**

Hay que tener cuidado de no confundir los **diferentes índices que gestionan el progreso del juego**. Observa los campos definidos en la estructura:

- **FSMs de los elementos:**

- `f`: estructura base de la FSM de Simone de tipo `sm_t` y que **ha de ser el primer elemento de la estructura**.
- `p_fsm_button`: puntero a la FSM del botón de usuario.
- `p_fsm_keyboard`: puntero a la FSM del teclado matricial.
- `p_fsm_rgb_light`: puntero a la FSM del LED RGB.

- **Secuencia de datos:**

- `seq_colors`: un array que almacena la lista de `SEQUENCE_LENGTH` **colores** (`rgb_color_t`) de la secuencia actual.
- `seq_intensities`: un array paralelo al anterior, también de longitud `SEQUENCE_LENGTH`, y que almacena la **intensidad** ([0-100]) de cada color como un entero.
- `level`: almacena como un entero el nivel de dificultad actual definido en un enumerado (fácil, medio, difícil).

- **Índices de control** (¡Cuidado aquí!):

- `seq_idx`: indica la **longitud actual** de la secuencia que se debe jugar. Si estamos en la ronda 3, este índice valdrá 3. Determina hasta dónde tiene que llegar la máquina reproduciendo y hasta dónde tiene que llegar el jugador repitiendo.
- `playback_idx`: es el índice de **Simone**. Recorre la secuencia del array de colores y de intensidades. Indica qué color de la secuencia se está mostrando actualmente por los LED.
- `player_idx`: es el índice del **jugador**. Recorre también la secuencia, pero para comparar si el valor pulsado por el usuario es correcto.

- **Flags y otros campos:**

- `player_key`: almacena **el carácter** de la tecla que acaba de pulsar el usuario para poder verificarla con la correspondiente del color que debería haber pulsado el usuario
- `playback_over`: es un **booleano** que usaremos para controlar el parpadeo de los LED (encendido/apagado) durante la reproducción de la secuencia de colores.
- `level`: almacena el nivel de dificultad actual del juego como un entero. Albergará los valores del enumerado que contiene los niveles `LEVEL_EASY`, `LEVEL_MEDIUM` y `LEVEL_HARD`.
- `on_off_press_time_ms`: entero que almacena el tiempo que el botón de usuario ha estado presionado (para gestionar el encendido y apagado del sistema).

Completa la estructura y documéntala en `fsm_simone.h`.

Consejo sobre los índices

El juego consiste esencialmente en comparar índices.

19. La máquina Simone reproduce desde `0` hasta `seq_idx` usando su índice `playback_idx`.
20. El jugador repite desde `0` hasta `seq_idx` usando su índice `player_idx`.
21. Si `player_idx` alcanza a `seq_idx` es que todo han sido aciertos, por tanto ¡ronda superada! Se incrementa `seq_idx` y vuelve a empezar.

💡 Importante: Nombres de las constantes

Para que vuestro código pase los test automáticos de los profesores, debéis respetar escrupulosamente los nombres de los `#define` de tiempos y teclas, así como los nombres de los estados en el `enum FSM_SIMONE` definidos en el fichero de cabecera proporcionado.

Especificación de la máquina de estados

El juego consta de **7 estados**: 5 de juego y 2 de gestión de bajo consumo. **Mantén los nombres proporcionados de los estados**. Tienes que **completar y documentar todas las funciones de la tabla de transiciones y funciones auxiliares faltantes, así como la propia tabla de transiciones**. Sigue los criterios que hemos usado en las tres versiones anteriores.

La lógica del juego se divide en los siguientes estados principales. Estudia detenidamente qué debe ocurrir en cada uno y, sobre todo, qué condiciones provocan las transiciones a los siguientes estados.

ESTADO `IDLE`

Es el estado de reposo. El sistema está dormido esperando a que el usuario quiera jugar.

Entradas	Salidas
(1) al arrancar el sistema	(1) Por encendido del usuario
(2) Por victoria (desde <code>WAIT_KEY</code>)	(2) Por inactividad
(3) Por derrota (desde <code>WAIT_KEY</code>)	
(4) Por apagado del usuario (desde <code>WAIT_KEY</code>)	

Transición (1): Por encendido del usuario

La **función de comprobación** `check_on()` debe detectar si el jugador ha pulsado el botón de usuario durante el tiempo definido en `main.c` (`SIMONE_ON_OFF_PRESS_TIME_MS`). Si esto sucede, pasará al estado `ADD_COLOR` para iniciar la partida.

La **función de acción** `do_init_game()` inicializa las variables del juego. Debe resetear la duración del botón de usuario, la tecla del teclado matricial, los índices de control (`seq_idx`, `playback_idx`, `player_idx`), y las variables `playback_over`, y `player_key` (esta última al carácter definido en `KEY_NO_KEY_PRESSED`).

El nivel de dificultad lo inicializa a `LEVEL_EASY`. Inicializa cada elemento del array de secuencia de colores al color `color_off` (ver colores en `rgb_colors.c`), y cada elemento del array de las intensidades a `0`.

Llama a una **función privada auxiliar** (`add_color()`) pasándole un puntero a la máquina de estados de Simone **para añadir un color e intensidad aleatorios a la secuencia**. Será el primer color de la ronda 1.

Para que el LED RGB muestre el color, esta función debe activar el *status* de la FSM RGB light llamando a la función apropiada de dicha FSM.

Por último, antes de salir, imprime un mensaje de inicio al usuario. Algo como:

```
printf("[SIMONE][%d] Simone game INIT\n", port_system_get_millis());
```

Función auxiliar `_add_color`

Esta función privada que encapsula la generación aleatoria recibe un puntero a la FSM de Simone y debe:

22. Generar un índice aleatorio para seleccionar un color del array `p_colors_library`. Este array debe colocarse al inicio de `fsm_simone.c` y contiene direcciones los 6 colores usados en el juego.

```
const rgb_color_t *p_colors_library[] = {&color_red, &color_green, &color_blue, &color_yellow, &color_turquoise, &color_white};
```

Para generar el índice aleatorio usa la función `rand()` de la `<stdlib.h>`, y el operador módulo `%` para acotar el valor al rango `0` a `NUMBER_OF_COLORS_GAME`. El valor aleatorio se generará gracias a la semilla `srand()` iniciada en `fsm_simone_init()`.

23. Generar una intensidad aleatoria respetando los rangos definidos para el nivel actual (usando los define `LEVEL_X_MIN_INTENSITY`).

Como la función `rand()` devuelve un valor entre `0` y `RAND_MAX`, puedes usar la siguiente fórmula para acotar el valor al rango deseado:

```
random_num = (random_num % (max - min + 1)) + min;
```

Donde los valores máximos y mínimos dependen del nivel actual `level`.

24. Si `seq_idx` ha alcanzado el valor `SEQUENCE_LENGTH`, reseteamos `seq_idx`. Si no, guardamos el color e intensidad generados en las posiciones `seq_idx` de los arrays `seq_colors` y `seq_intensities`, respectivamente, y luego incrementamos `seq_idx` en 1.

Transición (2): Por inactividad

Si no hay actividad, el sistema puede dormirse pasando al estado `SLEEP_WHILE_IDLE`.

La **función de comprobación** `check_no_activity()` devuelve directamente el valor inverso al de su contraria `check_activity()`. Esta última, lo que hace es devolver `true` si alguna de las FSM de los elementos (botón, teclado, RGB light) está activa. Para ello, llama a las funciones de comprobación de actividad que hemos implementado en la **sección de bajo consumo** (`fsm_xxx_check_activity()`).

La **función de acción** `do_sleep_idle()` debe poner el sistema en un estado de bajo consumo. Para ello llama a la función de *sleep* del `PORT` del sistema que hemos implementado en la **sección de bajo consumo**.

ESTADO `ADD_COLOR`

Este es un estado de transición. El sistema no se detiene aquí esperando eventos externos, sino que realiza las operaciones lógicas necesarias para preparar la secuencia de la siguiente ronda antes de reproducirla.

Entradas	Salidas
(1) Al iniciar partida (desde <code>IDLE</code>)	(1) Secuencia actualizada
(2) Al completar ronda (desde <code>WAIT_KEY</code>)	

Transición (1): Secuencia actualizada

La función de comprobación `check_color_added()` verifica si la longitud de la secuencia (`seq_idx`) es diferente del índice del jugador (`player_idx`). Como la función auxiliar `_add_color()` habrá añadido un nuevo color, esta condición se cumplirá inmediatamente, permitiendo el paso al estado de reproducción de la secuencia `PLAYBACK`.

La función de acción `do_playback()` es el **core de la reproducción**. Su objetivo es gestionar el parpadeo de los LED respetando los tiempos de cada nivel. Dado que esta función se llama repetidamente, utiliza la variable `playback_over` como un selector para alternar entre dos fases: **mostrar color y pausa**.

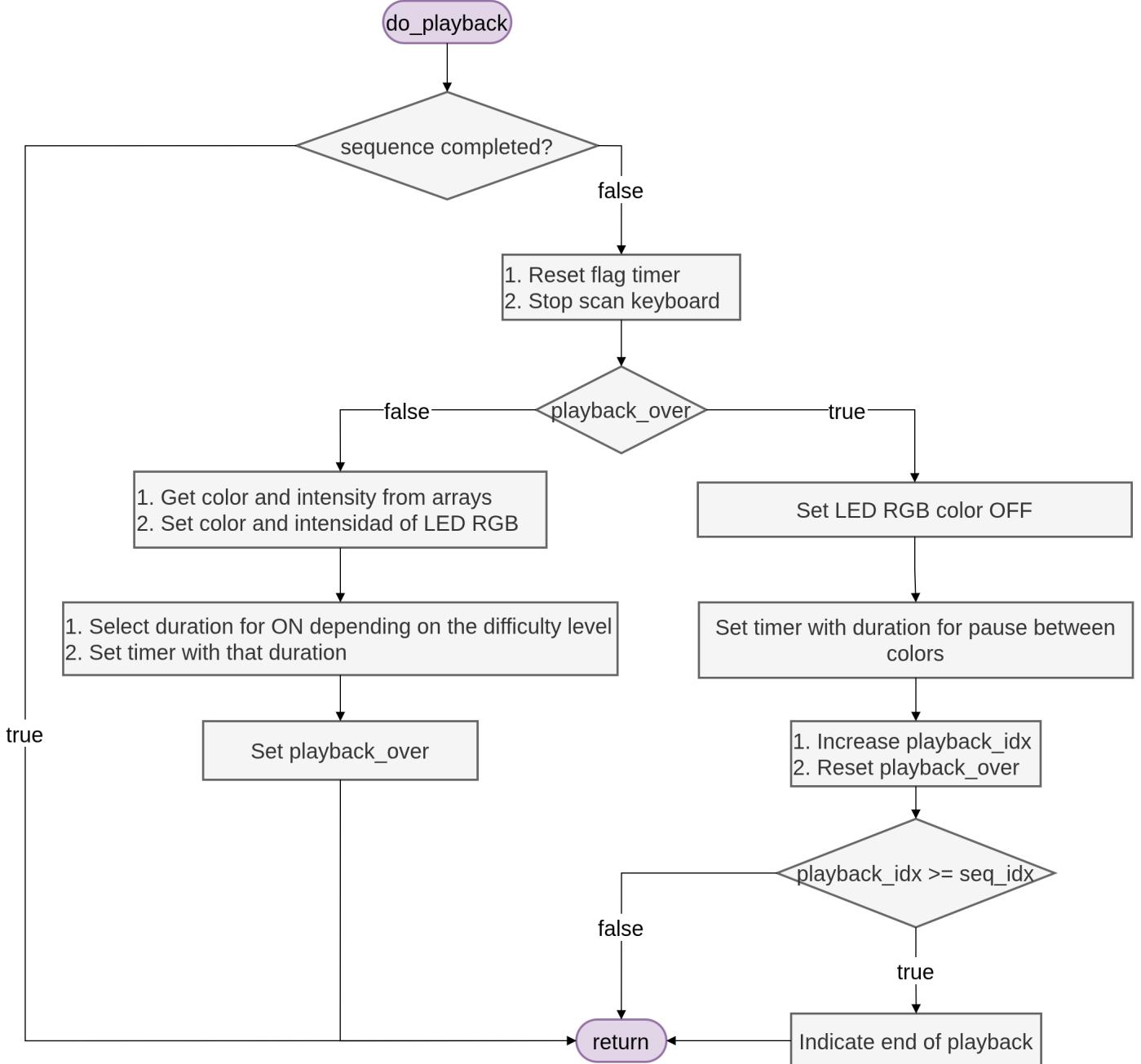


Diagrama de flujo de la función `'do_playback()'`.

Aunque el [diagrama](#) resume el flujo lógico, la implementación correcta de `do_playback()` requiere prestar atención a varios detalles técnicos para mantener la estabilidad del sistema:

25. **La naturaleza no bloqueante del temporizador:** Cuando el diagrama indica "Set timer de Simone", debes llamar a `port_simone_set_timer_timeout()`. Hay que entender que esta función **no detiene la ejecución del código** (no es un `delay`), sino que programa una interrupción futura. El microcontrolador se dormirá en el estado `SLEEP_WHILE_PLAYBACK` hasta que ese tiempo expire, interrumpa y vuelva a comprobarse la tabla de transiciones.
26. **Protección contra entradas espurias:** Una de las primeras acciones es detener el escaneo del teclado. Si no se hace, el usuario podría pulsar teclas mientras se muestran las luces; esas pulsaciones se quedarían guardadas en el *struct HW* del teclado y se procesarían erróneamente en cuanto el juego pasara al estado de espera, provocando que se detecte como una tecla mal pulsada al inicio de la ronda siguiente.
27. **Acceso a la memoria de la secuencia:** En la fase de encendido, debes recuperar la información almacenada previamente. Usa la variable `playback_idx` para acceder a los arrays paralelos `seq_colors` y `seq_intensities`. Recuerda que la función de encendido del LED (`fsm_rgb_light_set_color_intensity`) requiere ambos parámetros.
28. **Sincronización de índices:** Fíjate bien en la comparación final. Comparamos `playback_idx` (lo que estamos mostrando ahora) con `seq_idx` (la longitud total de la secuencia actual).
 - Si `seq_idx` es 3, significa que hay colores en las posiciones 0, 1 y 2.
 - Cuando terminamos de mostrar el color 2 y su pausa, incrementamos `playback_idx` a 3.
 - Como 3 es mayor o igual que 3, sabemos que hemos terminado.

El marcador de fin de playback

Para marcar el final de *playback* y poder comprobar al inicio si ha acabado o no, podemos usar varios mecanismos. Puedes usar, por ejemplo, una variable global, o puedes usar un valor inválido (que nunca vaya a ocurrir en el índice `playback_idx`). Cualquiera que uses, ten en cuenta que este marcador será la forma de comunicar a la función de comprobación `check_playback_over()` que la tarea de reproducción ha concluido.

ESTADO `PLAYBACK`

En este estado el sistema ha tomado el control para mostrar la secuencia de colores al jugador como se ha mostrado en el [flujo](#)grama de `do_playback`.

Entradas	Salidas
(1) Desde <code>ADD_COLOR</code>	(1) Apagado manual
(2) Desde <code>SLEEP_WHILE_PLAYBACK</code>	(2) Turno del jugador
	(3) Por inactividad

Transición (1): Apagado manual

La **función de comprobación** `check_off()` verifica si el botón de la placa se ha mantenido pulsado el tiempo suficiente. Es idéntica a la función `check_on()`. Si se cumple, el sistema debe volver al estado de reposo `IDLE`.

La **función de acción** `do_stop_simone()`: **resetea la duración** del botón de usuario, **desactiva el estado** del LED, **resetea el nivel** de dificultad a `LEVEL_EASY`, e imprime un **mensaje** de despedida indicando que el juego ha terminado y que puede presionar el botón para iniciar una nueva partida.

Transición (2): Turno del jugador

La **función de comprobación** `check_playback_over()` determina si la máquina ha terminado de reproducir toda la secuencia y, además, ha terminado el tiempo de espera del último apagado. Debe devolver `true` solo si se cumplen dos condiciones simultáneamente:

29. El marcador de fin de reproducción está activado (establecido en `do_playback()`).

30. El temporizador ha expirado (`port_simone_get_timeout_status()`).

La **función de acción** `do_start_player_sequence()` prepara el sistema para escuchar al usuario. La función debe:

31. **Reiniciar flag** `playback_over` y **reiniciar el índice** del jugador `player_idx` para empezar a comprobar desde el principio.

32. **Apagar LED** con el color `color_off` llamando a la función correspondiente de la FSM del LED.

33. **Set timeout** del temporizador con el tiempo máximo que tiene el usuario para reaccionar (`SIMONE_TIME_WAIT_INPUT_MS`).

34. **Iniciar el escaneo del teclado** llamando a la función correspondiente de la FSM del teclado, ya que se desactivó durante la reproducción.

35. **Imprimir** un mensaje por consola informando al usuario de que es su turno y cuántos segundos tiene para responder entre pulsaciones.

Transición (3): Por inactividad

Si no ha expirado el temporizador mientras mostramos un color o una pausa, el sistema puede dormirse para ahorrar energía.

La **función de comprobación** `check_no_activity()` devuelve el valor inverso al de su contraria `check_activity()`.

La **función de acción** `do_sleep_playback()` debe poner el sistema en un estado de bajo consumo. Para ello llama a la función de `sleep` del `PORT` del sistema que hemos implementado en la [sección de bajo consumo](#).

ESTADO `WAIT_KEY`

Es el turno del jugador. El sistema espera cualquier reacción por parte del usuario, ya sea para apagar el juego, introducir una tecla de la secuencia, o porque se ha agotado el tiempo.

Entradas	Salidas
(1) Desde <code>PLAYBACK</code>	(1) Apagado manual
(2) Desde <code>VERIFY_INPUT</code>	(2) Victoria final
	(3) Derrota por tiempo
	(4) Fin de ronda
	(5) Pulsación de una tecla

Transición (1): Apagado manual

La **función de comprobación** `check_off()` verifica si el botón de la placa se ha mantenido pulsado el tiempo suficiente. Es idéntica a la función `check_on()`. Si se cumple, el sistema debe volver al estado de reposo `IDLE`.

La **función de acción** `do_stop_simone()`: **resetea la duración** del botón de usuario, **desactiva el estado** del LED, **resetea el nivel** de dificultad a `LEVEL_EASY`, e imprime un **mensaje** de despedida indicando que el juego ha terminado y que puede presionar el botón para iniciar una nueva partida.

Transición (2): Victoria final

La **función de comprobación** `check_winner()` es la más estricta. Deben cumplirse 3 condiciones: se cumple solo si (1) el jugador ha terminado la secuencia (debes jugar con los índices del jugador y la secuencia), (2) el índice que recorre el array de la secuencia ha llegado o superado la longitud máxima (`SEQUENCE_LENGTH`) y, además, (3) estamos en el nivel `level1` de dificultad más alto (`LEVEL_HARD`).

Si se cumplen estas 3 condiciones, el jugador ha ganado la partida y pasa al estado de reposo `IDLE`.

La **función de acción** `do_winner()` **detiene el temporizador** (`port_simone_stop_timer()`), y muestra un **mensaje** de felicitación por consola indicando cuántos colores ha conseguido recordar el jugador.

Transición (3): Derrota por tiempo

La **función de comprobación** `check_player_key_timeout()` verifica si el temporizador de espera de usuario ha expirado (`port_simone_get_timeout_status()`). Si el jugador tarda demasiado en pensar (`SIMONE_TIME_WAIT_INPUT_MS`), la condición se cumple y pasa al estado de reposo `IDLE`.

La **función de acción** `do_game_over_timeout()` gestiona el fin de la partida por tiempo. **Detiene el temporizador** (`port_simone_stop_timer()`), **reinicia todos** los índices y elementos de la estructura de Simone (`seq_idx`, `player_idx`, etc.), **detiene el escaneo** del teclado antes de volver al reposo. Por último imprime un **mensaje** de *Game Over*, indicando alguna estadística relevante (por ejemplo, cuántos colores ha conseguido recordar el jugador).

Transición (4): Fin de ronda

La **función de comprobación** `check_player_round_end()` verifica si (1) el jugador ha reproducido con éxito toda la secuencia actual (comparando los índices `player_idx` y `seq_idx`), pero (2) aún no ha cumplido las condiciones de victoria total; esto es, no ha alcanzado la longitud máxima de la secuencia o no está en el nivel más alto.

Si se cumplen estas dos condiciones, el jugador ha superado la ronda y pasa al estado `ADD_COLOR` para preparar la siguiente ronda.

La **función de acción** `do_add_color()` prepara el sistema para el siguiente nivel o secuencia. Puedes ver su lógica detallada en el [flujo](#).

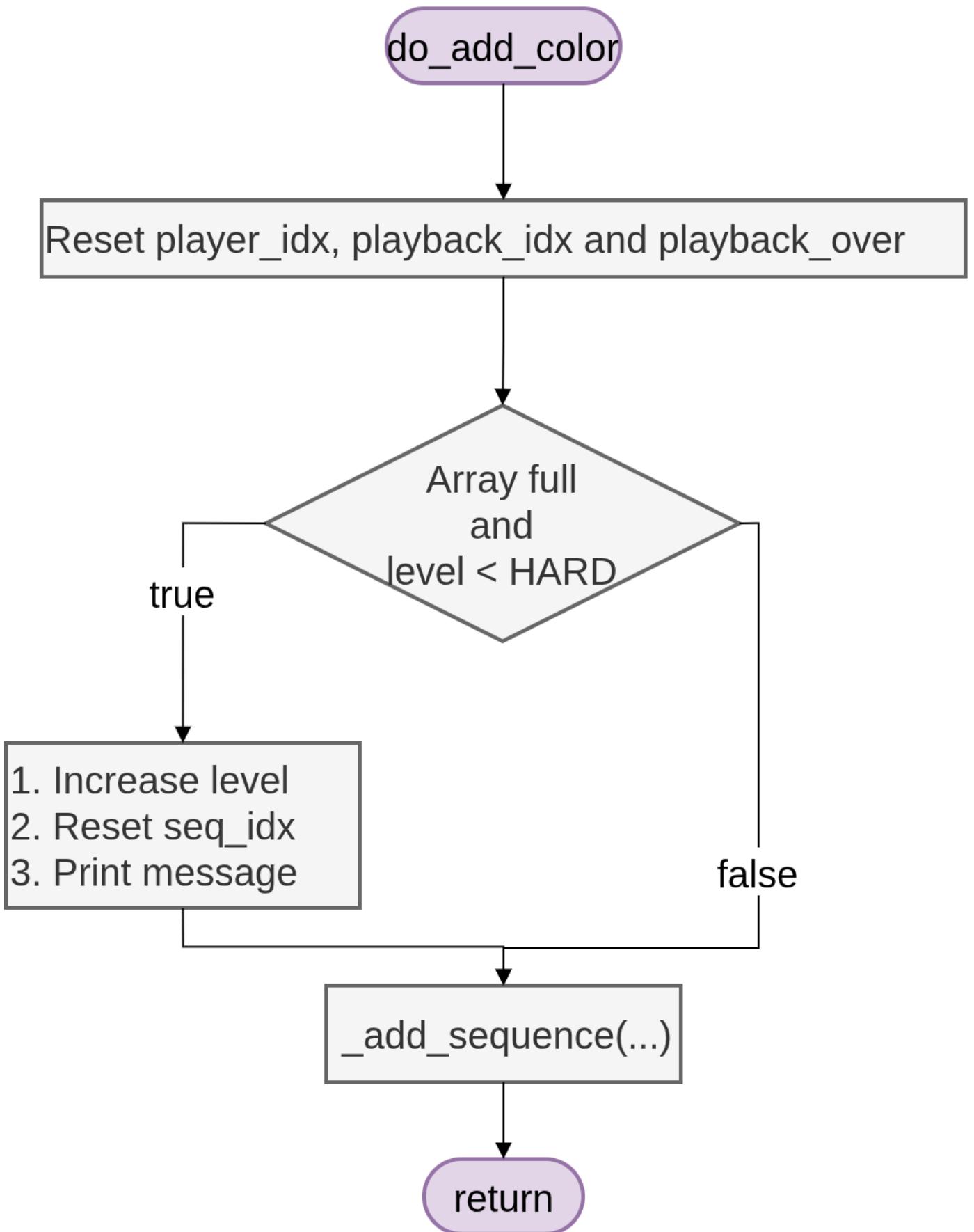


Diagrama de flujo de la función `do_add_color`.

Transición (5): Pulsación de una tecla

La función de comprobación `check_any_key_pressed()` utiliza el driver del teclado para detectar si hay alguna tecla disponible en el buffer. Devuelve `true` si el usuario ha pulsado algo (valor de tecla leída es distinto de `KEY_NO_KEY_PRESSED`). En tal caso, el sistema transiciona al estado intermedio `VERIFY_INPUT` para validar la pulsación.

La función de acción `do_capture_input()` realiza de nuevo la lectura de la tecla pulsada y proporciona *feedback* visual inmediato al usuario del color de la tecla que haya pulsado (sea correcta, o no). Debe:

36. **Obtiene la tecla del teclado** y la guarda en la variable `player_key`. Luego, **resetea el valor de la tecla** del teclado para evitar lecturas repetidas usando la función correspondiente de la FSM del teclado.
37. **Traduce la tecla a color** usando la función auxiliar `_get_color_from_key()`, y **enciende el LED** al máximo brillo con el color resultante.
38. Pone el temporizador de Simone con un tiempo breve de *feedback* visual (definido en `SIMONE_TIME_VISUAL_FEEDBACK_MS`).

ESTADO `VERIFY_INPUT`

Este es un estado temporal de retención. El sistema entra aquí justo después de que el usuario pulse una tecla para mantener el LED encendido durante un breve instante (`SIMONE_TIME_VISUAL_FEEDBACK_MS`), permitiendo al jugador ver qué color ha seleccionado. En este estado se comprueba si la tecla leída es la esperada, o no.

Entradas	Salidas
(1) Desde <code>WAIT_KEY</code>	(1) Tecla correcta (2) Tecla incorrecta

Transición (1): Tecla correcta

La función de comprobación `check_input_valid()` realiza una validación en dos pasos:

39. **Espera visual:** Primero verifica si el temporizador de feedback visual ha expirado (`!port_simone_get_timeout_status()`). Si el tiempo no ha pasado, devuelve `false` y sale de la función.
40. **Validación lógica:** Si el tiempo ha pasado, compara la tecla pulsada guardada en la estructura con la esperada según la secuencia (ayúdate de la función auxiliar `_get_key_from_color()`). Devuelve `true` solo si coinciden. En este caso transicionará de vuelta al estado `WAIT_KEY` para esperar la siguiente pulsación.

La función de acción `do_valid_key()` consolida el progreso del jugador. Debe:

41. **Apagar** el LED con el color `color_off`.
42. **Aumentar** el índice del jugador.
43. **Reiniciar la tecla** guardada en la estructura a `KEY_NO_KEY_PRESSED`.
44. **Reiniciar el temporizador** (`port_simone_set_timer_timeout()`) para dar tiempo al usuario a pulsar la siguiente tecla.

Transición (2): Tecla incorrecta

La función de comprobación `check_input_invalid_finished()` sigue la misma lógica temporal que la anterior, pero devuelve `true` si la tecla pulsada es **diferente** a la esperada.

La función de acción `do_game_over_invalid_key()` gestiona la derrota:

45. **Apaga** el LED para finalizar el feedback.
46. **Reinicia** todos los elementos de la estructura de Simone (`seq_idx`, `player_idx`, `player_key`, etc.) para que la próxima vez se empiece desde cero.

47. **Reiniciar el temporizador** de Simone (`port_simone_stop_timer()`) y *detiene el escaneo** del teclado usando la función correspondiente de la FSM del teclado.
48. **Muestra por consola** información sobre qué tecla se esperaba y cuál se pulsó realmente, junto con un mensaje de *Game Over*. Hace uso de la función auxiliar `_get_key_from_color()`.

ESTADO `SLEEP_WHILE_IDLE`

Es el estado de bajo consumo. El sistema entra aquí cuando está en reposo (`IDLE`) y no hay ninguna interacción por parte del usuario, permitiendo ahorrar energía mientras se espera a que se inicie una nueva partida.

Entradas	Salidas
(1) Desde <code>IDLE</code>	(1) Detección de actividad
(2) Desde <code>SLEEP_WHILE_IDLE</code> (autotransición)	(2) Sin actividad

Transición (1): Detección de actividad

La **función de comprobación** `check_activity()` verifica si algún periférico ha generado un evento (botón pulsado, tecla pulsada, etc.). Al despertar por una interrupción, esta condición se cumple y el sistema transiciona de vuelta a `IDLE` para procesar dicho evento.

En esta transición **no hay función de acción** asociada (es `NULL`), ya que la propia salida del estado de sueño es suficiente para reactivar la lógica principal.

Transición (2): Sin actividad

La **función de comprobación** `check_no_activity()` confirma que el sistema sigue inactivo. Esta autotransición sirve para gestionar el bajo consumo en modo depuración, cuando se despierta por un *breakpoint* o similar.

La **función de acción** `do_sleep_idle()` detendrá el reloj de la CPU hasta que ocurra la próxima interrupción.

ESTADO `SLEEP_WHILE_PLAYBACK`

Este estado gestiona el bajo consumo **durante la reproducción**. Mientras el LED está encendido mostrando un color o apagado durante una pausa, no es necesario que la CPU esté consumiendo ciclos. El sistema duerme aquí hasta que el temporizador interrumpe.

Entradas	Salidas
(1) Desde <code>PLAYBACK</code>	(1) Timeout del color/pausa
(2) Desde <code>SLEEP_WHILE_PLAYBACK</code> (autotransición)	(2) Sin actividad

Transición (1): Timeout del color/pausa

La **función de comprobación** `check_playback_color_timeout()` consulta al si el temporizador (`port_simone_get_timeout_status()`) configurado en el paso anterior ha expirado. Si es así, significa que es hora de cambiar el estado del LED.

La **función de acción** asociada es `do_playback()`. Esta transición devuelve al sistema al estado `PLAYBACK`, ejecutando inmediatamente la lógica de alternancia de luces (encender/apagar) descrita en el *fluograma de dicho estado*.

Funcionamiento cíclico

Observa que el sistema entra y sale constantemente entre `PLAYBACK` y `SLEEP_WHILE_PLAYBACK`.

49. `PLAYBACK` configura el LED y el temporizador, y salta a dormir.
50. Espera dormido en bajo consumo.
51. Timer interrumpe y vuelve a `PLAYBACK` para cambiar el LED, y puede volver a dormir de nuevo.

Transición (2): Sin actividad

La función de comprobación `check_no_activity()` verifica que no hay eventos pendientes.

La función de acción `do_sleep_playback()` detendrá el reloj de la CPU hasta que ocurra la próxima interrupción de la misma forma que lo hace `do_sleep_idle()`, o `do_sleep_playback()`.

Funciones de bajo consumo

Habrás notado que hay 3 funciones que hacen lo mismo: `do_sleep_idle()`, `do_sleep_playback()` y `do_sleep_playback()`. Esto es una buena práctica para poder saber dónde está y de dónde viene el sistema cuando se está depurando.

Bajo consumo durante la lectura del teclado

Nótese que **no hay bajo consumo en el estado `WAIT_KEY`**. Esto es intencionado, ya que el jugador debe poder interactuar en cualquier momento. Por la forma en la que se excitan y leen las filas y columnas, la gestión del bajo consumo aquí es posible, pero más complicada de manejar. **Se deja como implementación a elegir en la Versión 5.**

INICIALIZACIÓN DE LA FSM SIMONE

Ya hemos codificado las funciones de entrada y salida, ahora vamos a codificar las funciones privadas que nos quedan.

Codifica la función `fsm_simone_init()` de forma análoga a las anteriores máquinas de estados.

52. Llama a la función `fsm_init()` pasándole el puntero a la máquina de estados, y el array de transiciones.
53. Inicializa el HW asociado a la FSM de Simone llamando a `port_simone_init()`.
54. Inicializa en la estructura todos los elementos que serán recibidos: los punteros a las máquinas de estados de los elementos del sistema *Simone*, el tiempo de pulsación del botón para encender y apagar, y el nivel inicial.
55. Inicializa la semilla aleatoria con la función `srand(time(NULL))` para asegurar que los números aleatorios generados sean diferentes en cada ejecución. Deberás importar la cabecera `<time.h>` para usar la función `time()`.
56. Imprime un mensaje por consola indicando al usuario que debe pulsar el botón para iniciar una nueva partida.

Codifica las funciones `fsm_simone_fire()` y `fsm_simone_destroy()` de forma análoga a las anteriores máquinas de estados.

Ya hemos terminado con el FSM, ahora vamos a integrar todas las FSM en el `main.c` y a probarlo.

3.5.6 Integración HW-SW de la FSM Simone

Ha llegado la hora de integrar la parte HW-SW del sistema, y depurar. Vamos a escribir las líneas de código necesarias en `main.c` para probar que funciona. **Procedamos:**

57. Abre el fichero `main.c` e incluye las cabeceras necesarias.

58. Define la macro `SIMONE_ON_OFF_PRESS_TIME_MS` como indica la API para definir una pulsación larga como aquella que supere \1 s\). ¡Ojo, porque el tiempo hay que darlo en **milisegundos!** Este es el tiempo que se debe mantener pulsado el botón para encender y apagar el juego.

59. Despues de la inicialización del sistema con la llamada a la función `port_system_init()`, crea la máquina de estados para el *botón*. Dale un nombre representativo (*e.g.* `p_fsm_button`). Para ello llama a la función `fsm_button_new()` con los argumentos necesarios. A continuación, haz lo propio con la máquina de estados del *teclado matricial*, y con la del *RGB light* trasero. Dales nombres representativos.

60. Crea la máquina de estados para el sistema *Simone*, puedes darle un nombre representativo (*e.g.* `p_fsm_simone`). ¡No pases los valores “a pincho”, usa los `#define` que has creado!

61. En el bucle `while`, lanza constantemente la función `fsm_xxx_fire()`, para las máquinas de estados del *botón*, el teclado, el RGB light y *Simone*.

Conviene que el sistema sea lo último porque depende de la actualización del estado de las FSM de los elementos anteriores.

62. Por ultimo solo nos queda un aspecto meramente formal, casi académico. Cuando creamos las máquinas de estado con las funciones `fsm_xxx_new()` estamos reservando memoria de forma dinámica (con la función `malloc()`). Cuando las máquinas de estado dejan de usarse, esa memoria debe ser liberada para poder ser usada por otras partes del código. Esa liberación se hará con la llamada a la función `fsm_destroy()`.

Después del bucle `while` llama a `fsm_xxx_destroy()` para cada una de las FSM pasándole su tipo concreto. Esto libera la memoria de cada una de las máquinas de estado creadas: *botón*, *teclado matricial*, *RGB light*, y *Simone*.

Como se decía, esto es pura ortodoxia, porque el bucle `while` del `main` es infinito, y nunca saldrá de ahí, por lo que nuestras FSM nunca dejarán de usarse y las líneas que acabas de escribir con `fsm_xxx_destroy()` nunca se ejecutarán. No obstante, conviene que sepas que así debería hacerse.

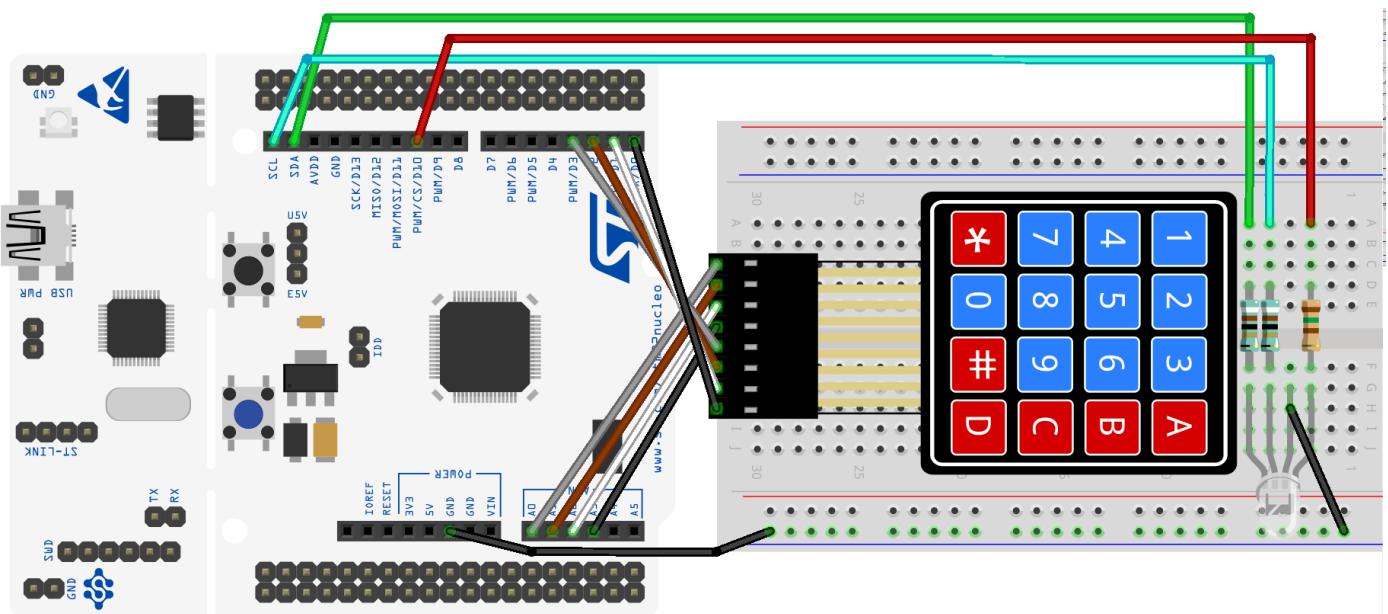
63. Compila y comprueba que no tiene ningún error. Si tienes, corrígelos.

Puede ser buen momento ahora para **documentar todo el código con Doxygen**.

3.5.7 Test de integración de Simone

Test de funcionalidades

Ya has probado los test de ejemplo de los distintos elementos y que se os han sido proporcionado. Ahora vamos a probar el sistema *Simone* completo. No se os va a pedir hacer un test formal, pero con todo el trabajo que has hecho, es conveniente que compruebes que todo funciona correctamente. Monta el circuito como se muestra en la . Prueba todo el sistema como en [el vídeo demostración Simone](#). **Procedamos:**



Montaje final con el teclado, el botón y el LED RGB.

64. Conecta la placa **Nucleo-STM32** al ordenador.
 65. Compila y carga el programa `main` en la placa ( `Clean and Debug`). Comprueba que no tienes errores de compilación.
 66. Prueba que el botón enciende y apaga el juego *Simone*.
 67. Prueba que el RGB light se enciende de manera acorde en la primera ronda y enciende un color y mostrando los mensajes por la terminal oportuna
 68. Prueba que el teclado recoge correctamente las teclas, tanto cuando aciertos, como cuando fallas.
 69. Prueba que el nivel de dificultad sube correctamente cada vez que se completa una secuencia de longitud `SEQUENCE_LENGTH`, hasta un máximo de 3 niveles.
 70. Prueba que mientras está haciendo el *playback* no responde al teclado.
 71. Prueba que puedes apagarlo y encenderlo en cualquier momento.
 72. Prueba que, estando apagada, el juego *Simone* no responde al teclado ni muestra nada por el LED RGB.
 73. Prueba, en general, el funcionamiento correcto como en el vídeo de demostración. Si encuentras algún error, corrígetelo.

Comprobación de bajo consumo

Comprobaremos que el modo *sleep* de bajo consumo se gestiona correctamente. Compila y comprueba que no tienes errores de sintaxis o de código. Para comprobar que el sistema está dormido, podemos hacerlo de dos formas:

74. Lo más habitual —si no tenemos acceso a un depurador y si tenemos que caracterizar nuestro producto— sería hacerlo mediante la medición del consumo del microcontrolador (¡no de los elementos HW de nuestro sistema!). Para medir el consumo, ve el punto “6.6 JP6 (IDD)” del manual de la placa ². Si vas a medirlo, también deberías desconectar los *jumpers* del [ST-LINK](#) del conector [CN2](#). Para ver el ahorro tendríamos que medirlo en ejecución sobre versión final. Esto puedes hacerlo si deseas como funcionalidad extra en la Versión 5, e incluir la información en la documentación del código (fichero [README.md](#)).

75. Lo que haremos para demostrar que el sistema alterna entre el modo *despierto* y *dormido* será depurando. Continúa con la depuración sin poner puntos de parada. Cuando el sistema esté inactivo, pausa la depuración y comprueba que se ha detenido en la línea de código tras la llamada a *wait for interrupt* ([_WFI\(\)](#)), similar a como se muestra en la . Esto querrá decir que, efectivamente, la ejecución estaba detenida *esperando una interrupción*, se ha despertado, y ha pasado a la siguiente línea de código.

```

130  /* Select Stop mode entry : Request Wait For Interrupt */
131  __WFI() ← Aquí se queda detenido
132
133 }
134 } ← Aquí sigue cuando se despierta
135
136 void port_system_sleep()
137 {
138     port_system_systick_suspend();
139     port_system_power_sleep();
140 }
```

CALL STACK

- port_system_power_sleep@0x08002dfc ...
- port_system_sleep@0x08002dfc port/n...
- do_sleep@0x0800153c common/src/fsm...
- fsm_fire@0x08001dd4 common/src/fsm.c
- main@0x08001cf8 common/src/retina.c

Restauración del modo despierto vista en depuración.

Realiza con *Paint*, *Drawio*, o cualquier programa que elijas el diagrama de la FSM del sistema con todas sus transiciones. Añádela a tu [README.md](#).

¡Ya tenemos el sistema *Simone* funcionando! **No olvides documentarlo** (vídeo "[MatrixMCU] Documentación de código con Doxygen"). En la siguiente versión podrás añadir más funcionalidades a tu elección. Puedes incluir más capturas o imágenes para enriquecer la documentación.

Guarda una copia de su proyecto como `simone_v4` para tener un punto de partida para la siguiente versión, y una copia de seguridad por si algo falla. **Esta copia súbela al buzón de entrega de la asignatura separada de la que hagas con la versión 5, que tiene otro buzón.** ¡Ánimo!

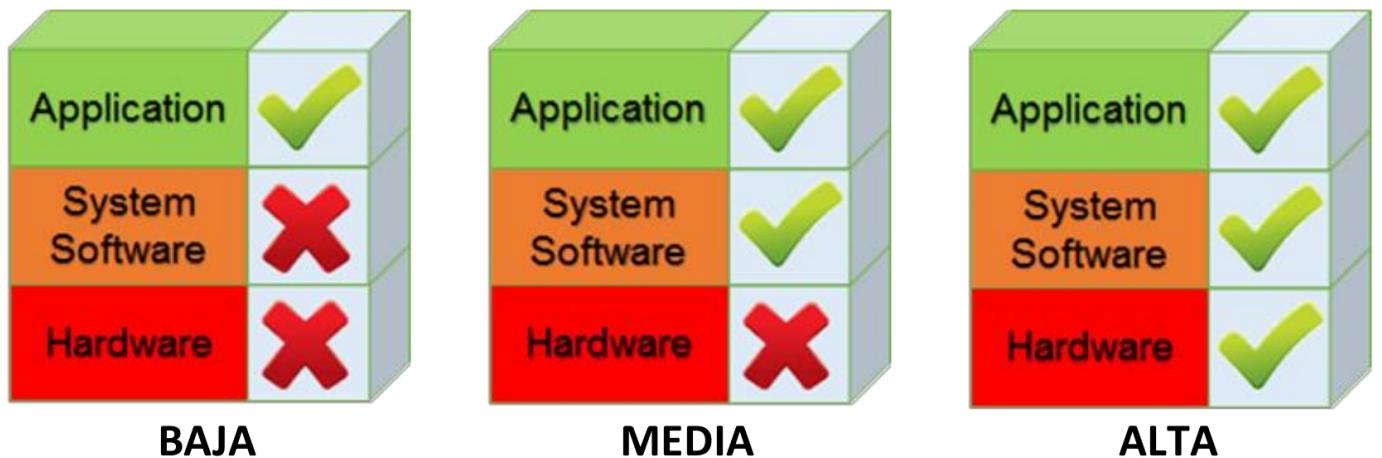
76. Josué Pagán Ortiz, Pedro José Malagón Marzo, Román Cárdenas Rodríguez, and Juan José Gómez Valverde. *Fundamentos teóricos de sistemas basados en microcontrolador STM32. Sistemas Digitales II, Sistemas Electrónicos*. Josué Pagán Ortiz, Madrid, March 2025. URL: <https://oa.upm.es/88460/>. ←
77. STMicroelectronics. Um1724 user manual. stm32 nucleo-64 boards. Technical Report, STMicroelectronics, 2020. URL: https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf. ←

3.6 Versión 5: funcionalidades de libre elección

En las últimas sesiones puedes poner a punto la versión final del sistema e incluir diversas funcionalidades a tu elección. Con ello podrás alcanzar la máxima nota. **Recuerda que (i) el proyecto es el (70%) de la nota total, y (ii) que el proyecto propuesto con especificaciones mínimas obligatorias tiene un máximo de 8 puntos. Estas funcionalidades de libre elección son un (20%) de la nota final, ¡no las dejes!**

3.6.1 Posibles funcionalidades

Las sugerencias que se muestran son orientativas. La valoración de la dificultad (de diseño y de implementación) también. No obstante, se valorarán especialmente aquellas funcionalidades que conlleven algún tipo de esfuerzo por parte del alumno a un mayor número de niveles. Así, se tendrán en cuenta los 3 niveles de la arquitectura típica de cualquier sistema embebido: (i) nivel **SW** o de aplicación (nivel superior en verde de la pila presentada en la), nivel **HW** (nivel inferior en rojo), y nivel de **SW** de sistema o nivel dedicado a la integración **HW-SW** (nivel intermedio en naranja), mediante el uso de *drivers* (i.e. controladores o librerías) **SW** para el manejo de dispositivos.



Niveles de valoración de las posibles funcionalidades a implementar.

Así, el impacto de la funcionalidad se valorará según las escalas de la [figura de niveles](#), y podría puntuar, **orientativamente**, como se indica:

- **Solo el primer nivel: BAJA valoración**, por tratarse de una funcionalidad que solo requiere modificar código. **Hasta (0.5) puntos.**
- **Solo los dos primeros niveles: valoración MEDIA**, por tratarse de una funcionalidad que, además de modificar el código, requiere del uso de algún nuevo recurso del **STM32F446RE** o la placa (como por ejemplo otro temporizador, PWM, uso de **ADC**, **FSM**) o de una configuración alternativa de los ya usados (bien sean **HW** o **SW**). **Entre (0.5) y (1.0) puntos.**
- **Los tres niveles: valoración ALTA**, por afectar a todos los niveles de diseño del sistema, por lo que es fundamental que se incluya nuevo **HW** al sistema. **Entre (1.0) y (1.5) puntos.**

En cualquier caso, las funcionalidades aquí propuestas o las que tú sugieras, pueden implementarse de distintas formas más o menos **complejas** y con distintos grados de perfección (**calidad**), por lo que la calificación final dependerá de ambos aspectos. Igualmente, la **originalidad** o la **novedad** de tus propuestas, será tenida en cuenta para la evaluación. **Hacer una funcionalidad no implica necesariamente conseguir toda la puntuación. Del mismo modo, podrían conseguirse más puntos de los previstos, si se merece.**

ATENCIÓN

Es obligatoria para todas las funcionalidades meter su documentación Doxygen incluyendo, si es necesario, diagramas o imágenes que expliquen el funcionamiento de la nueva funcionalidad. **Se ha de hacer un vídeo demostrativo corto, 2-3 minutos con lo más relevante de tus propuestas. Incluye un enlace público al vídeo en el README.md.**

Recuerda que tienes el vídeo de ayuda "[MatrixMCU] Documentación de código con Doxygen".

Podrías hacer uso de HW disponible en el laboratorio como teclados matriciales de 16 botones y pantallas **LCD**. También puedes añadir el tuyo. A continuación se muestran algunas posibles ideas para añadir al sistema o convertirlo en parte de otro proyecto mayor. Pero recuerda que **¡las más interesantes son las que a ti se te puedan ocurrir!**

Si tienes dudas sobre la implementación o valoración de estas u otras funcionalidades, no dudes en ponerte en contacto con cualquiera de los profesores de la asignatura.

Guarda una copia de tu proyecto como `simone_v5` separada de la que contiene los requisitos básicos V1-V4, y súbelo a Moodle en el buzón preparado para V5. **No olvides incluir el vídeo demostrativo en el README.md.**

Descripción	Dificultad
Añadir un modo inverso para que el jugador tenga que repetir la secuencia empezando por el último color mostrado hasta el primero.	 BAJA
Tecla wildcard para permitir repetir la última secuencia.	 BAJA
Sistema de vidas para permitir un margen de error antes del <i>Game Over</i> , indicando los fallos restantes con parpadeos o colores específicos.	 BAJA
Añadir estadísticas de partida para medir y mostrar por consola el tiempo medio de reacción del jugador entre colores o la ronda máxima alcanzada.	 BAJA
Meter el tiempo de anti-rebote como característica HW del botón y que no se le pase a la FSM del <i>botón</i> , sino que se le pida al PORT .	 BAJA
Que haya un indicador LED para mostrar las distintas situaciones del sistema (cuando se ha encendido, apagado, etc.).	 BAJA
Añadir un nuevo botón para gestionar algún aspecto de la gestión del juego, o añadir <i>estadísticas</i> de uso en un campo de la FSM.	 BAJA-MEDIA
Que el botón mida el tiempo de pulsación con <i>input capture</i> (en lugar de con <i>ticks</i>) en proporción al tiempo que ha estado encendido el LED.	 BAJA-MEDIA
Implementar alguna funcionalidad que tenga en cuenta que el botón de usuario está pulsado más de un tiempo determinado.	 BAJA-MEDIA
Añadir un zumbador (Buzzer) para generar tonos musicales distintos mediante PWM asociados a cada color sincronizados con la luz.	 MEDIA-ALTA
Diseñar o adaptar diseños de impresiones 3D para Nucleo y hacer el sistema más robusto (ver diseños en Thingiverse o Tinkercad).	 MEDIA
Multijugador local integrando un segundo teclado matricial y modificando la FSM para permitir un modo <i>versus</i> (uno propone secuencia, otro repite).	 MEDIA-ALTA
Desarrollo de test unitarios y ejemplos exhaustivos de todo el proyecto (cobertura de código superior al habitual).	 MEDIA-ALTA
Multijugador remoto conectando dos placas STM32 mediante UART, I2C o transceptores inalámbricos para jugar una partida sincronizada a distancia.	 ALTA

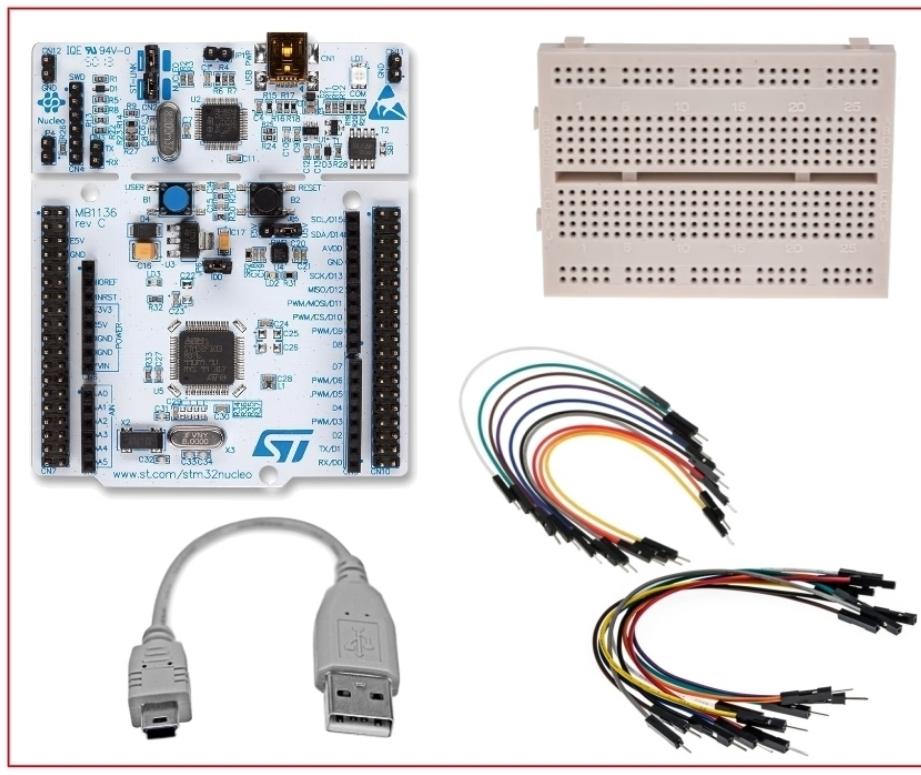
Descripción	Dificultad
Crear una interfaz gráfica de PC que se comunique por puerto serie y muestre el tablero de colores y puntuaciones en la pantalla del ordenador en tiempo real.	 ALTA
Integrar un lector de tarjetas RFID o cualquier otra forma de comunicación inalámbrica que permita interactuar con el sistema.	 ALTA
Nuevos esquemas y montajes PCB.	 ALTA
Integrar interfaces de visualización alternativa (display LCD, web, móvil), comunicación con la nube...	 ALTA

4. Apéndices

4.1 Lista de materiales (BOM)

En los enlaces podréis encontrar algunos suministradores y tener acceso a las hojas de características (*datasheets*) de los componentes.

Comp.	Concepto	Descripción	Uds.	Partida	URL
-	Nucleo-F446RE	Placa de desarrollo	1	BASE	L1, L2, L3
-	Cable USB A-Mini B	Importante: cable de alimentación y datos	1	BASE	Link
-	Latiguillos macho-macho	Cables de conexión para protoboard	+10	BASE	Link
-	Latiguillos macho-hembra	Cables de conexión para protoboard	+10	BASE	Link
-	Protoboard	Tamaño mínimo: \text{80} \times 60 mm	1	BASE	Link
-	Teclado matricial	Membrana 4x4 alfanumérico	1	TECLADO	L1, L2, L3
D1	LED RGB cátodo común	De inserción. e.g. \text{(60}^\circ\text{) visión}	1	DISPLAY	Link
R_R	Resistencia \text{(150}\sim\text{\Omega)}	De inserción. Tol: \text{(\leq 10\%)} Pot: \text{(\frac{1}{4}-\frac{1}{2} W)}	1	DISPLAY	Link
R_G, R_B	Resistencia \text{(91}\sim\text{\Omega)}	De inserción. Tol: \text{(\leq 10\%)} Pot: \text{(\frac{1}{4}-\frac{1}{2} W)}	2	DISPLAY	Link



TECLADO MATRICIAL



DISPLAY

BASE

Figura BOM. Materiales del proyecto *Simone* agrupados por partida.

5. Acrónimos

5.1 Acrónimos

Acrónimo	Definición
ADC	Analog-Digital Converter (Conversor Analógico-Digital)
AHB	Advanced High-performance Bus (Bus Avanzado de Alto rendimiento)
APB	Advanced Peripheral Bus (Bus Avanzado de Periféricos)
API	Application Programming Interface (Interfaz de Programación de Aplicaciones)
ARM	Advanced Machine
ASCII	American Standard Code for Information Interchange, Código Estadounidense Estándar para el Intercambio de Información
BOM	Bill Of Materials (Lista De Materiales)
CELT	Circuitos Electrónicos
CISC	Complex Instruction Set Computer (Computador con Conjunto de Instrucciones Complejo)
CMSIS	Common Microcontroller Software Interface Standard
CPU	Central Processing Unit (Unidad Central de Procesamiento)
DIY	Do It Yourself (Hazlo Tú Mismo)
EXTI	External Interrupt/~Event Controller (Controlador de Interrupciones Externas)
FSM	Finite State Machine (Máquina de Estados Finitos)
FPU	Floating Point Unit (Unidad de Punto Flotante)
FW	Firmware
GPIO	General Purpose Input/Output (Entrada/Salida de Propósito General)
HAL	Hardware Abstraction Layer (Capa de Abstracción Hardware)
HSE	High Speed External clock
HSI	High Speed Internal clock

Acrónimo	Definición
HW	Hardware
IDE	Integrated Development Environment (Entorno de Desarrollo Integrado)
IoT	Internet of Things
ISR	Interrupt Service Routine (Rutina de Servicio/Atención a la Interrupción)
ITM	Instrumentation Trace Macrocell
LCD	Liquid-Crystal RGB light (Pantalla de Cristal Líquido)
LED	Light-Emitting Diode (Diodo Emisor de Luz)
MCU	Microcontroller Unit (Unidad de Microcontrolador)
NVIC	Nested Vectored Interrupt Controller (Controlador de Vector de Interrupciones Anidadas)
PC	Personal Computer (Ordenador Personal)
PCB	Printed Circuit Board (Placa de Circuito Impreso)
PWM	Pulse Width Modulation (Modulación de Ancho de Pulso)
RCC	Reset and Clock Controller (Controlador de Reinicio y Reloj)
SDG1	Sistemas Digitales I
SDG2	Sistemas Digitales II
ST	STMicroelectronics
SW	Software
USB	Universal Serial Bus