

Escuela Técnica Superior de
Ingenieros de Telecomunicación

Fundamentos teóricos de sistemas basados en microcontrolador STM32

-

Sistemas Digitales II
Sistemas Electrónicos



Escuela Técnica Superior de Ingenieros de Telecomunicación

Fundamentos teóricos de sistemas basados en microcontrolador STM32

-

Sistemas Digitales II
Sistemas Electrónicos

Autores:
Josué Pagán Ortiz
Pedro José Malagón Marzo
Román Cárdenas Rodríguez
Juan José Gómez Valverde
Departamento de Ingeniería Electrónica
Contacto: j.pagan@upm.es

Versión 1.2.2

Copyright 2025 – Universidad Politécnica de Madrid
Todos los derechos reservados.
Material docente. No está permitida la venta o distribución sin autorización de los autores.
Las imágenes son de elaboración propia, cuentan con permiso de los autores, o son de dominio público.

Prefacio

Este libro contiene los fundamentos teórico-prácticos necesarios para el desarrollo de prácticas de laboratorio utilizando la placa **Nucleo-STM32**. Es un libro de consulta, aunque las introducciones de los capítulos y secciones bien merecen una lectura para situarse.

Si bien este libro se gestó como libro pensado para las asignaturas de Sistemas Digitales II del Grado en Ingeniería de Tecnologías y Sistemas de Telecomunicación y Sistemas Electrónicos del Grado en Ingeniería Biomédica de la Universidad Politécnica de Madrid, puede ser de utilidad para cualquier persona que quiera iniciarse en el mundo de los sistemas embebidos y la programación de microcontroladores. En este libro se asume que el lector tiene conocimientos básicos de programación en C, y de sistemas digitales (combinacionales y secuenciales), y que debería haber estudiado los conceptos de arquitectura de un microcontrolador, y de los periféricos más comunes. Es por ello que no se profundiza en estos conceptos, sino que se hace un repaso de los más importantes y relacionados con el microcontrolador **STM32F446RE**.

Este libro pretende ser un alivio para el alumno en la consulta de los manuales de referencia y *datasheets* del microcontrolador con el que trabajaremos (**STM32F446RE**). No obstante, no es substitutivo de dichos documentos. Este libro puede no cubrir por completo todos los bloques y periféricos del microcontrolador en la versión actual, e irán apareciendo nuevas versiones con más periféricos según se vaya adaptando a las necesidades de las asignaturas que lo usen.

Se trabajará con el lenguaje de programación `C` a bajo nivel (*bare metal*). Se emplaza al alumno a usar librerías de alto nivel (HAL) si así lo desea. Existen muchísimos recursos y ejemplos *online*, así como libros.

Este libro de fundamentos teóricos se complementa con otro libro de tutoriales aplicados a la placa **Nucleo-STM32F446RE** y orientados a la programación de máquinas de estado finitas.

Esperemos que le sea de utilidad y que, al final, haya despertado su interés por los sistemas embebidos.

Índice general

Prefacio v

Lista de acrónimos xv

1. Placas de desarrollo Nucleo-STM32 3
- 1.1. Placas de desarrollo Nucleo-STM32 . . . 4
  - 1.1.1. Por qué Nucleo-STM32 . . . 8
- 1.2. Placa de desarrollo **Nucleo-STM32F446RE** . . . 9
  - 1.2.1. Alimentación de la placa . . . 10
  - 1.2.2. Luces LED de la placa . . . 13
  - 1.2.3. Botones de la placa . . . 13
  - 1.2.4. Medida de consumo del microcontrolador . . . 14
  - 1.2.5. ST-LINK/V2-1 . . . 14
  - 1.2.6. Conectores y zócalos de la placa . . . 14

2. Arquitectura del procesador ARM 19
- 2.1. Procesadores ARM . . . 19
  - 2.1.1. Familia ARM Cortex-M . . . 22

3. El microcontrolador STM32F446RE 25
- 3.1. Características básicas del **STM32F446RE** . . . 25
- 3.2. Diagrama de bloques interno . . . 27
- 3.3. Características eléctricas y alimentación . . . 28
- 3.4. Modos de bajo consumo . . . 31
- 3.5. Arranque del sistema . . . 33
- 3.6. Resumen . . . 34

4. Entradas y salidas de propósito general (GPIOs) 35
- 4.1. Configuración de GPIOs . . . 36
- 4.2. Registros de las GPIOs . . . 37
  - 4.2.1. GPIO port mode register (GPIOx_MODER) . . . 37
  - 4.2.2. GPIO port pull-up/pull-down register (GPIOx_PUPDR) . . . 41

4.2.3. GPIO alternate function registers (GPIOx_AFRL y GPIOx_AFRH) 42
4.2.4. GPIO port input data register (GPIOx_IDR) 43
4.2.5. GPIO port output data register (GPIOx_ODR) 43
4.2.6. GPIO port bit set/reset register (GPIOx_BSRR) 44

5. Circuito de reloj **47**
5.1. Relojes de periféricos 49
5.2. Temporizadores (timers) 51
5.2.1. Registros de los timers 53
5.2.2. Control register (TIMx_CR1) 53
5.2.3. DMA/Interrupt Enable Register (TIMx_DIER) 54
5.2.4. Capture/Compare mode register (TIMx_CCMR1 y TIMx_CCMR2) 54
5.2.5. Capture/Compare Enable Register (TIMx_CCER) 56
5.2.6. Counter (TIMx_CNT) 56
5.2.7. Prescaler (TIMx_PSC) 56
5.2.8. Auto-reload Register (TIMx_ARR) 56
5.2.9. Event Generation Register (TIMx_EGR) 56
5.2.10. Capture/Compare Register TIMx_CCRx 56
5.2.11. Status Register (TIMx_SR) 58
5.2.12. Ejemplo: timer para interrupción periódica 58
5.2.13. Ejemplo: timer para PWM 61
5.2.14. Ejemplo: captura de entrada (input capture) 65

6. Interrupciones externas **73**
6.1. El NVIC 74
6.2. Interrupciones externas (EXTI) 75
6.3. Registros de interrupciones externas 78
6.3.1. Interrupt Mask Register (**EXTI_IMR**) 78
6.3.2. Rising Trigger Selection Register (**EXTI_RTSR**) 78
6.3.3. Falling Trigger Selection Register (**EXTI_FTSR**) 79
6.3.4. Pending Register (**EXTI_PR**) 79
6.4. Ejemplo de interrupción externa 79

7. ADC: Analog-to-Digital Converter **83**
7.1. Introducción 83
7.2. Tiempo de Conversión 85
7.2.1. Ejemplo de cálculo del tiempo de conversión 86
7.3. Cálculo de la Salida Digital 86
7.4. Error de Conversión 87
7.4.1. Ejemplo práctico de cálculo del error cuantificación 87

7.5. Conversor Analógico a Digital (ADC) del Nucleo-STM32F446RE . . 88
7.5.1. Características Principales 88
7.6. Registros del Convertidor Analógico a Digital (ADC) 89
7.6.1. Registros del ADC 89
7.7. Ejemplo de uso del ADC1 con un potenciómetro 90
7.7.1. Código completo del ejemplo 90
7.7.1.1. Inicialización de GPIO 93
7.7.1.2. Inicialización del ADC1 93
7.7.1.3. Configuración de TIM2 93
7.7.1.4. Función de atención de la Interrupción del ADC 93
7.7.1.5. Modo de Bajo Consumo (Stop Mode) 94

8. USART: Universal Synchronous and Asynchronous Receiver-Transmitter 95
8.1. Introducción a la USART 95
8.1.1. Comunicación USART 97
8.2. Configuración de la USART 99
8.2.1. Comunicación USART mediante interrupciones 100
8.3. Registros de la USART 103
8.3.1. USART Status Register (USART_SR) 103
8.3.2. USART Data Register (USART_DR) 105
8.3.3. USART Control Register 1 (USART_CR1) 106
8.3.4. USART Control Register 2 (USART_CR2) 108
8.3.5. USART Baud Rate Register (USART_BRR) 109
8.3.5.1. Ejemplos de cálculo del baud rate 110
8.4. Ejemplo de comunicación USART por interrupciones 112
8.4.1. Código de configuración de las GPIO de la USART 113
8.4.2. Código de configuración de comunicación e interrupción USART 116
8.4.3. Código de la ISR de la USART 117
8.4.4. Código para gestionar el envío y recepción de datos 119

9. Desarrollando para Nucleo-STM32 121
9.1. Programación bare metal y HAL 122
9.2. Metodología de programación usada en este libro 124
9.2.1. Estructura del entorno MatrixMCU 128
9.2.2. Estructura de proyecto 130
9.2.3. Cuándo un código es common o port 133
9.2.4. Documentación del código 133

10. Test unitarios y ejemplos de integración **135**
10.1. Test unitarios 136
10.2. Ejemplo de uso o test de integración 137
10.3. Ejecución de los test 138

Bibliografía **144**

Apéndices

A. Otras herramientas de desarrollo **145**

B. Generación de documentación de código con Doxygen **149**
B.1. Generando la API en Windows con Doxygen 149
B.2. Generando la API en Linux con Doxygen 150
B.3. Generando la API en MacOS con Doxygen 150

C. Tabla de caracteres ASCII **151**

Índice de figuras

1.1. Placa de desarrollo Arduino-UNO 4
1.2. Placas Nucleo según el número de pines de sus microcontroladores. . . 5
1.3. Familia de placas **Nucleo-STM32** 6
1.4. Características de la serie STM32F4 7
1.5. *Ejemplo de numeración de los microcontroladores STM32.* 7
1.6. Diagrama de bloques Hardware (HW) de la **Nucleo-STM32F446RE** 10
1.7. Componentes de la **Nucleo-STM32** 11
1.8. Posiciones del conector `JP5` para alimentación 11
1.9. *Programador/ depurador in-circuit ST-LINK/V2-1* 14
1.10. Conectores y zócalos de la placa **Nucleo-STM32F446RE** 15
1.11. Pinout y funciones en los header en la **Nucleo-STM32F446RE**. . . . 16
1.12. Detalle del LED LD2 en esquemático. 17

2.1. Logotipo de la empresa ARM. 20
2.2. Familias de procesadores ARM. 21

3.1. Diagrama de bloques del circuito del **STM32F446RE** 27
3.2. Diagrama de bloques y buses en detalle del **STM32F446RE**. 29
3.3. Extracto del datasheet con las tensiones que toleran algunos pines. . . . 30
3.4. Características de los modos de bajo consumo. 32
3.5. Secciones del fichero de arranque `startup_stm32f446xx.s`. 34

4.1. Entrada y salida para sensar y actuar sobre el entorno. 35
4.2. Registros GPIOx_MODER, GPIOx_PUPDR y GPIOx_AFRL. 40
4.3. Funciones alternativas en la línea/ pin 5 del port A. 43
4.4. Registros GPIOx_IDR y GPIOx_ODR. 44
4.5. Registro GPIOx_BSRR 45

5.1. Sistemas de generación del reloj del sistema. 48
5.2. Circuito del árbol de relojes del **STM32F446RE**. 50
5.3. Registros de activación de relojes de GPIO y temporizadores. 51
5.4. Características de los temporizadores del **STM32F446RE**. 53
5.5. Diagrama de bloques de un timer de propósito general. 54

5.6. Registros TIMx_CR1, TIMx_DIER y TIMx_CCMR1. 55
5.7. Regs. TIMx_CCER, TIMx_CNT, TIMx_PSC, TIMx_ARR y TIMx_EGR. 57
5.8. Registros TIMx_CCR1 y TIMx_ST. 58
5.9. Esquema de operación de un temporizador. 59
5.10. Señales PWM. 61
5.11. Señal de entrada para medida por *input capture* por flancos de subida. 65

6.1. Extracto de la tabla de vectores de interrupción del *reference manual*. . 76
6.2. Mapeo de las líneas de las GPIO a las líneas de interrupción `EXTI0-EXTI15`. 77
6.3. SYSCFG_EXTICR4 para asignar la interrupción `EXTI13` al pin `PC13`. . 77
6.4. Registros EXTI_IMR y EXTI_RTSR. 78
6.5. Registros EXTI_FTSR y EXTI_PR. 80

7.1. Diagrama de conexiones de un ADC con Sample&Hold y SAR. 84
7.2. Conexiones y flujo de programa para potenciómetro con ADC 91

8.1. Conector DB9 y conversor USB-serie. 96
8.2. Conexión *full-duplex* de la USART con un ordenador 98
8.3. Ejemplo de transmisión de una palabra de 8 bits con 1 bit de *stop*. . . . 98
8.4. Ejemplo de transmisión de 8 bits, 1 bit de *stop* a 9600 baudios. 99
8.5. Interrupciones en transmisión de la USART. 102
8.6. Diagrama de bloques de la USART en el **STM32F446RE**. 104
8.7. Registros de estado y de datos de la USART USART_SR y USART_DR. 107
8.8. Registros de control de la USART USART_CR1 y USART_CR2. 109
8.9. Registro de configuración de la velocidad de TX/ RX USART_BRR . . 109
8.10. Ejemplo de cálculo de la velocidad real de transmisión sin error. 111
8.11. GPIOs de la USART1 de la placa **Nucleo-STM32F446RE**. 114
8.12. Advertencia de uso de la USART2 en la placa. 114

9.1. Esquema de desarrollo de proyecto a implementar, para ST. 125
9.2. Adaptaciones del esquema del proyecto a otras plataformas. 126
9.3. Estructura de CMSIS en su versión 5.9.0. 127
9.4. Toolkit MatrixMCU y proyecto para compilación multiplataforma. . . . 129
9.5. Estructura de proyecto con división de directorios common y port. . . . 131
9.6. Fichero `CMakeLists.txt` de la raíz del proyecto. 132
9.7. Diagrama de flujo para saber si una función va en `common` o en `port`. . 134

10.1. Estructura de directorios de test unitarios y ejemplos en MatrixMCU. . 136
10.2. Estructura de código de un test unitario. 137
10.3. Estructura de código de un test de integración. 138
10.4. Lista de los posibles test unitarios y de integración. 139

10.5. Ejemplo de ejecución de los test unitarios de la parte `PORT`. 140
10.6. Ejemplo de ejecución de los test unitarios de la parte `COMMON`. 140
10.7. Ejecución de un ejemplo de uso o test de integración. 141

A.1. Aspecto de la interfaz de CubeMX. 147

C.1. Tabla de caracteres ASCII. 151

Lista de acrónimos

ADC Analog-Digital Converter (Conversor Analógico-Digital) 5

AHB Advanced High-performance Bus (Bus Avanzado de Alto rendimiento) 28

APB Advanced Peripheral Bus (Bus Avanzado de Periféricos) 48

API Application Programming Interface (Interfaz de Programación de Aplicaciones) 121

ARM Advanced RISC Machine 20

ASCII American Standard Code for Information Interchange, Código Estadounidense Estándar para el Intercambio de Información 106

CAN Controller Area Network 5

CISC Complex Instruction Set Computer (Computador con Conjunto de Instrucciones Complejo) 20

CMSIS Common Microcontroller Software Interface Standard 38

CPU Central Processing Unit (Unidad Central de Procesamiento) 20

DAC Digital-Analog Converter (Conversor Digital-Analógico) 26

DMA Direct Memory Access (Acceso Directo a Memoria) 75

DSP Digital Signal Processor (Procesador de Señales Digitales) 22

EXTI External Interrupt/ Event Controller (Controlador de Interrupciones Externas) 75

FPU Floating Point Unit (Unidad de Punto Flotante) 25

FW Firmware 33

GDB GNU Debugger 14

GPIO General Purpose Input/Output (Entrada/Salida de Propósito General) 26

HAL Hardware Abstraction Layer (Capa de Abstracción Hardware) 122

HSE High Speed External clock 48

HSI High Speed Internal clock 48

HW Hardware xi

IDE Integrated Development Environment (Entorno de Desarrollo Integrado) 128

I2C Inter-Integrated Circuit 20

IoT Internet of Things 20

ISA Instruction Set Architecture 20

ISR Interrupt Service Routine (Rutina de Servicio/Atención a la Interrupción) 33

JTAG Joint Test Action Group 14

GDB GNU Debugger 14

LCD Liquid-Crystal Display (Pantalla de Cristal Líquido) 26

LED Light-Emitting Diode (Diodo Emisor de Luz) 1

MCU Microcontroller Unit (Unidad de Microcontrolador) 5

MSB Most Significant Bit (Bit Más Significativo) 106

NVIC Nested Vectored Interrupt Controller (Controlador de Vector de Interrupciones Anidadas) 68

PCB Printed Circuit Board (Placa de Circuito Impreso) 3

PLL Phase Locked Loop 48

PWM Pulse Width Modulation (Modulación de Ancho de Pulso) 15

RCC Reset and Clock Controller (Controlador de Reinicio y Reloj) 48

RISC Reduced Instruction Set Computer (Computador con Conjunto de Instrucciones Reducido) 20

RTC Real Time Clock (Reloj de Tiempo Real) 32

RTOS Real Time Operating System (Sistema Operativo de Tiempo Real) ... 122

SAR Successive Approximation Register (Registro de Aproximación Sucesiva) 84

SPI Serial Peripheral Interface 20

SRAM Static Random Access Memory 32

ST STMicroelectronics 3

SW Software 28

SVD System View Description 126

SWD JTAG/serial Wire Debugging Software 10

UART Universal Asynchronous Receiver-Transmitter (Transmisor-Receptor Asíncrono Universal) 26

USART Universal Synchronous and Asynchronous Receiver-Transmitter (Transmisor-Receptor Síncrono y Asíncrono Universal) 26

XML Extensible Markup Language 130

Fundamentos teórico-prácticos

En este documento se presentan los conceptos teóricos asociados a la práctica, manejo y programación de dispositivos basados en microcontrolador. En particular, nos centraremos en la placa **Nucleo-STM32F446RE**, aunque en términos generales son aplicables a cualquier placa de desarrollo **Nucleo-STM32**.

En este documento encontrará información útil sobre (i) las placas de desarrollo (Capítulo 1), (ii) sobre la arquitectura del procesador (Capítulo 2), (iii) sobre los detalles del microcontrolador **STM32F446RE** (Capítulo 3), y (iv) sobre los entornos de desarrollo y programación (Capítulo 9). Además, complementario a este libro está el de tutoriales [1], en el se proponen una serie de ejercicios para familiarizarse con la metodología de trabajo de la con ejemplos básicos guiado para hacer parpadear un Light-Emitting Diode (Diodo Emisor de Luz) (LED) e introducción a las máquinas de estados.

Capítulo 1

Placas de desarrollo Nucleo-STM32

Josué Pagán

En este capítulo se describen las especificaciones generales de las placas **Nucleo-STM32** de la empresa STMicroelectronics (ST), y las particularidades de la placa **Nucleo-STM32F446RE** que usaremos en el laboratorio.

Las placas de desarrollo son dispositivos formados por un sistema microcontrolador, periféricos básicos de entrada-salida (no es un requisito), y conectores de acceso a pines del microcontrolador. Las placas de desarrollo tienen el objetivo de permitir a los desarrolladores el prototipado rápido para probar y testar su código antes de realizar un diseño final de una Printed Circuit Board (Placa de Circuito Impreso) (PCB) y llevar el sistema a producción para su despliegue.

Generalmente, las placas de desarrollo son económicas y los fabricantes de microcontroladores suelen ser los mismos que las diseñan y distribuyen códigos e interfaces de desarrollo para popularizarlas entre los desarrolladores y hacerse valer frente a la competencia. Un ejemplo es la placa **Nucleo-STM32F446RE** que usaremos en el laboratorio; desarrollada por ST, integrando el microcontrolador **STM32F446RE** de la misma compañía.

En otras ocasiones, las placas de desarrollo tienen un objetivo académico o lúdico. Así pues, podemos encontrar placas de desarrollo que no están diseñadas por el fabricante del microcontrolador que porta. El ejemplo quizás más popular de estas placas de desarrollo es cualquiera de las placas Arduino, como la Arduino-UNO (Figura 1.1) de la empresa Arduino, y que porta la el microcontrolador ATMega328P de la empresa ATMEL (ahora de Microchip). Arduino se ha popularizado tanto que la huella del *pinout*[1] de sus conectores se ha convertido en un estándar *de facto*. Tanto es así que la placa **Nucleo-STM32F446RE** que usaremos tiene este *pinout*, lo que permite usar *shields* que cumplan con este formato.

[1]El *pinout* es el "patillaje" o disposición de los pines de conexión.

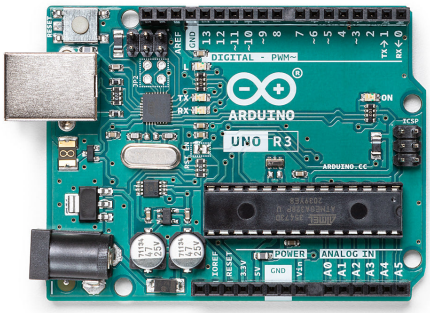


Figura 1.1: *Placa de desarrollo Arduino-UNO con el microcontrolador ATMega328P de la empresa ATMEL (ahora de Microchip).*

1.1. Placas de desarrollo Nucleo-STM32

La familia de placas de desarrollo **Nucleo-STM32** son placas de bajo coste con potentes microcontroladores basados en la arquitectura ARM-Cortex de 32 bits. A fecha de redacción de este documento existen 64 placas Nucleo que se agrupan en 3 tamaños según el número de pines del microcontrolador que llevan: 32, 64 o 144 pines. En la Figura 1.2 se muestran los 3 tipos de placas **Nucleo-STM32**. Estas, a su vez, se dividen en otros 4 subgrupos como se muestra en la Figura 1.3[2]: (i) ***mainstream***, (ii) ***ultra-bajo consumo***, (iii) ***alto rendimiento***, (iv) e ***inalámbricas***. Veamos algunas características de estas agrupaciones[3]:

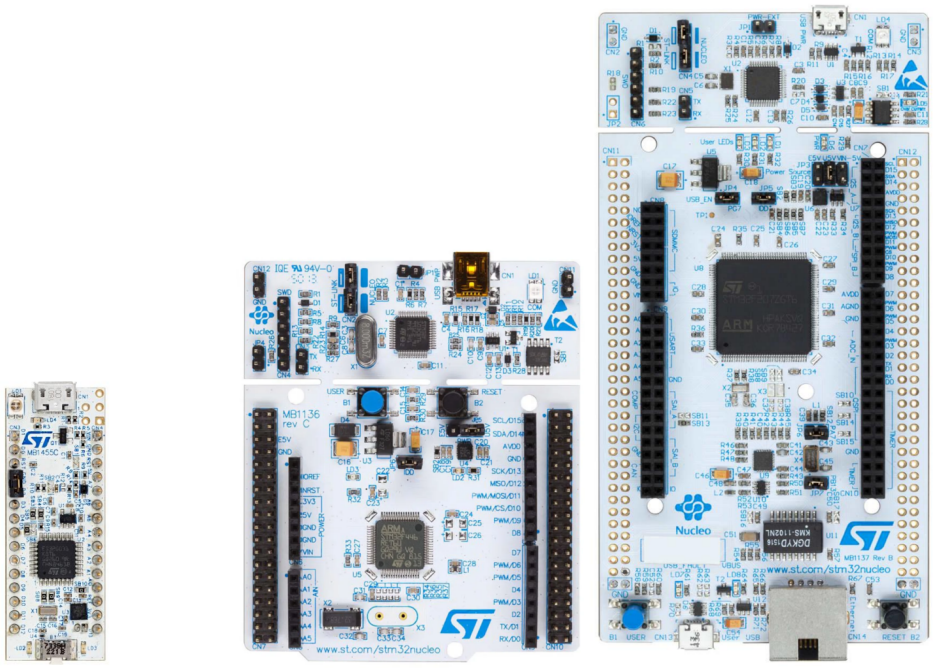
- Nucleo-X: las placas Nucleo-32 son placas pequeñas compatibles con el *pinout* de Arduino Nano. Las placas Nucleo-64 y Nucleo-144 son compatibles con los *pinouts* de Arduino UNO y las conexiones de las placas Morpho desarrolladas por la propia ST.

- ***mainstream***: orientado para un amplio abanico de aplicaciones que responden a las necesidades de gran parte de los productos del mercado de consumo. Permite aplicaciones en tiempo real, como el control de motores,y son asequibles, robustas y con larga vida útil. Las podemos encontrar en 5 series de microcontroladores: STM32F0, STM32G0, STM32F1, STM32F3, y STM32G4.

- ***ultra-bajo consumo***: ofrecen a los desarrolladores de sistemas integrados de bajo consumo un equilibrio entre rendimiento, potencia, seguridad y rentabilidad. Tienen un consumo energético extraordinariamente bajo a altas temperaturas, y algunas de las series llegan a necesitar corrientes de solo 170 nA en modo de bajo

[2]Placas Nucleo: `https://www.st.com/en/evaluation-tools/stm32-nucleo-boards.html#overview`. Accedido: 10-12-2022.

[3]Descripción de placas Nucleo: `https://www.st.com/en/microcontrollers-microprocessors.html`. Accedido: 10-12-2022.



(a) Nucleo-STM32 32 pines *(b)* Nucleo-STM32 64 pines *(c)* Nucleo-STM32 144 pines

Figura 1.2: Placas **Nucleo-STM32** según el número de pines de sus microcontroladores.

consumo, con tiempos de activación bajos como 3,5 μs. Las podemos encontrar en 6 series: STM32L0, STM32L1, STM32L4, STM32L4+, STM32L5, y STM32U5.

- ***alto rendimiento***: son Microcontroller Unit (Unidad de Microcontrolador) (MCU) que optimizan la ejecución de código, las transferencias y el procesamiento de datos. Las podemos encontrar en 4 series: STM32F7, STM32H7, **STM32F4** (**la que usaremos en este libro**), y STM32F2.

  La Figura 1.4[4] muestra una tabla resumen de las características de los microcontroladores de la serie **STM32F4**, resaltando el STM32F446 que usaremos en el libro. Algunas de estas MCU tienen conexión dedicada para uso de cámaras digitales[5], bus de comunicaciones Controller Area Network (CAN) (muy usado en automoción), cuentan con Analog-Digital Converter (Conversor Analógico-Digital) (ADC) de 12 bits, trabajan en un amplio rango de temperaturas, pueden llegar a trabajar con frecuencias de reloj de sistema de

[4]Características de los microcontroladores de la serie **STM32F4**: `https://www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32f4-series.html`. Accedido: 10-12-2022.

[5]Arducam DCMI-Compatible Camera Modules for STM32: `https://www.arducam.com/stm32-cameras-that-support-dcmi/`. Accedido: 10-12-2022.

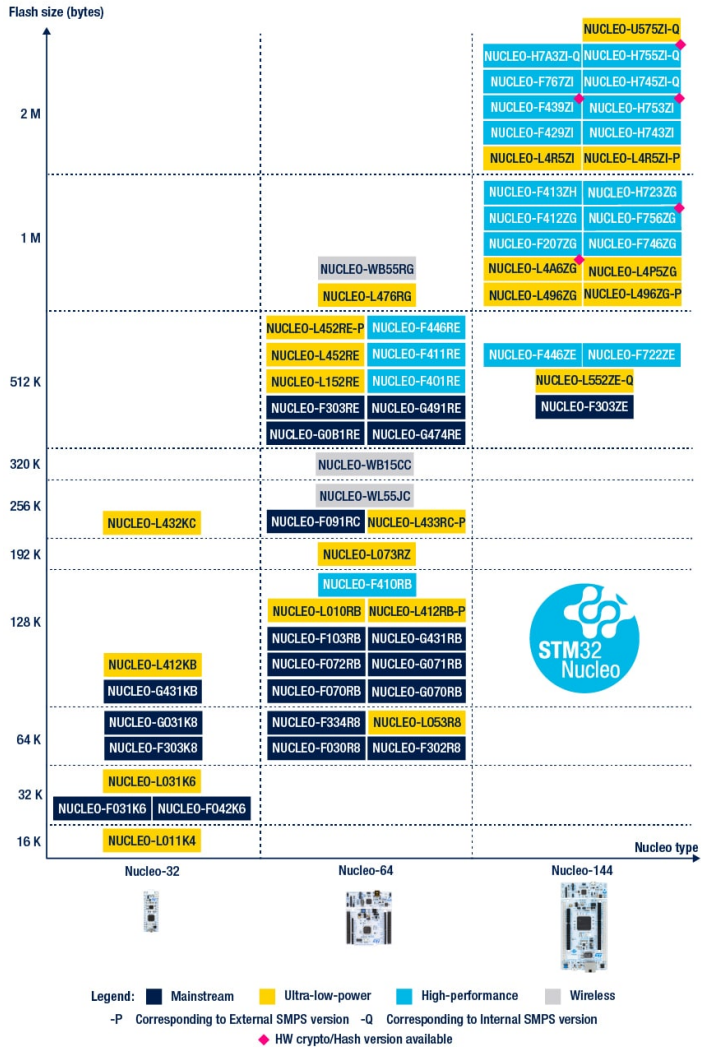
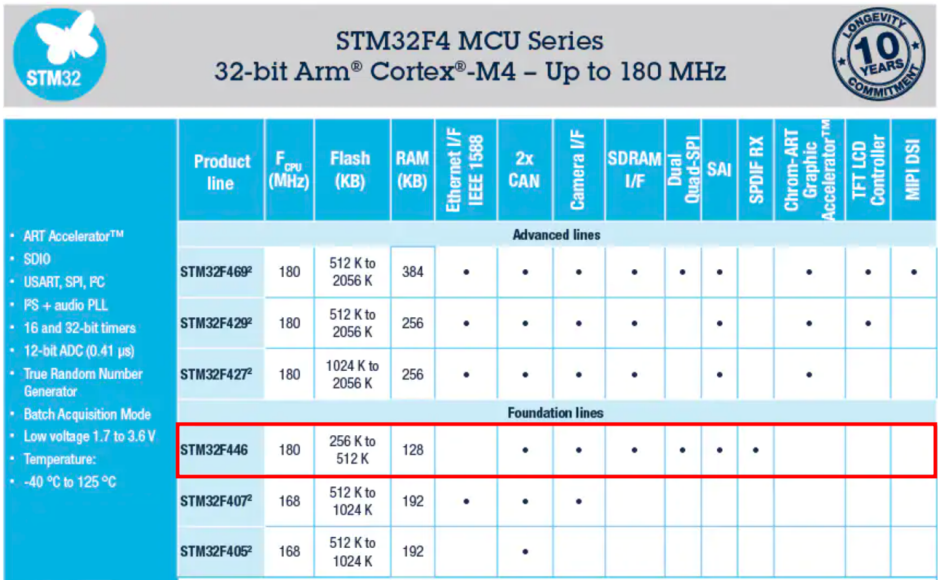


Figura 1.3: Familia de placas **Nucleo-STM32** según su *pinout* y memoria *flash*.

180 MHz, *etc.*

- ***inalámbricas***: las MCU STM32Wx tienen conectividad inalámbrica en el rango de frecuencias sub-GHz y de 2,4GHz, e implementan múltiples protocolos de comunicación con bajo consumo de corriente y funciones de seguridad integradas. Las MCU STM32Wx tienen una arquitectura de doble núcleo con una MCU y un transceptor radio formando un *System-on-Chip*. Existen 2 series según los protocolos: la serie STM32WB con estándares Bluetooth, Zigbee, Thread y Matter; y la serie STM32WL que implementa los estándares LoRa, Sigfox, M-Bus y Mioty.

Gran parte de las características de un microcontrolador están definidas en su numeración. La numeración de los microcontroladores STM32 se lee de la siguiente manera: "$STM32\ x\ yyy\ z\ p\ q$", como se muestra en la Figura 1.5. Tomando de



STM32F4 MCU Series
32-bit Arm® Cortex®-M4 – Up to 180 MHz

- ART Accelerator™
- SDIO
- USART, SPI, I²C
- I²S + audio PLL
- 16 and 32-bit timers
- 12-bit ADC (0.41 µs)
- True Random Number Generator
- Batch Acquisition Mode
- Low voltage 1.7 to 3.6 V
- Temperature:
- -40 °C to 125 °C

Product line	F_{CPU} (MHz)	Flash (KB)	RAM (KB)	Ethernet I/F IEEE 1588	2x CAN	Camera I/F	SDRAM I/F	Dual Quad-SPI	SAI	SPDIF RX	Chrom-ART Graphic Accelerator™	TFT LCD Controller	MIPI DSI
Advanced lines													
STM32F469²	180	512 K to 2056 K	384	•	•	•	•	•	•		•	•	•
STM32F429²	180	512 K to 2056 K	256	•	•	•	•		•		•	•	
STM32F427²	180	1024 K to 2056 K	256	•	•	•	•		•		•		
Foundation lines													
STM32F446	180	256 K to 512 K	128		•	•	•	•	•	•			
STM32F407²	168	512 K to 1024 K	192	•	•	•							
STM32F405²	168	512 K to 1024 K	192		•								

Figura 1.4: *Características de algunos de los microcontroladores de la serie **STM32F4**. Resaltado el STM32F446 que usaremos.*

ejemplo el que usaremos en el libro —mostrado en la Figura 1.5—, podemos decir que el microcontrolador **STM32F446RE**:

- **STM32**: tiene una arquitectura de 32 bits.
- **F**: es de una serie de propósito general.
- **446**: su arquitectura es ARM-Cortex M4.
- **R**: tiene 64 pines.
- **E**: tiene una memoria *flash* de 512 *Kbytes*.

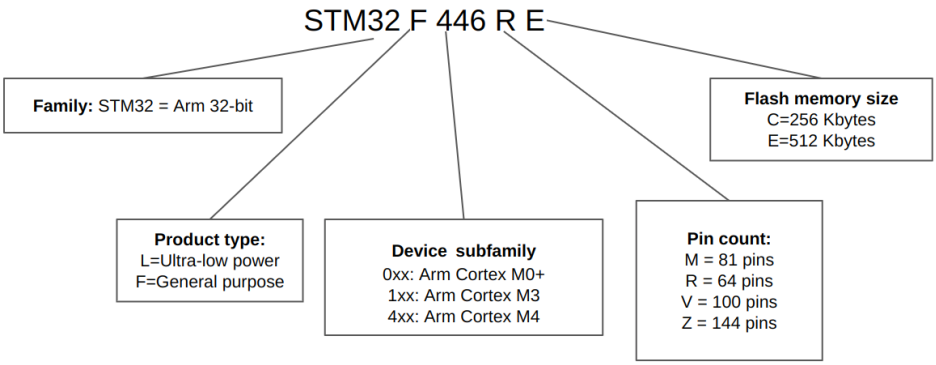


Figura 1.5: *Ejemplo de numeración de los microcontroladores STM32.*

1.1.1. Por qué Nucleo-STM32

Se preguntará el lector por qué se ha elegido una placa **Nucleo-STM32** porque, como bien sabrá, existen muchas plataformas de desarrollo de bajo coste de numerosos fabricantes, y muchos microcontroladores diferentes. Por supuesto, existe un motivo académico, pero también estratégico de cara al futuro del alumnado, y de las asignaturas que hacen uso de este libro. A continuación se explicarán las decisiones de los motivos que le permitan comprender el contexto en el que se encuentra y, quizás, en un futuro, tomar alguna decisión fundamentada.

Hoy en día, es muy probable que un estudiante de ingeniería haya tenido en sus manos, haya programado, o haya estudiado en alguna asignatura una placa Arduino. Eso no era común en la década de los *10's*, pero su accesibilidad económica y la integración en algunas asignaturas o actividades de los itinerarios tecnológicos de bachillerato, hace que actualmente un porcentaje considerable de alumnos tengan un placa similar en casa, y ya estén familiarizados. Ahora hay que entender cómo funcionan.

Las placas **Nucleo-STM32** son económicas, lo que permite que el alumno pueda trabajar en casa con su propia placa. Estas plataformas son versátiles gracias a la gran cantidad de placas con microcontroladores de la misma familia que distribuye ST, y su compatibilidad con el *pinout* de Arduino. Se basan en arquitecturas ARM, que inundan el mercado de la electrónica —y de las que hablaremos más adelante—, y son transversales a diversas asignaturas de Electrónica en diferentes planes de estudio de cualquier Escuela o Facultad como las relacionadas con Sistemas Digitales, Sistemas Electrónicos de Control, Sistemas de Adquisición de Datos, Sistemas Electrónicos para IoT, *etc.*

Se asume que ha estudiado y tiene los conceptos básicos de análisis y del diseño de la arquitectura de un microcontrolador. En este libro se va a trabajar con los conceptos previos adquiridos para implementar un sistema completo. La forma en la que se ha planteado el desarrollo de de los proyectos a desarrollar con este libro, permitirá que, en caso de problemas de abastecimiento con esta plataforma, se pueda cambiar rápidamente a otra. ¿Qué quiere decir esto último? Que el lector, utilizando una plataforma particular **Nucleo-STM32** y la metodología a seguir con este libro, será capaz de generalizar el caso y replicar el entorno en otro contexto para proyectos propios o ajenos: entretenimiento, estudios, entorno laboral...

Es difícil elegir una plataforma, un microcontrolador, un fabricante, una arquitectura..., incluso para gente con experiencia. Aunque inevitablemente estemos dejando algún criterio fuera de nuestro alcance, creemos que la elección de una plataforma **Nucleo-STM32** es buena y coherente para nuestros alumnos.

En la siguiente sección entraremos en los detalles de la placa **Nucleo-STM32F446RE** con la que trabajaremos.

1.2. Placa de desarrollo Nucleo-STM32F446RE

La Figura 1.2b muestra la placa **Nucleo-STM32F446RE** que usaremos en el laboratorio. Es una placa de *alto rendimiento* que tiene una MCU de 64 pines. Esta placa es compatible con el *pinout* de Arduino-UNO (Figura 1.1), por lo que se le pueden acoplar multitud de *shields* con este formato[6], además de las desarrolladas por ST con el *pinout* Morpho, o Arduino[7]. La placa **Nucleo-STM32F446RE** tiene las siguientes características:

- 1 LED de comunicación (**rojo**/**naranja**/**verde**) (`LD1` en la Figura 1.7)
- 1 LED de usuario (**verde**) (`LD2`)
- 1 LED de alimentación (**rojo**) (`LD3`)
- 1 botón/pulsador de usuario (**azul**) (`B1 USER`)
- 1 botón/pulsador de *reset* (**negro**) (`B2 RESET`)
- 1 cristal oscilador a 32768 kHz (`XC2 32 KHz`)
- 1 conector Morpho (`CN7` y `CN10`)
- 1 zócalo de expansión Arduino-UNO (`CN5`, `CN6`, `CN8` y `CN9`)
- Fuentes de alimentación: el USB del ST-LINK (V_{BUS}) (`CN1`), o fuentes externas
- Un programador/depurador ST-LINK/V2-1 con un puerto de comunicaciones (COM) y un puerto de depuración (`ST-LINK part` en la Figura 1.6)

La placa **Nucleo-STM32F446RE**—al igual que muchas otras de su familia— es muy popular. Podemos encontrar mucha información sobre las placas **Nucleo-STM32** 64 pines en el **manual de usuario: User Manual UM1724, STM32 Nucleo-64 boards** [2]. En este documento podemos ver que se trata de placas con 2 partes bien diferenciadas como se muestra en el diagrama de bloques de la Figura 1.6: (i) la parte más pequeña `ST-LINK part` con el puerto minu-USB, y (ii) la parte con la MCU. La parte del depurador/programador ST-LINK es muy común en placas de desarrollo como la nuestra, pero no en las placas desplegadas en producción. Esta parte contiene, además del depurador/programador ST-LINK, la

[6] 20 placas soportadas por la **Nucleo-STM32F446RE**: `https://os.mbed.com/blog/entry/20-shields-on-10-st-nucleo-platforms/`. Accedido: 18-12-2022.

[7] Placas de expansión desarrollada por ST: `https://os.mbed.com/teams/ST/wiki/Matrix-of-tested-boards`. Accedido: 18-12-2022.

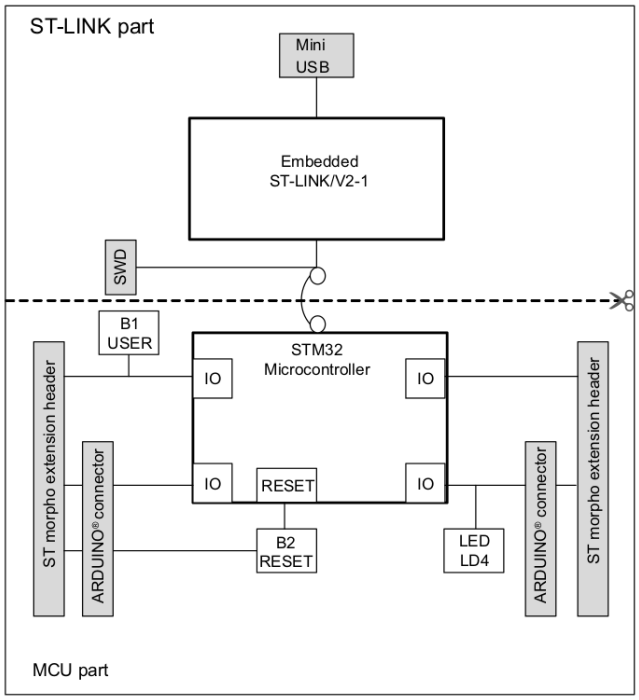


Figura 1.6: Diagrama de bloques HW de la **Nucleo-STM32F446RE**

alimentación de la placa. Esta parte podría recortarse, por ejemplo para reducir espacio, en cuyo caso necesitaríamos alimentar el microcontrolador mediante `VIN`, `E5V` y `3.3V` a través de los conectores `CN7` o `CN6` marcados también en la Figura 1.7. En tal caso, para poder seguir programando el microcontrolador sí necesitaremos conectar un programador `ST-LINK` con cables JTAG/serial Wire Debugging Software (SWD) al conector `CN7`.

1.2.1. Alimentación de la placa

La forma más habitual de alimentar la placa será a través del conector mini-USB (`CN1`). Es el mismo conector que utilizamos para programar y depurar, por lo que **es muy importante que el cable USB que utilicemos tenga los cables de datos, y no solo alimentación**. También podremos utilizar una fuente de alimentación externa (de $7 - 12\ V$) por V_{IN}, $5\ V$ por `E5V`, o $+3{,}3\ V$.

Alimentación por USB

Cuando se alimente la placa a través del conector mini-USB (`CN1`), se debe conectar un *jumper* entre los pines 1 y 2 del conector `JP5`; esta es la situación por defecto cuando compramos la placa y que, generalmente, no modificamos.

El conector `JP1` de la parte ST-LINK se utiliza para configurar el máximo de corriente permitida por la MCU. Por defecto **no hay ningún *jumper* conectado,**

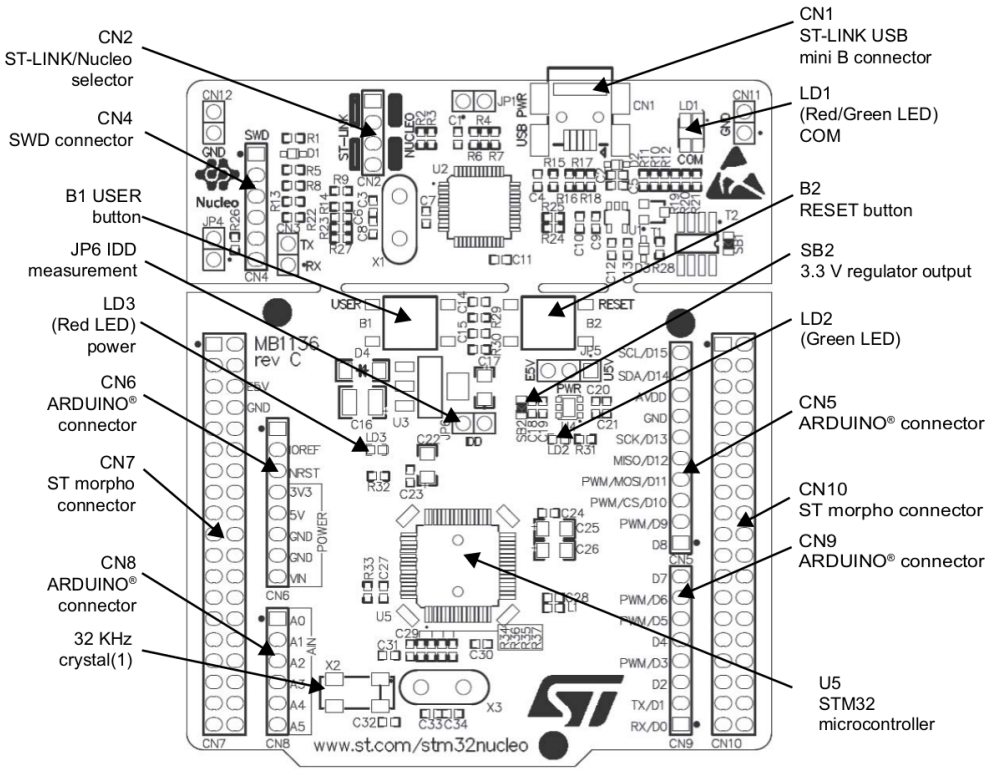


Figura 1.7: Componentes de la Nucleo-STM32



(a) Jumper para alimentación por mini-USB *(b) Jumper* para alimentación por V_{IN} o E5V

Figura 1.8: Posiciones del *jumper* JP5 para elegir alimentación.

lo que permite una corriente máxima de 300 mA. Si se conecta un *jumper*, la corriente máxima es de 100 mA. Si el consumo de corriente sobrepasa los 300 mA, se debe alimentar la placa a través de V_{IN} o E5V.

Alimentación por fuentes externas

Se puede usar una fuente de alimentación de externa por las entradas V_{IN} o E5V configurando los *jumpers* de los conectores JP1 y JP5. Para ello: (i) se coloca el conector JP5 como se muestra en la Figura 1.8b, y (ii) se quita el *jumper* de JP1, después, se puede alimentar:

- mediante V_{IN} a través del pin 8 del zócalo CN6, o a través del pin 24 del conector CN7. El rango de tensiones permitido es de $7-12$ V, con un consumo máximo de 800 mA a 7 V, 450 mA entre $7-9$ V y un máximo de 250 mA entre $9-12$ V.

- mediante `E5V` a través del pin 6 del conector `CN7`. El rango de tensiones permitido es muy limitado, entre $4{,}75 - 5{,}25\ V$, con un consumo máximo de 500 mA.

Es posible que tengamos que usar fuentes externas porque tenemos requisitos de consumo mayores en nuestra aplicación. Esto no quiere decir que no se pueda usar la conexión USB para comunicación, programar o depurar. En esta situación:

1. Se coloca el conector `JP5` como se muestra en la Figura 1.8b.
2. Se quita el *jumper* de `JP1`.
3. Se alimenta la placa mediante V_{IN} o `E5V`
4. Comprobamos que se enciende **rojo** el LED `LD3`.
5. Por último, se conecta el cable USB al PC.

Alimentación externa de 3,3 V

Por último, mencionar que también se puede alimentar la placa usando una fuente externa de 3,3 V a través del pin 4 del zócalo `CN6`. El rango de tensiones permitido es de $3{,}0 - 3{,}6\ V$. No obstante, con esta configuración la parte de la placa ST-LINK no funciona, por lo que no se puede programar ni depurar.

ATENCIÓN

Cuando se alimenta la placa por mini-USB, V_{IN} o `E5V`, **se puede obtener una tensión de** 5 V **en el pin 5 del zócalo** `CN6`**, o en el pin 18 del conector** `CN7`. Típicamente **usaremos esta salida de tensión para alimentar los componentes** de las *shields* Arduino que se conecten, o de componentes que montemos en una *protoboard* externa.

ATENCIÓN

Tenga en cuenta que la placa se apoya sobre los pines de los conectores `CN7` y `CN10`. Estos pines son metálicos y, como se ha mencionado, algunos llevan la alimentación. **Tenga la precaución de no colocar la placa sobre una superficie metálica cuando esté conectada, o podría provocar un cortocircuito.** Si lo considera, podría imprimirse una caja 3D de las muchas que hay disponibles online[a].

[a] Cajas 3D para Nucleo-STM32: `https://www.yeggi.com/q/stm32+nucleo+board+case/`. Accedido: 18-12-2022.

1.2.2. Luces LED de la placa

LD1 es un LED tricolor (**rojo**/**naranja**/**verde**) que nos informa del estado de comunicación del ST-LINK. El color por defecto de LD1 es **rojo**, y se enciende **verde** cuando hay comunicación entre el PC y el ST-LINK/V2-1. Los diferentes estados del LD1 son:

- **Parpadeo lento rojo**: cuando se enciende, antes de la inicialización del USB.
- **Parpadeo rápido rojo**: tras la primera comunicación correcta en el PC y el ST-LINK/V2-1.
- **Rojo fijo**: cuando se completa la inicialización entre el PC y el ST-LINK/V2-1.
- **Verde fijo**: cuando se completa la comunicación de inicialización entre el PC y el ST-LINK/V2-1 correctamente.
- **Parpadeo rojo/verde**: durante la comunicación con la placa.
- **Verde fijo**: cuando la comunicación ha terminado correctamente.
- **Naranja fijo**: cuando hay un fallo en la comunicación.

LD2 es un LED **verde** que controla el usuario. En la placa **Nucleo-STM32F446RE** LD2 está conectado al pin 6 del zócalo CN5 que corresponde con el PA5 (puerto A, línea/pin 5) del microcontrolador, y el pin digital Arduino D13. Poner un 1 lógico en este pin enciende LD2, y un 0 lógico, lo apaga.

LD3 es un LED **rojo** que se enciende para avisar que la placa se está alimentando con 5 V.

1.2.3. Botones de la placa

Hay 2 botones/pulsadores en la placa. El botón **B2** es el botón de *reset* de la placa, y cuando se pulsa durante la ejecución del programa, se reinicia la ejecución del mismo.

El botón **B1** es un botón cuya entrada puede controlar el usuario. El botón está conectado al pin 23 del conector CN7 que corresponde con el PC13 (puerto C, línea/pin 13) del microcontrolador, y no tiene correspondencia con el conector Arduino.

1.2.4. Medida de consumo del microcontrolador

El conector `JP6` —etiquetado como `IDD`— puede usarse para medir el consumo de corriente del microcontrolador quitando el *jumper* de este conector y conectando un amperímetro entre sus pines. Por defecto el *jumper* está conectado cerrando el circuito y permitiendo la alimentación del microcontrolador; de otro modo, si no se conecta un amperímetro, el microcontrolador no está alimentado.

1.2.5. ST-LINK/V2-1

El programador/ depurador in-circuit ST-LINK/V2-1 es la segunda versión de un programador/ depurador de la empresa ST (ver Figura 1.9). El ST-LINK utiliza el estándar o interfaz de comunicación del Joint Test Action Group (JTAG) (o su adaptación para ARM SWD). En nuestra placa **Nucleo-STM32** 64 pines no necesitamos el dispositivo de la Figura 1.9, pues ya está integrado en la placa, que es el bloque `ST-LINK part` de la Figura 1.6. Para que nuestro PC pueda comunicarse con el ST-LINK/V2-1 necesitamos instalar un *driver*, y para que podamos depurar necesitamos un programa que se comunique entre el depurador SW GNU Debugger (GDB) y el depurador HW que es el ST-LINK/V2-1; dicho programa será en nuestro caso `OpenOCD`.



Figura 1.9: *Programador/ depurador in-circuit ST-LINK/V2-1*

1.2.6. Conectores y zócalos de la placa

La Figura 1.10 muestra los conectores y zócalos para entrada/ salida de la placa **Nucleo-STM32F446RE**. Cada placa **Nucleo-STM32** podrá tener una disposición diferente de pines, y conviene revisar el manual de usuario para cada. Los conectores

`CN7` y `CN10` tienen el formato *Morpho* de ST, y los zócalos `CN5`, `CN6`, `CN8` y `CN9`, son los de expansión compatibles con Arduino-UNO[8].

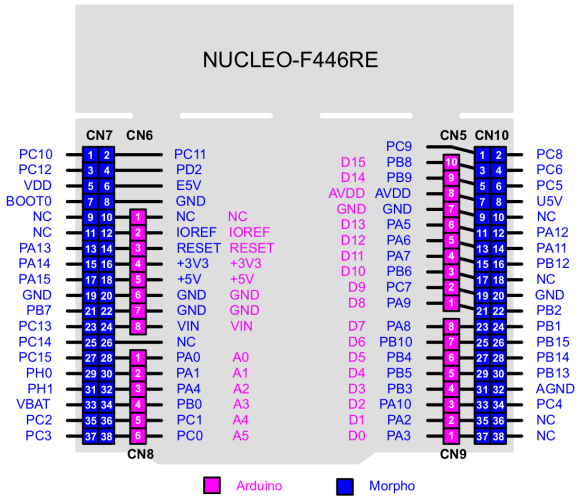


Figura 1.10: Conectores y zócalos de la placa **Nucleo-STM32F446RE**

La Figura 1.11[9] muestra el *pinout* y las funciones de cada uno de lo pines los conectores compatibles con Arduino-UNO y Morpho.

¿Cómo interpretar estas imágenes? Pongamos un ejemplo. Antes hemos mencionado que el LED `LD2` "*está conectado al pin 6 del zócalo* ***`CN5`*** *que corresponde con el* ***`PA5`*** *(puerto A, línea/pin 5) del microcontrolador, y el pin digital Arduino* ***`D13`***". **¿Cómo sabemos esta información? Fijándonos en el esquemático del circuito de la placa que nos da el fabricante**[10]. La Figura 1.12 muestra un detalle del esquemático donde se ve, a la izquierda, el `LD2` que conecta con el *nodo* `D13`. En la misma figura, a la derecha, vemos otra parte del esquemático donde se muestra parte del microcontrolador **STM32F446RE**, y vemos que el nodo `D13` está conectado al pin 21 del chip, que se corresponde con el puerto A-línea 5, `PA5` o `PORTA5`—lo podemos decir de muchas maneras—. En el conector Arduino `CN5`, vemos que está conectado al pin 6, y también al pin 11 del conector Morpho `CN10`.

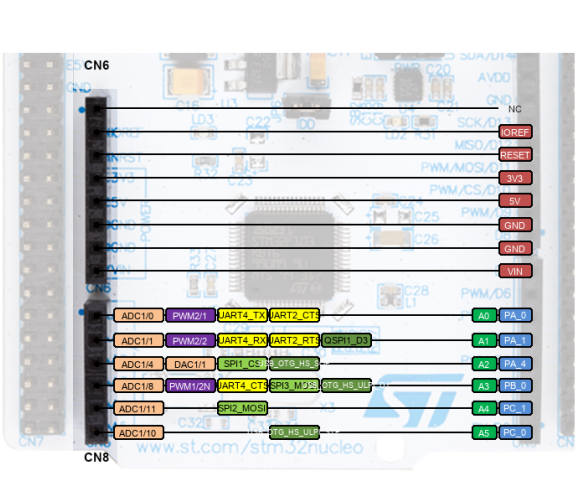
Con esta información, si vamos a la Figura 1.11b, podemos ver que este pin: 

, está conectado al `PA5`, que es el `D13` de Arduino, que puede ser un pin analógico (`ADC1/5`), que se puede generar una señal Pulse Width Modulation (Modulación de Ancho de Pulso) (PWM) asociada al *timer*

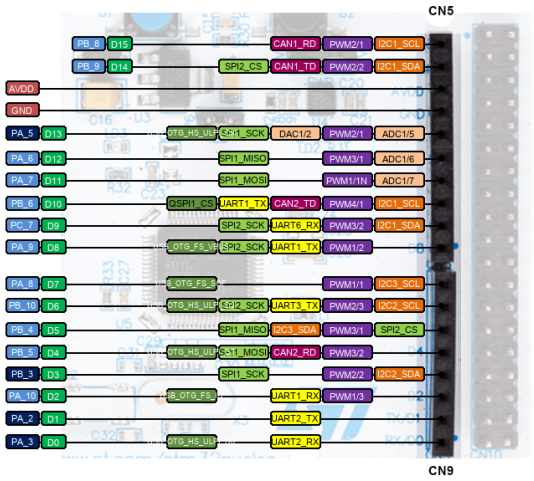
[8] Típicamente se le llama zócalo (*socket*, en inglés) cuando es un conector hembra, y "conector", a secas, cuando es macho. No obstante puede ser que les denominemos indistintamente conectores o *headers*.

[9] *Pinout* y funciones de los conectores de la placa **Nucleo-STM32F446RE**: `https://os.mbed.com/platforms/ST-Nucleo-F446RE/`. Accedido: 20-12-2022.

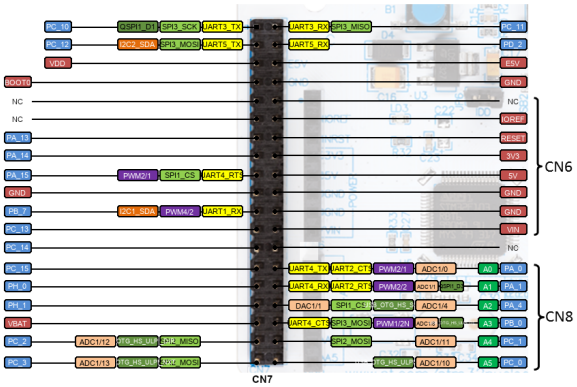
[10] Esquemáticos del circuito de la placa **Nucleo-STM32F446RE**: `https://www.st.com/en/evaluation-tools/nucleo-f446re.html#cad-resources`. Accedido: 20-12-2022.



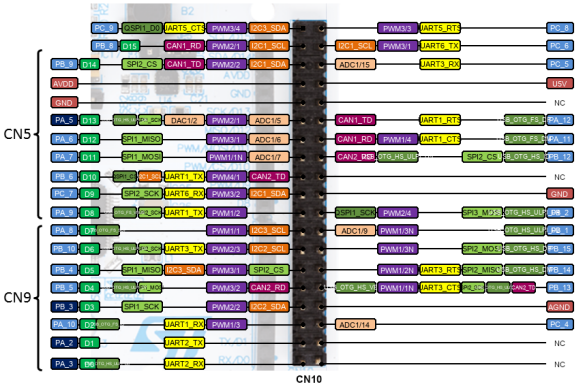
(a) Pinout y funciones *header*-Arduino izquierdo.



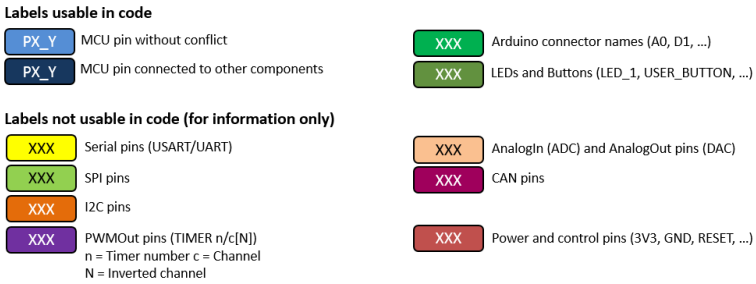
(b) Pinout y funciones *header*-Arduino derecho.



(c) Pinout y funciones *header*-Morpho izquierdo.



(d) Pinout y funciones *header*-Morpho derecho.



(e) Leyenda de funciones de los pines.

Figura 1.11: *Pinout* y funciones de los pines de los *header* en la **Nucleo-STM32F446RE**.

2, canal 1 (`PWM2/1`).

Indudablemente, siempre que vayamos a trabajar con una placa, no importa el fabricante, conviene tener a mano: (i) el esquemático de la placa, (ii) el *datasheet*, y (iii) el manual de usuario del microcontrolador. En este capítulo hemos aprendido a diferenciar y comprender los tipos de placas **Nucleo-STM32**, y hemos visto las características de las Nucleo de 64 pines. También hemos visto, someramente, cómo

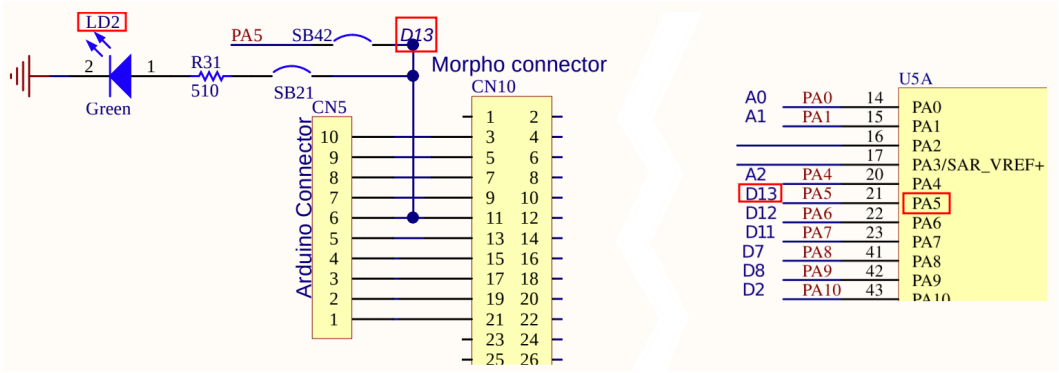


Figura 1.12: Detalle del LED LD2 en esquemático.

identificar dónde está un pin en la placa, y las funciones que podemos esperar de este. A continuación, veremos los detalles del microcontrolador de nuestra placa.

Capítulo 2

Arquitectura del procesador ARM

Josué Pagán

En este capítulo nos vamos a centrar en la arquitectura de los microcontroladores de las placas **Nucleo-STM32**. Es importante porque, inevitablemente, en el trabajo diario con un microcontrolador deberemos pararnos en aspectos de su arquitectura, y que impactan en la programación de nuestro sistema. Cuestiones como el acceso a registros, temporizadores, o modos de bajo consumo, están íntimamente relacionados con la arquitectura del microcontrolador.

2.1. Procesadores ARM

La elección de un microcontrolador es una tarea difícil que implica varias decisiones y que está condicionada por los requisitos del problema. Podemos fijarnos en:

- El coste
- La velocidad del procesador
- El consumo energético
- El tamaño (y número de pines)
- El número de pines de entrada/ salida analógicos y digitales
- La corriente máxima que puede proporcionar o recibir un puerto digital
- La resolución (número de bits) y precisión de los puertos analógicos
- Tamaño de la memoria de datos y de programa
- Número de temporizadores
- Si da soporte, por ejemplo a PWM

- Si maneja buses especiales: USB, CAN, protocolos Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), ...
- El rango de tensiones de alimentación

Por ejemplo, para sistemas empotrados (o embebidos) que vayan a estar alimentados por batería, es interesante que el consumo sea bajo, y la velocidad de procesamiento alta también, aunque aumentar esta última implica aumentar el consumo, por lo que hay que buscar un equilibro. Ya vimos un resumen de las capacidades de los microcontroladores de las placas **Nucleo-STM32** en la Figura 1.3.

Los microcontroladores de las placas **Nucleo-STM32** tienen arquitectura Advanced RISC Machine (ARM). ARM es el nombre de una empresa que quizás no le suene, pero casi con total seguridad ha tenido en sus manos algún dispositivo que use su tecnología. ARM (Figura 2.1) es una empresa multinacional con sede en Reino Unido cuyo modelo de negocio se basa en vender licencias del diseño de chips a otras empresas que los fabrican. ARM no fabrica. Por cada chip que un fabricante use con una arquitectura con diseño ARM, esta recibe regalías (*royalties*). ARM diseña el procesador, *i.e.*, la Central Processing Unit (Unidad Central de Procesamiento) (CPU), y cada fabricante—como ST—le añade sus propios periféricos para formar un microcontrolador completo: puertos de entrada/ salida, USB, comunicaciones serie, inalámbricas, ...



Figura 2.1: Logotipo de la empresa ARM.

Existe una amplia gama de empresas que los utilizan en muchos tipos diferentes de dispositivos (móviles, automoción, ordenadores, dispositivos Internet of Things (IoT), *etc.*) como Apple, Nvidia, NXP, Samsung, Qualcomm, Renesas, Texas Instruments, ST, Microchip, Infineon, y muchas más. ARM se caracterizó por diseñar arquitecturas Reduced Instruction Set Computer (Computador con Conjunto de Instrucciones Reducido) (RISC) de 32 bits en las últimas 2 décadas, aunque recientemente también ha empezado a diseñar arquitecturas de 64 bits (*e.g.*, el procesador M1 de Apple). Las arquitecturas RISC tienen un menor número de instrucciones y están más optimizadas en número de transistores y velocidad que las arquitecturas Complex Instruction Set Computer (Computador con Conjunto de Instrucciones Complejo) (CISC). El conjunto de instrucciones de ensamblador (Instruction Set Architecture (ISA)) de ARM se llama *ARM-Thumb* (es distinto al estudiado en **SDG1!** (**SDG1!**))[1].

[1]Conjunto de instrucciones ARM-Thumb: `https://developer.arm.com/documentation/ddi0210/c/Introduction/Instruction-set-summary/Thumb-instruction-summary?lang=en`. Accedido: 21-12-2022.

Existen varias familias de procesadores ARM (ver Figura 2.2) [3]. Los más antiguos ARM1 y ARM2 fueron mejorados a lo largo de los años hasta los ARM11, pero a partir de la década de los *00's*, empezaron a denominarse Cortex. Hay varios modelos Cortex de microcontroladores ARM disponibles en arquitecturas de 32 bits, y dependiendo del tipo de aplicación se les incluye una letra en su nombre:

- **Cortex-A (Application)**: son utilizados en aplicaciones que requieren de un sistema operativo como teléfonos móviles, tablets, o televisores. Son las arquitecturas que llevan algunas Raspberry Pi, o las placas BeagleBone.

- **Cortex-R (Real Time)**: son utilizados cuando el problema requiere de tiempo real como en equipos de redes de comunicaciones (*switches, routers*), o en automoción.

- **Cortex-M (Microcontroller)**: son utilizados en entornos embebidos como electrodomésticos, o dispositivos IoT. Son aplicaciones con menos requerimientos y más sencillas, de bajo coste y bajo consumo.

- **Cortex-X (Custom)**: los encontraremos en equipos de altas prestaciones como PC, o servidores.

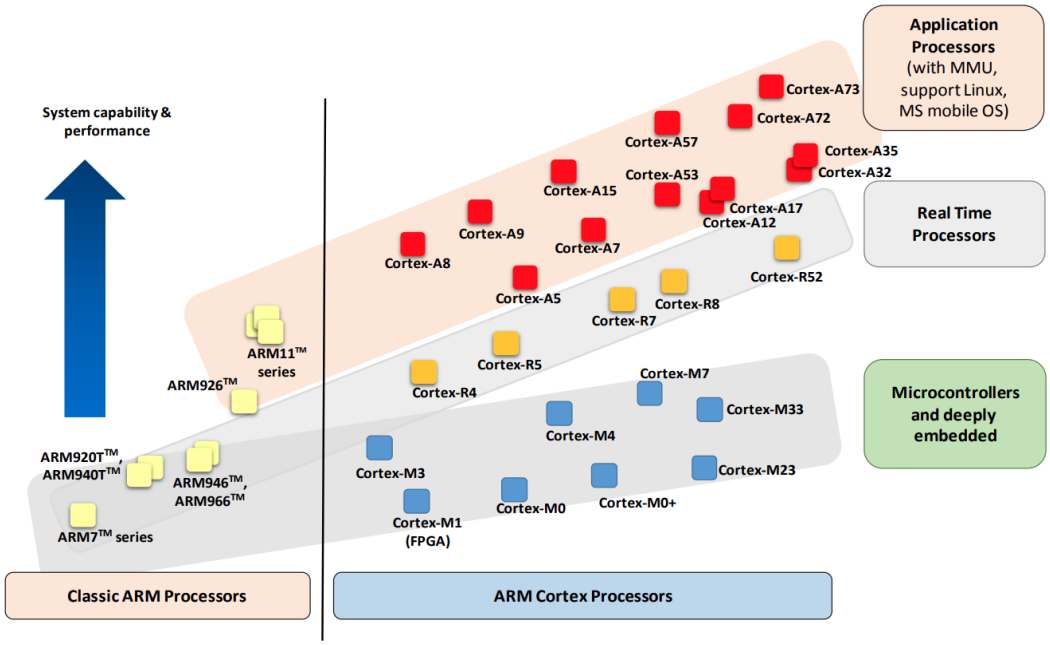


Figura 2.2: Familias de procesadores ARM.

2.1.1. Familia ARM Cortex-M

La familia de procesadores Cortex-M, a pesar de tener menos prestaciones, tienen características que permiten categorizarlos como microcontroladores bastante potentes comparados con otros del mercado. Por ejemplo, los procesadores Cortex-M4 y Cortex-M7 pueden trabajar con frecuencias de reloj máximas de hasta 400 MHz [3], incluyen unidad de punto flotante, y Digital Signal Processor (Procesador de Señales Digitales) (DSP). Por otro lado, para aplicaciones de bajo consumo y poco rendimiento, los procesadores Cortex-M0 y Cortex-M0+ son una buena opción, pues consumen alrededor de 10 $\mu W/MHz$ [4]. Algunas de las características de los procesadores de esta familia se resumen en las siguiente lista [3]:

- **Cortex-M0**: procesador pequeño para microcontroladores de bajo coste y muy bajo consumo.
- **Cortex-M0+**: menor consumo y más eficiencia que el Cortex-M0.
- **Cortex-M1**: procesador pequeño optimizado para diseños FPGA.
- **Cortex-M3**: procesador embebido pequeño pero potente para microcontroladores de bajo consumo que cuenta con una ISA mayor (llamada *Thumb 2*) para tareas más complejas. Incluye funciones de depuración.
- **Cortex-M4**: con todas las características del Cortex-M3, con instrucciones para el procesamiento digital de señales (como el divisor de hardware), y operaciones de multiplicación más rápidas. Dispone de una unidad de coma flotante de precisión simple (estándar IEEE-754).
- **Cortex-M7**: procesador de alto rendimiento para microcontroladores de gama alta y aplicaciones de procesamiento intensivo. Añade soporte para coma flotante de doble precisión y mayor velocidad.
- **Cortex-M23**: procesador pequeño para diseños de ultra-bajo consumo y bajo coste, similar al Cortex-M0+ con mejoras en la ISA. Soporta *TrustZone*, un sistema de seguridad para aislar aún más las zonas de memoria y que proteja de posibles atacantes que traten de acceder a los datos.
- **Cortex-M33**: similar a Cortex-M3 y Cortex-M4, pero más eficientes y compatible con *TrustZone*.

Aunque las arquitecturas sean distintas, los códigos de las arquitecturas son compatibles de las más simples a las más complejas. Por ejemplo, un código desarrollado para Cortex-M0, puede ejecutarse en procesadores Cortex-M3, Cortex-M4, o Cortex-M7 solo asegurando la disponibilidad de los puertos de entrada/salida.

IMPORTANTE

Toda esta amalgama de acrónimos que hemos visto, como *ingenieros, físicos, informáticos...* que somos, nos deben sonar, y debemos, al menos, saberlos poner en su sitio. Cuando acabemos la carrera y estemos *por ahí* trabajando, aunque no nos dediquemos a la electrónica tenemos que saber que:

- ARM es una empresa que **diseña procesadores** (CPUs) con un conjunto de instrucciones (ISA) reducido (RISC) llamado *Thumb*, o *Thumb 2*.
- Los procesadores se agrupan en familias, según las características de sus **arquitecturas**. Las **familias** de ARM se llaman *Cortex*.
- Empresas como ST pagan la licencia del diseño y le añaden **periféricos** a la CPU como USBs, I2C, comunicaciones inalámbricas, *etc.* Al final, las empresas **fabrican un microcontrolador**.
- Las empresas que fabrican el microcontrolador suelen diseñar y fabricar placas de desarrollo, como cualquiera de las **Nucleo-STM32**. **En este libro trabajaremos con una placa Nucleo-STM32F446RE fabricada por la empresa ST, que integra el microcontrolador STM32F446RE que tiene una arquitectura Cortex-M4**.
- Otras veces, son otras empresas las que fabrican dispositivos completos (no tienen por qué ser de desarrollo) e integran los microcontroladores. Por ejemplo, la Raspberry Pi 4 —cuyo fabricante es Raspberry Pi Foundation—, usa un procesador Broadcom BCM2711 de la empresa Broadcom, que ha integrado una CPU de ARM, con la arquitectura Cortex-A72.

Capítulo 3

El microcontrolador STM32F446RE

Josué Pagán

Ya hemos mencionado que decantarse por un microcontrolador u otro no es una tarea fácil. En este capítulo vamos a hablar del microcontrolador **STM32F446RE** montado en la placa **Nucleo-STM32F446RE**y sus características básicas (Sección 3.1), el diagrama de bloques del procesador (Sección 3.2), las características eléctricas y alimentación (Sección 3.3), los modos de bajo consumo (Sección 3.4), y el arranque del sistema (Sección 3.5). Este capítulo da paso a los subsiguientes, donde se profundiza en los periféricos del microcontrolador. Por ejemplo, hablaremos del detalle de los registros más importantes de las entradas/ salidas (Capítulo 4), del circuito de reloj y los temporizadores (Capítulo 5), interrupciones (Capítulo 6), o de las interfaces de comunicación, entre otros.

Hablar de la arquitectura del microcontrolador da para redactar libros completos. No obstante, en este capítulo, nos centraremos en los aspectos más generales que debería conocer, y en los detalles que necesitará para seguir el libro. Recuerde siempre tener a mano el *datasheet* [5] y el *reference manual* [6] para ver explicaciones más detalladas. Muchas veces oiréis decir que ambos documentos son *"el libro de la asignatura"*. Y así es. En este libro *separamos el trigo de la paja* y te ayudamos a interpretar dichos documentos para centrarnos en lo que nos interesa.

3.1. Características básicas del STM32F446RE

El microcontrolador **STM32F4**, como nos indica su nombre (ver Figura 1.5), se basa en una arquitectura **Cortex-M4**. Aquí se resumen algunas de sus características, aunque puede encontrar todos los detalles en el *datasheet*.

- CPU ARM Cortex-M4 de 32 bits con Floating Point Unit (Unidad de Punto Flotante) (FPU)
- Frecuencia de CPU hasta 180 MHz

- Rango de tensiones de alimentación $V_{DD} = [1{,}7\ V - 3{,}6\ V]$
- 512 KB de memoria Flash
- 128 KB de memoria SRAM
- 1 temporizador (*timer*) de sistema ***SysTick***
- 2 *watchdog timers*
- 10 *timers* de 16 bits *propósito general*: `TIM2` (extensible a 32 bits), `TIM3`, `TIM4`, `TIM5` (extensible a 32 bits), `TIM9` a `TIM14`
- 2 *timers* de control avanzado: `TIM1` y `TIM8`
- 2 *timers* básicos: `TIM6` y `TIM7`
- 4 interfaces SPI (hasta 45 $Mbits/s$)
- 4 interfaces I2C
- 4 Universal Synchronous and Asynchronous Receiver-Transmitter (Transmisor-Receptor Síncrono y Asíncrono Universal) (USART)
- 2 Universal Asynchronous Receiver-Transmitter (Transmisor-Receptor Asíncrono Universal) (UART)
- 2 CAN
- Interfaz paralela para Liquid-Crystal Display (Pantalla de Cristal Líquido) (LCD)
- Interfaz paralela para cámara digital (hasta 54 $Mbits/s$)
- 50 General Purpose Input/Output (Entrada/Salida de Propósito General) (GPIO) con capacidad de interrupción externa
- 3 ADC de 12 bits con 16 canales
- 2 canales Digital-Analog Converter (Conversor Digital-Analógico) (DAC) de 12 bits

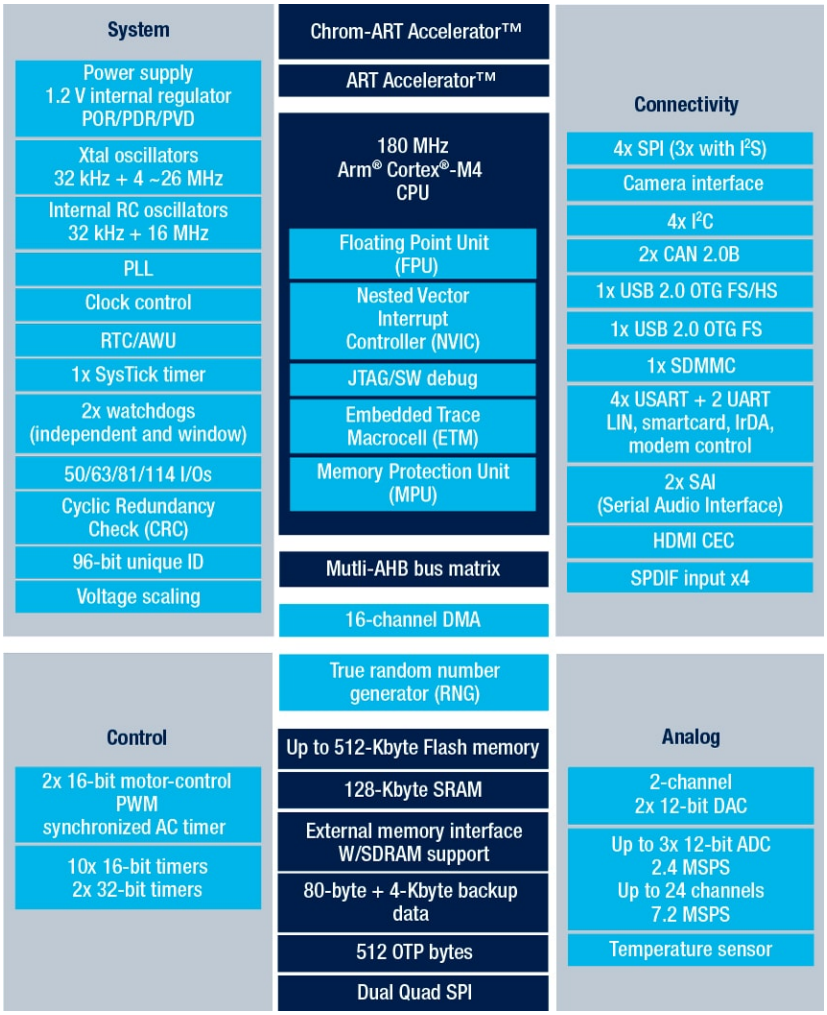


Figura 3.1: Diagrama de bloques del circuito del **STM32F446RE**

3.2. Diagrama de bloques interno

La Figura 3.1 muestra muestra el diagrama funcional[1] de bloques del circuito del **STM32F446RE**[2].

En el centro de la figura podemos ver la CPU indicándonos su frecuencia máxima de funcionamiento. Este tipo de diagramas nos dejan ver qué parte corresponde al diseño de la empresa ARM: el procesador, y qué parte al diseño del fabricante ST: los periféricos y la gestión del sistema (*System* en la Figura 3.1).

El bloque de sistema comprende, principalmente, la gestión de la alimentación del microcontrolador, los relojes, y las GPIO (*I/O* en la Figura 3.1). El bloque de conexiones digitales comprende todas aquéllos buses como I2C, SPI, CAN, *etc.* También podemos destacar el bloque de manejo de canales analógicos, tanto para

[1] Recordar —aunque sea para otro microcontrolador— los diagramas estudiados en teoría.

[2] Características del microcontrolador **STM32F446RE**:
`https://www.st.com/en/microcontrollers-microprocessors/stm32f446re.html`. Accedido: 24-12-2022.

conversión ADC, como para DAC. Por último, en esta familia, podemos ver que los temporizadores (*timers*) más genéricos —no el de sistema *SysTick*, ni los *watchdog*— se gestionan en un bloque aparte, donde también está el control de PWM —que en este microcontrolador depende de los *timers*— para, por ejemplo, manejar motores.

NOTA

Ya dijimos que la familia Cortex-M4 tiene una unidad de punto flotante FPU (estándar `IEEE 754`); y también sabemos que su arquitectura es de 32 bits. Es decir, que los números tienen un tamaño de 32 bits. Si programando, en `C`, el usuario define una variable como `double`, que ocupa 64 bits, como el microcontrolador solo trabaja con 32 bits por sus características HW, el compilador cruzado adaptará el código para hacer la operación mediante Software (SW).
Se recomienda repasar la parte de tipos decimales en el vídeo de `ejemplos`: **[SDG1-C] [2] Tipos, almacenamiento, cualificadores, y estilo en C**

Por completitud, conviene dejar para su deleite un diagrama de bloques y buses de comunicación más detallado. No se preocupe, su comprensión completa no es tarea fácil, pero a los largo del desarrollo del libro y durante la explicación de este capítulo, si recurre al diagrama de la Figura 3.2 irá encajando poco a poco las piezas del puzzle. En la Figura 3.2 podemos ver los buses principales `Advanced High-performance Bus (Bus Avanzado de Alto rendimiento) (AHB)` con sus ramificaciones. En el Capítulo 5, comentaremos más sobre estos buses que "llevan" el reloj.

3.3. Características eléctricas y alimentación

A la hora de conectar cualquier sensor o actuador a nuestro dispositivo, es importante conocer cuáles son las tensiones y corrientes máximas permitidas por cada pin, y en total.

Conviene recordar que todas los GPIO son de alta corriente, y que **la corriente máxima de salida de un pin es de** $25\ mA$, donde **la suma de las corrientes de salida de todos los pines no debe superar** $120\ mA$ (ver características de corriente en el *datasheet* [5]). Si necesita excitar cargas que consuman mucha corriente, use un transistor, optoacoplador, relé... o, en general, cualquier *driver* que proporcione dicha corriente para no dañar nuestro microcontrolador.

También hay que tener en cuenta que el rango de tensiones de alimentación del microcontrolador es $V_{DD} = [1{,}7\ V - 3{,}6\ V]$. Típicamente, en nuestra **Nucleo-STM32F446RE** encontraremos salidas a $3{,}3\ V$, aunque algunos pines son tolerantes a entradas/ salidas de $5\ V$, **pero cuidado, no todos lo son**. En general

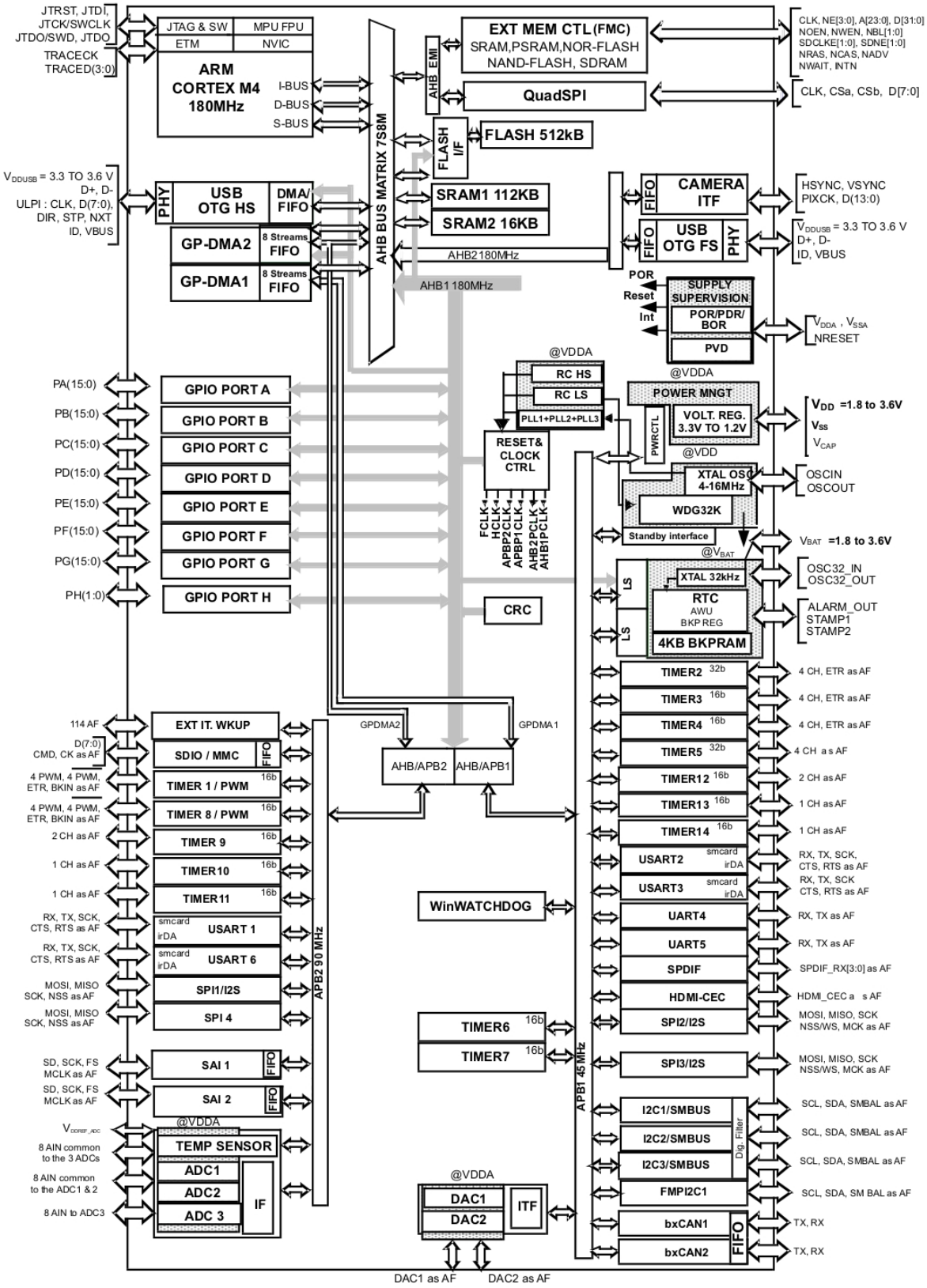


Figura 3.2: Diagrama de bloques y buses en detalle del **STM32F446RE**.

esta información podemos encontrarla en el *datasheet* [5] (ver la tabla *"Table 10. STM32F446xx pin and ball descriptions"* del *datasheet* con todas las características disponibles).

El fabricante ST indica con el acrónimo `FT` que un pin es tolerante a tensiones de 5 V, pero en otros *datasheets* lo puede encontrar de otra forma. En el caso de nuestro **STM32F446RE**, podemos ver que el puerto `PA5` donde está conectado el LED `LD2` no es tolerante a salidas de 5 V. No obstante, el puerto `PA6`, que también está accesible en nuestra placa, sí lo es, como se muestra en la Figura 3.3.

Table 9. Legend/abbreviations used in the pinout table

Name	Abbreviation	Definition
I/O structure	FT	5 V tolerant I/O

Table 10. STM32F446xx pin and ball descriptions

Pin number LQFP64	Pin name (function after reset)	Pin type	I/O structure	Notes	Alternate functions	Additional functions
21	PA5	I/O	TTa	-	TIM2_CH1/TIM2_ETR, TIM8_CH1N, SPI1_SCK/I2S1_CK, OTG_HS_ULPI_CK, EVENTOUT	ADC12_IN5, DAC_OUT2
22	PA6	I/O	FT	-	TIM1_BKIN, TIM3_CH1, TIM8_BKIN, SPI1_MISO, I2S2_MCK, TIM13_CH1, DCMI_PIXCLK, EVENTOUT	ADC12_IN6

Figura 3.3: Extracto del *datasheet* [5] con las tensiones que toleran algunos pines.

NOTA

Aunque no sea dependiente del microcontrolador sino de la placa, conviene advertir que las placa **Nucleo-STM32F446RE** proporciona en el *header* `CN6` acceso a pines con tensiones de alimentación de 3,3 V y 5 V, así como varias referencias a tierra (`GND`).

Estos pines sirven para alimentar otros componentes electrónicos y, como ya se mencionó en la Sección 1.2, **la placa proporciona una corriente máxima de 300 mA. Si el consumo de corriente sobrepasa los 300 mA, se debe alimentar la placa a través de V_{IN} o `E5V`.**

También es buen momento para recordar que **no se debe colocar la placa sobre una superficie conductora o provocaremos un cortocircuito**. Del mismo modo, **no cortocircuite los pines de alimentación y `GND`, y tenga precaución de mirar la tensión máxima permitida por cada pin antes de introducir cualquier señal**.

3.4. Modos de bajo consumo

El microcontrolador cuenta internamente con un regulador de voltaje que gestiona los modos de consumo. Los distintos modos de consumo nos permiten apagar o encender módulos del microcontrolador para ahorrar energía cuando la CPU no necesita estar ejecutándose; como cuando esperamos una interrupción externa. La gestión de los modos de bajo consumo es indispensables en los dispositivos embebidos alimentados por batería como, por ejemplo, los *smart watches*.

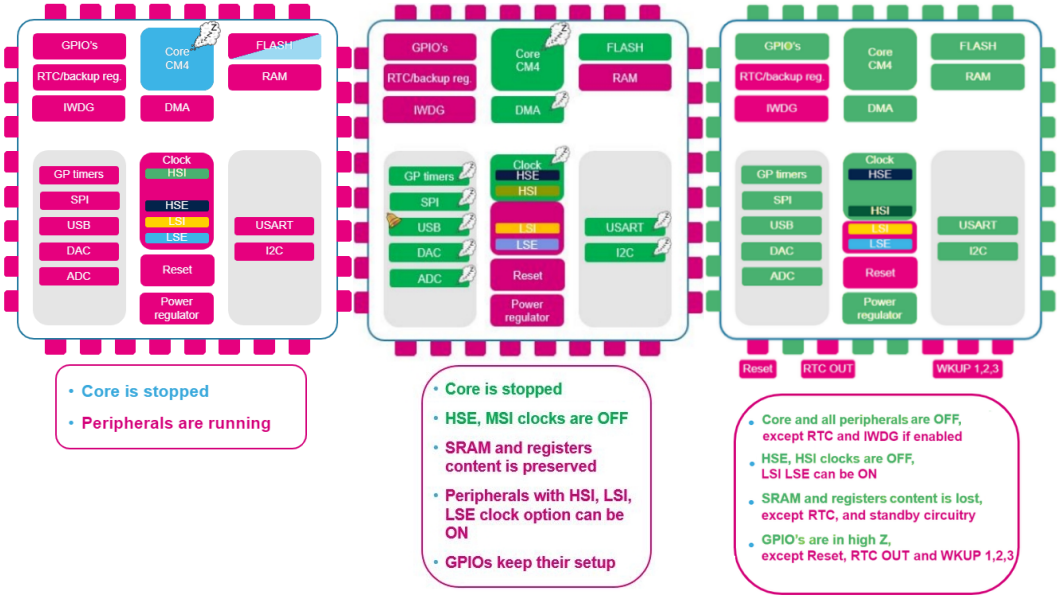
Nuestro microcontrolador gestiona cuatro modos de energía, 3 de ellos de bajo consumo que se muestran en la Figura 3.4[3]:

1. ***Run* o ejecución**. Es el modo por defecto del microcontrolador cuando arranca o tras un reinicio.

2. ***Sleep* o dormido**: se detiene el contador de programa de la CPU y también la FPU. Todos los periféricos continúan operando y la CPU se despierta cuando hay una interrupción.

3. ***Stop***: se detienen todos los relojes y algunos periféricos como la USART y el I2C.

[3]Modos de bajo consumo del microcontrolador: `https://wiki.st.com/stm32mcu/wiki/Getting_started_with_PWR`. Accedido: 10-12-2022.



(a) Modo *Sleep*. *(b)* Modo *Stop*. *(c)* Modo *Standby*.

Figura 3.4: Características de los modos de bajo consumo.

4. ***Standby* o modo espera**: se apagan todos los osciladores y el regulador de voltaje, así como todos los periféricos. Se pierde el contenido de los registros y de la memoria Static Random Access Memory (SRAM). El sistema se despierta por una interrupción de *reset*, *watchdog*, o del reloj Real Time Clock (Reloj de Tiempo Real) (RTC).

En los sistemas que no requieren guardar estados puede ser recomendable irse a un modo *Standby*. En el resto de ocasiones, y dependiendo de los periféricos que manejemos, el modo *Stop* es una buena opción para ahorrar energía.

En las tablas *"Table 23"* a *"Table 27"* (*"Typical and maximum current consumptions in..."*) del *datasheet* podemos ver los consumos estimados del microcontrolador en cada uno de los modos. A una frecuencia de 16 MHz, trabajando con el reloj interno `HSI`, a 25 $°C$ el microcontrolador consume en el caso peor:

- *Run* mode: 13,80 mA
- *Sleep* mode: 13,10 mA (un 5,08 % de ahorro con respecto al modo *Run*)
- *Stop* mode: 1,20 mA (un 91,30 % de ahorro con respecto al modo *Run*)
- *Standby* mode: 8 μA (un 99,94 % de ahorro con respecto al modo *Run*)

3.5. Arranque del sistema

Antes de entrar en detalle con los periféricos, es importante que conozca cómo arranca el sistema. Todos los sistemas tienen un código de arranque que al encenderse configura la memoria, los relojes, las fuentes de alimentación, *etc.* Puede asimilarlo a lo que hace la BIOS de un PC al arrancar: un Firmware (FW) que realiza la inicialización del HW durante el arranque. Los microcontroladores también deben hacerlo. Nuestro **STM32F446RE** no es una excepción.

Cuando arranca la MCU no se ejecuta inmediatamente la función `main()` de nuestro programa, sino que se ejecutan una secuencia de códigos de inicialización. Al final de esta secuencia de arranque (`boot`), es cuando finalmente se ejecuta el `main()`. A continuación, se presenta la secuencia de procesos que lleva a cabo el sistema durante el arranque [7]:

1. El código que se ejecuta al encender el dispositivo es el del fichero en ensamblador `startup_stm32f446xx.s`. Este fichero contiene:

   - Un código que prepara la memoria para ejecutar nuestro programa
   - La definición de los vectores de interrupción (*i.e.*, direcciones de memoria donde van a saltar las Interrupt Service Routine (Rutina de Servicio/Atención a la Interrupción) (ISR)). De hecho, todo parte de la tabla de vectores de interrupción (ver Figura 6.1), que está en una zona fija de la memoria *flash*. Cada una de las entradas de la tabla ocupa 32 bits y contiene la dirección de memoria donde está el código de la ISR a la que tiene que saltar.

2. Si abrimos el fichero `startup_stm32f446xx.s`, el encabezado (Figura 3.5a) nos dice que lo primero que hace es ubicar el *puntero de pila* (`SP`, *stack pointer*) y el *contador de programa* (`PC`, *program counter*) en la etiqueta `Reset_Handler`. Por lo que, **el primer código que se ejecuta es el referido por la etiqueta de ensamblador** `Reset_Handler` (ver Figura 3.5b).

3. Después llama a la función `SystemInit()` que deberá hacer otras inicializaciones, como la de la FPU o memorias externas.

4. Por último irá a la función `main()`.

5. **Todavía queda inicializar los relojes del sistema y la alimentación**. Las llamadas a la/las función/nes que hacen esto podría hacerlas el fichero `startup_stm32f446xx.s` pero en la forma en la que vamos a implementar los proyectos en este libro, nos toca hacer dicha llamada e implementación a

nosotros, por lo que **es muy importante que nos se nos olvide hacer dicha llamada, o el programa no funcionará.**

```
1  ;******************************************************************************
2  ;* File Name          : startup_stm32f446xx.s
3  ;* Author             : MCD Application Team
4  ;* Description        : STM32F446x devices vector table for MDK-ARM toolchain.
5  ;*                      This module performs:
6  ;*                      - Set the initial SP
7  ;*                      - Set the initial PC == Reset_Handler
8  ;*                      - Set the vector table entries with the exceptions ISR address
9  ;*                      After Reset the CortexM4 processor is in Thread mode,
10 ;*                      priority is Privileged, and the Stack is set to Main.
11 ;******************************************************************************
```

(a) Encabezado del fichero de *startup*.

```
179  ; Reset handler
180  Reset_Handler    PROC
181                   EXPORT  Reset_Handler             [WEAK]
182          IMPORT  SystemInit
183          IMPORT  __main
184
185                   LDR     R0, =SystemInit
186                   BLX     R0
187                   LDR     R0, =__main
188                   BX      R0
189                   ENDP
```

(b) `Reset_Handler` en el fichero de *startup*.

Figura 3.5: Secciones del fichero de arranque `startup_stm32f446xx.s`.

3.6. Resumen

En este capítulo hemos visto los detalles más relevantes de la arquitectura Cortex-M4 del microcontrolador **STM32F446RE**, y las características básicas del microcontrolador como su alimentación, los modos de bajo consumo, y el arranque del sistema. En los capítulos siguientes veremos los detalles de los periféricos que nos interesan y cómo configurarlos, describiendo algunos de los registros que más nos van a interesar. Se verá cómo configurar una GPIO para controlar el LED `LD2` de la placa **Nucleo-STM32F446RE**, cómo actuar ante la interrupción de una entrada externa como el botón `B1` de dicha placa, también veremos cómo generar y gestionar ISRs de un temporizador, configurado como contador para interrupciones internas y cómo PWM, entre otras muchas cosas.

Este capítulos nos abre la puerta, en definitiva, a que en los siguientes veamos los entresijos HW de nuestra plataforma. Tenemos la suerte de que, generalmente, los fabricantes diseñan placas de desarrollo que, aunque incorporen distintos microcontroladores, si son de la misma familia, o serie, mantienen el *pinout*, donde con alta probabilidad las funciones de los pines elegidos suelen ser intercambiables. Tal es el caso que, por ejemplo, las **Nucleo-STM32** de la serie *F4xx* son casi intercambiables en sus entradas/salidas, aunque siempre hay que comprobar las configuraciones de reloj, temporizadores, *etc.* En el capítulo Capítulo 9 veremos los detalles del *setup* SW que proponemos para que, dado el caso de que quisiésemos cambiar de placa o microcontrolador, los cambios sean mínimos y controlados.

Capítulo 4

Entradas y salidas de propósito general (GPIOs)

Josué Pagán

Como todo microcontrolador que se precie el **STM32F446RE** tiene puertos de entrada/ salida de propósito general y nosotros debemos definir por SW cómo han de comportarse. Conviene que recuerde los conceptos generales de los bloques de entrada/ salida que ha estudiado, pues son aplicables a cualquier microcontrolador, aunque la arquitectura sea distinta.

Las entradas y salidas de propósito general (GPIOs) nos permiten sensar magnitudes físicas del entorno y actuar sobre él como muestra la Figura 4.1.

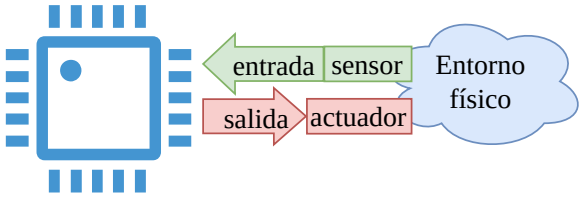


Figura 4.1: Entrada y salida para sensar y actuar sobre el entorno.

Las GPIOs se organizan en puertos que se nombran con letras, y líneas que se enumeran con números. El microcontrolador **STM32F446RE** tiene 8 puertos que se llaman: `GPIOA`, `GPIOB`, `GPIOC`,..., `GPIOH`. Todos tienen 16 líneas excepto el puerto `H` que tiene 2.

Nos vamos a entretener un poco en esta sección porque trabajaremos en detalle con alguno de los registros de las GPIO. Pondremos imágenes de los registros de mayor interés para nuestra aplicación y una explicación de cómo interpretar la información del *reference manual* [6] . Recuerde de nuevo que en el *datasheet* [5] y en el *reference manual* tiene explicaciones más detalladas.

NOTA

En el *pinout* de las placas **Nucleo-STM32** no están accesibles todos los puertos del microcontrolador, pues el fabricante ST nos da acceso solamente a algunos de ellos mediante los *header*—en general esto pasa en casi todas las placas de desarrollo. Todos los fabricantes que venden placas de desarrollo con el *pinout* de Arduino (UNO, o Nano), aseguran que los puertos accesibles a través de estos pines sean compatibles, por lo que si usamos estos pines y cambiamos de microcontrolador en una placa también compatible con Arduino, sabemos que tendremos las mismas funciones disponibles con muy pocas modificaciones, o ninguna.

4.1. Configuración de GPIOs

Cada uno de los pines GPIO puede configurarse por SW según las necesidades de la aplicación (conviene que mire el circuito *"Figure 18. Basic structure of a 5 V tolerant I/O port bit"* del *reference manual* [6]). Se pueden configurar:

- como **entrada** flotante,
- como **entrada** en *pull-up*,
- como **entrada** en *pull-down*,
- como **analógica**,
- como **salida** en *open-drain* (drenador abierto) con o sin *pull-up* o *pull-down*,
- como **salida** *push-pull* con o sin *pull-up* o *pull-down*,
- como **función alternativa**. La mayoría de los pines GPIO se comparten con funciones alternativas digitales o analógicas. Pueden configurarse también en drenador abierto o *push-pull*, y con o sin *pull-up* o *pull-down*,
- Todas los GPIO son de alta corriente. **La corriente máxima de salida de un pin es de** 25 mA. **La suma de las corrientes de salida de todos los pines no debe superar** 120 mA (ver características de corriente en el *datasheet*).
- Todas las GPIO permiten configurar su velocidad para gestionar mejor el ruido interno, el consumo de energía y las emisiones electromagnéticas. Se permite una frecuencia máxima de entrada/ salida de hasta 90 MHz.
- A las GPIO se les pueden asignar interrupciones (ver Capítulo 6).

4.2. Registros de las GPIOs

Cada GPIO tiene 10 registros para su configuración. Podemos ver todos los registros de las GPIO en el apartado *"7.4. GPIO registers"* del *reference manual* [6]. Resaltamos en negrita los que más usaremos, la `x` de **GPIOx_MODER** hay que sustituirla por la letra del puerto:

- 4 registros de control de 32 bits: **GPIOx_MODER**, `GPIOx_OTYPER`, `GPIOx_OSPEEDR` y **GPIOx_PUPDR**.
- 2 registros de datos: **GPIOx_IDR** para leer el valor de entrada y **GPIOx_ODR** para escribir el valor de salida.
- 1 registro de *set/reset*: **GPIOx_BSRR** para escritura atómica (sin interrupción) a nivel de bit sobre el registro de salida `GPIOx_ODR`.
- 1 registro de bloqueo: `GPIOx_LCKR`.
- 2 registros de función alternativa: **GPIOx_AFRH** y **GPIOx_AFRL**.

4.2.1. GPIO port mode register (GPIOx_MODER)

El registro **GPIOx_MODER** sirve para configurar el **modo** de un pin como (i) entrada, (ii) salida, (iii) función alternativa, o (iv) analógico. En general trabajaremos (i) **con máscaras para seleccionar los bits** que queremos modificar de un registro, (ii) **limpiarlos**, y (iii) **escribir** el valor deseado, o (iv) **leer** el valor guardado.

Vamos a ver detallado, paso a paso, cómo se haría la configuración del registro **GPIOx_MODER** para definir el puerto A-línea 5 (`PA5`) como salida para encender y apagar el LED `LD2` de la placa **Nucleo-STM32F446RE**. Para el resto de registros sería análogo. **Debe tener soltura con la representación binaria, hexadecimal, y decimal, y con el concepto de máscara**. Se recomienda ver el vídeo de otros cursos con contenido relacionado (a partir del minuto 2:50).

IMPORTANTE

La representación de los registros en el *reference manual*, suele ser de 32 bits dividiendo los 16 más significativos en la parte de arriba (posiciones `16` a `31`), y los 16 menos significativos en la parte de abajo (posiciones `0` a `15`). Los bits pueden agruparse, o no, con un nombre que implica una función.

Hay veces que algunos bits de algunos registros no están implementados, *i.e.*, el fabricante no nos deja acceder a dichas posiciones de memoria.

Los bits de los registros pueden ser de lectura y escritura, o a veces, solo de lectura.

Los bits de los registros tienen valores por defecto cuando se arranca o reinicia el microcontrolador; son los *reset* o *default values* que nos indique el fabricante. Estos valores pueden ser un valor indefinido, todo `'0s'`, o tener valores concretos. En definitiva, por defecto hay una configuración inicial del sistema.

IMPORTANTE

Antes de configurar una GPIO, primero, siempre hay que habilitar el reloj del puerto con el que vayamos a trabajar. En caso de trabajar con el `PA5`, en `C` —con los `#define` del estándar Common Microcontroller Software Interface Standard (CMSIS)— haremos:

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

Mirando el registro **GPIOx_MODER** en la Figura 4.2a, vemos que la '`y`' ha de ser sustituida por el número de la línea/ pin, en nuestro caso, el `5`. Y por tanto, debemos configurar los bits **MODER5** (posiciones `10` y `11`) poniendo un `'01'` para que sea una salida (*output*). Para ello (i) **definimos la máscara para seleccionar los bits**, (ii) **los limpiamos**, y (iii) **escribimos** el valor `'01'`:

1. **Definimos la máscara**. Podemos hacerlo de dos formas:

   - Esta forma es genérica y nos permite usarla en una función o macro recibiendo como argumento la línea del puerto. **Definimos una máscara base binaria** con tantos `1s` como bits queremos seleccionar —*base* quiere decir: en las posiciones de bits menos significativas—. Seguidamente **desplazamos esa máscara base a la izquierda** tantas posiciones como sean necesarias.

     Por ejemplo, para seleccionar los bits **MODER5** (posiciones `10` y `11`), escribimos la máscara base binaria `0b11`, que son dos bits, por tanto, dos

1s, que indican las posiciones 0 y 1. 0b11 en hexadecimal lo representamos como 0x03 y en decimal como 3. A continuación desplazamos estos bits 10 posiciones a la izquierda de tal forma que el '1' de la posición 0 de nuestra máscara quedará *señalando* la posición 10 del registro, y el '1' de la posición 1 quedará en la posición 11. En C, con #define podría hacerse:

```
#define LINEA_5 5
#define BASE_MODER_MASK 0x03
#define MODER5_MASK (BASE_MODER_MASK << 2*LINEA_5) /* Desplaza los bits 0
    b11 a la izquierda 2*5=10 posiciones */
```

- La otra forma consistiría en definir las máscaras necesarias para cada conjunto de bits que queramos seleccionar. Esta opción le resultará más cómoda y fácil de entender al principio, pero cuando tenga soltura y vea que repite muchas líneas de código, es menos flexible y se pueden cometer más errores, preferirá la anterior. En C, podría hacerse:

```
#define MODER5_MASK 0x0C00 /* En binario: 0b1100 0000 0000 */
```

2. Limpiamos lo que hubiese escrito en esas posiciones del registro haciendo **el producto bit a bit (&) con la inversa (~) de la máscara**.

```
GPIOA->MODER &= ~MODER5_MASK;
```

3. Escribimos el valor haciendo **la suma bit a bit (|) con el valor deseado**. Para ello hacemos el desplazamiento a izquierdas del valor deseado tantas posiciones como sean necesarias.

```
#define GPIO_MODE_OUT 0x01U /* La U indica unsigned (sin signo)*/
GPIOA->MODER |= (GPIO_MODE_OUT << 2*LINEA_5);
```

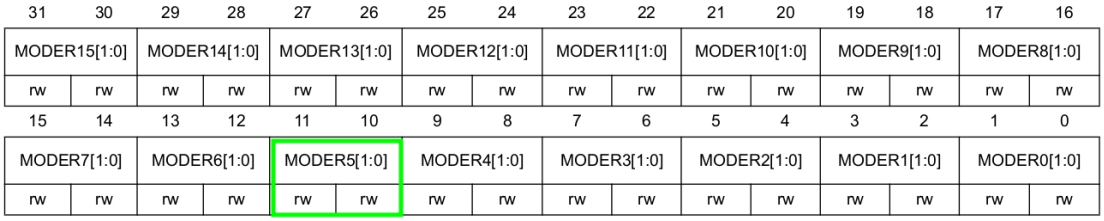
Otra opción, menos flexible, sería definir una máscara para *salida* solo para los bits MODER5, pero no se recomienda:

```
#define MODER5_AS_OUTPUT 0x0400 /* En binario: 0b0100 0000 0000*/
GPIOA->MODER |= MODER5_AS_OUTPUT;
```

Ya tenemos el pin al que está conectado el LED LD2 definido como salida.

NOTA

Puede trabajar con la representación decimal o hexadecimal indistintamente (la representación binaria no es aceptada por todos los compiladores).
También habrá notado que la forma de acceder a los registros de una GPIO se hace con GPIOx->registro. Esto es porque las GPIOx están definidas como estructuras por CMSIS. Se explica en el Capítulo 9.



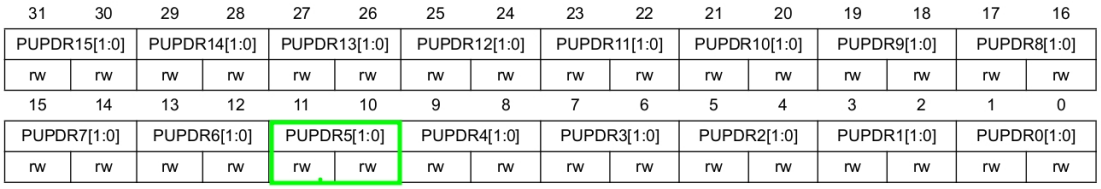
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Reset values:
- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

(a) GPIOx_MODER. Configuración de salida para LED LD2.



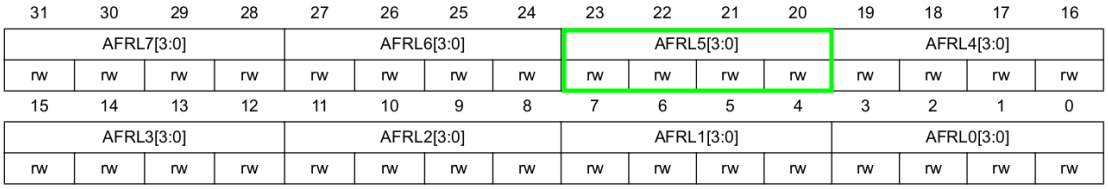
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

Reset values:
- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

(b) GPIOx_PUPDR. Configuración no-pull para LED LD2.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFRLy:** Alternate function selection for port x bit y (y = 0..7)

These bits are written by software to configure alternate function I/Os

AFRLy selection:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

(c) GPIOx_AFRL. Configuración de PWM en timer 2 u 8 para LED LD2.

Figura 4.2: Registros GPIOx_MODER, GPIOx_PUPDR y GPIOx_AFRL.

ATENCIÓN

El fabricante ST nos indica que cada bit de cada GPIO es totalmente programable. No obstante el acceso individual a bits no está permitido. **El acceso a los registros ha de ser en palabras de 32**, 16, u 8 bits.

Por ejemplo, esta escritura:

✓
```
GPIOA->ODR |= 0x20; /*Lectura y escritura de 32 bits. 0x20 es la mascara hexadecimal del 6o bit (linea 5), que en binario seria: 0b0010 0000*/
```

no puede hacerse de esta manera:

×
```
GPIOA->MODER[5] = 1; /*Hipotetica escritura de un solo bit */
```

En otros microcontroladores podría ser factible hacerlo así, todo depende del fabricante.

IMPORTANTE

Las operaciones que podemos hacer con las máscaras y los registros (o variable) son:

- **Leer** multiplicando bit a bit (&) por la máscara: `valor = REGISTRO & MASK;`
- **Limpiar** multiplicando bit a bit (&) por la inversa (~) de la máscara: `REGISTRO &= ~MASK;`
- **Escribir** sumando bit a bit (|) el valor: `REGISTRO |= valor;`

4.2.2. GPIO port pull-up/pull-down register (GPIOx_PUPDR)

Todos los pines de las GPIO tienen resistencias *pull-up y pull-down* que pueden activarse, o desactivarse en función del circuito que montemos en el pin. (ver el circuito *"Figure 18. Basic structure of a 5 V tolerant I/O port bit"* del *reference manual* [6]). El registro **GPIOx_PUPDR** de la Figura 4.2b sirve para especificar la configuración de dichas resistencias.

En nuestro caso, el circuito del LED `LD2` no necesita ninguna resistencia de *pull-up o pull-down* (ver Figura 1.12), por lo que podemos simplemente limpiar los bits **PUPD5** del **GPIOx_PUPDR** dejando el valor `'00'`:

```
1 #define BASE_PUPDR_MASK 0x03
```

```
2 #define PUPD5_MASK (BASE_PUPDR_MASK << 2*LINEA_5) /* Desplaza los bits 0b11 a la
      izquierda 2*5=10 posiciones */
3 GPIOA->PUPDR &= ~PUPD5_MASK; /* Limpiamos*/
```

Ya tenemos el pin al que está conectado el LED `LD2` configurado.

4.2.3. GPIO alternate function registers (GPIOx_AFRL y GPIOx_AFRH)

Ya comentamos que un pin puede actuar como una entrada o salida digital, como un pin analógico, o puede hacer una **función alternativa**. Por ejemplo, el `LD2`, como está configurado según lo expuesto anteriormente, solo puede estar encendido totalmente, o apagado. Imagine que quisiésemos encenderlo al 50 % de su intensidad, para que brille menos. Esto podríamos hacerlo haciendo una modulación PWM en dicho pin con un ciclo de trabajo (*duty cycle*) del 50 % (conviene repasar los conceptos de PWM que haya estudiado), pero deberíamos configurar este pin con la **función alternativa** de *timer* para poder hacer PWM. En este microcontrolador no existe un módulo de PWM independiente, sino que se consigue con los temporizadores.

Tenemos hasta 16 funciones alternativas disponibles por cada línea GPIO (ver la tabla *"Table 11. Alternate function"* del *datasheet* con todas las opciones disponibles). Necesitamos 4 bits para poder definir hasta 16 funciones alternativas. Como tenemos 16 líneas GPIO, necesitamos por tanto $4\ bits * 16 = 64$ bits. Es por ello que no tenemos solo un registro de función alternativa GPIOx_AFR, sino que necesitamos dos, cada uno de 32 bits. Estos son los registros **GPIOx_AFRL** (*alternate function register low* para las líneas de `0` a `7`) y **GPIOx_AFRH** (*alternate function register high* para las líneas de `8` a `15`).

Así pues, si quisiésemos poner PWM en el `PA5` para modular la luminosidad del LED `LD2`, deberíamos escribir en los bits **AFRL5** (ver Figura 4.2c) la función alternativa `AF1` o `AF3`, según lo indicado en la Figura 4.3. Si eligiésemos el *timer* `2` sería `AF1` según el *reference manual*. En `C` —con el estándar y máscaras de CMSIS—, podría ser:

```
1 #define ALT_FUNC_TIM2 0x01U   /* AF1 segun reference manual */
2 #define BASE_ALT_MASK 0x0F  /* En binario: 0b1111 (4 bits)*/
3 #define AFRL5_MASK (BASE_ALT_MASK << 4*LINEA_5) /* Desplaza los bits 0b1111 a la
      izquierda 4*5=20 posiciones */
4
5 /* AFRL y AFRH no existen en la estructura GPIOA, sino que son AFR[0] y AFR[1]
      respectivamente */
6 GPIOA->AFR[0] &= ~AFRL5_MASK; /* Limpiamos*/
7 GPIOA->AFR[0] |= (ALT_FUNC_TIM2 << 4*LINEA_5); /* Escribimos valor*/
```

También podemos averiguarlo si miramos a la Figura 1.11b, donde podemos ver que el `PA5` puede tomar la función alternativa PWM en el canal 1 del *timer* `2` (`PWM2/1`) 

. No obstante, esta información tan detallada no la proporcionan todos los fabricantes de placas de desarrollo, y

conviene siempre mirar el *datasheet*.

Si eligiésemos el *timer* 8, modificaríamos el `#define` y la escritura, y se mantienen las máscaras:

```
#define ALT_FUNC_TIM8 0x03U   /* AF3 segun reference manual */
GPIOA->AFRL &= ~AFRL5_MASK; /* Limpiamos*/
GPIOA->AFRL |= (ALT_FUNC_TIM8 << 4*LINEA_5); /* Escribimos valor*/
```

Port	AF0	AF1	AF2	AF3
	SYS	TIM1/2	TIM3/4/5	TIM8/9/10/11 CEC
PA5	-	TIM2_CH1/ TIM2_ETR	-	TIM8_ CH1N

Figura 4.3: Funciones alternativas en la línea/ pin 5 del port A.

La GPIO ya estaría configurada, pero **ahora quedaría configurar el modo** del *timer* elegido para que actúe como PWM. Esto lo veremos más adelante, y en el proyecto.

4.2.4. GPIO port input data register (GPIOx_IDR)

Este registro sirve para **leer el valor binario de una GPIO**. El valor se actualiza en cada ciclo de reloj. Aunque la GPIO esté configurada como salida, podemos leer el valor del pin. La Figura 4.4a muestra el registro GPIOx_IDR. En la figura se indica que solo sus 16 primeros bits están accesibles en modo lectura solo (`r`), y que su valor al inicio es indefinido (`X`).

Si quisiésemos leer el estado del LED `LD2`, en `C` podríamos hacerlo así:

```
#define IDR5_MASK 0x20 /* En binario 0b0010 0000 */
bool value = (bool)(GPIOA->IDR & IDR5_MASK); /* Leemos el valor */
```

4.2.5. GPIO port output data register (GPIOx_ODR)

Este registro sirve para **escribir un valor binario en una GPIO**. La Figura 4.4b muestra el registro `GPIOx_ODR`. Es interesante la nota que nos indica el *reference manual*, en la que dice que **para una lectura atómica debemos utilizar el registro GPIOx_BSRR**. La escritura en el registro **GPIOx_ODR** puede verse interrumpida por una ISR (de un reloj, entrada externa, *etc.*). Esto puede ser un problema si la ISR por algún motivo tarda mucho o se queda bloqueada y necesitamos escribir con velocidad en el pin. En casos críticos, o por seguridad, para asegurar que la escritura se hace sin interrupción —de manera atómica— podemos hacerla con el registro **GPIOx_BSRR**.

Si quisiésemos encender (`set`) y/o apagar (`reset`) el estado del LED, podríamos hacerlo así:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **IDRy**: Port input data (y = 0..15)

These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

Reset value: 0x0000 XXXX (where X means undefined)

(a) GPIOx_IDR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..H).

(b) GPIOx_ODR

Figura 4.4: Registros GPIOx_IDR y GPIOx_ODR.

```
1 #define ODR5_MASK 0x20 /* En binario 0b0010 0000 */
2 GPIOA->ODR |= ODR5_MASK; /* Encender: escribimos un 1 logico */
3 GPIOA->ODR &= ~ODR5_MASK;  /* Apagar: escribimos un 0 logico (limpiamos) */
```

4.2.6. GPIO port bit set/reset register (GPIOx_BSRR)

Cuando hacemos una escritura en el registro `GPIOx_ODR` la hacemos con `OR-igual`. La línea anterior:

```
1 GPIOA->ODR |= ODR5_MASK;
```

realmente es:

```
1 GPIOA->ODR = GPIOA->ODR | ODR5_MASK; /* (i) Lectura de ODR, (ii) OR y (iii)
      escritura en ODR */
```

Si entre esa lectura y la escritura alguien (otra función, ISR) lee y escribe en el registro `GPIO_ODR`, estaremos luego nosotros machacando lo que la otra función quería hacer, **esto es lo que se llama condición de carrera**. Así, como hemos comentado antes, **para una lectura atómica debemos utilizar el registro GPIOx_BSRR**. **Atómico quiere decir que** una vez que se accede al registro para escribir en él, **la operación se ejecuta íntegramente sin interrupción.**

En el registro `GPIO_BSRR` (ver Figura 4.5), si escribimos un `'1'` en los **16 bits menos significativos, sirve para poner el pin correspondiente a nivel alto**

(`BSy` → *bit set*). Si escribimos un `'1'` en los **16 bits más significativos, sirve para poner el pin correspondiente a nivel bajo** (`BRy` → *bit reset*). En realidad no ataca directamente al pin, sino al registro `GPIOx_ODR`, pero sin que le interrumpan.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)
These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Resets the corresponding ODRx bit
Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)
These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

Figura 4.5: Registro GPIOx_BSRR

Escribir un `'0'` no hace nada en el registro, por lo que si quisiésemos encender (`set`) y/o apagar (`reset`) el estado del LED, podríamos hacerlo así:

```
1 #define BSRR5_MASK 0x20 /* En binario 0b0010 0000 */
2 GPIOA->BSRR = BSRR5_MASK; /* Encender: los 16 LSB. = porque no afecta escribir 0 */
3 GPIOA->BSRR = BSRR5_MASK << 16; /* Apagar: los 16 MSB. Desplazo 16 bits (o nueva
      mascara) */
```

Capítulo 5

Circuito de reloj

Josué Pagán

En este capítulo veremos las características principales de los temporizadores para que tenga conocimiento de las fuentes de reloj del sistema y periféricos. El circuito de reloj del **STM32F446RE** es muy potente y complejo y nos permite configurar el reloj del sistema y de los periféricos de muchas formas diferentes.

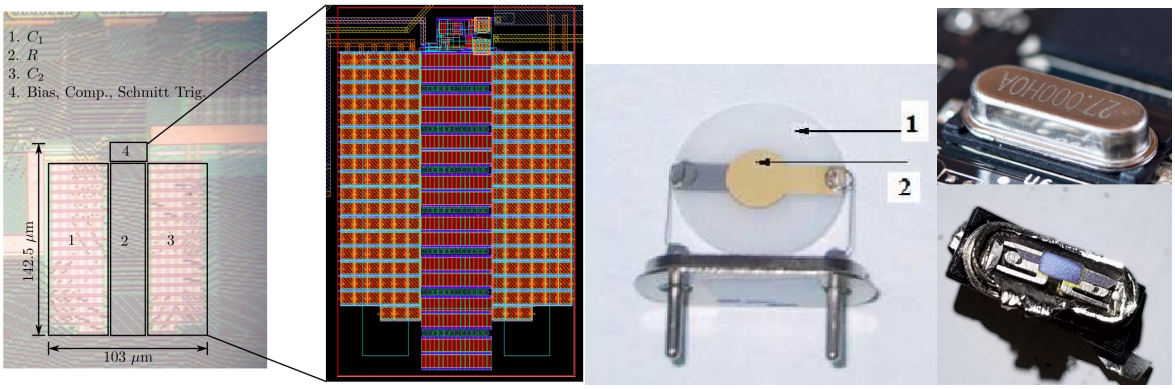
Primeramente conviene que sepa que la señal de reloj es una señal cuadrada y periódica que se puede generar dentro (interna), o fuera (externa) del microcontrolador.

Los relojes internos suelen ser, típicamente, circuitos `RC` formados por una cascada de resistencias y condensadores litografiados en el *chip*. La Figura 5.1a [8] muestra una imagen microscópica y el *layout* de un oscilador `RC` sencillo que oscila a 10 kHz. Los relojes internos no consiguen muy altas frecuencias y suelen tener desviaciones que aumentan con la temperatura y pueden resultar críticas en algunas aplicaciones. Uno de los beneficios de los relojes internos es que son baratos y no requieren del montaje de un circuito exterior.

Por otro lado, los relojes externos son más estables con los cambios de temperatura y a lo largo del tiempo. Un tipo de reloj externo muy usado y que podemos montar en la placa **Nucleo-STM32F446RE** (componente `X3`, ver Figura 1.7) es el reloj de cristal de cuarzo. Se trata de una lámina de cristal de cuarzo que mediante el efecto piezoeléctrico proporciona una tensión periódica muy estable que extraemos mediante electrodos metálicos (ver Figura 5.1b [9]). Suele tener un circuito de acondicionamiento para filtrar y amplificar la señal. Lo podemos encontrar en encapsulados como el de la Figura 5.1c[1].

Fuentes de reloj del STM32F446RE

[1]Autor: Chamblis. CC BY-SA 4.0 `https://en.wikipedia.org/wiki/Crystal_oscillator#/media/File:Quartz_crystal_internal.jpg`



(a) Oscilador RC. *(b)* Lámina de cuarzo. *(c)* Encapusaldo e interior.

Figura 5.1: Sistemas de generación del reloj del sistema.

El Reset and Clock Controller (Controlador de Reinicio y Reloj) (RCC) es el sistema que gestiona los mecanismos de reinicio y de generación de los relojes del sistema y periféricos. El RCC es muy flexible y ofrece una gran variedad de fuentes de reloj, que se pueden seleccionar en función de los requisitos de bajo consumo, precisión y rendimiento.

La Figura 5.2 muestra el circuito del árbol de relojes del **STM32F446RE**, señalando el camino principal por defecto del reloj del sistema (el SYSCLK). Al arrancar el sistema, el oscilador RC interno de 16 MHz es el reloj por defecto de la CPU. Este oscilador tiene una precisión del 1 % a 25 $°C$. Nuestro SW puede elegir como SYSCLK el oscilador interno RC o una fuente de reloj externa de $4 - 26$ MHz. Este reloj puede monitorizarse para detectar fallos en su señal. Si se detecta un fallo, el sistema vuelve automáticamente al oscilador RC interno y se genera una interrupción. El SYSCLK se introduce en un Phase Locked Loop (PLL) que permite aumentar la frecuencia hasta 180 MHz.

Varios *prescalers* permiten configurar: (i) los dos buses AHB, (ii) el Advanced Peripheral Bus (Bus Avanzado de Periféricos) (APB) de alta velocidad APB2, y (iii) el APB de baja velocidad APB1. La frecuencia máxima de los dos buses AHB es de 180 MHz, la del APB2 es de 90 MHz, y la del APB1 de baja velocidad es de 45 MHz.

Se pueden utilizar 3 fuentes de reloj diferentes para controlar el reloj del sistema SYSCLK:

- Reloj oscilador High Speed Internal clock (HSI) (HSI RC en la Figura 5.2).
- Reloj oscilador High Speed External clock (HSE) (HSE OSC).
- Relojes generados por los PLL principales.

Además se dispone de 2 fuentes de reloj secundarias:

- Un RC interno de baja frecuencia de 32 kHz (LSI RC) que gestiona el *watchdog timer*, y opcionalmente, el RTC utilizado para despertarse automáticamente en los modos de bajo consumo *Stop* o *Standby*.
- Un cristal externo de baja frecuencia de 32,768 kHz (LSE OSC, componente X2 montado en la placa, ver Figura 1.7) que gestiona, opcionalmente, el reloj RTC (RTCCLK).

5.1. Relojes de periféricos

Los relojes de periféricos son señales derivadas de las fuentes de reloj. Si en el margen izquierdo de la Figura 5.2 vemos las entradas de las fuentes de reloj externas e internas, en el margen derecho de la misma figura vemos los relojes de periféricos (*clocks*) que se derivan de los distintos buses. Cada fuente de reloj puede activarse o desactivarse independientemente, cuando no se utiliza, para optimizar el consumo de energía.

Después de cada reinicio del dispositivo todos los relojes periféricos se desactivan—una GPIO, un *timer*, I2C...—, excepto el de las memorias SRAM y Flash. Antes de utilizar un periférico hay que habilitar su reloj en el registro **RCC_AHBxENR** o **RCC_APBxENR**.

Los buses de reloj AHB y APB proporcionan conexiones síncronas y permiten una selección de la frecuencia de periféricos flexible. El reloj AHB es una señal preescalada, o no (*prescaler* = 1), del reloj del sistema SYSCLK. De él salen, a la frecuencia del AHB:

- El reloj de la CPU.
- El reloj FCLK Cortex.
- Un reloj de periféricos.

Un reloj muy importante es el SysTick o *Cortex System Timer*. Este *timer* es la señal de reloj AHB dividida entre 8. El reloj AHB es a su vez el SYSCLK preescalado.

IMPORTANTE

El SysTick es típicamente configurado para generar interrupciones cada 1 ms **y aumentar un contador que lleva la cuenta de *"tiempo del sistema"*. Se utiliza como temporizador de referencia en muchas aplicaciones**. Cada vez que el SysTick genera una interrupción se activa su ISR (SysTick_IRQn). Esta interrupción es del procesador, de ARM, no del fabricante.

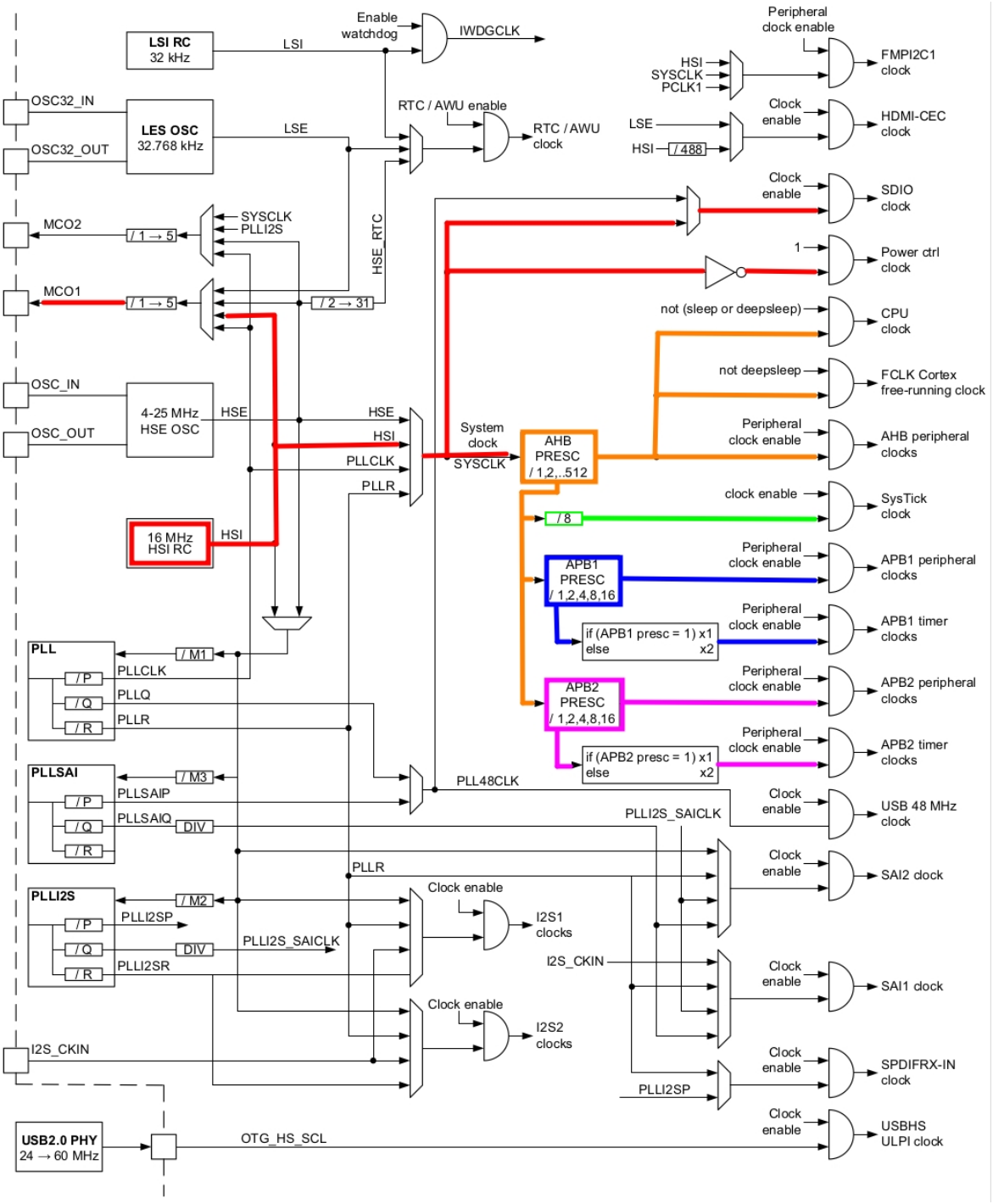


Figura 5.2: Circuito del árbol de relojes del **STM32F446RE**.

El reloj APB1 es una señal preescalada, o no (*prescaler* = 1), del AHB. El reloj APB2 es una señal preescalada, o no (*prescaler* = 1), del AHB. De cada uno de ellos salen, a las frecuencias del APB1 y del APB2 respectivamente:

- Un reloj *clock* de periféricos.
- Un reloj *timer clock* de referencia hacia los temporizadores del microcontrolador.

Todos los relojes de periféricos del RCC se controlan mediante registros. Los bits de los registros **RCC_AHBxENR** o **RCC_APBxENR** nos permiten, por ejemplo, activar los relojes de las GPIOs, o de los *timers*, como se muestra en la Figura 5.3. En la figura se señalan, como ejemplo, algunos de bits de activación de los relojes de GPIO, o temporizadores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	OTGHS ULPIEN	OTGHS EN	Res.	Res.	Res.	Res.	Res.	Res.	DMA2 EN	DMA1 EN	Res.	Res.	BKP SRAMEN	Res.	Res.
	rw	rw							rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CRC EN	Res.	Res.	Res.	Res.	GPIOH EN	GPIOG EN	GPIOF EN	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw					rw	rw	rw	rw	rw	rw	rw	rw

(a) RCC_AHB1ENR para activar los relojes, entre otros, de las GPIOs.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	DAC EN	PWR EN	CEC EN	CAN2 EN	CAN1 EN	FMPI2C1 EN	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART3 EN	USART2 EN	SPDIFRX EN
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Res.	Res.	WWDG EN	Res.	Res.	TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

(b) RCC_APB1ENR para activar los relojes, entre otros, de algunos *timers*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	SAI2 EN	SAI1 EN	Res.	Res.	Res.	TIM11 EN	TIM10 EN	TIM9 EN
								rw	rw				rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	SYSCFG EN	SPI4 EN	SPI1 EN	SDIO EN	ADC3 EN	ADC2 EN	ADC1 EN	Res.	Res.	USART6 EN	USART1 EN	Res.	Res.	TIM8 EN	TIM1 EN
	rw	rw	rw	rw	rw	rw	rw			rw	rw			rw	rw

(c) RCC_APB2ENR para activar los relojes, entre otros, de algunos *timers*.

Figura 5.3: Registros de activación de relojes de GPIO y temporizadores.

5.2. Temporizadores (timers)

Los temporizadores o *timers* son componentes muy importantes en todos los microcontroladores. Sirven para llevar la cuenta de unidades de tiempo para sí

mismo, nuestro programa, u otros periféricos. Un *timer* es, fundamentalmente, un **contador independiente que cuenta el número de ciclos de una fuente de reloj**. Como hemos visto en la Figura 5.2, los *timers* funcionan en base a una fuente de reloj que puede venir de los buses de reloj `APB1` y `APB2`, que son señales preescaladas del `SYSCLK`; no obstante, todavía pueden ser muy rápidas y se pueden preescalar individualmente cada uno de los *timers*.

Los *timers* tienen muchas aplicaciones, como por ejemplo [4]:

- Captura de entrada (*input capture*): Medir la duración de una señal digital de entrada para inferir su frecuencia o periodo.
- Generar señales PWM para controlar dispositivos.
- Generar interrupciones periódicas.
- Generar pulsos a una frecuencia deseada.

El **STM32F446RE** tiene 14 *timers* de 16 bits. De ellos, 10 son de propósito general: `TIM2` (extensible a 32 bits), `TIM3`, `TIM4`, `TIM5` (extensible a 32 bits), y `TIM9` a `TIM14`. Además, tiene 2 *timers* de control avanzado: `TIM1` y `TIM8`. Por último, cuenta con 2 *timers* básicos: `TIM6` y `TIM7`. Las características de todos ellos se muestran en la Figura 5.4 (ver *"Table 6. Timer feature comparison"* del *datasheet*). Los *timers* pueden contar en sentido ascendente, y/ o descendente.

Los *timers* funcionan en paralelo a la CPU, por lo que el programa principal no debe estar atento a las tareas de temporización hasta que ocurra algún evento que queramos gestionar. Casi todos los *timer* tienen canales (ver Figura 5.4 y Figura 5.5), y cada canal puede implementar una funcionalidad distinta. Además, los *timers* pueden conectarse a otros módulos del procesador —como el ADC u otros *timers*— y servir de sistema de temporización de los mismos.

La forma de trabajar con los *timers* es, generalmente, como sigue:

1. Se activa la fuente de reloj del *timer*.
2. Se configura la funcionalidad.
3. Se configuran las interrupciones o eventos.
4. Por último, se habilita el *timer*.

Los contadores de los *timers* (registros `CNT`) tienen 3 modos de contar: creciente *up-counting*, decreciente *down-counting*, y centrado *center-aligned counting*. En el modo creciente, el contador cuenta desde 0 hasta el valor del registro de auto-recarga (`ARR`). En el modo decreciente, el contador cuenta desde el valor del registro de auto-recarga

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/ compare channels	Complementary output	Max interface clock (MHz)	Max timer clock (MHz)(1)
Advanced-control	TIM1, TIM8	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	Yes	90	180
General purpose	TIM2, TIM5	32-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	45	90/180
	TIM3, TIM4	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	45	90/180
	TIM9	16-bit	Up	Any integer between 1 and 65536	No	2	No	90	180
	TIM10, TIM11	16-bit	Up	Any integer between 1 and 65536	No	1	No	90	180
	TIM12	16-bit	Up	Any integer between 1 and 65536	No	2	No	45	90/180
	TIM13, TIM14	16-bit	Up	Any integer between 1 and 65536	No	1	No	45	90/180
Basic	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	Yes	0	No	45	90/180

Figura 5.4: Características de los temporizadores del **STM32F446RE**.

(`ARR`) hasta 0. En el modo centrado, el contador cuenta desde 0 hasta el valor del registro de auto-recarga (`ARR`), y luego de vuelta a 0.

El contador (`CNT`) genera una onda de sierra periódica (o triangular en el modo centrado). El periodo del *timer* se controla por la frecuencia de reloj del *timer* (f_{CLK}) y el valor del registro de auto-recarga (`ARR`). Pero la fuente de reloj que alimenta al contador `CNT` del *timer* se puede ralentizar por un factor llamado **preescaler** (registro `PSC`), lo que permite contar tiempos más largos. El periodo de un *timer* es, en su forma más general, como se muestra en la (5.1).

$$T_{TIMER} = \frac{1}{f_{TIMER}} = \frac{PSC + 1}{f_{CLK}} \cdot (ARR + 1) \tag{5.1}$$

5.2.1. Registros de los timers

Cada temporizador **x** tiene varios registros de control y de estado que nos son de interés. La Figura 5.5 muestra un diagrama de bloques de un *timer* de propósito general en el que aparecen algunos de dichos registros que se explican a continuación.

5.2.2. Control register (TIMx_CR1)

Configura y habilita el *timer* **x**. En la Figura 5.6a se muestra este registro. A destacar **el bit `ARPE` que activa la pre-recarga, usada con PWM.** Para que el

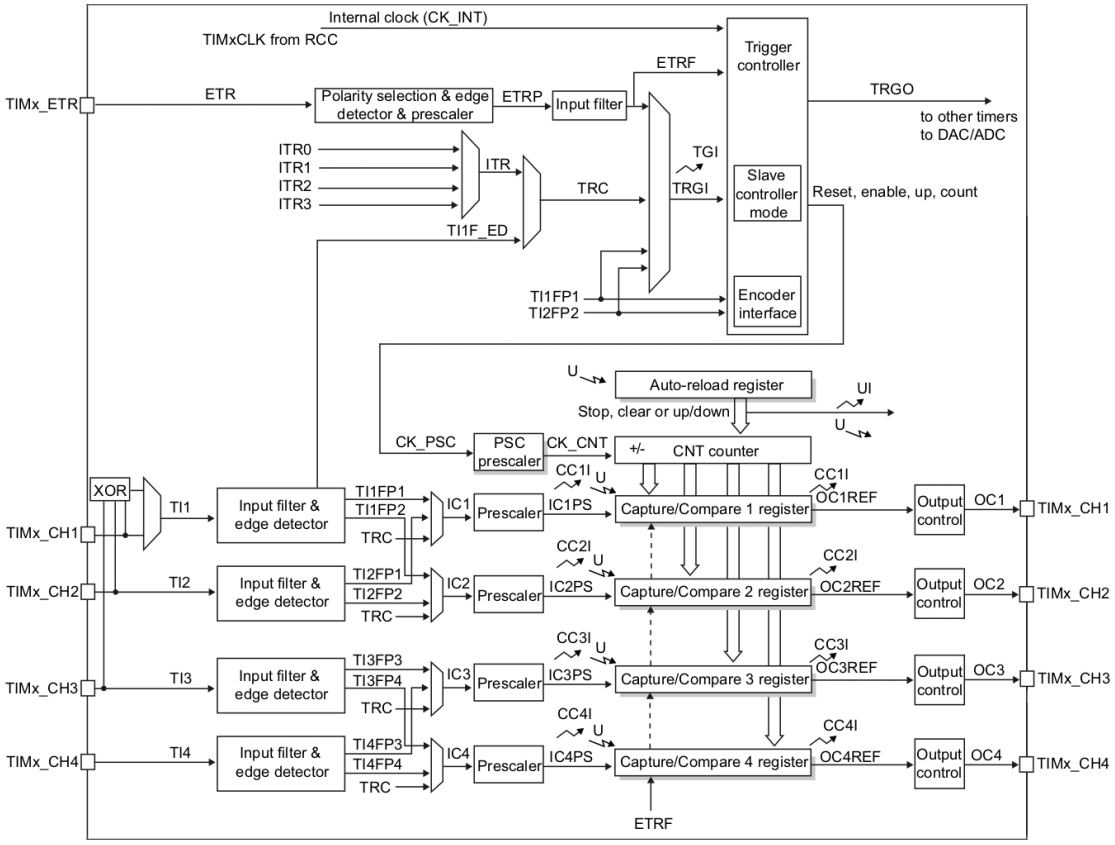


Figura 5.5: Diagrama de bloques de un timer de propósito general.

contador cuente, ha de estar habilitado ('1' en bit `CEN`).

5.2.3. DMA/Interrupt Enable Register (TIMx_DIER)

Habilita las interrupciones. El bit `UIE` es común a todos los *timers*, gestionen o no `DMA`.

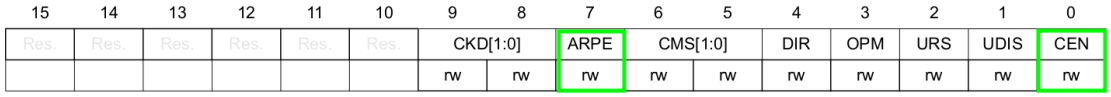
5.2.4. Capture/Compare mode register (TIMx_CCMR1 y TIMx_CCMR2)

TIMx_CCMR1 es el registro para configurar los canales 1 y 2 de los *timers*. El registro **TIMx_CCMR2** es para configurar los canales 3 y 4.

Los canales de un *timer* pueden utilizarse en entrada (modo captura de entrada, *input capture*) o en salida (modo comparación). La dirección de un canal se define configurando los bits `CC1S` o `CC2S` (por defecto definidos como salida, ver Figura 5.6c). **Todos los demás bits de este registro tienen una función diferente en modo de entrada y en modo de salida. Cuando el canal está configurado como salida, tenemos que fijarnos en los nombres de los bits que empiezan por `OCxx`; cuando está configurado como entrada empiezan por `ICxx`.** Para entender

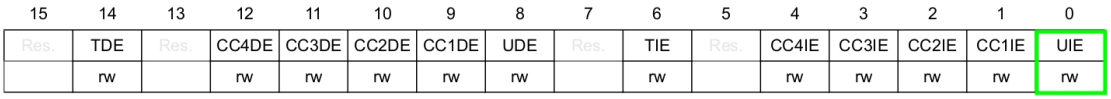
en detalle el funcionamiento de los bits, ver el apartado *"17.4.7 TIMx capture/compare mode register 1 (TIMx_CCMR1)"* del *reference manual*.

En la Figura 5.6c se destacan en verde los conjuntos de bits que usaremos para configuración el canal en modo PWM: `OCxM` para elegir PWM y `OCxPE`, eligiendo los bits `CCxS` en dirección de salida. En rojo se resaltan los bits `CCxS` para elegir la fuente de entrada en modo de captura.



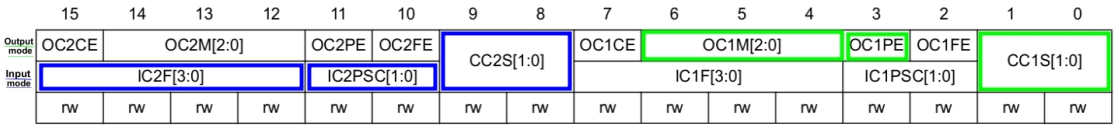
Bits 15:10 Reserved, must be kept at reset value.
Bits 9:8 **CKD[1:0]**: Clock division
Bit 7 **ARPE**: Auto-reload preload enable
Bits 6:5 **CMS[1:0]**: Center-aligned mode selection
Bit 4 **DIR**: Direction
Bit 3 **OPM**: One pulse mode
Bit 2 **URS**: Update request source
Bit 1 **UDIS**: Update disable
Bit 0 **CEN**: Counter enable

(a) TIMx_CR1 para `TIM1-TIM5` y `TIM8`.



Bit 15 Reserved, must be kept at reset value.
Bit 14 **TDE**: Trigger DMA request enable
0: Trigger DMA request disabled.
1: Trigger DMA request enabled.
Bit 13 Reserved, always read as 0
Bit 12 **CC4DE**: Capture/Compare 4 DMA request enable
0: CC4 DMA request disabled.
1: CC4 DMA request enabled.
Bit 11 **CC3DE**: Capture/Compare 3 DMA request enable
Bit 10 **CC2DE**: Capture/Compare 2 DMA request enable
Bit 9 **CC1DE**: Capture/Compare 1 DMA request enable
Bit 8 **UDE**: Update DMA request enable
Bit 7 Reserved, must be kept at reset value.
Bit 6 **TIE**: Trigger interrupt enable
0: Trigger interrupt disabled.
1: Trigger interrupt enabled.
Bit 5 Reserved, must be kept at reset value.
Bit 4 **CC4IE**: Capture/Compare 4 interrupt enable
0: CC4 interrupt disabled.
1: CC4 interrupt enabled.
Bit 3 **CC3IE**: Capture/Compare 3 interrupt enable
Bit 2 **CC2IE**: Capture/Compare 2 interrupt enable
Bit 1 **CC1IE**: Capture/Compare 1 interrupt enable
Bit 0 **UIE**: Update interrupt enable

(b) TIMx_DIER para `TIM2-TIM5`.



Output compare mode
Bit 15 **OC2CE:** Output compare 2 clear enable
Bits 14:12 **OC2M[2:0]**: Output compare 2 mode
Bit 11 **OC2PE**: Output compare 2 preload enable
Bit 10 **OC2FE**: Output compare 2 fast enable
Bits 9:8 **CC2S[1:0]**: Capture/Compare 2 selection
Bit 7 **OC1CE:** Output compare 1 clear enable
Bits 6:4 **OC1M**: Output compare 1 mode
Bit 3 **OC1PE**: Output compare 1 preload enable
Bit 2 **OC1FE**: Output compare 1 fast enable
Bits 1:0 **CC1S**: Capture/Compare 1 selection

Input capture mode
Bits 15:12 **IC2F**: Input capture 2 filter
Bits 11:10 **IC2PSC[1:0]**: Input capture 2 prescaler
Bits 9:8 **CC2S**: Capture/compare 2 selection
Bits 7:4 **IC1F**: Input capture 1 filter
Bits 3:2 **IC1PSC**: Input capture 1 prescaler
Bits 1:0 **CC1S**: Capture/Compare 1 selection

(c) TIMx_CCMR1 para TIM2-TIM5.

Figura 5.6: Registros TIMx_CR1, TIMx_DIER y TIMx_CCMR1 (ver detalles en el *reference manual*).

5.2.5. Capture/Compare Enable Register (TIMx_CCER)

Registro para habilitar o deshabilitar el *input capture/ output compare*. Este registro controla todos los canales del *timer*. Lo usaremos para habilitar o deshabilitar el PWM, por ejemplo.

5.2.6. Counter (TIMx_CNT)

Este registro es el contador de ciclos. Tiene 16 o 32 bits, dependiendo del *timer*. Con 16 bits, este registro puede contar $2^{16} = 65536$ *ciclos*, desde el valor 0 hasta el 65535. Este contador se incrementa con la salida del prescaler (ver Figura 5.5). **El contenido de este contador se compara con el valor guardado en el registro de *auto-reload*, y se genera una interrupción cuando coinciden.**

5.2.7. Prescaler (TIMx_PSC)

El contenido de este registro guarda el valor por el que se va a dividir el reloj del temporizador. Se permite dividir la frecuencia de reloj por cualquier valor entre 1 y $2^{16} - 1(65535)$. El valor de prescaler que se elija hace que la frecuencia a la que el registro **TIMx_CNT** se cambie sea $\frac{f_{CLK_PSC}}{PSC+1}$, por lo que un prescaler de 0, hace que la frecuencia de **TIMx_CNT** sea igual que la frecuencia de reloj del `APB1` o `APB2`.

5.2.8. Auto-reload Register (TIMx_ARR)

Este registro contiene el valor que se cargará en el registro interno de auto-reload. **Lo usaremos en PWM. Aunque no se use, hay que escribir algún valor no nulo, o de lo contrario el contador CNT se bloqueará**.

5.2.9. Event Generation Register (TIMx_EGR)

Registro usado para generar eventos de forma manual (ver Figura 5.7e). **Siempre activaremos el bit `UG` (`'1'`) para generar un evento que reinicia los valores de los registros internos.** El hardware lo bajará a `'0'`.

5.2.10. Capture/Compare Register TIMx_CCRx

Este registro está descrito para cada uno de los 4 canales. Cuando la dirección del canal está como salida (*output compare*), el registro, **en modo PWM, contiene el valor del ancho del pulso del canal, quedando el ciclo de trabajo (duty cycle) definido por los valores de ARR y CCRx** como:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

Bit 15 **CC4NP**: *Capture/Compare 4 output Polarity.*
Bit 14 Reserved, must be kept at reset value.
Bit 13 **CC4P**: *Capture/Compare 4 output Polarity.*
Bit 12 **CC4E**: *Capture/Compare 4 output enable.*
Bit 11 **CC3NP**: *Capture/Compare 3 output Polarity.*
Bit 10 Reserved, must be kept at reset value.
Bit 9 **CC3P**: *Capture/Compare 3 output Polarity.*
Bit 8 **CC3E**: *Capture/Compare 3 output enable.*
Bit 7 **CC2NP**: *Capture/Compare 2 output Polarity.*
Bit 6 Reserved, must be kept at reset value.
Bit 5 **CC2P**: *Capture/Compare 2 output Polarity.*
Bit 4 **CC2E**: *Capture/Compare 2 output enable.*
Bit 3 **CC1NP**: *Capture/Compare 1 output Polarity.*
Bit 2 Reserved, must be kept at reset value.
Bit 1 **CC1P**: *Capture/Compare 1 output Polarity.*
Bit 0 **CC1E**: *Capture/Compare 1 output enable.*

(a) TIMx_CCER para `TIM2-TIM5`.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **CNT[15:0]**: Counter value

(b) TIMx_CNT para `TIM2-TIM5`.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **PSC[15:0]**: Prescaler value
The counter clock frequency CK_CNT is equal to f_{CK_PSC} / (PSC[15:0] + 1).
PSC contains the value to be loaded in the active prescaler register at each update event.

(c) TIMx_PSC para `TIM2-TIM5`.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **ARR[15:0]**: Auto-reload value
ARR is the value to be loaded in the actual auto-reload register.
The counter is blocked while the auto-reload value is null.

(d) TIMx_ARR para `TIM2-TIM5`.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TG	Res.	CC4G	CC3G	CC2G	CC1G	UG
									w		w	w	w	w	w

Bits 15:7 Reserved, must be kept at reset value.
Bit 6 **TG**: Trigger generation
Bit 5 Reserved, must be kept at reset value.
Bit 4 **CC4G**: Capture/compare 4 generation
Bit 3 **CC3G**: Capture/compare 3 generation
Bit 2 **CC2G**: Capture/compare 2 generation
Bit 1 **CC1G**: Capture/compare 1 generation
Bit 0 **UG**: Update generation
This bit can be set by software, it is automatically cleared by hardware.
0: No action
1: Re-initialize the counter and updates the registers.
Note that the prescaler counter is cleared too

(e) TIMx_EGR para `TIM2-TIM5`.

Figura 5.7: Registros TIMx_CCER, TIMx_CNT, TIMx_PSC, TIMx_ARR y TIMx_EGR (ver detalles en el *reference manual*).

$$DC_{PWM} = \frac{T_{ON}}{T_{PWM}} \cdot 100 = \frac{CCRx}{ARR + 1} \cdot 100\ [\%] \tag{5.2}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CCR1[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 **CCR1[31:16]**: High Capture/Compare 1 value (on TIM2 and TIM5).
Bits 15:0 **CCR1[15:0]**: Low Capture/Compare 1 value

(a) TIMx_CCR1 para `TIM2-TIM5`.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CC4OF	CC3OF	CC2OF	CC1OF	Res.	Res.	TIF	Res.	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Bits 15:13 Reserved, must be kept at reset value.
Bit 12 **CC4OF**: Capture/Compare 4 overcapture flag
0: No overcapture has been detected
1: The counter value has been captured in TIMx_CCR1 register while CC1IF flag was already set
Bit 11 **CC3OF**: Capture/Compare 3 overcapture flag
Bit 10 **CC2OF**: Capture/compare 2 overcapture flag
Bit 9 **CC1OF**: Capture/Compare 1 overcapture flag
Bits 8:7 Reserved, must be kept at reset value.

Bit 6 **TIF**: Trigger interrupt flag
Bit 5 Reserved, must be kept at reset value.
Bit 4 **CC4IF**: Capture/Compare 4 interrupt flag
Bit 3 **CC3IF**: Capture/Compare 3 interrupt flag
Bit 2 **CC2IF**: Capture/Compare 2 interrupt flag
Bit 1 **CC1IF**: Capture/compare 1 interrupt flag
Bit 0 **UIF**: Update interrupt flag
0: No update occurred.
1: Update interrupt pending.

(b) TIMx_SR para `TIM2-TIM5`.

Figura 5.8: Registros TIMx_CCR1 y TIMx_ST (ver detalles en el *reference manual*).

Si la dirección del canal está como entrada (*input capture*), en este registro es donde se almacena el valor que haya en `CNT` cuando ocurra el evento externo.

5.2.11. Status Register (TIMx_SR)

Es el registro de estado del *timer*. Indica con distintos bits si ha ocurrido un evento en dicho canal tanto si está en dirección de salida (`CCyOF`), como entrada (`CCyIF`). **Cuando solo usamos un canal, podemos mirar el bit `UIF` unicamente. Este *flag* (ver Figura 5.8b) nos indica que ha sucedido un evento —no especifica canal, para eso están los bits correspondientes— y que está sin atender. Cuando atendamos la interrupción, podemos limpiar este y/ o el bit correspondiente al canal escribiendo un `'0'` en el bit.**

5.2.12. Ejemplo: timer para interrupción periódica

A continuación vamos a ver un ejemplo con los pasos que deberíamos seguir para configurar el *timer* `TIM2` para que genere interrupciones cada 1000 *ms*, y encienda el LED `LD2` en su ISR —no nos preocupamos ahora de quién apagaría el LED—. La parte relacionada con interrupciones la entenderá mejor en el Capítulo 6; no se preocupe ahora por ella.

La Figura 5.9 muestra un esquema simplificado de cómo opera un temporizador [4].

Para saber qué valores debemos colocar en los registros de *prescaler* (`PSC`) y *auto-reload* (`ARR`) tenemos la Ec. 5.3.

$$ARR = \frac{f_{CLK} \cdot T_{INTERR}}{PSC + 1} - 1 \tag{5.3}$$

donde f_{CLK} es la frecuencia de reloj del *timer* en Hz, T_{INTERR} es el periodo de interrupción en segundos, y `PSC` es el valor del *prescaler* (de 0 a 65,535). Como se puede comprobar, no existe una solución única, sino que dependiendo de el valor de `PSC` que elijamos, tendremos un valor de `ARR`. Se prefieren valores que den resultados de `ARR` enteros y cometer el mínimo error en el periodo de interrupción.

En nuestro ejemplo, para una frecuencia de `SYSCLK` de 16 MHz usando el oscilador interno `HSI`, si elegimos el valor $PSC = 0$, quedaría:

$$ARR = \frac{16 \cdot 10^6 \cdot 1}{0 + 1} - 1 = 15{,}999{,}999 \tag{5.4}$$

Este valor no se puede representar con 16 bits, pues el máximo valor que podemos poner es $ARR_{max} = 2^{16} - 1 = 65{,}535$. Con una sencilla cuenta, podemos ver cuál sería el valor mínimo de *prescaler* (PSC_{min}) que podemos poner:

$$PSC_{min} = \left\lceil \frac{f_{CLK} \cdot T_{INTERR}}{ARR_{max} + 1} - 1 \right\rceil \rightarrow PSC_{min} = \left\lceil \frac{16 \cdot 10^6 \cdot 1}{65{,}536} - 1 \right\rceil = 245 \tag{5.5}$$

Cualquier valor menor que este haría sobrepasarse a `ARR`. Por ejemplo, elegimos $PSC = 399$, quedando $ARR = 39{,}999$. Así pues, cuando el contador `CNT` llegue a dicho valor, el *timer* generará una interrupción.

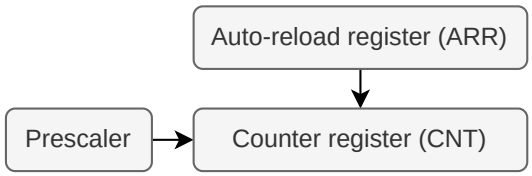


Figura 5.9: Esquema de operación de un temporizador.

En `C` —con el estándar y máscaras de CMSIS—, el código para el problema propuesto podría quedar como sigue. Para la configuración de la GPIO creamos la función `gpio_config()`.

```
1 #define BIT5_MASK 0x0020 /* Binario: 0b0010 0000 */
2 #define MODER5_AS_OUTPUT 0x0400 /* Binario: 0b0100 0000 0000*/
3 #define PUPD5_MASK  0x0C00 /* Binario: 0b1100 0000 0000*/
4
5 void gpio_config()
6 {
7     /* Primero, habilitamos siempre el reloj del periferico */
8     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; /* GPIOA clock enable */
```

```
9
10      /* LED LD2 como salida sin pull en PA5 */
11      GPIOA->MODER |= MODER5_AS_OUTPUT;
12      GPIOA->PUPDR &= ~PUPD5_MASK; /* Limpiamos y tambien queda definido sin pull */
13  }
```

Para la configuración del *timer* creamos la función `timer_setup()`.

```
1   void timer_setup()
2   {
3       /* Primero, habilitamos siempre el reloj del timer */
4       RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; /* TIM2 con APB1. */
5
6       /* Temporizacion */
7       /* 1) Fuente de reloj: interno (bits CKD=00) */
8       TIM2->CR1 = 0x0000; /* Aunque es el valor por defecto */
9
10      /* 2) Deshabilita el contador (no es indispensable). */
11      TIM2->CR1 &= ~TIM_CR1_CEN;
12
13      /* 3) Aseguramos inicio del contador a 0 */
14      TIM2->CNT = 0;
15
16      /* 4) Cargamos el autorreload */
17      TIM2->ARR = 39999;
18
19      /* 5) Cargamos el prescaler */
20      TIM2->PSC = 399;
21
22      /* 6) (IMPORTANTE) Re-inicializa el contador y actualiza los registros.
            IMPORTANTE realizarlo despues de cargar ARR y PSC */
23      /* Los valores de PSC y ARR estan en los registro de precarga. Para cargarlos
            en los registros activos, necesitamos que se actualice un evento. Esto
            podemos hacerlo manualmente actualizando el registro TIMx->EGR, o esperar a
             que pase el timer. */
24      TIM2->EGR = TIM_EGR_UG; /* Genera evento de actualizacion */
25
26      /* Interrupciones */
27      /* 7) Limpiamos las interrupciones (si hubiese) */
28      TIM2->SR = ~TIM_SR_UIF;
29
30      /* 8) Habilitamos las interrupciones en el registro */
31      TIM2->DIER |= TIM_DIER_UIE; /* Interrumpe al acabar de contar */
32
33      /* 9) Elegimos prioridad de interrupciones y las habilitamos globalmente en el
            NVIC */
34      NVIC_SetPriority(TIM2_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(), 1,
            0)); /* Priority 1, sub-priority 0 */
35      NVIC_EnableIRQ(TIM2_IRQn); /* Habilita interrupcion */
36
37      /* 10) Habilita timer para empezar a contar (podria hacerse en otro momento) */
38      TIM2->CR1 |= TIM_CR1_CEN;
39  }
```

La zona de memoria en la que se encuentra la ISR del `TIM2` está asociada a la interrupción `TIM2_IRQn` (ver fichero `startup_stm32f446xx.s`).

```
1   /* La ISR debe tener este nombre obligatoriamente */
2   void TIM2_IRQHandler(void)
3   {
```

```
4 ||     GPIOA->ODR |= BIT5_MASK; /* Encender LED */
5 ||     TIM2->SR &= ~TIM_SR_UIF; /* Limpiar flag de interrupcion del timer */
6 || }
```

5.2.13. Ejemplo: timer para PWM

A continuación vamos a ver un ejemplo de cómo configurar un *timer* en modo *output compare* para generar una señal PWM con un ciclo de trabajo (*duty cycle*) $DC_{PWM} = 50\,\%$ en el LED `LD2`. Esto permitirá **que el LED luzca con una intensidad menor, proporcional al ciclo de trabajo del PWM.**

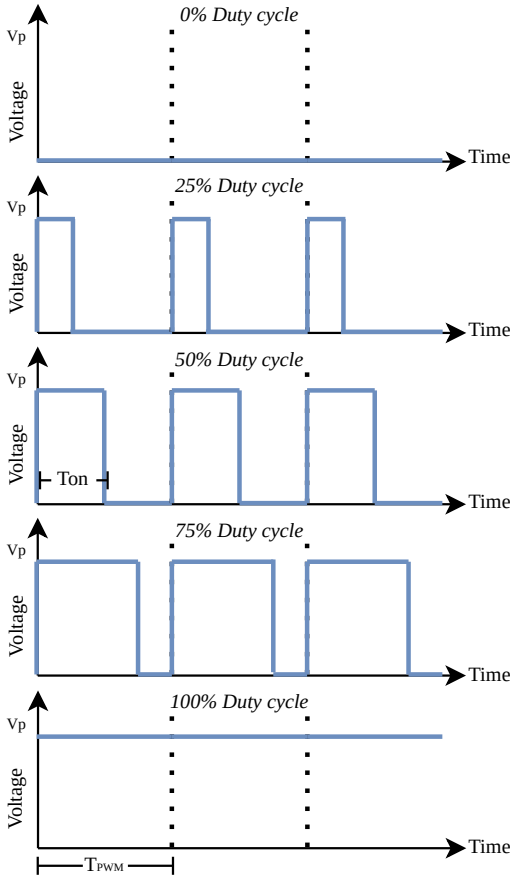


Figura 5.10: Señales PWM.

La Figura 5.10 nos recuerda cómo es la forma de onda de una señal PWM para distintos ciclos de trabajo. De esta figura tenemos que **destacar el periodo T_{PWM} de la señal PWM, y el tiempo en el que está activa la señal** T_{ON}, cuya relación nos marca el *duty cycle* como se muestra en la Ec. 5.2. Si en este momento tiene dudas sobre PWM, le conviene repasar los conceptos teóricos estudiados en otras asignaturas.

Para un $DC_{PWM} = 50\,\%$ deseado, despejando, tenemos que $T_{ON} = 0{,}5 * T_{PWM}$, por lo que, a nivel de registros: $CCRx = 0{,}5 * (ARR + 1)$. Ahora solo nos falta saber

cuánto es el periodo del *timer* del PWM T_{PWM}, y el *timer* que vamos a utilizar.

El periodo de PWM lo debemos establecer nosotros. Quizás se esté preguntando cómo una señal que enciende y que apaga un LED con un determinado *duty cycle*, vaya a hacer que lo veamos brillar a distintas intensidades. Aquí entra en juego el valor del periodo de PWM y la capacidad de integración de la visión humana. Podemos decir que la visión humana puede procesar entre $25 - 50$ imágenes por segundo, por lo que, para no notar el parpadeo del LED, se recomienda tener un periodo de PWM no menor a 50 Hz o 20 ms. Por lo que ya sabemos que $T_{PWM} = 20\ ms$.

Hemos elegido que el LED brille al 50 % de su intensidad, *i.e.*, que la tensión media en el puerto `PA5` sea 1,65 V (la mitad de 3,3 V, que es la salida máxima de dicho pin). Habrá recordado que el valor medio de una señal PWM $V_{avg_{PWM}}$ es el de la integral de una señal cuadrada periódica...

$$V_{avg_{PWM}} = \frac{1}{T_{PWM}} \int_0^{T_{PWM}} PWM(t)\, dt = \frac{1}{T_{PWM}} (T_{ON} \cdot V_p) = DC_{PWM} \cdot V_p\ [V] \quad (5.6)$$

Lo que confirma **que el LED luce con una intensidad proporcional al ciclo de trabajo del PWM.** Esta configuración podremos comprobarla midiendo con el osciloscopio en pin `D13` del *header* de la placa **Nucleo-STM32F446RE**.

Ahora vamos a elegir el *timer*. No podemos usar cualquier *timer* que queramos para controlar la GPIO del `LD2`. Recordemos lo ya visto en la Sección 4.2.3. Sabemos que el LED de la placa **Nucleo-STM32F446RE** está conectado al puerto `PA5`. Para saber qué *timer* y canal tenemos que usar, miramos la Figura 1.11b, y vemos que en ese pin 

tenemos que usar el `PWM2/1`, *i.e.*, *timer* `TIM2`, canal `1`, por lo que el registro `CCRx` a usar es el `CCR1`. Si nos fijamos en dicha figura y en la del *header* izquierdo, vemos que ese mismo *timer* y canal son usados en los puertos `PA0` y `PB8` también.

Por supuesto, tenemos que configurar dicho pin como función alternativa para PWM como se hizo en la Sección 4.2.3.

Como hicimos en el ejercicio anterior, calculamos los registros `ARR`, `PSC` con la Ec. 5.3. Ahora también calculamos `CCR1`. Existen múltiples combinaciones posibles, una podría ser:

$$PSC_{min} = \left\lceil \frac{f_{CLK} \cdot T_{PWM}}{ARR_{max} + 1} - 1 \right\rceil \rightarrow PSC_{min} = \left\lceil \frac{16 \cdot 10^6 \cdot 20 \cdot 10^{-3}}{65{,}536} - 1 \right\rceil = 4 \quad (5.7)$$

Usando la Ec. 5.3, para $PSC = PSC_{min}$:

$$ARR = \frac{f_{CLK} \cdot T_{PWM}}{PSC + 1} - 1 = \frac{16 \cdot 10^6 \cdot 20 \cdot 10^{-3}}{4 + 1} - 1 = 63999 \quad (5.8)$$

Por lo que:

$$CCR1 = 0{,}5 \cdot (ARR + 1) = 32000 \quad (5.9)$$

NOTA

Si al hacer la división de la Ec. 5.3 obtenemos números decimales, estaremos cometiendo una imprecisión. Intentaremos buscar un valor de `PSC` válido que nos de una división entera. Si no fuese posible, elegiríamos el que implique menos imprecisión en el redondeo.

En `C` —con el estándar y máscaras de CMSIS—, el código para el problema propuesto podría quedar como sigue. Para la configuración de la GPIO creamos la función `gpio_config()`. No debemos olvidar que hay que definir el pin con función alternativa (ya no es *output*).

```
1  #define LINEA_5 5
2  #define BIT5_MASK 0x0020 /* Binario: 0b0010 0000 */
3  #define MODER5_AS_ALTERNATE 0x0800 /* Binario: 0b1000 0000 0000*/
4  #define PUPD5_MASK  0x0C00 /* Binario: 0b1100 0000 0000*/
5
6  #define ALT_FUNC_TIM2 0x01U   /* AF1 segun datasheet */
7  #define BASE_ALT_MASK 0x0F  /* En binario: 0b1111 (4 bits)*/
8  #define AFRL5_MASK (BASE_ALT_MASK << 4*LINEA_5) /* Desplaza los bits 0b1111 a la
       izquierda 4*5=20 posiciones */
9
10 void gpio_config()
11 {
12     /* Primero, habilitamos siempre el reloj del periferico */
13     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; /* GPIOA clock enable */
14
15     /* LED LD2 como funcion alternativa sin pull en PA5 */
16     GPIOA->MODER |= MODER5_AS_ALTERNATE;
17     GPIOA->PUPDR &= ~PUPD5_MASK; /* Limpiamos y tambien queda definido sin pull */
18
19     /* Funcion alternativa AF1 (0x01) en la GPIOA (ver tabla 11 del datasheet) */
20     /* AFRL y AFRH no existen en la estructura GPIOA, sino que son AFR[0] y AFR[1]
           respectivamente */
21     GPIOA->AFR[0] &= ~AFRL5_MASK; /* Limpiamos*/
22     GPIOA->AFR[0] |= (ALT_FUNC_TIM2 << 4*LINEA_5); /* Escribimos valor*/
23 }
```

Para la configuración del *timer* creamos la función `timer_setup()`. **Hay que tener mucho cuidado con elegir bien los registros y los bits. En este caso estamos trabajando con el canal 1.**

```c
#define TIM_AS_PWM1_MASK 0x0060 /* Binario: 0b0110 0000 */

void timer_setup()
{
    /* Primero, habilitamos siempre el reloj del timer */
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; /* TIM2 con APB1. */

    /* Temporizacion */
    /* 1) Fuente de reloj: interno (bits CKD=00) */
    TIM2->CR1 = 0x0000; /* Aunque es el valor por defecto */

    /* 2) Deshabilita el contador (no es indispensable). */
    TIM2->CR1 &= ~TIM_CR1_CEN;

    /* 3) Aseguramos inicio del contador a 0 */
    TIM2->CNT = 0;

    /* 4) Cargamos el autorreload */
    TIM2->ARR = 63999;

    /* 5) Cargamos el prescaler */
    TIM2->PSC = 4;

    /* 6) (IMPORTANTE) Re-inicializa el contador y actualiza los registros.
        IMPORTANTE que este despues de cargar ARR y PSC */
    /* Los valores de PSC y ARR estan en los registro de precarga. Para cargarlos
        en los registros activos, necesitamos que se actualice un evento. Esto
        podemos hacerlo manualmente actualizando el registro TIMx->EGR, o esperar a
         que pase el timer. */
    TIM2->EGR = TIM_EGR_UG; /* Genera evento de actualizacion */

    /* PWM. CANAL 1 */
    /* 7) Deshabilitar el output compare para hacer modificaciones (no es
        indispensable) */
    TIM2->CCER &= ~TIM_CCER_CC1E;

    /* 8) Habilitamos modo PWM y habilitamos el preload */
    TIM2->CCMR1 |= TIM_AS_PWM1_MASK; /* Modo PWM 1 */
    TIM2->CCMR1 |= TIM_CCMR1_OC1PE;  /* Habilita el preload */

    /* 9) Ancho del pulso de PWM */
    TIM2->CCR1 = 32000;

    /* 10) Habilita output compare para empezar a contar (podria hacerse en otro
        momento) */
    TIM2->CCER |= TIM_CCER_CC1E;

    /* 11) Habilita timer para empezar a contar (podria hacerse en otro momento) */
    TIM2->CR1 |= TIM_CR1_CEN;
}
```

Ya tenemos las capacidades para generar interrupciones internas con *timers* y modulaciones PWM. Podríamos, incluso, cambiar el `CCRx` del PWM en la ISR de otro *timer* para que la intensidad sea variable con el tiempo, o hacer que dependa de un botón, *etc.*

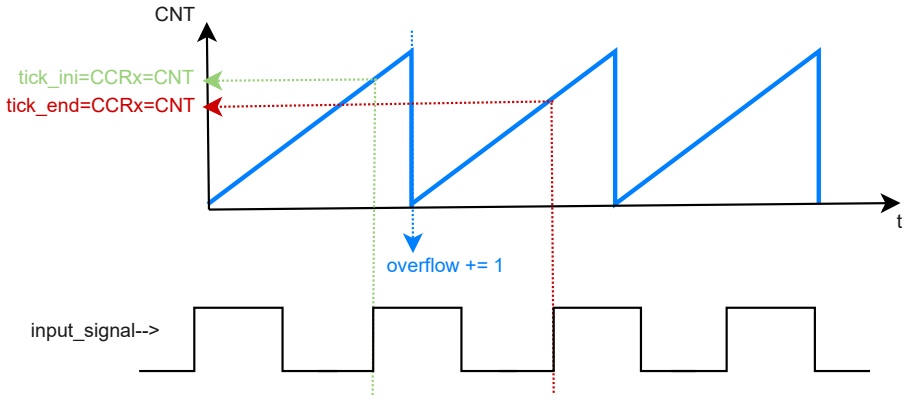


Figura 5.11: Señal de entrada para medida por *input capture* por flancos de subida.

5.2.14. Ejemplo: captura de entrada (input capture)

En esta ocasión vamos a ver cómo configurar un temporizador en modo captura de entrada (*input capture*) para que nos permita medir la duración de una señal digital a la entrada de un pin del microcontrolador. La señal puede ser una onda cuadrada de la que pudiésemos calcular su frecuencia (o periodo), o un pulso resultante de la salida de algún HW como un botón, un sensor, *etc.* La idea es que el temporizador capture el valor del contador en el momento en que se producen dos flancos en la señal de entrada y mida los tiempos entre ellos: (i) dos flancos de subida, (ii) dos de bajada, o (iii) uno de subida y uno de bajada.

Si medimos dos flancos de igual polaridad obtendremos el periodo de la señal, si medimos dos flancos de distinta polaridad obtendremos el ancho del pulso.

Cuando sucede el evento indicado (flanco), salta la ISR del temporizador, y en ella podemos leer el valor del contador en ese instante. El valor del contador `CNT` se guarda en el registro `CCRx` (*Capture/Compare Register*) del temporizador.

En un caso general, no sabremos qué frecuencia tiene la señal de entrada, y deberemos tener en cuenta que puede haber desbordamientos (*overflow*) del contador. En tal caso, habría que tener en cuenta el valor del contador en el momento de la interrupción, y el número de desbordamientos que ha habido. Tendremos que guardar el número de desbordamientos y el valor del contador en el momento de la interrupción, y hacer los cálculos pertinentes. Una pequeña fórmula para calcular el tiempo entre dos flancos de subida como en se muestra en la Figura 5.11, sería:

$$time = (tick_{end} + overflows \cdot (ARR + 1) - tick_{ini}) \cdot timer_{clk_{resolution}} \tag{5.10}$$

Donde $tick_{ini}$ y $tick_{end}$ son los valores del contador en el momento de los flancos,

$overflows$ es el número de desbordamientos, ARR es el valor del auto-reload, y $timer_{clk_{resolution}}$ es la resolución del reloj que alimenta al temporizador, y que depende del prescaler que hayamos elegido. En este caso, el valor de *time* será el tiempo entre los dos flancos de subida.

En la Figura 5.6c se destacan en azul los conjuntos de bits que usaremos para configuración el canal en modo captura de entrada: `ICxF` para el filtro, `OCxPSC` para el prescaler de entrada, y `CCxS` para elegir la fuente de entrada.

NOTA

- **Filtro de entrada** (`ICxF`): es un filtro digital que se aplica a la señal de entrada para evitar que se capturen ruido o señales espúreas. Si no se necesita, se deshabilita poniendo el campo a `0`.
- **Prescaler de entrada** (`ICxPSC`): también llamado divisor de frecuencia, se aplica a la señal de entrada para reducir su frecuencia. Si no se necesita, se deshabilita poniendo el campo a `0`.
- **Fuente de entrada** (`CCxS`): para saber si tu señal va a TI1 o TI2 (si usas CCMR1), o va a TI3 o TI4 (si usas CCMR2), lo primero es consultar la tabla de mapeo de pines (*"Table 11. Alternate function"*) del *datasheet*. Ahí veremos qué función alternativa corresponde a nuestra GPIO y el canal asociado (1, 2, 3, o 4). Ese canal es el que se asocia de manera natural a la entrada TIx, y por eso configuraremos típicamente el campo CCxS con el valor `0b01` (ICx mapeado en TIx), porque es el que le corresponde de manera natural. Dicho de otro modo, *"channel X"* de un *timer* se llama físicamente `TIx`, y así lo mapearemos habitualmente. Las otras opciones se usan en casos especiales:
  - Mapear en el otro canal TIx (CCxS = `0b10`) se emplea cuando quieres capturar la misma señal que está entrando por la patilla del otro canal.
  - Mapear en TRC (CCxS = `0b11`) se usa si en lugar de capturar una señal de un pin, necesitas capturar una señal interna (*trigger*) de un *timer* configurada por el bit `TS` en `TIMx_SMCR`.

Para configurar el temporizador en modo captura de entrada, debemos seguir los siguientes pasos:

1. Elegimos la GPIO que vamos a usar y miramos la tabla de funciones alternativas para saber qué *timer* y canal debemos usar. Configuramos la GPIO en **modo alternativo** como se hizo en la Sección 4.2.3.

2. Configuramos el temporizador en modo captura de entrada:

   a) Como siempre, lo primero será **habilitar la fuente de reloj** del temporizador para que el módulo esté activo y se pueda configurar. Ello se

hace habilitando el reloj del temporizador en el registro `RCC->APBxENR`.

b) Por seguridad, **deshabilitar el contador** del temporizador si es que está activo. Para ello configuraremos en el registro `CR1` del temporizador el campo `CEN` poniéndolo a `0`.

c) Elegir **la resolución del reloj temporizador** con la que vamos a medir los tiempos. Para ello configuraremos el prescaler (`PSC`).

d) El valor del auto-reload (`ARR`) determina el periodo del temporizador. Para el modo *input capture*, el valor de `ARR` en sí no afecta directamente al proceso de captura, pero sí determina el periodo de desbordamiento del temporizador, lo que puede ser relevante dependiendo de la aplicación. Típicamente queremos asegurar que el temporizador no desborde con demasiadas veces, ya que podría interferir con la medición de la captura de la señal de entrada de manera precisa. **Elegiremos valores altos de `ARR`. Si no tenemos ninguna restricción, podemos dejarlo al máximo (`0xFFFF`).**

e) Por seguridad, haremos que el contador se reinicie cada vez que hay un desbordamiento. Activaremos el **modo de auto-reload** en el temporizador. Para ello configuraremos en el registro `CR1` del temporizador el campo `ARPE` poniéndolo a `1`.

f) Forzaremos la actualización de los registros de `ARR` y `PSC` para que se carguen en los registros activos. Para ello configuraremos en el registro `EGR` del temporizador el campo `UG` poniéndolo a `1`.

g) Configurar la **dirección del canal en modo captura de entrada**. Para ello, configuraremos en el registro `CCMRx` del temporizador **el campo** `CCxS` **—típicamente a** `0b01`—.

h) Deshabilitar el **filtro digital** de la señal de entrada si no lo necesitamos. Para ello configuraremos en el registro `CCMRx` del temporizador el campo `ICxF`. Si no lo necesitamos lo dejaremos a `0`.

i) Configurar el **flanco de captura** que queremos detectar. Para ello configuraremos en el registro `CCER` del temporizador los campos `CCxP` para flancos de subida, `CCxNP` para flancos de bajada, o ambos para ambos flancos.

j) Configurar el **prescaler de entrada** si lo necesitamos. Para ello configuraremos en el registro `CCMRx` del temporizador el campo `ICxPSC`. Si no lo necesitamos lo dejaremos a `0`.

k) Habilitar la **captura de entrada** en el temporizador. Para ello configuraremos en el registro `CCER` del temporizador el campo `CCxE` del

canal que usemos poniéndolo a `1`.

l) Habilitar las **interrupciones** en el temporizador. Para ello configuraremos en el registro `DIER` del temporizador el campo `CCxIE` del canal que usemos poniéndolo a `1`.

m) Activar el **contador** del temporizador si es que es necesario hacerlo en este momento. Para ello configuraremos en el registro `CR1` del temporizador el campo `CEN` poniéndolo a `1`.

n) Elegir la **prioridad de la interrupción** y habilitarla en el Nested Vectored Interrupt Controller (Controlador de Vector de Interrupciones Anidadas) (NVIC). Para ello configuraremos el registro `TIMx_IRQn` con la prioridad deseada y habilitaremos la interrupción en el NVIC.

Vamos a hacer un ejemplo para medir el periodo de una señal cuadrada de 1 kHz que entra por el pin `PA1` de la placa **Nucleo-STM32F446RE**. La señal la podemos conseguir con la salida de test de un osciloscopio, o un generador de señales. La resolución del reloj del temporizador la dejaremos en 1 μs para que sea precisa.

Si miramos la *"Table 11. Alternate function"* del *datasheet*, vemos que el pin `PA1` tiene asociada la función alternativa `AF1` que corresponde al *timer* `TIM5` y al canal `2` (por tanto, mapeado a TI2). Así pues, configuramos la GPIO, lo haremos en la función `gpio_config()`. **Hay que asegurarse que el pin de entrada del microcontrolador no reciba una señal de mayor amplitud que la permitida por el fabricante, o que no tenga un nivel de tensión mayor que el permitido.**

```
1  #define LINEA_1 1 /* Pin PA1 */
2  #define BIT1_MASK 0x02 /* Binario: 0b0000 0010 */
3  #define MODER1_AS_ALTERNATE 0x08 /* Binario: 0b0000 1000 */
4  #define PUPD1_MASK  0x0C /* Binario: 0b0000 1100 */
5  #define ALT_FUNC_TIM5 0x02U   /* AF2 segun datasheet */
6  #define BASE_ALT_MASK 0x0F  /* En binario: 0b1111 (4 bits)*/
7  #define AFRL1_MASK (BASE_ALT_MASK << 4*LINEA_1) /* Desplaza los bits 0b1111 a la
       izquierda 4*1=4 posiciones */
8
9  void gpio_config()
10 {
11     /* Primero, habilitamos siempre el reloj del periferico */
12     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; /* GPIOA clock enable */
13
14     /* Pin PA1 como funcion alternativa sin pull */
15     GPIOA->MODER |= MODER1_AS_ALTERNATE;
16     GPIOA->PUPDR &= ~PUPD1_MASK; /* Limpiamos y tambien queda definido sin pull */
17
18     /* Funcion alternativa AF2 (0x02) en la GPIOA (ver tabla 11 del datasheet) */
19     /* AFRL y AFRH no existen en la estructura GPIOA, sino que son AFR[0] y AFR[1]
           respectivamente */
20     GPIOA->AFR[0] &= ~AFRL1_MASK; /* Limpiamos*/
21     GPIOA->AFR[0] |= (ALT_FUNC_TIM5 << 4*LINEA_1); /* Escribimos valor*/
22 }
```

Para la configuración del *timer* creamos la función `timer_setup()`. **Hay que tener mucho cuidado con elegir bien los registros y los bits. En este caso estamos trabajando con el canal 2.** Le daremos un nivel de prioridad de interrupción de 1, y subprioridad de 0. Con un reloj de 16 MHz, para generar *ticks* de reloj cada 1 μs, el valor de `PSC` para un reloj de sistema `SYSCLK` de 16 MHz sería:

$$PSC = \left\lceil \frac{f_{CLK} \cdot T_{CLK}}{ARR + 1} - 1 \right\rceil \rightarrow PSC = \left\lceil \frac{16 \cdot 10^6 \cdot 1 \cdot 10^{-6}}{0 + 1} - 1 \right\rceil = 15, \tag{5.11}$$

i.e., un prescalado de 16.

Para evitar muchos desbodamientos, elegiremos un valor de `ARR` alto, por ejemplo, el máximo de 16 bits. Por lo que el valor de `ARR` sería $0xFFFF$.

```
1  void timer_setup()
2  {
3      /* Primero, habilitamos siempre el reloj del timer */
4      RCC->APB1ENR |= RCC_APB1ENR_TIM5EN; /* TIM5 con APB1. */
5
6      /* Temporizacion */
7      /* 1) Fuente de reloj: interno (bits CKD=00) */
8      TIM5->CR1 = 0x0000; /* Aunque es el valor por defecto */
9
10     /* 2) Deshabilita el contador */
11     TIM5->CR1 &= ~TIM_CR1_CEN;
12
13     /* 3) Aseguramos inicio del contador a 0 */
14     TIM5->CNT = 0;
15
16     /* 4) Cargamos el autorreload */
17     TIM5->ARR = 0xFFFF; /* 16 bits */
18
19     /* 5) Cargamos el prescaler */
20     TIM5->PSC = 15;
21
22     /* 6) Habilita el modo de auto-reload */
23     TIM5->CR1 |= TIM_CR1_ARPE;
24
25     /* 7) (IMPORTANTE) Re-inicializa el contador y actualiza los registros.
           IMPORTANTE que este despues de cargar ARR y PSC */
26     TIM5->EGR |= TIM_EGR_UG; /* Genera evento de actualizacion */
27
28     /* Captura de entrada. CANAL 2 */
29     /* 8) Configurar el canal en modo captura de entrada */
30     /* 00 = output, 01 = input mapped on timer TI2 (input 2), 10 = input mapped on
           timer TI1 (input 1), 11 = input mapped on slave timer */
31     TIM5->CCMR1 &= ~TIM_CCMR1_CC2S; /* Limpiamos para asegurar que esta a 0 */
32     TIM5->CCMR1 |= (0x1 << TIM_CCMR1_CC2S_Pos); /* 01 = input mapped on TI2 */
33
34     /* 9) Deshabilita el filtro digital para queremos capturar todos los eventos */
35     TIM5->CCMR1 &= ~TIM_CCMR1_IC2F;
36
37     /* 10) Detectar flancos solo de una polaridad: flanco de subida, por ejemplo */
38     /* 00 = rising edge, 01 = falling edge, 10 = reserved, 11 = both edges */
39     TIM5->CCER |= (1 << TIM_CCER_CC2P_Pos | 0 << TIM_CCER_CC2NP_Pos); /* Rising
           edge */
```

```
40
41      /* 11) Prescaler de entrada. Para capturar cada transicion valida, poner a 0 */
42      TIM5->CCMR1 &= ~(TIM_CCMR1_IC2PSC);
43
44      /* 12) Habilitar captura de entrada */
45      TIM5->CCER |= TIM_CCER_CC2E;
46
47      /* 13) Habilitar interrupciones: captura de entrada y actualizacion */
48      TIM5->DIER |= TIM_DIER_CC2IE; /* Interrumpe al capturar */
49      TIM5->DIER |= TIM_DIER_UIE;   /* Interrumpe al actualizar */
50
51      /* 14) Elegir prioridad de interrupcion y habilitar globalmente en el NVIC */
52      NVIC_SetPriority(TIM5_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(), 1,
            0)); /* Priority 1, sub-priority 0 */
53      NVIC_EnableIRQ(TIM5_IRQn); /* Habilita interrupcion */
54
55      /* 15) Habilita timer para empezar a contar (podria hacerse en otro momento) */
56      TIM5->CR1 |= TIM_CR1_CEN;
57  }
```

Por último, nos faltaría la ISR del temporizador. En ella leeremos el valor del contador en el momento en que se produzca el evento de captura de entrada. En este caso, la ISR se activará cuando se produzca un flanco de subida en la señal de entrada —porque así lo hemos configurado—.

En la ISR, guardaremos el valor del contador en una variable, y limpiaremos el *flag* de interrupción del temporizador. Como en el ejemplo de interrupciones periódicas, la zona de memoria en la que se encuentra la ISR del `TIM5` está asociada a la interrupción `TIM5_IRQn` (ver fichero `startup_stm32f446xx.s`).

```
1   #include <stdio.h>          /* Para printf */
2   #define TIMER_PERIOD_US 10  /* 10 us */
3
4   /* La ISR debe tener este nombre obligatoriamente */
5   void TIM5_IRQHandler(void)
6   {
7       static uint32_t tick_ini = 0; /* static para que no se borre entre llamadas */
8       static uint32_t overflows = 0; /* static para que no se borre entre llamadas */
9
10      /* Comprobamos si la interrupcion ha sido por overflow del contador */
11      if (TIM5->SR & TIM_SR_UIF)
12      {
13          /* Aumentamos el contador de overflows */
14          overflows++;
15
16          /* Limpiamos el flag de interrupcion del timer */
17          TIM5->SR &= ~TIM_SR_UIF;
18      }
19
20      /* Comprobamos si la interrupcion ha sido por captura de entrada */
21      if (TIM5->SR & TIM_SR_CC2IF != 0)
22      {
23          /* Guardamos el valor del contador en una variable global */
24          /* Si es el primer flanco, guardamos el valor en tick_ini */
25          if (tick_ini == 0)
26          {
27              tick_ini = TIM5->CCR2;
```

```
28          }
29          /* Si es el segundo flanco, guardamos el valor en tick_end y calculamos el
                tiempo entre los dos flancos */
30          else
31          {
32              uint32_t tick_end = TIM5->CCR2;
33              uint32_t time = (tick_end + overflows * (TIM5->ARR + 1) - tick_ini) *
                    TIMER_PERIOD_US;
34              printf("Periodo de la senal de entrada: %d us\n", time);
35              tick_ini = 0; /* Reiniciamos los valores */
36              overflows = 0;
37          }
38
39          /* Limpiamos el flag de interrupcion del timer */
40          TIM5->SR &= ~TIM_SR_CC2IF;
41      }
42  }
```

Los cálculos del tiempo se podrían hacer en un `main()` si se desea, trabajando con variables globales.

Capítulo 6

Interrupciones externas

Josué Pagán

En este capítulo veremos la importancia de las interrupciones en los microcontroladores, y se presentan los detalles de los mecanismos de interrupción del **STM32F446RE**. Se recomienda repasar los conceptos de gestión de interrupciones que haya estudiado en teoría.

Conviene recordar el concepto de *"mecanismo de sincronización"*, que es la manera que tiene la CPU de saber si tiene que atender un periférico, o realizar una acción. Tenemos 2 formas: (i) el ***polling*** o espera activa, y (ii) la interrupción:

- El ***polling*** o espera activa: mediante SW lee continuamente el valor de un registro. **Es bloqueante**, *i.e.*, el programa no hace otra cosa más que comprobar dicho valor hasta que se cumpla una condición dada. **Es, por tanto, ineficiente**. Suele implementarse en bucles como `while`.

- **La interrupción** es una señal asíncrona que cambia el estado de un registro y cuando se produce avisa a la CPU. Mientras no se produce el cambio en el registro la CPU puede estar ejecutando el programa principal, otra interrupción previa o estar en algún modo de bajo consumo. **Es eficiente y no bloquea la ejecución principal** (si se gestiona bien, claro). Implementan las ISR.

Una interrupción es un evento (externo o interno) que detiene el contador de programa de la CPU, que deja de ejecutar lo que estaba haciendo y *cambia de contexto* para atender a la rutina de interrupción ISR. **Las ISR deben ser rápidas —para no detener mucho el proceso principal—, por lo que debemos escribir pocas líneas de código, y salir cuanto antes**. Una interrupción externa sucede, por ejemplo, cuando hay un flanco (cambio de tensión) de subida o de bajada, en un pin de una GPIO. Una interrupción interna sucede, por ejemplo, cuando un temporizador llega a la cuenta establecida para que nos interrumpa.

La CPU de un microcontrolador puede gestionar decenas de interrupciones de distinto origen. Las interrupciones están deshabilitadas en el arranque del sistema,

generalmente. **Antes de que una interrupción pueda ser aceptada por la CPU se debe cumplir** que:

1. **Estén habilitadas las interrupciones de la fuente** que queremos gestionar (*e.g.*, las de una GPIO, un *timer*, un ADC, ...). Además, la interrupción no debe estar enmascarada en el registro del periférico.

2. La fuente genere la interrupción.

3. **Estén habilitadas las interrupciones globales del procesador**.

Las interrupciones tienen **prioridades y subprioridades**. Si una interrupción de mayor prioridad interrumpe a otra que se estaba ejecutando, esta última se detiene, *se guarda su contexto en la pila (stack)*, y se ejecuta la ISR de mayor prioridad. Así, las interrupciones se van anidando. Cuando se termina la ejecución de la de mayor prioridad, se vuelve a la de menor prioridad para que termine lo que estaba haciendo. Si ambas interrupciones tienen igual prioridad, se atenderá la de mayor subprioridad. Es por todo esto que es importante que las ISR se ejecuten rápido.

Las interrupciones se identifican mediante números enteros que definen su prioridad por defecto asociándoles números negativos a las interrupciones definidas por ARM Cortex-M, y números positivos las definidas por el fabricante ST. Estos números podemos verlos definidos según el estándar CMSIS, en el fichero de cabecera de cada dispositivo `stm32f446xx.h` para el microcontrolador **STM32F446RE**.

6.1. El NVIC

El NVIC es el controlador que se encarga de gestionar las interrupciones que llegan a la CPU. Es común a todos los microcontroladores ARM Cortex-M.

El NVIC es capaz de gestionar 16 niveles de prioridad. Donde **el nivel 0 es el de máxima prioridad, y el nivel 15 el de mínima prioridad**. Controla 96 interrupciones *"enmascarables"*.

Algunas interrupciones han de atenderse sí o sí desalojando cualquier otra independientemente de su prioridad. Son interrupciones relacionadas con eventos excepcionales en el sistema, como un fallo del sistema, de alimentación, un bucle infinito, *etc.* A estas interrupciones —en contraposición a las otras— se le denominan *"no enmascarables"*.

Se deben cumplir las siguientes condiciones en el dispositivo y dentro de la CPU para que se acepte una interrupción [4]:

- Cada fuente de interrupción tiene un bit independiente para habilitarla (*enable*). **Este bit debe estar a '1' para que se pueda aceptar la interrupción.**

- Cada fuente de interrupción tiene un bit independiente de estado (*flag*). **Este *flag* se activa automáticamente por HW cuando se da la interrupción. Este *flag* lo ha de desactivar el usuario por SW.**
- El NVIC recibe la interrupción, y envía a la CPU la que tenga máxima prioridad.

Cada interrupción, habitualmente tiene un área de memoria dedicada donde se ubica el código de su ISR [4]. Para saber en qué zona de memoria se encuentra una ISR se utiliza el *vector de interrupción*. También **es posible encontrar que varias fuentes de interrupción compartan la misma zona de memoria para sus ISR, y es tarea del programador identificar cuál de las fuentes ha realizado la interrupción**. Estas zonas de memoria tienen una etiqueta en ensamblador con el nombre de la **ISR**. El diseñador del procesador y el fabricante del microcontrolador proporcionan estos **nombres preestablecidos** en la *tabla de vectores* que se encuentran, según el estándar CMSIS, definidos en el fichero de ensamblador de cada dispositivo `startup_stm32f446xx.s` para el microcontrolador **STM32F446RE**. La Figura 6.1 muestra alguno de los nombres de interrupciones habituales (ver la tabla *"Table 38. Vector table for STM32F446xx"* del *reference manual* [6] con todos los nombres disponibles).

Eventos

En un microcontrolador también podemos tener **eventos**. Los eventos, a diferencia de las interrupciones, no implican el salto a una ISR, *i.e.*, no ejecutan código. Sirven para despertar cuando ocurre una situación pero no tenemos que ejecutar un código específico. **Un evento siempre ocurre esté, o no, activada la interrupción. Si no asociamos un periférico a una interrupción, tendremos un evento que podríamos comprobar mediante *polling* o espera activa en un `while`.**

Un evento sí puede generar interrupción en otro módulo, por ejemplo, un *timer* periódico al que no se le ha asociado una interrupción en el `NVIC` nos despierta con un evento. Este se puede usar para lanzar del ADC y guardarlo en memoria mediante Direct Memory Access (Acceso Directo a Memoria) (DMA).

Existen más diferencias en la gestión de eventos e interrupciones, pero en general no usaremos eventos, sino interrupciones.

6.2. Interrupciones externas (EXTI)

El procesador del **STM32F446RE** tiene 23 líneas de interrupciones externas para las GPIO, el RTC, USB, *etc.* El External Interrupt/ Event Controller (Controlador de Interrupciones Externas) (EXTI) es un controlador que gestiona la detección de flanco

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	fixed	Reset	Reset	0x0000 0004
-	-2	fixed	NMI	Non maskable interrupt, Clock Security System	0x0000 0008
-	-1	fixed	HardFault	All class of fault	0x0000 000C
-	0	settable	MemManage	Memory management	0x0000 0010
-	4	settable	Debug Monitor	Debug Monitor	0x0000 0030
-	6	settable	Systick	System tick timer	0x0000 003C
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
23	30	settable	EXTI9_5	EXTI Line[9:5] interrupts	0x0000 009C
25	32	settable	TIM1_UP_TIM10	TIM1 Update interrupt and TIM10 global interrupt	0x0000 00A4
28	35	settable	TIM2	TIM2 global interrupt	0x0000 00B0
29	36	settable	TIM3	TIM3 global interrupt	0x0000 00B4
40	47	settable	EXTI15_10	EXTI Line[15:10] interrupts	0x0000 00E0

Figura 6.1: Extracto de la tabla de vectores de interrupción del *reference manual*.

para generar las interrupciones de las 23 líneas mencionadas. Para cada una de las 23 líneas de detección se puede configurar el disparador (*trigger*) que decide si se detecta el cambio de tensión en flanco de subida, de bajada, o ambos.

IMPORTANTE

Los pines que van a generar una interrupción externa deben ser configurados como entradas digitales, *i.e.*, debemos excribir `00` en los bits del registro `GPIO->MODER` correspondiente.

Nos vamos a centrar en la gestión de las interrupciones de las GPIO. El mapeo de las interrupciones externas de las GPIO se muestra en la Figura 6.2. 16 multiplexores mapean los pines/ líneas de las GPIO a los bits de interrupciones externas: de la interrupción `EXTI0` a la `EXTI15`.

Tenemos 4 registros llamados **External Interrupt Control Register SYSCFG_EXTICRx**, con $x = [1, .., 4]$. Estos registros tienen cada uno 16 bits disponibles, agrupados de 4 en 4 bits. Cada registro son 4 pines (*e.g.*, `SYSCFG->EXTICR[3]` para pines `12` a `15`) como muestra la Figura 6.3. Cada grupo de

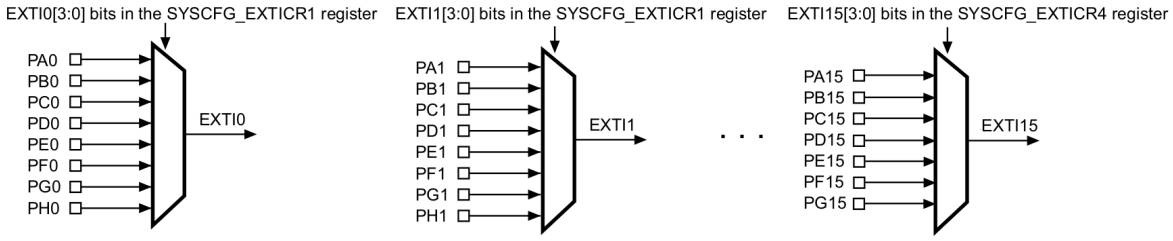


Figura 6.2: Mapeo de los pines/ líneas de las GPIO a las líneas de interrupción `EXTI0`-`EXTI15`.

4 bits indica el puerto GPIO asociado con la línea EXTI correspondiente. Cada grupo de 4 bits tienen en el nombre de la interrupción el número de una línea de la GPIO (de `EXTI0` a `EXTI15`) para las 16 líneas de GPIO que tenemos. Para asignar un pin de una GPIO a una interrupción debemos hacerlo por SW. Lo mejor es verlo con un ejemplo:

Imaginemos que queremos que el botón `B1` de la placa **Nucleo-STM32** nos interrumpa cada vez que se pulse, *i.e.* que haya un flanco de subida; un cambio de tensión de bajo a alto, para encender el LED `LD2`. Según el esquemático de la placa, el botón está conectado al puerto `PC13` (`GPIOC`, pin/ línea 13). Como el botón está conectado al `PC13`, se debe seleccionar obligatoriamente la línea de interrupción externa `EXTI13`. Para asociar la interrupción debemos escribir en los bits `EXTI13` del registro **SYSCFG_EXTICR4** el valor `0x0010`, como nos muestra la Figura 6.3 extraída del *reference manual* [6]. En la Sección 6.4 encontrará un **código de ejemplo**.

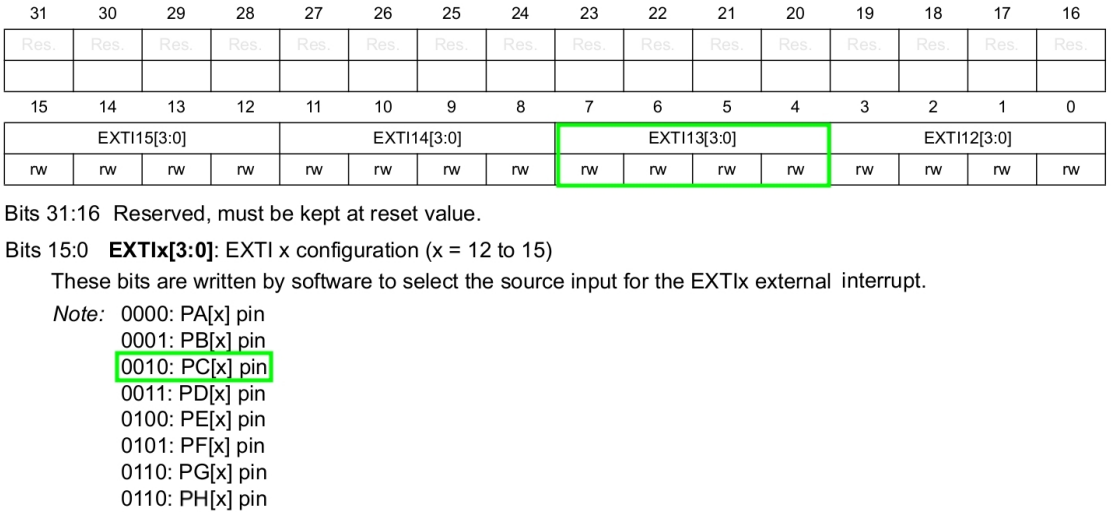


Figura 6.3: SYSCFG_EXTICR4 para asignar la interrupción `EXTI13` al pin `PC13` donde está conectado el botón `B1` de la placa **Nucleo-STM32**.

6.3. Registros de interrupciones externas

Existen varios registros de configuración y estado de las interrupciones externas que son de interés y que utilizaremos en el proyecto. Se muestran en la Figura 6.4 y la Figura 6.5 y son los que se muestran a continuación.

6.3.1. Interrupt Mask Register (EXTI_IMR)

Es el registro de máscaras de interrupciones externas. **Las interrupciones se pueden habilitar o enmascarar** de forma global, con el `NVIC`, o **de forma local con el registro EXTI_IMR**. Tiene 16 bits accesibles (uno por cada línea de GPIO). Escribir un `'1'` habilita la interrupción de esa línea de `EXTI` localmente; un `'0'` la deshabilita.

6.3.2. Rising Trigger Selection Register (EXTI_RTSR)

Es el registro usado para configurar una interrupción en flanco de subida (tensión de entrada en el pin pasa de bajo a alto). Un `'1'` habilita la interrupción en flanco de subida.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	MR22	MR21	MR20	MR19	MR18	MR17	MR16
									rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:23 Reserved, must be kept at reset value.
Bits 22:0 **MRx:** Interrupt mask on line x
0: Interrupt request from line x is masked
1: Interrupt request from line x is not masked

(a) EXTI_IMR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	TR20	Res.	TR18	TR17	TR16
									rw	rw	rw		rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:23 Reserved, must be kept at reset value.
Bits 22:20 **TRx:** Rising trigger event configuration bit of line x
Bits 18:0 0: Rising trigger disabled (for Event and Interrupt) for input line
1: Rising trigger enabled (for Event and Interrupt) for input line
Bit 19 Reserved, must be kept at reset value.

(b) EXTI_RTSR

Figura 6.4: Registros EXTI_IMR y EXTI_RTSR.

6.3.3. Falling Trigger Selection Register (EXTI_FTSR)

Es el registro usado para configurar una interrupción en flanco de bajada (tensión de entrada en el pin pasa de alto a bajo). Un `'1'` habilita la interrupción en flanco de bajada.

NOTA

Si queremos que se detecte la interrupción tanto en flanco de subida, como de bajada, debemos activar los bits en ambos registros EXTI_RTSR y EXTI_FTSR.

6.3.4. Pending Register (EXTI_PR)

Es el registro de estado de las interrupciones externas. Cuando ocurre un flanco en una de las líneas, se activa (un '1') el bit correspondiente, indicando que ha saltado dicho evento. **Una vez atendida la interrupción en la ISR, se debe escribir un `'1'` en el bit correspondiente para limpiar el estado *pendiente* de la interrupción** para cancelar la petición de interrupción y que no vuelva a saltar la ISR constantemente. **Si no se limpia el *flag* no se podrán detectar nuevos eventos**. Esto se suele hacer en la propia ISR.

6.4. Ejemplo de interrupción externa

Continuando con el ejemplo anterior, en el que queremos que el botón `B1` de la placa **Nucleo-STM32** nos interrumpa cada vez que se pulse para encender el LED `LD2` podemos hacer que la función en `C` —con el estándar CMSIS— tenga una pinta similar a esta:

```
1  #define BIT5_MASK 0x0020 /* Binario: 0b0010 0000 */
2  #define BIT13_MASK 0x2000 /* Binario: 0b0010 0000 0000 0000 */
3
4  #define MODER13_MASK 0x0C00 0000 /* Binario: 0b1100 0000 0000 0000 0000 0000*/
5  #define MODER5_AS_OUTPUT 0x0400 /* Binario: 0b0100 0000 0000*/
6
7  #define PUDR13_MASK 0x0C00 0000 /* Binario: 0b1100 0000 0000 0000 0000 0000*/
8  #define PUPD5_MASK  0x0C00 /* Binario: 0b1100 0000 0000*/
9
10 #define EXTI13_MASK 0x00F0 /* Binario: 0b1111 0000 */
11 #define PORTC_EXTI13_MASK 0x0020 /* Binario: 0b0010 0000 */
12
13 void gpio_config()
14 {
15     /* Primero, habilitamos siempre el reloj de los perifericos */
16     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; /* GPIOA clock enable */
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	TR20	Res.	TR18	TR17	TR16
									rw	rw	rw		rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:20 **TRx:** Falling trigger event configuration bit of line x
Bits 18:0 0: Falling trigger disabled (for Event and Interrupt) for input line
1: Falling trigger enabled (for Event and Interrupt) for input line.

Bit 19 Reserved, must be kept at reset value.

(a) EXTI_FTSR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PR22	PR21	PR20	PR19	PR18	PR17	PR16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Bits 31:23 Reserved, must be kept at reset value.
Bits 22:0 **PRx:** Pending bit on line x
0: No trigger request occurred
1: selected trigger request occurred
This bit is set when the selected edge event arrives on the external interrupt line.
This bit is cleared by programming it to '1'.

(b) EXTI_PR

Figura 6.5: Registros EXTI_FTSR y EXTI_PR.

```
17      RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN; /* GPIOC clock enable */
18
19      /* Boton B1 como entrada sin pull en PC13 */
20      GPIOC->MODER &= ~MODER13_MASK; /* Limpiamos y tambien queda definido como
            entrada */
21      GPIOC->PUPDR &= ~PUPD13_MASK; /* Limpiamos y tambien queda definido sin pull */
22
23      /* Habilitamos el reloj de control de configuracion del sistema */
24      RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN; /* SYSCFG clock enable */
25
26      /* Asigna interrupcion externa en PC13 a EXTI13 */
27      SYSCFG->EXTICR[3] &= ~EXTI13_MASK; /* Limpiamos. EXTICR[3] es EXTICR4 */
28      SYSCFG->EXTICR[3] |= PORTC_EXTI13_MASK; /* Asociamos EXTI13 al puerto C. */
29
30      /* Disparo de la interrupcion por flanco de subida */
31      EXTI->RTSR |= BIT13_MASK;
32
33      /* Habilitamos interrupcion local */
34      EXTI->IMR |= BIT13_MASK;
35
36      /* Damos prioridad y habilitamos interrupcion de la fuente de la GPIO del boton
            */
37      NVIC_SetPriority(EXTI15_10_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping()
            , 1, 0));
38      NVIC_EnableIRQ(EXTI15_10_IRQn);
39
40      /* LED LD2 como salida sin pull en PA5 */
41      GPIOA->MODER |= MODER5_AS_OUTPUT;
```

```
42      GPIOA->PUPDR &= ~PUPD5_MASK; /* Limpiamos y tambien queda definido sin pull */
43  }
```

La zona de memoria en la que se encuentra la ISR del `EXTI13` está compartida con otras. En particular, su ISR está asociada a la interrupción `EXTI15_10_IRQn` para las líneas de la 10 a la 15 de las GPIO.

```
1   /* La ISR debe tener este nombre obligatoriamente */
2   void EXTI15_10_IRQHandler(void)
3   {
4       GPIOA->ODR |= BIT5_MASK; /* Encender LED */
5       EXTI->PR = BIT13_MASK; /* Limpiar flag, escribiendo un 1 */
6   }
```

Capítulo 7

ADC: Analog-to-Digital Converter

Josué Pagán
Juan José Gómez

En este capítulo vamos a explicar los conceptos más generales de la conversión analógica-digital (ADC) y cómo se implementa en el microcontrolador **STM32F446RE**. La conversión analógica-digital es un proceso que se utiliza para convertir una señal analógica de una magnitud física en una señal digital. La señal digital es la que el microcontrolador puede procesar y almacenar en su memoria.

7.1. Introducción

En general, los módulos ADC de los microcontroladores son de los periféricos más complejos y con mayor número de configuraciones. Este capítulo no cubre en su totalidad todas las configuraciones posibles de un ADC, pero sí cubre las configuraciones más comunes y necesarias para la mayoría de las aplicaciones. En particular, se cubrirán las configuraciones del ADC del microcontrolador **STM32F446RE**.

Un conversor ADC genera un número digital de precisión finita, con o sin signo, que representa una aproximación de la magnitud de una señal de voltaje analógico en relación con un voltaje de referencia. Este voltaje de referencia es fijo y puede ser proporcionado por un circuito interno del microcontrolador o por una fuente externa conectada a un pin específico del controlador. El ADC no puede convertir valores de voltaje que excedan esta referencia.

Los parámetros fundamentales de un ADC incluyen:

- **Frecuencia de muestreo**. Indica la cantidad de conversiones que el ADC puede realizar por segundo. Dependiendo del tipo de ADC, esta cifra puede alcanzar millones o incluso miles de millones de muestras por segundo.

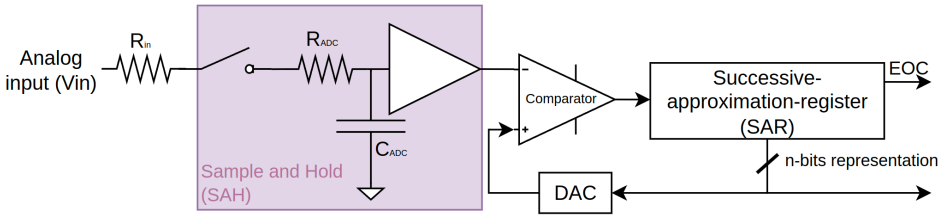


Figura 7.1: Diagrama de conexiones típico de un ADC con *Sample & Hold* y SAR.

- **Reolución**. Se refiere al número de bits del valor digital producido por el ADC. Las resoluciones más comunes varían entre 6 y 24 bits.
- **Potencia disipada**. Representa la eficiencia energética del ADC, es decir, el consumo de energía requerido para su funcionamiento.

Las arquitecturas de ADC más comunes son **Sigma-Delta** para aplicaciones de baja velocidad, **Aproximaciones Sucesivas** (Successive Approximation Register (Registro de Aproximación Sucesiva) (SAR)) para aplicaciones de bajo consumo y ***Pipelined ADC*** para aplicaciones de alta velocidad.

El ADC integrado en los microcontroladores STM32 se basa en la arquitectura SAR, la cual, como se ilustra en la Figura 7.1, está compuesta por dos elementos principales: el circuito de *Sample & Hold*, responsable de capturar y mantener la señal analógica, y el módulo de Aproximaciones Sucesivas, encargado de convertir la señal al valor digital final.

El *Sample & Hold* es un circuito muy sencillo que mantiene constante el valor de la señal analógica de entrada durante el tiempo que dura la conversión. La técnica de *Sample & Hold* (muestreo y retención) es de las más sencillas y efectivas, aunque existen otras técnicas más complejas como la de *Track & Hold* (seguimiento y retención) que permiten una mayor velocidad de conversión [10].

Con el *Sample & Hold* un conmutador controlado por el temporizador del ADC hace pasar la corriente por la resistencia R_{ADC}, y carga la tensión resultante en el condensador C_{ADC}. Cuando el conmutador se abre, la tensión en el condensador se mantiene constante, y es la que se compara con la tensión de referencia. Aquí empieza la fase de conversión.

El SAR es un circuito que digitaliza una señal analógica comparándola iterativamente con valores de referencia generados internamente. Su funcionamiento se basa en un **algoritmo de búsqueda binaria**, que identifica de manera progresiva el valor digital que mejor representa la señal de entrada.

El proceso de conversión se desarrolla en los siguientes pasos:

1. **Muestreo**: Se captura y mantiene el voltaje de entrada (V_{in}) utilizando un circuito (*Sample-and-Hold*).

2. **Inicialización**: El registro de aproximaciones sucesivas (SAR) se establece en un valor intermedio (típicamente el bit más alto activado, es decir, 1000...000 en binario).

3. **Comparación Iterativa**: En cada ciclo, el DAC interno genera un voltaje proporcional al valor almacenado en elSAR y lo compara con V_{in}.

   - Si el voltaje del DAC es menor que V_{in}, el bit de prueba actual se mantiene en 1.
   - Si el voltaje del DAC es mayor que V_{in}, el bit se pone en 0.

4. **Refinamiento del Valor**: Se repite el proceso desplazando la prueba al siguiente bit menos significativo hasta completar la resolución del ADC (por ejemplo, 12 iteraciones en un ADC de 12 bits).

5. **Salida del Resultado**: Una vez finalizado el proceso, el registro SAR contiene el código digital correspondiente a V_{in}, que se almacena en un registro de datos y puede ser leído por el sistema.

Este método permite una conversión rápida y eficiente, ya que en cada iteración se reduce a la mitad el rango de incertidumbre, logrando así una alta precisión en un tiempo mínimo.

7.2. Tiempo de Conversión

El tiempo total de conversión de un ADC SAR es la suma de dos componentes:

- **Tiempo de muestreo** (t_{samp}): Es el tiempo que el circuito de muestreo y retención (*Sample-and-Hold*) necesita para capturar y estabilizar la señal analógica de entrada.
- **Tiempo de conversión SAR** (t_{conv}): Es el tiempo que tarda el algoritmo de búsqueda binaria en realizar la conversión digital.

Por lo tanto, el tiempo total de conversión se expresa como:

$$t_{total} = t_{samp} + t_{conv} \tag{7.1}$$

Donde:

- t_{conv} depende del número de bits de resolución y del periodo del reloj del ADC, según la ecuación:

$$t_{conv} = (N + 1) \cdot t_{clk} \tag{7.2}$$

- t_{samp} es definido por el tiempo de muestreo programado en el ADC.

7.2.1. Ejemplo de cálculo del tiempo de conversión

Supongamos un ADC de 12 bits con una frecuencia de reloj de 16 MHz y un tiempo de muestreo de 4 ciclos de reloj. Los valores se calculan de la siguiente manera:

- Tiempo de muestreo:

$$t_{samp} = 4 \times \frac{1}{16 \times 10^6} = 4 \times 62{,}5 \text{ ns} = 250 \text{ ns} \tag{7.3}$$

- Tiempo de conversión SAR:

$$t_{conv} = 12 \times \frac{1}{16 \times 10^6} = 13 \times 62{,}5 \text{ ns} = 750 \text{ ns} \tag{7.4}$$

- Tiempo total de conversión:

$$t_{total} = 250 \text{ ns} + 750 \text{ ns} = 1000 \text{ ns} = 1\,\mu\text{s} \tag{7.5}$$

Esto significa que con esta configuración el ADC podría realizar 1 millón de conversiones por segundo (1 **MSPS!** (**MSPS!**)).

7.3. Cálculo de la Salida Digital

La salida digital (D) de un ADC depende del voltaje de entrada (V_{in}), los voltajes de referencia positivo (V_{ref+}) y negativo (V_{ref-}), y la resolución del ADC en bits (N). La relación se expresa mediante la siguiente fórmula:

$$D = \left\lfloor \frac{(V_{in} - V_{ref-})}{(V_{ref+} - V_{ref-})} \times (2^N - 1) + 0{,}5 \right\rfloor$$

donde:

- D: Valor digital resultante.
- V_{in}: Voltaje de entrada analógico.
- V_{ref+}: Voltaje de referencia positivo del ADC.
- V_{ref-}: Voltaje de referencia negativo del ADC.
- N: Número de bits de resolución del ADC.
- $\lfloor x \rfloor$: Función piso que redondea x al entero más cercano por debajo.

La inclusión de V_{ref-} permite al ADC manejar señales que no necesariamente están referenciadas a 0 V, ampliando su rango. Además, al sumar 0.5 antes de aplicar la función piso, se logra un redondeo al entero más cercano en lugar de un truncamiento.

7.4. Error de Conversión

Durante el proceso de conversión, se introduce un **error de cuantificación** debido a la naturaleza discreta de la salida digital. Este error representa la diferencia entre el valor analógico real y su representación digital más cercana. El **tamaño del paso de cuantificación** (Q) se calcula como:

$$Q = \frac{(V_{ref+} - V_{ref-})}{(2^N - 1)}$$

El **error de cuantificación** máximo es de $\pm\frac{Q}{2}$, ya que la señal analógica puede diferir del valor digitalizado hasta la mitad del paso de cuantificación.

7.4.1. Ejemplo práctico de cálculo del error cuantificación

Supongamos un ADC de 12 bits ($N = 12$) con un voltaje de referencia positivo de 3.3 V ($V_{ref+} = 3{,}3\,\text{V}$) y un voltaje de referencia negativo de 0 V ($V_{ref-} = 0\,\text{V}$). Si se aplica un voltaje de entrada de 1.65 V ($V_{in} = 1{,}65\,\text{V}$), los cálculos serían los siguientes:

1. **Cálculo del tamaño del paso de cuantificación**:

$$Q = \frac{(3{,}3\,\text{V} - 0\,\text{V})}{(2^{12} - 1)} = \frac{3{,}3\,\text{V}}{4095} \approx 0{,}00080586\,\text{V} \approx 0{,}80586\,\text{mV}$$

2. **Cálculo de la salida digital**:

$$\begin{aligned} D &= \left\lfloor \frac{(1{,}65\,\text{V} - 0\,\text{V})}{(3{,}3\,\text{V} - 0\,\text{V})} \times 4095 + 0{,}5 \right\rfloor \\ &= \left\lfloor \frac{1{,}65\,\text{V}}{3{,}3\,\text{V}} \times 4095 + 0{,}5 \right\rfloor \\ &= \lfloor 2047{,}5 + 0{,}5 \rfloor \\ &= \lfloor 2048 \rfloor \\ &= 2048 \end{aligned} \tag{7.6}$$

3. **Cálculo del voltaje digitalizado**:

$$V_{out} = D \times Q = 2048 \times 0{,}80586\,\text{mV} \approx 1{,}651\,\text{V}$$

4. **Cálculo del error de cuantificación**:

$$\text{Error} = V_{in} - V_{out} = 1{,}65\,\text{V} - 1{,}651\,\text{V} = -0{,}001\,\text{V} = -1\,\text{mV}$$

En este caso, el voltaje de entrada de 1,65 V se convierte en un valor digital de 2048, que corresponde a un voltaje digitalizado de aproximadamente 1,651 V. El error de cuantificación en este ejemplo es de -1 mV, lo que indica que el valor digitalizado es ligeramente superior al voltaje de entrada real.

7.5. Conversor Analógico a Digital (ADC) del Nucleo-STM32F446RE

El microcontrolador **Nucleo-STM32F446RE** incorpora tres ADC de 12 bits. Estos ADC permiten la adquisición precisa de señales analógicas, siendo adecuados para una amplia gama de aplicaciones para sistemas empotrados, como la interfaz con sensores, el procesamiento de señales y sistemas de control.

7.5.1. Características Principales

- **Resolución y Configuración**: El ADC admite múltiples resoluciones de 12 bits, 10 bits, 8 bits y 6 bits.
- **Múltiples Canales**: Hasta 19 canales de entrada multiplexados, incluyendo 16 fuentes externas, 2 fuentes internas y 1 canal VBAT para la monitorización del voltaje de la batería.
- **Modos de Conversión**: El ADC puede operar en modos de conversión única, continua, en escaneo y en modo discontinuo.
- **Manejo de Interrupciones y Eventos**: Soporta la generación de interrupciones al final de una conversión, al finalizar conversiones inyectadas y en casos de detección por el *watchdog* analógico o eventos de sobrecarga.
- **Disparo Externo**: Fuentes de disparo externo configurables con selección de polaridad para conversiones tanto regulares como inyectadas.
- **Watchdog Analógico**: Permite la monitorización de señales de entrada y genera una interrupción si el voltaje medido supera los umbrales predefinidos.
- **Operación en Modo Multi-ADC**: Modos de doble y triple ADC que permiten muestreo simultáneo, conversiones intercaladas y procesamiento en paralelo para aplicaciones de alta velocidad.
- **Integración con DMA**: Los ADC pueden conectarse con el controlador de Acceso Directo a Memoria (DMA) para un manejo eficiente de datos, reduciendo la carga de la CPU.
- **Requisitos de Alimentación**: Opera con un voltaje de alimentación entre 2.4V y 3.6V para funcionamiento a máxima velocidad, con soporte hasta 1.8V a velocidades reducidas.
- **Control de Tiempo de Muestreo**: Tiempo de muestreo programable por canal para optimizar la relación entre precisión y velocidad.

7.6. Registros del Convertidor Analógico a Digital (ADC)

El ADC del **Nucleo-STM32F446RE** cuenta con múltiples registros de control, estado y configuración que permiten gestionar su funcionamiento. A continuación, se describen los principales registros utilizados en la operación del ADC.

7.6.1. Registros del ADC

A continuación, se describen los registros más importantes del ADC **Nucleo-STM32F446RE** y sus principales bits de configuración:

- **ADC_SR (Registro de estado)**. Destacamos los bits:
  - **EOC** (Bit 1): Indica el fin de una conversión.
  - **OVR** (Bit 5): Señala un desbordamiento en la conversión.
  - **AWD** (Bit 0): Indica si se activó el watchdog analógico. El watchdog analógico es un mecanismo de supervisión que genera una interrupción cuando la señal de entrada del ADC excede los umbrales de voltaje predefinidos, permitiendo detectar valores fuera de rango.
- **ADC_CR1 (Registro de control 1)**. Destacamos los campos:
  - **RES[1:0]** (Bits 24-25): Configura la resolución del ADC (12, 10, 8 o 6 bits).
  - **SCAN** (Bit 8): Habilita el modo de escaneo.
  - **AWDEN** (Bit 23): Activa el watchdog analógico en canales regulares.
- **ADC_CR2 (Registro de control 2)**. Contiene entro otros,los campos:
  - **ADON** (Bit 0): Habilita o deshabilita el ADC.
  - **CONT** (Bit 1): Activa el modo de conversión continua.
  - **SWSTART** (Bit 30): Inicia manualmente una conversión.
- **ADC_SMPR1 y ADC_SMPR2 (Registros de tiempo de muestreo)**. Configuran el tiempo de muestreo para cada canal del ADC. Los valores de tiempo de muestreo varían entre 3 y 480 ciclos de reloj.
- **ADC_SQR1, ADC_SQR2 y ADC_SQR3 (Registros de secuencia de conversión)**. Definen el orden en el que se realizarán las conversiones en modo escaneo. El campo **L[3:0]** (Bits 20-23 en ADC_SQR1) especifica la cantidad de conversiones en la secuencia.

- **ADC_JSQR (Registro de secuencia de conversión inyectada)**. Define la secuencia de conversiones en modo inyectado. El campo **JL[1:0]** (Bits 20-21) indica el número de conversiones en el grupo inyectado. El modo inyectado del ADC permite realizar conversiones prioritarias y programadas de ciertos canales sin interrumpir una secuencia de conversiones regulares, ideal para lecturas críticas en momentos específicos.
- **ADC_DR (Registro de datos de conversión)**. Almacena el resultado de la conversión en 12 bits.
- **ADC_CCR (Registro de control común)**
  - **VBATE** (Bit 22): Habilita la medición del voltaje de batería.
  - **TSVREFE** (Bit 23): Habilita el sensor de temperatura interno y la referencia de voltaje.
- **ADC_CDR (Registro de datos en modo multi-ADC)**. Contiene los resultados de conversión cuando se usa el modo multi-ADC.

7.7. Ejemplo de uso del ADC1 con un potenciómetro

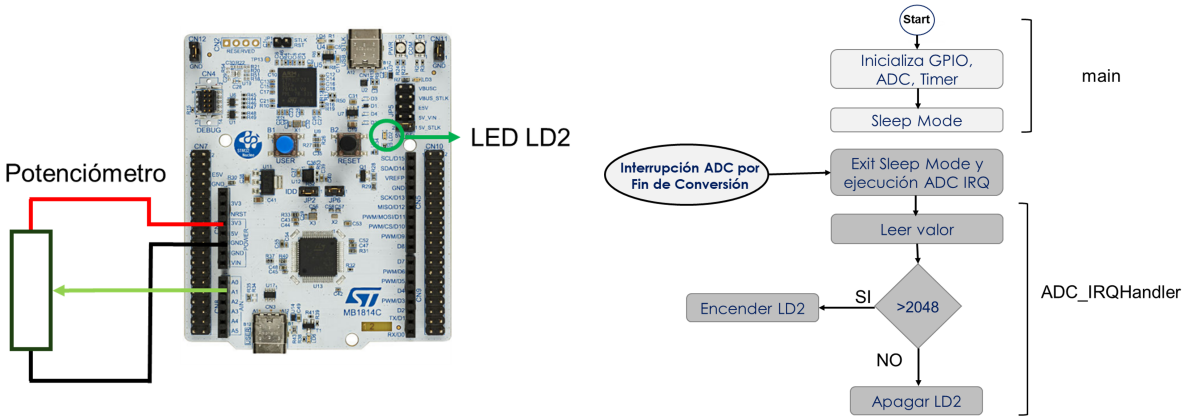
A continuación, veremos un ejemplo de uso del ADC1 del microcontrolador **Nucleo-STM32F446RE** utilizando el *GPIO Port A Pin 1*, es decir, PA1, para conectar un potenciómetro.

Un potenciómetro es una resistencia variable con tres terminales que actúa como un divisor de tensión ajustable. Su funcionamiento se basa en un contacto móvil que desliza sobre una resistencia fija. Cuando los terminales externos se conectan a Vcc y masa, el terminal central genera un voltaje que varía entre 0 y Vcc, dependiendo de la posición del contacto móvil.

En este ejemplo, mediremos el voltaje de entrada ajustado por el potenciómetro (Figura 7.2a). Si el voltaje de entrada (V_{input}) supera la mitad de V_{cc}, encenderemos el LED LD2 (Figura 7.2b).

7.7.1. Código completo del ejemplo

```
1 /* Standard C includes */
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <stdio.h>
5
6 /* HW dependent includes */
7 #include "port_system.h"
8
9 #include "stm32f4xx.h"
```



(a) Esquema de conexiones. *(b)* Flujo del programa.

Figura 7.2: Esquema de conexiones y flujo del programa para el ejemplo del ADC1 con un potenciómetro.

```
10
11 // Mask definition LED LD2
12 #define LD2_PIN 5           /*!< GPIO pin of the LED2 in the Nucleo board */
13 #define GPIO_MODE_OUT 0x01  /*!< GPIO as output */
14 #define GPIO_MODE_CLEAN 0x03 /*!< GPIO clear*/
15 #define GPIO_ODR 0x01       /*!< GPIO ODR  */
16 #define MODER5_MASK_CLEAR (GPIO_MODE_CLEAN << LD2_PIN * 2)  /*!< Mask to Clean
       MODE5 Register */
17 #define MODER5_MASK_OUT (GPIO_MODE_OUT << LD2_PIN * 2)  /*!< Output mode for the
       LED2 in the MODE Register */
18 #define BSRR5_SET_MASK  (1U << LD2_PIN)        /*!< Mask to set LED2 (PA5) in BSRR
       */
19 #define BSRR5_RESET_MASK (1U << (LD2_PIN + 16)) /*!< Mask to reset LED2 (PA5) in
       BSRR */
20
21
22 // Mask definition for potenciometer pin
23 #define POT_PIN 1            /*!< GPIO pin connected to the potentiometer (PA1) */
24 #define GPIO_MODE_ANALOG 0x03 /*!< GPIO mode as analog */
25 #define GPIO_MODE_CLEAN 0x03  /*!< GPIO mode clear mask */
26 #define MODER1_MASK_CLEAR (GPIO_MODE_CLEAN << (POT_PIN * 2)) /*!< Mask to clear
       MODE1 bits */
27 #define MODER1_MASK_ANALOG (GPIO_MODE_ANALOG << (POT_PIN * 2)) /*!< Analog mode for
        PA1 in MODER register */
28
29 volatile uint16_t adc_value = 0;
30
31 void ADC1_Init(void);
32 void GPIO_Init(void);
33 void TIM2_Init(void);
34
35 int main(void) {
36     GPIO_Init();     // Initialize GPIO for LED and potentiometer
37     ADC1_Init();     // Initialize ADC1
38     TIM2_Init();     // Initialize TIM2
39
40     while (1) {
41         // Main loop is now free for other tasks or low-power sleep mode
```

```
42          __WFI();  // Enter low-power mode (Wait For Interrupt)
43      }
44  }
45
46  void GPIO_Init(void) {
47      // Enable GPIOA clock
48      RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
49
50      // Configure PA5 as output
51      GPIOA->MODER &= ~MODER5_MASK_CLEAR; // Clear mode bits for PA5
52      GPIOA->MODER |= MODER5_MASK_OUT;    // Set PA5 to output mode
53
54      // Configure PA1 as analog input
55      GPIOA->MODER &= ~MODER1_MASK_CLEAR; // Clear mode bits for PA1
56      GPIOA->MODER |= MODER1_MASK_ANALOG; // Set PA1 to analog mode
57  }
58
59  void ADC1_Init(void) {
60      // Enable ADC1 clock
61      RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
62
63      // ADC1 configuration
64      ADC1->CR2 = 0; // RES is 0, 12 bits of resolution
65      ADC1->CR2 = 0; // Reset CR2 to ensure ADON is 0
66      ADC1->SQR3 = 1; // Set channel 1 (PA1) as the first conversion in regular
            sequence
67      ADC1->SMPR2 |= (0x7UL << ADC_SMPR2_SMP1_Pos); // Set sample time for channel 1
68      ADC1->CR2 |= ADC_CR2_CONT; // Enable continuous conversion mode
69      ADC1->CR2 |= ADC_CR2_ADON; // Enable ADC1
70
71      // Configure ADC to trigger on TIM2 update event
72      ADC1->CR2 |= (0x6UL << ADC_CR2_EXTSEL_Pos); // Select TIM2_TRGO as trigger
73      ADC1->CR2 |= ADC_CR2_EXTEN_0;  // Enable trigger on rising edge
74
75      // Enable End of Conversion (EOC) Interrupt
76      ADC1->CR1 |= ADC_CR1_EOCIE;
77
78      // Enable ADC interrupt in NVIC
79      NVIC_EnableIRQ(ADC_IRQn);
80  }
81
82  void TIM2_Init(void) {
83      RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;  // Enable TIM2 clock
84      TIM2->PSC = 25 - 1;  // Min Prescaler for period of timer 100 ms (assuming 16
            MHz clock)
85      TIM2->ARR = 64000 - 1;   // Auto-reload value (sets trigger rate)
86      TIM2->CR2 |= TIM_CR2_MMS_1; // Update event as TRGO (trigger output)
87      TIM2->CR1 |= TIM_CR1_CEN;  // Enable timer
88  }
89
90
91  // ADC Interrupt Handler
92  void ADC_IRQHandler(void) {
93      if (ADC1->SR & ADC_SR_EOC) {
94          uint16_t adc_value = ADC1->DR; // Read ADC value
95          ADC1->SR &= ~ADC_SR_EOC; // Clear EOC flag
96
97          // Update LED based on ADC value
98          if (adc_value > 2048) {
```

```
99              GPIOA->BSRR = BSRR5_SET_MASK;
100         } else {
101             GPIOA->BSRR = BSRR5_RESET_MASK;
102         }
103     }
104 }
105
106 void Enter_StopMode(void) {
107     RCC->APB1ENR |= RCC_APB1ENR_PWREN; // Enable Power Interface Clock
108     PWR->CR |= PWR_CR_LPDS; // Enable low-power mode in Stop Mode
109     SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; // Configure to enter deep sleep mode
110     __WFI(); // Enter Stop Mode (wait for interrupt)
111 }
```

7.7.1.1. Inicialización de GPIO

La función `GPIO_Init()` configura los pines GPIOA para salida y entrada analógica. El pin PA5 (LED LD2) se configura como salida modificando el registro `MODER`, mientras que el pin PA1 (Potenciómetro) se configura como entrada analógica habilitando la funcionalidad del ADC en el registro `MODER`.

7.7.1.2. Inicialización del ADC1

La función `ADC1_Init()` configura el ADC1 para leer el valor analógico desde PA1. Se habilita el reloj del ADC1 a través de `RCC->APB2ENR`, se configura el ADC para la conversión de un solo canal (PA1) y se establece el tiempo de muestreo. Además, se habilita el modo de conversión continua para lecturas ininterrumpidas y se configura el ADC1 para activarse con eventos de actualización de TIM2.

7.7.1.3. Configuración de TIM2

La función `TIM2_Init()` configura TIM2 como un disparador periódico para las conversiones ADC. Se habilita el reloj de TIM2 en `RCC->APB1ENR`, se establece el prescaler y el valor de recarga automática para generar una señal de activación cada 100 ms y se configura el modo maestro de TIM2 (TRGO) para generar eventos de actualización para el ADC.

7.7.1.4. Función de atención de la Interrupción del ADC

La función `ADC_IRQHandler()` procesa las conversiones ADC. Se lee el valor ADC desde el registro `ADC1->DR` y se borra el *flag* EOC. Si el valor ADC es mayor a 2048, se enciende el LED LD2 (PA5 se activa en `BSRR`). En caso contrario, se apaga el LED LD2 (PA5 se reinicia en `BSRR`).

7.7.1.5. Modo de Bajo Consumo (Stop Mode)

La función `Enter_StopMode()` configura el microcontrolador para entrar en modo *Stop* para ahorrar energía. Se habilita el reloj de gestión de consumo de energía, se configura el bit de bajo consumo en modo *Deep Sleep* y se llama a `__WFI()` para esperar una interrupción, en este caso de fin de conversión por parte del ADC.

Capítulo 8

USART: Universal Synchronous and Asynchronous Receiver-Transmitter

Josué Pagán

En este capítulo vamos a explicar los conceptos más generales de comunicación USART y cómo configurar el periférico USART del microcontrolador **STM32F446RE** para comunicarse con un ordenador u otro dispositivo mediante esta interfaz.

8.1. Introducción a la USART

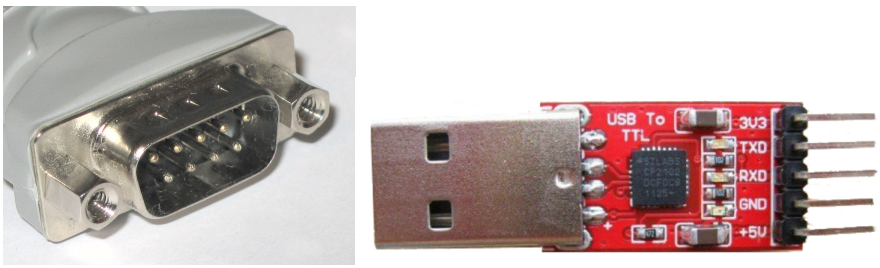
USART es el acrónimo de *"Universal Synchronous/Asynchronous Receiver/Transmitter"*, y es un protocolo de comunicación serie que se encuentra implementa como periférico en muchos microcontroladores utilizado tanto el modo síncrono (USART), como asíncrono (USART).

La ventaja de una comunicación serie es que solo se necesita un par de cables para transmitir y recibir datos, y no tiene los problemas de sincronización que tiene la comunicación paralela. Este protocolo no alcanza velocidades de transmisión tan altas como otros protocolos serie como SPI o I2C, pero es muy sencillo de implementar y es muy común en dispositivos embebidos. Lo podemos encontrar en dispositivos como módems, impresoras, ratones, teclados, pantallas o letreros luminosos, comunicación con módulos externos de *GPS*, *Bluetooth*, *GSM*, *SmartCards*, ...

Recuerde que en un sistema de comunicación en modo asíncrono, la transmisión y recepción de datos no están sincronizadas por una marca de tiempos común. En el modo síncrono, sí, y están sincronizadas por un reloj compartido entre los dispositivos conectados. En este capítulo nos centraremos en el modo asíncrono, que es el más común.

USART ha estado íntimamente relacionado con el estándar RS232. Este estándar define las características eléctricas y mecánicas de los conectores usados en

comunicaciones serie como los conectores DB9 de la Figura 8.1a[1]. Como los ordenadores modernos no tienen puertos serie, se utilizan conversores USB a serie, como el *FT232* de FTDI, o el *CP2102* de Silicon Labs de la Figura 8.1b[2] que usaremos[3].



(a) Conector DB9 macho. *(b)* Conversor USB-serie con chip CP2102.

Figura 8.1: Conector DB9 y conversor USB-serie.

Nuestro microcontrolador **STM32F446RE** tiene varios periféricos USART. Entre las principales características de USART podemos destacar las siguientes:

- Tiene dos líneas de comunicación: **Tx** y **Rx** para transmitir y recibir datos, respectivamente.
- Se puede configurar para transmitir y recibir datos en modo asíncrono o síncrono (alguno de los periféricos solo síncrono).
- Se puede configurar para transmitir y recibir datos en modo *full-duplex* o *half-duplex*. Típicamente se usa en modo *full-duplex*, que permite transmitir y recibir datos al mismo tiempo.
- Se puede configurar su velocidad mediante el *baud rate*, que es el número de símbolos que se transmiten por segundo, llegando a velocidades de hasta de 11.25 Mbps en el microcontrolador **STM32F446RE**.
- Se puede configurar la longitud (número de bits) de los datos transmitidos. 7 o 8 bits en el caso de nuestro microcontrolador.
- Incluye bits de inicio (*start*) y de parada (*stop*) para indicar el inicio y el final de la transmisión de un mensaje. Cuando el receptor detecta un bit de inicio, sabe que va a recibir un mensaje y lo lee los bits equivalentes a la tasa de baudios configurada. Cuando detecta un bit de parada, sabe que el mensaje ha terminado.

[1]Autor: Mike1024. Public domain. `https://upload.wikimedia.org/wikipedia/commons/7/78/9_pin_d-sub_connector_male_closeup.jpg?20060727150904`

[2]Autor: `https://stak.com/`. Accedido: 12/11/2023.

[3]Si su sistema operativo no reconoce cualquier conversor que conecte, es porque tiene los *drivers* instalados. Búsquelos, suelen estar en la web del fabricante. Aquí puede encontrar los de Silicon Labs para la familia *CP210x* `https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers`.

- Puede implementar un mecanismo de detección de errores mediante bits de paridad.
- Puede conectar solo dos dispositivos a la vez.
- Permite generar interrupciones cuando se recibe un dato.

En este capítulo no nos vamos a centrar en todo los detalles de este protocolo, sino en cómo configurar el periférico USART del microcontrolador **STM32F446RE** para comunicarse con un ordenador u otro dispositivo mediante esta interfaz. Para más información sobre el protocolo USART puede consultar el *reference manual* [6]. A continuación se describe el comportamiento básico en transmisión y recepción de datos.

8.1.1. Comunicación USART

Como se ha mencionado, la comunicación mediante USART nos va a permitir comunicar nuestra placa con un ordenador u otro dispositivo. En la Figura 8.2 se muestra un ejemplo de típico de conexión con un ordenador mediante un conversor USB-serie con el microcontrolador **STM32F446RE**. El conversor USB-serie se conecta al ordenador mediante un cable USB. Esta configuración nos permite enviar y recibir datos desde el ordenador. El conversor USB-serie se conecta a la USART del microcontrolador mediante dos cables: **Tx** y **Rx** para transmitir y recibir datos, respectivamente.

ATENCIÓN

Preste atención a que el pin de transmisión del microcontrolador está conectado al de recepción del conversor y viceversa.
Si la placa **Nucleo-STM32** se conectase a cualquier otro dispositivo, habría que conectar los pines de transmisión de la placa con los de recepción del otro dispositivo (o placa) y viceversa de igual modo. En esta situación, no es necesario el conversor USB-serie.

Entre el transmisor y el receptor hay que establecer puntos de sincronismo para que el receptor sepa cuándo empieza y termina una palabra. En la comunicación síncrona, esta tarea se apoya en la señal de reloj, pero en la comunicación asíncrona el transmisor y el receptor han de apoyarse en otros mecanismos como los bits de inicio (*start*) y de parada (*stop*). En la Figura 8.3 se muestra un ejemplo de transmisión de una palabra de 8 bits con 1 bit de *stop*. **La secuencia de transmisión sería la siguiente:**

- En la comunicación asíncrona en estado de reposo, **si el transmisor está habilitado**, la línea de comunicación está a nivel alto.

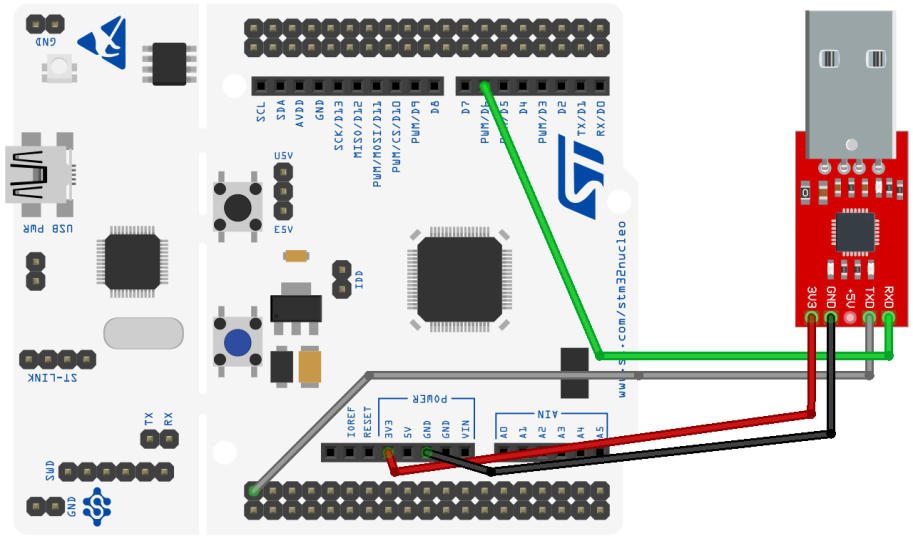


Figura 8.2: Conexión *full-duplex* de la USART-3 de la **Nucleo-STM32** con un ordenador usando un conversor USB-serie.

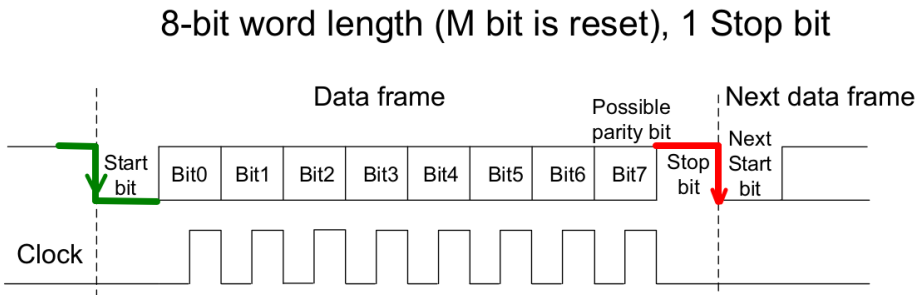


Figura 8.3: Ejemplo de transmisión de una palabra de 8 bits con 1 bit de *stop*.

- El bit de inicio es un bit de **valor** `'0'` que indica al receptor que va a recibir una palabra. Siempre lo veremos como una caída de tensión alto a bajo.
- A continuación se transmite la información. En una comunicación serie, la información se transmite en *palabras* (*words*), y estas se envían bit a bit. **Se transmite el bit menos significativo primero.**
- Después de la transmisión de la información, se transmite el bit de paridad (si existe).
- Después de la transmisión de la información, se transmite el bit de parada. El bit de parada es un bit de **valor** `'1'` que indica al receptor que la palabra ha terminado.

La secuencia de recepción sería la siguiente:

- El receptor detecta un bit de inicio, y sabe que va a recibir una palabra.

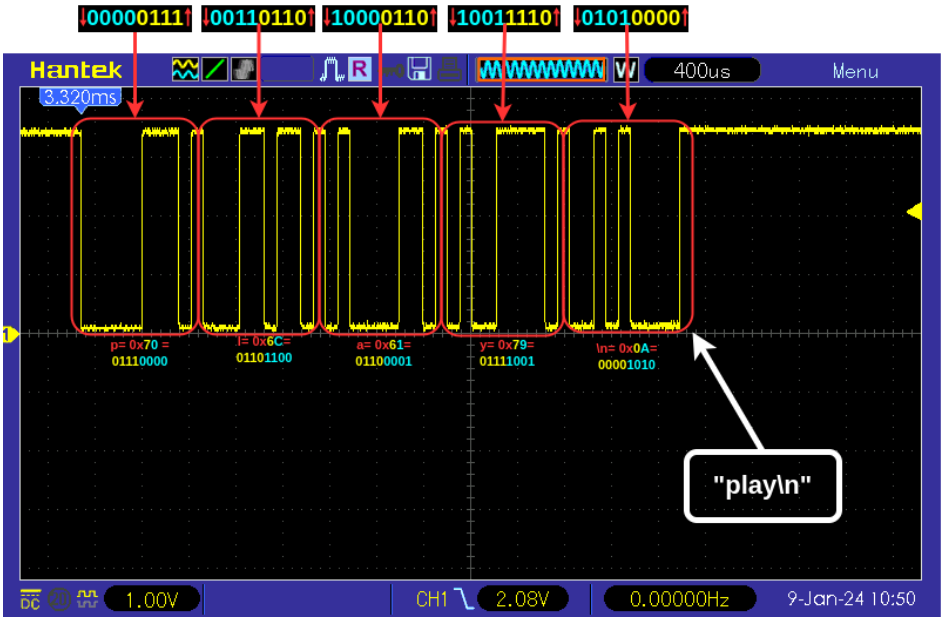


Figura 8.4: Ejemplo de transmisión de 8 bits, 1 bit de *stop* a 9600 baudios. Ejemplo de transmisión de "play\n" con 8 bits con 1 bit de *stop* a 9600 baudios.

- El receptor lee la información. En modo asíncrono, se utilizan técnicas de sobremuestreo (*oversampling*) para detectar el valor de cada bit y discernir entre un bit válido o ruido.

- El receptor detecta el bit de paridad (si existe) y comprueba si es correcto.

- El receptor detecta el bit de parada, y sabe que la palabra ha terminado.

En la Figura 8.4 puede ver una captura de osciloscopio ejemplo de transmisión de la palabra `play\n` con 8 bits con 1 bit de `stop` a 9600 baudios. Como son 5 caracteres y cada uno ocupa 8 bits más el bit de `start` y el bit de `stop`, se transmiten $5*10 = 50 bits$. A una velocidad de 9600 bits por segundo, podemos comprobar que la transmisión de la palabra `play\n` tarda $\frac{50}{9600} = 5{,}2\ ms$. En la Figura 8.4 se puede ver que el ancho de 1 cuadro es de 400 μs y que, efectivamente ocupa unos 13 cuadros, que son $13 * 400\ \mu s = 5{,}2\ ms$.

8.2. Configuración de la USART

La comunicación se basa en una configuración previa de parámetros entre el transmisor y el receptor. Un ejemplo de configuración de parámetros es el siguiente: *velocidad de transmisión de 9600 baudios, 8 bits de datos, 1 bit de parada y sin bit de paridad, y se representa como* `9600-8-N-1`. Esto es:

- **Velocidad de transmisión**: **9600 baudios**, es decir, el número de bits que se transmiten por segundo. En este caso, se transmiten 9600 bits por segundo. Esto determina el tiempo de 1 bit, que es el tiempo que se mantiene cada bit durante la transmisión. En este caso, el tiempo de 1 bit sería de $1/9600\ s = 104{,}17\ \mu s$. Para que la comunicación sea correcta, el transmisor y el receptor han de tener la misma tasa de baudios (*baud rate*). En nuestro **STM32F446RE** se configura mediante el registro `USART_BRR` (ver Sección 8.3.5).

- **Longitud de la palabra**: **8 bits**. Típicamente se transmiten palabras de 8 bits, pero se puede configurar para transmitir palabras de 7 bits. La longitud de *palabra* se configura mediante el bit `M` del registro `USART_CR1` (ver Sección 8.3.3). Cada bit se mantiene durante 1 tiempo de bit.

- **Bit de paridad**: **N** (*sin bit de paridad*). Típicamente no se usa bit de paridad, pero se puede configurar para transmitir un bit de paridad par o impar. Se añade al final de la palabra para detectar errores de transmisión, e indica si el número de bits de valor `1` es par o impar (según el tipo de paridad configurado). Podemos encontrar **N** (*sin bit de paridad*), **E** (*paridad par*), u **O** (*paridad impar*). La paridad se configura mediante los bits `PCE` y `PS` del registro `USART_CR1` (ver Sección 8.3.3).

- **Bit de parada**: **1 bit**. Típicamente 1. Se puede configurar para transmitir 2 bits de parada. El criterio de parada se configura mediante el bit `STOP` del registro `USART_CR2` (ver Sección 8.3.4)

Tanto para poder transmitir como para poder recibir, el periférico USART debe estar habilitado. Para ello, se activa el bit `UE` del registro `USART_CR1` (ver Sección 8.3.3).

8.2.1. Comunicación USART mediante interrupciones

Cuando esperamos recibir un mensaje, podríamos hacerlo mediante espera activa (*polling*) o mediante interrupciones. En el caso de espera activa, el microcontrolador comprueba si se ha recibido un dato en el registro `USART_DR` (ver Sección 8.3.2). En el caso de interrupciones, el microcontrolador comprueba si se ha recibido un dato en el registro `USART_SR` y genera una interrupción si es así. En ambos casos, el microcontrolador lee el dato recibido en el registro `USART_DR`. Como ya puede intuir, nos centraremos en el uso de interrupciones para recibir y transmitir datos (ver ejemplo en Sección 8.4).

En esta sección vamos a explicar las interrupciones en la USART. Los registros y bits a los que se va a hacer referencia se pueden ver en la Sección 8.3.

Recepción de datos por interrupción En recepción solo manejaremos un bit de interrupción, el bit `RXNE`. Este bit nos permite saber cuándo se ha recibido un carácter y está listo para ser leído.

La secuencia de interrupción en recepción es como sigue:

1. Cuando se recibe un carácter, se almacena en el registro de datos (`USART_DR`) y el bit `RXNE` (*Read Data Register Not Empty*) del registro de estado de la USART (`USART_SR`) se pone a `1` si la interrupción `RXNEIE` está habilitada.

2. En la ISR de la USART se lee el carácter recibido. Cuando se lee dato, el bit `RXNE` se pone a `0` automáticamente, aunque también puedes ponerlo a `0` escribiendo un `0` en él. Esto se recomienda solo para multibuffer.

Transmisión de datos por interrupción Vamos a centrarnos en una transmisión **de varios caracteres uno a uno**, no multibuffer.

Cuando se transmite un mensaje carácter a carácter usando la USART tenemos dos posibles interrupciones: `TXE` y `TC`. La interrupción `TXE` se utiliza para escribir datos en el registro de datos. Se activa cuando el registro de datos (`USART_DR`) está vacío y listo para ser cargado con el siguiente carácter a transmitir. La interrupción `TC` se utiliza para realizar alguna acción cuando la transmisión se completa. Se activa cuando el último bit ha sido desplazado fuera del registro de desplazamiento de transmisión a la línea de transmisión. Lo puede ver mejor en la Figura 8.5.

Si solo estamos transmitiendo un carácter a la vez y no se necesita realizar ninguna acción cuando la transmisión se completa, se puede ignorar la interrupción `TC` y solo usar la interrupción `TXE`. Sin embargo, si se estuviesen transmitiendo múltiples caracteres en una secuencia y se necesita saber cuándo se ha completado la transmisión de toda la secuencia, entonces se necesitará la interrupción `TC`.

La secuencia de interrupción en transmisión cuando se van a transmitir varios caracteres (uno a uno) es como sigue:

1. Primero, el usuario pone el primer carácter a transmitir en el registro de datos `USART_DR`. El primer carácter se envía normalmente en el código principal, no en una rutina de interrupción. Sea como fuere es el paso `1` de la Figura 8.5.

2. El bit de interrupción `TXE` del registro de estado `USART_SR` se pone automáticamente a `0` y se pasa a transmitir el carácter actual al registro de desplazamiento de transmisión (*Transmit Shift Register*). Paso `2` de la Figura 8.5.

    - En ese momento se ha vaciado el registro de datos `USART_DR` y, si el bit `TXEIE` del registro de control `USART_CR1` está habilitado, se genera una interrupción

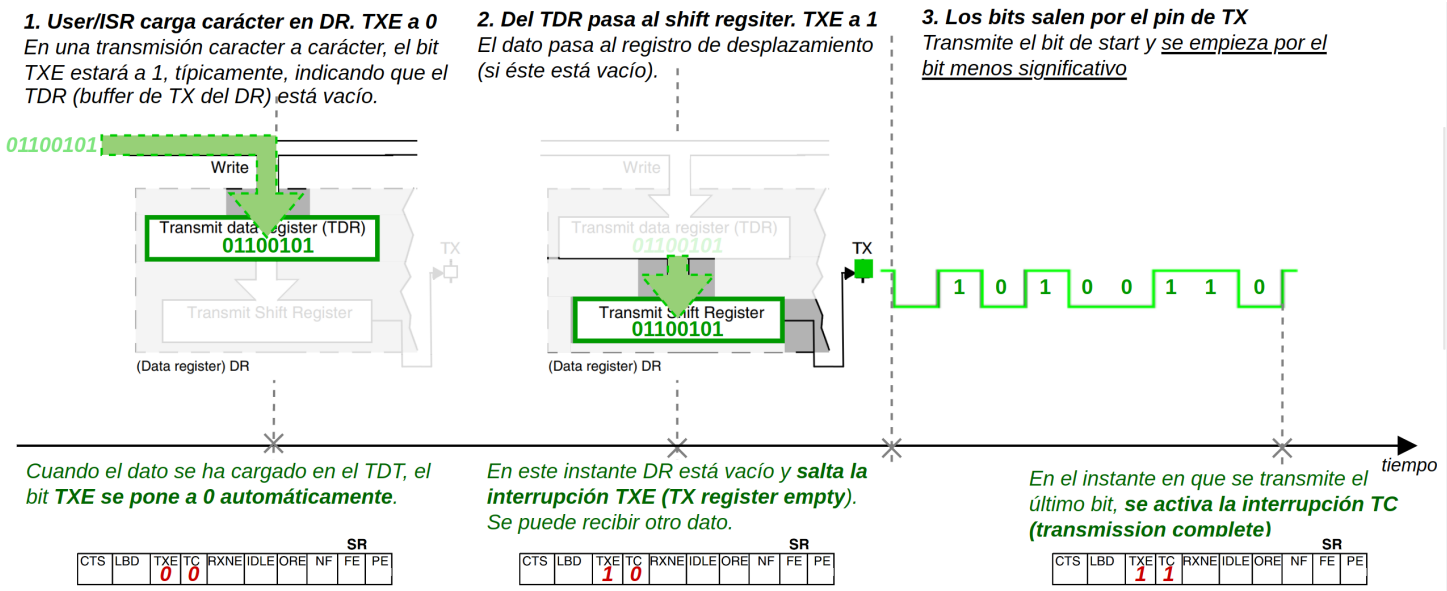


Figura 8.5: Interrupciones en transmisión de la USART.

en `TXE`. Ya se puede volver a escribir el siguiente carácter a transmitir en el registro de datos `USART_DR`.

El envío del siguiente carácter se hará, típicamente, en la rutina de atención a la interrupción de la USART (la ISR), *i.e.* se vuelve al paso `1` de la Figura 8.5, aunque no se pasará el dato al *Transmit Shift Register* hasta que se complete la transmisión del carácter actual.

3. Mientras tanto, los bits salen por el puerto de transmisión, *i.e. salen por el cable*. Después del bit de `start`, se transmiten los bits empezando por el menos significativo. Cuando se transmite el último bit, se envía el bit de `paridad` (si existe) y el de `stop` y se activa el bit `TC` del registro de estado `USART_SR`. Esto indica que la transmisión del carácter actual se ha completado totalmente. Paso `3` de la Figura 8.5.

Merece la pena repetir cuál es la diferencia entre bit de interrupción `TC` (*Transmission Complete*) y el `TXE` (*Transmit Data Register Empty*) en una USART. La diferencia radica en cuándo se activan durante el proceso de transmisión:

- `TXE` se activa cuando el registro de datos de la USART está vacío y listo para ser cargado con el siguiente carácter a transmitir.
- `TC` se activa cuando la transmisión del carácter actual se ha completado totalmente, es decir, cuando el último bit ha sido desplazado fuera del registro de desplazamiento de transmisión.

Por lo tanto, aunque ambos se activan cuando el registro de datos de la USART está vacío, no se activan al mismo tiempo. `TXE` se activa primero, luego se activa `TC`.

8.3. Registros de la USART

Cada microcontrolador implementa un número de periféricos USART. En nuestro caso, el *Datasheet* del microcontrolador **STM32F446RE** nos indica que tiene 4 periféricos USART. Cada uno de ellos tiene sus propios registros de configuración y estado. En la Figura 8.6 se muestra un diagrama de bloques de la USART del microcontrolador **STM32F446RE**. Los registros más importantes con los que trabajaremos se muestran a continuación. El fabricante nos indica en el *Reference Manual* que todos los registros son de 32 bits a los que hay que acceder a la vez.

8.3.1. USART Status Register (USART_SR)

El registro `USART_SR` contiene los flags de estado de la USART. Los flags de estado son bits que se activan cuando se produce un evento. En la Figura 8.7a se muestra

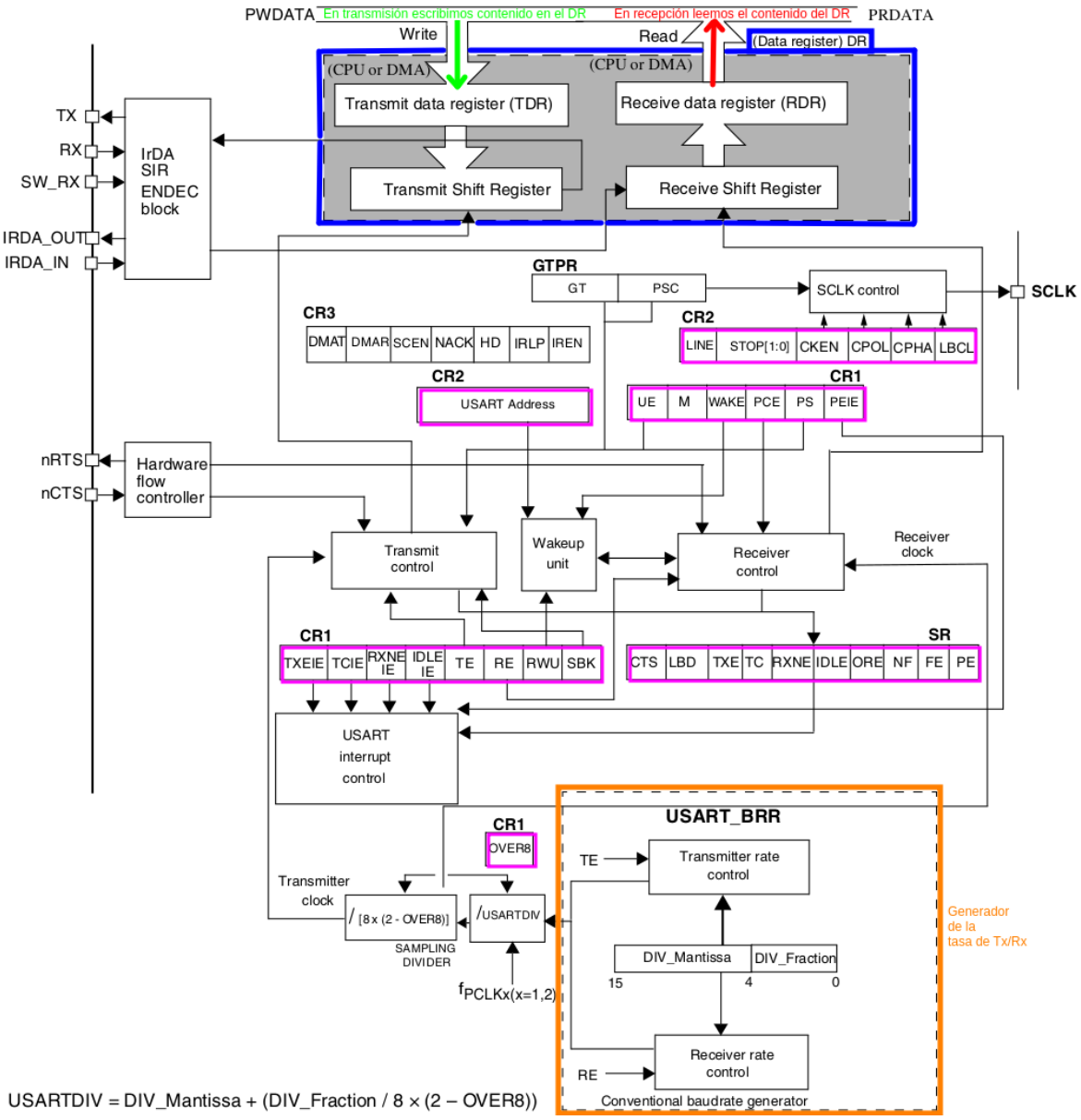


Figura 8.6: Diagrama de bloques de la USART en el **STM32F446RE**.

el registro `USART_SR` del microcontrolador **STM32F446RE**. Los flags de estado más importantes son los siguientes:

- `TXE`: **Transmit data register Empty**. Este bit se activa cuando el registro de transmisión `USART_DR` está vacío y listo para aceptar nuevos datos para la transmisión. Indica que se puede cargar un nuevo byte de datos en el registro DR para que se transmita. Se borra automáticamente, por hardware, escribiendo un dato en el registro `USART_DR`. Este bit lo usaremos en transmisiones de datos simples, que será el caso más común.

  Cuando se necesita enviar un byte de datos, se verifica si el bit `TXE` está activado, lo que indica que el registro de datos de datos (`USART_DR`) está vacío y listo para

aceptar nuevos datos. En ese momento, se carga el byte de datos que se desea transmitir en el registro `USART_DR`.

Después de cargar los datos en el registro `USART_DR`, comienza la transmisión. Los datos se envían secuencialmente, uno a uno, desde este registro a través de la línea de transmisión (*i.e.*, *sale por el cable*). Durante este proceso, el hardware de la USART maneja la transmisión de los bits.

Este bit se activa por hardware cuando el contenido del buffer `TDR` se ha transferido al registro de desplazamiento de transmisión `Transmit Shift Register`. Puede ver estos dos buffer en el diagrama de bloques de la Figura 8.6.

Por ejemplo, para comprobar el estado de este bit en la USART1, lo haríamos de la siguiente manera: `USART1->SR & USART_SR_TXE`; donde `USART1` es un puntero a la estructura de la USART1, y `USART_SR_TXE` es un `#define`, una máscara, del estándar CMSIS que indica la posición del bit `TXE` en el registro `USART_SR`. Si el resultado de la operación es distinto de cero, el bit `TXE` está activo.

- `TC`: **Transmission Complete**. Este bit indica que toda la trama de datos, incluyendo bits de inicio, datos, bits de parada, y posiblemente bits de paridad, se ha transmitido completamente por el puerto de salida, quedando la USART está lista para transmitir una nueva trama.

  Por ejemplo, para comprobar el estado de este bit en la USART1, lo haríamos de la siguiente manera: `USART1->SR & USART_SR_TC`. Si el resultado de la operación es distinto de cero, el bit `TC` está activo.

- `RXNE`: **Read data register Not Empty**. Este bit se activa cuando se ha recibido un byte completo de datos y está listo para ser leído desde el registro de recepción de datos `USART_DR`. Indica que se ha recibido un nuevo byte de datos y que se puede leer desde el registro `USART_DR`. Se borra automáticamente, por hardware, leyendo el registro `USART_DR`. Este bit lo usaremos en recepciones de datos simples, que será el caso más común.

  De forma análoga a los bits anteriores, para comprobar el estado de este bit en la USART1, lo haríamos de la siguiente manera: `USART1->SR & USART_SR_RXNE`. Si el resultado de la operación es distinto de cero, el bit `RXNE` está activo.

8.3.2. USART Data Register (USART_DR)

El registro `USART_DR` es el registro de datos de la USART. En él se almacenan los datos que se van a transmitir o los datos que se han recibido. En la Figura 8.7b se

muestra el registro `USART_DR` del microcontrolador **STM32F446RE**. Este registro es de 32 bits, pero solo se usan los 9 bits menos significativos.

Este registro, internamente está compuesto por dos buffer, uno para la transmisión (TDR) y otro para la recepción (RDR). El buffer TDR proporciona la interfaz entre el bus de datos interno y el registro de desplazamiento de salida (véase la Figura 8.6), y el registro RDR proporciona la interfaz entre el registro de desplazamiento de entrada y el bus interno de datos.

Cuando se habilita el bit de paridad en una comunicación (bit `PCE` a `'1'` en el registro `USART_CR1`), en transmisión, el valor escrito en el Most Significant Bit (Bit Más Significativo) (MSB) —bit 7 u 8 dependiendo de la longitud de los datos— no tiene efecto porque es reemplazado por el bit de paridad. Un ejemplo de escritura en la `USART1` tras comprobar que está vacío el buffer de transmisión, sería de la siguiente manera:

```
1 char data = 0x4A; // Data to send. ASCII code for 'J'
2 if (USART1->SR & USART_SR_TXE) {
3     USART1->DR = data;
4 }
```

De igual forma, en la recepción, si el bit de paridad está activo, el valor leído en el MSB es el bit de paridad recibido. Un ejemplo de lectura en la `USART1` tras comprobar que hay datos, sería de la siguiente manera:

```
1 char data;
2 if (USART1->SR & USART_SR_RXNE) {
3     data = USART1->DR; // Later, use data variable
4 }
```

IMPORTANTE

Preste atención a que los tipos de las variables de datos en transmisión y recepción son `char` y esto es porque típicamente vamos a enviar y recibir caracteres American Standard Code for Information Interchange, Código Estadounidense Estándar para el Intercambio de Información (ASCII). Mire la tabla ASCII de la Figura C.1 y repase la sección *6. Representación: tabla ASCII, conversiones printf* del vídeo de tipos de datos en `C`.

8.3.3. USART Control Register 1 (USART_CR1)

Los registros de control `USART_CR1`, `USART_CR2`, y `USART_CR3` contienen los bits de configuración de la USART. En la Figura 8.8a se muestra el registro `USART_CR1` del microcontrolador **STM32F446RE**. Los bits de configuración a los que más prestaremos atención son los siguientes:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

(a) USART_SR.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	DR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

(b) USART_DR.

Figura 8.7: Registros de estado y de datos de la USART USART_SR y USART_DR.

- OVER8: **Oversampling mode**. Este bit se utiliza para configurar el modo de sobremuestreo de la USART. Si se activa (‘1’), el sobremuestreo que se hace de la línea de transmisión es de *x8*. Por defecto está desactivado, y se hace un sobremuestreo de la línea de transmisión de *x16*.

- UE: **USART Enable**. Habilita o deshabilita el funcionamiento del módulo USART. Si se deshabilita, la USART no funcionará. Es buena práctica deshabilitar la USART mientras se están configurando el resto de bits y registros, y activarlo cuando se haya acabado; como en cualquier periférico.

- M: **Word length**. Este bit determina la longitud del dato transmitido y recibido en bits. Si se activa (‘1’), se selecciona una longitud de 9 bits; si está desactivado (‘0’), se selecciona una longitud de 8 bits. Típicamente se desactiva y se trabaja con 8 bits.

- WAKE: **Wakeup method**. Este bit controla el método de activación del receptor en modo de bajo consumo. Si se activa (‘1’), el receptor se activa cuando se detecta un bit de inicio. Si se desactiva (‘0’), el receptor se activa cuando se detecta un cambio de nivel en la línea de recepción.

  Aunque pueda parecer lo mismo detectar un bit de inicio (de alto a bajo en la línea, ver Figura 8.3), que un cambio de nivel (de alto a bajo o de bajo a alto en la línea), no lo es. En el primer caso, el receptor se activa cuando se detecta un bit de inicio, y se desactiva cuando se detecta un bit de parada. En el segundo caso, el receptor se activa cuando se detecta un cambio de nivel, y se desactiva cuando se detecta otro cambio de nivel. La configuración del bit WAKE es solo un factor de cómo se manejan las interrupciones que activan el módulo USART, pero no suele representar una diferencia significativa en el consumo de energía.

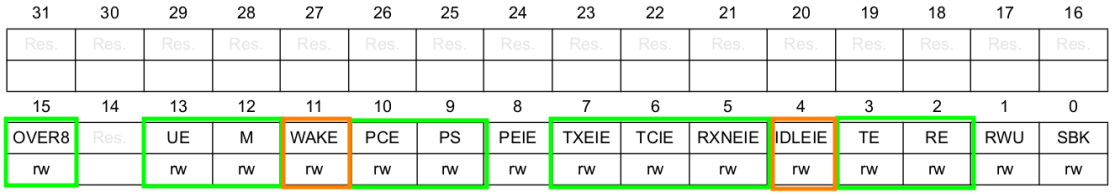
- `PCE`: **Parity Control Enable**: Este bit habilita (`'1'`) o deshabilita (`'0'`) la función de control de paridad en la transmisión de datos.
- `PS`: **Parity Selection**: En combinación con el bit `PCE`, determina si la paridad será par (`'0'`) o impar (`'1'`).
- `TXEIE`: **TXE Interrupt Enable**: Este bit habilita o deshabilita la interrupción asociada con el registro de datos en transmisión cuando está vacío. Si se activa, se genera una interrupción cuando el bit `TXE` del registro `USART_SR` (ver Sección 8.3.1) es (`'1'`).
- `TCIE`: **Transmission Complete Interrupt Enable**: Este bit habilita o deshabilita la interrupción asociada a la transmisión completa de un dato. Si se activa, se genera una interrupción cuando el bit `TC` del registro `USART_SR` (ver Sección 8.3.1) es (`'1'`).
- `RXNEIE`: **RXNE Interrupt Enable**: Este bit habilita o deshabilita la interrupción asociada con la recepción de datos. Si se activa, se genera una interrupción cuando el bit `RXNE` del registro `USART_SR` (ver Sección 8.3.1) es (`'1'`).
- `IDLEIE`: **IDLE Interrupt Enable**: Este bit habilita o deshabilita la interrupción generada por la detección de una línea inactiva (*idle line*) en la comunicación serie.
- `TE`: **Transmitter Enable**: Este bit permite habilitar o deshabilitar el transmisor de la USART.
- `RE`: **Receiver Enable**: Este bit permite habilitar o deshabilitar el receptor de la USART.

Puede ver un ejemplo de configuración de la `USART1` en la Sección 8.4.

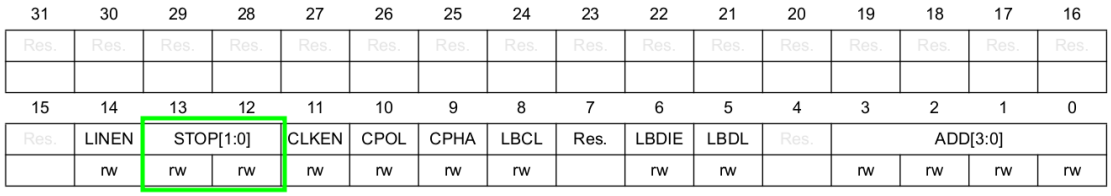
8.3.4. USART Control Register 2 (USART_CR2)

El registro `USART_CR2` contiene más bits de configuración de la USART. En la Figura 8.8b se muestra el registro `USART_CR2` del microcontrolador **STM32F446RE**. En este caso nos interesan los siguientes bits:

- `STOP`: **STOP bits**. Este campo formado por 2 bits, determina el número de bits de parada en el comando de transmisión. Las dos configuraciones más comunes son (`'00'`), para elegir una longitud de 1 bit de parada, y `'10'` para elegir una longitud de 2 bits de parada. Típicamente se desactiva y se trabaja con 1 bit de parada.



(a) `USART_CR1`.



(b) `USART_CR2`.

Figura 8.8: Registros de control de la USART USART_CR1 y USART_CR2.

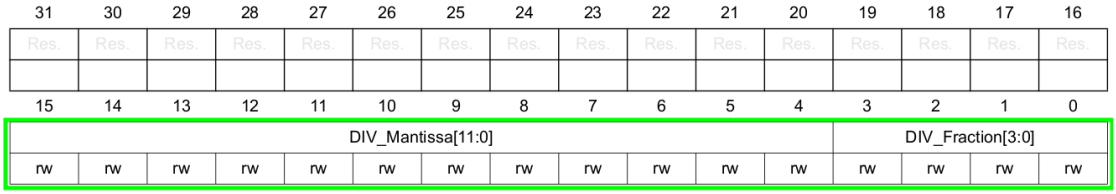


Figura 8.9: Registro de configuración de la velocidad de transmisión/ recepción USART_BRR.

Interesante comentar el campo `ADDR` de este registro. Este campo se usa para la detección de direcciones en modo multiprocesador. En este modo, la USART puede detectar si la dirección de recepción coincide con la dirección de la USART. Este modo no lo vamos a usar, pero es interesante conocer que a las USART se le pueden dar direcciones, útil en comunicaciones multiprocesador.

8.3.5. USART Baud Rate Register (USART_BRR)

El registro `USART_BRR` contiene los campos de configuración de la velocidad de transmisión/ recepción de la USART. El *baud rate* es común para la transmisión y la recepción. Se configura mediante el registro `USART_BRR` que tiene 2 campos. en los 16 bits menos significativos. De éstos, los 12 bits más significativos se usan para la parte entera (campo `DIV_Mantissa`) y los 4 menos significativos para la parte decimal (campo `DIV_Fraction`). En la Figura 8.9 se muestra el registro `USART_BRR` del microcontrolador **STM32F446RE**.

Algunas velocidades de comunicación habituales son 1200, 2400, 9600, 19200, 38400, 57600 o 115200 baudios, o lo que es lo mismo, 1.2, 2.4, 9.6, 19.2, 38.4, 57.6 y 115.2 kbps (*kilobits por segundo*) —ya que 1 símbolo se representa por 1 bit—. Estas velocidades

casi estandarizadas vienen de las velocidades históricas de los teletipos, pero se pueden usar otras. Para conseguir esos números se usan divisores de la frecuencia de reloj del microcontrolador, por lo que podrá intuir que, dependiendo de la velocidad deseada, los valores rara vez serán exactos, tendremos decimales, y habrá un error (*desviación*). Típicamente, los sistemas tienen una tolerancia de error en recepción del 1 % al 5 %.

El método de sobremuestreo, antes mencionado, se elige mediante el bit `OVER8` del registro `USART_CR1` (ver Sección 8.3.3). Si se activa, se usa un sobremuestreo de 8 veces la velocidad de transmisión, y si se desactiva (valor por defecto en nuestro **STM32F446RE**), se usa un sobremuestreo de 16 veces la velocidad de transmisión.

Para configurar el *baud rate* se utiliza la siguiente fórmula:

$$baudrate = \frac{f_{CLK}}{8 \times (2 - OVER8) \times USARTDIV}, \tag{8.1}$$

donde `USARTDIV` es un número de punto fijo sin signo que se codifica en el registro `USART_BRR` (ver Sección 8.3.5).

8.3.5.1. Ejemplos de cálculo del baud rate

A continuación se muestran unos ejemplos de configuración del *baud rate*. Se va a trabajar con el *oversampling* por defecto (`OVER8=0`). Se recomienda ver en detalle los ejemplos del *reference manual* [6] para ver otras situaciones.

Ejemplo de cálculo de la velocidad real de transmisión sin error

Supongamos una frecuencia de reloj del microcontrolador de 16 MHz y sobremuestreo del *baud rate* de 16 (`OVER8=0`). Para configurar un *baud rate* de 1200 baudios, se calcula el valor del registro `USART_BRR` usando la Ec. 8.1:

$$baudrate = \frac{f_{CLK}}{8 \times (2 - OVER8) \times USARTDIV} = \frac{16\ MHz}{8 \times (2 - 0) \times USARTDIV} = 1200\ baudios, \tag{8.2}$$

por lo que despejando el valor de `USARTDIV` en decimal (`0d` indica base 10) se obtiene:

$$USARTDIV = \frac{f_{CLK}}{8 \times (2 - OVER8) \times baudrate} = \frac{16\ MHz}{8 \times (2 - 0) \times 1200\ baudios} = \texttt{0d}833{,}34. \tag{8.3}$$

Para representar ese número en el registro `USART_BRR` tenemos que convertir a hexadecimal (o binario) la parte entera y la parte decimal por separado. Para la parte entera tenemos 12 bits, y 4 para la parte decimal.

Para la parte entera, con 12 bits, tenemos:

$$\texttt{0d}833 = \texttt{0x}0341 = \texttt{0b}0011\ 0100\ 0001 (\texttt{0b} \text{ indica base 2, binario}). \tag{8.4}$$

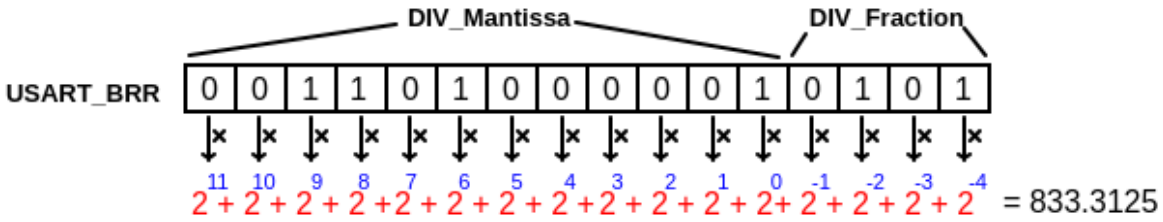


Figura 8.10: Ejemplo de cálculo de la velocidad real de transmisión sin error.

Para convertir el número fraccionario 0,34 en binario podemos usar el *método de la multiplicación sucesiva por 2*. Para ello, multiplicamos la parte decimal por 2 y nos quedamos con la parte entera (que será `0`, o `1`). El resultado de la multiplicación lo volvemos a multiplicar por 2 y nos quedamos con la parte entera. Repetimos el proceso hasta que la parte decimal sea 0 o hasta que tengamos los bits que necesitamos. En nuestro caso, 4 bits para la parte decimal, por lo que repetimos el proceso 4 veces:

$$\begin{aligned} 0{,}34 \times 2 &= 0{,}68 \rightarrow 0 \\ 0{,}68 \times 2 &= 1{,}36 \rightarrow 1 \\ 0{,}36 \times 2 &= 0{,}72 \rightarrow 0 \\ 0{,}72 \times 2 &= 1{,}44 \rightarrow 1 \end{aligned} \tag{8.5}$$

Así, la la parte decimal en binario será `0b0101`. El valor del registro `USART_BRR` queda como: `USART_BRR` = `0x3415` = `0b0011 0100 0001 0101`.

Para ver cuánto error hemos cometido, tenemos que calcular el *baud rate* real. Para ello, tenemos que volver a convertir el valor a decimal, como se muestra en la Figura 8.10.

Recalculando el *baud rate*:

$$baudrate = \frac{f_{CLK}}{8 \times (2 - OVER8) \times USARTDIV} = \frac{16\ MHz}{8 \times (2-0) \times 833{,}3125} = 1200{,}03\ baudios, \tag{8.6}$$

lo que nos indica que el error es de:

$$error = \frac{1200{,}03 - 1200}{1200} = 0{,}0025\,\%, \tag{8.7}$$

que es un error despreciable dentro de la tolerancia de error de los sistemas.

Ejemplo de cálculo de la velocidad real de transmisión con error

Repetimos el ejemplo anterior, pero para un *baud rate* de 921600 baudios. Con la misma frecuencia de reloj del microcontrolador de 16 MHz y sobremuestreo de 16 (`OVER8=0`), tenemos:

$$baudrate = \frac{16\ MHz}{8 \times (2-0) \times USARTDIV} = 921600\ baudios, \tag{8.8}$$

por lo que despejando el valor de `USARTDIV` se obtiene:

$$USARTDIV = \frac{16\ MHz}{8 \times (2-0) \times 921600\ baudios} = \texttt{0d1,085} \tag{8.9}$$

Para la parte entera, con 12 bits, tenemos:

$$\texttt{0d1} = \texttt{0x1} = \texttt{0b0000 0000 0001}. \tag{8.10}$$

Convertimos el número 0,085 en binario usando el *método de la multiplicación sucesiva por 2*. Con 4 bits para la parte decimal, repetimos el proceso 4 veces:

$$\begin{aligned} 0{,}085 \times 2 &= 0{,}17 \rightarrow 0 \\ 0{,}17 \times 2 &= 0{,}34 \rightarrow 0 \\ 0{,}34 \times 2 &= 0{,}68 \rightarrow 0 \\ 0{,}68 \times 2 &= 1{,}36 \rightarrow 1 \end{aligned} \tag{8.11}$$

por lo que la parte decimal en binario es `0b0001`. El valor del registro `USART_BRR` queda como: `USART_BRR` = `0x0011` = `0b0000 0000 0001 0001`.

Para ver cuánto error hemos cometido, recalculamos el *baud rate* real volviendo a convertir el valor a decimal. Haciendo las cuentas tendríamos la suma de $2^0 + 2^{-4} = 1{,}0625$. Recalculando el *baud rate*:

$$baudrate = \frac{16\ MHz}{8 \times (2-0) \times \texttt{0d1,0625}} = 941176,47\ baudios, \tag{8.12}$$

lo que nos indica que el error es de:

$$error = \frac{941176{,}47 - 921600}{921600} = 2{,}12\,\%, \tag{8.13}$$

que es un error relativamente grande y, aunque se encuentra en los márgenes habituales aceptados por los sistemas, pudiera provocar problemas de comunicación.

8.4. Ejemplo de comunicación USART por interrupciones

En este ejemplo vamos a configurar la `USART1` del microcontrolador STM32F446RE para comunicarse bidireccionalmente con un ordenador mediante un conversor USB-serie. El microcontrolador podrá enviar mensajes al ordenador y viceversa. Para ello, se configura la `USART1` en

modo asíncrono, con: una velocidad de transmisión de 1200 baudios, 8 bits de datos, sin bit de paridad, y 1 bit de parada. Esta configuración se representa como `1200-8-N-1`. Se va a configurar la ISR para que nos interrumpa cuando se reciba un dato y cuando se haya completado una transmisión.

Lo que hay que considerar principalmente cuando realice una comunicación serie es:

1. Configuración de las GPIOs de la USART elegida (Sección 8.4.1).
2. Configuración de la comunicación e interrupciones USART (Sección 8.4.2).
3. Configuración de la ISR de la USART (Sección 8.4.3).
4. Códigos que gestionan la transmisión de datos y recepción de datos (Sección 8.4.4).

A continuación intercalaremos explicación del ejemplo con el código que para nuestro **STM32F446RE** usando máscaras que definamos nosotros, o usando las definidas en el estándar CMSIS por fabricante, si es más conveniente o cómodo[4].

8.4.1. Código de configuración de las GPIO de la USART

Como ya se ha mencionado, configurar la USART implica también configurar las GPIOs por las que se transmitirán y recibirán los datos. **Las GPIO se configuran en modo alternativo**.

Para ver qué GPIOs de la placa se usan para cualquier USART tenemos que mirar el *pinout* o *datasheet* de la misma. En el caso de la placa **Nucleo-STM32F446RE**, el *pinout* se muestra en la Figura 1.11 y el *datasheet* en [5]. Si nos fijamos en los códigos amarillos de la Figura 8.11 vemos que son los pines `PA9` para transmisión (también puede ser `PB6`, porque está duplicado) y `PA10` para recepción de la `USART1`. Así pues, vamos a trabajar con las `GPIOA_9` y `GPIOA_10`.

Para configurar las GPIOs de la USART, se siguen los siguientes pasos:

1. Se habilita el reloj de las GPIO.
2. Se configura el modo de las GPIO como alternativo (no entrada o salida).
3. Se configura la resistencia de *pull-up-down* de las GPIO como *pull-up*, *i.e.* por defecto la línea de transmisión y recepción está en alto.

[4]Mire el fichero `stm32f446xx.h` para ver la definición de las macros de los registros y campos de los mismos según el fabricante.

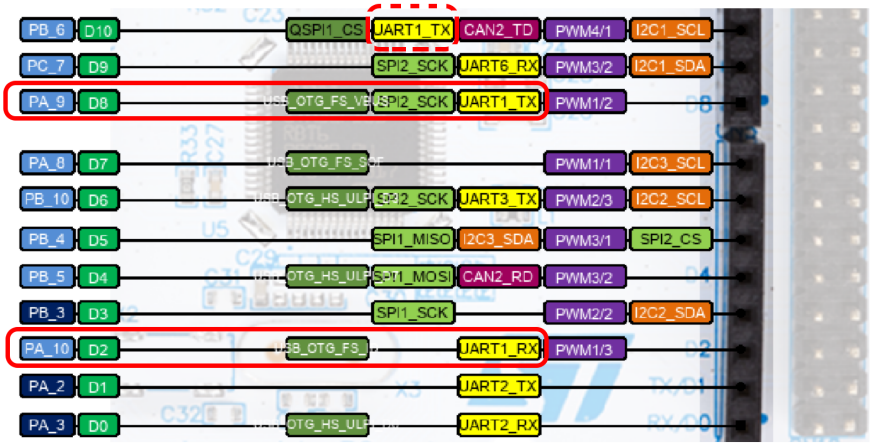


Figura 8.11: GPIOs de la USART1 de la placa **Nucleo-STM32F446RE**.

4. Se configura la GPIO de transmisión como la función alternativa `USART_TX`.

5. Se configura la GPIO de recepción como la función alternativa `USART_RX`.

ATENCIÓN

La USART2 no se puede utilizar por defecto con la **Nucleo-STM32** porque los pines están siendo usados por el programador STLINK. Si se quiere usar, hay que soldar la unión (no recomendado). Ver [2] en el que se explica en detalle el extracto de la Figura 8.12.

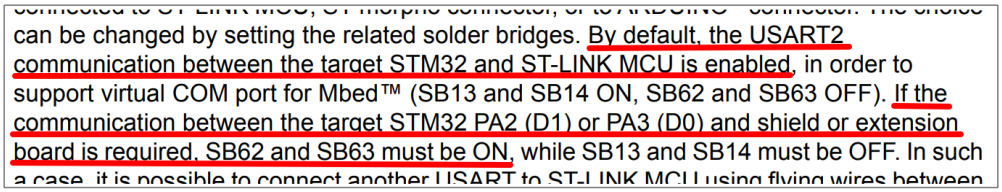

can be changed by setting the related solder bridges. By default, the USART2 communication between the target STM32 and ST-LINK MCU is enabled, in order to support virtual COM port for Mbed™ (SB13 and SB14 ON, SB62 and SB63 OFF). If the communication between the target STM32 PA2 (D1) or PA3 (D0) and shield or extension board is required, SB62 and SB63 must be ON, while SB13 and SB14 must be OFF. In such

Figura 8.12: Advertencia de uso de la USART2 en la placa.

Para configurar las GPIOs podemos implementar el código de la función `port_usart1_gpio_setup()` que se muestra a continuación. Asegúrese de tener soltura con la notación binaria, hexadecimal y decimal, y con las máscaras de bits.

```
1 /* Definiciones de nuestras propias mascaras */
2 #define LINEA_9 9  /* Para el TX */
3 #define LINEA_10 10  /* Para el RX */
4
5 #define MODE_LINEA_9_MASK (0x03 << 2*LINEA_9)  /* Desplaza los bits 0b11 a la
      izquierda 2*9=18 posiciones */
6 #define MODE_LINEA_10_MASK (0x03 << 2*LINEA_10)  /* Desplaza los bits 0b11 a la
      izquierda 2*10=20 posiciones */
7
```

```
8  #define PUPDR_LINEA_9_MASK (0x03 << 2*LINEA_9)  /* Desplaza los bits 0b11 a la
       izquierda 2*9=18 posiciones */
9  #define PUPDR_LINEA_10_MASK (0x03 << 2*LINEA_10)  /* Desplaza los bits 0b11 a la
       izquierda 2*10=20 posiciones */
10
11 #define MODE_ALTERNATE 0x02U  /* Modo alternativo */
12 #define MODE_PULL_UP 0x01U  /* Modo pull-up */
13
14 #define ALT_FUNC_USART1_TX 0x07U   /* AF7 para funcion alternativa USART1 TX segun
       datasheet */
15 #define ALT_FUNC_USART1_RX 0x07U   /* AF7 para funcion alternativa USART1 RX segun
       datasheet */
16
17 #define AFRH_LINEA_9_MASK (0x0F << 4*(LINEA_9 & 0x7U)) /* Desplaza los bits 0b1111
       a la izquierda 4*(0x9 & 0x7)=4 posiciones */
18 #define AFRH_LINEA_10_MASK (0x0F << 4*(LINEA_10 & 0x7U)) /* Desplaza los bits 0
       b1111 a la izquierda 4*(0xA & 0x7)=8 posiciones */
19
20 void port_usart1_gpio_setup()
21 {
22     /* Configura PA9 (TX) */
23     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Habilita el reloj de la GPIOA (uso
           macro de CMSIS)
24
25     /* Poner modo alternativo */
26     GPIOA->MODER &= ~MODE_LINEA_9_MASK; // Asegura que los bits estan limpios
27     GPIOA->MODER |= (MODE_ALTERNATE << 2*LINEA_9); // Pon modo alternativo: 0b10
           <<2*9=0x00040000
28
29     /* Poner a pull-up */
30     GPIOA->PUPDR &= ~PUPDR_LINEA_9_MASK; // Asegura que los bits estan limpios
31     GPIOA->PUPDR |= (MODE_PULL_UP << 2*LINEA_9); // Pon pull-up: 0b01<<2*9=0
           x00040000
32
33     /* Configura la function alternativa para trabajar como USART1 TX */
34     /* Como es el pin 9, trabajamos con el registro AFRH (high) */
35     /* Miramos Table 11 del Datasheet para see que USART1 TX es la funcion
           alternativa 7 (AF7) */
36     GPIOA->AFR[1] &= ~AFRH_LINEA_9_MASK; // Asegura que los bits estan limpios
37     GPIOA->AFR[1] |= (ALT_FUNC_USART1_TX << 4*(LINEA_9 & 0x7U)); // Pon AF7: 0x07
           <<(4*(0b1001 & 0b0111))=0x0070
38
39
40     /* Configure PA10 (RX) */
41     /* Hay que habilitar el reloj de la GPIOA (pero ya esta activado) */
42     /* Poner modo alternativo */
43     GPIOA->MODER &= ~MODE_LINEA_10_MASK; // Asegura que los bits estan limpios
44     GPIOA->MODER |= (MODE_ALTERNATE << 2*LINEA_10); // Pon modo alternativo: 0b10
           <<2*10=0x00100000
45
46     /* Poner a pull-up */
47     GPIOA->PUPDR &= ~PUPDR_LINEA_10_MASK; // Asegura que los bits estan limpios
48     GPIOA->PUPDR |= (MODE_PULL_UP << 2*LINEA_10); // Pon pull-up: 0b01<<2*10=0
           x00100000
49
50     /* Configura la function alternativa para trabajar como USART1 RX */
51     /* Como es el pin 10, trabajamos con el registro AFRH (high) */
52     /* Miramos Table 11 del Datasheet para see que USART1 RX es la funcion
           alternativa 7 (AF7) */
```

```
53      GPIOA->AFR[1] &= ~AFRH_LINEA_10_MASK; // Asegura que los bits estan limpios
54      GPIOA->AFR[1] |= (ALT_FUNC_USART1_RX << 4*(LINEA_10 & 0x7U)); // Pon AF7: 0x07
            <<(4*(0b1010 & 0b0111))=0x0700
55  }
```

8.4.2. Código de configuración de comunicación e interrupción USART

Para configurar la comunicación de la USART se siguen los siguientes pasos:

1. Se habilita el reloj de la USART.
2. Se deshabilita la USART para tocar los registros.
3. Se configura la longitud de los datos.
4. Se configura el bit de parada.
5. Se configura el bit de paridad.
6. Se configura el *oversampling* (si es necesario).
7. Se configura el *baud rate* (ver ejemplo de cálculo en Sección 8.3.5.1).
8. Se habilita la transmisión y la recepción (o solo una de las dos, según sea necesario).
9. Se habilitan las interrupciones de recepción (si es necesario).
10. Se deshabilitan las interrupciones de transmisión (solo se habilita si es necesario).

    Las interrupciones que podemos tener en transmisión son 2: cuando el buffer de transmisión está vacío (`TXE`) y cuando se ha completado la transmisión (`TC`). Generalmente se activan y desactivan conjuntamente, pero la interrupción de buffer de transmisión vacío es bastante molesta porque suele el estado habitual del buffer es estar vacío, por lo que se suele desactivar cuando no es necesario.
11. Se configura la prioridad de la interrupción de la USART.
12. Se habilitan las interrupciones de la USART globalmente.
13. Se habilita la USART.

Podemos implementar el código que se muestra a continuación en la función `usart_config()`. Recuerde que en este ejemplo, se configura la `USART1` en modo asíncrono, con: una velocidad de transmisión de 1200 baudios, 8 bits de datos, sin bit de paridad y 1 bit de parada (`1200-8-N-1`).

```
1  void usart_config()
2  {
3      /* Primero habilitamos el reloj de perifericos de la USART1 */
4      RCC->APB2ENR |= RCC_APB2ENR_USART1EN;
5
6      /* Deshabilitamos la USART1 para tocar los registros */
7      USART1->CR1 &= ~USART_CR1_UE; // USART_CR1_UE es de CMSIS (ver stm32f446xx.h)
8
9      /* Configuramos la longitud de los datos a 8 bits */
10     USART1->CR1 &= ~USART_CR1_M; // Limpiamos el bit M (8 bits)
11
12     /* Configuramos el bit de parada a 1 bit */
13     USART1->CR2 &= ~USART_CR2_STOP; // Limpiamos los bits STOP (1 bit)
14
15     /* Configuramos el bit de paridad a no paridad */
16     USART1->CR1 &= ~USART_CR1_PCE; // Limpiamos el bit PCE (no paridad)
17
18     /* Aseguramos oversampling de 16 (por defecto) */
19     USART1->CR1 &= ~USART_CR1_OVER8; // Limpiamos el bit OVER8 (oversampling de 16)
20
21     /* Configuramos el baud rate (ver ejemplo de calculo) */
22     USART1->BRR = 0x3415; // 1200 baudios
23
24     /* Habilitamos la transmision y la recepcion */
25     USART1->CR1 |= USART_CR1_TE | USART_CR1_RE;
26
27     /* Habilitamos las interrupcion de recepcion */
28     USART1->CR1 |= USART_CR1_RXNEIE; // Habilitamos la interrupcion de recepcion
29
30     /* No habilitamos las interrupciones de transmision generalmente al inicio,
           sino solo cuando es necesario */
31     USART1->CR1 &= ~(USART_CR1_TXEIE | USART_CR1_TCIE); // Deshabilitamos las
           interrupciones de buffer vacio y transmision completa, por ahora
32
33     /* Habilitamos las interrupciones de la USART1 globalmente */
34     NVIC_SetPriority(USART1_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),
           1, 0)); /* Prioridad 1, sub-prioridad 0, por ejemplo */
35     NVIC_EnableIRQ(USART1_IRQn); // Habilitamos la interrupcion de la USART1
36
37     /* Por ultimo, habilitamos la USART1 */
38     USART1->CR1 |= USART_CR1_UE;
39 }
```

8.4.3. Código de la ISR de la USART

Típicamente, la USART nos va a interrumpir cuando se recibe un dato, cuando el buffer de datos está vacío, o cuando enviamos un dato y el envío se ha completado, o cuando se detecta una línea inactiva (*idle line*) en la comunicación serie (menos habitual).

La ISR de la USART dependerá del caso de uso concreto, pero en general, si se tienen activas las tres interrupciones mencionadas, se siguen los siguientes pasos:

- Si está habilitada la interrupción de recepción (bit `RXNEIE` del registro `USART_CR1`):

1. Se comprueba que el bit `RXNE` del registro `USART_SR` está activo, indicando que hay datos por leer (ver Sección 8.3.1).
2. Se lee el dato recibido del registro `USART_DR` (ver Sección 8.3.2). Típicamente, en una variable global o llamando a una función que gestione el dato recibido, porque las ISR no pueden recibir argumentos.
3. El bit `RXNE` se limpia automáticamente cuando se lee el registro `USART_DR`, por lo que no es necesario forzar su valor a 0, típicamente.

- Si está habilitada la interrupción de buffer de transmisión vacío (bit `TXEIE` del registro `USART_CR1`):

1. Se comprueba que el bit `TXE` del registro `USART_SR` está activo, indicando que el buffer está vacío y se puede enviar un dato (ver Sección 8.3.1).
2. Se carga el dato a enviar en el registro `USART_DR` (ver Sección 8.3.2). El dato está en una variable global o se llama a una función que gestione el dato a enviar, porque las ISR no pueden recibir argumentos.
3. El bit `TXE` se limpia automáticamente cuando se carga el dato en `USART_DR`, por lo que no es necesario forzar su valor a 0, típicamente.

IMPORTANTE

Si no está habilitada la interrupción de *buffer de transmisión vacío*, conviene que antes de cargar el dato en el registro `USART_DR` se compruebe que el buffer no contiene nada, mirando que el bit `TXE=1`. Así, evitamos que se sobrescriba un dato que todavía no se haya enviado.

- Si está habilitada la interrupción de transmisión completa (bit `TCIE` del registro `USART_CR1`):

1. Se comprueba que el bit `TC` del registro `USART_SR` está activo, indicando que se ha completado la transmisión (ver Sección 8.3.1).
2. Se limpia el flag de interrupción de transmisión completa `TC` para estar preparado para la siguiente transmisión.

IMPORTANTE

Cuando entramos en una ISR no sabemos por qué interrupcion hemos llegado hasta ella, y tenemos que comprobar todas aquellas que tengamos habilitadas.

Si están las tres interrupciones anteriormente habilitadas (`RXNEIE`, `TXEIE` y `TCIE`), la ISR de la `USART1` quedaría como se muestra a continuación. Recuerde que el nombre de la ISR de la `USART1` es `USART1_IRQHandler()` y no se puede cambiar.

```
1  char data_rx; // Variable global para almacenar el dato recibido
2  char data_tx = 'H'; // Variable global con el dato a enviar. Ejemplo, una 'H'
3
4  void USART1_IRQHandler(void) {
5      /* Cuando entramos en una ISR no sabemos por que interrupcion hemos llegado
           hasta ella y tenemos que comprobar todas aquellas que tengamos habilitadas
           */
6
7      /* Si esta habilitada la interrupcion de recepcion */
8      if (USART1->CR1 & USART_CR1_RXNEIE) {
9          /* Compruebo que la interrupcion es por dato recibido */
10         if (USART1->SR & USART_SR_RXNE) {
11             data_rx = USART1->DR; // Leo el dato recibido
12         }
13     }
14
15     /* Si esta habilitada la interrupcion de buffer de transmision vacio */
16     if (USART1->CR1 & USART_CR1_TXEIE) {
17         /* Compruebo que la interrupcion es por buffer de transmision vacio */
18         if (USART1->SR & USART_SR_TXE) {
19             USART1->DR = data_tx; // Escribo el dato a enviar
20         }
21     }
22
23     /* Si esta habilitada la interrupcion de transmision completa */
24     if (USART1->CR1 & USART_CR1_TCIE) {
25         /* Compruebo que la interrupcion es por transmision completa */
26         if (USART1->SR & USART_SR_TC) {
27             USART1->SR &= ~USART_SR_TC; // Limpio el flag de transmision completa
28         }
29     }
30 }
```

8.4.4. Código para gestionar el envío y recepción de datos

Si no estamos trabajando con las interrupciones, y queremos enviar datos, se siguen los siguientes pasos:

1. Se comprueba que el registro `USART1_SR` (ver Sección 8.3.1) está vacío (`TXE=1`).
2. Se carga el dato a enviar en el registro `USART1_DR` (ver Sección 8.3.2).

Pudiendo quedar un código como este:

```
1  char data_tx = 'H'; // Variable con el dato a enviar. Ejemplo, una 'H'
2  while (!(USART1->SR & USART_SR_TXE)); // Espera (obligatoria) a que el buffer de
       transmision este vacio
3  USART1->DR = data_tx; // Carga el dato a enviar en el registro de datos
```

Si no estamos trabajando con las interrupciones, y queremos recibir datos, se siguen los siguientes pasos:

1. Se comprueba que el registro `USART1_SR` (ver Sección 8.3.1) tiene datos (`RXNE=1`).

2. Se lee el dato recibido del registro `USART1_DR` (ver Sección 8.3.2).

Pudiendo quedar un código como este:

```
1 char data_rx; // Variable para almacenar el dato recibido
2 while (!(USART1->SR & USART_SR_RXNE)); // Espera (NO obligatoria) a que el buffer
      de recepcion tenga datos
3 data_rx = USART1->DR; // Lee el dato recibido del registro de datos
```

Vea el video de Youtube de este enlace para ver una demo de comunicacion USART con interrupciones y el codigo completo de las secciones mostradas en este capitulo en este proyecto de GitHub[5].

[5]Vídeo: `https://www.youtube.com/watch?v=e2QBfF0xO4g&ab_channel=SDG2DIE-UPM`. Código: `https://github.com/sdg2DieUpm/example-MatrixMCU-usart`

Capítulo 9

Desarrollando para Nucleo-STM32

Josué Pagán
Pedro José Malagón
Román Cárdenas

En este capítulo vamos a centrarnos en las herramientas que nos van a permitir desarrollar los códigos de nuestros programas, y explicaremos la estructura y metodología de proyecto que usaremos. Pero antes de empezar, si no lo tiene claro aún, es buen momento para poner en orden estos tres conceptos: (i) HW, (ii) FW, y (iii) SW.

Del HW ya hemos hablado bastante; es nuestra placa **Nucleo-STM32F446RE**, con el microcontrolador **STM32F446RE**, y existen numerosas herramientas para desarrollar sobre este HW. Pero, ¿qué vamos a desarrollar FW o SW? ¿Cuál es la diferencia entre ellos? Generalmente llamamos FW al código de bajo nivel que interactúa directamente con el HW, con sus registros, y SW al código de más alto nivel que no interactúa directamente con el HW, sino con el sistema operativo.

Cuando el fabricante nos da librerías de acceso a periféricos cuyo manejo a bajo nivel sería muy tedioso, nos proporcionan lo que llamamos el *middleware*, que nos da flexibilidad en el manejo de funciones estándares para dicho periférico, *e.g.*, un módulo WiFi, Ethernet, Bluetooth,*etc.* El middleware es código que actúa como un puente entre el FW (y por ende, el HW) y las aplicaciones de alto nivel. El middleware proporciona una interfaz de programación de aplicaciones (Application Programming Interface (Interfaz de Programación de Aplicaciones) (API)) que simplifica el acceso a funciones de hardware específicas.

¿Se ha dicho sistema operativo en un microcontrolador? ¿Qué diferencia hay entre un microcontrolador y un microprocesador? Hace unos años se podría decir que un microprocesador es un microcontrolador más potente que tiene sistema operativo. Pero hoy día la última afirmación no es del todo correcta. Los microcontroladores de hoy día son tan potentes como los microprocesadores de hace unos años, y sí, ya hay sistemas operativos para microcontrolador. Es cierto que un microprocesador es más

potente, tiene más núcleos, algunos son vectoriales... los encontraremos en los ordenadores y equipos de alto rendimiento, ejecuten, o no, tareas específicas. Y también es cierto que los sistemas operativos que nos pueden venir a la cabeza (*e.g.*, `Android, Windows, MacOS`, *etc.*) no son los mismos que encontraremos en un microcontrolador, y que típicamente se les conoce como sistemas operativos embebidos o Real Time Operating System (Sistema Operativo de Tiempo Real) (RTOS). El *tiempo real* quiere decir *tiempo acotado*, *i.e.*, que aseguramos que una tarea va a cumplirse con *plazos*. Esta característica no se cumple generalmente en los sistemas operativos en microprocesadores.

El sistema operativo nos sirve para abstraernos del HW y centrarnos en el desarrollo de la aplicación. Es por ello que, en los últimos años, con el *boom* de los dispositivos IoT, es muy normal que un desarrollador quiera centrarse en hacer nuevas aplicaciones sin pelearse en los detalles del HW del dispositivo para el que va a trabajar. Desarrollar en un RTOS para microcontrolador nos ayuda a ello. Algunos de los nombres más populares son: `FreeRTOS (Amazon), Zephyr (Linux Foundation), MbedOS (ARM), embOS (SEGGER) o Keil RTX (ARM)`.

Pero además del sistema operativo, hay otras diferencias entre microcontrolador y microprocesador. Los microcontroladores suelen tener más periféricos integrados, como conversores analógico-digitales, interfaces de comunicación, *etc.* y suelen tener más memoria integrada. Los microprocesadores suelen tener más núcleos, y más potencia de cálculo. Pero, ¿qué es más importante? ¿Qué es mejor? Pues depende del caso de uso. Si queremos un dispositivo que haga una tarea específica, como un termostato, un microcontrolador es más que suficiente. Si queremos un dispositivo que haga muchas tareas, como un ordenador, un microprocesador es más adecuado. Si queremos un dispositivo que haga muchas tareas, pero que no necesite de un sistema operativo, un microcontrolador con sin sistema operativo es una buena opción. Si queremos un dispositivo que haga muchas tareas, y que necesite de un sistema operativo, un microprocesador con un sistema operativo en tiempo real es una buena opción. Y así podríamos seguir con más ejemplos.

9.1. Programación bare metal y HAL

En este libro no vamos a trabajar a nivel de SW con RTOS (esto podrá hacerlo en estudios de cursos de especialidad). Nosotros **vamos a trabajar a nivel de FW**, usando el lenguaje de programación `C`. Así que en este punto nos toca introducir otros dos conceptos: (i) programación *bare metal* o *sobre máquina desnuda*, y (ii) la Hardware Abstraction Layer (Capa de Abstracción Hardware) (HAL).

Cuando hablamos de **programación *bare metal*** estamos haciendo referencia a que nuestro código trabaja directamente con los registros de nuestro microcontrolador,

con sus nombres, sus direcciones de memoria... Es una programación muy cercana al HW, como vimos en los ejemplos del Capítulo 3. No vamos a programar en ensamblador —muy útil cuando se necesita un control fino—, sino en C. Pero tampoco vamos a llegar al nivel de tener que definir nosotros las zonas de memoria de los registros, sino que nos apoyaremos en el estándar CMSIS. CMSIS es una API[1], una interfaz con la que nos comunicaremos para acceder a los registros del microcontrolador. Está desarrollada por y para arquitecturas ARM Cortex, y cada uno de los fabricantes —en nuestro caso, ST— adapta la interfaz a sus microcontroladores. Por ejemplo, CMSIS nos permite encender un LED escribiendo:

```
GPIOA->ODR |= 0x02;
```

porque en el fichero `stm32f446xx.h` define `GPIOA` como un puntero a un `struct` de tipo `GPIO_TypeDef`, que contiene todos los registros de la GPIO apuntando a la dirección de memoria correspondiente.

Cuando hablamos de **programación haciendo uso de la HAL**, estamos refiriéndonos a una serie de códigos —entendidos como librerías— que proporcionan los fabricantes para que los desarrolladores trabajen más rápido. Como su propio nombre indica, las HAL son una capa de abstracción del HW, *i.e.*, nos permite escribir y reutilizar código sin pensar en el HW que subyace. Por ejemplo, para encender el LED LD2, con la HAL de nuestro **STM32F446RE** podríamos escribirlo como:

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
```

No tenemos que saber nada de los registros, ni definir máscaras; casi todo se hace con llamadas a funciones que nos abstraen del HW. Se estará preguntando entonces:

- **¿Es una HAL un sistema operativo?** No, no lo es. No tiene *kernel*, no gestiona procesos, *etc.* Una HAL es básicamente un conjunto de funciones, numerosas `macros`, y `#define`. En un código que usa HAL, escribiremos con muchas mayúsculas —porque las macros y `#define`, por convención, se suelen escribir con mayúscula—.

- **¿Vamos a programar con HAL?** Sí, y no. Vamos a programar en *bare metal* apoyándonos en CMSIS, que depende de ARM, cuando todos los conceptos de acceso a periféricos se manejen bien, se podría usar la HAL de ST.

- **¿Por qué no la usamos desde el principio si tanto nos ayuda?** Por el contexto en el que estamos: grado universitario, asignatura troncal de conceptos básicos. No sabemos si usted continuará su especialidad por la rama de electrónica, no sabemos si, además, luego se interesará por el área de sistemas, pero sí sabemos que debe conocer el bajo nivel de cómo funcionan los sistemas

[1] También podrá leer que CMSIS puede considerarse una HAL.

que nos rodean, y eso hace que *tengamos que bajar al barro*. Todo este conocimiento alguien tiene que saber implementarlo y, nos toca a los buenos de los ingenieros conocer cómo se hace (a algún afortunado informático, también 🙂).

- Está bien, podemos usar la HAL más adelante, que parece más sencilla, pero **¿me vas a explicar la HAL?** No. La HAL está bien, muy bien cuando queremos enfocarnos en la aplicación y olvidarnos de los detalles HW, pero cada fabricante tiene su HAL. Es circunstancial que estemos usando una plataforma de ST; podríamos estar usando una plataforma de la empresa `Nordic`, o de `Microchip (Atmel)`, o de `NXP`, o de `Renesas`, o de `Analog Instrumentes`, o... Y todas estas empresas tienen su HAL. Todas distintas. Pero todas estas fabrican dispositivos con arquitectura ARM-Cortex de 32 bits, como la nuestra, y sobre todos ellos podríamos transponer nuestro código *bare metal* con el estándar CMSIS, con cambios muy localizados. ¿Quién sabe si una crisis de semiconductores nos hará cambiar de plataforma en un futuro cercano? ¿Y si usted quiere reusar su código en otra? No atarnos a la HAL nos da libertad.

  Le proporcionaremos enlaces y referencias para que encuentre explicaciones y códigos que usen la HAL de ST; la red está plagado de ellos.

- **Entonces, ¿HAL y CMSIS son incompatibles?** No, en absoluto, de hecho la HAL se comunica con CMSIS. Lo entenderás todo mejor en la siguiente sección donde se explica la metodología de programación que vamos a seguir en el libro.

9.2. Metodología de programación usada en este libro

La Figura 9.1 resume la estructura de la metodología de programación que vamos a implementar en el libro. En resumen, podemos decir que vamos a desarrollar un proyecto donde **nuestro código —el `Código de usuario` de la imagen— está dividido en 2: (i) la parte `COMMON`, y (ii) la parte `PORT`**. Esto nos da independencia y modularidad.

IMPORTANTE

La parte `COMMON` (de *común*) contiene toda la lógica del problema **que es independiente de la plataforma** sobre la que se vaya a ejecutar, *i.e.*, esta parte **no hace acceso a registros, temporizadores, interrupciones, *etc.* que dependan del HW**. Por otro lado, **la parte `PORT` (de *portable*) contiene todo el código que sí depende del HW** sobre el que se vaya a ejecutar.

Así pues, si queremos encender el LED `LD2`, en algún fichero en la carpeta `common`, podríamos escribir una llamada a una función así:

```
1 ||     port_ld2_on();
```

Donde su implementación estará en algún fichero de la carpeta `port`:

```
1 || void port_ld2_on()
2 || {
3 ||     GPIOA->ODR |= 0x02;
4 || }
```

De este modo, si tenemos que cambiar de dispositivo, solo tendríamos que adaptar las líneas que hagan referencia a registros de la carpeta `port`, quedando todo el resto del código sin cambios. Esto nos permite incluso **emular el comportamiento en C nativo, en un ordenador, para poder desarrollar incluso sin tener la placa** físicamente. De este modo tendríamos en `PORT` una carpeta para `native` donde la función anterior quedaría:

```
1 || void port_ld2_on()
2 || {
3 ||     printf("LED LD2 ON\n"); /* No tenemos LED fisico, lo escribimos por pantalla*/
4 || }
```

En la Figura 9.1 se muestran 4 niveles en los que podemos intervenir, y a la izquierda de la misma los agentes implicados. La idea es que, aunque cambiemos de dispositivo, nuestro código cambie solo lo más mínimo y estos cambios estén centralizados en los ficheros de `PORT`. Describimos los 4 niveles empezando de abajo a arriba:

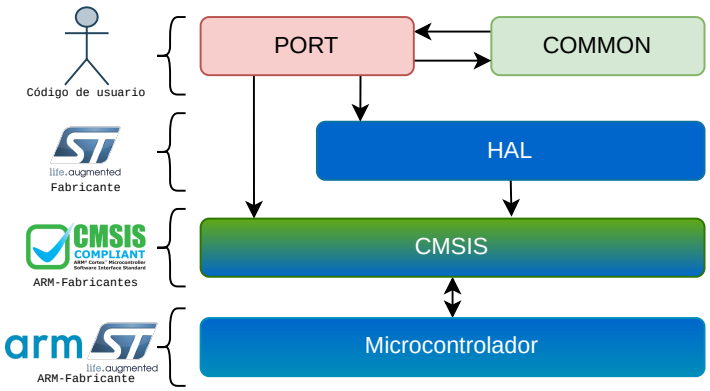
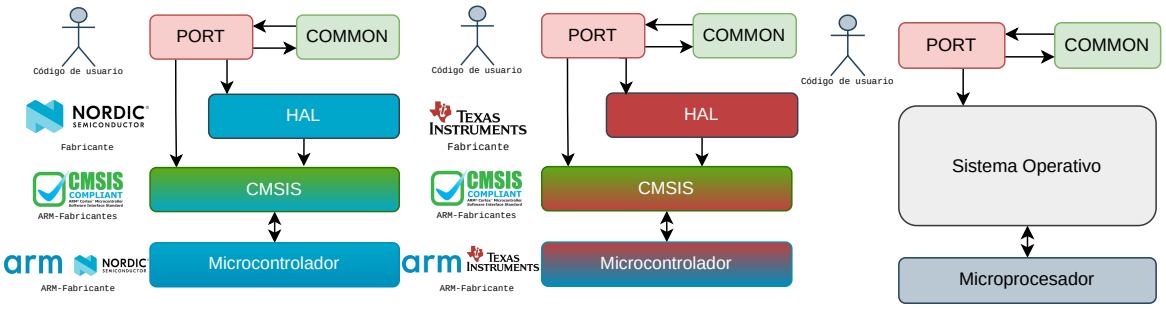


Figura 9.1: Esquema de desarrollo de proyecto a implementar, para ST.

1. **El microcontrolador**: es el nivel HW de nuestro sistema. Podemos actuar eligiendo un fabricante u otro, o incluso una arquitectura distinta. ARM nos proporciona la arquitectura del procesador que es licenciada a los fabricantes, como ST, que añaden periféricos. Podríamos cambiar el fabricante y mantener la arquitectura. Por ejemplo, si queremos desarrollar sobre una placa `Nordic-nrF52840`, trabajaríamos con un microcontrolador de la empresa Nordic



(a) Nordic. *(b)* Texas Instruments. *(c)* Ordenador (nativo).

Figura 9.2: Adaptaciones del esquema de desarrollo del proyecto a otras plataformas.

que tiene una arquitectura ARM, como se muestra en la Figura 9.2a, o trabajar con `Texas Instruments` (Figura 9.2b), ¡o sobre un ordenador! (Figura 9.2c).

2. **CMSIS**: o la interfaz de comunicación con el HW. ARM establece los estándares de esta interfaz de comunicación, y los fabricantes se adaptan a ellos. De esta forma se evitó que cada fabricante desarrollase su interfaz sobre la arquitectura. Bien es cierto que sobre el estándar CMSIS cada fabricante tiene que hacer sus propias adaptaciones relacionadas con sus periféricos. La Figura 9.3[2], de CMSIS, nos muestra una estructura de programación similar a la propuesta en la Figura 9.1. `CMSIS-Core` es con la capa que nos vamos a comunicar, pues nos da acceso a periféricos estándar de comunicación (*Communication peripherals*), al procesador, así como al arranque del sistema (ficheros `startup_nombre-dispositivo.s`), y un acceso consistente a los registros de periféricos y servicios de interrupciones (`SysTick`, `NVIC`...).

   Note que CMSIS también da soporte a sistemas operativos de tiempo real (CMSIS-RTOS en al imagen). Y las versiones más recientes dan soporte para aquellas arquitecturas que implementan HW para ejecutar modelos de redes neuronales y *machine learning* (CMSIS-NN en la imagen). CMSIS incluso propone el estándar de definición de ficheros System View Description (SVD), que relacionan nombres de registros de los microcontroladores y zonas de memoria para que puedan ser interpretados por el depurador (CMSIS-SVD). Para más detalle, se aconseja leer la documentación oficial para Cortex-M en: `https://arm-software.github.io/CMSIS_5/Core/html/index.html`.

3. **La HAL**: ya hemos hablado antes de la capa de abstracción HW. Aquí no interviene ARM para nada. Es totalmente dependiente del fabricante y, como se muestra en las Figuras 9.1 y 9.3, esta HAL se comunica con CMSIS. Los

[2]Modos de bajo consumo del microcontrolador: `https://arm-software.github.io/CMSIS_5/General/html/index.html`. Accedido: 25-01-2025.

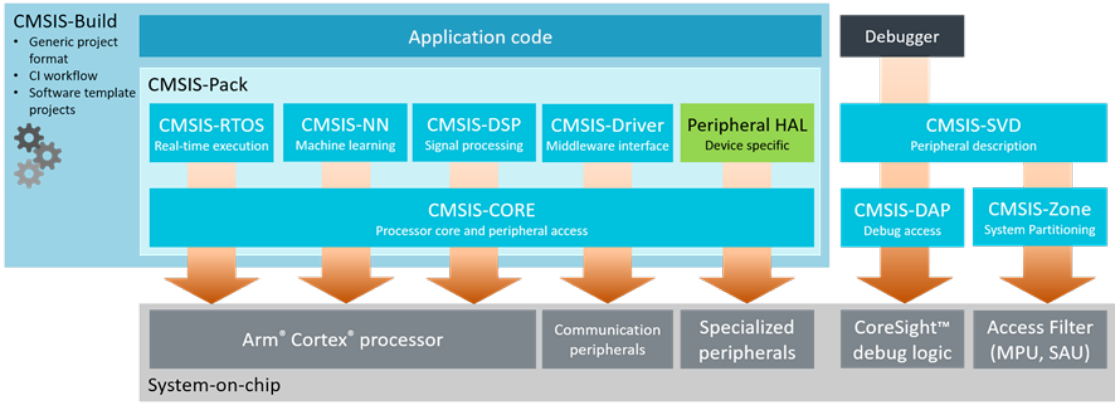


Figura 9.3: Estructura de CMSIS en su versión 5.9.0.

nombres de las funciones varían entre fabricantes, aunque todas siguen una lógica a la hora de nombrarlas. A continuación ponemos cómo se escribe un `'1'` para encender un LED con la HAL de ST, Nordic y Renesas (obviamos toda la parte de configuración de la GPIO):

```
1 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET); /* HAL de ST */
2 nrf_gpio_pin_set(LED_PIN); /* HAL de Nordic */
3 g_ioport.p_api->pinWrite(LED_PIN, IOPORT_LEVEL_HIGH); /* HAL de Renesas */
```

Como ve, el uso de HAL hace el código menos flexible, aunque en proyectos grandes o en empresas que trabajen siempre con los mismos dispositivos, puede hacerse necesario. Pero **¡ojo!, si trabajamos en una empresa y tenemos que certificar y registrar nuestro FW, no podremos hacerlo usando la HAL. Los fabricantes lo proporcionan *as it*, y no se hacen responsables de fallos.** Por eso, no se podrá usar en industria aeroespacial, automovilística, sanitaria... y quizás solo quede relegado al prototipado. Tiene una excelente documentación, aunque muy extensa en el manual *"UM1725. Description of STM32F4 HAL and low-layer drivers"* [11]. El uso de `STM32CubeMX` le proporcionará, mediante un interfaz gráfico, crear código de configuración usando la HAL.

4. **Código de usuario**: El código que nosotros desarrollemos va a ser dividido en dos carpetas: `common`, y `port`. Los códigos de la carpeta `common` permanecen inmutables aunque cambiemos de dispositivo, y tendremos tantas carpetas `port/nombre-dispositivo` como plataformas sobre las que vayamos a desarrollar. Los códigos de las carpetas `port`, sí se adaptan a cada plataforma.

   Cuando compilemos para ordenador, no tendremos las capas de CMSIS, ni HAL, sino que nuestro código será `C` de más alto nivel. Principalmente lo usaremos para emular el comportamiento ya que típicamente no tendremos acceso al mismo HW externo (sensores, actuadores). Un caso particular sería si compilamos para

una Raspberry Pi, en el que tenemos un sistema operativo Linux y *drivers* y *middleware* para acceso a GPIOs.

A continuación se explica la estructura de directorios propuesta para trabajar con esta metodología.

9.2.1. Estructura del entorno MatrixMCU

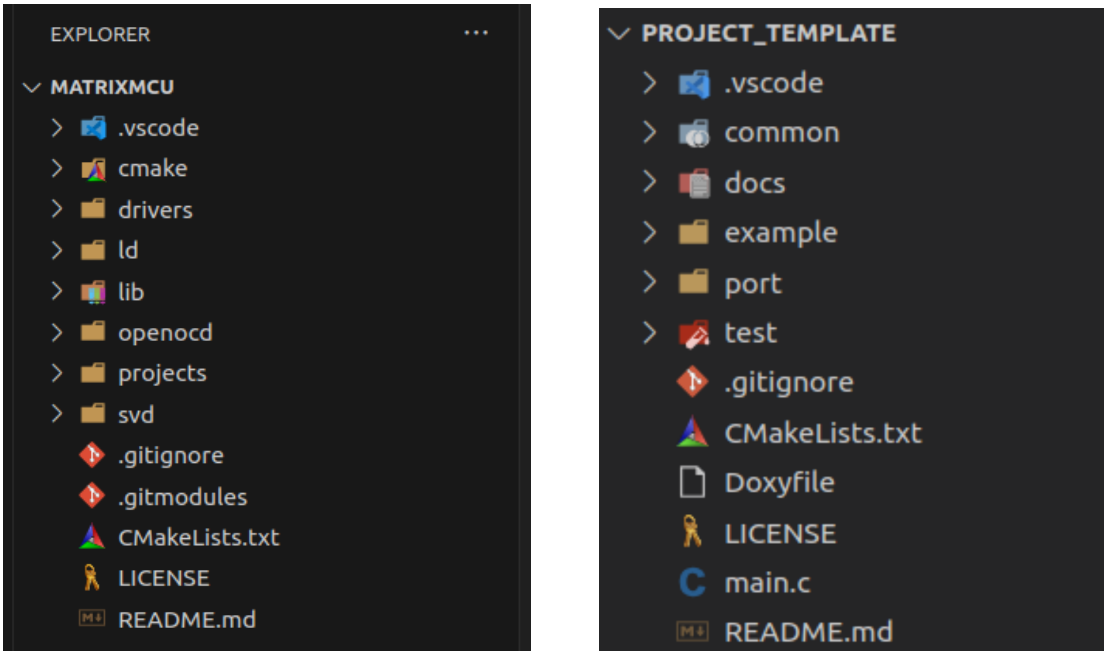
Vamos a trabajar con `VSCode` de Microsoft, y todos los detalles los tiene en *"Guía de instalación de herramientas para compilación cruzada en C"* [12]. Siga la guía para instalarlo en su ordenador personal y poder trabajar en casa; podrá compilar, cargar y depurar el código usando VSCode, y su placa **Nucleo-STM32**.

Se ha elegido VSCode es por ser menos específica del dispositivo, y porque permite la compilación multiplataforma fácilmente. No es un Integrated Development Environment (Entorno de Desarrollo Integrado) (IDE) al uso, sino un editor de texto *vitaminado*, al que le instalamos determinadas extensiones para que nos facilite el desarrollo. A continuación se van presentar brevemente otras herramientas de desarrollo con las que podríamos trabajar para nuestra plataforma. Estas sí son muy específicas y mucho más potentes, sobre todo para la depuración.

Asumimos una estructura de directorios como la propuesta en el capítulo *"Guía de instalación de herramientas para compilación cruzada en C"* de la *"Guía de instalación de VSCode y herramientas de desarrollo en C"* [12], mostradas en las Figuras 9.4a y 9.4b y cuyas **rutas no contienen espacios, caracteres especiales, tildes, ni ñ**. Vamos a mantener dicha estructura de directorios para el entorno `MatrixMCU` y para los proyectos como el de la plantilla `project_template` de la Figura 9.4b. Asegúrese de que tiene instalado la *toolkit* de `MatrixMCU` y que puede compilar y cargar el código de la plantilla `project_template` en su placa **Nucleo-STM32F446RE** como se hace en el mencionado capítulo.

El directorio de `MatrixMCU` contiene todo lo necesario para poder desarrollar proyectos con la estructura que se comenta. A destacar de lo que encontramos en la Figura 9.4a:

- `cmake`: contiene los ficheros de configuración para compilar con `CMake`. `CMake` es una herramienta de compilación multiplataforma que nos permite generar ficheros de configuración para distintos compiladores independiente del sistema operativo en el que se trabaje (Linux, Windows, o iOS). No vamos a entrar en detalle, pero sí que vamos a usarla para generar los ficheros de configuración de compilación. En la Figura 9.4b vemos que hay un fichero `CMakeLists.txt` en cada directorio. Este fichero es el que usaremos para generar los ficheros de configuración de compilación.



(a) Directorios de MatrixMCU. *(b)* Proyecto multiplataforma.

Figura 9.4: Toolkit MatrixMCU y proyecto para compilación multiplataforma.

- `drivers`: contiene las librerías de acceso a periféricos de ST con su HAL y los ficheros de `CMSIS` que es la interfaz de comunicación con el HW. También se incluirían los códigos para trabajar con otras plataformas que soporte el entorno.

  Todo lo que en este libro aparece y los ejemplos proporcionados se apoyan en la API CMSIS. Todos los ficheros necesarios están en la carpeta `CMSIS` de `drivers` para cada plataforma.

  Se puede usar la HAL de ST, claro, pero esto se ha de especificar para cada proyecto. Más abajo se explica cómo.

- `ld`: contiene los ficheros de *linker* para cada microcontrolador. El *linker* es la herramienta que se encarga de enlazar los ficheros objeto (`.o`) y generar el ejecutable. El *linker* necesita saber dónde están las zonas de memoria del microcontrolador, y eso es lo que le proporcionamos en estos ficheros. Habrá un fichero por cada plataforma soportada por el entorno.

- `lib`: contiene las librerías más genéricas que usarán los proyectos. En nuestro caso, contienen las librerías `fsm` y `unity`. La primera es una librería desarrollada por el equipo docente para implementar máquinas de estados finitos. La segunda es una librería de test unitarios que usaremos para hacer pruebas de nuestro código.

- `openocd`: están los ficheros de configuración para el depurador `OpenOCD`.

- `projects`: contiene los proyectos que vayamos a desarrollar. En la Figura 9.4b vemos un proyecto de ejemplo que puede usar como plantilla para sus proyectos. Más abajo se explica en detalle.

- `svd`: contiene los ficheros SVD de cada microcontrolador. Son ficheros Extensible Markup Language (XML) que describen la arquitectura del procesador y los registros de periféricos.

9.2.2. Estructura de proyecto

Para cada proyecto tendremos una estructura como la de la de la Figura 9.4b que se detalla en al Figura 9.5.

Dónde vamos a tocar código

Preste atención a las descripciones de los elementos que se muestran en la Figura 9.5 para entender la estructura de proyecto. En la carpeta `common` se encuentra todo el código que no depende del HW, y en la carpeta `port` se encuentra todo el código que sí depende del HW.

IMPORTANTE

En la carpeta `port` habrá una subcarpeta por cada plataforma que se quiera soportar y un directorio `include` común a todas las plataformas que será *contrato* con el usuario, donde estarán aquellas funciones que pueden ser llamadas desde `common` y no tienen dependencias de la plataforma en los argumentos recibidos ni tipos devueltos.

Todas las implementaciones de las funciones que se encuentren en `include` estarán en las subcarpetas `src` de `port` que correspondan a cada plataforma. Además, cada plataforma tendrá su propio directorio `include` con las funciones específicas de cada una.

En la Figura 9.5 se muestra un ejemplo para la placa **Nucleo-STM32F446RE**y un ordenador. El fichero `main.c` es el fichero principal y punto de entrada del proyecto.

A priori todos los proyectos los vamos a dejar en la carpeta `projects` de MatrixMCU, pero pudieran estar en otro directorio, aunque no se recomienda. Si así fuese, habría que indicarlo en el fichero `CMakeLists.txt` más externo, el que está en la raíz del proyecto (el que aparece más abajo en la Figura 9.4b). Por defecto no debería tocar nada de este fichero, pero si lo necesitase, lo que se puede configurar y que se muestra en la Figura 9.6 es:

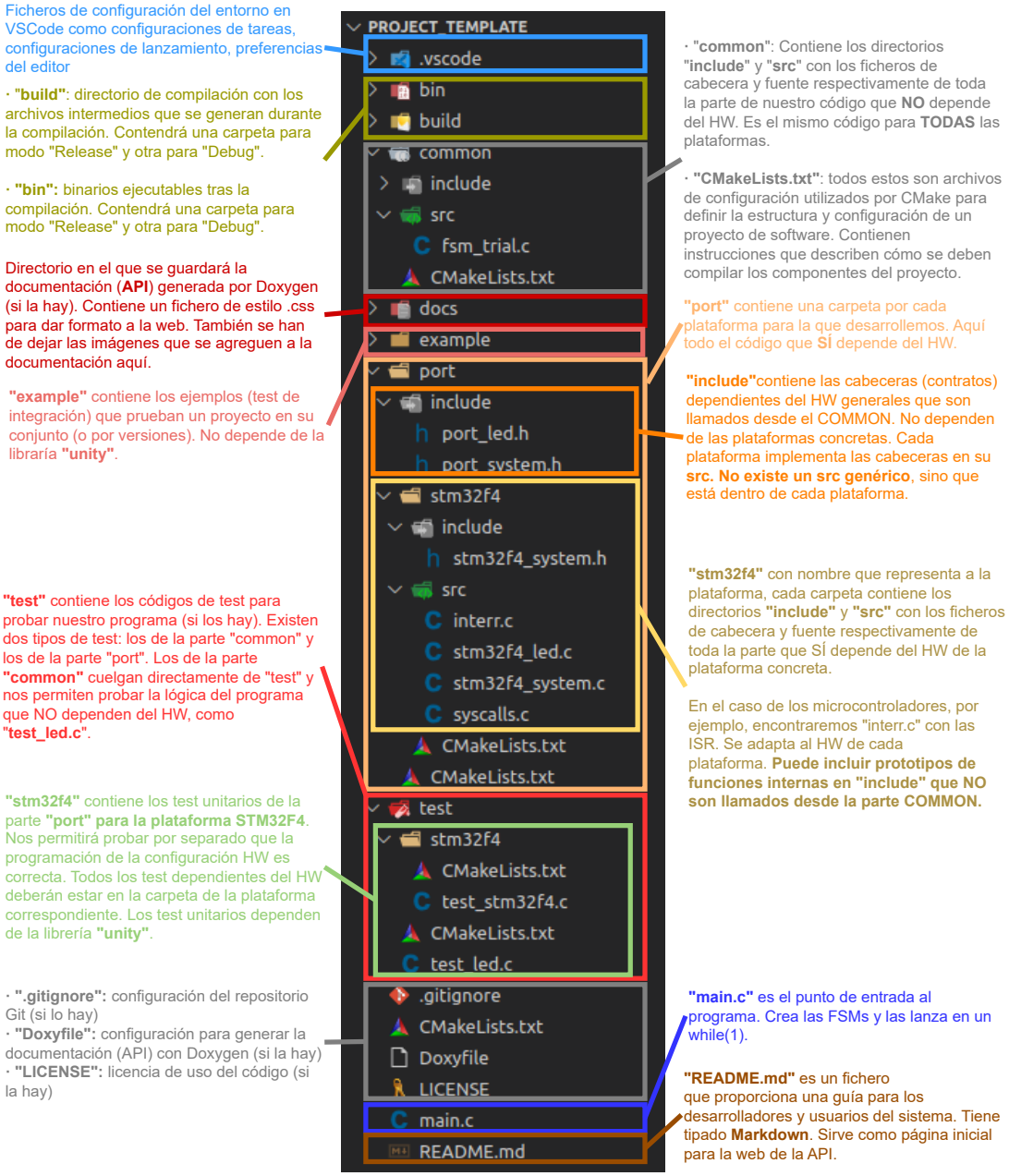


Figura 9.5: Estructura de proyecto con división de directorios common y port.

- **Nombre del proyecto**: se puede cambiar el nombre del proyecto en la variable `PROJECT_NAME`. Por defecto, el nombre del proyecto es el nombre de la carpeta donde se encuentra el fichero `CMakeLists.txt`.

- **Verboso**: se puede cambiar la variable `VERBOSE` a `false` para que se oculten todos los mensajes de compilación. Por defecto, la variable está a `true`.

- **Directorio de MatrixMCU**: se puede cambiar el directorio de MatrixMCU

```
SET(PROJECT_NAME project) # set it to your project name

IF(NOT DEFINED VERBOSE)
    SET(VERBOSE true) # set it to true to print additional information
    MESSAGE(STATUS "No verbosity selected, using default (${VERBOSE}). You can override it by passing -DVERBOSE=<verbose> to cmake")
ENDIF()
IF(NOT DEFINED MATRIXMCU)
    SET(MATRIXMCU  ${CMAKE_CURRENT_SOURCE_DIR}/../..) # set it to your default path to MatrixMCU directory
    MESSAGE(STATUS "No path to MatrixMCU directory selected, using default (${MATRIXMCU}). You can override it by passing -DMATRIXMCU=<MATRIXMCU>
        cmake")
ENDIF()
IF(NOT DEFINED USE_FSM)
    SET(USE_FSM true) # set it to true to use FSM library by default
    MESSAGE(STATUS "No FSM library usage selected, using default (${USE_FSM}). You can override it by passing -DUSE_FSM=<use_fsm> to cmake")
ENDIF()
IF(NOT DEFINED USE_HAL)
    SET(USE_HAL false) # set it to true to use HAL library by default
    MESSAGE(STATUS "No HAL library usage selected, using default (${USE_HAL}). You can override it by passing -DUSE_HAL=<use_hal> to cmake")
ENDIF()
IF(NOT DEFINED PLATFORM)
    SET(PLATFORM "stm32f446re") # PORTABILITY: change this to your platform
    MESSAGE(STATUS "No platform selected, using default (${PLATFORM}). You can override it by passing -DPLATFORM=<platform> to cmake")
ENDIF()
IF(NOT DEFINED CMAKE_BUILD_TYPE)
    SET(CMAKE_BUILD_TYPE Debug) # set it to your default build type
    MESSAGE(STATUS "No build type selected, using default (${CMAKE_BUILD_TYPE}). You can override it by passing -DCMAKE_BUILD_TYPE=<build_type> t
ENDIF()
IF (NOT DEFINED USE_SEMIHOSTING)
    SET(USE_SEMIHOSTING true)
    MESSAGE(STATUS "Semihosting not specified, using default (${USE_SEMIHOSTING}). You can override it by passing -DUSE_SEMIHOSTING=<use_semihost
        cmake")
ENDIF()

###############################################################################
## IF YOU DON'T KNOW WHAT YOU ARE DOING, DO **NOT** EDIT THIS FILE FROM THIS POINT ON ##
###############################################################################
```

Figura 9.6: Fichero `CMakeLists.txt` de la raíz del proyecto.

en la variable `MATRIXMCU_DIR`. Por defecto, el directorio de MatrixMCU es el directorio donde se encuentra el fichero `CMakeLists.txt`.

- **Uso de librería FSM**: se puede cambiar el uso de la librería FSM en la variable `USE_FSM`. Por defecto, el uso de la librería FSM está activado.

- **Directorio de HAL**: aquí se elige si se va a usar, o no, la HAL. Por defecto, no se usa la HAL, para ahorrar memoria y tiempo de compilación. Si se quiere usar, se ha de cambiar la opción `USE_HAL` a `true`.

- **Tipo de compilación**: se puede cambiar el tipo de compilación en la variable `CMAKE_BUILD_TYPE`. Por defecto, el tipo de compilación es `Debug` y en las carpetas `build` y `bin` se crean los ficheros de depuración. Si se quiere compilar en modo `Release`, se ha de cambiar la opción a `Release`.

- **Uso de *semihosting***: el *semihosting* es una técnica que permite al programa del embebido comunicarse con el entorno de desarrollo (el PC) y ejecutar en él funciones de entrada/salida como `printf()`. El resultado saldrá por la ventana de depuración `gdb-server`. Por defecto, el uso de *semihosting* está activado y hace que no se necesiten las implementaciones del fichero `syscalls.c`.

Qué código nos pueden dar (o podemos implementar) para test

La Figura 9.5 también hace referencia a la carpeta de `test` y `example`, tan importante para un desarrollo exitoso. Puede ser que nos proporcionen o que desarrollemos nosotros mismos código para test. Tendremos dos tipos de test: los test unitarios y los ejemplos o test de integración. En el Capítulo 10 se explica cómo usarlos.

Qué documentación nos pueden dar (o podemos implementar)

El fichero `Doxyfile` es el fichero de configuración de `Doxygen` que se usa para generar la documentación del proyecto. Puede ver el vídeo de Youtube "[MatrixMCU] Documentación de código con Doxygen" para más información. El fichero `README.md` es el fichero de descripción del proyecto que se muestra en la página principal del repositorio y que se usa para dar una descripción del proyecto, autores, fecha, cómo usarlo, enlace a un vídeo demostrativo, imágenes...

Cuando se usa el programa Doxygen para generar automáticamente la API del código, se pondrá en la carpeta `doc`. En la carpeta `doc` proporcionada, además, hay un fichero `.css` que se usa para dar formato a la documentación. En esta carpeta también se puede poner cualquier otro fichero que se quiera incluir en la documentación, como imágenes, vídeos, *etc.*

Qué ficheros auxiliares existen

En la Figura 9.5 también se hace referencia a archivos auxiliares como `CMakeLists.txt`. Son ficheros de configuración de `CMake` que se usan para generar los ficheros de reglas de compilación (los `Makefiles`).

9.2.3. Cuándo un código es common o port

Cuando escribimos una función y no sabemos dónde colocar su prototipo e implementación, hemos de preguntarnos si la función depende del HW o no. Si depende del HW, la función ha de ir en la carpeta `port`. Si no depende del HW, la función ha de ir en la carpeta `common`. Para más detalle, se puede seguir el diagrama de flujo de la Figura 9.7.

Si te confundes, las reglas de compilación que se han definido en el fichero `CMakeLists.txt` darán error.

9.2.4. Documentación del código

La documentación del código es muy importante. No solo para que otros puedan entender nuestro código, sino también para que nosotros mismos podamos recordar qué hace cada función. Cuando se trabaja en un proyecto grande, con muchas

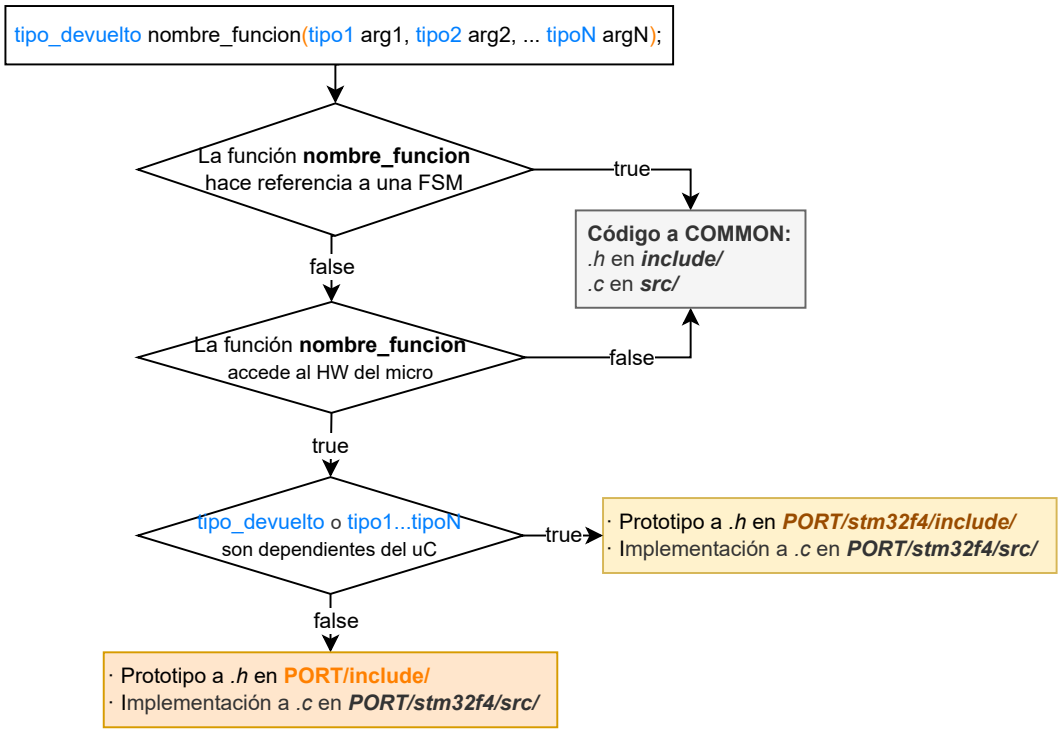


Figura 9.7: Diagrama de flujo para saber si una función va en `common` o en `port`.

personas, o cuando se va a distribuir el código o dárselo a un cliente, la documentación es imprescindible.

Para documentar el código, en MatrixMCU se va a usar `Doxygen`. `Doxygen` es una herramienta que genera documentación a partir de comentarios en el código fuente. Los comentarios se escriben en un formato especial y `Doxygen` se encarga de generar la documentación en formato `HTML`, `PDF`, `LaTeX`, *etc.*

Vaya al Apéndice B para ver cómo se documenta el código con `Doxygen`. Ahí se explica cómo se instala y cómo se usa. También se puede ver el vídeo de Youtube "[MatrixMCU] Documentación de código con Doxygen" para más información.

Capítulo 10

Test unitarios y ejemplos de integración

Román Cárdenas
Josué Pagán

En desarrollo de SW el ***diseño orientado a test*** es una práctica que permite mejorar la calidad del código y reducir el número de errores. El mundo del diseño orientado a test es muy amplio y no se pretende abarcarlo en este capítulo. El diseño orientado a test es una metodología por la que primero se definen los test que se deben superar para que el código funcione correctamente y, después, se desarrolla el código que pasa los test. Tampoco se pretende enseñar a desarrollar test en este capítulo, sino a usarlos. En este capítulo se describen los test unitarios y ejemplos de uso (también llamados *test de integración*) que utilizaremos en los proyectos desarrollados con la *toolkit* MatrixMCU.

¡Importante!, **los test no comprueban la sintaxis, ni la semántica del código**. Los test comprueban que el código funciona correctamente. Es decir, que se comporta como se espera que se comporte. Además, **los test solo comprueban lo que se les pida, y lo que no se compruebe no se garantiza que funcione correctamente**. Por ello, es importante que los test sean lo más exhaustivos posibles. El diseño orientado a test tiene una metodología para conseguirlo, pero se sale del alcance de este libro. En los proyectos trabajaremos con test sencillos que nos permitan comprobar los aspectos más relevantes de los códigos que desarrollemos.

Como se ha mencionado en el Capítulo 9, los proyectos desarrollados con la *toolkit* MatrixMCU se presentan con una estructura modular que separa el código entre lo que es *común* a cualquier plataforma (`COMMON`), y lo que es *portable* y dependiente del HW de la plataforma (`PORT`). Cuando desarrollemos proyectos conviene comprobar de manera incremental que el código funciona correctamente. Para ello, vamos a apoyarnos en los test unitarios y test de integración.

Tendremos pues un conjunto de **test unitarios que comprueban**

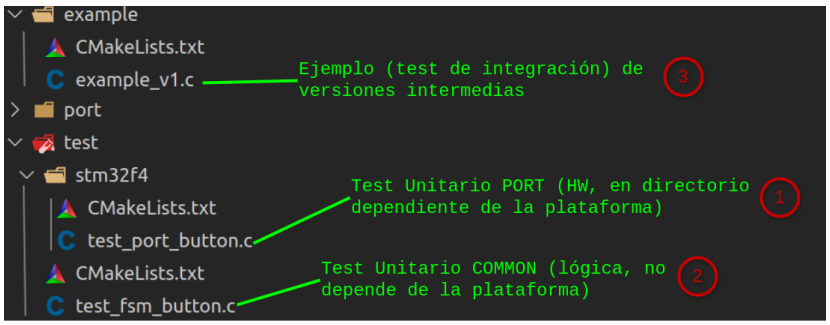


Figura 10.1: Estructura de directorios de test unitarios y ejemplos en MatrixMCU.

especificaciones de las partes `COMMON` y `PORT` de forma independiente. Y, por otro lado, tendremos **test de integración que comprueban especificaciones de la aplicación final en su conjunto**, como muestra la Figura 10.1.

En la *toolkit* MatrixMCU, los test se desarrollan con la librería **Unity**. Unity es una librería de test unitarios para C que permite desarrollar test de forma sencilla, desde microcontroladores de 8 bits hasta sistemas basados en microprocesador de 64 bits.

Lo interesante de los test es que se ejecutan sobre la plataforma y obtendremos mensajes por la terminal que nos indicarán si un test ha pasado (`PASS`), o no (`FAIL`). Los test pueden ejecutar comprobaciones de diverso tipo; todas ellas se basan en `ASSERTS`. Un `ASSERT` es una comprobación que se realiza sobre una expresión que queremos que sea cierta. Si la expresión se cumple, el test pasa. Se puede comprobar si una variable es igual a un valor deseado, si está dentro de un rango de valores, se puede comprobar que sea distinta de un valor, *etc.* Se pueden comparar arrays, cadenas de caracteres, punteros...

10.1. Test unitarios

En desarrollo de software, un test unitario es una forma de comprobar el correcto funcionamiento de una unidad de código. Una vez haya pasado los test unitarios, podemos pasar a los ejemplos de prueba o test de integración.

En la Figura 10.2 se muestra la estructura del código de un test unitario. En la parte superior se incluyen las librerías necesarias para el test. Siempre incluiremos `#include <unity.h>`, que es la librería de test unitarios. Además, incluiremos las librerías que necesitemos para el test, en cada caso, dependiendo de lo que queramos testear.

Los test unitarios siempre tienen la misma estructura:

- En primer lugar, tenemos una función que se llama `setUp()` y se encarga de inicializar las variables que se van a utilizar en el test.
- En segundo lugar, se define una función que se ejecuta después de cada test.

```c
/* HW independent libraries */
#include <stdlib.h>
#include <unity.h>
#include "port_button.h"
#include "port_system.h"
/* HW dependent libraries */
#include "stm32f4_system.h"
#include "stm32f4_button.h"
#include "stm32f4xx.h"

void setUp(void)
{
    // Init system here
}

void tearDown(void)
{
    // Clean up stuff here
}

void test_pins(void)…

void test_regs(void)…

int main(void)
{
    port_system_init();
    UNITY_BEGIN();
    RUN_TEST(test_pins);
    RUN_TEST(test_regs);
    exit(UNITY_END());
}
```

Figura 10.2: Estructura de código de un test unitario.

Esta función se llama `tearDown()` y se encarga de liberar los recursos que se han utilizado en el test.

- En tercer lugar, se definen las funciones que se encargan de realizar los test. Estas funciones se llaman `test_XXX()`, donde `XXX` es el nombre del test. El nombre es libre, pero es recomendable que sea descriptivo, y puede haber tantos test como se quiera. Se recomienda que cada test compruebe una funcionalidad concreta.

- Finalmente, se define una función `main()` que se encarga de ejecutar los test. Cada test se ejecuta en una macro `RUN_TEST()`, que recibe como parámetro el nombre de la función que se encarga de realizar el test. Independientemente de que el test haya pasado, o no, se muestra un mensaje por la terminal indicando el resultado y se pasa al siguiente test. Si el resultado es `FAIL` se muestra un mensaje indicando el motivo del fallo para poder corregirlo.

- El test se detendrá en la última línea de código `exit(UNITY_END())` y la depuración se quedará ahí en bucle hasta que se pulse Stop.

10.2. Ejemplo de uso o test de integración

Los ejemplos de uso o test de integración comprueban que el código funciona correctamente en su conjunto. Es decir, que las partes `COMMON` y `PORT` funcionan

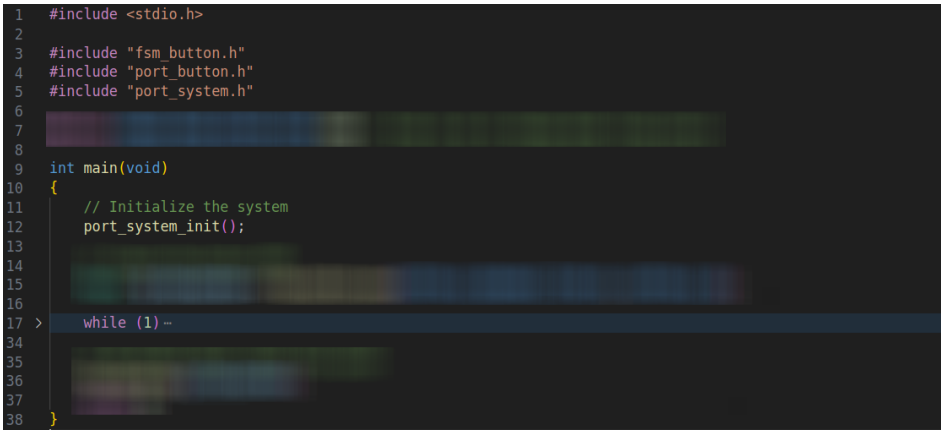


Figura 10.3: Estructura de código de un test de integración.

correctamente cuando se integran. Como siempre, comprueba que funciona aquéllo que se le pida, por lo que es importante que los ejemplos de integración sean lo más exhaustivos posibles. Puede ser que, si no cubrimos todos los casos, el código funcione correctamente en los casos que hemos comprobado, pero no en otros.

Los test de integración, no hacen uso de la librería `Unity`, sino que se ejecutan sobre la plataforma. Es, en resumen, un código más o menos grande que parte, como no, de un `main()` y que hace uso de las funciones que se han desarrollado para el proyecto. A lo largo de la ejecución se ejecutan varios bloques condicionales `if` que imprimen por la terminal un mensaje indicando el resultado de la ejecución. Es tarea del desarrollador comprobar que el resultado es el esperado. En la Figura 10.3 se muestra la estructura de un test de integración para comprobar el funcionamiento de un código que lee la duración de un pulso de botón.

10.3. Ejecución de los test

Como se ha mencionado anteriormente, los test se ejecutan sobre la plataforma. Esto implica que, si estamos testeando para, por ejemplo, la placa **Nucleo-STM32**, para poder recibir los mensajes de los test, debemos tener conectada la placa al ordenador y la terminal de depuración `gdb-server` abierta.

Para cada versión de un proyecto se recomienda ejecutar los test unitarios y ejemplos de integración para comprobar que todo funciona correctamente y poder pasar a la siguiente versión sin arrastrar errores. Es por ello que tendremos un orden a la hora de ejecutar los test, como se muestra en la Figura 10.1.

Primero conectamos la placa al ordenador. Para ejecutar los test pulsaremos sobre el **icono de depuración** y **daremos Debug** sobre la plataforma que queramos depurar. Se nos abrirá un desplegable con todos los códigos del proyecto

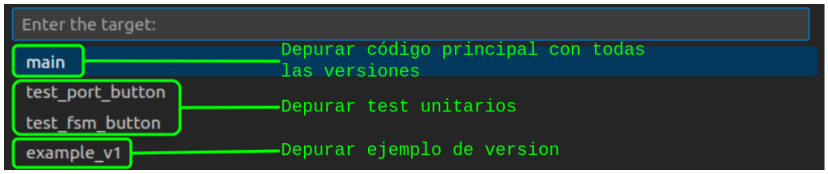


Figura 10.4: Lista de los posibles test unitarios y de integración.

que podemos depurar, como se muestra en la Figura 10.4[1]. Por defecto se ejecuta el código principal del proyecto, pero podemos elegir el código cualquier otro. En este caso, **elegiremos el código de test que queremos**. Al ejecutarlo, se parará en la depuración en la primera línea del código de test. Si pulsamos Continue, se ejecutará el test y se mostrará el resultado por la terminal `gdb-server`. También, si lo desea, puede poner puntos de parada y ejecutar el código paso a paso para ver cómo van pasando las pruebas del test.

1. **Lo primero que ejecutaremos son los test unitarios de la parte `PORT`.**

   La parte `COMMON` hace uso de la parte `PORT` para acceder a los recursos de la plataforma. Por ello, los test unitarios de la parte `PORT` deberán ejecutarse primero, pues si no funciona la parte dependiente del HW, no podremos comprobar el resto de la lógica.

   En este tipo de test podemos probar, por ejemplo, que los registros de un periférico se configuran correctamente sin machacar otros registros, o que se eligen los pines correctos para un periférico, *etc.* En la Figura 10.5 se muestra la ejecución de los test unitarios de la parte `PORT` para la placa **Nucleo-STM32**. En este caso, se comprueba que el test llamado `test_regs()` pasa correctamente, mientras que el test llamado `test_pins()` falla. En este caso, el test falla porque se ha configurado un pin que no es el correcto y nos muestra un mensaje indicando el pin que se ha configurado y el pin que se esperaba configurar.

   Observe que este test está dentro del directorio dependiente de la plataforma (`test/stm32f4`), pues **será específico de cada plataforma.**

2. **Seguiremos con los test unitarios de la parte `COMMON`.**

   Una vez que hemos comprobado que la parte `PORT` funciona correctamente, podemos pasar a comprobar la parte `COMMON`. En este tipo de test podemos comprobar, por ejemplo, que se una máquina de estados está bien definida, que

[1]Los códigos de test se encuentran en la carpeta `test` del proyecto y para que aparezcan en la lista, debemos escribirlos en el fichero `launch.json` que se encuentra en la carpeta `.vscode`.

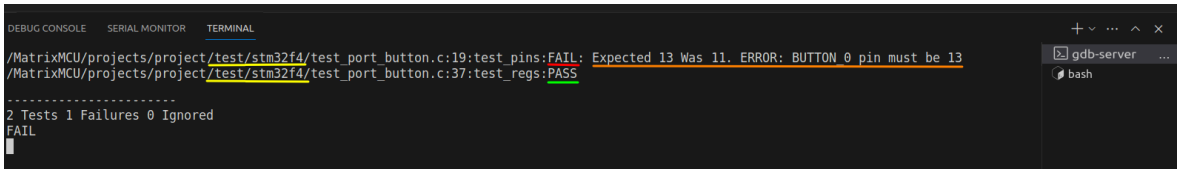


Figura 10.5: Ejemplo de ejecución de los test unitarios de la parte `PORT`.

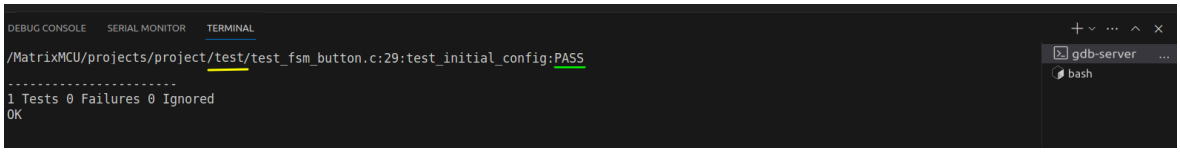


Figura 10.6: Ejemplo de ejecución de los test unitarios de la parte `COMMON`.

las transiciones se ejecutan correctamente, o que una función devuelve el valor esperado. En la Figura 10.6 se muestra la ejecución de los test unitarios de la parte `COMMON`. En este caso, se comprueba que el test llamado `test_fsm_button()` es el único y pasa correctamente.

Observe que este test no está dentro del directorio dependiente de la plataforma, sino que está en la raíz del directorio `test/`, pues **será genérico para cualquier plataforma.**

3. **El ejemplo de prueba o test de integración será el último test que ejecutaremos** una vez comprobado que la parte `PORT` y la parte `COMMON` funcionan correctamente.

   Como se ha comentado anteriormente, el test de integración es un código de ejemplo *ad hoc* que hace uso de las funciones que se han desarrollado para el proyecto y que queremos comprobar. No es el proyecto en sí mismo, sino una adaptación de este con condicionantes para comprobar determinadas situaciones.

   En el caso del ejemplo de la Figura 10.3, se comprueba que las distintas funciones que se han desarrollado para el manejo del botón funcionan correctamente. En este caso, se comprueba que cuando se pulsa y suelta el botón se mide la duración del pulso. La duración se imprime por pantalla. En este caso el desarrollador debe comprobar que los valores obtenidos son los esperados. Se podrían probar más casos y hacer un código donde se den instrucciones al usuario usando `printf()` y se compruebe el comportamiento del sistema.

En el mundo de los test hay muchas más cosas que se pueden hacer, pero con lo que se ha explicado en este capítulo es suficiente para poder desarrollar proyectos con la *toolkit* MatrixMCU. **En los proyectos se proporcionarán los test unitarios y ejemplos de uso necesarios para comprobar el correcto funcionamiento del código.**



Figura 10.7: Ejecución de un ejemplo de uso o test de integración.

Bibliografía

[1] R. Cárdenas Rodríguez, J. Pagán Ortiz, and I. Martín Fernández, *Tutoriales sobre los fundamentos teóricos de sistemas basados en microcontrolador STM32*. Universidad Politécnica de Madrid, 2025.

[2] STMicroelectronics, "Um1724 user manual. stm32 nucleo-64 boards." STMicroelectronics, Tech. Rep., 2020. [Online]. Available: https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf

[3] J. Yiu, "White paper: Arm cortex-m for beginners an overview of the arm cortex-m processor family and comparison," ARM Limited, Tech. Rep., March 2017. [Online]. Available: https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2057-00-00-00-01-28-35/Cortex_2D00_M-for-Beginners-_2D00_-2017_5F00_EN_5F00_v2.pdf

[4] D. Ibrahim, *Nucleo Boards Programming with the STM32CubeIDE: Hands-on in More Than 50 Projects*, ser. Elektor books. Elektor International Media B.V.., 2020.

[5] STMicroelectronics, "Stm32f446xc/e," STMicroelectronics, Tech. Rep., 2021. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f446re.pdf

[6] ——, "Rm0390 reference manual. stm32f446xx advanced arm-based 32-bit mcus." STMicroelectronics, Tech. Rep., 2021. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[7] C. Santoro, "The startup sequence of stm32," 2022, accessed: 16-01-2023. [Online]. Available: https://www.dmi.unict.it/santoro/teaching/lsm/slides/StartupSTM32.pdf

[8] P. Harikumar, "Low-voltage analog-to-digital converters and mixed-signal interfaces," Ph.D. dissertation, Linköping University Electronic Press, 2016.

[9] E. Kalmykova and C. A. Mgina, "Biorecognition ability of polysaccharides as piezo quartz biosensors," *Tanzania Journal of Science*, vol. 38, no. 3, pp. 209–220, 2012.

[10] W. Jung, "Applying ic sample-hold amplifiers," *Analog Devices Inc*, 1995.

[11] STMicroelectronics, "Um1725 user manual. description of stm32f4 hal and low-layer drivers." STMicroelectronics, Tech. Rep., 2021. [Online]. Available: https://www.st.com/resource/en/user_manual/um1725-description-of-stm32f4-hal-and-lowlayer-drivers-stmicroelectronics.pdf

[12] J. Pagán Ortiz, D. Capellán Martín, R. Cárdenas Rodríguez, A. de Gracia Herranz, and S. Esteban Romero, *Guía de instalación de herramientas para compilación multiplataforma en C*. Universidad Politécnica de Madrid, 2025.

Apéndice A

Otras herramientas de desarrollo

Josué Pagán

Existen varias IDE que soportan la programación de nuestras placas **Nucleo-STM32**. En esta sección vamos a nombrar algunas de ellas. Todos los IDE dan soporte a compiladores de `C` y `C++`. Algunos de los más relevantes son:

- **Embedded Workbench for ARM (EWARM)**: desarrollado por IAR Systems, está optimizado para dispositivos Cortex-M. La parte más interesante es que **permite la depuración con simulación**. Esto es muy potente, porque nos da la posibilidad de trabajar sin el dispositivo. En nuestro caso, para el libro, hemos implementado la forma de *emulación* (que no simulación) en PC, para emular el comportamiento a nivel de aplicación, pero no permite simular el estado de los registros. Es una plataforma de desarrollo profesional y su versión gratuita solo está disponible durante 30 días.

- **ARM Mbed**: es una herramienta de diseño rápido. Diseñada por ARM, permite el desarrollo y depuración *online* (ARM Mbed), sin necesidad de instalar nada —con limitaciones—, aunque también existe ARM Mbed Studio para instalarlo en nuestro ordenador. Proporciona muchos ejemplos, y una API para desarrollo rápido de aplicaciones IoT en `C++`. Soporta muchas placas de desarrollo de distintos fabricantes. Por supuesto, da soporte al sistema operativo en tiempo real (RTOS) MbedOS.

- **Keil**: a veces llamado MDK-Keil, desarrollado por `Keil`, es un SW de desarrollo para dispositivos ARM Cortex-M. Permite la depuración, y soporta numerosos dispositivos de muchos fabricantes. Nos permite ver el código desensamblado y el estado de los registros. Es una herramienta potente, profesional, a la que solo tenemos acceso a su versión *lite*, que está limitada para proyectos pequeños.

- **STM32CubeIDE** y **STM32CubeMx**: STM32CubeIDE es una

herramienta basada en `Eclipse` que permite depuración y facilita la creación de proyectos para dispositivos STM32. Esta herramienta está desarrollada por ST y permite la integración de STM32CubeMX. `STM32CubeMX` es un inicializador de código que con una interfaz gráfica que genera automáticamente el código con la configuración de sistema, relojes y periféricos que hayamos seleccionando con el ratón. Tiene la ventaja de que es muy rápido y nos ayuda en la difícil configuración de registros. Tiene la desventaja de que nos abstrae totalmente de dicha configuración, y genera un código muy verboso y redundante que puede ser difícil de seguir y depurar.

La Figura A.1 muestra el aspecto de esta interfaz gráfica en un ejemplo para la placa **Nucleo-STM32F446RE**. Se muestra el *pinout* del microcontrolador; seleccionando cada pin podemos acceder a su configuración, así como a la de los *timers*, relojes y periféricos en el resto de pestañas. El código que genera se basa en la HAL de ST. **Puede usar CubeMX en la parte avanzada del proyecto para ayudarse a generar códigos de configuración para periféricos que no estén explicados en el libro.** Podrá usarlo en asignaturas del Grado de Ingeniería de Tecnologías y Servicios de Telecomunicación como Sistemas Electrónicos de Control u otras de Máster de la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad Politécnica de Madrid, donde el interés está más en la aplicación que en el manejo del dispositivo.

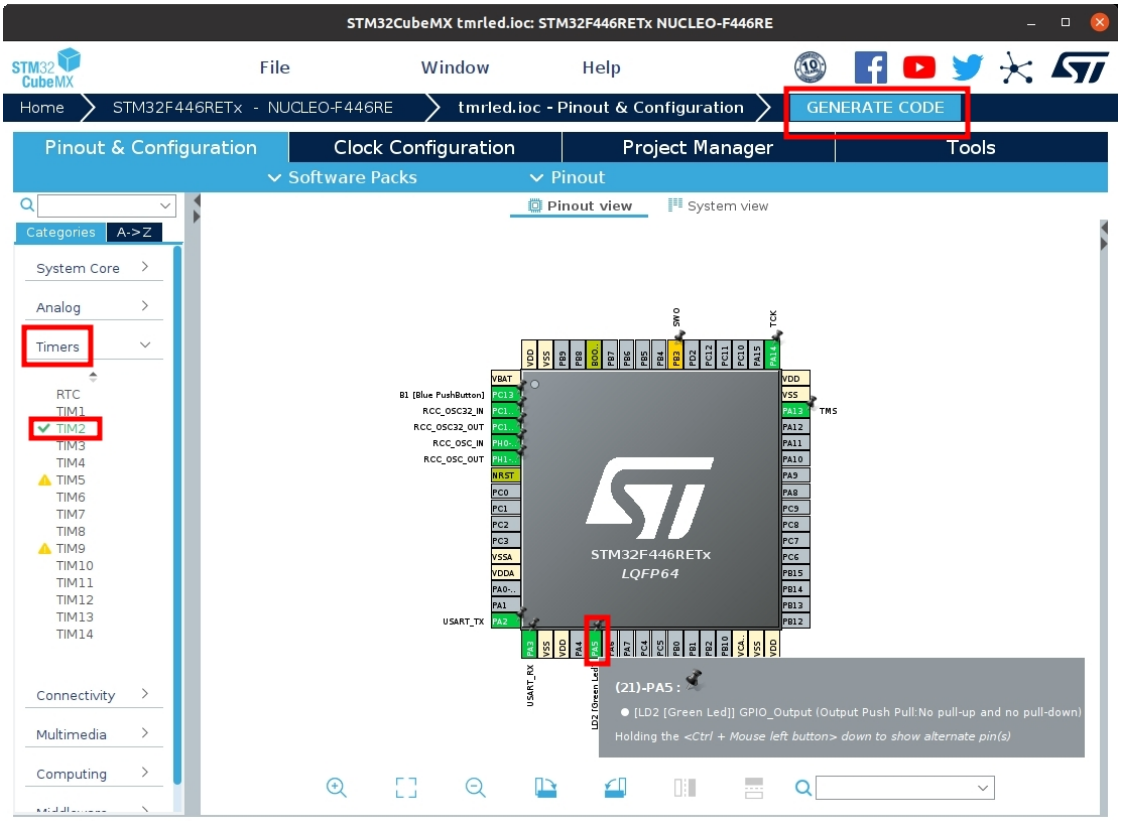


Figura A.1: Aspecto de la interfaz de CubeMX. Destaca la selección de pin para LED `LD2` y configuración de *timer*.

Apéndice B

Generación de documentación de código con Doxygen

Josué Pagán

En esta sección encontrará un resumen de lo que se enseña en el vídeo "[MatrixMCU] Documentación de código con Doxygen", del que se recomienda su visionado.

Del repositorio descargado en `https://github.com/sdg1DieUpm/c_basics` podemos observar que existe, entre otras cosas, un documento llamado `Doxyfile`. Este documento contiene las reglas para generar la API web de nuestro código. Por como está configurado, depende de `README.md` y de `doxygen-awesome.css` y, aunque no son estrictamente necesarios hacen que la vista de la API generada sea más llamativa:

- **README.md**: es el fichero de extensión `Markdown` que encontramos en cualquier repositorio git. Este fichero será la homepage de nuestra API, haciendo que sea más visual. Podremos modificarlo para añadir imágenes, enlaces, títulos, *etc.* como cualquier documento `Markdown`. Tiene una sintaxis sencilla, puede encontrar mucha información *online*; un buen punto de partida puede ser: `https://www.markdownguide.org/basic-syntax/`.

- **doxygen-awesome.css**: `CCS` sirve para dar formato al código HTML generado. HTML tiene elementos como tablas, botones, menús, cuyo aspecto no es muy llamativo, para eso tenemos CCS. Este documento simplemente contiene algunas reglas de formateo haciendo que el HTML generado *se vea* mejor.

B.1. Generando la API en Windows con Doxygen

Como estamos trabajando con WSL, que es Linux, siga los pasos para Linux desde una terminal de WSL.

B.2. Generando la API en Linux con Doxygen

1. Abrimos una terminal y nos colocamos donde tengamos el fichero `Doxyfile` que se recomienda que esté dentro de la carpeta del proyecto, a la altura de `common` y `port`.

2. Escribimos el siguiente comando:

   `doxygen Doxyfile`, y damos *enter* para compilar

3. Inmediatamente se habrá generado el código HTML de la web de nuestra API en `./docs/html`. Podemos abrir el fichero `index.html` con nuestro navegador; esta página muestra primeramente el contenido que tuviésemos en `README.md`.

B.3. Generando la API en MacOS con Doxygen

1. Abrimos una terminal y nos colocamos donde tengamos el fichero `Doxyfile` que se recomienda que esté dentro de la carpeta del proyecto, a la altura de `common` y `port`.

2. Escribimos el siguiente comando: `doxygen Doxyfile`, y damos *enter* para compilar

3. Inmediatamente se habrá generado el código HTML de la web de nuestra API en `./docs/html`. Podemos abrir el fichero `index.html` con nuestro navegador; esta página muestra primeramente el contenido que tuviésemos en `README.md`.

NOTA

Otra forma muy conveniente de abrir la documentación en MacOS es insertar el siguiente comando desde la terminal (estando en el directorio con el fichero `Doxyfile`):
`open docs/html/index.html`
En MacOS, el comando `open <file>` trata de abrir el fichero indicado con la aplicación configurada por defecto para ese tipo de ficheros. En el caso de ficheros HTML, esta aplicación es nuestro navegador.

Apéndice C

Tabla de caracteres ASCII

Josué Pagán

Aquí se muestra la tabla ASCII[1]. En la imagen se indican aquellos caracteres que son imprimibles, y cuáles no.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_			
127	7F	[DEL]									

No imprimibles

Imprimibles

Figura C.1: Tabla de caracteres ASCII.

[1]Autor original: Psychpsyo. Wikimedia Commons `https://simple.m.wikipedia.org/wiki/File:ASCII-Table.svg`