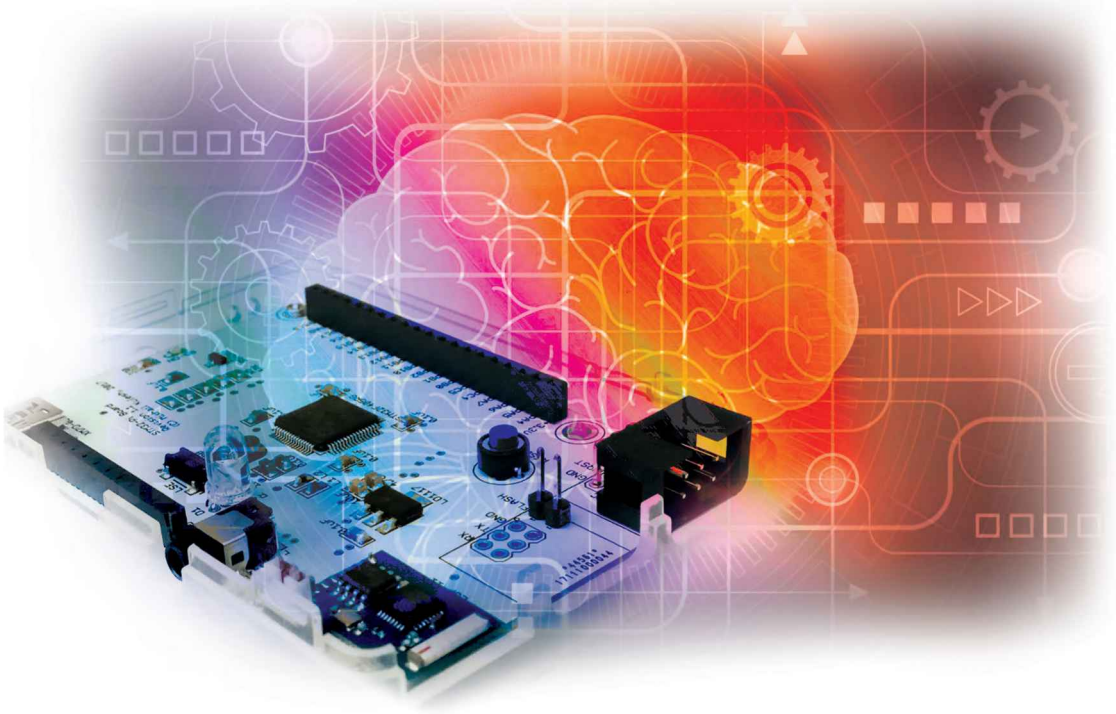
Microcontrolador STM32

Programación y Desarrollo



Jesús María Pestano Herrera

www

Desde **www.ra-ma.es** podrá descargar material adicional.

Ra-Ma®

Microcontrolador STM32

Programación y Desarrollo

Microcontrolador STM32

Programación y Desarrollo

Jesús María Pestano Herrera

Ra-Ma®



Microcontrolador STM32. Programación y Desarrollo
© Jesús María Pestano Herrera
© De la edición: Ra-Ma 2018

MARCAS COMERCIALES. Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. RA-MA ha intentado a lo largo de este libro distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma. Los datos de los ejemplos y pantallas son ficticios a no ser que se especifique lo contrario.

RA-MA es marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso ni tampoco de cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa o de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente, ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro sin autorización previa y por escrito de RA-MA; su contenido está protegido por la ley vigente, que establece penas de prisión y/o multas a quienes, intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

Editado por:
RA-MA Editorial
Calle Jarama, 3A, Polígono Industrial Igarsa
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: *editorial@ra-ma.com*
Internet: *www.ra-ma.es* y *www.ra-ma.com*
ISBN: 978-84-9964-755-5
Depósito legal: M-21410-2018
Maquetación: Antonio García Tomé
Diseño de portada: Antonio García Tomé
Filmación e impresión: Safekat
Impreso en España en junio de 2018

En agradecimiento a la inestimable ayuda
de mi mujer Loly y de mi hija Patricia.

ÍNDICE

PRESENTACIÓN DEL AUTOR **11**

PARTE I. INTRODUCCIÓN **15**

CAPÍTULO 1. STM32 INTRODUCCIÓN **17**

1.1 ARM CORTEX 17

1.2 ¿QUÉ ES EL STM32F103? 19

CAPÍTULO 2. PREPARACIÓN PREVIA DE LA PLACA **23**

2.1 SELECCIÓN DEL SWITCH BOOT0 PARA PROGRAMACIÓN 23

CAPÍTULO 3. PRIMEROS PASOS EN LA PROGRAMACIÓN **25**

3.1 PROGRAMAR NUESTRA PLACA 25

3.2 PRIMEROS PASOS CON EL IDE DE ARDUINO 28

3.2.1 PROGRAMANDO NUESTRA PLACA CON ARDUINO 31

3.2.2 SOLUCIÓN A ALGUNOS ERRORES INICIALES 31

PARTE II. ENTORNOS DE DESARROLLO **33**

CAPÍTULO 4. PRIMEROS PASOS CON COOCOX COIDE **35**

4.1 QUÉ ES EL COOCOX COIDE 35

4.2 INSTALACIÓN DEL ENTORNO 36

4.3 CREAR NUESTRO PRIMER PROYECTO EN COOCOX COIDE 40

4.4 PROGRAMANDO LA PLACA CON EL ADAPTADOR ST-LINK EN COOCOX COIDE 54

4.5 SOLUCIÓN DE ALGUNOS ERRORES EN LA PROGRAMACIÓN 57

4.6 OPCIONES DE DEPURACIÓN CON EL COOCOX COIDE 58

CAPÍTULO 5. PRIMEROS PASOS CON KEIL MDK ARM 67
5.1 INSTALACIÓN 68
5.2 CREAR UN PRIMER PROYECTO CON KEIL 72

CAPÍTULO 6. PROGRAMANDO NUESTRA PLACA 85
6.1 PROGRAMANDO CON EL ADAPTADOR USB A RS232 85
6.1.1 INSTALACIÓN DEL DRIVER DEL ADAPTADOR USB A RS232 CH340 85
6.1.2 INSTALACIÓN DEL DRIVER DEL ADAPTADOR USB A RS232 PROLIFIC (PL-2303) 87
6.1.3 PROGRAMAR LA PLACA CON UN ADAPTADOR USB a RS232 y EL FLASH LOADER DEMONSTRATOR DE ST 92
6.2 PROGRAMANDO CON EL ADAPTADOR ST-LINK 96
6.2.1 PROGRAMANDO EN ARDUINO CON EL ADAPTADOR ST-LINK 98
6.2.2 PROGRAMANDO DIRECTAMENTE CON EL SOFTWARE ST-LINK UTILITY 99
6.2.3 CONFIGURAR EL KEIL PARA PROGRAMAR CON EL ADAPTADOR ST-LINK (Método 1) 101
6.2.4 CONFIGURAR EL KEIL PARA PROGRAMAR CON EL ADAPTADOR ST-LINK (Método 2) 104
6.2.5 ACTUALIZACIÓN DEL FIRMWARE DEL ADAPTADOR ST-LINK 107

CAPÍTULO 7. PRINCIPIOS BÁSICOS DEL HARDWARE DE LOS STM32 113
7.1 PUERTOS Y PINES 114
7.2 PERIFÉRICOS INTERNOS 115
7.3 MÉTODO DE PROGRAMACIÓN 116

PARTE III. PROGRAMACIÓN DEL MICROCONTROLADOR STM-32 CON C++ 119

CAPÍTULO 8. PROGRAMACIÓN GPIO 121
8.1 PROGRAMACIÓN SYSCLK 130
8.1.1 PLL 131

CAPÍTULO 9. PROGRAMACIÓN USART 135
9.1 EJEMPLO DE CONFIGURACIÓN PUERTO USART 136
9.2 EJEMPLO CON OTRO PUERTO USART 140
9.3 EJEMPLO REMAPEO DE PUERTO USART 145
9.4 EJEMPLO DE INTERRUPCIONES DEL PUERTO USART 149

CAPÍTULO 10. PROGRAMACIÓN DE INTERRUPCIONES (NVIC) 151
10.1 EJEMPLO DE CONTROL DE INTERRUPCIÓN EXTI_0 155
10.2 EJEMPLO DE CONTROL DE INTERRUPCIÓN USART 160
10.3 EJEMPLO DE DETECCIÓN DE MOVIMIENTO Y EXTI9_5 163

CAPÍTULO 11. PROGRAMACIÓN TIMER 167
11.1 EJEMPLO DE TIMER COMO TEMPORIZADOR 174
11.2 EJEMPLO DEL TIMER COMO CONTADOR 177
11.2.1 EJEMPLO DE MEDICIÓN DE TIEMPOS ENTRE DOS EVENTOS 177
11.2.2 EJEMPLO DE USO DEL SENSOR HC-SR04 181
11.2.3 OTRO EJEMPLO DE MEDICIÓN DE TIEMPOS ENTRE EVENTOS 186
11.3 CONTROL DEL WATCHDOG TIMER 188
11.3.1 EJEMPLO DE EMPLEO DEL IWDG 189
11.3.2 EJEMPLO DE EMPLEO DEL WWDG 194

CAPÍTULO 12. PROGRAMACIÓN PWM 201
12.1 EJEMPLO DE SEÑAL PWM 204
12.2 EJEMPLO DE SEÑAL PWM CONTROLANDO EL BRILLO DE UN LED 207
12.3 EJEMPLO PWM CONTROL DE BRILLO DE UN LED TRICOLOR 210
12.4 EJEMPLO PWM CONTROLANDO UN SERVO MOTOR 213
12.5 EJEMPLO PWM GENERANDO SONIDOS EN UN ALTAVOZ 216

CAPÍTULO 13. PROGRAMACIÓN ADC 221
13.1 EJEMPLO ADC EN MODO CONTINUO 226
13.2 EJEMPLO CON EL SENSOR DE TEMPERATURA INTERNO 232
13.3 EJEMPLO ADC EN MODO MÚLTIPLES CANALES. (DMA) 235
13.4 EJEMPLO ADC EN MODO MÚLTIPLES CANALES INYECTADO 239
13.5 EJEMPLO ADC CON WATCHDOG. (AWD) 243
13.6 MÓDULO DAC 246

CAPÍTULO 14. PROGRAMACIÓN DMA 251
14.1 EJEMPLO DE COMUNICACIÓN SERIAL CON EL USART EMPLEANDO EL DMA 256

CAPÍTULO 15. PROGRAMACIÓN RTC 261
15.1 EJEMPLO DE CONTROL DE HORARIO CON EL RTC 272
15.2 EJEMPLO DE CONFIGURACIÓN DE UNA ALARMA CON EL RTC 277
15.3 EJEMPLO DE CONFIGURACIÓN DE UN CALENDARIO CON EL RTC 285

CAPÍTULO 16. PROGRAMACIÓN BKP Y FLASH 295
16.1 EJEMPLO DE UTILIZACIÓN DEL BKP 296
16.2 EJEMPLO DE UTILIZACIÓN DE VARIOS REGISTROS DEL BKP 298
16.3 EMPLEO DE LA MEMORIA FLASH 300

CAPÍTULO 17. PROGRAMACIÓN CON PANTALLAS 309
17.1 EJEMPLO DE EMPLEO DE PANTALLAS LCD 16X2 310
17.2 EJEMPLOS DE EMPLEO DE PANTALLAS MEDIANTE ADAPTADOR I2C 318
17.3 EJEMPLO DE EMPLEO DE PANTALLAS OLED 322

CAPÍTULO 18. PROGRAMACIÓN I2C 325
18.1 EJEMPLO DE EMPLEO DE UNA EEPROM I2C 328

CAPÍTULO 19. PROGRAMACIÓN SPI 337
19.1 EJEMPLO DE PROGRAMACIÓN SPI DE UN MAX7912 339

CAPÍTULO 20. PROGRAMACIÓN USB 353
20.1 EJEMPLO DE CONEXIÓN USB COMO PUERTO COM VIRTUAL 355
20.2 EJEMPLO DE EMULACIÓN DE UN RATÓN Y UN TECLADO USB 360

MATERIAL ADICIONAL 363

PRESENTACIÓN DEL AUTOR

Siempre he sido un apasionado de la electrónica y la informática; creando y desarrollando proyectos de robótica, de automatización y electrónica con los que seguir aprendiendo y experimentando con todo lo que caía en mis manos. Durante estos años, he seguido muy de cerca y de primera mano la evolución que han tenido estas tecnologías, con las que empecé en el año 1985 cuando compré mi primer Philips MSX, con un flamante microprocesador Zilog Z80 de 8 bits.

Más tarde, con la aparición de los primeros circuitos de lógica programable, como los chips PAL y GAL, se abrió un nuevo mundo de posibilidades a la creación de circuitos que me permitieron crear proyectos destinados a realizar funciones específicas, que, hoy en día, han evolucionado hasta los actuales microcontroladores que forman los sistemas electrónicos que pueden poseer, por ejemplo, una cafetera, un horno microondas, un smartphone o un coche; lo que se conoce actualmente, como el "Internet de las cosas".

En nuestros días los microcontroladores ya no se distancian tanto de los microprocesadores que componen el corazón de cualquier sistema; hasta el punto, de ser casi utilizados en ambientes de igual a igual. De esto es, precisamente, de lo que trata este libro.

INFORMACIÓN SOBRE MARCAS Y REGISTROS

- ✓ ATMEL® CORP. Empresa de fabricación de microcontroladores, fundada en 1984 por George Perlegos. En 2016 fue adquirida por la empresa **Microchip Technology**, fabricante de los microcontroladores PIC (*https://www.microchip.com/*).
- ✓ AVR®. Familia de microcontroladores con tecnología RISC del fabricante estadounidense **Atmel Corp**.
- ✓ ARDUINO®. Y demás logotipos de Arduino publicados en esta obra, son marcas registradas de Arduino AG (*https://www.arduino.cc*).
- ✓ WINDOWS© Y MICROSOFT©. Son marcas registradas de Microsoft Corporation (*https://www.microsoft.com/*).
- ✓ STMICROELECTRONICS®. Empresa multinacional de fabricación de componentes electrónicos (*http://www.st.com*).
- ✓ ST-LINK/V2®. Circuito programador y depurador desarrollado y comercializado por la empresa STMicroelectronics (*http://www.st.com/en/development-tools/st-link-v2.html*).
- ✓ MAC OSX Y APPLE. Son marcas registradas de Apple Inc. (*https://www.apple.com*).
- ✓ COOCOX IDE. Es marca registrada de Active-Semi, Inc. (*https://active-semi.com*).

PARTE I

INTRODUCCIÓN

1

STM32 INTRODUCCIÓN

1.1 ARM CORTEX



Figura 1.1

La arquitectura de los micros ARM Cortex®, se basa principalmente en su sistema de instrucciones reducido RISC (*Reduced Instruction Set Computer*), que se desarrolló en las universidades de Standford y Berkeley en 1980; que, aunque no brinda la complejidad que poseen los micros con tecnología CISC (*Complex Intruction Set Computer*) que utilizan los Intel x86 y los AMD x86-64, sí incorporan una novedad y mejora evolutiva importante respecto a esta última: la tecnología denominada "Thumb"; en la que el conjunto de instrucciones ha evolucionado significativamente añadiendo funciones y flexibilidad al comprimir instrucciones de 32 a 16 bits y en la que cada instrucción, es ejecutada en un solo ciclo de reloj. Esto repercute sobre todo, en la densidad de código que se genera en la compilación;

facilitando mucho el límite de tamaño del espacio de memoria en los sistemas embebidos y produciendo una ejecución más rápida que en un procesador CISC con la misma velocidad de reloj.

ARM Holdings es una empresa multinacional británica que, aunque está dedicada a los semiconductores, no fabrica microprocesadores. Su principal negocio es el diseño de procesadores que luego vende como licencias IP (*intelectual property*) de propiedad industrial a otras empresas que sí fabrican micros ARM, y, a las que, también les ofrece servicios de consultoría de diseño y software. Ofrece así soluciones esenciales para el proceso de fabricación, su desarrollo y las validaciones finales de funcionalidad de sus diseños.

Gran cantidad de empresas fabricantes de semiconductores o integrados han firmado con ARM sus licencias: Analog Devices, AppliedMicro, Atmel, Broadcom, Cirrus Logic, Energy Micro, Faraday Technology, Freescale, Fujitsu, Intel, IBM, Infineon Technologies, Marvell Technology Group, Nintendo, NXP Semiconductors, OKI, Qualcomm, Samsung, Sharp, STMicroelectronics, and Texas Instruments.

En el año 2003, se produjo un revulsivo en el mercado de fabricantes de Procesadores de la Industria Móvil (MIPI), donde se creó una iniciativa industrial que unió a las empresas ARM, Nokia, STMicroelectronics y Texas Instruments y en la que se estableció un objetivo común de promover y definir un estándar para interfaces de procesadores de aplicaciones. En octubre de 2006, la empresa STMicroelectronics anunciaba que se unía con una licencia de ARM al desarrollo y fabricación de núcleos ARM Cortex-M3 y, posteriormente en Junio de 2007, anunciaba la comercialización de la serie STM32F1 basada en los ARM Cortex-M3.

De esta empresa STMicroelectronics y de su familia de microcontroladores STM32 es de lo que trata este libro.

Más concretamente su serie STM32F1, que ha convertido a esta empresa en pionera en el mundo de los microcontroladores ARM Cortex-M; imponiéndose en los dispositivos con un alto rendimiento, un significativo bajo consumo de energía y voltaje y sobre todo, a unos precios accesibles.

Esta serie de microcontroladores poseen características muy avanzadas y una potencia importante, con núcleos de 32 bits basados en los procesadores ARM Cortex-M3, memorias de 16 Kilobytes hasta 1 Mb y velocidad de procesamiento que va desde 72 MHz hasta 180 MHz.

Junto con esta familia de microcontroladores, esta empresa ha desarrollado también, todo un conjunto de herramientas de software gratuitas de edición y compilación de código en C y C++, que proporcionan a los desarrolladores un marco

inicial muy interesante para la creación e integración de proyectos de cualquier tamaño.

Otra característica importante de esta familia de micros, es que ha sido añadida a la familia de Arduino®; pudiendo ser programados también con el IDE de Arduino®. Más adelante en nuestro libro, explicaremos los pasos necesarios para su compilación y programación mediante diferentes compiladores y entornos más sobresalientes de forma detallada y sencilla.

Como base para los proyectos y ejemplos que se incluyen en nuestro libro, hemos escogido un microcontrolador de esta serie, el STM32F103, que se incluyen en gran parte de las placas más populares y económicas para el desarrollo de proyectos.

1.2 ¿QUÉ ES EL STM32F103?

En este libro, hemos seleccionado para el desarrollo de los ejemplos que se explican, una de las placas más económica y sencilla con este micro; la "blue pill" que se muestra en la Figura 1.3.

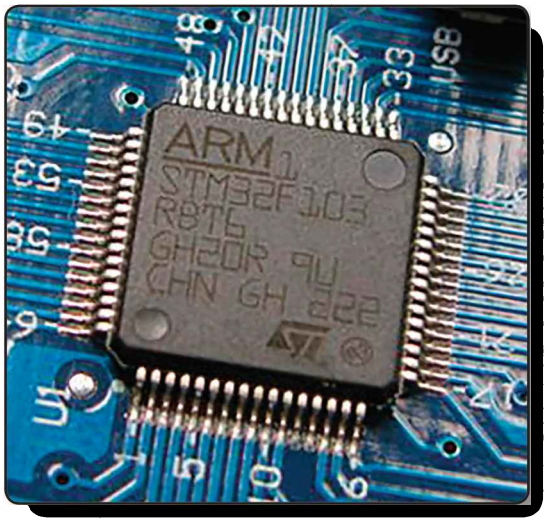


Figura 1.2

No obstante, como establece el principio de los ARM Cortex M, toda la programación que se explicará en este libro puede compilarse y ejecutarse en cualquier micro superior de la familia sin ninguna restricción; salvo la de cambiar determinados números de pines o puertos según el micro empleado.

Esta placa posee un STM32F103C8, perteneciente a los microcontroladores de media densidad de la familia ARM Cortex-M3, fabricado por STMicroelectronics y que entre sus características más importantes tiene:

- Frecuencia de 72 MHz (1.25 DMIPS/MHz).
- 64 Kilobytes de memoria Flash.
- 20 Kilobytes de SRAM.
- 8 MHz de reloj en placa.
- RTC (reloj de tiempo real) integrado.
- Modo Sleep, Stop y Standby.
- 26 entradas y salidas digitales, la mayoría tolerantes a 5 Vcc.
- Interrupciones en todos los pines I/O.
- 2 conversores A/D de 12-bits de 1 µseg (10 entradas analógicas).
- 7 temporizadores TIM.
- 2 puertos I2C.
- 3 puertos RS232 USARTs.
- 2 puertos SPI a 18 Mbit/seg.
- 1 puerto CAN.
- Micro USB para alimentación de la placa y comunicaciones.

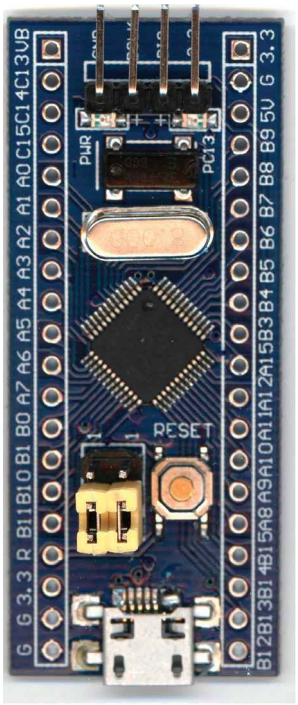


Figura 1.3

Funciona con una tensión de entre 2 Vcc y 3.6 Vcc, lo que le otorga un pequeño consumo de 150 mA; que la hace muy atractiva para proyectos con baterías o que requieran de muy bajos consumos. Posee un gran número de entradas y salidas que pueden programarse como interrupciones; un variado número de periféricos conectados a dos buses de reloj; dos ADC de 12 bits; siete timers de propósito general de 16 bits; varios dispositivos de comunicaciones estándar y avanzadas como I2C, SPI, USART, USB, CAN y además una memoria flash que puede utilizarse tanto como soporte de datos como para guardar nuestro firmware.



Figura 1.4

Cuando se emprende un nuevo proyecto en programación, es importante poder contar con ejemplos y librerías que nos aporten conocimiento y sencillez en la realización de estos. Por ello, es tan significativo el que estos microcontroladores se comercialicen bajo el "estándar de interfaz de software de microcontrolador Cortex", **-CMSIS** (*Cortex Microcontroller Software Interface Standard*)-, que otorga un soporte en la programación de todos los periféricos y componentes integrados en estos microcontroladores ARM Cortex-M3.

En este libro abordaremos los conceptos principales para la programación y desarrollo de nuestros proyectos con los periféricos y módulos que contienen estos microcontroladores bajo el lenguaje C++ dejando, por el momento, la terminología técnica más compleja fuera del ámbito de esta obra.

Casi todos los ejemplos que veremos en este libro, han sido realizados y comprobados para las placas más sencillas del mercado por lo que, en poco tiempo, podrán aprender a desarrollar su propio firmware para este tipo de microcontroladores.

2

PREPARACIÓN PREVIA DE LA PLACA

2.1 SELECCIÓN DEL SWITCH BOOTO PARA PROGRAMACIÓN

Estos microcontroladores ofrecen la posibilidad de poder cargar nuestro firmware tanto en la memoria SRAM interna como en su memoria Flash; para ello, poseen uno de los pines de su chip denominado BOOT0. Cuando el microcontrolador se inicia, por defecto comprueba el estado en que está ese pin y dependiendo de si está a masa GND o a señal de tensión VDD, seleccionará si el gestor de arranque reubicará, el firmware cargado en él, en la memoria RAM o en la memoria Flash interna.

Para controlar esta funcionalidad, nuestra placa posee unos jumpers, que se muestran en la Figura 2.2; con los que podemos configurar y establecer este modo de inicio de nuestro microcontrolador. Si queremos que nuestra programación se ejecute en la RAM, para depuración y pruebas, el jumper BOOT0 deberá estar en la posición '1'; de modo que, se ejecute solo cuando está alimentada la placa y después se borre al reiniciar el sistema o al pulsar el botón de reset.

Si este jumper, el BOOT0, está en su posición '0', nuestra programación se cargará y ocupará la memoria de usuario Flash y se ejecutará cada vez que se inicie el microcontrolador, no perdiéndose aunque pulsemos el botón reset de la placa; quedando grabado de forma permanente, hasta que volvamos a grabar un nuevo proyecto en la placa.

BOOT Mode Selection Pins		Boot Mode	Aliasing
BOOT1	BOOT0		
x	0	Flash	Memoria flash de usuario
0	1	RAM	Memoria RAM
1	1	Ext. SRAM	Mem.ext.SRAM

Figura 2.1

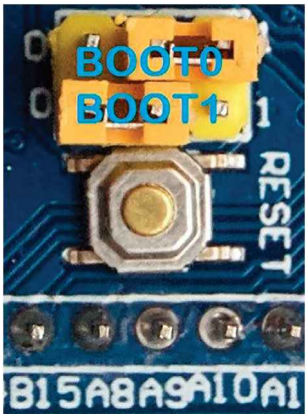


Figura 2.2

3

PRIMEROS PASOS EN LA PROGRAMACIÓN

3.1 PROGRAMAR NUESTRA PLACA

La mayoría de placas con este microcontrolador pueden ser programadas mediante un sencillo adaptador de puerto USB a RS232, al poseer un "bootloader" que cuando se inicia comprueba el puerto RS232 por si es necesario cargar archivos o firmware nuevo.

Sin embargo, el fabricante STMicroelectronics, también ha desarrollado un adaptador, el ST-Link V2, que nos permite no solo grabar proyectos nuevos en sus micros, sino que puede conectarse directamente a estos para realizar procesos de comprobación y depuración del código, mediante un interfaz especial SWIM o interfaz JTAG.

Como resultado, nuestras placas pueden ser programadas mediante dos métodos: con el adaptador ST-Link (Figuras 3-1 y 3-2) que posee un protocolo especial del fabricante STMicroelectronics y con un adaptador USB a serie (Figuras 3-3 y 3-4).



Figura 3.1. Programadores ST-Link originales de la casa STMicroelectronics.

También existen en el mercado adaptadores clónicos de este programador. (Figura 3-2)



Figura 3.2. Adaptador compatible.

Cualquiera de las placas existentes en el mercado de **USB a Serial**, tanto con el chip CH340, como con el FT232RL, o con el PL 2303, nos sirven; pero **lo más importante es que deben poseer salida de voltaje a 3.3 Vcc.** Si tiene salida de **señal DTR**, mejor.

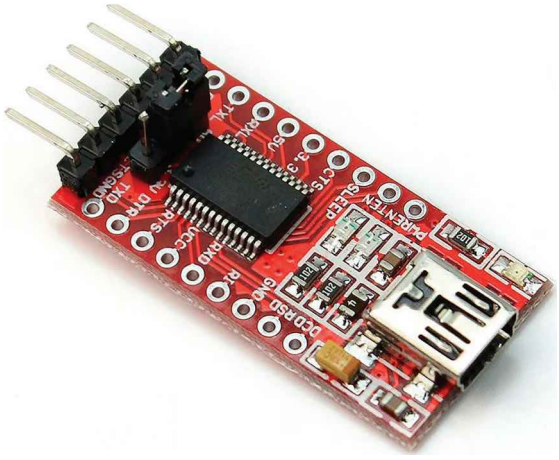


Figura 3.3

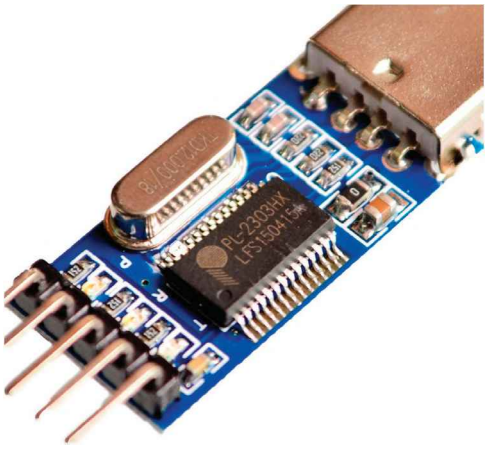


Figura 3.4

En este libro en el **Capítulo 6. PROGRAMANDO NUESTRA PLACA**, explicaremos los pasos necesarios para instalar, configurar y utilizar cada adaptador en la grabación de nuestra placa.

3.2 PRIMEROS PASOS CON EL IDE DE ARDUINO

Uno de los editores y compiladores más difundido hoy en día, entre los entusiastas del desarrollo de proyectos personales, es el entorno de Arduino®.

Éste ofrece un editor, el Arduino® IDE y una serie de placas que se comercializan bajo la denominación ARDUINO®, marca comercial de la empresa Arduino AG y de la que existe gran cantidad de información en internet y en la página oficial "*http://www.arduino.cc*".

Es por todo ello por lo que he incluido este apartado en el libro con el fin de que los lectores puedan usar estos microcontroladores en este entorno y poder utilizar las librerías y ejemplos que existen.

Para poder incluir los microcontroladores de la familia STM32 en el **IDE de Arduino®**, necesitaremos instalar primero una librería que nos reconozca y nos permita manejarlos**.** Estas librerías hoy en día son parte del repositorio incluido en el propio IDE y se instalan directamente desde él. A continuación explicaremos los pasos necesarios:

- **Paso 1**. Descargue e instale la última versión del IDE de la página oficial.
- **Paso 2**. Al iniciarlo, en el menú Herramientas, elegiremos la opción "**Placa**", seleccionaremos el "**Gestor de tarjetas…**", Figura 3.5; después en la pantalla de la Figura 3.6 buscaremos en la lista de librerías a instalar la "**Arduino SAM Boards (32-bits ARM Cortex-M3) by Arduino DUE**" de la lista de placas disponibles.

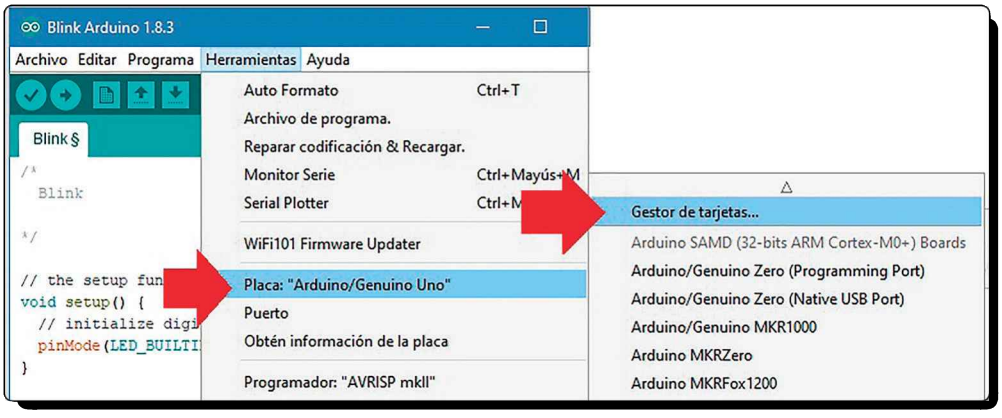


Figura 3.5

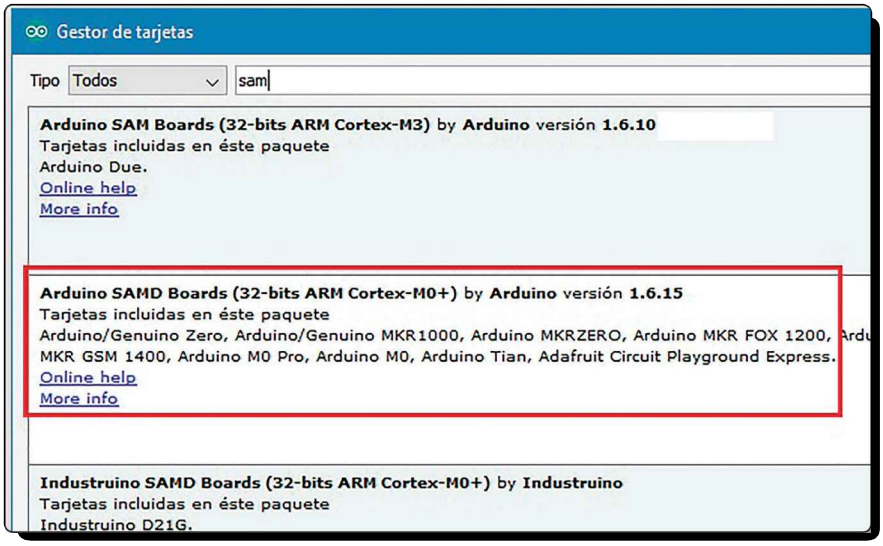


Figura 3.6

Con esto habremos descargado las librerías con las que se reconocerán las placas con micros STM32.

- **Paso 3**. Cierre y vuelva a iniciar el IDE, ahora en el menú "**Herramientas**" ya nos aparecerán las placas "**STM32 Boards...**" donde seleccionaremos la placa apropiada. En nuestro caso la "**Generic STM 32F103C Series**". (Figura 3-7)

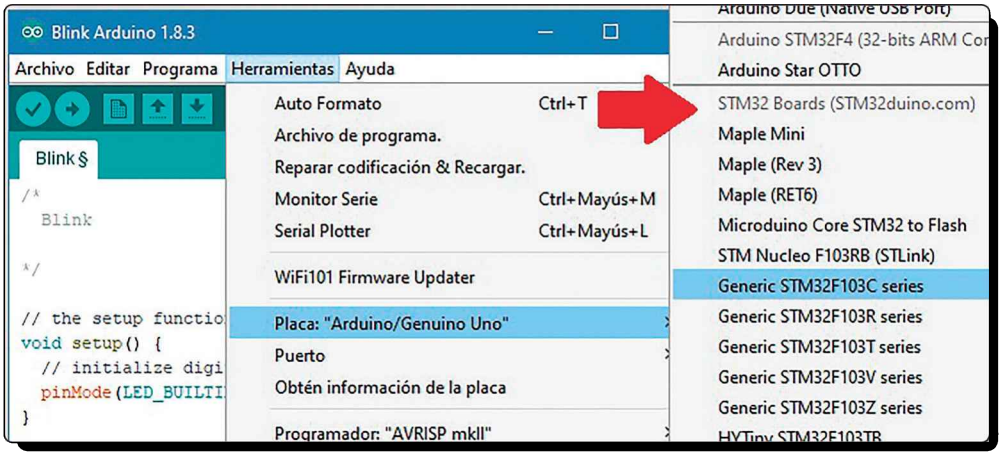


Figura 3.7

A continuación, debemos configurar a través del menú **Herramientas** las siguientes opciones (Figura 3-8).

El método empleado para programar la placa dependerá del adaptador que utilicemos. Si empleamos un adaptador USB a Serie, deberemos escoger en "**Upload method: "Serial**" y el "**Puerto: COMx**" –donde 'x' será el número del puerto que nos reconozca nuestro ordenador en donde está instalado el adaptador utilizado-.

Y si el adaptador es un **ST-Link**, deberemos seleccionar "**ST-Link**".

Ambos métodos los explicaremos con detalle en posteriores capítulos.

[Opciones según el Arduino IDE versión 1.8.x]

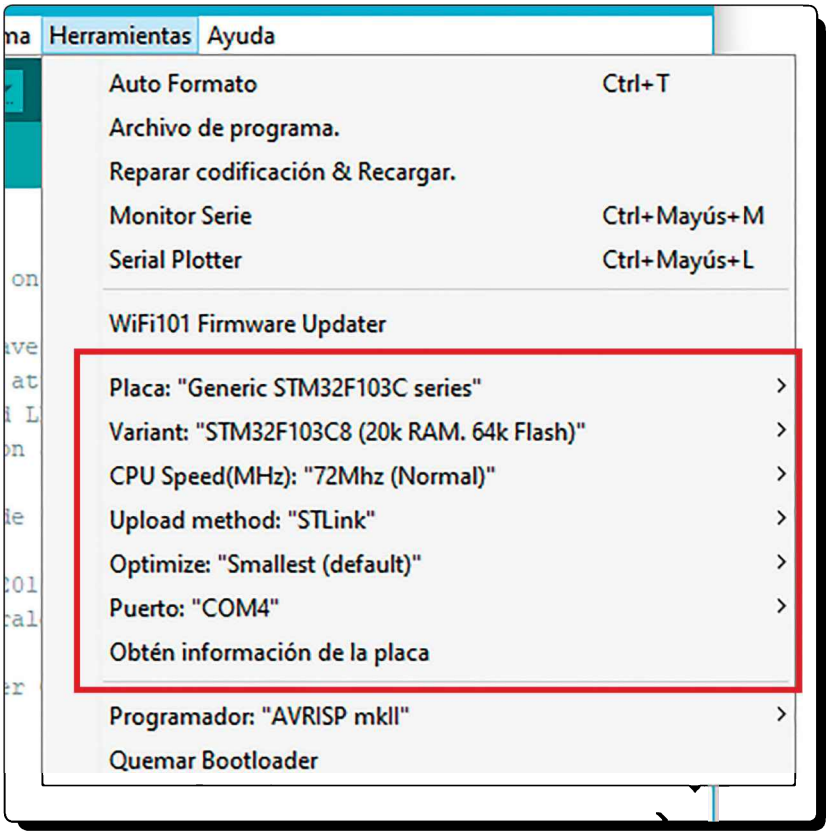


Figura 3.8

3.2.1 PROGRAMANDO NUESTRA PLACA CON ARDUINO

A continuación utilizaremos un sketch de los que vienen como ejemplo en la librería base del IDE para cargar un programa de prueba en nuestra placa por primera vez, comprobando así los pasos necesarios para probar la placa en esta primera ocasión.

- **Paso 1**. Conectaremos la placa a nuestro ordenador, utilizando el adaptador que tengamos para programar.
- **Paso 2**. Iniciamos el IDE de Arduino y una vez seleccionada el tipo de placa en el menú de Herramientas, recordemos que debemos seleccionar "Generic STM32F103C series", buscamos en los ejemplos del menú "**Archivo**", en el grupo "**Basics**" elegimos el de "**Blink**" que nos enciende y apaga un led.
- **Paso 3**. Después, seleccionamos la opción del adaptador que utilizamos para programar, en la opción de "**Upload method**:" ya sea "**Serial**" para un adaptador USB-RS232 o "**STLink**" si utilizamos el ST-Link V2.
- **Paso 4**. Por último seleccionamos la opción de cargar el sketch en nuestra placa.

Si todo nos ha ido correctamente, veremos cómo se enciende y apaga el led de pruebas que posee nuestra placa y habremos realizado la comprobación correctamente.

3.2.2 SOLUCIÓN A ALGUNOS ERRORES INICIALES

Es muy posible que no todo vaya todo lo bien que querríamos en nuestros primeros pasos. Pero si se producen estos fallos o errores, que son muy comunes cuando comenzamos a utilizar esta placa, esta es la manera de solucionarlo.

Si se produce el siguiente error:

"Error compilación en tarjeta Generic STM32F103C Series"
("Error compiling for board Generic STM32F103C series")

Puede deberse a que estemos utilizando una versión del IDE de Arduino incompatible con nuestro Core o librería STM-32 Arduino. Recuerde comprobar qué librerías hemos descargado e instalado en el IDE para que estemos seguros de que nos funcionen.

Si no se ha seleccionado correctamente el "Upload metod: "Serial" se producirá el error : Couldn't find the DFU device: [1EAF:0003] (Figura 3.9).

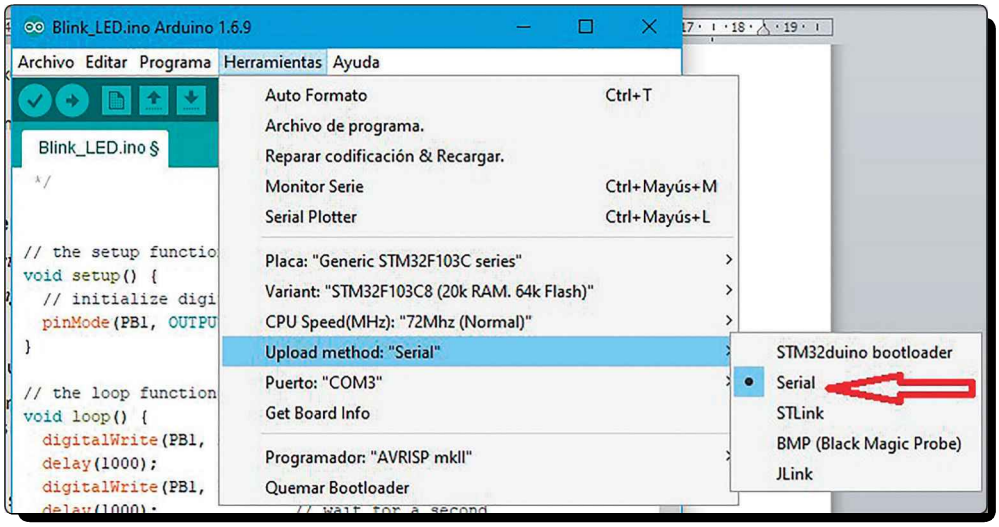


Figura 3.9

Un error que se produce muy a menudo, es el mensaje:

Using Parser : Raw BINARY
Interface serial_w32: 230400 SEL

Simplemente deberemos pulsar el **botón de reset en la placa** y volver a cargar el fichero.

PARTE II

ENTORNOS DE DESARROLLO

Como ya hemos dicho al principio, la empresa STMicroelectronics ofrece junto con la comercialización de sus microcontroladores una serie de herramientas para los desarrolladores de software que facilitan muchísimo la realización de proyectos.

Suministra, de manera gratuita, varias herramientas de software sin limitación ninguna para la creación, depuración y programación, para todos los dispositivos de la familia STM32 basados en los ARM Cortex-M0, M0+, M3 y M4.

Existen como entornos de desarrollo para estos micros, el **CooCox CoIDE**, el **AC6 SystemWorkbench**, **el Keil MDK-ARM** y el **STMCubeMX**; basados en el sistema Eclipse e incluyendo todas las librerías, configuraciones y firmware necesarios para cualquier chip de la gama de los que contienen también múltiples ejemplos. Todos ellos tienen versiones para sistemas basados en Windows®, Linux y Mac OSx©, con compatibilidad con el inglés, chino, japonés y coreano. Existiendo además continuas actualizaciones y soporte por parte de los distribuidores de forma gratuita.

Todas estas plataformas se pueden descargar directamente y de forma gratuita de los sitios web del fabricante.

En nuestro libro, hablaremos de los tres más importantes y explicaremos paso a paso cómo instalarlos y como crear nuestros primeros proyectos con: el **CooCox CoIDE, el STMCubeMX** y el **Keil MDK-ARM**.

Todos ellos, son compatibles entre sí, permitiendo su portabilidad de un desarrollo a otro.



Figura 4.1

4

PRIMEROS PASOS CON COOCOX COIDE

4.1 QUÉ ES EL COOCOX COIDE

CooCox proporciona un entorno de desarrollo mediante un conjunto completo de herramientas de software **GRATUITAS** que permiten el proceso rápido y la depuración de aplicaciones con todas las librerías y complementos necesarios para dispositivos con microcontroladores basados en los ARM Cortex-M. Estas herramientas incluyen CoIDE, un IDE o entorno integrado y basado en las herramientas Eclipse y GCC (*GCC-ARM-Embedded*) con mejoras y simplificación específicas para los Cortex M, permitiendo a la vez, que sus usuarios una vez registrados gratuitamente en su web, puedan utilizar una plataforma de colaboración de elementos de código en la nube que les permite cargar y descargar componentes de códigos reutilizables.



Figura 4.2

CoIDE puede organizar, extraer y compartir conocimientos a través de la sabiduría colectiva. Todo ello accesible a través de la siguiente página web: *http://www.coocox.org*

4.2 INSTALACIÓN DEL ENTORNO

A continuación veamos los pasos a seguir para descargar CooCox IDE e instalarlo en nuestro ordenador.

- **Paso 1.** Nos descargamos la última versión desde la siguiente dirección:

  http://coocox.org/software.html

- **Paso 2.** Ejecutaremos el programa de instalación que se nos descargó.



Figura 4.3

- **Paso 3.** Durante la instalación, en la Figura 4.4, nos pedirá que indiquemos el directorio donde se va a instalar el programa. Por defecto nos muestra un directorio en el raíz de nuestro ordenador; es recomendable que escojamos esa ubicación, ya que, cuando utilizamos como directorio otro dentro de la carpeta de programas, será preciso después autorizaciones y permisos de usuario especiales para poder ejecutar algunos comandos de programación de nuestra placa, lo que puede provocar errores.

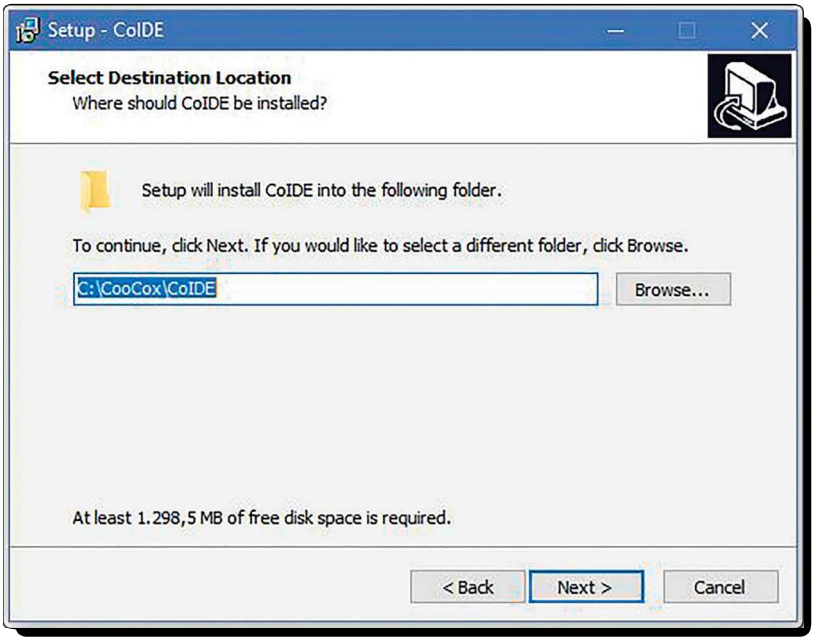


Figura 4.4



Figura 4.5

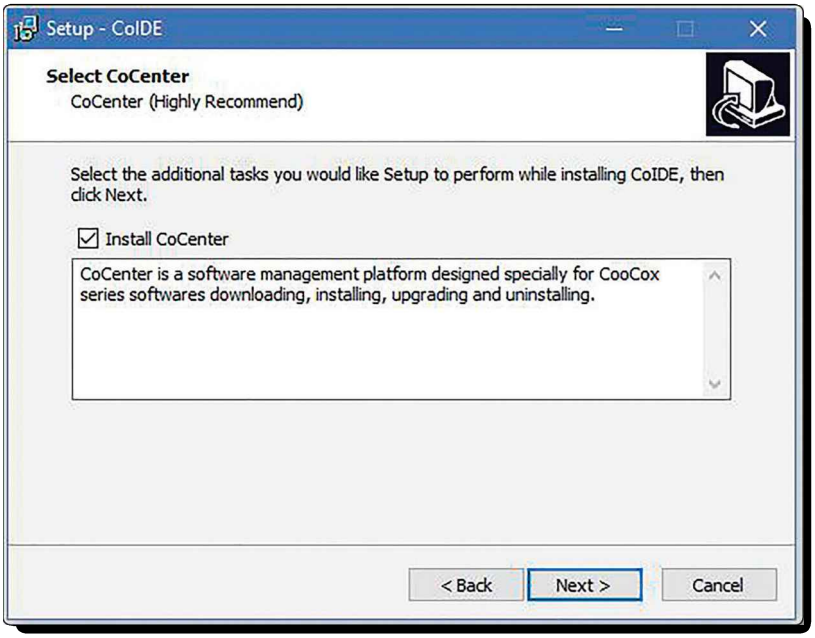


Figura 4.6

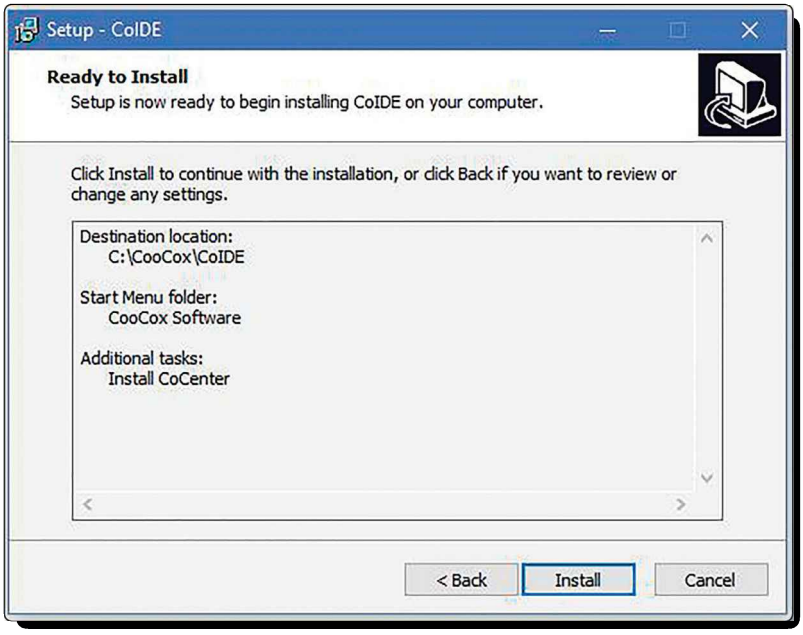


Figura 4.7

Tras pulsar en “Install” comenzará el proceso de instalación:

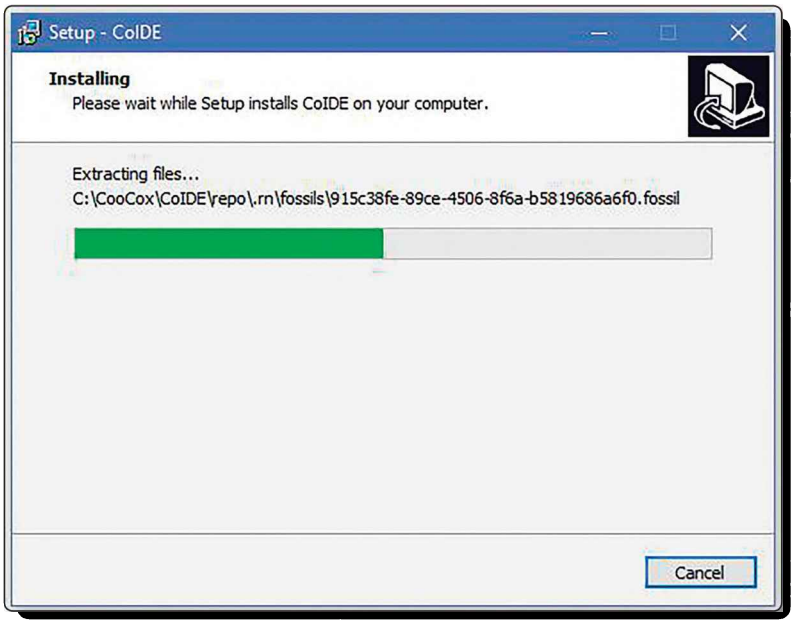


Figura 4.8

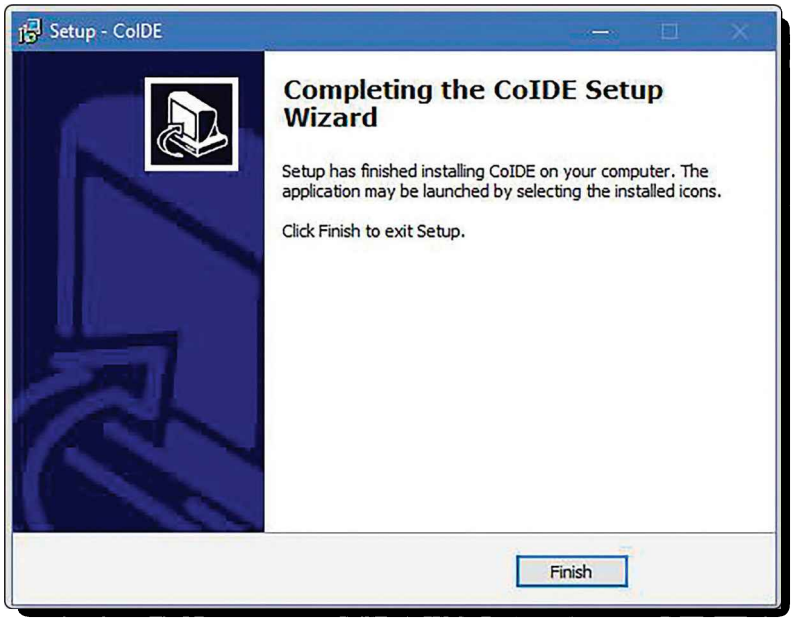


Figura 4.9

- **Paso 4.** Una vez realizada la instalación se nos crearán los siguientes dos iconos en nuestro escritorio (Figura 4-10)



Figura 4.10

El **CoIDE** es el icono del programa que inicia el entorno de desarrollo y el icono **CoCenter** nos permitirá acceder a la plataforma desde la cual conseguiremos librerías y pequeños trozos de código que podemos añadir a nuestros programas.

4.3 CREAR NUESTRO PRIMER PROYECTO EN COOCOX COIDE

Veamos ahora cómo crear nuestro primer proyecto.

Como siempre, crearemos el típico código que enciende un LED en la placa.

- **Paso 1.** Para crear un nuevo proyecto, seleccionamos en el menú "**Project**" la opción de "**New Project**", Figura 4.11.

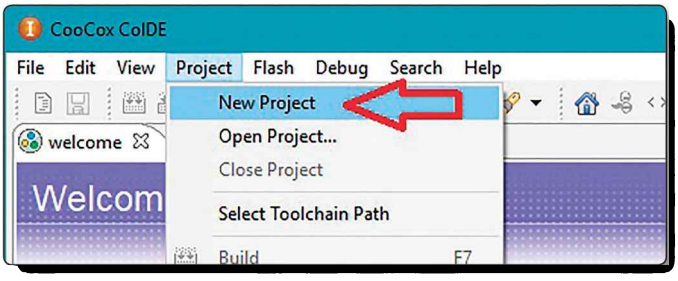


Figura 4.11

- **Paso 2.** La siguiente ventana, Figura 4.12, nos pedirá que indiquemos qué nombre tendrá nuestro proyecto. En nuestro ejemplo, pondremos "**Prueba_LED**".

  Es importante señalar, que por defecto el CooCox CoIDE guardará todos los proyectos dentro de una carpeta "*\workspace*" en el directorio donde lo hayamos instalado; por lo que, si queremos indicar una ubicación diferente, deberemos desmarcar la opción "**Use dafault path**".

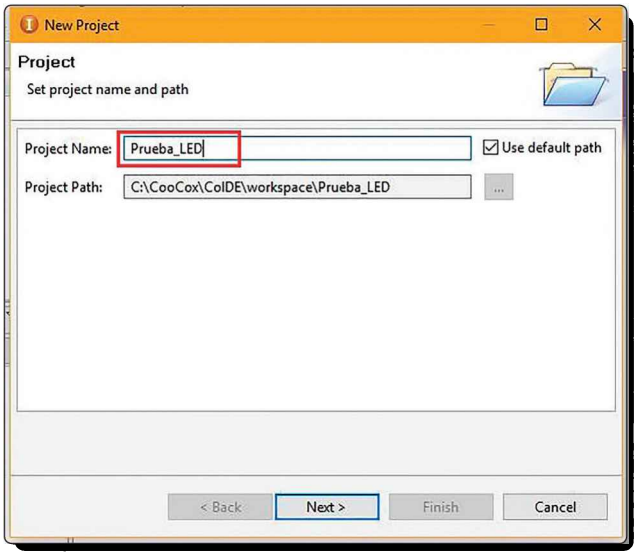


Figura 4.12

Paso 3. En la ventana de la Figura 4.13, nos pedirá que indiquemos si queremos seleccionar el chip o la placa para la que vamos a establecer nuestro proyecto. Tras colocar el cursor sobre el icono de chip, pulsamos para seguir.

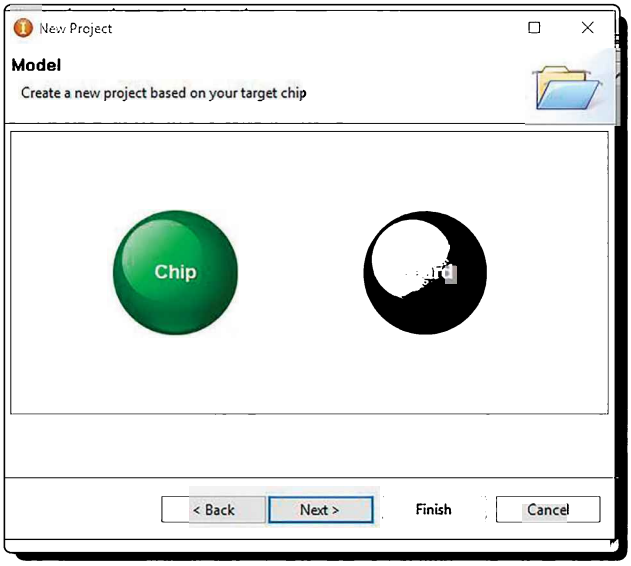


Figura 4.13

Paso 4. A continuación, deberemos buscar y seleccionar en la lista de fabricantes de microcontroladores de la izquierda de la pantalla, en la Figura 4.14, la marca "**ST**", luego la familia "**STM32F03x**" y por último, nuestro microcontrolador que, en nuestra placa de pruebas, es el "**STM32F103C8**".

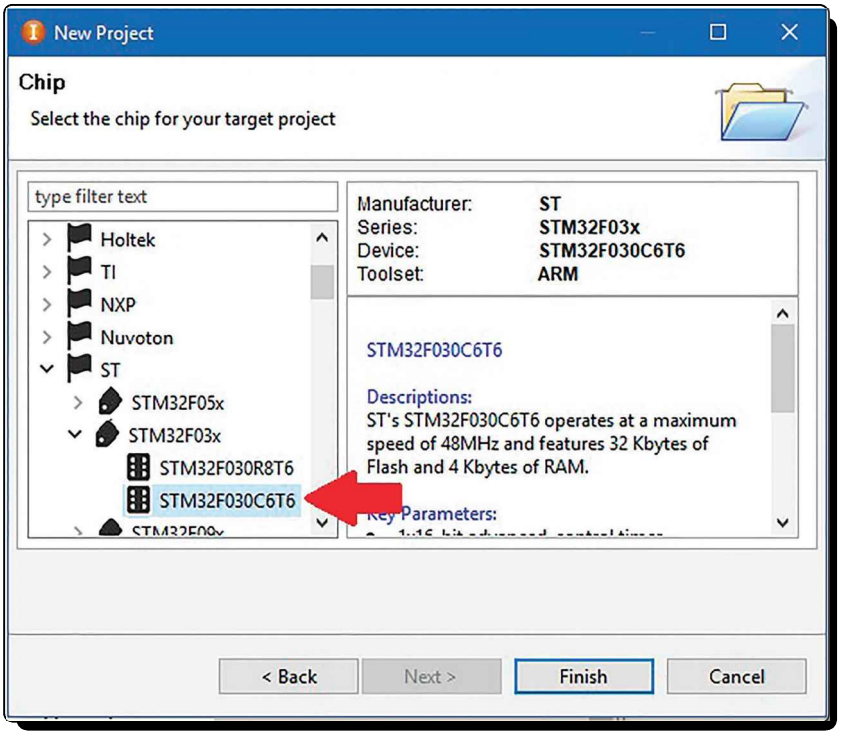


Figura 4.14

Paso 5. A continuación, se nos mostrará el entorno normal de trabajo.

Vemos que en las dos ventanas de la izquierda, nos aparecen: en la de "Componentes", el microcontrolador que hemos seleccionado –en color rojo- el **"[STM32F103C8]**" y, en la ventana inferior de la izquierda, nuestro proyecto "**Prueba_LED**" y el primer fichero en donde crearemos nuestro código. En la parte central de dicha pantalla, nos muestra un "Step 3 –Select Basic Components" – donde deberemos seleccionar los componentes básicos".

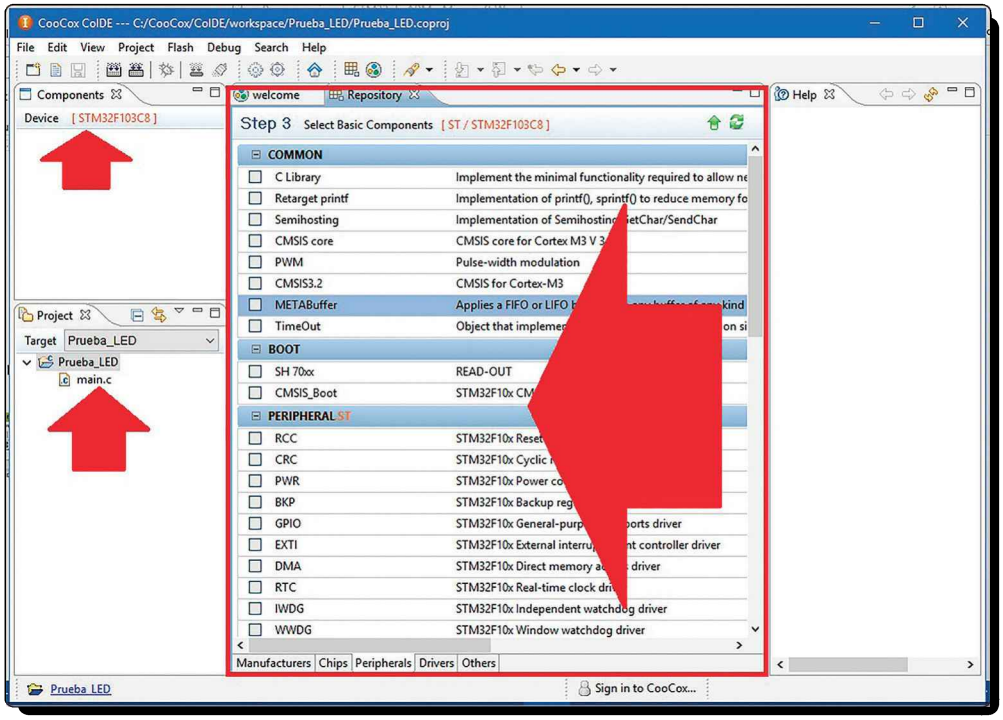


Figura 4.15

Este microcontrolador posee una serie de periféricos internos: puertos, registros de control, memorias, relojes, etc., y el fabricante nos ofrece además, una completa serie de librerías API que nos facilitan su configuración y programación. Pues bien, en el siguiente paso, necesitaremos seleccionar y cargar a nuestro proyecto aquellas librerías relacionadas con las funciones y periféricos que vayamos a emplear; y que se añadirán luego de forma interna en nuestra compilación. Nosotros solo tendremos que crear la subrutinas necesarias que llamen a los comandos que se incluyen en esas bibliotecas, para añadir determinadas funciones o configuraciones.

◤ **Paso 6.** Existen un grupo de librerías básicas que debemos siempre añadir a nuestros proyectos y que son imprescindibles:

COMMON:
- C Library y CMSIS core.

BOOT:
- CMSIS_Boot.

Después, en el apartado **PERIPHERAL ST,** iremos seleccionando aquellas librerías que nos permitirán controlar y gestionar los periféricos.

Por ejemplo, en nuestro primer proyecto de prueba, donde manipulamos un LED de muestra en la placa, necesitaremos señalar y añadir a nuestro proyecto. (Figura 4-16) las siguientes librerías.

PERIPHERAL ST:

- **GPIO.** (Librería para el control de los puertos de nuestro microcontrolador).
- **RCC.** (Librería para el control de los timers internos).

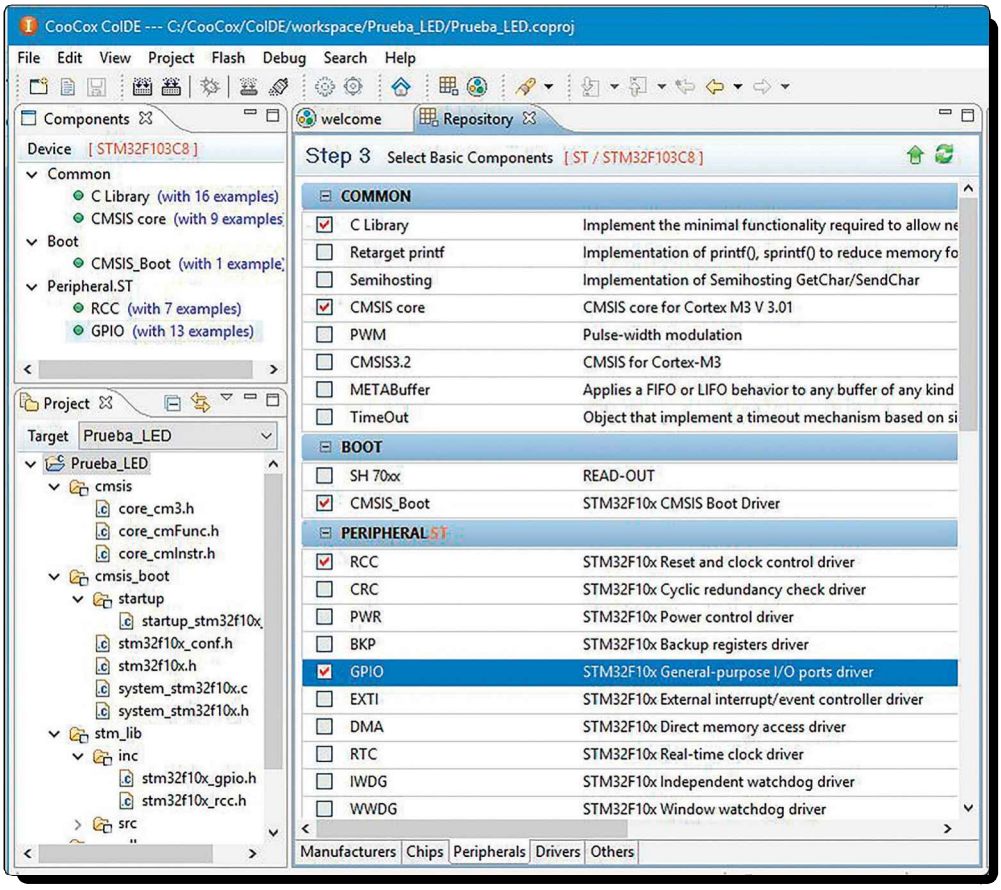


Figura 4.16

Obsérvese que, a medida que vamos marcando las librerías necesarias, en las ventanas de la izquierda de la pantalla se nos van añadiendo módulos de programación a nuestro proyecto, que luego serán compilados junto con nuestro código de programación.

- **Paso 7.** También veremos a la izquierda de la pantalla (Figura 4-17), en la ventana "**Project**" que existirá ya creado automáticamente un fichero "**main.c**", y que, al pulsar sobre él, se nos abrirá una nueva ventana con el editor en el centro de la pantalla para poder entrar nuestro código.

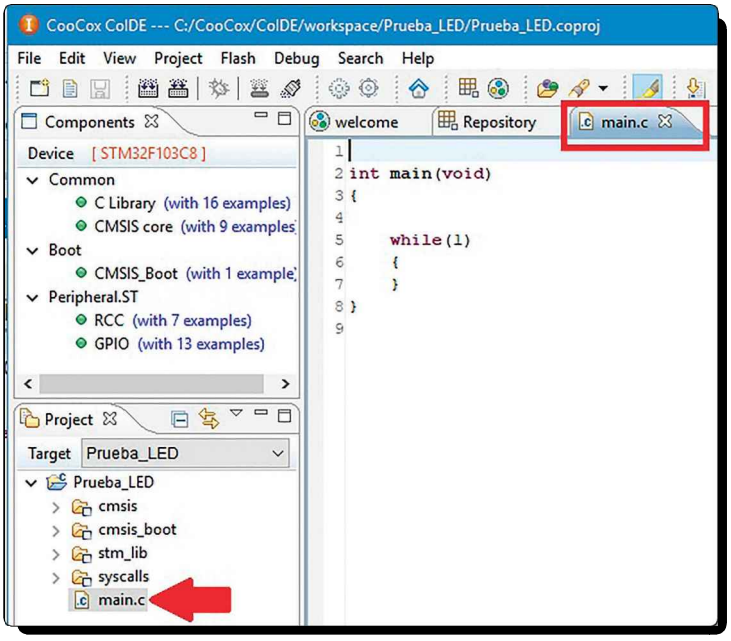


Figura 4.17

…y, en el que procedemos a copiar y pegar el siguiente código de ejemplo:

```
/*****************************************************************
 * Project Name: GPIO_LED_1
 * File Name: main.c
 * Revision:
 * Microcontroller name:STM32F103xxxx ARM
 * Compiler: CooCox IDE v.1.7.8
 * Author:   Jesus Pestano
 * E-mail:   jmpestanoh@gmail.com
 * Date :    2018.01.08
 * Important:Select in "Options for Target" the option
 *           (x) Use MicroLib
 * ===============================================================
```

```
* PROGRAMA DE PRUEBA QUE ENCIENDE UN LED.
*
* Enciende y apaga un LED de prueba de la placa conectado
* al pin PC13.
*
*******************************************************************************/

// Libreria principal del microcontrolador
#include "stm32f10x.h"

// Las librerias para los perifericos
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"

/* Definimos el Puerto y pin que tendra el LED */
#define LED_Pin     GPIO_Pin_13     // Creamos la variable LED_Pin como 'GPIO_Pin_13'
#define LED_GPIO    GPIOC           // Creamos la variable LED_GPIO como 'GPIOC'
#define RCC_APB2Periph_GPIOx RCC_APB2Periph_GPIOC

// Enumeramos las funciones que se anadiran
// ------------------------------------------------------------------
void Delay_ms(unsigned int nCount);
GPIO_InitTypeDef GPIO_StructureInit;

// Modulo Principal
//**********************************************************
int main(void){

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOx, ENABLE);   /* Establecemos el reloj del Puerto GPIOC (

    GPIO_StructureInit.GPIO_Pin = LED_Pin;          // Asignamos el Pin PC13 para configuracion
    GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;   // Establecemos la velocidad a 50MHz (Speed)
    GPIO_StructureInit.GPIO_Mode = GPIO_Mode_Out_PP;    // Establecemos el pin como Salida(output) y pu
    GPIO_Init(LED_GPIO, &GPIO_StructureInit);       // Iniciamos la estructura en el compilador

    while(1){
        GPIO_ResetBits(LED_GPIO, LED_Pin);      /* Borra el pin PC13 */
        Delay_ms(50);                   // Retardo de 50 ms
        GPIO_SetBits(LED_GPIO, LED_Pin);        /* Activa el pin PC13 */
        Delay_ms(100);                  // Retardo de 50 ms
    }
}

// Modulo que genera un retardo (ms)
//-----------------------------------------------------------
void Delay_ms(unsigned int nCount){
        unsigned int i, j;
        for(i = 0; i < nCount; i++)
        {
                for(j = 0; j < 0x2AFF; j++){;}
        }
}
```

Figura 4.18

Por ahora no nos entretendremos en explicar el código del ejemplo; a partir del capítulo 8 de nuestro libro ya se detallan paso a paso.

Veremos que, automáticamente, el editor del CooIDE reconocerá los comandos e instrucciones que hemos añadido a nuestro fichero "***main.c***" tal como se muestra en la Figura 4.19.

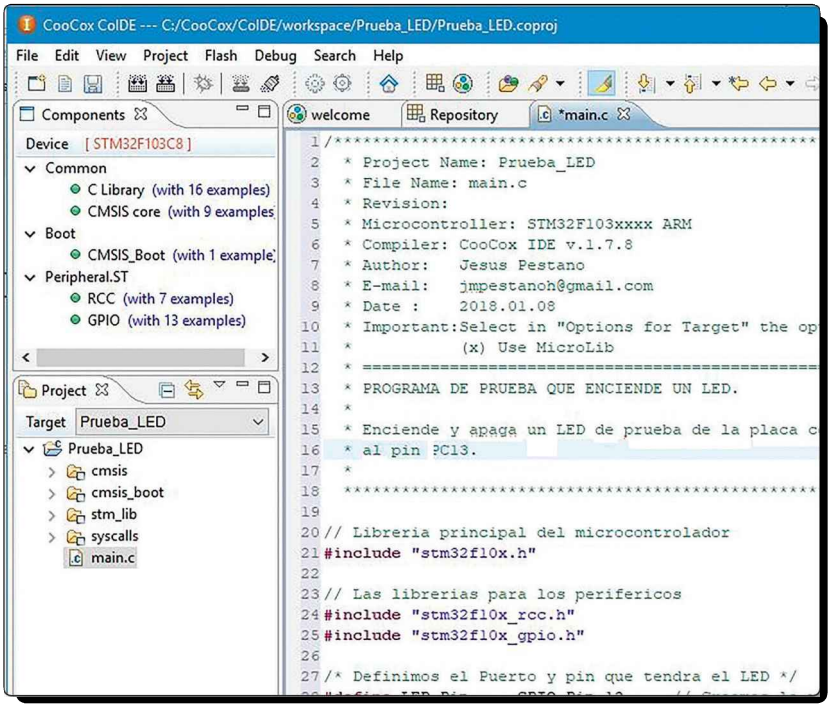


Figura 4.19

- **Paso 8.** Ahora, en el menú "**File**" seleccionaremos y pulsaremos sobre la opción "**Save**" para guardar el fichero "***main.c***" que hemos creado.

- **Paso 9.** Lo siguiente será compilar nuestro programa, para ello pulsaremos en "**Build**". (Figura 4-20)

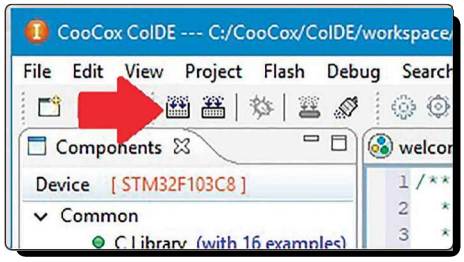


Figura 4.20

En este paso, si hemos seguido todos los pasos anteriores por primera vez, veremos que se nos mostrará una ventana de error como el de la Figura 4.21. No nos preocupemos, necesitaremos añadir unas librerías

específicas en nuestro entorno CooCox CoIDE que una vez instaladas, ya, no nos las solicitará más.

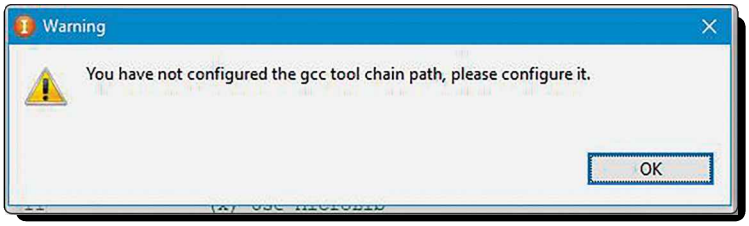


Figura 4.21

Este procedimiento pasamos a explicarlo a continuación.

Instalación de la librería del Compilador "GNU Tools for ARM Embedded Processors"

- **Paso 1.** Cerraremos la ventana de error y minimizaremos el escritorio del CooCox CoIDE.

- **Paso 2.** Descargaremos esta librería desde la página oficial siguiente:

  https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

  Pulsaremos botón "**Download**" y seleccionaremos la primera opción que se nos muestra de "**Windows 32bits**". Tal como se muestra en la Figura 4.22.

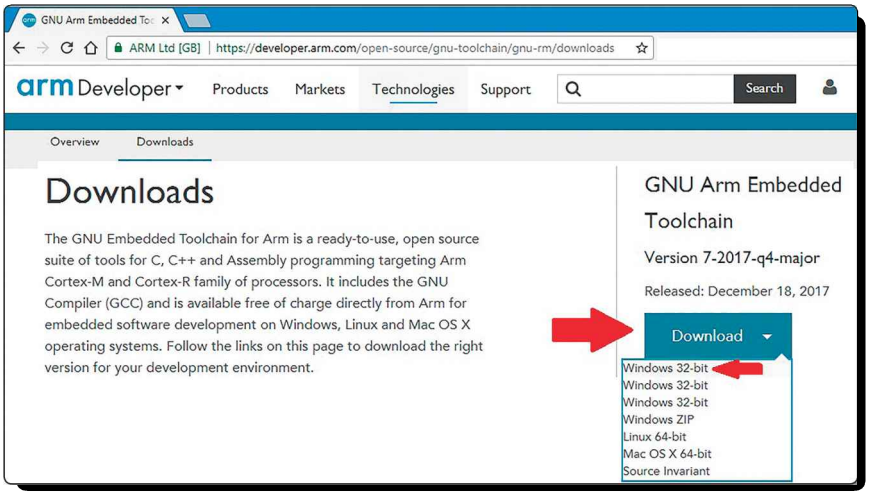


Figura 4.22

- **Paso 3.** Se nos descargará un fichero ejecutable: "*gcc-arm-none-eabi-7-2017-q4-major-win32.exe*" que debemos iniciar con permisos de Administrador y con el que comenzará su instalación.

- **Paso 4.** En una primera ventana al iniciar el proceso de instalación, nos solicitará que seleccionemos el idioma por defecto. Pulsaremos en la tecla "**Ok**", para continuar.



Figura 4.23

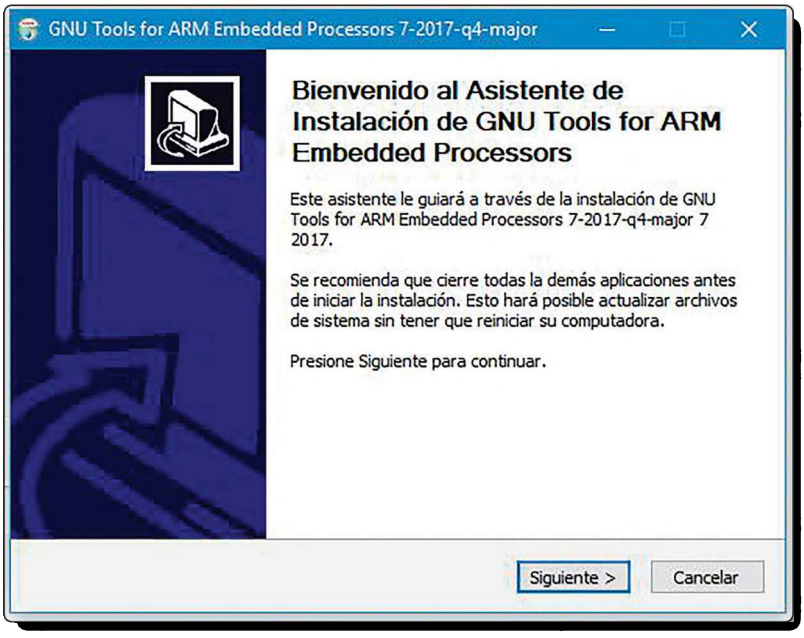


Figura 4.24

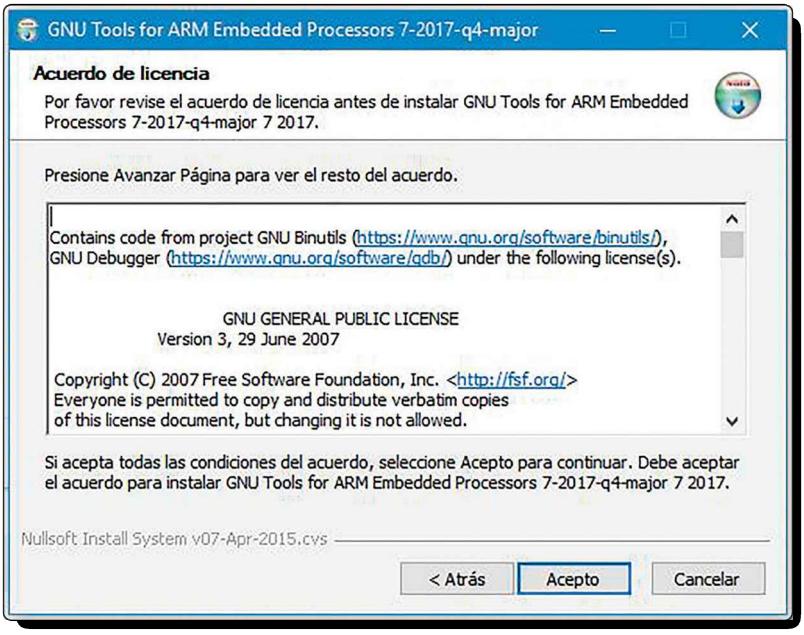


Figura 4.25

Paso 5. En la ventana de la Figura 4.25, debemos aceptar las condiciones de uso del programa y a continuación, se nos solicitará la ubicación donde instalaremos las librerías en nuestro ordenador.

Es conveniente que estas utilidades las instalemos también en una ubicación que nos resulte más conveniente (por el problema de los permisos de usuario del Windows). En nuestro caso, vamos a instalarlas en una carpeta dentro del propio directorio donde instalamos el entorno CooCox CoIDE; y que previamente deberemos haber creado. Por ejemplo en la siguiente dirección, dentro del directorio c:\Coocox\CoIDE:

C:\CooCox\CoIDE**GNU Tools ARM Embedded**

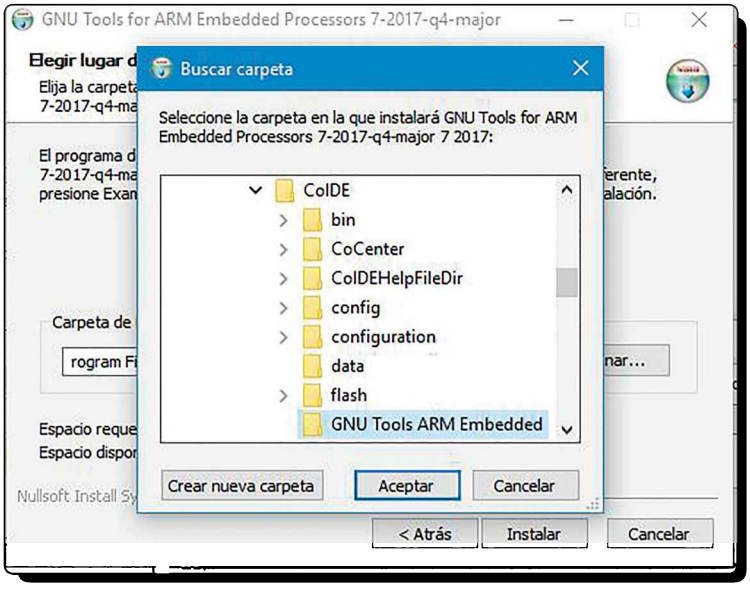


Figura 4.26

◤ **Paso 6**. Una vez creada la carpeta y seleccionada, seguiremos con la instalación, según la Figura 4.27.

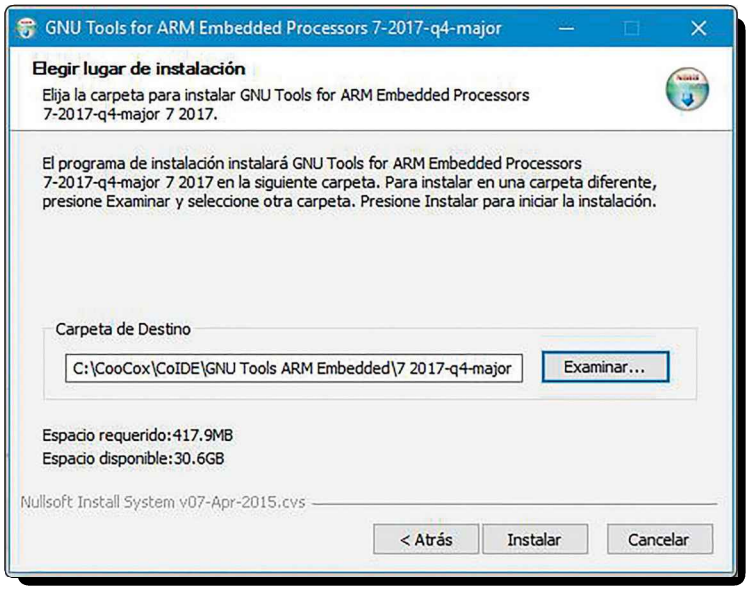


Figura 4.27

Y continuará con el proceso de instalación (Figuras 4-28 y 4-29)

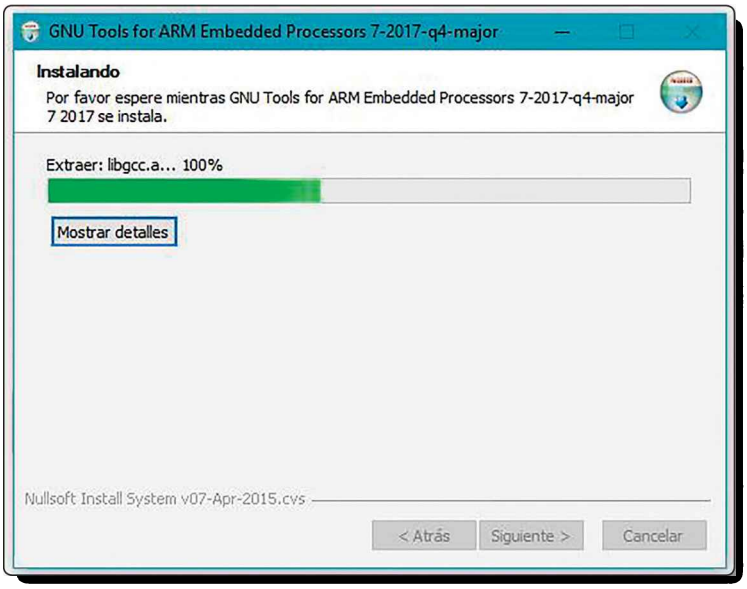


Figura 4.28

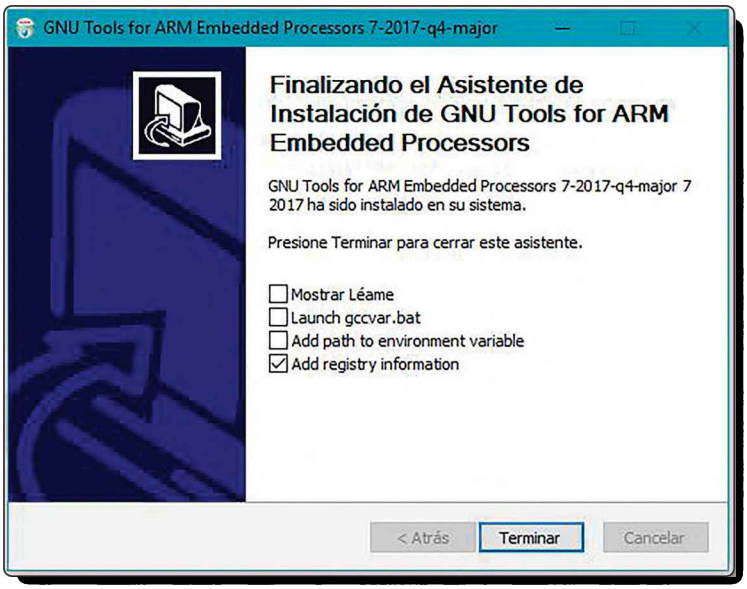


Figura 4.29

Paso 7. Tras realizarse la instalación, regresaremos a la ventana que teníamos abierta, o si la hubiéramos cerrado, iniciamos de nuevo el entorno CooCox CoIDE, para continuar con el proceso de creación de nuestro proyecto.

Si hemos cerrado el entorno CooCox IDE anteriormente, pulsamos en el botón "**Build**" – que señalamos en el Paso 8 -. Si todavía tenemos abierto la ventana con el mensaje de error, pulsamos en "**Ok**" y continuará en la Figura 4.30.

Procedemos de la siguiente forma:

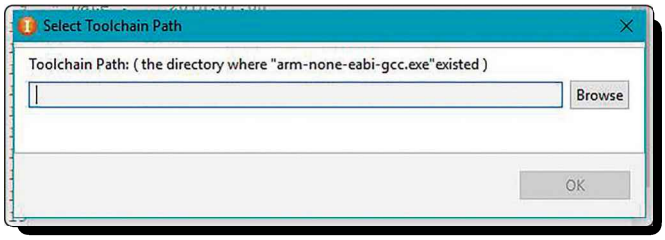


Figura 4.30

Vemos que se nos pide que indiquemos, dónde se encuentra el fichero "**arm-none-eabi-gcc.exe**", que instalamos anteriormente y que se encuentra en la siguiente dirección:

C:\CooCox\CoIDE\GNU Tools ARM Embedded\7 2017-q4-major\bin

Por lo que buscamos esa ubicación y la seleccionamos, como se indican en las Figuras 4-31 y 4-32.

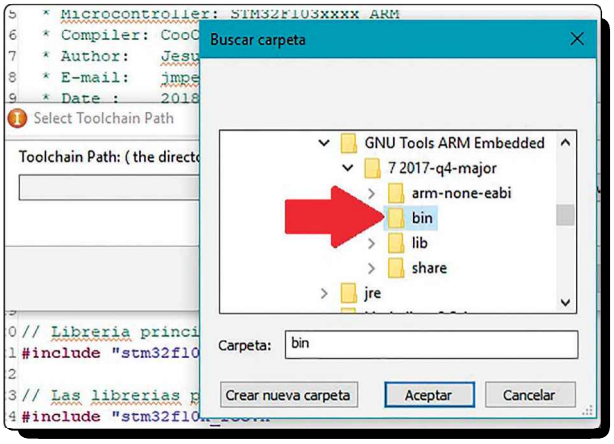


Figura 4.31

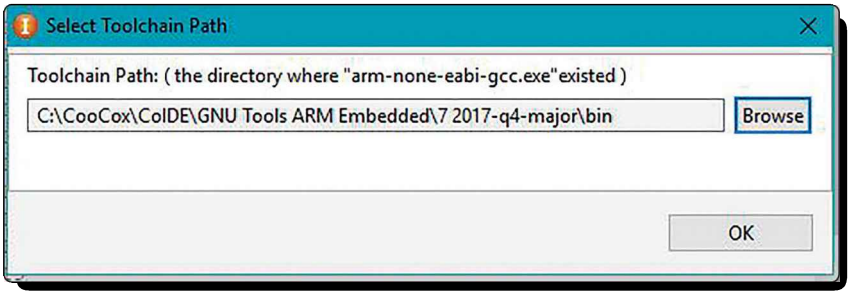


Figura 4.32

◤ **Paso 8.** Ahora ya comienza el proceso de compilación de nuestro código y, si todo ha ido correctamente, se nos abrirá una ventana en la parte inferior de nuestro entorno de desarrollo, tal como muestra la Figura 4.33.

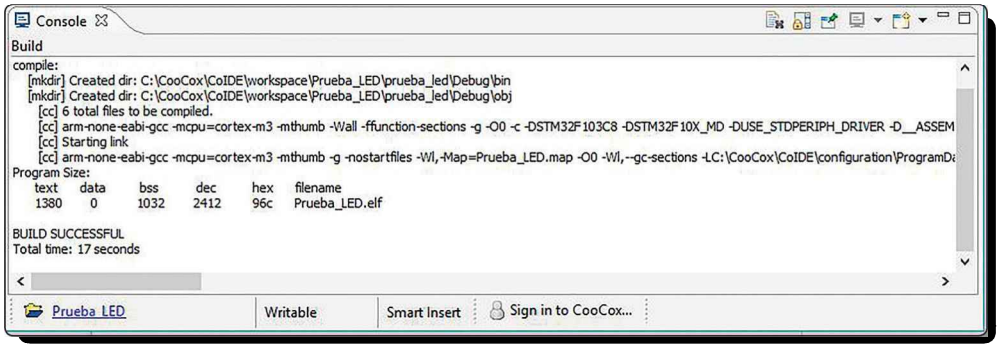


Figura 4.33

Correcto, ya hemos creado nuestro primer proyecto bajo **CooCox CoIDE**.

4.4 PROGRAMANDO LA PLACA CON EL ADAPTADOR ST-LINK EN COOCOX COIDE

Continuando con el proceso de creación de nuestro primer proyecto en el CooIDE, nuestro siguiente paso, será grabarlo en nuestra placa.

Para ello, si el proceso de compilación "**Buil**", en el paso 9, se ejecutó correctamente, lo siguiente será configurar el adaptador que emplearemos para programar nuestra placa.

Vimos al principio que existían dos métodos de programación de nuestro microcontrolador. Uno mediante un adaptador USB a RS232 y otro mediante una placa específica ST-Link/V2.

En nuestra primer ejemplo, utilizaremos el adaptador ST-Link.

- **Paso 10.** Lo primero será ir al menú "**View**" y seleccionar la opción de "**Configuration…**".

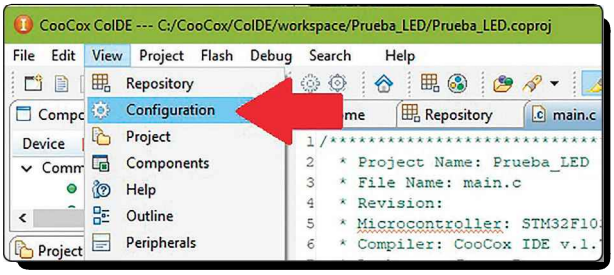


Figura 4.34

- **Paso 11.** En la ventana que se nos abrirá, seleccionamos la pestaña "**Debugger**" y comprobamos que, en el apartado "**Adapter**", aparece la siguiente configuración que se muestra en la Figura 4.35 y que será la de por defecto.

Adapter: ST-Link Port: SWD Max Clock(Hz): 1M

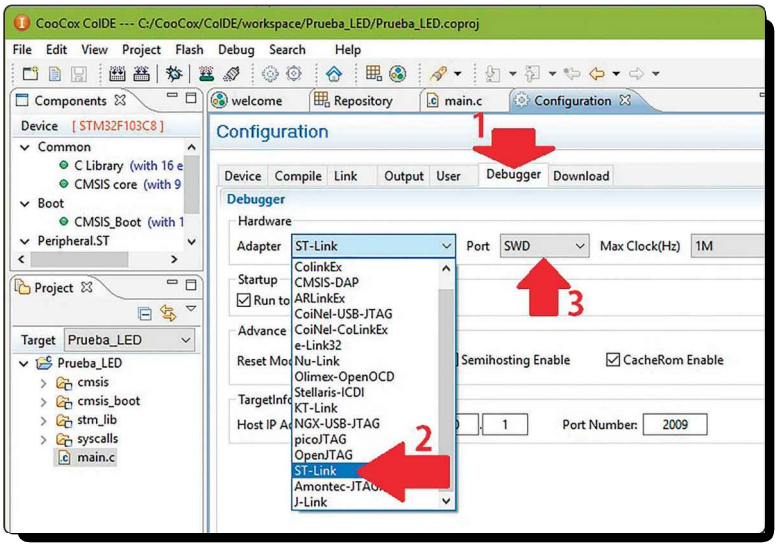


Figura 4.35

Una vez comprobado que la configuración es la correcta.

Cerramos la ventana "**Configuration**" y regresamos al editor con nuestro código de ejemplo.

Paso 12. Ahora pulsamos sobre el botón "**Download Code To Flash**"… que se indica en la Figura 4.36.

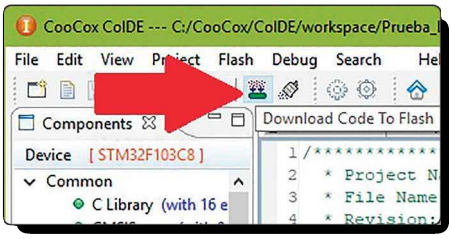


Figura 4.36

… y veremos cómo en la última línea de la pantalla, a la derecha, se muestra el mensaje que se está cargando el firmware en nuestra placa (Figura 4-37).

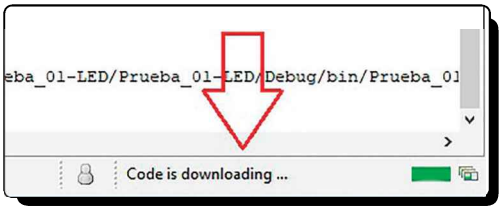


Figura 4.37

Además en la ventana inferior del entorno, en el lado izquierdo, en "**Console**", aparecerán los mensajes, indicando que se ha realizado correctamente. (Figura 4-38)

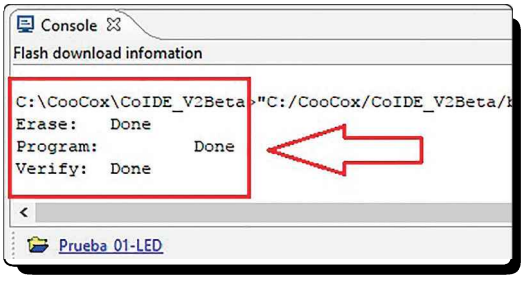


Figura 4.38

4.5 SOLUCIÓN DE ALGUNOS ERRORES EN LA PROGRAMACIÓN

Es posible que durante estos últimos pasos, nos hayan aparecido algunos mensajes de error y que no sepamos resolverlos; por lo que a continuación, explicamos los pasos necesarios para algunos de los errores más comunes que se pueden dar.

Al intentar abrir un proyecto que fue creado con otra versión del entorno **CooCox CoIDE**, se muestra el mensaje de error siguiente:

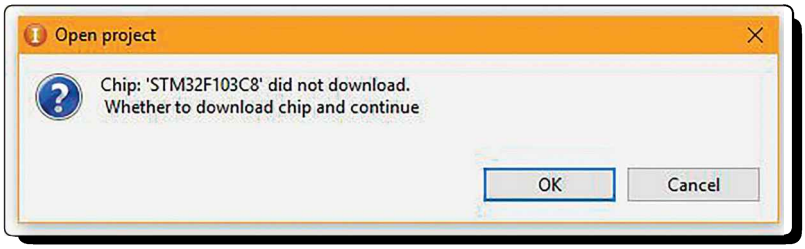


Figura 4.39

Y, tras pulsar "Ok" se vuelve a mostrar el siguiente mensaje de error:

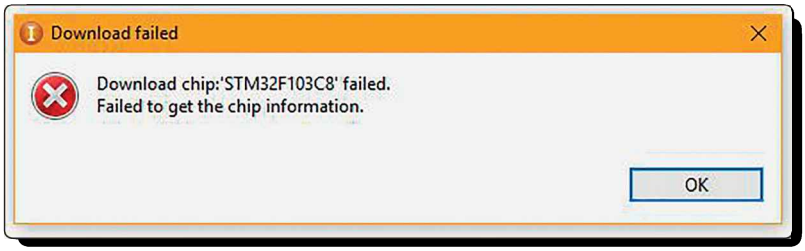


Figura 4.40

SOLUCIÓN

El problema es debido a que a partir de la versiones 1.7.x y la actual versión 2.0 no se nombran de la misma manera los chips microcontroladores dentro del fichero de configuración.

Para ello seguiremos los siguientes pasos:

1. Abrimos el archivo "***subproyecto*. Coproj**" con un editor de texto cualquiera, o **mejor guardaremos una copia primero de dicho archivo para poder utilizarlo de nuevo con la versión con la que fue creado inicialmente**.

2. Buscaremos la línea donde aparezca el siguiente texto "**chipName=”STM32F103C8**" o cualquier otro y cámbielo por "**STM32F103C8T6**" añadiendo el detalle de 'T6' al final del nombre.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Project version="2G - 1.7.8" name="Example_ADC">
  <Target name="Example_ADC" isCurrent="1">
    <Device manufacturerId="9" manufacturerName="ST" chipId="305" chipName="STM32F103C8" boardId="" boardName=""/>
    <BuildOption>
```

Figura 4.41

3. Guardamos de nuevo el fichero e intentamos abrirlo de nuevo, comprobaremos que ahora sí se abre el proyecto.

4.6 OPCIONES DE DEPURACIÓN CON EL COOCOX COIDE

Mediante la opción de depuración que posee el entorno CoIDE, es posible comprobar y depurar nuestros programas, aunque el sistema esté constante y automáticamente comprobando la sintaxis de nuestras instrucciones. El sistema también nos permitirá, mediante herramientas especiales, depurar nuestros proyectos incluso antes de cargarlo en la placa o circuito.

Cuando tengamos desarrollado un proyecto, que ya hayamos compilado "Build" y haya ido todo bien, si pulsamos en la tecla "**Debug**" o pulsando las teclas "**Ctrl+ F5**" entraremos en el modo depuración del sistema.

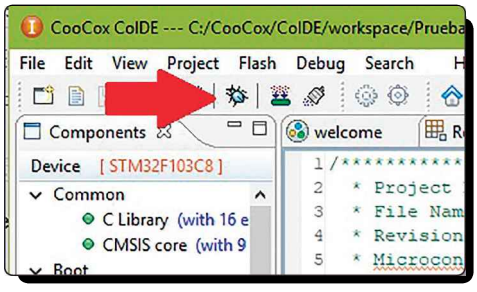


Figura 4.42

Puede que al intentar proceder, recibamos el siguiente error en la ventana inferior:

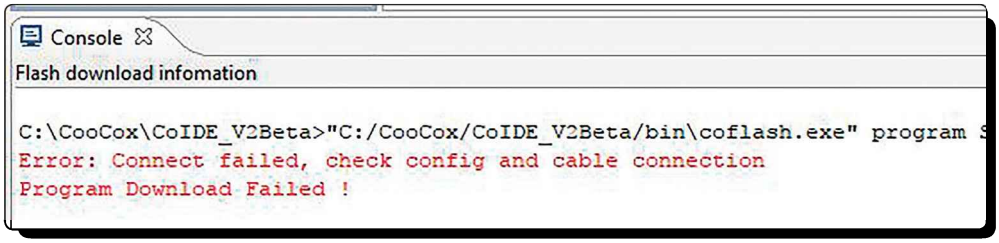


Figura 4.43

Deberemos tener en cuenta, que el proceso de depuración, solo funcionará si tenemos conectada la placa con nuestro ordenador con el adaptador ST-Link.

Por lo que al pulsar en "**Debug**", se nos abrirá la siguiente pantalla para preparar el sistema en el modo depuración.

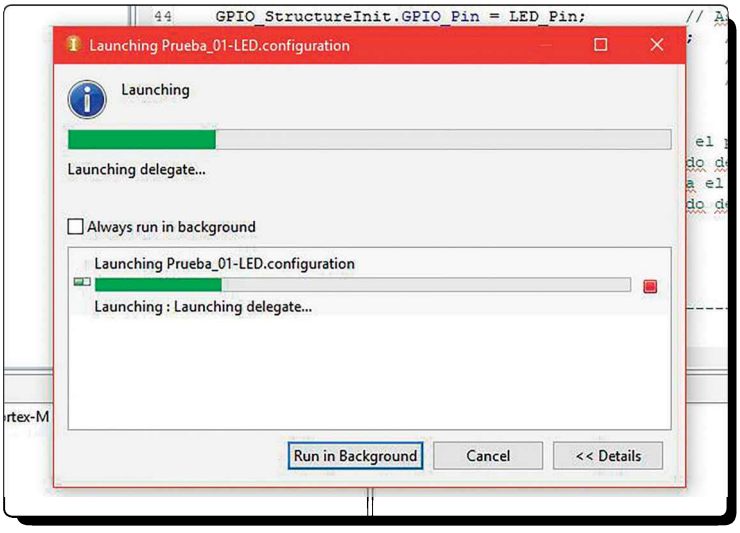


Figura 4.44

Se cambia el contenido normal del escritorio del entorno de trabajo, apareciendo otras ventanas nuevas como se muestran en la Figura 4.45.

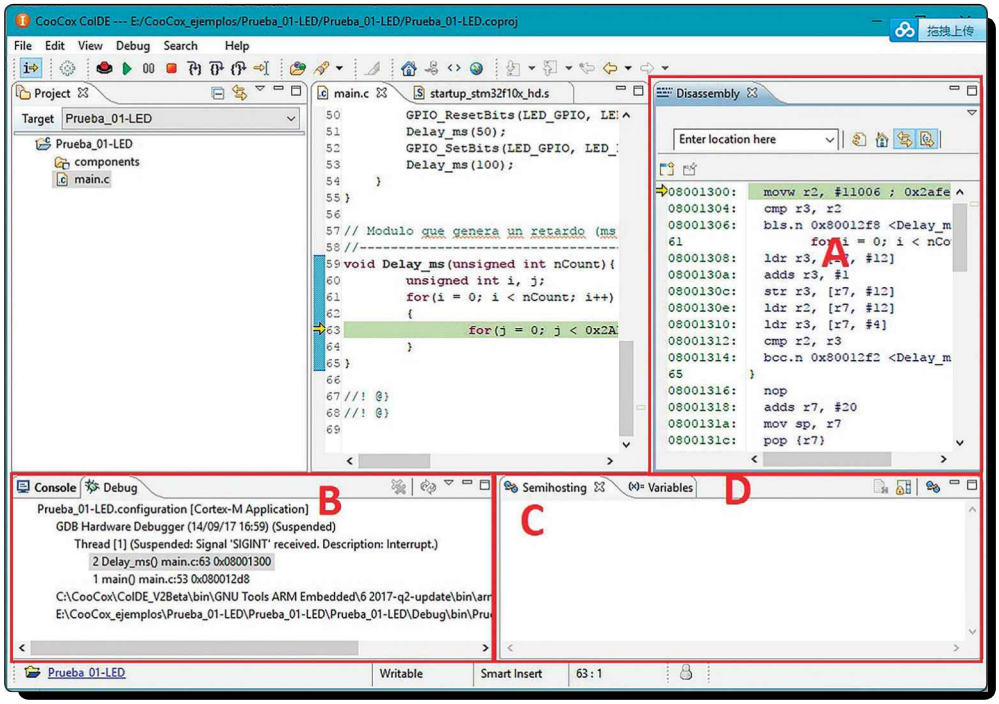


Figura 4.45

En la parte superior, en los botones de comandos, aparecerán unos nuevos iconos:

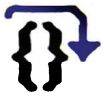
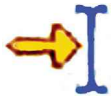


Figura 4.46



Figura 4.47

Inicia el funcionamiento del modo depuración en el que se ejecutarán una a una las instrucciones y comandos de nuestro programa; a la vez que se irán ejecutando en nuestra placa. (Tecla F5)

 **Figura 4.48**	Detiene el programa y produce un reset en el microcontrolador.
 **Figura 4.49**	Detiene la ejecución. (Tecla F9).
 **Figura 4.50**	Finaliza y cierra el modo depuración. (Teclas Ctrl+F5).
 **Figura 4.51**	Configura que el proceso se ejecute de una instrucción a la siguiente en nuestro programa.
 **Figura 4.52**	Genera que se repitan los pasos solo en el comando que se está señalando en ese momento.
 **Figura 4.53**	Produce la ejecución del programa paso a paso de los comandos externos al área señalada.
 **Figura 4.54**	Produce la ejecución del programa paso a paso de aquellos comandos a continuación del área señalada.
 **Figura 4.55**	Ejecuta solo la línea que tenemos seleccionada.

Pantallas del nuevo escritorio de depuración

Cuando ejecutemos el modo depuración, podemos observar en nuestro editor, en el fichero seleccionado "***main.c***", que se remarcan las lineas con un fondo verde y con una flecha amarilla a la izquierda de la línea, indicando como un puntero, el lugar en donde se encuentra la ejecución del programa.

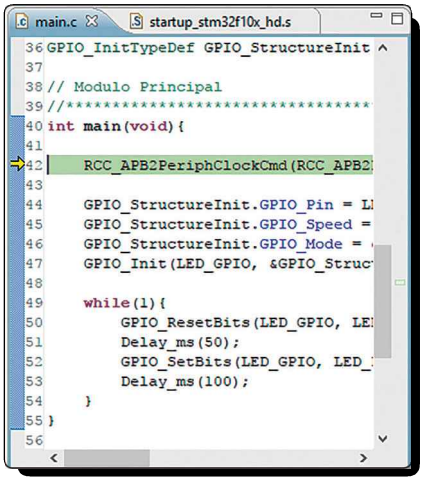


Figura 4.47

En la ventana de la derecha, aparece la ventana "**Disassembly**". Muestra nuestro programa convertido a líneas de código máquina. E igualmente, se producirá el paso de línea a línea durante el proceso de ejecución de nuestro programa.

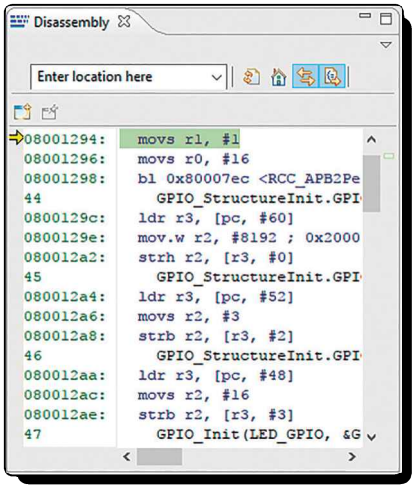


Figura 4.48

Mediante la selección de diferentes opciones en el menú "**View**", podremos mostrar otras ventanas en la parte baja, a la derecha de la pantalla.

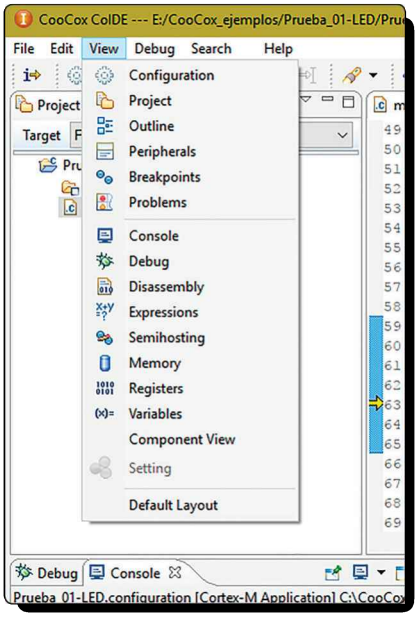


Figura 4.49

Se muestra por ejemplo:

El contenido de la memoria del microcontrolador al seleccionar "**Memory**":

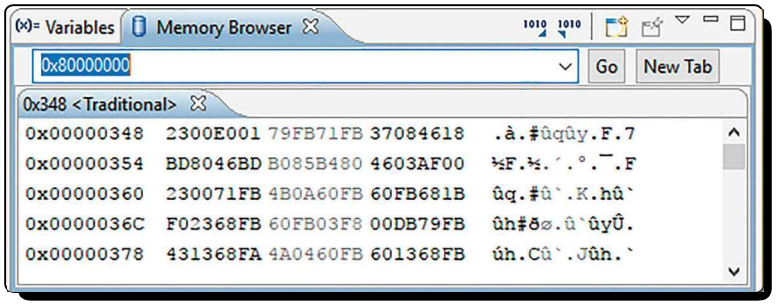


Figura 4.50

Los valores que tienen las variables que hemos creado en nuestro programa, al seleccionar "**Variables**"

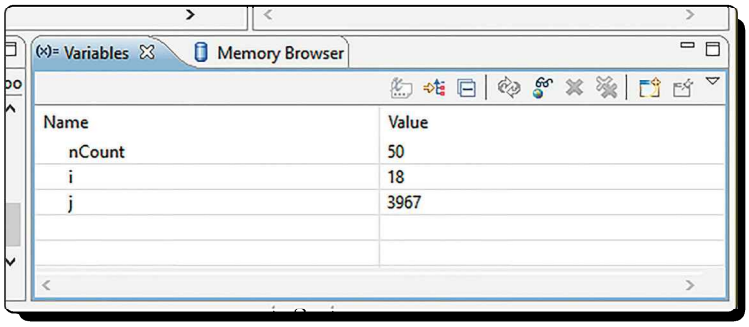


Figura 4.51

Además se podrán comprobar los valores que pueden tener los distintos periféricos y elementos de nuestro programa, no solo durante el proceso de ejecución, sino cómo quedan al final de un determinado intervalo o pausa.

También es posible realizar determinadas funciones con las variables. Por ejemplo, al seleccionar con el botón derecho de nuestro ratón en cualquiera de las variables, se desplegará un submenú, en el que por ejemplo al escoger la opción "**Change Value…**" se puede comprobar qué pasaría si una determinada variable tuviera otro valor en un momento dado.

O mediante la opción "**Cast To Type…**" cambiar el tipo de variable asignado en nuestro programa.

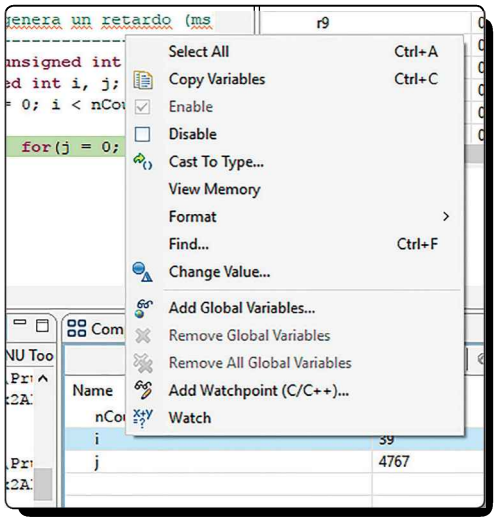


Figura 4.52

De esta forma se pueden realizar comprobaciones y pruebas en nuestra programación cambiando los valores y tipos de variable del proyecto original.

Con todos estos pasos, hemos aprendido todo lo necesario para la creación de cualquier proyecto mediante el entorno de desarrollo CooCox CoIDE, variando solo el tipo de microcontrolador en los primeros pasos descritos y añadiendo alguna librería más de las necesarias según los periféricos y módulos que necesitemos en nuestros nuevos proyectos.

5

PRIMEROS PASOS CON KEIL MDK ARM



Figura 5.1



Figura 5.2

El **Keil® MDK** es, posiblemente, la solución más completa para el desarrollo de aplicaciones en sistemas basados en microcontroladores ARM; pero, desgraciadamente, es una solución de pago, no obstante, la versión de demostración es accesible desde la página del fabricante.

Su programación se basa en el lenguaje C/C++ estándar, aunque permite también trabajar directamente con lenguaje ensamblador.

A continuación, pasamos a describir los primeros pasos para su instalación y puesta en marcha.

5.1 INSTALACIÓN

Primero nos descargaremos la versión de evaluación **Keil MDK** desde la página oficial en Internet. La última versión en el momento de creación de nuestro libro es la versión 5.25.

http://www2.keil.com/mdk5

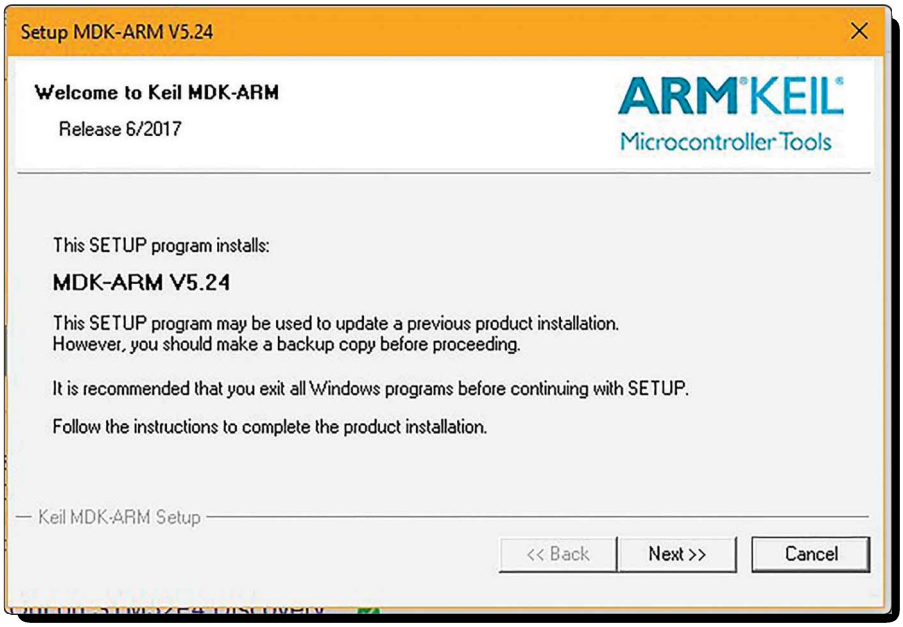


Figura 5.3

En las siguientes pantallas se nos pedirá que seleccionemos el directorio donde se instalará el programa. Seleccionamos en "**C:\Keil\Keil_v5**" y los datos del nombre para el registro del programa.

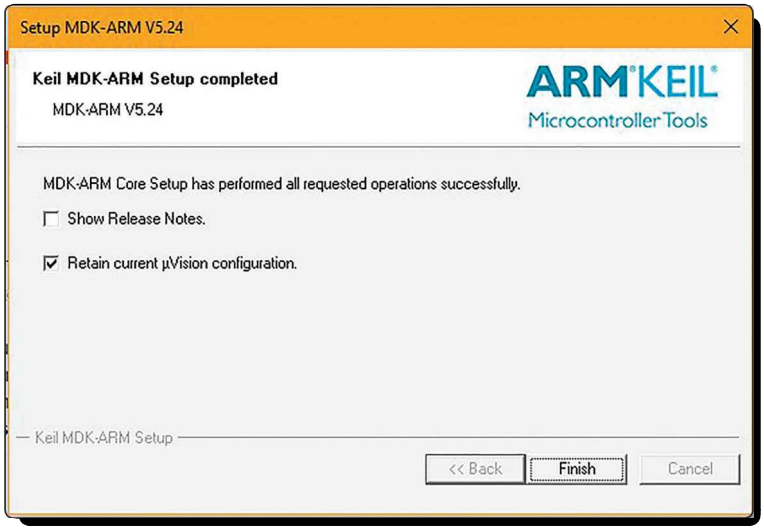


Figura 5.4

Comenzará el proceso de instalación.

Tras la última pantalla, se habrá instalado la aplicación en nuestro equipo.

A continuación, Figura 5.5, se nos pedirá que instalemos las librerías necesarias para nuestro microcontrolador.

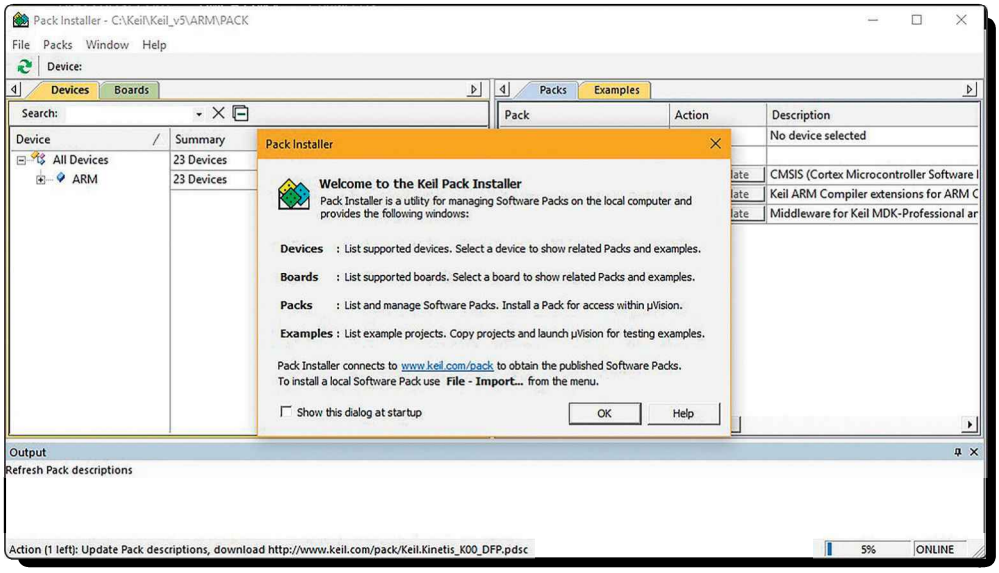


Figura 5.5

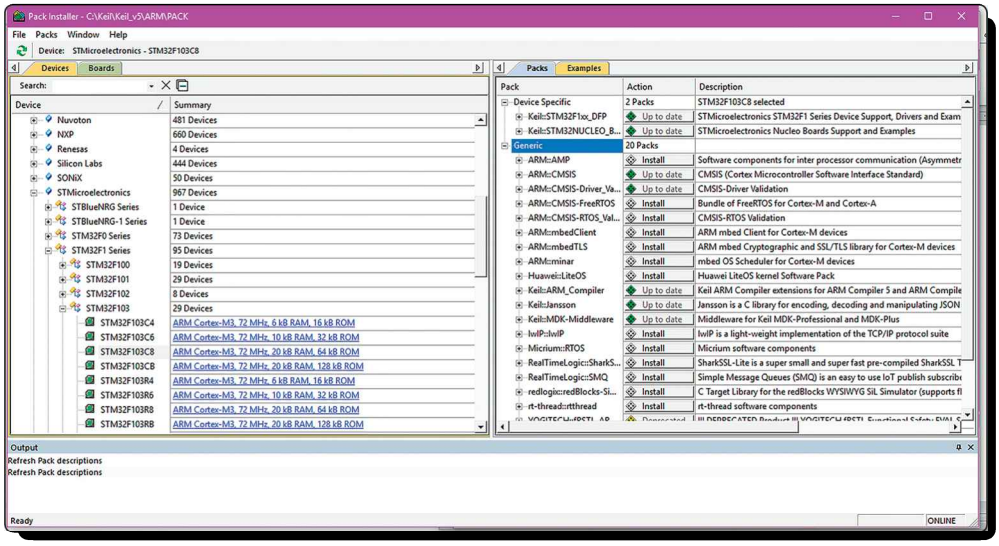


Figura 5.6

En la pantalla de la Figura 5.6, se nos muestran dos ventanas. En la ventana de la izquierda buscamos y seleccionamos los micros "**STMicroelectronics"**, luego la familia **"STM32F1 Series**" y a continuación nuestro microcontrolador, el **"STM32F101C8**".

Después en la ventana de la derecha, en el apartado de "**Pack",** buscamos y seleccionamos la librería "**Keil::STM32F1xx_DFP**" y pulsamos en "**Install**".

Abandonamos la pantalla, seleccionando **Exit** en el menú File.

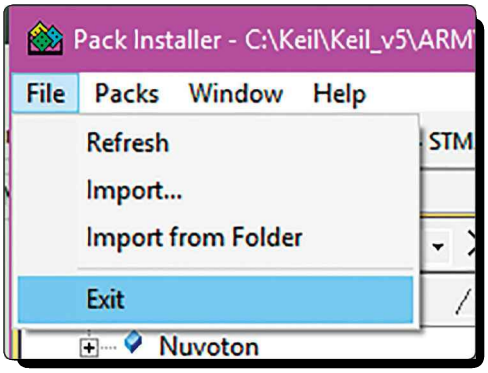


Figura 5.7

Con todo ello, ya tendremos instalada y configurada la aplicación y en nuestro escritorio se nos habrá instalado el siguiente icono:



Figura 5.8

Ahora al iniciar el programa Keil, se mostrara la pantalla de la Figura 5.9, con el entorno de trabajo.

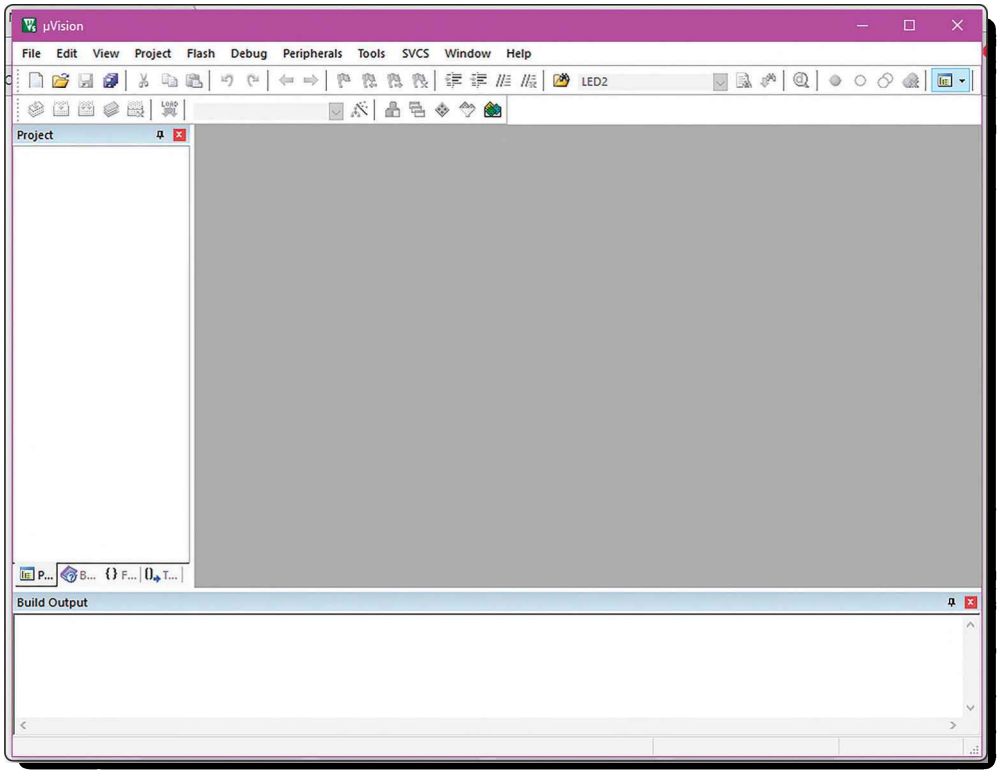


Figura 5.9

5.2 CREAR UN PRIMER PROYECTO CON KEIL

A continuación, procedemos a explicar los pasos necesarios para crear un proyecto con el entorno de trabajo del Keil MDK.

Paso 1. Crear un nuevo proyecto.

Tras iniciar el programa, seleccionamos la opción "**New µVision Project**" en el menú **"Project".**

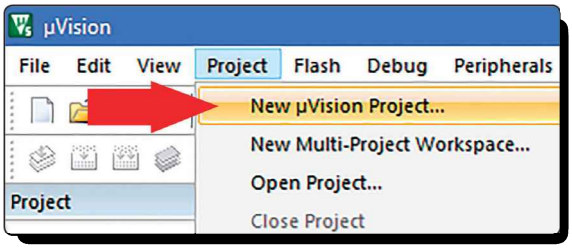


Figura 5.10

En la pantalla que se nos abrirá, como la de la Figura 5.11, debemos crear la carpeta que contendrá nuestro proyecto; que en nuestro ejemplo la nombraremos como "**Prueba_01**".

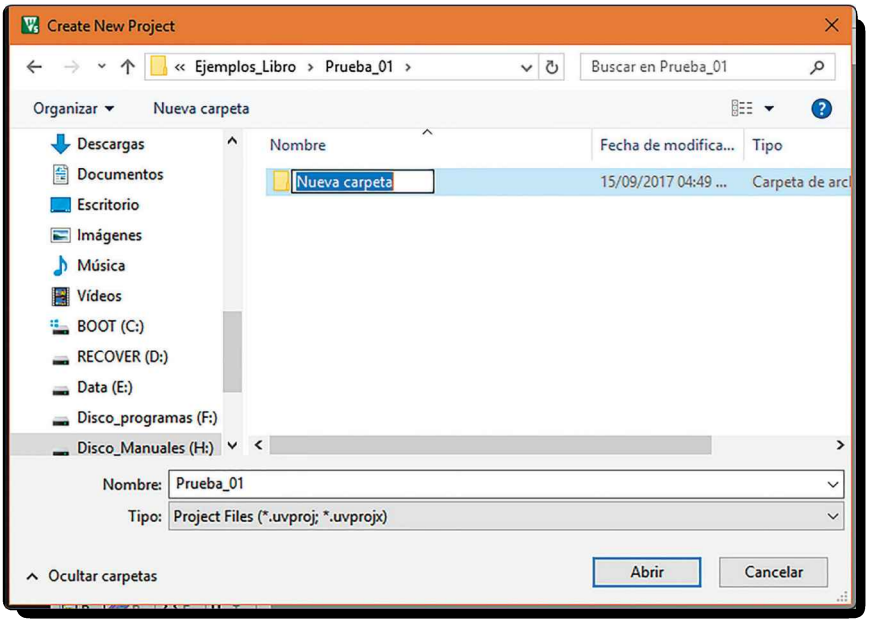


Figura 5.11

Previamente, creamos en la propia ventana la carpeta que después debemos seleccionar para que nos aparezca en la barra de "**Nombre**"; creándose así el nombre que tendrá nuestro proyecto. Figura 5.11.

A continuación, según la Figura 5.12, abrimos la carpeta para ahora crear dentro nuestro proyecto.

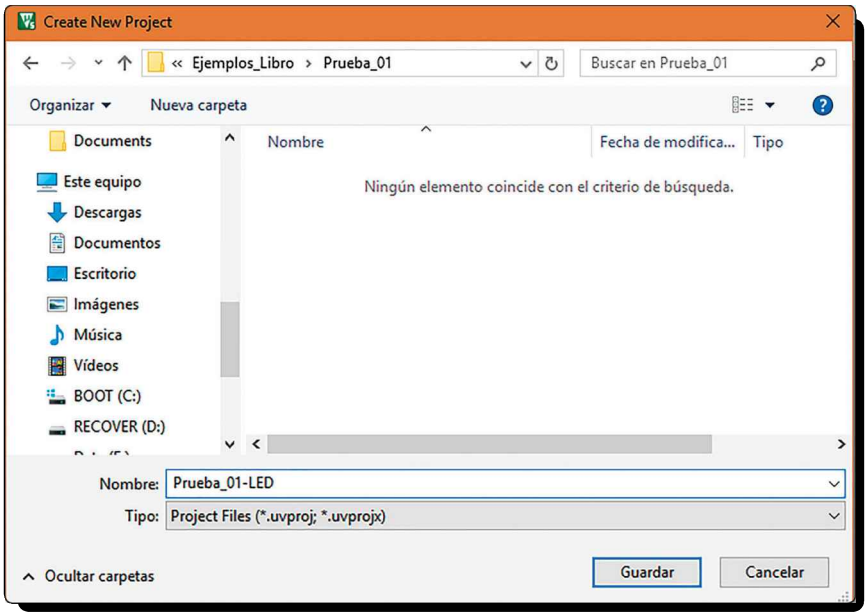


Figura 5.12

Paso 2. Configurar el microprocesador del nuevo proyecto.

La siguiente pantalla, la Figura 5.13, nos pedirá que indiquemos qué microprocesador vamos a utilizar para nuestro proyecto y lo seleccionaremos de una lista:

Seleccionamos primero el fabricante: "**STMElectronics"** luego la familia: **"STM32F1 Series"** y por último el microcontrolador: "**STM32F103C8"** Figura 5.14:

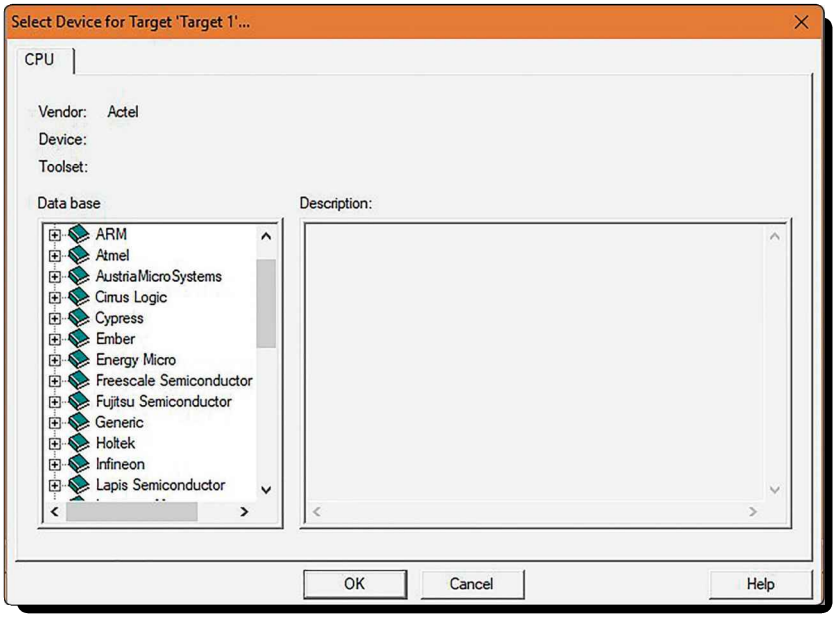


Figura 5.13

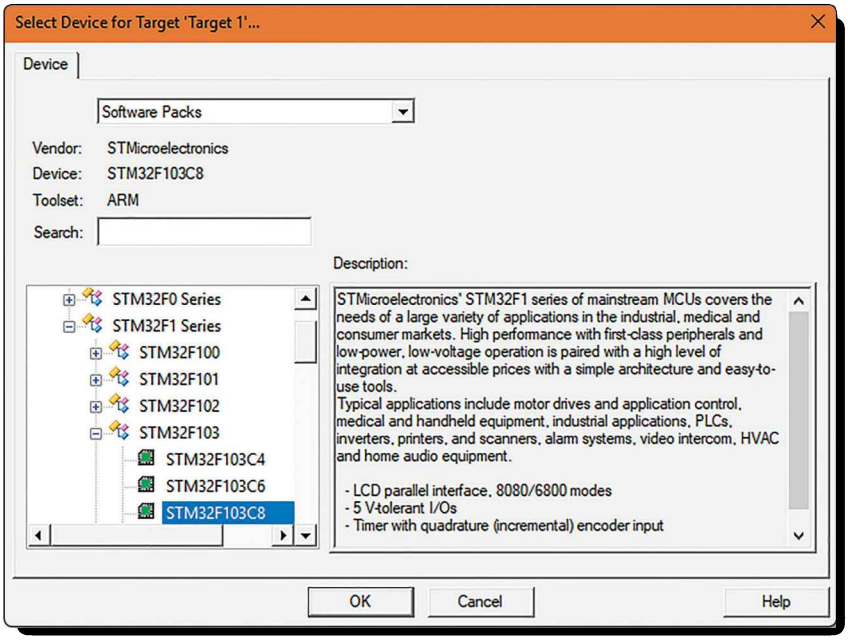


Figura 5.14

A continuación se nos abrirá una pantalla como la de la Figura 5.15, donde debemos seleccionar las librerías que vamos a utilizar en nuestro proyecto.

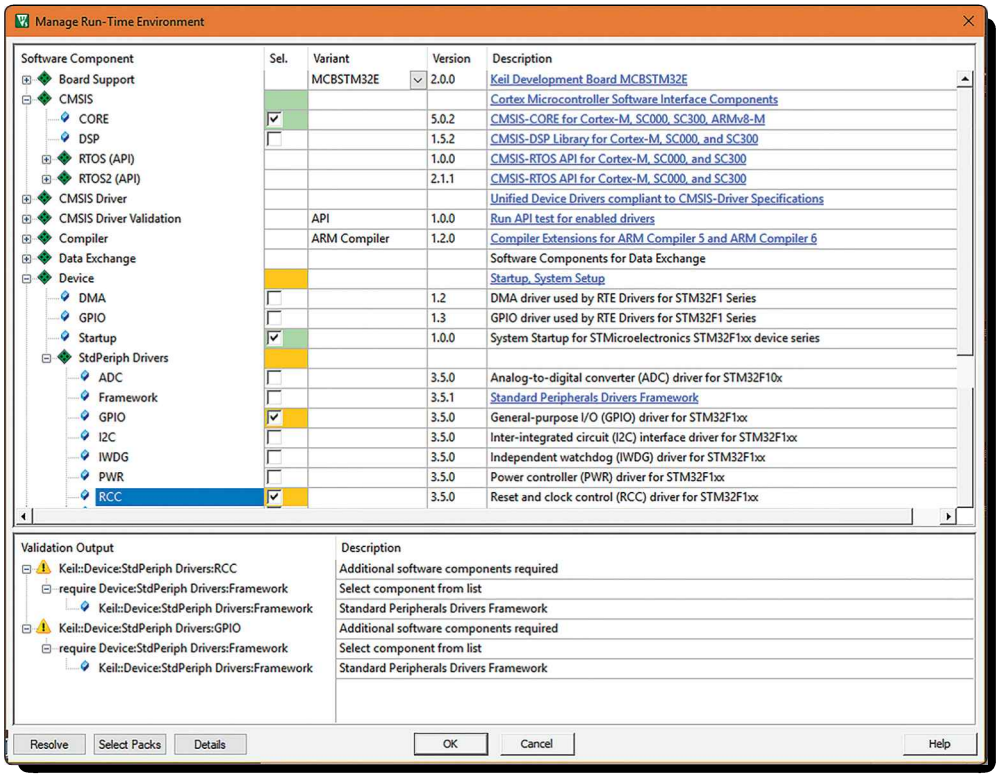


Figura 5.15

Buscamos y seleccionamos el apartado "**Device".** En su interior, seleccionamos en el apartado **"SedPeriph Drivers"** las librerías para **"GPIO"** y "**RCC**". Durante este proceso, se nos mostrarán unos cuadrados en amarillo, en la segunda columna, como en la Figura 5.15, indicando que las librerías todavía no están cargadas para ser añadidas a nuestro proyecto, Si pulsamos en el botón "**Resolve**" de la parte inferior de la pantalla, se pondrán en verde, indicando que han sido cargadas. Para terminar pulsamos "**Ok**".

Las librerías básicas para todo proyecto que debemos seleccionar siempre son las que se indican en la Figura 5.16:

Apartado	Librería
CMSIS	Core
Device	Startup
StdPeriph Drivers	**GPIO** – Librería control de puertos **RCC** – Librería control timers

Figura 5.16

Como podemos ver en la Figura 5.17, en el apartado "**StdPeriph Drivers**", tendremos una lista de las librerías necesarias para el control de los periféricos del microcontrolador. Estas las cargaremos en el momento de crear el proyecto o durante el proceso de creación, cuando sepamos cuáles vamos a necesitar.

Software Component	Description
DMA	DMA driver used by RTE Drivers for STM32F1 Series
GPIO	GPIO driver used by RTE Drivers for STM32F1 Series
Startup	System Startup for STMicroelectronics STM32F1xx device series
StdPeriph Drivers	
ADC	Analog-to-digital converter (ADC) driver for STM32F10x
BKP	Backup registers (BKP) driver for STM32F10x
CAN	Controller area network (CAN) driver for STM32F1xx
CEC	Consumer electronics control controller (CEC) driver for STM32F1xx
CRC	CRC calculation unit (CRC) driver for STM32F1xx
DAC	Digital-to-analog converter (DAC) driver for STM32F1xx
DBGMCU	MCU debug component (DBGMCU) driver for STM32F1xx
DMA	DMA controller (DMA) driver for STM32F1xx
EXTI	External interrupt/event controller (EXTI) driver for STM32F1xx
FSMC	Flexible Static Memory Controller (FSMC) driver for STM32F11x
Flash	Embedded Flash memory driver for STM32F1xx
Framework	Standard Peripherals Drivers Framework
GPIO	General-purpose I/O (GPIO) driver for STM32F1xx
I2C	Inter-integrated circuit (I2C) interface driver for STM32F1xx
IWDG	Independent watchdog (IWDG) driver for STM32F1xx
PWR	Power controller (PWR) driver for STM32F1xx
RCC	Reset and clock control (RCC) driver for STM32F1xx
RTC	Real-time clock (RTC) driver for STM32F1xx
SDIO	Secure digital (SDIO) interface driver for STM32F1xx
SPI	Serial peripheral interface (SPI) driver for STM32F1xx
TIM	Timers (TIM) driver for STM32F1xx
USART	Universal synchronous asynchronous receiver transmitter (USART) driver for STM32F1xx
WWDG	Window watchdog (WWDG) driver for STM32F1xx

Figura 5.17

Paso 3. Crear nuestro código.

En el entorno de trabajo, en la ventana de la izquierda "**Project**", se nos habrán creado automáticamente una estructura en árbol de los componentes de nuestro proyecto, tal y como aparece en la Figura 5.18.

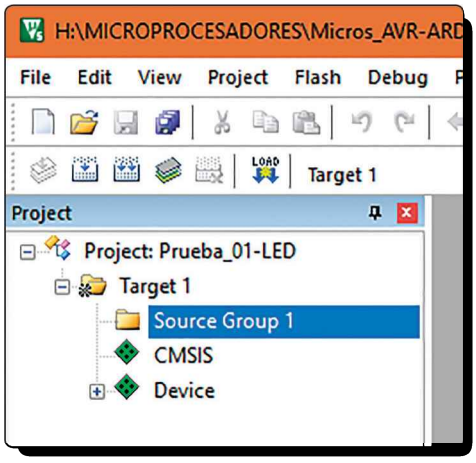


Figura 5.18

Pulsamos, sobre el texto "**Source Group1**" y lo renombramos como "**User Aplication**" como se ve en la Figura 5.19.

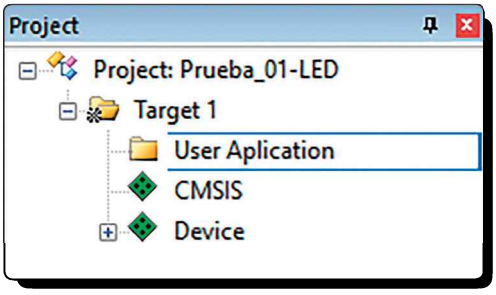


Figura 5.19

El siguiente paso será crear nuestro fichero que contendrá el código en C++. Para ello pulsamos con el botón derecho de nuestro ratón sobre "**User Aplication**" en la ventana **Project** donde se encuentra nuestro proyecto; y en el menú desplegado, seleccionamos "**Add New Item to Group** "**User Aplication…**". Figura 5.20.

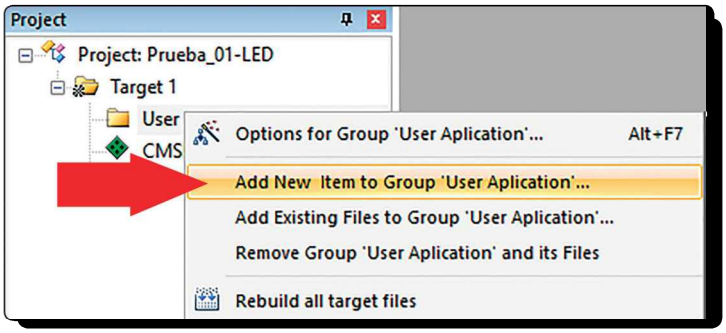


Figura 5.20

A continuación, en la ventana de la Figura 5.21, seleccionamos el tipo de archivo a crear "**C File…**" e introducimos el nombre que le vamos a dar y que lo ponemos en el apartado "Name", que en nuestro caso le pondremos "**main.c**".

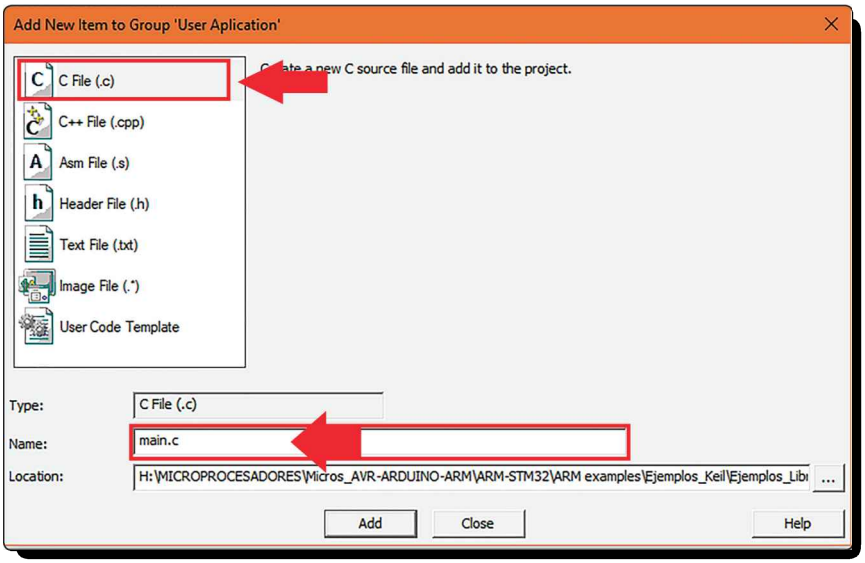


Figura 5.21

Tras pulsar en el botón "**Add**", no solo se nos creará el fichero, sino también se nos abrirá un editor con el fichero creado en blanco para que escribamos nuestro programa.

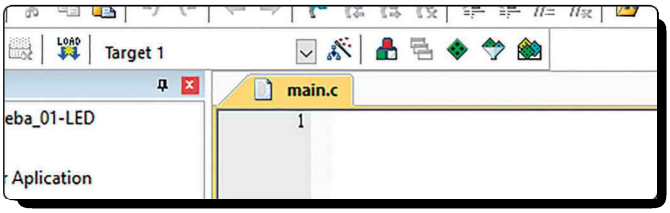


Figura 5.22

En él escribiremos inicialmente, el siguiente código de prueba:

```
#include "stm32f10x.h"

int main(void)
{

}

```

Figura 5.23

Pulsamos en el botón de "**Rebuild**" compilación, como se muestra en la Figura 5.24, y así comprobaremos nuestra configuración hasta ahora.

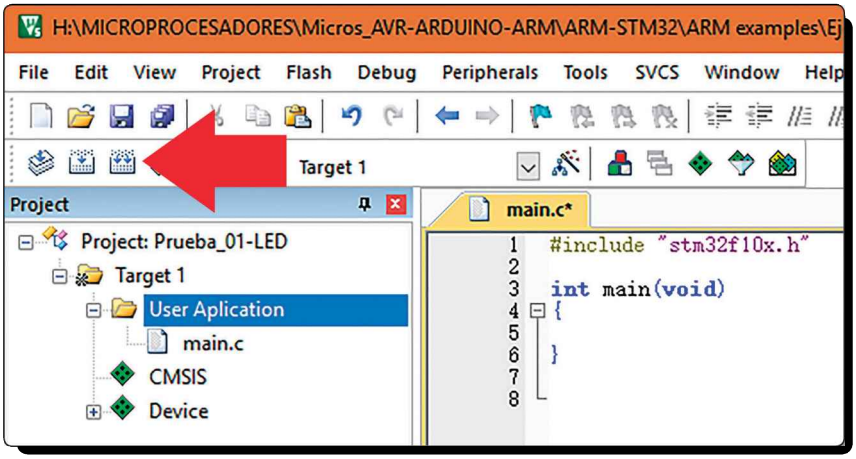


Figura 5.24

Y, para comprobar que hemos configurado todo correctamente hasta este momento, se nos mostrará la siguiente pantalla en "**Build Output**", en la parte inferior del entorno de trabajo. Figura 5.25.

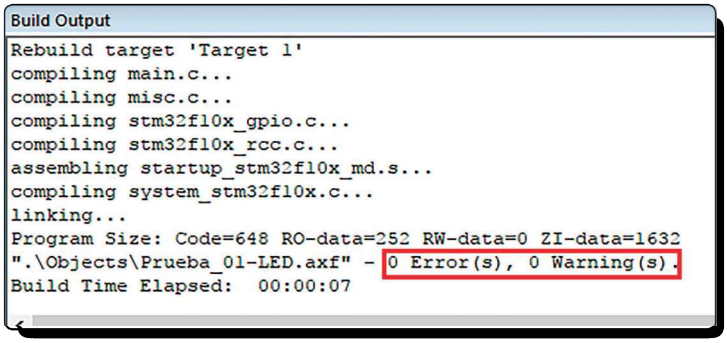


Figura 5.25

Donde, señalado con un cuadro en rojo, se mostrará si la compilación ha encontrado algún error.

A continuación, copiamos el resto del código que se muestra en la Figura 5.26 para encender el LED de prueba de nuestra placa.

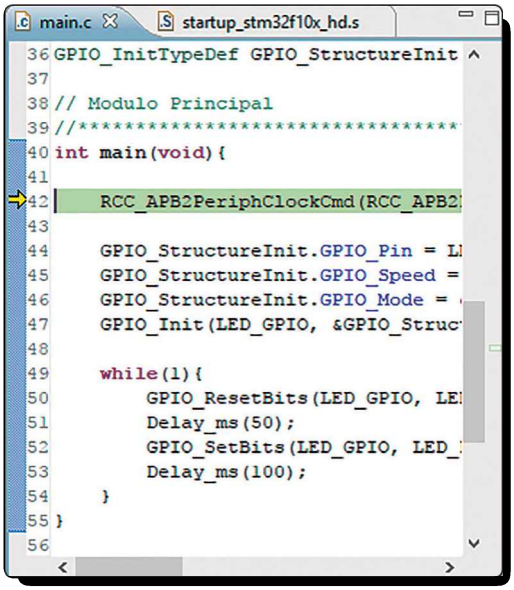


Figura 5.26

◤ **Paso 4.** Últimos parámetros de nuestro proyecto.

Pulsamos en el botón de configuración, Figura 5.27, para seguir configurando los parámetros de uso para nuestro proyecto.

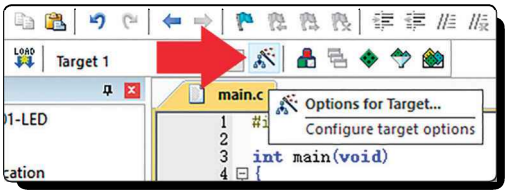


Figura 5.27

Establecemos, en la pestaña "**Target**" de la Figura 5.28, la frecuencia del reloj de nuestra placa, que es de **8 MHz** y corresponde a la velocidad del oscilador que posee nuestra placa de pruebas.

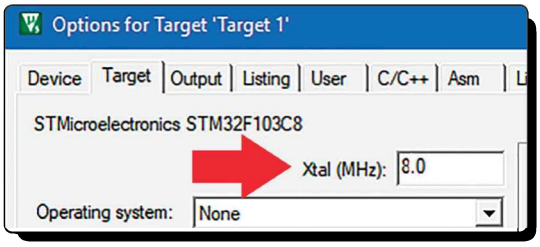


Figura 5.28

Establecemos, en la pestaña "**Output**", de la Figura 5.29, que se cree el fichero **.hex** necesario para grabar nuestra placa, dejando activa solo estas dos opciones:

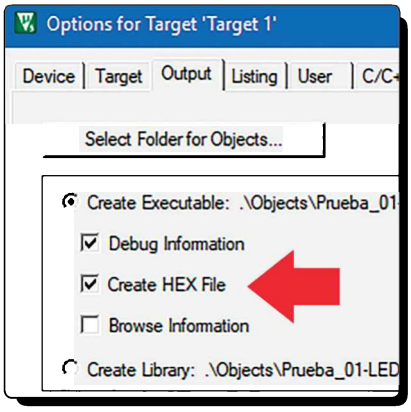


Figura 5.29

En un siguiente paso establecemos el método o adaptador que utilizamos para la grabación de nuestra placa.

En el Keil, para utilizar uno u otro adaptador para programar nuestra placa, será necesario configurarlo de acuerdo a los pasos que se explican en el capítulo 6 de este libro, en PROGRAMANDO NUESTRA PLACA.

- **Paso 5.** Compilar nuestro proyecto.

  Volveremos a pulsar el botón de compilación de la Figura 5.24 anterior y si todo ha ido correctamente se nos mostrarán los mensajes como los de la Figura 5.30:

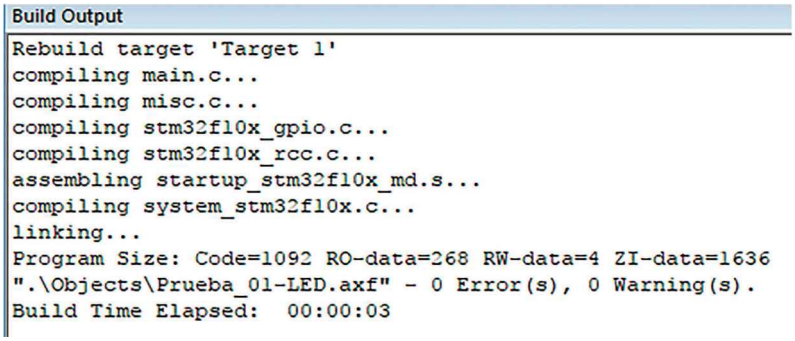


Figura 5.30

Hasta aquí, hemos visto y dado los pasos para crear una estructura mínima que nos sirva de base para cualquier otro proyecto nuevo.

Podemos guardar este proyecto en una carpeta a parte en nuestro ordenador, y utilizarlo como estructura de base para copiarlo en otro nuevo proyecto, teniendo así ya realizados todos los pasos previos necesarios en la creación de nuestros nuevos proyectos, sin tener que repetir los pasos ya realizados. Bastará con continuar los siguientes pasos y reutilizarlo para otros proyectos:

1. Crearemos un directorio o carpeta con el nombre del nuevo proyecto.
2. Copiamos el contenido completo del interior del proyecto anterior base, según se muestra en la Figura 5.31.

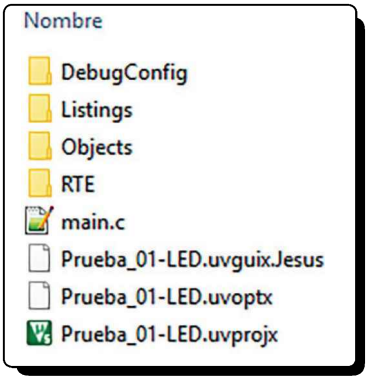


Figura 5.31

3. Eliminaremos el fichero "main.c" del primer proyecto o crearemos un fichero en blanco donde podemos copiar las líneas de código de nuestro nuevo proyecto.

4. Renombraremos los tres ficheros del proyecto anterior con el nombre del nuevo proyecto. Es decir, el nombre de los ficheros que contiene el texto "Prueba_01-LED.xxxx.xx", lo cambiaremos por el nombre del nuevo proyecto, por ejemplo "Proyecto_02.xxx.xx".

5. Cuando iniciemos el entorno de trabajo del Keil y abramos el proyecto nuevo, seleccionando el fichero del nuevo proyecto con la extensión ".uvprojx", vemos cómo se abrirá, ya configurado, el nuevo proyecto.

6

PROGRAMANDO NUESTRA PLACA

Como ya indicamos anteriormente, nuestra placa puede ser programada mediante un adaptador USB a RS232 o con un adaptador ST-Link. A continuación explicaremos la configuración y utilización de estos dos adaptadores.

6.1 PROGRAMANDO CON EL ADAPTADOR USB A RS232

6.1.1 INSTALACIÓN DEL DRIVER DEL ADAPTADOR USB A RS232 CH340

Existen en el mercado varios tipos de adaptadores USB a RS232 con diferentes chip controladores. Uno de los más comunes en las placas, es el chip CH340 fabricado por la empresa china Nanjing QinHeng Electronics Co., Ltd. (*http://www.wch.cn/*), del que explicamos a continuación su utilización para programar nuestra placa.

Este chip es reconocible por la mayoría de sistemas Windows y su driver se instalará automáticamente.

Para utilizarlo como programador, conectaremos el adaptador a nuestra placa de pruebas de la manera que se muestra en la Figura 6.1; donde la salida Tx del adaptador, deberá ser conectada al pin PA9 y la entrada Rx al pin PA10, los cuales se corresponden con los pines del puerto USART1 de nuestro microcontrolador.

Cuidado con el voltaje seleccionado en el adaptador ***el pin de 5Vcc sigue dando voltios*** **ya que debemos configurar el adaptador para 3.3 Vcc.**

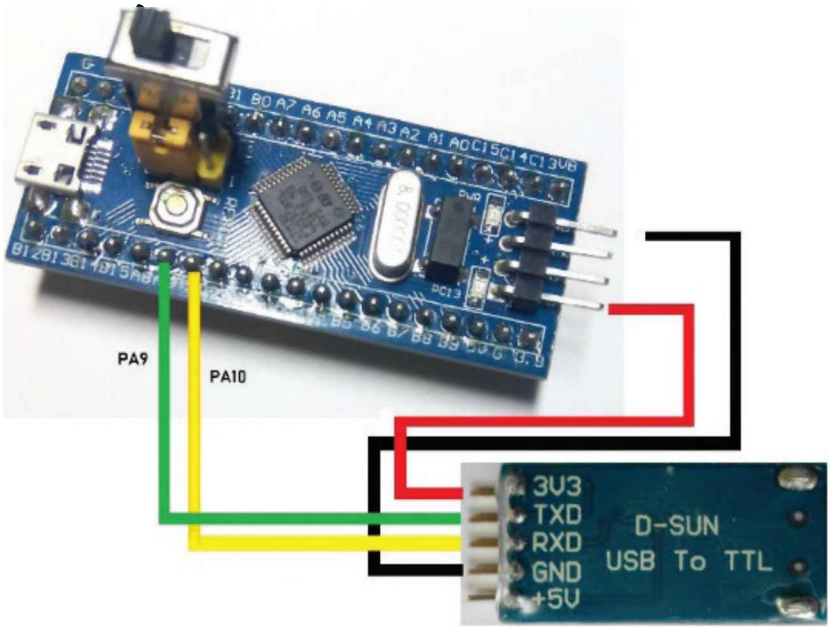


Figura 6.1

El adaptador deberá ser reconocido en nuestro Windows como aparece en la Figura 6.2 en nuestro "Administrador de dispositivos".



Figura 6.2

6.1.2 INSTALACIÓN DEL DRIVER DEL ADAPTADOR USB A RS232 PROLIFIC (PL-2303)

Otro tipo de adaptador que podemos encontrar en el mercado posee el chip fabricado por la empresa taiwanesa Prolific Technology Inc., el PL-2303 (*http://www.prolific.com.tw*). Existe alguna dificultad en la instalación de este driver controlador que pasamos a explicar.

- **Paso 1.** Dependiendo de la versión de Windows que tengamos, descargaremos la última versión del driver de la siguiente dirección:

  http://www.prolific.com.tw/US/ShowProduct.aspx?p_id=225&pcid=41

  Si la versión de Windows es la 8 o superior, será mejor que descarguemos una versión más antigua del driver; ya que con estas versiones, los drivers más actuales tendrán problemas debido a las protecciones de seguridad que posee el Windows y provocará el error: "Código 10" .

  http://wp.brodzinski.net/wp-content/uploads/2014/10/IO-Cable_PL-2303_Drivers-Generic_Windows_PL2303_Prolific.zip

- **Paso 2.** Descomprimimos el fichero y ejecutamos el programa instalador "**PL2303_DriverInstaller_Vxxx**", es importante que el instalador lo ejecutemos con permisos de Administrador.

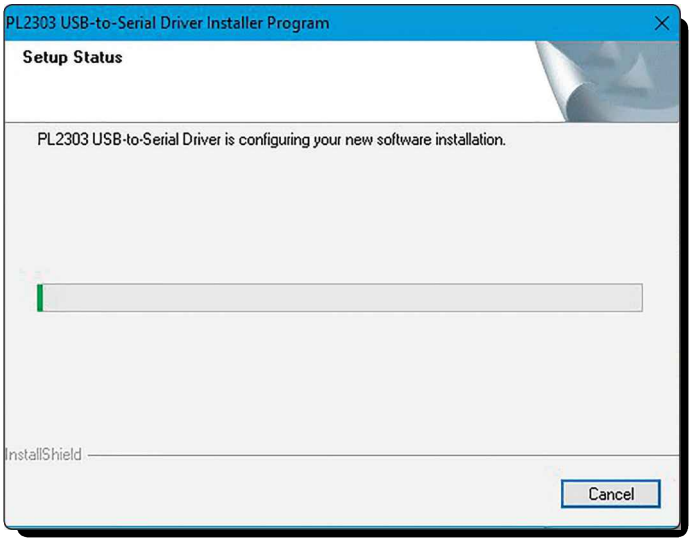


Figura 6.3

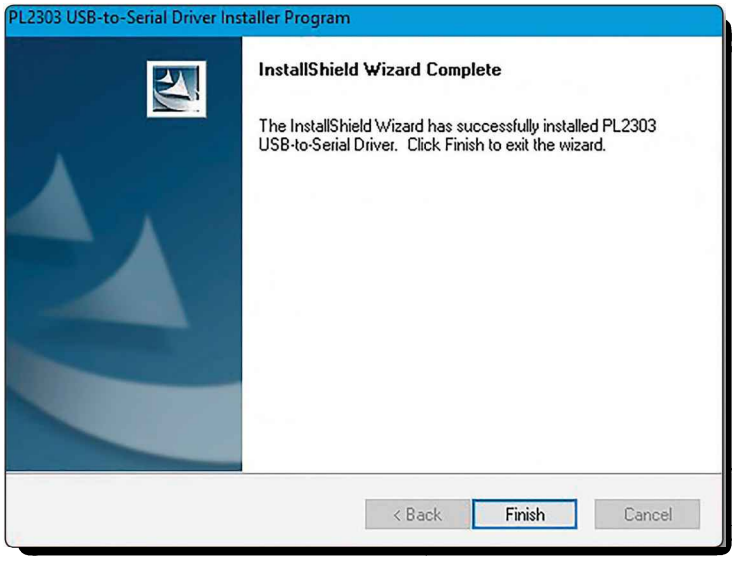


Figura 6.4

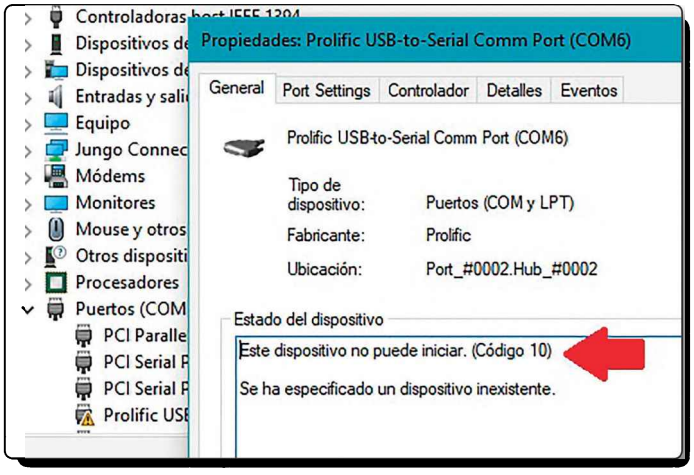


Figura 6.5

Paso 3. Tal como comentamos en el paso 2, el error que se produce al instalar el driver (a pesar de haber descargado una versión más antigua para solucionarlo, debido a que nuestro Windows por defecto instalará la versión del driver más moderna),se solucionará siguiendo los siguientes pasos:

Observamos que al conectar nuestro adaptador, este se ha instalado mostrando el error "***Este dispositivo no puede iniciar (Código 10)***", (Figura 6-5).

- **Paso 4.** En el administrador de dispositivos seleccionamos nuestro adaptador y vamos a la opción de "**Propiedades**", como se muestra en la Figura 6.6:

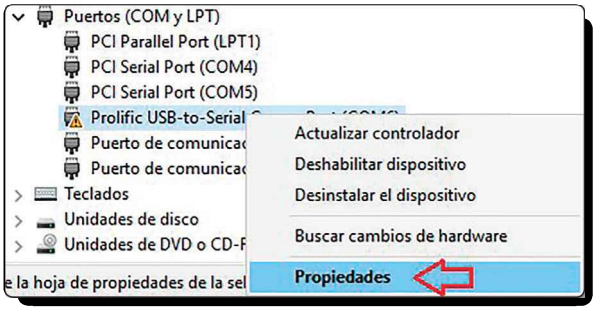


Figura 6.6

- **Paso 5.** Seleccionamos la pestaña "**Controlador**" y pulsamos en "**Actualizar controlador**", como se indica en la Figura 6.7.

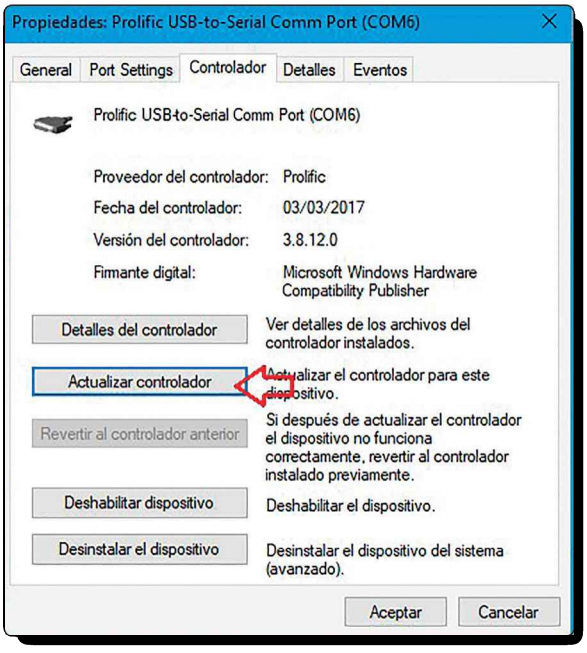


Figura 6.7

- **Paso 6.** En la ventana que se nos abrirá, Figura 6.8, seleccionaremos "**Buscar software de controlador en el equipo**":

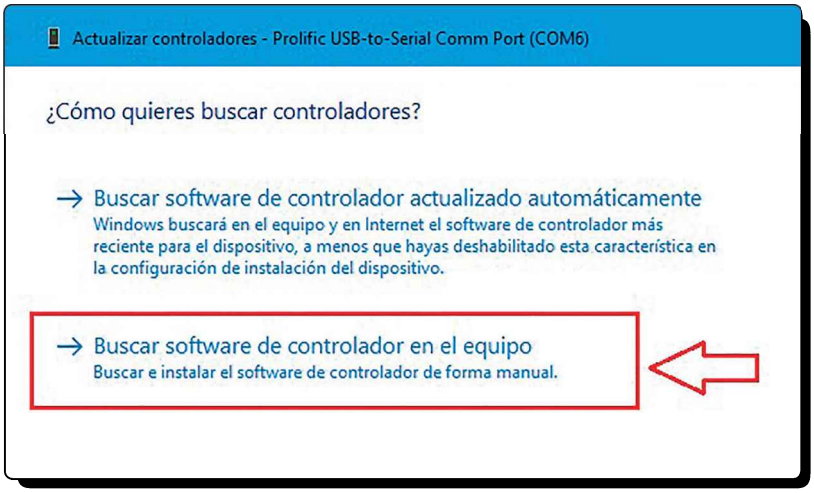


Figura 6.8

- **Paso 7.** En la siguiente pantalla, Figura 6-9, seleccionaremos "**Elegir en una lista de controladores disponibles en el equipo**":

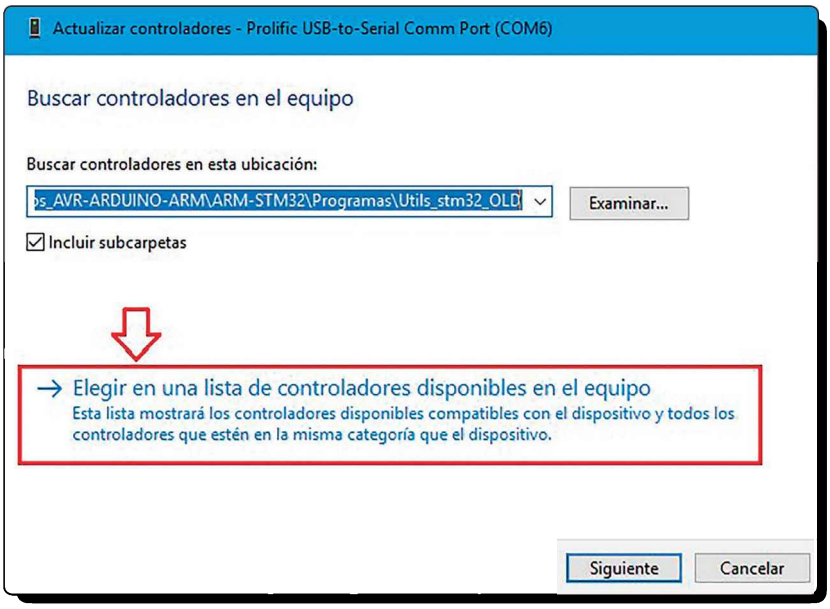


Figura 6.9

- **Paso 8.** A continuación, en la pantalla de la Figura 6.10, seleccionaremos concretamente el controlador que corresponde a la versión "**3.3.2.105..**" del 2008, que habíamos descargado al principio, ya que puede haber sucedido que nuestro Windows hubiera instalado él mismo, un driver distinto automáticamente.

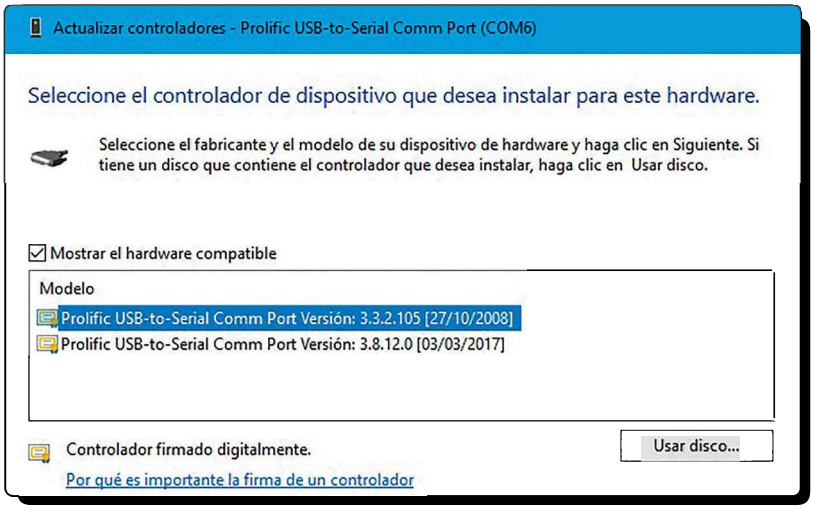


Figura 6.10

- **Paso 9.** Esta versión sí deberá ya funcionar sin el error (Código10) tal como se muestra en la Figura 6.11.



Figura 6.11

Puede que el Windows nos pida reiniciar el equipo para establecer los cambios. Lo haremos.

6.1.3 PROGRAMAR LA PLACA CON UN ADAPTADOR USB a RS232 y EL FLASH LOADER DEMONSTRATOR DE ST

El proveedor STMicroelectronics facilita también una herramienta con la que es posible programar directamente nuestra placa sin necesidad de utilizar los entornos de trabajo. Se puede descargar gratuitamente y los pasos para su uso e instalación son los que se muestran a continuación.

- **Paso 1.** Nos descargaremos la última versión del "**Flash loader demonstrator**" desde la casa oficial STMicroelectronics en la siguiente dirección: *http://www.st.com/en/development-tools/flasher-stm32.html*



Figura 6.12

- **Paso 2.** Al iniciarlo, se nos mostrará una pantalla como la de la Figura 6.13 y seleccionamos el puerto correspondiente.

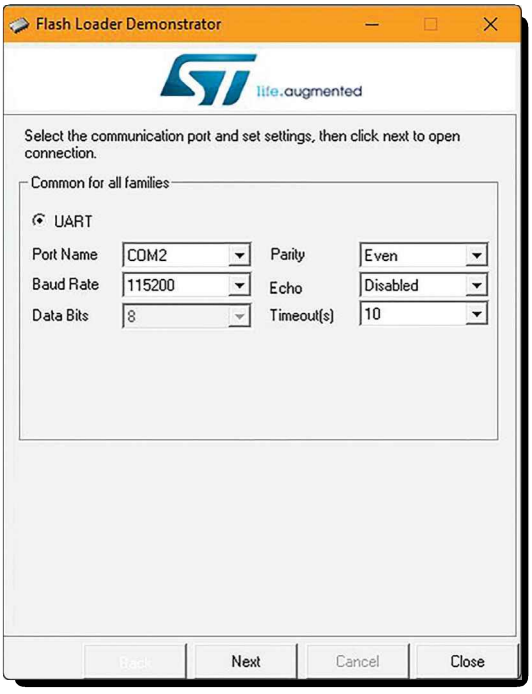


Figura 6.13

◤ **Paso 3.** En la siguiente pantalla, será necesario pulsar el botón de reset de nuestra placa para que continúe el proceso.

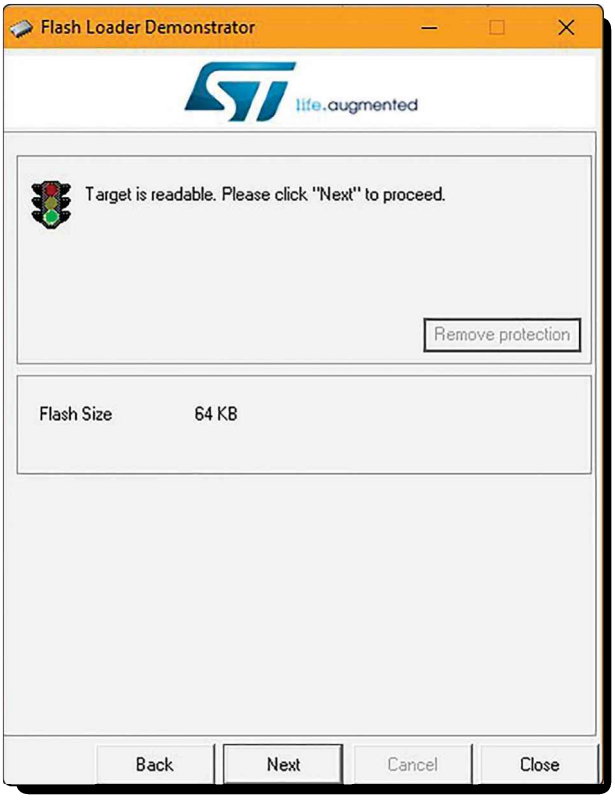


Figura 6.14

ⓘ NOTAS

Es importante que antes de realizar el siguiente paso, pulsemos el botón "Reset" de nuestra placa.

- **Paso 5.** El programa detectará el microcontrolador que posee la placa que tenemos conectada y mostrará información de sus características automáticamente (Figura 6.15).

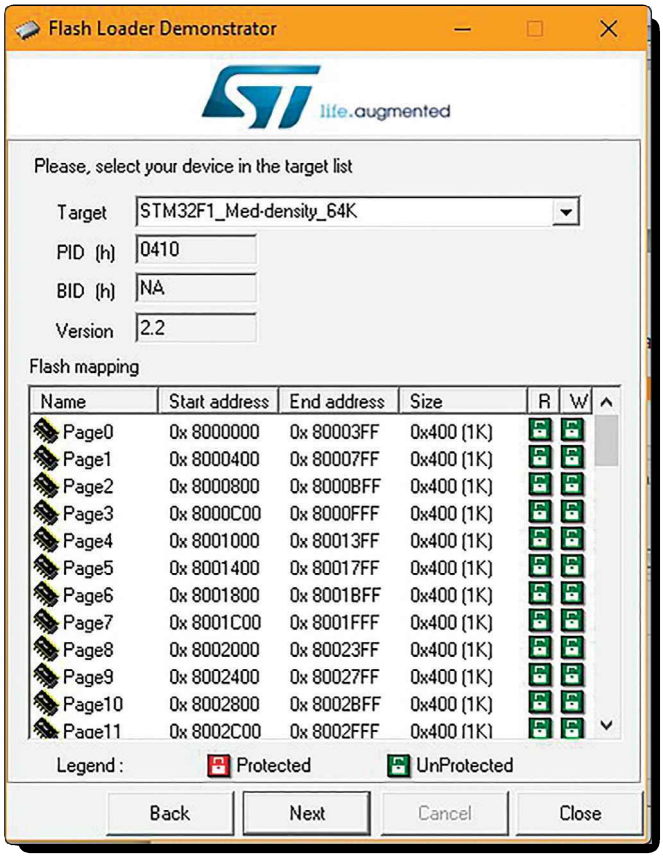


Figura 6.15

- **Paso 6.** En la siguiente pantalla, Figura 6.16, seleccionamos el apartado "**Download from file**" y cargamos el fichero ***.hex** que se generó cuando compilamos nuestro proyecto.

En esta ventana, también deberemos marcar las siguientes opciones:

- Jump to the user program.
- Verify after download.

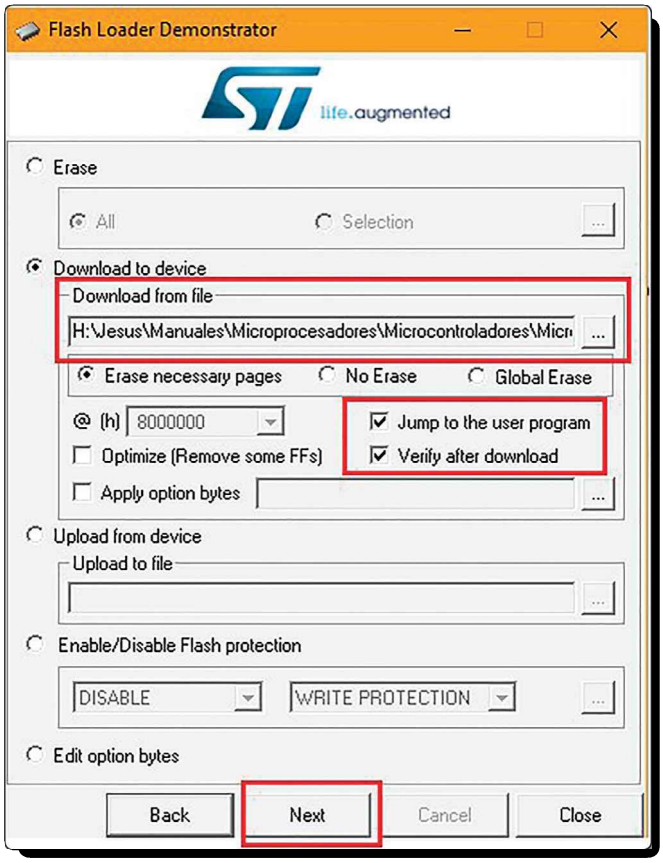


Figura 6.16

Paso 7. Si todo ha ido correctamente, se nos mostrará una franja en color verde como la de Figura 6.17, indicando que se ha programado correctamente nuestra placa.

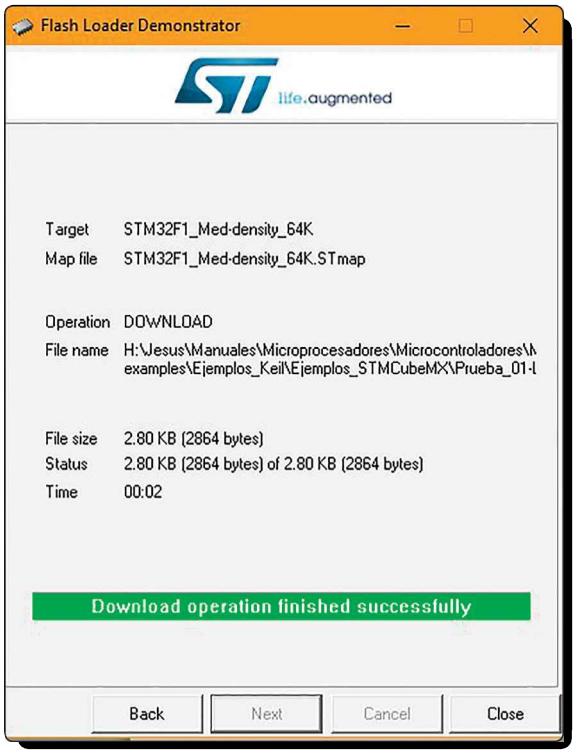


Figura 6.17

6.2 PROGRAMANDO CON EL ADAPTADOR ST-LINK

El módulo USB **ST-LINK V2** es un programador específico para los micros de la familia STM32 y STM8. Permite además, realizar procesos de depuración de nuestro código directamente en las placas desde los editores y entornos de trabajo.

A continuación indicamos los pasos necesarios para instalarlo en nuestro sistema y cómo utilizarlo para programar nuestras placas.

Instalación del dispositivo en Windows

Nos descargaremos los drivers correspondientes desde la propia página de STMicroelectronics en la siguiente dirección: *http://www.st.com/content/st_com/en/products/development-tools/hardware-development-tools/development-tool-hardware-for-mcus/debug-hardware-for-mcus/debug-hardware-for-stm32-mcus/st-link-v2.html*

Una vez instalado el driver, conectamos el adaptador a nuestro sistema y a la placa de pruebas para que nuestro sistema lo reconozca, de acuerdo a las conexiones que se indican en la Figura 6.18.

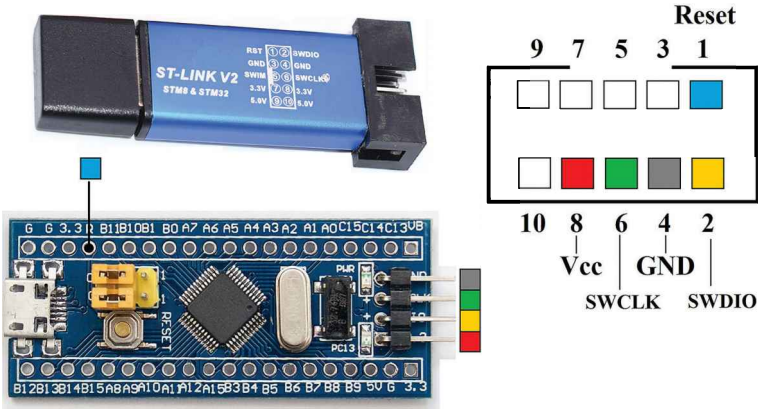


Figura 6.18

En nuestro sistema Windows, debe haberse instalado el driver como se muestra en las Figuras 6-19 y 6-20.

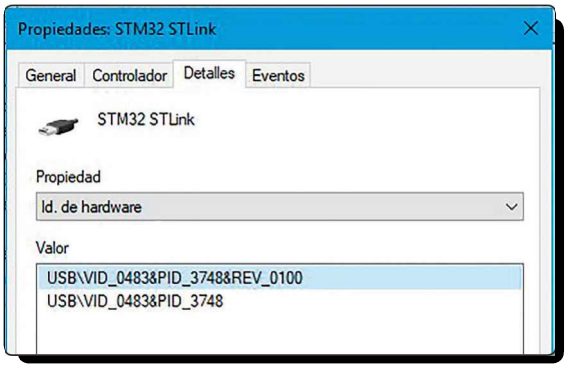


Figura 6.19

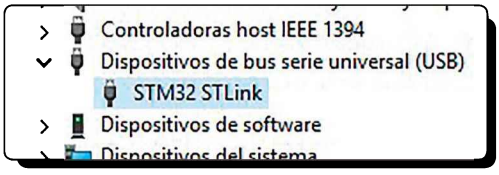


Figura 6.20

6.2.1 PROGRAMANDO EN ARDUINO CON EL ADAPTADOR ST-LINK

Una vez conectado el adaptador ST-LINK a nuestro ordenador y a la placa en la forma que se indica en la Figura 6.18 y 6.21, iniciamos el IDE de Arduino.

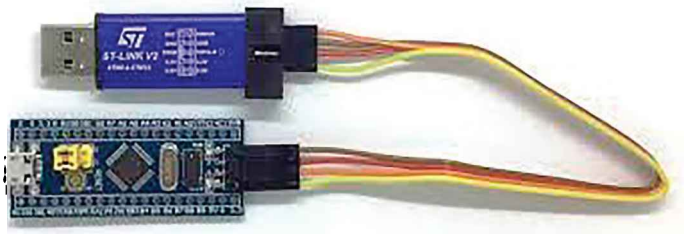


Figura 6.21

Una vez seleccionado el tipo de placa en el menú Herramientas, seleccionamos en el apartado "**Upload method: STLink**", como se muestra en la Figura 6.22.

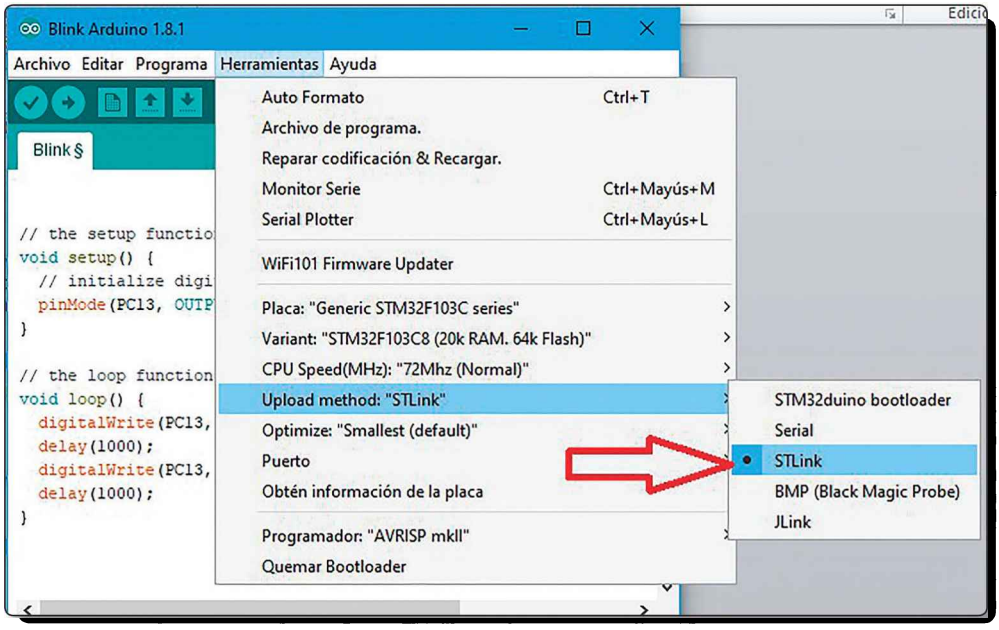


Figura 6.22

Después, bastará con cargar el sketch en nuestra placa, pulsando en el botón de siempre para que se programe.

No nos debe preocupar, que en el IDE de Arduino siga apareciendo en la línea inferior, el mensaje de que está seleccionado un puerto COMx como en la Figura 6.23. Ya que no se tendrá en cuenta y el programa nos grabará correctamente la placa.

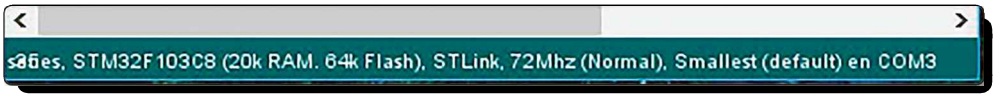


Figura 6.23

6.2.2 PROGRAMANDO DIRECTAMENTE CON EL SOFTWARE ST-LINK UTILITY

También para su empleo con este adaptador, el fabricante ST nos suministra una utilidad que podemos utilizar para grabar directamente en nuestras placas los ficheros binarios .hex que hayamos compilado.

Deberemos descargarnos el fichero para instalarlo desde la siguiente dirección de internet:

http://www.st.com/en/development-tools/stsw-link004.html

- **Paso 1.** Una vez instalado, al iniciarlo, seleccionaremos nuestro fichero a través del menú "**File**" en la opción "**Open file…**" (Figura 6-24).

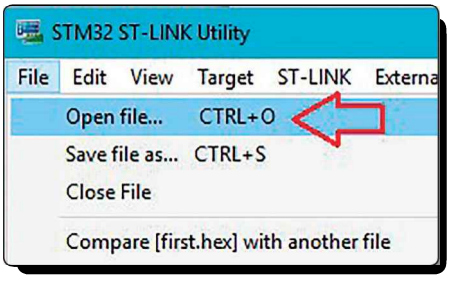


Figura 6.24

En los proyectos realizados con el entorno Keil MDK, el fichero binario con nuestra compilación suele situarse en el directorio "\Objects".

- **Paso 2.** Primero deberemos realizar el proceso de conexión con el microcontrolador, seleccionando la opción "**Connect**" en el menú "**Target**", como en la Figura 6.25.

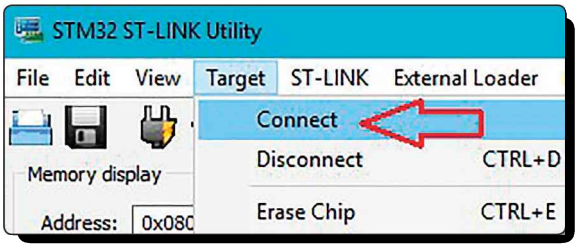


Figura 6.25

Después, regresamos al menú "**Target**" y seleccionamos la opción "**Program & Verify…**", para proceder a grabar en nuestra placa, como se ve en la Figura 6.26.

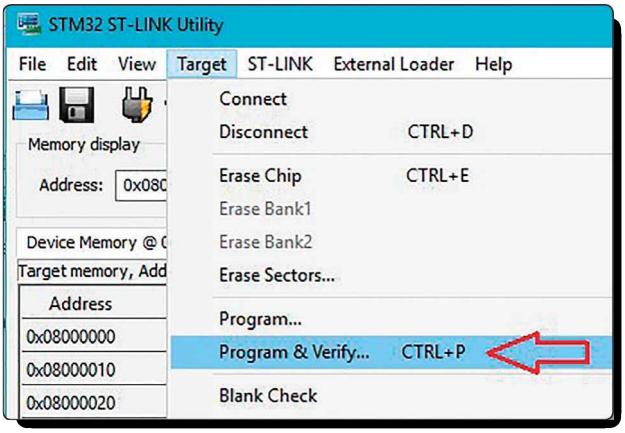


Figura 6.26

En la siguiente pantalla, Figura 6.27, pulsaremos el botón "**Start**" para que se inicie el proceso de grabación.

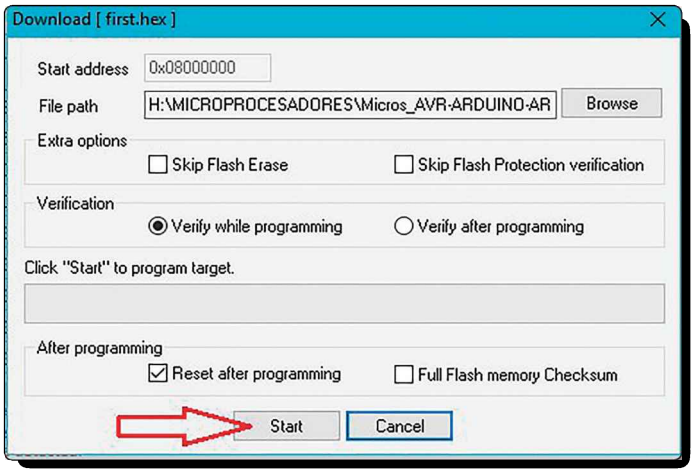


Figura 6.27

Se grabará nuestro programa en la placa.

6.2.3 CONFIGURAR EL KEIL PARA PROGRAMAR CON EL ADAPTADOR ST-LINK (Método 1)

Para poder utilizar el adaptador ST-Link como programador de nuestras placas desde el propio entorno de trabajo Keil, deberemos configurarlo antes.

Es importante que ya tengamos conectado el adaptador a nuestro ordenador y además tener abierto un proyecto en el entorno Keil para proceder con los siguientes pasos de configuración. Indicar también, que solo se establecerán las siguientes configuraciones en el proyecto que ya está abierto, siendo necesario repetir los pasos para que se establezcan en un nuevo proyecto.

- **Paso 1.** Tras iniciar el Keil, debemos seleccionar el icono "**Options for Target**", punto [1] de la Figura 6.28; y en la ventana que se nos abrirá, seleccionamos, punto [2], la pestaña "**Debug**" y dentro pulsamos en el botón "**Settings**", a la derecha de la figura , punto [3]. En la lista desplegable, buscamos y seleccionamos "**ST-Link Debugger**", punto [4]; una vez seleccionado, pulsaremos de nuevo en "**Settings**" donde se nos abrirá una nueva ventana.

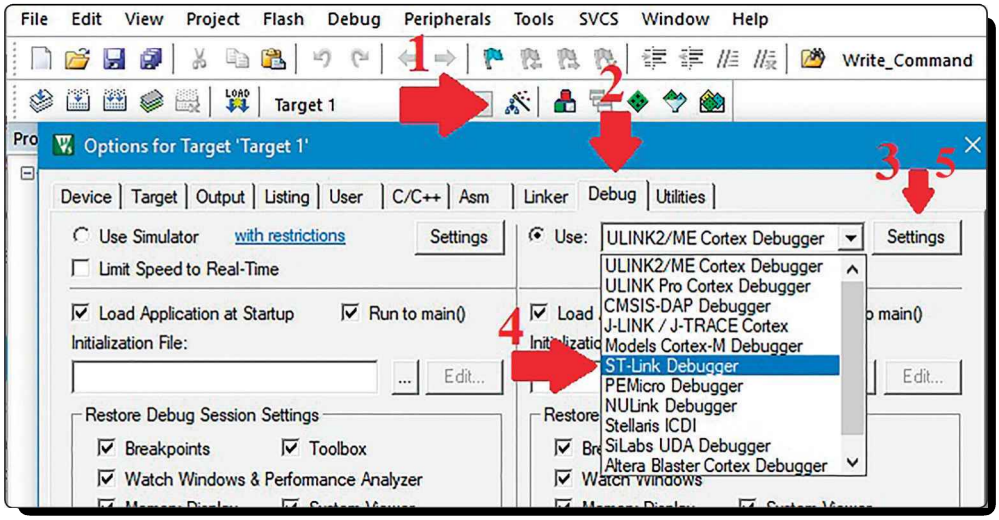


Figura 6.28

◤ **Paso 2.** Si tenemos conectado nuestro adaptador, el programa lo reconocerá y nos aparecerán los datos leídos en el mismo: modelo, número de serie, velocidad de reloj, etc.; como se muestra en la Figura 6.29.

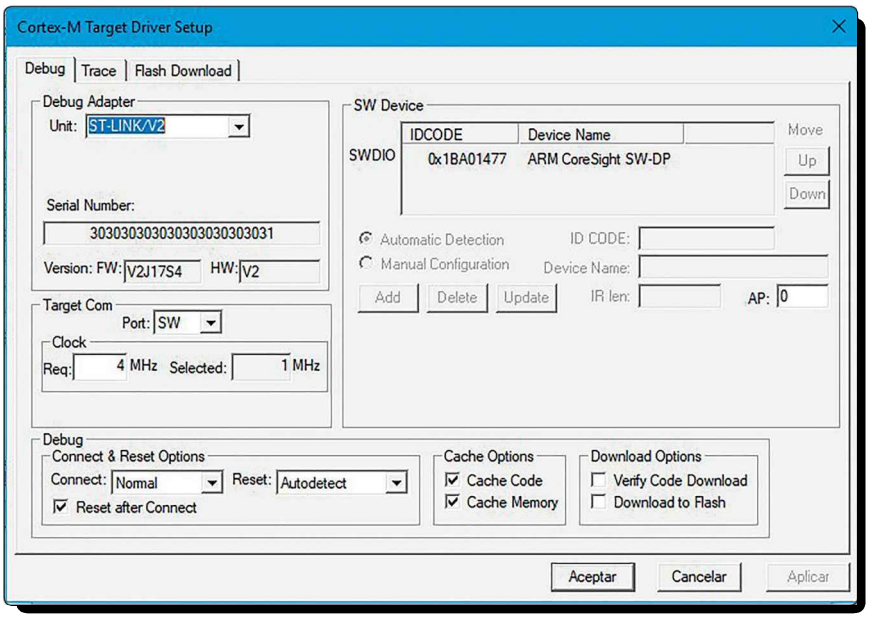


Figura 6.29

- **Paso 3** Para que el proceso se realice correctamente, debemos marcar en el apartado "**Download Options**" las opciones: "**Verify Code Download**" y "**Download to Flash**", como se muestra en la Figura 6.30. Indicando así, que se verifiquen las líneas de código durante la programación de nuestra placa y cargando el código en la memoria flash de usuario de nuestro microcontrolador.

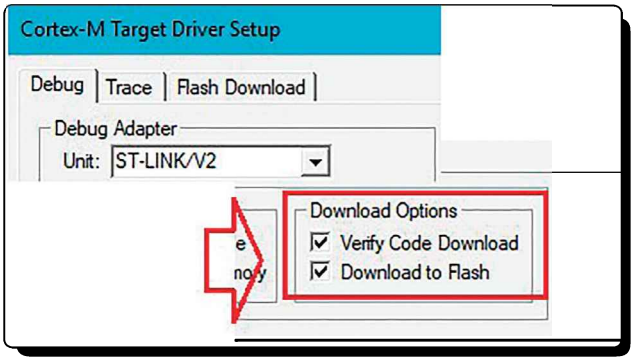


Figura 6.30

- **Paso 4** Después, en la misma pantalla, seleccionaremos la pestaña de "**Flash Download**" y en "**Download Function**", marcamos también la casilla de "**Reset and Run**" para indicar que se produzca un reset y se ejecute nuestro código nada más sea programada; dejando las demás opciones como aparecen por defecto. (Figura 6-31).

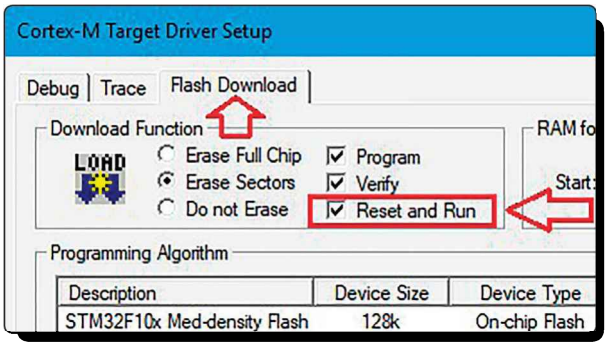


Figura 6.31

Para finalizar, pulsamos "Aceptar" y ya tendremos configurado nuestro adaptador ST-Link para poder así programar nuestras placas.

Para comprobar su correcta configuración, podemos pulsar en el botón "**Load**" como se muestra en la Figura 6.32 y así, se programará la placa a través del adaptador

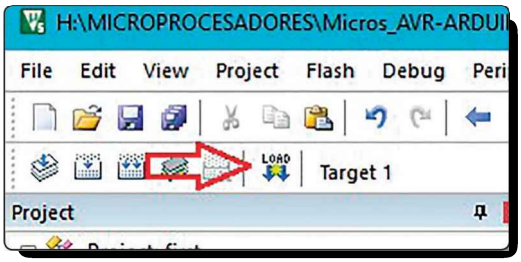


Figura 6.32

Al final del proceso, si todo es correcto, veremos el siguiente mensaje en la ventana de "Build Output" según muestra la Figura 6.33.

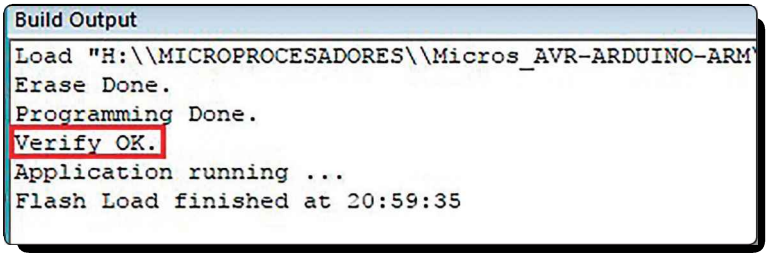


Figura 6.33

6.2.4 CONFIGURAR EL KEIL PARA PROGRAMAR CON EL ADAPTADOR ST-LINK

(Método 2)

Existe otra forma de configurar este adaptador para poder ser utilizado desde el Keil como programador de nuestras placas y que explicamos a continuación.

Para ello, necesitaremos copiar un programa que se nos instala cuando se cargan las librerías del microcontrolador STM32 en el Arduino IDE; según se explicó en el capítulo 3, apartado 3.2 de este libro: "PRIMEROS PASOS CON EL IDE DE ARDUINO".

- **Paso 1.** Debemos copiar completa la carpeta "tools\win**stlink**" que se creó en el directorio donde está instalado el IDE de Arduino, dentro del siguiente directorio:

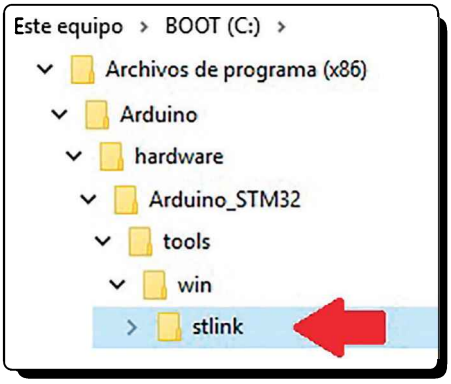


Figura 6.34

C:\Program Files (x86)\Arduino\hardware\Arduino_STM32\tools\win ***stlink***
(Copiar el directorio completo, en Keil o en una carpeta del raíz de C:\)

- **Paso 2.** Copiamos esta carpeta dentro del directorio donde se nos instaló el Keil, en la ubicación siguiente:

  C:\Keil\Keil_v5\Program\stlink

- **Paso 3.** Iniciamos el programa Keil y abrimos un proyecto. Pulsamos en el botón "**Options for Target**" y, seleccionamos la pestaña "**Output**", donde debemos marcar la casilla "**Create HEX File**", como se muestra en la Figura 6.35.

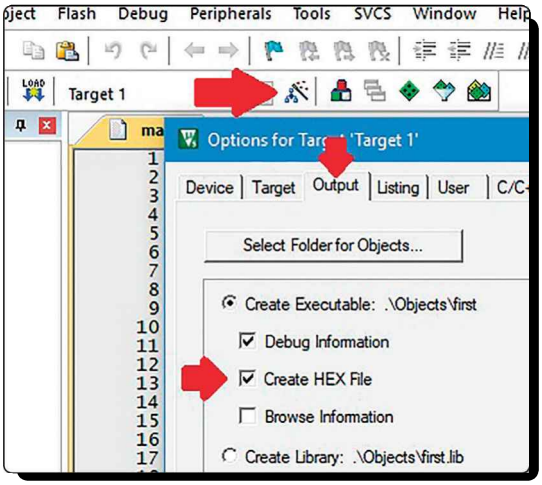


Figura 6.35

Paso 4. Después, seleccionaremos la pestaña "**Utilities**", Figura 6.36; donde marcaremos la casilla "**Use External Tool for Flash Programming**", y copiaremos las líneas siguientes en los dos campos que aparecen:

- En Command:
  - *C:\Keil\Keil_v5\Program\stlink\ST-LINK_CLI.exe*
    (Indicando el lugar donde hemos copiado la carpeta en el paso 2 anterior).
- En Argument:
  - -c SWD -p "$H@H.hex" -Rst -Run
    [ ] Run Independient

Desmarcamos la casilla "Run Independient".

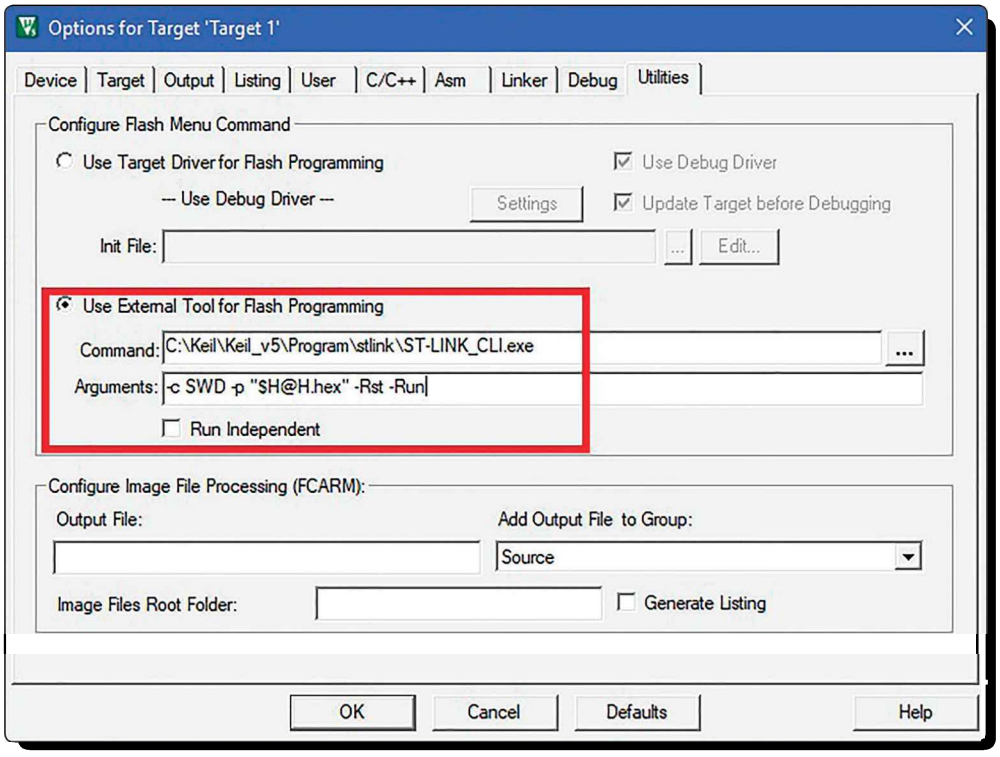


Figura 6.36

Mediante los parámetros que hemos introducido en el campo "**Arguments**", el Keil transfiere a la línea de ejecución del programa ST-LINK_Cli.exe, las siguientes opciones dentro del texto ***$H@H.hex***: el directorio con el nombre del proyecto que tengamos abierto, el directorio "\Objects" y el fichero binario HEX (que marcamos en el paso 3 para que se creara después del proceso de compilación).

Ahora en el proyecto que tenemos abierto, al pulsar "**Load**", (Figura 6-37), se procederá a programar la placa utilizando el adaptador ST-Link.

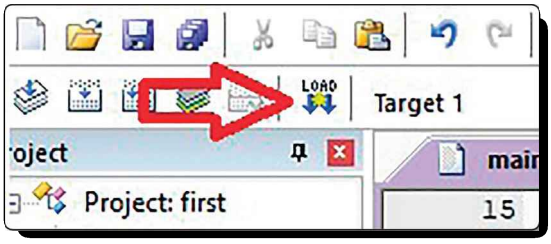


Figura 6.37

Deberemos tener en cuenta que esta configuración solo se establecerá en el proyecto que en ese momento, tengamos abierto y activo; por lo que si queremos que en otro proyecto nuevo se utilice el adaptador mediante esta configuración, se deberán repetir los mismos pasos.

6.2.5 ACTUALIZACIÓN DEL FIRMWARE DEL ADAPTADOR ST-LINK

Este adaptador y el firmware que posee pueden ser actualizados mediante una herramienta de software que nos facilita el fabricante.

El programa se llama "**ST-Link Utility**" y se puede descargar de la página web en la siguiente dirección:

http://www.st.com/en/development-tools/stsw-link004.html

Para actualizar el firmware de nuestro adaptador, una vez instalado y ejecutado, se nos muestra una pantalla igual a la que se muestra en la Figura 6.38.

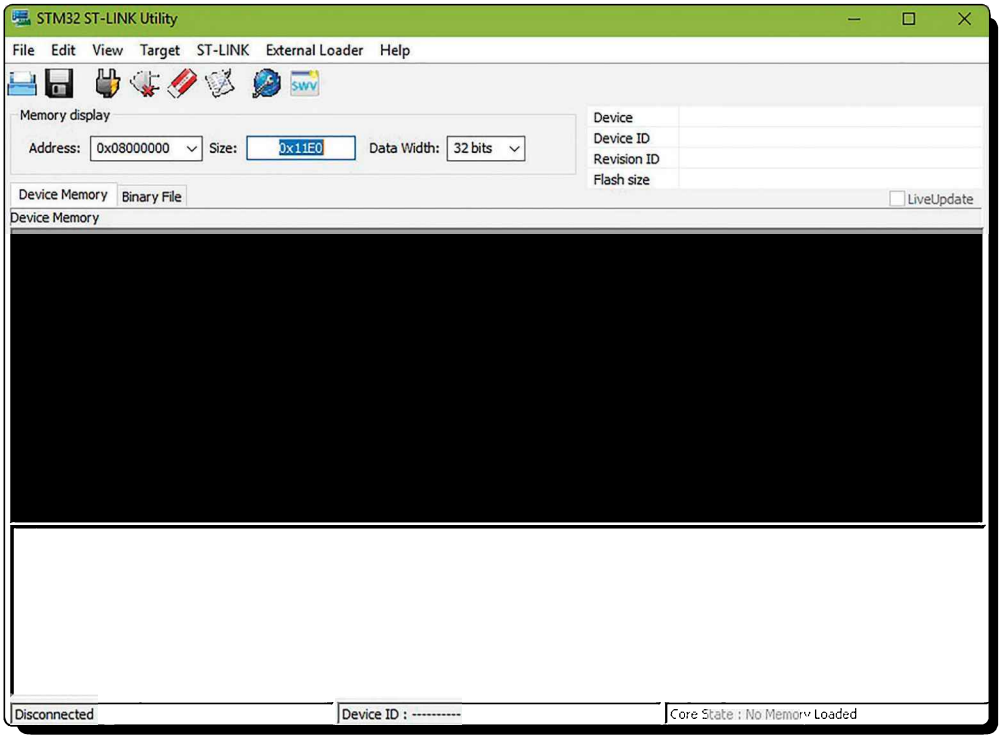


Figura 6.38

- **Paso 1.** Es importante preparar nuestra placa para este proceso especial, ya que requiere que el jumper BOOT de nuestra placa esté en la siguiente configuración.

  Desconecte la alimentación de la placa, coloque el jumper BOOT0 y el BOOT1 en su posición '0' y aplique alimentación para que la placa se ponga en modo DFU.

- **Paso 2.** Conectamos el adaptador a la placa y a nuestro ordenador, y a continuación, iniciamos el programa ST-Link Utility; donde deberemos comprobar si tenemos bien configuradas las opciones de nuestro programa. Para ello, seleccionamos en el menú "**Target**" la opción "**Settings**…" y comprobamos que las opciones que se muestran en la Figura 6.39 son las mismas que nos aparecen a nosotros.

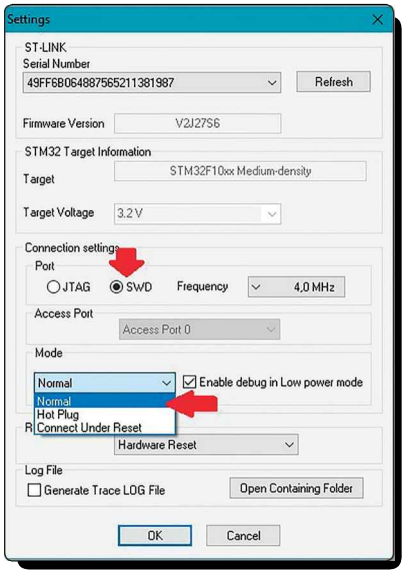


Figura 6.39

◤ **Paso 3.** Al pulsar en el botón "Ok", veremos que el software se conectará con nuestro microcontrolador y nos aparecerá en la pantalla una serie de información sobre el mismo, como se muestra en la Figura 6.40.

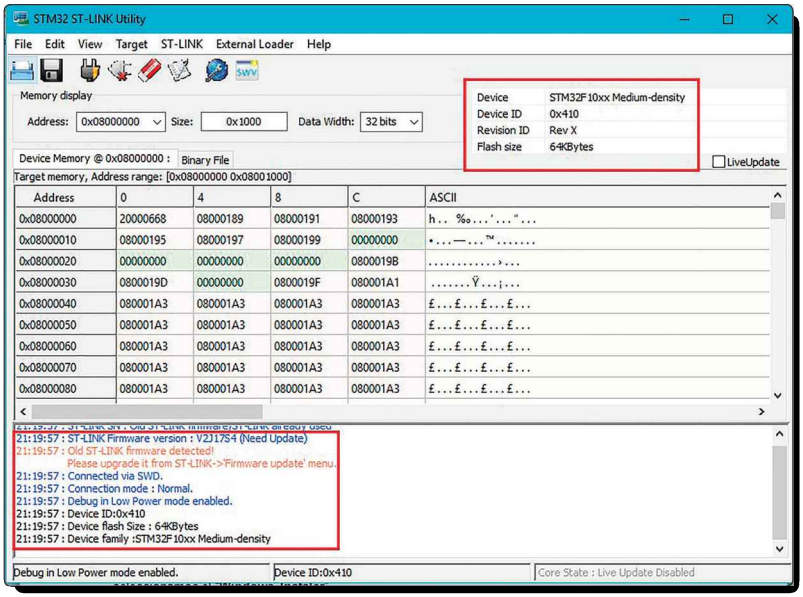


Figura 6.40

- **Paso 4.** Puede suceder que no se conecte con nuestro microcontrolador tras el paso anterior; por lo que seleccionaremos la opción "**Connect**" en el menú "**Target**" y a continuación volveremos al menú "**Target**" y seleccionaremos la opción "**Settings**". Ahora en las opciones, cambiaremos en el apartado "**Connection settings**" el "Port" a "**JTAG**" y en el apartado "**Mode**" seleccionaremos la opción "**Connect Under Reset**" (Figura 6-41).

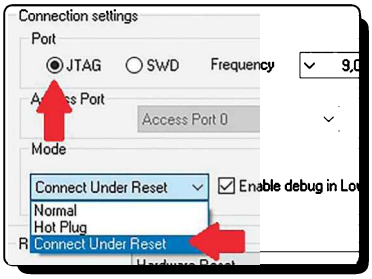


Figura 6.41

- **Paso 5.** Debemos desconectar y conectar de nuevo el adaptador a nuestro ordenador.

- **Paso 6.** Seleccionamos a continuación en el menú "**ST-LINK**" la opción "**Firmware update**".

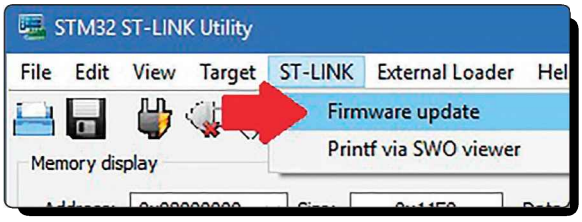


Figura 6.42

Y en la pantalla que se nos muestra en la Figura 6.43, vemos que nos aparecerá en "Firmware Versión: VX.xxx.xx" la versión de firmware que tiene nuestro adaptador. En la línea inferior, el programa se habrá conectado con el servidor de firmware de la casa ST y nos mostrará la última versión existente para nuestro adaptador. En caso de necesitar actualizarlo a una versión más moderna, pulsaremos en el botón "YES >>>" para que se actualice nuestro adaptador.

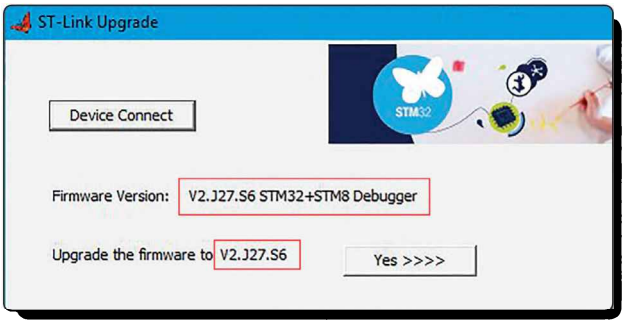


Figura 6.43

El final del proceso se nos avisará con una ventana con el mensaje que aparece en la Figura 6.44.

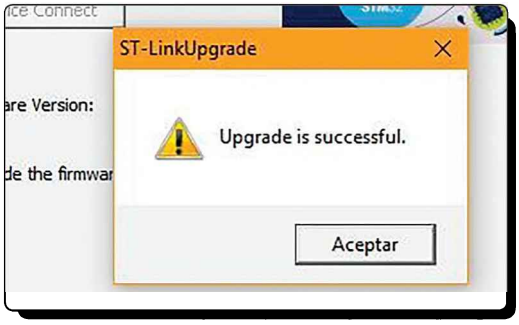


Figura 6.44

7

PRINCIPIOS BÁSICOS DEL HARDWARE DE LOS STM32

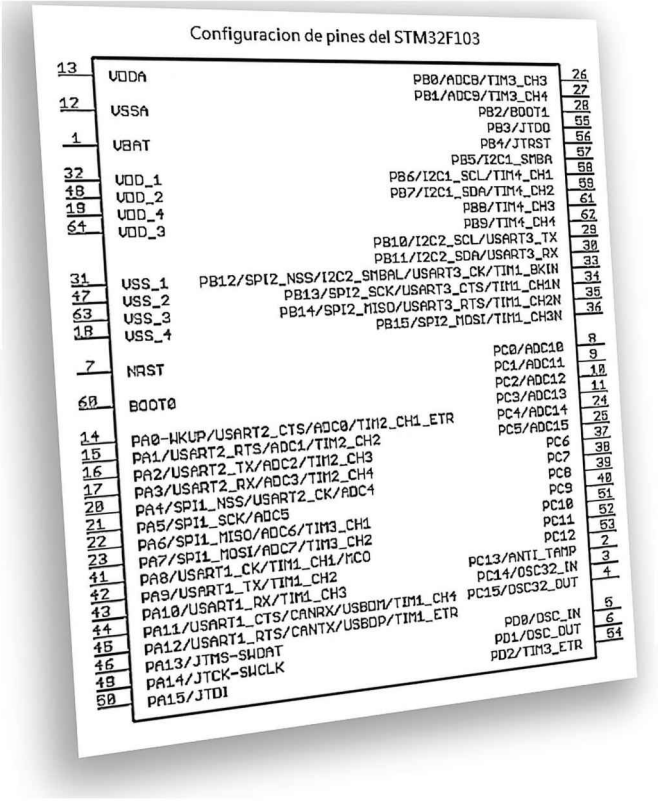


Figura 7.1

Antes de comenzar a estudiar los elementos de la programación de los microcontroladores STM32, comenzaremos con un breve resumen de los elementos que se usan en su programación, sin que con ello nos adentremos en los conocimientos técnicos profundos de sus ingeniería interna, ya que esta obra no abarca este aspecto.

7.1 PUERTOS Y PINES

En los microcontroladores de la familia STM32F1xx, cada puerto de E/S de propósito general de 16 bits tiene alrededor de diez registros internos de configuración de 32 bits. Todos ellos direccionables a través de un bus de 32 bits, donde a su vez, se interconectan con una circuitería interna de interfaces periféricas bastantes complejas.

Visto así, es muy difícil manejar toda esta configuración a través del direccionamiento de cada registro; por ello, es tan importante la biblioteca de interface que el diseñador de estos circuitos nos aporta ya que nos permite manejarlos mediante sencillas funciones que ya resumen ese proceso y garantizan una simplicidad.

Por ejemplo, con la librería "stm32f1xx_gpio.h", se implementa al principio en nuestra programación la gestión de todos los puertos y pines del microcontrolador mediante la GPIO, que define la mayoría de opciones necesarias.

Con sencillos comandos como GPIO_Mode_IN/OUT e indicando el número del pin, ya hemos configurado si ese pin será de entrada o de salida.

Para establecer una determinada configuración de un puerto o pin específico en nuestra programación, creamos una estructura que resume toda la operación, de forma sencilla, como vemos en la Figura 7.2.

```
1  /* Configuracion del pin PC13 para el LED */
2  GPIO_StructureInit.GPIO_Pin = GPIO_Pin_13;
3  GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;
4  GPIO_StructureInit.GPIO_Mode = GPIO_Mode_Out_PP;
5  GPIO_Init(GPIOC, &GPIO_StructureInit);
```

Figura 7.2

Veremos, más adelante, cómo se configuran y se opera con ellos.

7.2 PERIFÉRICOS INTERNOS

Dentro del propio microcontrolador existen una gran variedad de periféricos que permiten realizar determinados procesos: módulos de reloj para la sincronización entre los periféricos, convertidores de señales analógicas a digitales, dispositivos de memoria backup, dispositivos de bus de comunicaciones I2C, USART, etc.

Todos estos periféricos, a su vez poseen una gran cantidad de registros de control y registros de datos que a su vez, también poseen pines del microcontrolador asignados permanentemente o que pueden ser reasignados a otros pines para su gestión y comunicación con el exterior.

También para poder operar con estos dispositivos, la biblioteca de librerías que nos aporta STMicroelectronics nos facilita mediante funciones sencillas un esquema simple que realiza ese proceso.

Por ejemplo, en la Figura 7.3, vemos la estructura de configuración de los parámetros de un puerto de comunicaciones RS232, el USART1.

```
1  /* Configuracion GPIO para USART1 ---------------------------------------*/
2  USART_InitTypeDef USART_InitStructure;
3
4  /* Habilitar el reloj para el USART1  -----------------------------------*/
5  RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
6
7  /* Configuracion de comunicacion del USART1  ----------------------*/
8  USART_InitStructure.USART_BaudRate = 115200;
9  USART_InitStructure.USART_WordLength = USART_WordLength_8b;
10 USART_InitStructure.USART_StopBits = USART_StopBits_1;
11 USART_InitStructure.USART_Parity = USART_Parity_No;
12 USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
13 USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
14 USART_Init(USART1, &USART_InitStructure);
15 USART_Cmd(USART1, ENABLE);
```

Figura 7.3

Se nos aporta una serie de librerías como "stm32f10x_adc.c" para el módulo ADC; "stm32f10x_flash.c" para el manejo de la memoria flash interna; "stm32f10x_rcc.c" para el manejo de los relojes específicos, que solo con incluirlos al principio en nuestra programación se añaden a nuestra compilación y nos permiten realizar operaciones con estos periféricos con sencillas funciones.

7.3 MÉTODO DE PROGRAMACIÓN

La metódica general en nuestra programación, que se utilizará en los diferentes proyectos de ejemplo, que veremos en los próximos capítulos, será igual para casi todos los ejemplos y válida para cualquier entorno de desarrollo que empleemos de los suministrados por el diseñador. Al igual que será también válido para su uso con cualquier otro microcontrolador de la familia STM32 Cortex-Mx.

Recordemos que todos se basan en la tecnología **CMSIS** (*Cortex Microcontroller Software Interface Standard*) que establece un estándar que se utilizará en todos los procesadores Cortex-M.

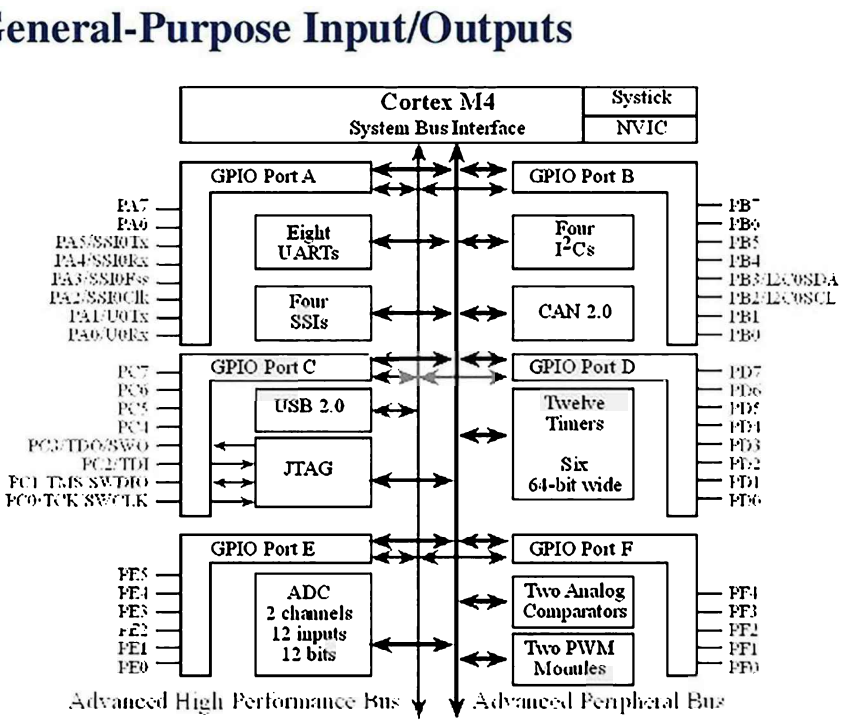


Figura 7.4

En la Figura 7.4 se detallan cada uno de los periféricos que contiene nuestra placa de prueba y los pines correspondientes en la Figura 7.5.

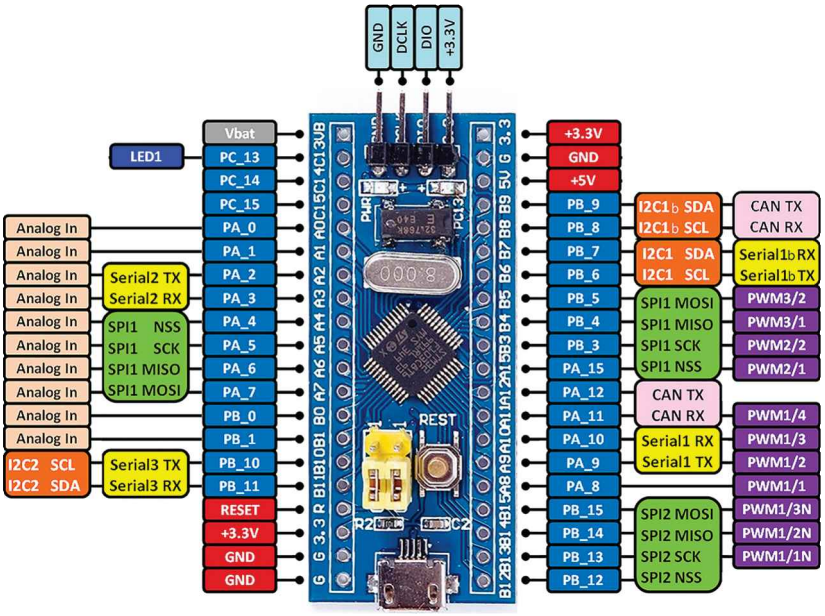

GND
DCLK
DIO
+3.3V
LED1
Vbat
PC_13
PC_14
PC_15
PA_0
PA_1
PA_2
PA_3
PA_4
PA_5
PA_6
PA_7
PB_0
PB_1
PB_10
PB_11
RESET
+3.3V
GND
GND
Analog In
Analog In
Analog In
Analog In
Analog In
Analog In
Analog In
Analog In
Analog In
Analog In
I2C2 SCL
I2C2 SDA
Serial2 TX
Serial2 RX
SPI1 NSS
SPI1 SCK
SPI1 MISO
SPI1 MOSI
Serial3 TX
Serial3 RX
+3.3V
GND
+5V
PB_9
PB_8
PB_7
PB_6
PB_5
PB_4
PB_3
PA_15
PA_12
PA_11
PA_10
PA_9
PA_8
PB_15
PB_14
PB_13
PB_12
I2C1b SDA
I2C1b SCL
I2C1 SDA
I2C1 SCL
SPI1 MOSI
SPI1 MISO
SPI1 SCK
SPI1 NSS
CAN TX
CAN RX
Serial1 RX
Serial1 TX
SPI2 MOSI
SPI2 MISO
SPI2 SCK
SPI2 NSS
CAN TX
CAN RX
Serial1b RX
Serial1b TX
PWM3/2
PWM3/1
PWM2/2
PWM2/1
PWM1/4
PWM1/3
PWM1/2
PWM1/1
PWM1/3N
PWM1/2N
PWM1/1N
REST

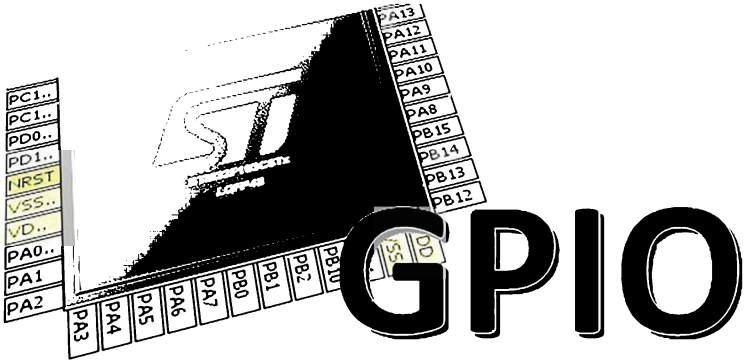
Figura 7.5

PARTE III

PROGRAMACIÓN DEL MICROCONTROLADOR STM-32 CON C++

8

PROGRAMACIÓN GPIO



Ejemplo de control de puertos GPIO

Configurar los parámetros del **GPIO** (*General Purpose Input/Output, Entradas/Salidas de Propósito General*) es la parte de nuestra programación donde debemos de especificar qué pines vamos usar y si serán de entrada o de salida.

En el primer programa que empleamos al principio del libro, en el que programábamos que un LED se encendiera y apagara, ahora podemos modificarlo ligeramente y tratar de configurar los pines del microcontrolador para tener además de un pin de salida en **PC13,** donde está conectado el LED de prueba de nuestra placa, otro de entrada en el pin **PB0,** en el que conectaremos un pulsador que cuando detecte que ha sido pulsado nos encienda el LED.

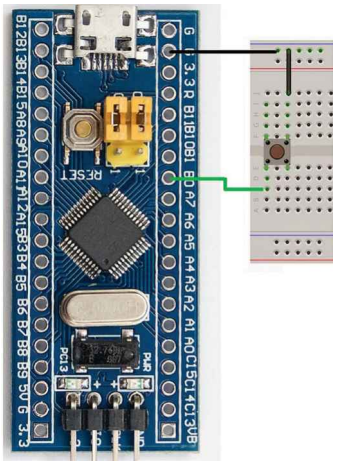


Figura 8.1

```
#include "stm32f10x.h"          // Libreria principal del micro
#include "stm32f10x_rcc.h"      // Librería de control RCC
#include "stm32f10x_gpio.h"     // Librería de control de GPIO

/*                  Modulo Principal                                    */
//-------------------------------------------------------------------------
int main(void){

      /* Nombramos la estructura que contendra los GPIOx ------------*/
      GPIO_InitTypeDef GPIO_StructureInit;

      /* Activamos el clock para GPIOA y GPIOC  ---------------------*/
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

      /* Configuramos el pin PC13 para el LED  ----------------------*/
      GPIO_StructureInit.GPIO_Pin = GPIO_Pin_13;
      GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_StructureInit.GPIO_Mode = GPIO_Mode_Out_PP;
      GPIO_Init(GPIOC, &GPIO_StructureInit);

      /* Configuramos el pin PB0 para el pulsador  ------------------*/
      GPIO_StructureInit.GPIO_Pin = GPIO_Pin_0;
      GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_StructureInit.GPIO_Mode = GPIO_Mode_IPU;
      GPIO_Init(GPIOB, &GPIO_StructureInit);

      while(1){

            // Comprueba cuando el pin PB0 esta a nivel bajo '0'
            if(GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_0) == 0 )
            {
                  // Encendemos el led
                  GPIO_ResetBits(GPIOC, GPIO_Pin_13);
            }
            else
            {
                  // Apagamos el led
                  GPIO_SetBits(GPIOC, GPIO_Pin_13);
            }
      }
}
```

Figura 8.2

A continuación explicaremos paso a paso el programa.

La estructura de nuestra programación utiliza la que ya conocemos de otros compiladores; es decir, lo primero que debemos introducir en nuestro código, son la llamadas a aquellas librerías que se añadirán al principio de nuestra compilación: las librerías de control de nuestro microcontrolador, la de los puertos o periféricos específicos que vayamos a utilizar y la denominación de las variables globales que serán utilizadas en nuestro código.

Por lo que, en primer lugar, tenemos los **#include**. Es el comando en C++ que nos permite incluir en nuestro código las librerías que posee el Keil para cargar la parte de programación que gestionará los periféricos del microcontrolador que utilizaremos.

En nuestro ejemplo, necesitaremos incluir las librerías siguientes:

```
1  /*     Librería principal del microcontrolador ---------------------------- */
2  #include "stm32f10x.h"
3
4  /*       Librerias de los perifericos    ------------------------------------ */
5  #include "stm32f10x_rcc.h"    // Librería de control RCC
6  #include "stm32f10x_gpio.h"   // Librería de control de GPIO
```

Figura 8.3

La librería que contiene los parámetros específicos del microcontrolador que vamos a emplear "***stm32f10x.h***" –recordar que estamos trabajando con una placa que posee el STM32F103C8- y luego, las librerías para controlar los periféricos que vamos a utilizar: el RCC "**stm32f10x_rcc.h**", ya que necesitamos utilizar los relojes de control de periféricos y la de GPIO "**stm32f10x_gpio.h**", porque necesitamos configurar los parámetros de los puertos y pines que vamos a usar en nuestra programación.

Además de escribirlas en nuestro código, para que se incluyan en nuestra compilación, es importante que recordemos lo que indicamos en nuestro Capítulo 5, que en el Keil MDK es necesario cargarlas desde el Administrador del entorno de compilación (*Manage Run-Time Enviromment*).

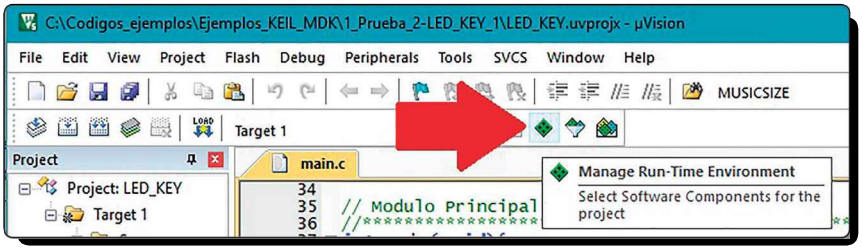


Figura 8.4

Y marcar en el apartado "**Device**" y en "**StdPeriphDrivers**" las librerías "**RCC**" y "**GPIO**".

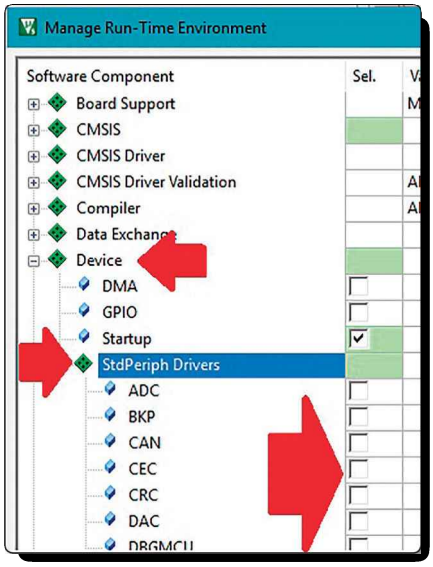


Figura 8.5

Y también, las librerías que siempre deben cargarse para todos nuestros programas y que constituyen el entorno de trabajo: en el apartado "**CMSIS**" la librería de "**CORE**" y en "**Device**" la librería "**Startup**". No obstante, cuando carguemos en nuestro compilador cualquiera de los proyectos de los ejemplos, estas opciones ya estarán establecidas y podrán comprobar cuáles están seleccionadas para su aprendizaje.

Lo siguiente será, siempre, configurar los pines del microcontrolador que vayamos a utilizar en nuestra programación y por ello, necesitaremos crear la "*estructura*" que configurará el uso de los **GPIO**.

En el lenguaje de programación C++, se denomina "*Estructura*" al sistema por el cual se organizará la configuración relativa a un concepto en varios apartados dentro de la misma variable.

En el sistema de programación de los STM32, existe el formato de configuración de todas las opciones necesarias para el uso de cualquier periférico o módulo, el organizarlas dentro de una "estructura" según la siguiente forma:

*Nombre_Periférico*_**InitTypeDef** [*Nombre_Mi_Esctrucura*]

Donde el "**Nombre_Periférico**" es el nombre o nomenclatura que tiene (GPIO, ADC, CAN, DMA…), el comando "**_InitTypeDef**" o "**TypeDef**" que es el que inicia o avisa de que a continuación se va a crear una lista de parámetros que configuran ese periférico o módulo y seguidamente "***Nombre_Mi_Estructura***", que será el nombre que tendrá dicha estructura. Como veremos a lo largo de nuestro libro, con diferentes ejemplos, cada periférico posee su lista de parámetros específico que deben configurarse.

Por todo ello, el primer bloque de nuestra programación será en el que se crea la estructura que contendrá la configuración GPIO de los puertos y pines.

```
1  /* Nombramos la estructura que contendra nuestro GPIOx  */
2  GPIO_InitTypeDef GPIO_StructureInit;
3
4  /* Activamos el reloj el GPIOC  ------------------------*/
5  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC,  ENABLE);
6
7  /* Configuramos el pin PC13 para el LED  ---------------*/
8  GPIO_StructureInit.GPIO_Pin = GPIO_Pin_13;
9  GPIO_StructureInit.GPIO_Mode = GPIO_Mode_Out_PP;
10 GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;
11
12 /* Iniciamos la estructura creada ----------------------*/
13 GPIO_Init(GPIOC, &GPIO_StructureInit);
```

Figura 8.6

Observamos, que en la primera línea, llamamos al comando de creación de la estructura **GPIO_InitTypedef** y al que hemos llamado GPIO_InitStructure. Este nombre lo ponemos nosotros, por lo que podemos nombrarla como queramos: MiEstructura_GPIO, etc.

El siguiente paso necesario será activar el reloj que gestionará nuestro puerto. Para ello, el siguiente comando "***RCC_APB2PeriphClockCmd***" nos activará el **APB2** (*Advanced Peripheral Bus*) que es el reloj que controla los periféricos y que es al que está conectado el puerto GPIO**C** que vamos a emplear. La selección del reloj del periférico, en el entorno de trabajo del Keil MDK, nos avisará con un

mensaje de error sobre el código cuando nos equivoquemos en la selección de un reloj que no lleve el puerto seleccionado. Cada dispositivo o periférico interno del microcontrolador está asociado a un determinado módulo de reloj, como veremos más adelante en cada proyecto de ejemplo.

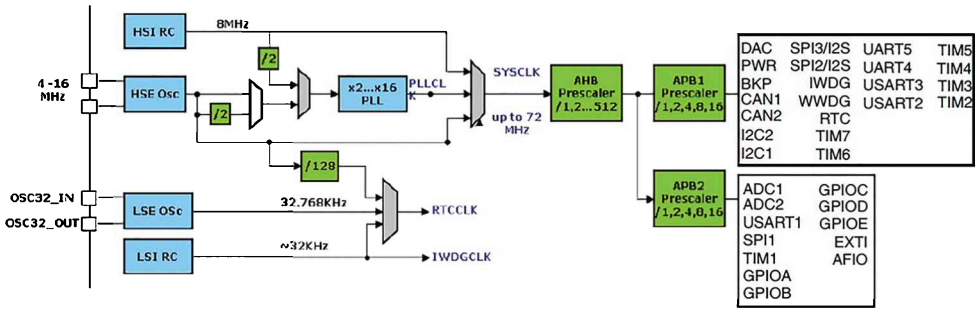


Figura 8.7

En la Figura 8.7 podemos observar la configuración interna de la ingeniería de relojes y módulos divisores que conectan con los periféricos.

Hay que tener en cuenta, que los diferentes relojes que aparecen en la Figura anterior, tienen frecuencias distintas según el listado siguiente:

Lista de Frecuencias de cada reloj del microcontrolador

- Clock Settings:
  - XTAL = 8.00 MHz
  - SYSCLK = 72.00 MHz
  - HCLK = SYSCLK = 72 MHz
  - PCLK1 = HCLK/2 = 36 MHz
  - PCLK2 = HCLK = 72 MHz
  - ADCLK = PCLK2/6 = 12 MHz
  - SYSTICK = HCLK/8 = 9 MHz

Siguiendo con los pasos para nuestro proyecto, necesitaremos configurar los pines que vayamos a emplear, y el cómo van a funcionar; si serán de entrada o de salida y si tienen o no las resistencias internas de salida activadas.

Por ejemplo, si queremos el pin **PC13** como una salida para activar un **LED**, tendremos que configurar los parámetros de número de pin (*GPIO_Pin*), el modo (*GPIO_Mode*), la velocidad (*GPIO_Speed*) y el puerto:

```
1 /* Configuramos el pin PC13 para el LED  ---------*/
2 GPIO_StructureInit.GPIO_Pin = GPIO_Pin_13;
3 GPIO_StructureInit.GPIO_Mode = GPIO_Mode_Out_PP;
4 GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;
5 GPIO_Init(GPIOC, &GPIO_StructureInit);
```

Figura 8.8

En esta parte del código bastará tan solo con cambiar el número del pin en la primera línea -donde pone GPIO_Pin_**13**- por otro número cualquiera como 8, 4 o 0 para indicar otro pin y en la línea cinco, en GPIO_Init, cambiar el puerto por otro, donde pone **GPIOC** por **GPIOB**, **GPIOD** o **GPIOA**, con el fin de establecer la configuración en otro pin y en otro puerto.

En la línea tres, es donde le decimos al compilador qué modo de funcionamiento queremos para ese pin, que sea de salida con *GPIO_Mode*_**Out**_PP o de entrada con *GPIO_Mode*_**In**_PP.

Otro aspecto que se configura en estas líneas, es el modo de conexión de la resistencia interna de los pines, donde dependiendo de si se configuran como salidas o entradas digitales, podrán ser:

Cuando son salidas se podrán configurar de dos maneras: *Mode*_**Out_PP** -tipo "*Push-Pull*"- un tipo interno donde la tensión de salida será independiente de la carga que conectemos a dicho pin; y el "*Mode*_**Out_OD**" -tipo "*Open Drenator*"- en el que el transistor interno funcionará solo como interruptor, necesitando que se le aporte al circuito una tensión externa para que sea una salida positiva o negativa.

En la configuración de entradas digitales, estas podrán ser de tres tipos: "*Mode*_**In_FLOATING**" -tipo flotante- donde dicho pin no posee polarización propia y la tensión será proporcionada por elementos externos, además de una lógica digital contraria en la que pulsador cerrado será '0' y abierto '1'. El otro modo posible es "*Mode*_**In_IPD**" -tipo "*Input Pull-Down*"- cuyo circuito interno estará conectado a tierra mediante su resistencia interna que producirá una lógica digital positiva, pulsador cerrado valdrá '1' y cerrado '0'. Y, por último, el modo "*Mode*_**In_IPU**" -tipo "*Pull-Up*"- donde la circuitería interna está conectada a tensión positiva y producirá una lógica inversa a la anterior, pulsador abierto valdrá '1' y cerrado '0'.

El otro parámetro a configurar es la velocidad del reloj, que será necesaria en el pin, mediante el comando "**GPIO_Speed**" que dependerá del microcontrolador y el reloj que posea la placa que estemos utilizando y que puede ser:

- GPIO_Speed_2MHz // Baja velocidad
- GPIO_Speed_10MHz // Media velocidad
- GPIO_Speed_50MHz // Alta velocidad

Normalmente se recomienda utilizar el valor 50 MHz, pero si observamos que no funciona probaremos con un valor inferior.

Un ejemplo de configuración de un pin como de entrada, sería el que vemos en la Figura 8.9, en el que la configuración del pin **PB0** detectará el estado del pulsador.

```
1  /* Activamos el clock para el GPIOC ------------------- */
2  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
3
4  /* Configuramos el pin PB0 para el pulsador -----------*/
5  GPIO_StructureInit.GPIO_Pin   = GPIO_Pin_0;
6  GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;
7  GPIO_StructureInit.GPIO_Mode  = GPIO_Mode_IPU;
8
9  /* Iniciamos la estructura creada ---------------------*/
10 GPIO_Init(GPIOB, &GPIO_StructureInit);
```

Figura 8.9

Ya tenemos configurada nuestra estructura GPIO necesaria para cada puerto y cada pin a emplear.

Lo siguiente en nuestra programación, será especificar una línea que nos permita comprobar si se pulsa el botón, o lo que es lo mismo, un comando que nos permita leer cuándo nuestro pin cambia a valor '0' o '1'.

```
if(GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_0) == 0 )
```

Figura 8.10

El comando "***GPIO_ReadInputDataBit(****Puertox, Pin)*" es el encargado de leer constantemente el valor del Pin específico de un puerto indicado.

Existen otros comandos con los que podemos leer no solo un bit determinado, sino también el contenido de todos los pines de un puerto determinado:

El comando ***GPIO_ReadInputData*** (*GPIO_TypeDe*f *GPIOx), en el que 'x' será la letra del puerto, y también el comando ***GPIO_ReadOutputData*** (*GPIO_TypeDef* *GPIOx), que nos leerá el contenido de todos los bits de un puerto de salida.

Y por último, crearemos las líneas que nos permitirán cambiar de apagado a encendido el LED conectado al pin PC13.

```
GPIO_SetBits(GPIOC, GPIO_Pin_13);  // Establece a '1' el pin PC13
GPIO_ResetBits(GPIOC, GPIO_Pin_13);// o a '0'  Low
```

Figura 8.11

Mediante los comandos GPIO_***SetBits*** (Puerto, pin) podemos cambiar el pin especificado a valor alto '1' y con GPIO_***ResetBits*** (Puerto, pin) cambiarlo a valor bajo '0'.

También encontramos, que mediante el comando GPIO_***Write*** (*GPIO_TypeDef* *GPIOx, **uint16_t PorValue**), podemos escribir un byte completo en todos los pines de un puerto determinado.

Otra forma de manipular los valores de salida de un pin, es usar el comando GPIO_***WriteBit*** (*GPIO_TypeDef* *GPIOx, unit16_t GPIO_Pin, **BitAction BitValue**), con el que es posible borrar o establecer también un valor para un bit determinado de un puerto.

Hay que indicar que, en el caso del LED de prueba de nuestra placa, tiene lógica invertida, es decir, cuando ponemos la entrada PC13 a '0' se encenderá y con un valor '1' se apagará.

Las dos líneas que configuran el reloj para cada puerto empleado son las que se establecen con el comando "***RCC_APB2PeriphCloCmd"***, según las líneas siguientes.

```
/* Activamos los relojes para GPIOC y GPIOB  ---------- */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
```

Figura 8.12

Se pueden resumir en una sola línea que englobe las dos, de la siguiente forma.

```
/* Activamos los relojes para GPIOC y GPIOB  -------------------------- */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC |RCC_APB2Periph_GPIOB, ENABLE);
```

Figura 8.13

8.1 PROGRAMACIÓN SYSCLK

El corazón de todo procesador es el reloj, que controla todas las funciones del mismo. En el anterior ejemplo del LED, utilizamos la gestión por defecto de ese reloj, pero en ocasiones es muy importante poder controlar nosotros esos tiempos.

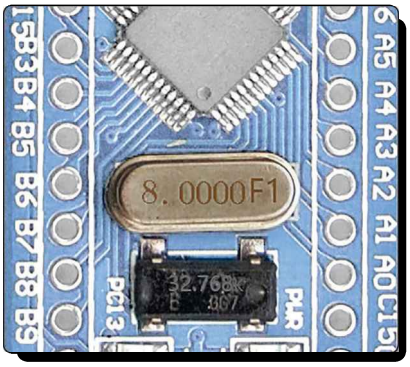


Figura 8.14

En un sistema de microcontrolador del tipo STM32, existen tres posibles fuentes primarias de reloj o frecuencias de sincronización:

- **HSE** (*High Speed External Oscillator*). Corresponde a un circuito que conecta el oscilador RC del exterior del microcontrolador. En nuestra placa de pruebas es de 8 MHz.

  Éste se conecta con un módulo multiplicador, el **PLL** (*Phase Locked-Loop*) que corresponde a un circuito en el que la frecuencia y la fase son realimentadas y modificadas; y que a su vez, se conecta con el **SYSCLK** que se utiliza para derivar la frecuencia hacia otros relojes que se emplean en los periféricos y en los temporizadores programables. Los microcontroladores STM32F100 pueden soportar una frecuencia SYSCLK máxima de 24 MHz y el resto, los STM32F1xx, como el de nuestra placa, de hasta 72 MHz.

- **HSI** (*High Speed Internal Oscillator*). Es una circuitería interna del propio microcontrolador que posee un oscilador propio interno de 8 MHz. Aunque tiene poca precisión y su frecuencia varía según se produzca el calentamiento del micro, no necesita de componentes externos.

- **LSE** (*Low Speed Oscillator*). Un oscilador de baja frecuencia a 32.768 kHz y que, generalmente, es empleado solo para periféricos como el sistema de backup cuando se emplea por el reloj de tiempo real (RTC).

Vimos en la Figura 8.7 el esquema de los relojes empleados en nuestro microcontrolador.

Aunque toda esta configuración ya la realiza internamente el Keil MDK desde el momento que configuramos la frecuencia de nuestro reloj en el apartado de "*Options for Target*", conviene saber sobre su configuración.

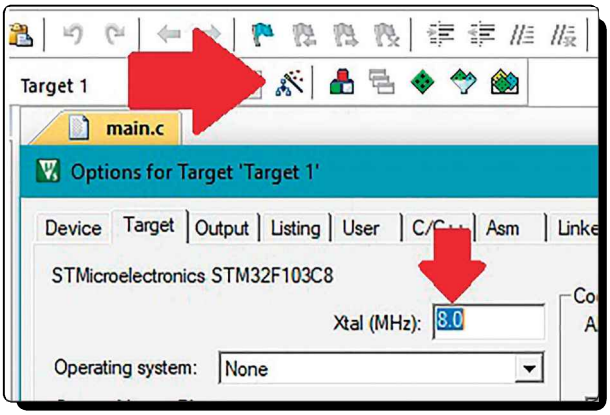


Figura 8.15

8.1.1 PLL

El propio microcontrolador tiene un módulo interno, el **PLL** (*Phase Locked Loop*) que gestiona la señal que genera el oscilador, permitiendo multiplexar las dos posibles fuentes de reloj y poder elegir utilizar el HSI o HSE como generador de la señal de reloj necesaria. Además cuenta con un **Prescaler** que nos permitirá multiplicar la frecuencia generada de entrada por un valor configurable.

Señalar también que el microcontrolador posee una salida que puede generar frecuencias de reloj, en el STM32F103 es el pin **PA8** (**MCO** – *Main Clock Output*) que puede utilizarse en dar sincronización a dispositivos externos. Programable mediante el comando "***RCC_MCO_nnnnn***":

- **RCC_MCO_NoClock**: Disable MCO.
- **RCC_MCO_SYSCLK**: La salida MCO lleva señal SYSCLK.
- **RCC_MCO_HSI**: La salida MCO lleva señal proveniente del HSI.
- **RCC_MCO_HSE**: La salida MCO lleva señal proveniente del HSE.
- **RCC_MCO_PLLCLK_Div2**: La señal de salida tiene la frecuencia dividida por 2 PLLClk.

Figura 8.16

CONFIGURACIÓN

Los pasos normales para la configuración de este módulo pasan por la configuración del **RCC** (*Real-Time Clock Control*):

1. **Activar la fuente de Reloj.**

   Como siempre necesitaremos crear la estructura que contendrá la configuración necesaria para el empleo de este módulo de acuerdo al siguiente formato:

   Mediante el comando "***RCC_HSEConfig"*** indicamos qué tipo de reloj vamos a utilizar.

```
/*  Habilitar reloj HSE  ----------*/
RCC_HSEConfig(RCC_HSE_ON);
```

Figura 8.17

Otra forma de indicarlo es mediante acceso directo a los registros como en la siguiente línea de comandos:

```
/*   Iniciamos reloj HSE --------------*/
RCC -> CR |= ((uint32_t) RCC_CR_HSEON);
```

Figura 8.18

En la siguiente línea de comando, esperamos a que el RCC esté listo:

```
/*  Esperamos que el reloj HSE este listo ----*/
HSEStartUpStatus = RCC_WaitForHSEStartUp();
```

Figura 8.19

Y si el HSE está operativo:

```
/* Si HSE esta listo  procedemos */
if (HSEStartUpStatus == SUCCESS);
```

Figura 8.20

Otra forma de programarlo, sería:

```
/* Esperando que HSE este listo procedemos ----------------------------*/
Do
{   HSEStatus = RCC -> CR & RCC_CR_HSERDY;
    StartUpCounter++;
    } while ((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));
```

Figura 8.21

En la primera línea sumamos la máscara HSE_ON al registro RCC_CR. Con esto ponemos a 1 el bit correspondiente a la activación. En el resto, esperamos a que el bit HSE_RDY se ponga a 1, aunque si pasa mucho tiempo (HSE_STARTUP_TIMEOUT) asumimos que hay un error y acabamos.

En el capítulo 15, en el que explicamos el funcionamiento del módulo RTC en los códigos de ejemplo, se incluyen ejemplos de las opciones de configuración para uno u otro reloj.

Otro ejemplo puede ser el siguiente, en el que tenemos que generar una frecuencia de 72 MHz que encienda y apague el LED de prueba de nuestra placa.

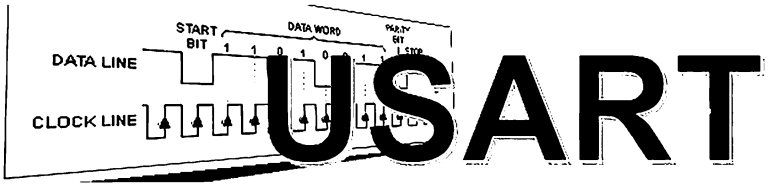
```
1   #include "stm32f10x.h"
2   #include "stm32f10x_rcc.h"
3   #include "stm32f10x_gpio.h"
4   #include "stm32f10x_tim.h"
5
6   // Lista de funciones empleadas
7   void TIM4_Config(void);
8   void TIM4_IRQHandler(void);
9   void GPIO_Config(void);
10
11  /*                    Modulo principal                          */
12  //==============================================================
13  int main(void)
14  {
15      TIM4_Config();  // Inicializamos el Timer4 y la interrupcion
16
17      GPIO_Config();  // Configuramos el GPIO del LED
18
19      while(1)
20      {
21          // No es necesario lineas de codigo adicionales
22          // ya que la funcion que se ejecuta esta configurada
23          // dentro de la funcion TIM4_IRQHandler
24      }
25  }
26
27
28  /*  Funcion que configura el GPIO del LED en PC13             */
29  //--------------------------------------------------------------
30  void GPIO_Config(void)
31  {
32      /* Inicializamos pin conectado LED PC13 ----------------*/
```

```
33      GPIO_InitTypeDef GPIO_InitStructure;
34
35      /* Iniciamos reloj para PORTC ---------------------------*/
37
38      /* Configura el pin PC13 como salida en push-pull para LED *
39      GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_13;
40      GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
41      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
42      GPIO_Init(GPIOC, &GPIO_InitStructure);
43  }
44
45  /*  Función que se ejecuta cuando se detecta el fin del periodo
46  //-------------------------------------------------------------------
47  void TIM4_Config(void)
48  {
49      /* Se crea la estructura del TIMER_4  ----------------------
50      TIM_TimeBaseInitTypeDef TIM_InitStructure;
51
52      /* Se habilita el reloj del TIMER 4 ------------------------
53      RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
54
55      /* Configuramos los parametros del TIMER4 para
56         detectar un desbordamiento cada vez que cuente
57         un periodo de tiempo que será cada 1000 veces por segundo
58      TIM_TimeBaseStructInit(&TIM_InitStructure);
59      TIM_InitStructure.TIM_Prescaler = 8000;
60      TIM_InitStructure.TIM_Period = 500;
61      TIM_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
62      TIM_TimeBaseInit(TIM4, &TIM_InitStructure);
63
64      /* Se creamos la interrupcion para TIM2 --------------------
65      TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
66
67      /*  Se inicializa el TIM2 programado  ----------------------
68      TIM_Cmd(TIM4, ENABLE);
69
70      /* Configuracion del NVIC ----------------------------------
71      /* Configurar e iniciar la interrupcion TIM4_IRQn
72      NVIC_InitTypeDef NVIC_InitStructure;
73      NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
74      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
75      NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
76      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
77      NVIC_Init(&NVIC_InitStructure);
78
79  }
80
81  /*   Funcion que se ejecuta cuando se detecta la interrupcion --
82  void TIM4_IRQHandler(void)
83  {
84          if (TIM_GetITStatus(TIM4, TIM_IT_Update) != RESET)
85          {
86              TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
87              GPIOC->ODR ^= GPIO_Pin_13;
88          }
89  }
```

Figura 8.22

9

PROGRAMACIÓN USART



En nuestro microcontrolador tenemos varios puertos RS-232 que podemos utilizar para programar el propio microcontrolador o para comunicación serie con otros periféricos o con nuestro ordenador.

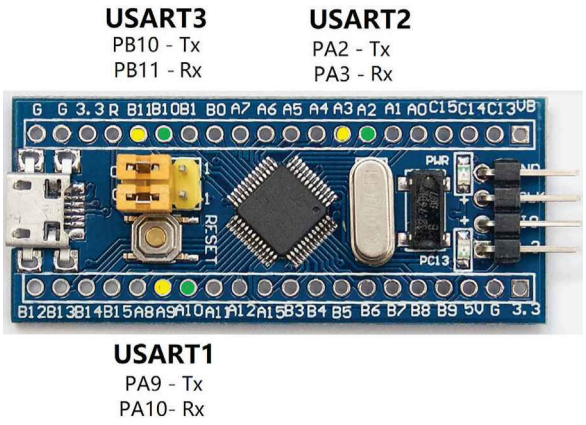


Figura 9.1

En los siguientes ejemplos veremos cómo se configuran y programan.

9.1 EJEMPLO DE CONFIGURACIÓN PUERTO USART

Ejemplo de comunicación serial

En nuestro primer código de ejemplo, conectaremos nuestro ordenador a la placa de pruebas, mediante el puerto USART1.

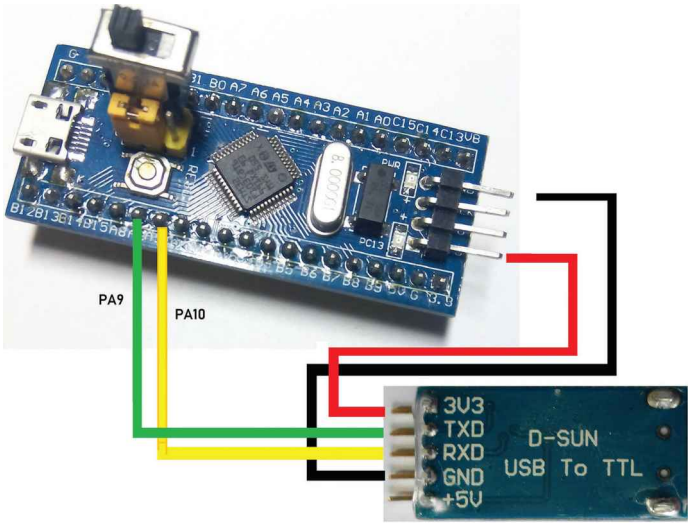


Figura 9.2

El primer paso, es añadir en la zona inicial de nuestro código, un **"#include "stm32f10x_usart.h"** para que se nos añada la librería que gestiona ese periférico junto con las otras que ya hemos empleado antes.

En nuestro ejemplo, utilizamos el puerto USART1, por lo que debemos establecer la configuración GPIO de los pines a utilizar, que serán pin **Tx** en **PA9** y pin **Rx** en **PA10**.

Toda esta configuración la encerramos en una única función, que creamos y a la que denominamos "***USART1_Config***", donde primero configuramos la estructura de los pines para que funcionen como salidas o entradas, después habilitamos el reloj correspondiente para el puerto GPIOA y el USART1, que en este caso es el mismo reloj APB2, y por último, los parámetros de configuración de la comunicación RS232.

En la estructura GPIO para los pines USART que hemos seleccionado, utilizamos la forma que ya conocemos de los ejemplos anteriores. Sin embargo, en esta ocasión, para los pines empleados en comunicaciones, el modo varía respecto a un GPIO normal; ya que el pin de transmisión o Tx en el parámetro de modo, "GPIO_Mode", se establece en "GPIO_**Mode_AF**_PP" -modo alternativo o "*Alternate Functions*"- un modo especial de determinados pines de nuestro microcontrolador, en los que estos funcionarán como entrada o salida dependiendo de la orden que se les asigne.

El modo alternativo, es un modo que pueden utilizar determinados pines asignados a periféricos como el I2C, SPI, USART, CCP, PWM, clock, ADC, etc.

El modo del otro pin, el de recepción o Rx, será ahora GPIO_Mode = GPIO_**Mode_FLOATING**, en modo flotante. En el que el pin internamente no posee tensión propia y es polarizado de acuerdo a la tensión exterior utilizada en la comunicación serial.

Con todo lo dicho anteriormente, la estructura típica del GPIO de los pines empleados en la comunicación USART, debe ser la que se muestra en la Figura 9.3.

```
1  /* Configuramos el GPIO para USART1 -----------------*/
2  GPIO_InitTypeDef GPIO_InitStructure;
3  USART_InitTypeDef USART_InitStructure;
4
5  /* Habilitamos los reloj para GPIO y el USART1 ------*/
6  RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 |
7                         RCC_APB2Periph_GPIOA, ENABLE);
8
9  /* Configuramos pin PA9 para Tx USART1 -------------------*/
10 GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_9;
11 GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF_PP;
12 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
13 GPIO_Init(GPIOA, &GPIO_InitStructure);
14
15 /* Configuramos pin PA10 para Rx USART1 -------------------*/
16 GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_10;
17 GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN_FLOATING;
18 GPIO_Init(GPIOA, &GPIO_InitStructure);
```

Figura 9.3

Seguidamente, establecemos los parámetros de comunicación serial del periférico USART, para lo que creamos también, una estructura que contendrá toda la configuración necesaria en una comunicación serie que ya conocemos: velocidad en baudios, longitud de palabra, paridad, bit de fin y si existirá control del flujo de la comunicación.

```
19
20  /* Configuramos el USART1  --------------------------*/
21  USART_InitStructure.USART_BaudRate    = 115200;
22  USART_InitStructure.USART_WordLength = USART_WordLength_8b;
23  USART_InitStructure.USART_StopBits    = USART_StopBits_1;
24  USART_InitStructure.USART_Parity      = USART_Parity_No;
25  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
26  USART_InitStructure.USART_Mode        = USART_Mode_Rx | USART_Mode_Tx;
27  USART_Init(USART1, &USART_InitStructure);
28  USART_Cmd(USART1, ENABLE);
```

Figura 9.4

En nuestro ejemplo, extraeremos toda esta función hacia un fichero aparte que contendrá todas estas líneas de código, que guardaremos y nombraremos como "**usart1.*c***". Creando otro fichero de cabecera de contenido, con la lista de funciones y que nombraremos "**usart1.*h***". De este modo estas líneas de código las podremos reutilizar en otros proyectos como si se tratara de una librería nueva para el control y gestión del puerto serie USARTx.

Por lo que, en nuestro fichero "main.c", al principio en la zona inicial de los "*include*" añadimos una línea de **"#include "usart1.h";** para que podamos llamar en nuestro código a las funciones "**USART1_Config**()" que hemos creado.

Esta estructura de ficheros, en nuestro proyecto, deberá aparecer tal como se muestra en la ventana de la izquierda, Figura 9.5, donde el Keil nos la muestra.

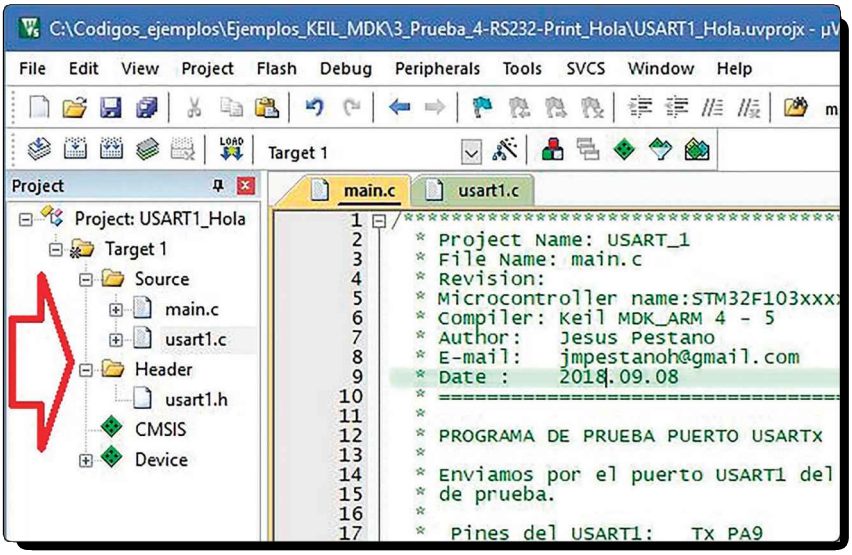


Figura 9.5

Reproducimos a continuación el contenido del fichero "**usart1.c**":

```
#include "usart1.h"

#include <stdarg.h>
#include <stdio.h>

/*           Funcion que configura el USART1            */
//-----------------------------------------------------------
void USART1_Config(void)
{
   /* Configuracion del GPIO para el USART1 ------------------*/
   GPIO_InitTypeDef GPIO_InitStructure;

   /* Se habilitamos los reloj para GPIO y el USART1 ---------*/
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 |
                          RCC_APB2Periph_GPIOA, ENABLE);

   /* Configuracion del pin PA9 para Tx en USART1 -------------*/
   GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_9;
   GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_Init(GPIOA, &GPIO_InitStructure);

   /* Configuracion del pin PA10 para Rx en USART1 -------------*/
   GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_10;
   GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN_FLOATING;
   GPIO_Init(GPIOA, &GPIO_InitStructure);

   /* Configuracion parametros comunicación del USART1  ------*/
   USART_InitTypeDef USART_InitStructure;
   USART_InitStructure.USART_BaudRate = 115200;
   USART_InitStructure.USART_WordLength = USART_WordLength_8b;
   USART_InitStructure.USART_StopBits = USART_StopBits_1;
   USART_InitStructure.USART_Parity = USART_Parity_No;
   USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
   USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
   USART_Init(USART1, &USART_InitStructure);
   USART_Cmd(USART1, ENABLE);
}


/* Funncion que nos permite utilizar el comando 'printf'     */
//-----------------------------------------------------------
int fputc(int ch, FILE *f)
{
     /* Envia dato */
     USART_SendData(USART1, (unsigned char) ch);

     while( USART_GetFlagStatus(USART1,USART_FLAG_TC)!= SET);
     return (ch);
}
```

Figura 9.6

También, el contenido del fichero "**usart1.h**":

```
#ifndef __USART1_H
#define __USART1_H

#include "stm32f10x.h"

void USART1_Config(void);
```

Figura 9.7

El contenido de nuestro módulo principal, el fichero "**main.c**":

```
1  #include "stm32f10x.h"
2  #include "stm32f10x_rcc.h"
3  #include "stm32f10x_gpio.h"
4  #include "stm32f10x_usart.h"
5
6  // LLama al modulo externo de configuracion del puerto USART1
7  #include "usart1.h"
8  #include <stdio.h>
9
10
11 void Delay_ms(unsigned int nCount);
12
13 /*     Modulo principal ---------------------------------- */
14 int main(void)
15 {
16       USART1_Config();   //LLama a la funcion que configura el USART1
17
18       for(;;)
19       {
20       printf("Esto es un texto de prueba por printf\r\n");
21       Delay_ms(200);
22       }
23 }
24
25 // Modulo que genera un retardo (ms)
26 //-------------------------------------
27 void Delay_ms(unsigned int nCount){
28             unsigned int i, j;
29             for(i = 0; i < nCount; i++)
30             {
31                         for(j = 0; j < 0x2AFF; j++){;}
32             }
33 }
```

Figura 9.8

9.2 EJEMPLO CON OTRO PUERTO USART

Como ya hemos indicado, nuestro microcontrolador STM32F103 posee tres puertos seriales USART.

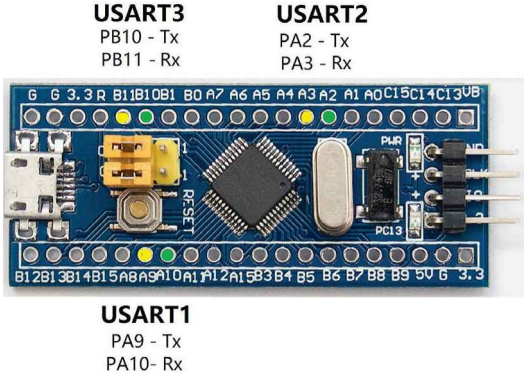


Figura 9.1

En el siguiente ejemplo, veremos cómo configurar otro puerto del STM32F103; por ejemplo el USART2 para reenviar por él mismo los mensajes que se reciben.

Mediante un programa de comunicaciones en nuestro ordenador, enviaremos con una configuración serial de 115200 baudios, cadenas de texto que se enviarán al micro STM32, señalando el fin del mensaje mediante la inclusión, al final del texto enviado, del código '\n' de fin de línea o valor ASCII (hex 0A).

Código del fichero "**usart2.c**"

```
#include "stm32f10x.h"

#include "usart2.h"

/*          Funcion que configura el USART2     */
//-----------------------------------------------
void USART2_Config()
{
      /* Creamos la estructura para el GPIO y USART ----------------*/
      GPIO_InitTypeDef GPIO_InitStruct;
      USART_InitTypeDef USART_InitStruct;

      /* Activamos el reloj para GPIOA   ----------------------------*/
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

      /* Configuramos Pin PA2 para Tx del USART2  -----------------*/
      GPIO_InitStruct.GPIO_Pin    = GPIO_Pin_2;
      GPIO_InitStruct.GPIO_Mode   = GPIO_Mode_AF_PP;
      GPIO_InitStruct.GPIO_Speed  = GPIO_Speed_2MHz;
      GPIO_Init(GPIOA, &GPIO_InitStruct);

      /* Configuramos el Pin PA3 para Rx del USART2 ---------------*/
      GPIO_InitStruct.GPIO_Pin    = GPIO_Pin_3;
      GPIO_InitStruct.GPIO_Mode   = GPIO_Mode_IN_FLOATING;
      GPIO_InitStruct.GPIO_Speed  = GPIO_Speed_2MHz;
      GPIO_Init(GPIOA, &GPIO_InitStruct);

      /* Configuramos el USART2 -----------------------------------*/
      /* Activamos el reloj para USART2   */
      RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);

      /* Velocidad Baudios: 115200
         Longitud de palabra: 8 bits
         Bit de Stop: 1
         Paridad: Ninguna
         Control de flujo: Ninguno  */

      USART_InitStruct.USART_BaudRate            = 115200;
      USART_InitStruct.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
      USART_InitStruct.USART_Mode                = USART_Mode_Rx | USART_Mode_Tx;
      USART_InitStruct.USART_Parity              = USART_Parity_No;
      USART_InitStruct.USART_StopBits            = USART_StopBits_1;
      USART_InitStruct.USART_WordLength          = USART_WordLength_8b;
      USART_Init(USART2, &USART_InitStruct);
      USART_Cmd(USART2, ENABLE);
}

/*     Funcion que envia un caracter por USART2       */
//-----------------------------------------------------
void USART2_PutChar(char c)
{
      // Esperamos hasta que el registro de datos transmitidos este lleno
      while (!USART_GetFlagStatus(USART2, USART_FLAG_TXE));

```

```
55      // Enviamos los caracteres por el USART2
56      USART_SendData(USART2, c);
57 }
58
59 /*     Funcion que envia una cadena 'String' por USART2    */
60 //-----------------------------------------------------------
61 void USART2_PutString(char *s)
62 {
63      // Enviamos una cadena por el puerto USART2
64      while (*s)
65      {
66          // Llama a la función 'USART2_PutChar() para ir enviado
67          // carácter a carácter el contenido de la cadena
68          USART2_PutChar(*s++);
69      }
70 }
71
72 /*    Funcion que lee un caracter recibido en el puerto USART2    */
73 //-----------------------------------------------------------------
74 uint16_t USART2_GetChar()
75 {
76      // Esperamos a que se reciba una caracter en USART2
77      while (!USART_GetFlagStatus(USART2, USART_FLAG_RXNE));
78
79      // Devuelve el caracter recibido
80      return USART_ReceiveData(USART2);
81 }
```

Figura 9.9

Es importante señalar que, aunque las líneas de código para la configuración del GPIO y el USART son prácticamente iguales para el USART1 del primer ejemplo, en el del segundo ejemplo, el USART2, además de la diferencia de pines a utilizar como Tx y Rx, también existe la diferencia de que el reloj que controla este USART2, varía. Para el **USART1** configurábamos el **APB2** y para el **USART2** y el **USART3** es el **APB1.** Quedando igual el reloj para el GPIOA.

```
/*   Activamos el reloj para USART1 y GPIOA   */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);

/*   Activamos el reloj para USART2   */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);
```

Figura 9.10

En este ejemplo, se observará también que, tanto para enviar como para recibir los caracteres por el puerto, se utiliza la comprobación de unas 'banderas' o registros, mediante el comando "*USART_**GetFlagstatus***(USARTx, USART_Flag_XXXX)".

Existen unos registros internos del módulo USART, donde se almacenan los datos que se reciben o se envían, como son TXE y el RXNE, que en las librerías del STM32 pueden configurarse en nuestro código, como banderas de estado de la comunicación y que son nombradas de la siguiente forma:

- **USART_FLAG_TXE** Nos informa si el registro de transmisión contiene algún dato.
- **USART_FLAG_RXNE** Informa si el registro de recepción contiene algún dato.

Mediante el empleo de líneas de código como la mostrada en la Figura 9.11, se pueden comprobar estos y saber cuándo se ha recibido un dato.

```
1  /*  Funcion que lee un ca_cter recibido en el puerto USART1   */
2  uint16_t USART1_GetChar()
3  {
4        // Esperamos a que se reciba una caracter en USART1
5        while (!USART_GetFlagStatus(USART1, USART_FLAG_RXNE));
6
7        // También se puede escribir así
8        // while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
9
10       // Devuelve el caracter recibido
11       return USART1->DR & 0x1FF;
12 }
```

Figura 9.11

Con el comando del lenguaje C++ *'while'*, creamos un tiempo de espera en nuestro código, en el que el sistema se queda comprobando constantemente el estado del registro o su 'bandera' **USART_FLAG_RXNE,** que se mantendrá en valor '0' mientras no se reciba ningún dato, pasando a valor '1' en el momento que se reciba alguno.

Otro registro del módulo USARTx es el registro **DR** (*Data Register*, Registro de Datos) donde se reciben y se envían los datos. Si se lee ese registro, se obtiene el último byte recibido. Si se escribe en él, se transmite el byte escrito. Teniendo en cuenta que dicho registro es de 32 bits, aunque en la comunicación serial solo se utilizan 9 bits para la transmisión, en nuestras líneas de código, aplicamos la máscara '0x1FF' a los datos entrantes para asegurarnos de no recibir basura adicional. Tan pronto como se ha leído el dato, el USART restablece automáticamente el registro RXNE a '0' y con ello su bandera.

Ahora la parte difícil es saber si se pueden escribir o leer nuevos datos en el registro DR.

Para transmitir un dato en nuestro código, lo siguiente será esperar a que el registro de transmisión esté vacío y a continuación escribir el dato. Mediante la comprobación del estado de la bandera "**USART_FLAG_TXNE**" con el comando "*while*" se produce esa espera, como se muestra en las líneas de la Figura 9.12.

```
1  /*     Funcion que envia un caracter por USART1                    */
2  //-------------------------------------------------------------------
3  void USART1_PutChar(char c)
4  {
5        // Esperamos hasta que el registro de datos transmitidos este lleno
6        while (!USART_GetFlagStatus(USART1, USART_FLAG_TXE));
7
8        // Tambien se puede escribir así
9        // while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
10
11       // Enviamos el caracter, escribiendo directamene en el registro
12       USART1->DR = c;
13
14       // o tambien con este comando
15       USART_SendData(USART1, c);
16 }
```

Figura 9.12

Una vez comprobado que el registro está vacío, se coloca directamente el dato en el mismo, mediante la instrucción "**USART1->DR**=*dato*" o empleando el comando **USART_SendData** (USART1, *dato*)".

Reproducimos el contenido completo del fichero principal "main.c".

```
1  #include "stm32f10x.h"
2  #include "stm32f10x_rcc.h"
3  #include "stm32f10x_gpio.h"
4  #include "stm32f10x_usart.h"
5
6  // Llamada a la libreria USART2 externa
7  #include "usart2.h"
8
9  /* Libreria auxiliar del Keil para el uso de la
10    instruccion 'memset'                                          */
11 #include <string.h>
12
13 /* Creamos la variable 'MESSAGE_SIZE' con el valor 16 como
14    maximo de caracteres a recibir en el buffer de comunicación  */
15 #define MESSAGE_SIZE    16
16
17 /* Creamos la variable indexada 'message' con un maximo de
18    de caracteres indicado en la variable 'MESSAGE_SIZE'         */
19 char message[MESSAGE_SIZE];
20
21 int i = 0;  // Cramos la variable numerica 'i' para el contador
22
23 int main(void)
24 {
25
26   USART2_Config();    // Inicializamos el USART2
27
28   while (1)
29   {
30     // Lee un caracter recibido en el USART2 y lo guarda
31     // en la variable 'c'
32     char c = USART2_GetChar();
33
```

```
34      // Comprueba si no se ha recibido el caracter de fin de linea
35      if (c != '\n')
36      {
37         // Concatenamos a la variable 'message' cada uno de los
38         // caracteres recibidos 'c' hasta el maximo indicado
39         // descontandose cada uno hasta llegar a 0
40         if (i < MESSAGE_SIZE - 1)
41         {
42              message[i] = c;
43              i++;
44         }
45         else
46         {
47              message[i] = c;
48              i = 0;
49         }
50      }
51      // Si se ha recibido el caracter '\n' de fin de linea
52      else
53      {
54          // Se reproduce por el mismo puerto USART2 el mensaje recibido
55          USART2_PutString(message);
56          USART2_PutChar('\n'); // Se envia el caracter de fin de linea '\n
57
58          // Reseteamos el 'buffer' con el borrado mediante el comando
59          // 'memset' de la variable indexada 'message[]'
60              memset(&message[0], 0, sizeof(message));
61              i = 0;
62              }
63        }
64  }
```

Figura 9.13

9.3 EJEMPLO REMAPEO DE PUERTO USART

Otra utilidad importante que poseen los microcontroladores STM32, es la posibilidad de remapear la función de determinados pines para su uso por varios periféricos. De modo que podamos utilizar, por ejemplo, un determinado periférico con otros pines para los que está configurado de forma estándar; se puede utilizar el módulo USART1, que por defecto utiliza los pines Tx en PA9 y Rx en PA10, remapearlo y utilizar los pines PB6 para Tx y PB7 para Rx; para el USART2, también es posible reasignar sus pines tal como se indican en las tablas de la Figura 9.14, del manual de características datasheet del dispositivo.

Table 43. USART2 remapping

Alternate functions	USART2_REMAP = 0	USART2_REMAP = 1(1)
USART2_CTS	PA0	PD3
USART2_RTS	PA1	PD4
USART2_TX	PA2	PD5
USART2_RX	PA3	PD6
USART2_CK	PA4	PD7

1. Remap available only for 100-pin and 144-pin packages.

Table 44. USART1 remapping

Alternate function	USART1_REMAP = 0	USART1_REMAP = 1
USART1_TX	PA9	PB6
USART1_RX	PA10	PB7

Figura 9.14

En nuestro siguiente ejemplo, utilizaremos esta función de remapeo de los pines del puerto **USART1** en otros pines como **PB6** y **PB7**.

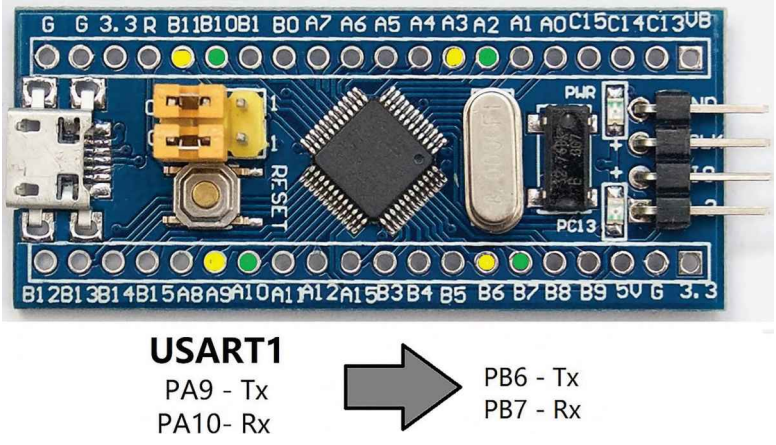


Figura 9.15

Contenido del fichero "**usartM.c**"

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_usart.h"

// Llamada al fich___ __ cabecera
#include "usartM.h"

#include <stdarg.h>          // Libreria para usar 'printf'

/* Funcion que configura los parametros del USART             */
//-------------------------------------------------------------
void USARTremap_Config(void)
{
      /* Creamos la estructura para el GPIO   ---------------*/
      GPIO_InitTypeDef GPIO_StructureInit;
      USART_InitTypeDef USART_StructureInit;

      /* Activamos el reloj para GPIOB, AFIO y USART1 -------*/
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB
                             RCC_APB2Periph_AFIO  |
                             RCC_APB2Periph_USART1, ENABLE);

      /* Iniciamos el remapeo de pines del USART1 para su
         uso con PB6 y PB7       ----------------------------*/
      GPIO_PinRemapConfig(GPIO_Remap_USART1, ENABLE);

      /* Configuramos el GPIO para el USART1 */
      /* Configuramos Pin PB6 para Tx del USART1 ------------*/
      GPIO_StructureInit.GPIO_Pin   = GPIO_Pin_6;
      GPIO_StructureInit.GPIO_Mode  = GPIO_Mode_AF_PP;
      GPIO_StructureInit.GPIO_Speed = GPIO_Speed_2MHz;
      GPIO_Init(GPIOB, &GPIO_StructureInit);

      /* Configuramos el Pin PB7 para Rx del USART1 ---------*/
      GPIO_StructureInit.GPIO_Pin   = GPIO_Pin_7;
      GPIO_StructureInit.GPIO_Mode  = GPIO_Mode_IN_FLOATING;
      GPIO_Init(GPIOB, &GPIO_StructureInit);

      /* Configuramos el USART1 -----------------------------*/
      /* Velocidad Baudios: 115200
         Longitud de palabra: 8 bits
         Bit de Stop: 1
         Paridad: Ninguna
         Control de flujo: Ninguna  */
      USART_StructureInit.USART_BaudRate   = 115200;
      USART_StructureInit.USART_WordLength = USART_WordLength_8b;
      USART_StructureInit.USART_StopBits   = USART_StopBits_1;
      USART_StructureInit.USART_Parity     = USART_Parity_No ;
      USART_StructureInit.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;
      USART_StructureInit.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
      USART_Init(USART1, &USART_StructureInit);

      /* Activamos el USART1 --------------------------------*/
      USART_Cmd(USART1, ENABLE);
}

/* Funcion que confi___ __ empleo de 'fputc' */
// Para enviar un byte por el pin de salida del USART1
int fputc(int ch, FILE *f)
{
      /* Envia dato */
      USART_SendData(USART1, (unsigned char) ch);

      while( USART_GetFlagStatus(USART1,USART_FLAG_TC)!= SET);
      return (ch);
}
```

Figura 9.16

Como podemos ver, en nuestro código de ejemplo, para redireccionar los pines del puerto USART1 estándar hacia los otros pines, se añaden dos comandos nuevos en la configuración del GPIO:

```
/* Creamos la estructura para el GPIO    */
GPIO_InitTypeDef GPIO_StructureInit;
USART_InitTypeDef USART_StructureInit;

/* Activamos el reloj para AFIO, USART1 y GPIOB   */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,   ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,  ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);

/* Iniciamos el remapeo de pines del USART1 para su uso con PB6 y PB7 */
GPIO_PinRemapConfig(GPIO_Remap_USART1, ENABLE);

/* Configuramos el GPIO para el USART1 */
/* Configuramos Pin PB6 para Tx del USART1 en modo push-pull */
GPIO_StructureInit.GPIO_Pin = GPIO_Pin_6;
GPIO_StructureInit.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_StructureInit.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOB, &GPIO_StructureInit);

/* Configuramos el Pin PB7 para Rx del USART1 en modo floating */
GPIO_StructureInit.GPIO_Pin = GPIO_Pin_7;
GPIO_StructureInit.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOB, &GPIO_StructureInit);
```

Figura 9.17

RCC_APB2Periph_AFIO, activa el redireccionamiento del reloj que controla el módulo USART1, ya que ahora debe controlar este periférico en otros pines. La otra nueva línea con el comando "**GPIO_PinRemapConfig(GPIO_Remap_USART1**, ENABLE)" activa el redireccionamiento de los pines.

A continuación, creamos la estructura normal con la configuración de GPIO y USART, pero especificando ahora que el pin Tx sea el **PB6** y el Rx el pin **PB7**.

Hay que señalar también, que esta función de remapeo de pines de un periférico USART hacia otros pines, no es una opción que funcione con cualquier pin de nuestro microcontrolador y que solo es factible en los pines que poseen esa opción, que se establecen en las características del propio microcontrolador.

Si nos fijamos, realmente no se está direccionando los pines simplemente; ya que los pines PB6 y PB7 se corresponden con los pines para el puerto I2C, sino que lo que verdaderamente se está redireccionando es el propio periférico.

Dentro de la librería "**stm32f10x_gpio.c**" podemos encontrar una lista de funciones que permiten utilizar esta opción con otros periféricos y de la que reproducimos parte en la Figura 9.18.

```
Asignación de funciones SPI1 alternativo: GPIO_Remap_SPI1
Asignación de funciones I2C1 alternativo: GPIO_Remap_I2C1
Asignación de funciones USART1 alternativo: GPIO_Remap_USART1
Asignación de funciones USART2 alternativo: GPIO_Remap_USART2
Asignación de funciones alternativo parcial USART3: GPIO_PartialRemap_USART3
Asignación de funciones USART3 completa alternativo: GPIO_FullRemap_USART3
Asignación de funciones alternativo parcial TIM1: GPIO_PartialRemap_TIM1
Asignación de funciones TIM1 completa alternativo: GPIO_FullRemap_TIM1
Asignación de funciones TIM2 parcial1 alternativo: GPIO_PartialRemap1_TIM2
Asignación de funciones TIM2 Partial2 alternativo: GPIO_PartialRemap2_TIM2
Asignación de funciones TIM2 completa alternativo: GPIO_FullRemap_TIM2
Asignación de funciones alternativo parcial TIM3: GPIO_PartialRemap_TIM3
Asignación de funciones TIM3 completa alternativo: GPIO_FullRemap_TIM3
GPIO_Remap_TIM4: TIM4 Asignación de funciones alternativo
Asignación de funciones CAN1 alternativo: GPIO_Remap1_CAN1
Asignación de funciones CAN1 alternativo: GPIO_Remap2_CAN1
GPIO_Remap_PD01: PD01 Asignación de funciones alternativo
```

Figura 9.18

9.4 EJEMPLO DE INTERRUPCIONES DEL PUERTO USART

Como en casi todos los procesadores, el microcontrolador STM32 posee también un sistema de interrupciones y excepciones que permite detectar cuando se produce un evento determinado y realizar, por ello, un proceso programado, liberando al resto de procesos que se estén ejecutando en ese momento de tener que estar atendiendo cuando se produce esa interrupción.

Conviene saber que el módulo USART posee también llamadas especiales, que más adelante pasaremos a explicar en el apartado siguiente 10.2 en el que exponemos el funcionamiento de las Interrupciones.

También indicar, que este periférico, posee varios canales distintos que se pueden utilizar en nuestra programación como interrupciones.

En el apartado 10.2 de interrupciones, describimos un ejemplo de empleo en el puerto USART.

10

PROGRAMACIÓN DE INTERRUPCIONES (NVIC)



El **NVIC** (*Nested Vectoried Interrupt Controller*) es el módulo encargado del control de las interrupciones en nuestro microcontrolador. Realiza las funciones de activar y desactivar las interrupciones, designar la prioridad y almacenar los datos individuales de las interrupciones que se generen de forma anidada.

Las interrupciones son un sistema por el cual el microcontrolador recibe notificaciones de eventos críticos o señales específicas que, cuando se detectan, es posible cambiar la ejecución del programa a una parte de la programación distinta. Mediante el empleo de las interrupciones se permite que el programa no se quede esperando ni se detenga el proceso que se está ejecutando en ese momento.

Como siempre, para configurar todas las opciones de dicho módulo deberemos crear una estructura que las contenga, de acuerdo al siguiente formato:

```
1  /* Creamos la estructura para NVIC ---------------------------------*/
2  NVIC_InitTypeDef NVIC_InitStructure;
3
4  /*  Configuracion del Grupo 0 de prioridad ------------------------*/
5  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
6
7  /* Configuracion de los parametros para la Interrupcion UASARTx --*/
8  NVIC_InitStructure.NVIC_IRQChannel                    =USART1_IRQn;
9  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
10 NVIC_InitStructure.NVIC_IRQChannelSubPriority  = 0;
11 NVIC_InitStructure.NVIC_IRQChannelCmd          = ENABLE;
12
13 /*  Iniciamos el NVIC   ----------------------------------------------*/
14 NVIC_Init(&NVIC_InitStructure);
```

Figura 10.1

PRIORIDAD DE LA INTERRUPCIÓN

Para configurar las prioridades en las interrupciones, se deben establecer una serie de parámetros.

En las librerías CMSIS se establecen las opciones de prioridad para las interrupciones agrupados en dos partes: 16 niveles distintos de prioridades y un segundo grupo de subprioridades de 4 bits. Que se establecen en función de que dos o más interrupciones estén pendientes.

Para establecer la prioridad de la interrupción, en los sistemas ARM Cortex-M se declaran primeramente por su grupo, en el que existen 5 posibilidades, que se configuran mediante el comando "***NVIC_PriotityGroup_x***", y mediante los comandos (**NVIC_IRQChannelPreemptionPriority** y **NVIC_IRQChannelSubPriority**), configuran los subgrupos posibles, cuyos valores dependerán del grupo que previamente se haya escogido

NVIC_PriorityGroup	NVIC_IRQChannel PreemptionPriority	NVIC IROChannel SubPriority	Description
NVIC_PriorityGroup_0	0	0-15	0 bits for pre-emption priority 4 bits for subpriority
NVIC_PriorityGroup_1	0-1	0-7	1 bits for pre-emption priority 3 bits for subpriority
NVIC_PriorityGroup_2	0-3	0-3	2 bits for pre-emption priority 2 bits for subpriority
NVIC_PriorityGroup_3	0-7	0-1	3 bits for pre-emption priority 1 bits for subpriority
NVIC_PriorityGroup_4	0-15	0	4 bits for pre-emption priority 0 bits for subpriority

Figura 10.2

Por ejemplo, según nuestra tabla de la Figura 10.2, si escogemos un "***NVIC_PriorityGroup_0***", solo podemos tener en nuestra interrupción una "***NVIC_IRQChannelPreemptionPriotity*** = 0" y asignar diferentes subprioridades entre 0 o 15. Esto significa, que nuestra interrupción solo será interrumpida por otra que contenga una prioridad superior, si no seguirá su proceso.

Por tanto, en este sistema de prioridades, el que tenga mayor prioridad –*PreemptionPriotity*– se ejecutará en primer lugar. Si hay dos interrupciones que tuvieran el mismo valor, se ejecutará la que tenga el mayor valor en la prioridad secundaria –*SubPriority*–. Si aun así, las dos interrupciones fueron programadas con los mismos números de prioridad primaria y secundaria, entonces el microcontrolador atenderá a la que ocurra primero, siendo por último concepto de prioridad el orden en que se produzcan.

En el código correspondiente que se muestra en la Figura 10.1 anterior, donde creamos la estructura NVIC necesaria sobre un puerto **USART1**, hemos establecido una ***PreemtionPriority*** = 0 y una ***SubPriority*** = 0.

En el siguiente ejemplo, de la Figura 10.3, vamos a establecer tres configuraciones para interrupciones diferentes, con prioridades distintas:

```
1  /* Creamos la estructura para NVIC ------------------------------*/
2  NVIC_InitTypeDef NVIC_InitStructure;
3
4  /*  Configuracion del Grupo 0 de prioridad ----------------------*/
5  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
6
7  /* Configuracion de los parametros para la Interrupcion UASARTx --*/
8  NVIC_InitStructure.NVIC_IRQChannel                   = USART1_IRQn;
9  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
10 NVIC_InitStructure.NVIC_IRQChannelSubPriority  = 0;
11 NVIC_InitStructure.NVIC_IRQChannelCmd          = ENABLE;
12
13 /* Configuracion de los parametros para la Interrupcion EXTI_0 --*/
14 NVIC_InitStructure.NVIC_IRQChannel                   = EXTI0_IRQn;
15 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
16 NVIC_InitStructure.NVIC_IRQChannelSubPriority  = 0;
17 NVIC_InitStructure.NVIC_IRQChannelCmd          = ENABLE;
18
19 /* Configuracion de los parametros para la Interrupcion EXTI_1 --*/
20 NVIC_InitStructure.NVIC_IRQChannel                   = EXTI1_IRQn;
21 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
22 NVIC_InitStructure.NVIC_IRQChannelSubPriority  = 1;
23 NVIC_InitStructure.NVIC_IRQChannelCmd          = ENABLE;
24
25 /*  Iniciamos el NVIC   -----------------------------------------*/
26 NVIC_Init(&NVIC_InitStructure);
```

Figura 10.3

Donde hemos establecido el **Grupo 1** con el comando "***NVIC_PriorityGroup_1***" y diferentes interrupciones con diferentes prioridades. Creando las interrupciones **EXTI0_IRQn** y **EXTI1_IRQn**, donde ambas pueden interrumpir la ejecución del código correspondiente a la de **USART1_IRQn** y a su vez, cualquiera de ellas –**EXTI0_IRQn** y **EXT1_IRQn**-, pueden interrumpirse una a la otra cuando ocurre alguna de ellas, y, en el caso de que las dos ocurran al mismo tiempo, aquella que tiene el "**NVIC_IRQChannelSubPriority = 1**" se ejecutará primero al tener un "**SubPriority**" superior.

ESPECIFICAR LA IRQ

Las **IRQ** (*Interrupt Request*) son señales especiales que recibe el procesador central para avisarle de que debe interrumpir el curso de ejecución actual y pasar el control del código para tratar esa llamada,

En el archivo "stm32f1xx.h" que suministra la casa STMicroelectronics y que podemos descargar de internet, enumera en su apartado "typedef enum IRQn", una lista de todas las interrupciones posibles en este microcontrolador y en toda la familia de microcontroladores Cortex-M3.

Mediante el comando "**NVIC_IRQChannel"**, podemos especificar casi todos los periféricos internos de nuestro microcontrolador y su canal de interrupciones que se comunica con la CPU central para avisarle de que se ha producido un evento que debe atender. En la Figura 10.4 podemos ver los IRQ que pueden configurarse en nuestro microcontrolador.

NVIC_IRQChannel	
WWDG_IRQChannel	DMAChannel1_IRQChannel
PVD_IRQChannel	DMAChannel2_IRQChannel
TAMPER_IRQChannel	DMAChannel3_IRQChannel
RTC_IRQChannel	DMAChannel4_IRQChannel
FlashItf_IRQChannel	DMAChannel5_IRQChannel
RCC_IRQChannel	DMAChannel6_IRQChannel
EXTI0_IRQChannel	DMAChannel7_IRQChannel
EXTI1_IRQChannel	ADC_IRQChannel
EXTI2_IRQChannel	USB_HP_CANTX_IRQChannel
EXTI3_IRQChannel	USB_LP_CAN_RX0_IRQChannel
EXTI4_IRQChannel	CAN_RX1_IRQChannel
	CAN_SCE_IRQChannel

Figura 10.4

Además, otro aspecto de la configuración y programación de interrupciones es que podemos establecer líneas de código específico, que se ejecuten cuando se detecten y agruparlas en funciones asociadas a esa interrupción, que luego pasarán a ejecutarse automáticamente en el momento en que se detecte que se ha producido la interrupción programada.

Las librerías CMSIS, también nos ofrecen funciones específicas para cada evento que podemos utilizar para encerrar en ellas líneas de código que se ejecutarán una vez el sistema detecte que se ha producido y aparte del resto de procesos programados, como por ejemplo:

- *void **HMI_Handler()*** {....} Encierra las líneas de instrucciones que se ejecutarán en el caso de que se produzca una excepción NMI (*Non Maskable Interrup*).
- *void **TIM2_Handler()***{...} Función que se ejecutará cuando se detecte un desbordamiento del Timer2.
- *void **EXTIO_Handler***() {...} Función que se ejecutará cuando se detecte la interrupción EXTI0 en el pin determinado.

10.1 EJEMPLO DE CONTROL DE INTERRUPCIÓN EXTI_0

El módulo **EXTI** (*External interrupt/event controller*) tiene como principal función, la detección de cambios de estado de la señal lógica y puede programarse para que se detecten flancos de subida, de bajada o ambos. En algunos pines de nuestro microcontrolador, a los que se les puede configurar interrupciones, que una vez detectadas, se les asigna determinados procesos que el sistema atenderá independientemente del proceso principal que se esté realizando.

Los microcontroladores STM32 poseen 20 posibles interrupciones externas, enumeradas del 0 al 19, y a las que cualquier pin de nuestro microcontrolador pueden ser asignadas. Casi todos los pines que pueden ser programados por el GPIO, están reasignados a las 16 líneas externas de interrupciones. No obstante, la asignación del número de pin se corresponderá con la numeración de la interrupción EXTI_(n), por ejemplo los pines 0 de los puertos A, B, C y D están asociados a la EXTI_0, y todos los pines 1 de los puertos lo están a la EXTI_1.

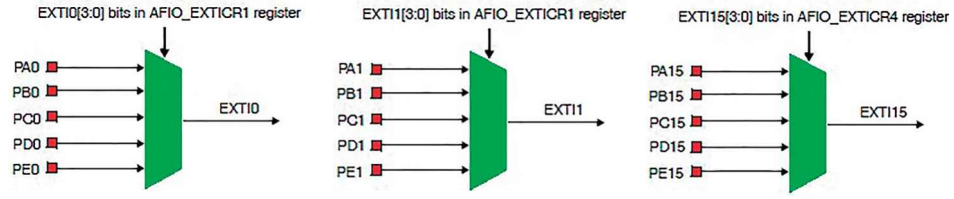


Figura 10.5

Además, existen cuatro líneas especiales más que están conectados a periféricos internos específicos para una gestión de interrupciones: como la **EXTI_16**, conectada a la salida PVD (*Programable Voltage Detector*) o control programable de alimentación del microcontrolador; la **EXTI_17**, conectada al módulo de eventos RTC (*Real Time clock*) o reloj de tiempo interno; la **EXTI_18** conectada a la señal de activación o "*Wakeup*" del puerto USB y la **EXTI_19** que para aquellos que posean dispositivo Ethernet está conectado a su señal de inicio (*Wakeup*).

En el siguiente código de ejemplo, controlaremos si se ha pulsado un botón conectado al pin **PA0** donde estará configurada la interrupción externa **EXTI_0** y que hará que se encienda un led que conectaremos al pin **PB1**.

Crearemos un código que, cuando detecte ese evento, encenderá y apagará el LED varias veces con un intervalo que también estableceremos.

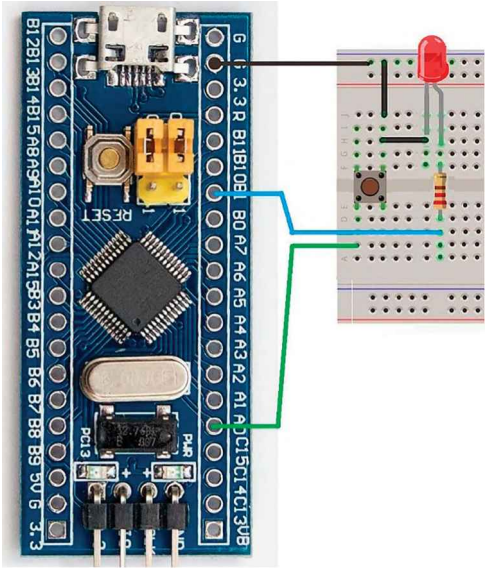


Figura 10.6

Código del fichero "***main.c***" de nuestro ejemplo.

```
#include "stm32f10x.h"
#include 'stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_exti.h"

// Llama a la libreria que contiene la funcion de retardo
#include "delay.h"

// Listado de funciones
void Configurar_LED(void);
void Configurar_BOTON(void);
void Config_Int(void);
void espera(unsigned int nCount);
unsigned char bandera = 0;

/*            Modulo principal                */
//-----------------------------------------------
int main(void){
      DelayInit();          // Inicializa la funcion Delay

      Configurar_LED();    // Configura pin del LED
      Configurar_BOTON();  // Configura pin del BOTON
      Config_Int();        //Configura EXTI Line0 (Pin PA0) en modo interrupcion

      while (1){

            // Blink LED en PC13 para control del programa
            GPIOC->BRR = GPIO_Pin_13;
            DelayMs(1000);
            GPIOC->BSRR = GPIO_Pin_13;
            DelayMs(1000);
      }
}

/* Funcion configura el GPIO para el LED                    */
//-------------------------------------------------------------
void Configurar_LED(void)
{
      /*  Creamos la estructura GPIO ----------------------*/
      GPIO_InitTypeDef GPIO_InitStructure;

      /*  Habilitamos el reloj para GPIOB  ----------------*/
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB |
                             RCC_APB2Periph_GPIOC, ENABLE);

      /* Configuramos PB1 para salida de LED --------------*/
      GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_1;
      GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_Init(GPIOB, &GPIO_InitStructure);

      /* Configuramos PC13 de salida para el LED 2----------*/
      GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_13;
      GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
      GPIO_Init(GPIOC, &GPIO_InitStructure);
}

/*   Funcion que configura GPIO para el BOTON            */
//-----------------------------------------------------------
void Configurar_BOTON(void)
{
      /* Creamos la estructrua GPIO ----------------------*/
      GPIO_InitTypeDef GPIO_InitStructure;

      /* Activamos el reloj para GPIOA   ----------------*/
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

      /* Configura pin PA0 como entrada ------------------*/
      GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_0;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
      GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

```
74
75 /*   Funcion que configura la interrupcion por PA0          */
76 //-------------------------------------------------------------
77 void Config Int(void)
78 {
79       /* Creamos la estructura para NVIC y EXTI -------------*/
80       NVIC_InitTypeDef NVIC_InitStructure;
81       EXTI_InitTypeDef EXTI_InitStructure;
82
83       //Configura la interrupcion EXTI linea 0 en PA0 -------*/
84       EXTI_InitStructure.EXTI_Line    = EXTI_Line0;
85       EXTI_InitStructure.EXTI_Mode    = EXTI_Mode_Interrupt;
86       EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
87       EXTI_InitStructure.EXTI_LineCmd = ENABLE;
88       EXTI_Init(&EXTI_InitStructure);
89
90       // Habilita NVIC para EXT_i canal IRQ ------------------*/
91       NVIC_InitStructure.NVIC_IRQChannel         = EXTI0_IRQn;
92       NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
93       NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
94       NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
95       NVIC_Init(&NVIC_InitStructure);
96 }
97
98 /*   Funcion que se ejecuta cuando se detecta la interrupcion    */
99 //-------------------------------------------------------------
100 void EXTI0_IRQHandler(void)
101 {
102       // Comprueba si la linea Linea0 de EXTI se pulsa
103       if(EXTI_GetITStatus(EXTI_Line0) != RESET)
104       {
105             int i;
106             // Enciende y apaga el LED en PB1
107             for (i = 0; i < 5; i++)
108             {
109                   GPIOB->BSRR = GPIO_Pin_1;
110                   espera(100);
111                   GPIOB->BRR = GPIO_Pin_1;
112                   espera(100);
113             }
114
115             // Desactiva la interrupcion para que se vuelva a comprc
116             EXTI_ClearITPendingBit(EXTI_Line0);
117       }
118 }
119
120 // Funcion de espera
121 void espera(unsigned int nCount)
122 {
123       unsigned int i, j;
124       for (i = 0; i < nCount; i++)
125             for (j = 0; j < 0x2AFF; j++);
126 }
```

Figura 10.7

Vemos en el código de la Figura 10.7, que se ha añadido una nueva librería, "**stm32f10x_exti.h**", correspondiente a la librería de gestión de interrupciones externas.

Por ejemplo, para configurar en vez de la **EXTI_0** la **EXTI15_10** con el **IRQ_10**, podemos tomar nuestro código del ejemplo anterior, donde controlábamos la interrupción EXTI_0 en el pin PA0, y cambiarlo todo para que, se detecte una interrupción en otro pin, el PB12; para ello, solo necesitaríamos cambiar las líneas del ejemplo como en el código de ejemplo de la Figura 10.8.

```
1     .
2     .
3     .
4   /* Creamos la estructura para NVIC Y EXTI  ------------------------*/
5   NVIC_InitTypeDef NVIC_InitStructure;
6   EXTI_InitTypeDef EXTI_InitStructure;
7
8   /*  Habilitamos los relojes para EXTI y redireccion de APB2 ------*/
9   RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
10  GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource12);
11
12  /* Configuracion de PB12 la linea 12 para EXTI_12-------------------*/
13  EXTI_InitStructure.EXTI_Line                    = EXTI_Line12;
14  EXTI_InitStructure.EXTI_Mode                    = EXTI_Mode_Interrupt;
15  EXTI_InitStructure.EXTI_Trigger                 = EXTI_Trigger_Falling;
16  EXTI_InitStructure.EXTI_LineCmd                 = ENABLE;
17
18  /* Iniciamos la EXTI  ------------------------------------------------*/
19  EXTI_Init(&EXTI_InitStructure);
20
21  /* Configuracion de los parametros para la Interrupcion EXTI_0 --*/
22  NVIC_InitStructure.NVIC_IRQChannel                   = EXTI15_10_IRQn;
23  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
24  NVIC_InitStructure.NVIC_IRQChannelSubPriority        = 0;
25  NVIC_InitStructure.NVIC_IRQChannelCmd                = ENABLE;
26
27  /*  Iniciamos el NVIC  ------------------------------------------------*/
28  NVIC_Init(&NVIC_InitStructure);
29  }
30
31  /*  Funcion que se ejecuta cuando se detecta la interrupcion EXTI_12 */
32  void EXTI15_10_IRQHandler(void)
33  {
34        // Comprueba si se ha pulsado el boton en Linea12 de EXTI
35        if(EXTI_GetITStatus(EXTI_Line12))
36        {
37          // Lineas que se ejutara cuando se detecte la interrupcion
38        }
39  }
```

Figura 10.8

Observamos que, a partir de la línea 20, en la Figura 10.8, la función asociada a esta interrupción, que se ejecutará cuando se detecte un cambio de flanco de borde descendente en el pin PB12, en donde conectamos ahora nuestro pulsador; cambia su nombre por "**EXTI15_10_IRQHandle**r()".

Si se desea conocer el total de interrupciones posibles en este tipo de microcontrolador, existe un listado completo de ellas que podemos consultar dentro del fichero "**startup_stm32f10x_md.s**" que crea el Keil cuando creamos un proyecto en la carpeta:

"\Mi_proyecto\RTE\Device\STM32F103C8" en su apartado "; External Interrupts".

10.2 EJEMPLO DE CONTROL DE INTERRUPCIÓN USART

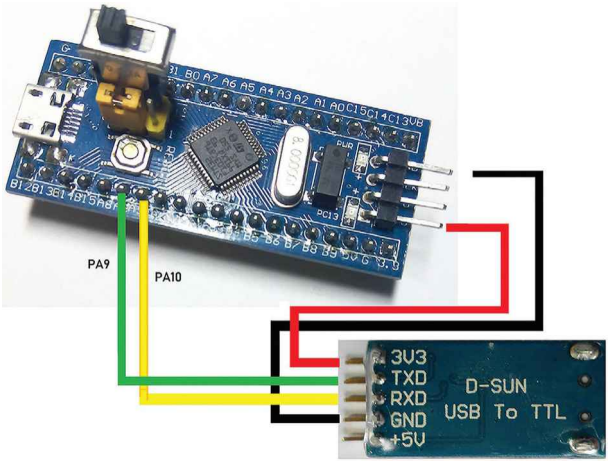


Figura 10.9

Como ya dijimos en el apartado sobre el empleo del módulo USART -el capítulo 9-; donde indicábamos que este dispositivo cuenta con canales para el control de interrupciones, en nuestro siguiente ejemplo, crearemos la programación necesaria para el control de esa interrupción que detectará la recepción de datos a través del puerto USART1.

Los pasos a seguir en la configuración y programación de la interrupción USART, como siempre, comienzan por la creación de las estructuras GPIO, USART y RCC que ya conocemos de los ejemplos del apartado 8 de nuestro libro. Ahora, en la estructura de configuración se añade una línea de comando nueva, la **USART_ITConfig(USART1, USART_IT_RXNE, ENABLE)** con la que habilitamos la "bandera" **USART_IT_RXNE** que corresponde con la interrupción que detectará la entrada de datos en el Rx, como se observa en el código de la Figura 10.10.

```
1   /*  Configuracion del USART1  */
2   USART_InitStructure.USART_BaudRate = 115200;
3   USART_InitStructure.USART_WordLength = USART_WordLength_8b;
4   USART_InitStructure.USART_StopBits = USART_StopBits_1;
5   USART_InitStructure.USART_Parity = USART_Parity_No ;
6   USART_InitStructure.USART_HardwareFlowControl= USART_HardwareFlowControl_None;
7   USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
8   USART_Init(USART1, &USART_InitStructure);
9
10  USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
11
12  USART_Cmd(USART1, ENABLE);
```

Figura 10.10

El siguiente bloque de nuestro código creará la estructura con la configuración necesaria para el control **NVIC** de la interrupción, cuyo formato ya conocemos.

```
1  /*  Creamos la estructura para NVIC    */
2  NVIC_InitTypeDef NVIC_InitStruct;
3
4  //  Inicializamos el Grupo4 para la prioridad del IRQ USART
5  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
6
7  // Establecemos el System Timer IRQ como de alta prioridad
8  NVIC_SetPriority(SysTick_IRQn, 0);
9
10 // Establecemos el IRQ para el USART1
11 NVIC_InitStruct.NVIC_IRQChannel = USART2_IRQn;
12 NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 1;
13 NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
14 // Iniciamos el NVIC configurado
15 NVIC_Init(&NVIC_InitStruct);
```

Figura 10.11

Las "banderas" (USART_Flags) son las fuentes de las interrupciones a detectar y configurar en nuestros programas. Existen varias fuentes posibles, como se muestran en la Figura 10.12 que, a su vez, podemos utilizar para comprobar el estado de esa interrupción, como veremos más adelante en la explicación de nuestro ejemplo.

USART_FLAG
USART_FLAG_CTS
USART_FLAG_LBD
USART_FLAG_TXE
USART_FLAG_TC
USART_FLAG_RXNE
USART_FLAG_IDLE
USART_FLAG_ORE
USART_FLAG_NE
USART_FLAG_FE
USART_FLAG_PE

Figura 10.12

La bandera **USART_IT_TXE** (*Transmit Data register empty*). Su función es la de indicar cuándo se ha llenado el buffer de transmisión y se borra automáticamente cuando se llama al comando "*USART*_**SendData**()".

La **USART_IT_RXNE** (*Receive Data register not empty*). Avisa de cuando el buffer de entrada de datos recibidos contiene un valor mayor de '0', indicando

que se ha recibido un dato. Se borra automáticamente cuando se llama al comando "*USART*_**ReceiveData**(*USARTx*)" y también manualmente, cuando se ejecuta el comando "*USART*_**ClearITPendinBit**(*USARTx*, **USART_IT_RXNE**)".

La **USART_IT_TC** (*Transmission complete*). Informa de cuando se ha completado la transmisión de datos. Se resetea automáticamente cuando se emplean los comandos "*USART*_**GetITStatus**(*USARTx*, **USART_IT_TC**)" y "*USART*_**SendData**()"; y manualmente cuando se emplea el comando "*USART*_**ClearITPendingBit**(*USARTx*, **USART_IT_TX**)".

En el siguiente bloque de código será necesario establecer la función que contendrá la parte de nuestra programación que se ejecutará cuando se detecte la interrupción, esta se crea mediante la función de servicio "**USART1_IRQHandler**" como en la Figura 10.13.

```
1  void USART1_IRQHandler(void)
2  {
3        u8 caracter;
4        // Comprobamos si existe un dato recibido comprobando el
5        // estado de la bandera 'USART_IT_RXNE'
6        if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
7        {
8        caracter = USART1->DR;
9        USART_SendData(USART1, caracter);  // Se reenvia el caracter
10       }
11
12 }
```

Figura 10.13

Donde lo primero que comprobamos, es si se recibe algún dato en la bandera "***USART_IT_RXNE***" en la línea 6 del código en la Figura 10.14:

```
if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
```

Figura 10.14

Una vez descargado el contenido del registro "DR" en la variable "carácter", en nuestro código, procedemos a reenviar el dato recibido con el comando "**USART_SendData**(USARTx, dato)" por el mismo puerto hacia el ordenador.

```
...
USART_SendData(USART1, caracter);
...
```

Figura 10.15

De tal forma que, en nuestro terminal del ordenador reaparecerán cada uno de los caracteres enviados hacia nuestra placa.

10.3 EJEMPLO DE DETECCIÓN DE MOVIMIENTO Y EXTI9_5

En el siguiente código configuraremos la interrupción **EXT9_5_IRQ,** que es la que se emplea para detectar cambios en los pines del 5 al 9 de cualquier puerto, para crear una sencilla alarma, que se activará con el movimiento de un cuerpo mediante la configuración de un sensor de movimiento por infrarrojos.

Un sensor **PIR** (*Passive Infrared*) es un dispositivo que detecta el movimiento de cualquier objeto o persona en una zona determinada, mediante la detección de calor o el cambio de temperatura de determinadas zonas.

En nuestro ejemplo emplearemos el HC-SR501. Un sencillo y barato dispositivo que se puede alimentar desde 5 Vcc hasta 12 Vcc. Cuenta con dos potenciómetros en su parte inferior para configurar tanto la sensibilidad de detección como el tiempo de retraso en una nueva detección.



Figura 10.16

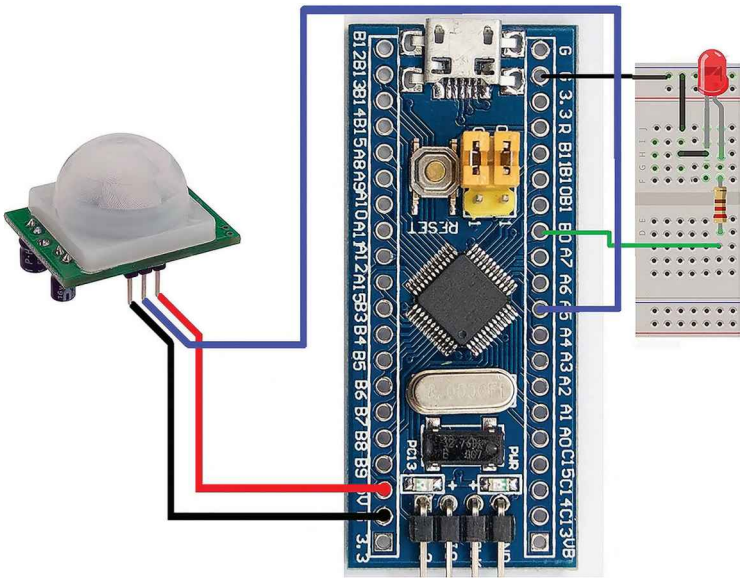


Figura 10.17

Código del fichero “main.c” de nuestro ejemplo:

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_usart.h"

// Libreria para control de Interrupciones Externas
#include "stm32f10x_exti.h"

// Libreria para manejo del USART1
#include "usart1.h"

// Libreria para manejo de variables
#include <stdio.h>

// Lista de funciones utilizadas
void GPIO_Config(void);
void TIM_Config(void);
void EXTI0_IRQHandler(void);

// Lista de varianles empleadas
volatile uint8_t Senal_ECHO = 0;
volatile uint16_t Valor_Escanner;

/*       Funcion que genera la señal de Trigger       */
//-----------------------------------------------------
void HCSR04_start() {
      int i;
      GPIO_SetBits(GPIOB, GPIO_Pin_15);   // Activamos PB15

      for(i=0;i<0x7200;i++);  // Contador para producir retardo
      GPIO_ResetBits(GPIOB, GPIO_Pin_15); // Apagamos PB15
}

/*      Funcion que cuenta el tiempo transcurrido     */
```

```
//-------------------------------------------------------
unsigned int HCSR04_get() {
      unsigned long Escanner;
      Escanner = (unsigned long)Valor_Escanner /73;
      return (unsigned int)Escanner;
}

/*                  Modulo principal                      */
//=============================================          =======
int main(void)
{
  GPIO_Config();     // Configuramos el GPIO de señales Trigger y Echo
  TIM_Config();      // Configuramos e inicializamos el TIMER
  USART1_Config();   // Configurtamos la comunicacion por USART1

  while(1)
  {
      // Comprobamos si la señal ECHO se ha recibido
      if (Senal_ECHO == 1) {
        // Imprimimos por el puerto USART1 el valor obtenido
        USART1_printf(USART1, "  Distancia %d cm\r\n", HCSR04_get());

        // Desactivamos la bandera de deteccion de la señal ECHO
        Senal_ECHO = 0;
      }
    }
}

/*    Funcion que configura el GPIO                       */
//-------------------------------------------------------------
void GPIO_Config(void)
{
    /* Configuramos GPIO para el LED en PC13 ---------------*/
    GPIO_InitTypeDef GPIO_InitStructure;

    /* Iniciamos el reloj para GPIOC   ---------------------*/
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

    GPIO_StructInit(&GPIO_InitStructure);

    /* Creamos señal Trigger en Pin PB15 como salida -------*/
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
    GPIO_InitStructure.GPIO_Pin     = GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode    = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed   = GPIO_Speed_2MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* Creamos señal Echo en pin PB0 como entrada ----------*/
    GPIO_InitStructure.GPIO_Pin     = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode    = GPIO_Mode_IPU;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* Creamos la estructura para el EXTI en PB0 y el NVIC -*/
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Habilitamos el reloj para el remapeo de PB0 ---------*/
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);

    /* Creamos el NVIC para la IRQ del EXTI_0 --------------*/
    NVIC_InitStructure.NVIC_IRQChannel    = EXTI0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    /* Creamos el GPIO para el pin PB0 de la EXTI_Line0 ----*/
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource0);

    /* Configuramos los parametros del la EXTI_0 -----------*/
    EXTI_InitStructure.EXTI_Line    = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode    = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising_Falling;
```

```
109     EXTI_InitStructure.EXTI_LineCmd = ENABLE;
110
111     /* Iniciamos la interrupcion */
112     EXTI_Init(&EXTI_InitStructure);
113 }
114
115 /*     Funcion que configura el TIMER_3                    */
116 //-------------------------------------------------------------
117 void TIM_Config(void)
118 {
119     /* Creamos la configuracion para el Timer 3 ------------*/
120     TIM_TimeBaseInitTypeDef TIMER_InitStructure;
121     TIM_TimeBaseStructInit(&TIMER_InitStructure);
122     TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
123     TIMER_InitStructure.TIM_Prescaler = 72;
124     TIM_TimeBaseInit(TIM3, &TIMER_InitStructure);
125     TIM_Cmd(TIM3, ENABLE);
126
127     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
128     TIM_TimeBaseInitTypeDef TIMER_InitStructure;
129     TIM_TimeBaseStructInit(&TIMER_InitStructure);
130     TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
131     TIMER_InitStructure.TIM_Prescaler   = 7200;
132     TIMER_InitStructure.TIM_Period      = 5000;
133     TIM_TimeBaseInit(TIM4, &TIMER_InitStructure);
134     TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
135     TIM_Cmd(TIM4, ENABLE);
136
137     /* Creamos la interrupcion del Timer 4  ------------------*/
138     NVIC_InitTypeDef NVIC_InitStructure;
139     NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
140     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
141     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
142     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
143     NVIC_Init(&NVIC_InitStructure);
144 }
145
146 /*  Funcion que se ejecuta cuando la interrupcion es detectada */
147 //-------------------------------------------------------------------
148 void TIM4_IRQHandler(void)
149 {
150         if (TIM_GetITStatus(TIM4, TIM_IT_Update) != RESET)
151         {
152             HCSR04_start();
153             Senal_ECHO = 1;
154             TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
155         }
156 }
157
158 /*  Funcion que se ejecuta cuando se detecta la Interrupcion EXTI_0  */
159 //-------------------------------------------------------------------
160 void EXTI0_IRQHandler(void)
161 {
162     // Comprueba si la bandera de estado esta activa
163     if (EXTI_GetITStatus(EXTI_Line0) != RESET)
164     {
165         // Comprueba si PB0 esta activo
166         if (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0) != 0) {
167             // Inicia el conteo del tiempo
168             TIM_SetCounter(TIM3, 0);
169             }
170         // COmprueba si el PB0 se ha desactivado
171             if (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0) == 0) {
172                   // Pasa el valor del conteo obtenido a la variable 'Escanner'
173                   Valor_Escanner = TIM_GetCounter(TIM3);
174             }
175             // Reinicia la Interrupcion EXT_0
176             EXTI_ClearITPendingBit(EXTI_Line0);
177       }
178 }
```

Figura 10.18

11

PROGRAMACIÓN TIMER



Nuestro microcontrolador, el STM32F103C8, posee 5 temporizadores de timer: 3 de propósito general (**TIM2**, **TIM3** y **TIM4**), 1 de control avanzado (**TIM1**) y un quinto timer especial que se puede sincronizar con una señal de frecuencia externa que se puede utilizar para controlar otros temporizadores (**TIMx_ETR**-*External trigger timer input*).

Los 5 timers se pueden emplear para:

- **Capturar señales de entrada** proveniente de otros periféricos externos.
- **Comparar una cuenta realizada internamente** con alguna señal de salida.
- **Generar frecuencias** PWM.
- **Generar salidas por pulsos** a emplear externamente.
- **Generar señales de interrupción** para eventos.

Básicamente sus funciones se pueden dividir en dos grandes grupos: una, la de generar una *base de tiempos* de sincronización para los periféricos internos o externos y otra, la de *capturar o contar las señales* o eventos tanto de periféricos del interior como del exterior de nuestro microcontrolador.

CANALES DE TIMER

TIMx	CH1	CH2	CH3	CH4
TIM1	PA8	PA9	PA10	PA11
TIM2	PA0	PA1	PA2	PA3
TIM3	PA6	PA7	PB0	PB1
TIM4	PB6	PB7	PB8	PB9

- PB13 (TIM1N_CH1)
- PB14 (TIM1N_CH2)
- PB15 (TIM1N_CH3)
- PA11 (TIM1_CH4)
- PA8 (TIM1_CH1)
- PA9 (TIM1_CH2)
- PA10 (TIM1_CH3)
- PA0 (TIM2_CH1)
- PA1 (TIM2_CH2)
- PA2 (TIM2_CH3)
- PA3 (TIM2_CH4)
- PA6: (TIM3_CH1)
- PA7: (TIM3_CH2)
- PB0: (TIM3_CH3)
- PB1: (TIM3_CH4)
- PB6: TIM4_CH1)
- PB7: TIM4_CH2)
- PB8: TIM4_CH3)
- PB9: TIM4_CH4)

Figura 11.1

Cada timer posee 4 canales que son los elementos de conexión por los cuales un temporizador interactúa con su entorno externo. Estos están asignados a algunos pines del microcontrolador como se muestra en la Figura 11.1.

Estos canales pueden configurarse tanto de salida, donde pueden generar temporizaciones, intervalos para el control de tareas o señales de onda PWM; como de entrada, en el que el canal podrá utilizarse para señalar tiempos de señales externas y detectar que el flanco pueda ser ascendente y/o descendente, calcular frecuencias o tiempos entre eventos.

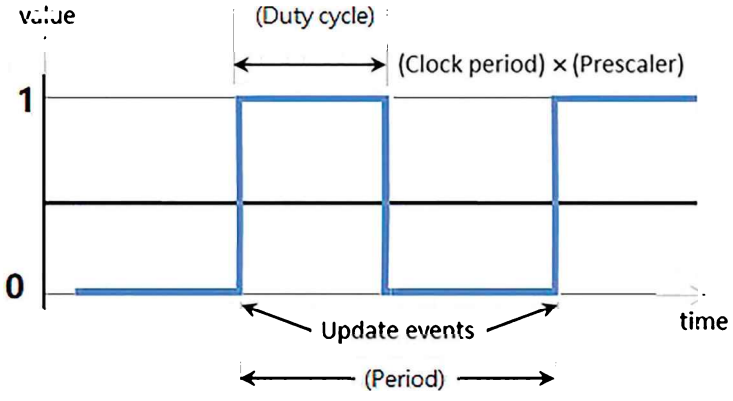


Figura 11.2

El funcionamiento de un timer básicamente es configurar: unos registros de 16 bits para que varíen la frecuencia del reloj de referencia de entrada y un prescaler que puede adquirir un valor de entre 1 y 65535.

El timer realiza un conteo programable hacia arriba o hacia abajo, de un valor que previamente se ha establecido en uno de los registros de 16 bits, el de auto recarga automática (**TIM_ARR** -*Auto-reload counter*):

En el modo hacia arriba, el conteo va de 0 al valor de ARR y al llegar a ese valor, el contador vuelve a 0 y se repite el conteo. Cuando se produce el restablecimiento a 0, se genera un evento de actualización (*update event*).

En el modo hacia abajo, el contador disminuye desde el valor en ARR hasta el 0; y luego se restablece a ARR, generando un evento en la actualización.

Existe también el modo "arriba/abajo" (**Up/Down mode**), en el que aunque el contador se produce parecido al modo "arriba", en el que el contaje va de 0 al valor de ARR, el evento se señalará antes del restablecimiento.

Para la configuración de los timer es necesario establecer una *base de tiempos* que nos permita tener una frecuencia de referencia para ser comparada con otra. Para la que es necesario calcular previamente tres valores: el valor del contador (**TIMx_CNT**), el valor del divisor o "*Prescaler*" (**TIMx_PSC**) y el valor de autorrecarga (**TIMx_ARR**).

- **TIMx_CNT** – (*Counter Register*) Registro de 16 bits donde se guarda el valor de la cuenta ascendente o descendente que realizará el temporizador.
- **TIMx_ARR** – Registro auto recarga del timer (*Auto-reload register*) que contiene el valor de recarga para el contador del timer.

  Si el contador del temporizador cuenta hacia arriba y alcanza el contenido del registro de auto recarga (**TIMx_ARR**), entonces, el temporizador se restablece automáticamente y se reinicia un nuevo ciclo de conteo.

  Si el contador del temporizador cuenta hacia abajo y alcanza el valor cero, entonces, el contador del temporizador se establece con el contenido del registro de recarga automática (**TIMx_ARR**) y se reinicia un nuevo ciclo de conteo.

  Cada vez que se reinicia un nuevo ciclo de conteo, se activa un "evento de actualización" o interrupción.

- **TIMx_PSC** – Divisor o prescaler que divide la frecuencia de reloj por un valor entre 1 y 65 535.

La relación matemática entre estos valores la podemos deducir de la siguiente ecuación:

$$Update\ Event\ (Hz) = \frac{Timer\ clock\ (Hz)}{(TIM_PSC + 1) * (TIM_ARR + 1)}$$

Figura 11.3

Donde el valor de "*Timer clock*" dependerá del reloj del timer correspondiente (APB1=36MHz y APB2=72MHz) según el módulo timer que vayamos a utilizar.

En nuestra programación, los pasos a seguir para configurar un timer son los siguientes:

- **Paso 1.** Será necesario configurar el GPIO. Debemos tener en cuenta, que cada timer tiene asignado un pin determinado para cada uno de los cuatro canales. Cualquiera de ellos se podrá configurar como entrada o salida dependiendo del proceso o función que vayamos a realizar con nuestro timer.

  Esta parte ya ha sido explicada en capítulos anteriores, por lo que no procede repetir aquí los comandos de creación de la estructura GPIO para un pin de canal TIMx_CHn.

- **Paso 2.** Crear la estructura con la configuración del "***timer***" a emplear, según las líneas mostradas en el ejemplo de la Figura 11.4; y que en este caso se basan en el **TIM2.**

```
1  /* Crea la estructura del TIMER 2 ------------------------*/
2  TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
3
4  /* Habilitacion del reloj del TIMER 2  -------------------*/
5  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
6
7  /* Configuramos los parametros del TIMER_2 ---------------*/
8  TIM_TimeBaseInitStruct.TIM_Prescaler = 3599;
9  TIM_TimeBaseInitStruct.TIM_Period = 9999;
10 TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV1;
11 TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;
12 TIM_TimeBaseInit(TIM2, &TIM_TimeBaseInitStruct);
13
14 /* Inicializa el TIMER 2 programado  ----------------------*/
15 TIM_Cmd(TIM2, ENABLE);
```

Figura 11.4

Mediante el comando "**TIM_TimeBaseInitTypeDef**[*nombreEstructuraInit*]" comenzamos la creación de la estructura necesaria para configurar todos los parámetros.

Posee los siguientes parámetros a configurar:

TIM_ClockDivision TIM_CounterMode TIM_Period
TIM_Prescaler TIM_RepetitionCounter

Figura 11.5

Habilitamos el reloj correspondiente que gestionará ese timer, mediante el comando que ya conocemos de "*RCC_APBxPeriphClockCmd*(**RCC_APB1/2Periph**, **TIMx**, ENABLE)". Donde 'x' será él número del "timer" que vayamos a utilizar, teniendo en cuenta que a los TIM2, TIM3 y TIM4 les corresponde el reloj **APB1** y al TIM1 el **APB2**.

En las siguientes líneas configuramos los valores de "**TIM_Period**" y "**TIM_Prescaler**" que producirán la frecuencia de base de tiempos que necesita nuestro proyecto.

Los valores de "**periodo**" y "**prescaler**" necesarios se calculan con las siguientes fórmulas:

1. Para calcular el periodo del timer (**TIM_Period**) que puede ser un valor entre 1 y 65 535 usaremos la siguiente:

$$TIM_{Period} = \left(\frac{Timer_{CLK}\ (Hz)}{Frec_{event}\ (Hz)} \right) - 1$$

Figura 11.6

Donde: **Timer_CLK** – Es la frecuencia de entrada al timer según el reloj que será de 72 MHz, que debemos pasar a Hz para aplicarlo a la ecuación.

Frec_event – Corresponde al valor de la frecuencia que queremos configurar expresada en Hz también (*Update event*).

Expongamos un ejemplo. En el supuesto de desear una base de tiempos para nuestro timer TIM2 de 16 kHz (= 16 000 Hz) sería:

TIM_Period = (72 000 000 / 16 000 Hz) -1 = **4499**

Figura 11.7

Habrá determinados valores de **TIM_Period** que obtendremos de aplicar la fórmula anterior, que nos resultarán más fáciles para deducir directamente los valores de auto recarga para **TIM_ARR** y para el valor

de prescaler **TIM_PSC**, simplemente al poder dividirlos por cantidades múltiplos de 100.

Veamos algunos ejemplos:

El TIM_Period resultante de la frecuencia 5 kHz = 14 400, donde podemos configurar:

TIM_ARR = (144 - 1) y **TIM_PSC** = (100 - 1), al dividir 14 400 / 100 = 144

Figura 11.8

El TM_Period de la frecuencia 10 Hz = 7 200 000, lo que nos da:

TIM_ARR = (7 200 – 1) y **TIM_PSC** = (1 000 – 1),
porque 7 200 000 / 1 000 = 7 200.

Figura 11.9

Para otras cantidades más complejas o menos evidentes habrá que calcular el prescaler con la siguiente fórmula.

2. Para calcular el divisor o prescaler (**TIM_Presca**ler), que puede ser un valor de 16 bits entre 1 y 65 535, se usará la siguiente fórmula:

$$TIM_{Prescaler} = \frac{SystemCoreClock\ (Hz)}{\frac{Frec_{Event}\ (Hz)}{(TIM_{Period} + 1)}}$$

Figura 11.10

Donde: **SystemCoreClock** – Corresponde al reloj central (HCLK) del microcontrolador que es de 72 MHz.

Frec_event – Corresponde al valor de la frecuencia que queremos configurar expresado en Hz (*Update event*).

TIM_Period – Es el valor del periodo que hemos obtenido en la ecuación anterior (TIMx_ARR).

Para nuestro ejemplo anterior, donde queríamos generar una base de tiempos para una frecuencia de 16 kHz, el prescaler sería:

TIM_Prescaler = 72 000 000 / 16 000 / (4 499 + 1) = **1**

Figura 11.11

El siguiente parámetro será indicar el "**TIM_ClockDivision**" o divisor para nuestro reloj, al que podemos darle los siguientes valores: ***TIM_CKD_DIV1***, ***TIM_CKD_DIV2***(x2) o ***TIM_CKD_DIV4***(x4).

A continuación, establecemos el modo en el que se realizará el conteo, mediante el comando "**TIM_CounterMode**" al que podemos dar los siguientes modos:

- ***TIM_CounterMode_Up*** – Conteo hacia arriba, cuando de valor 0 cuenta a ARR.
- ***TIM_CounterMode_Down*** – Conteo hacia abajo, cuando de ARR cuenta hacia 0.
- ***TIM_CounterMode_CenterAligned1*** – Modo conteo centrado cada 1.
  TIM_CounterMode_CenterAligned2 – Modo conteo centrado cada 2.
  TIM_CounterMode_CenterAligned3 – Modo conteo centrado cada 3.
  Los modos de conteo de alineación central, es cuando se detecta en cada cambio de voltaje de 0 a 1, de 1 a 2, de 3 a 4, etc ...

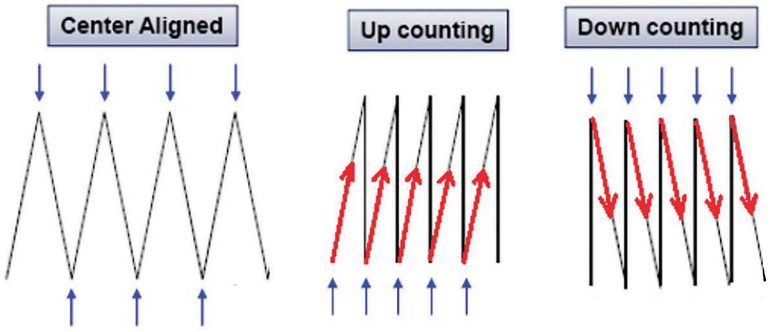


Figura 11.12

Luego, mediante el comando "**TIM_TimebaseInit** (**TIMx**, &[*nombreEstructuraInit*])" cargamos la configuración del timer que hemos programado en nuestra estructura.

Por último, necesitamos iniciar el timer con el comando "**TIM_Cmd**(**TIMx**, *ENABLE*)".

La biblioteca de funciones para los timers proporciona también dos funciones más, **TIM_GetITStatus** y **TIM_ClearITPendingBit**, que se utilizan para determinar el estado del temporizador y borrar las banderas de estado de los registros de los timer.

La función **TIM_GetITStatus** comprobará si está habilitada la interrupción para determinar el indicador de interrupción y **TIM_GetFlagStatus** directamente es utilizada para determinar el estado del indicador.

11.1 EJEMPLO DE TIMER COMO TEMPORIZADOR

En el siguiente código de ejemplo, configuraremos los timer TIM2 y TIM3 a diferentes frecuencias y conectaremos dos LED que se encenderán y apagarán cuando se detecten las interrupciones por desbordamiento en el modo ascendente de cada timer.

El LED1 estará conectado al pin PB1 de nuestra placa y se encenderá y apagará cuando se detecte el desbordamiento del timer 2 que estará configurado para una frecuencia de 2 Hz y que producirá que se encienda y apague cada 0,5 segundos.

El LED2 estará conectado al PC13, el led de pruebas de la propia placa; que estará relacionado con el desbordamiento del timer 3 que hemos configurado con una frecuencia de 10 Hz, lo que provocará que se encienda y apague cada 0.1 segundos.

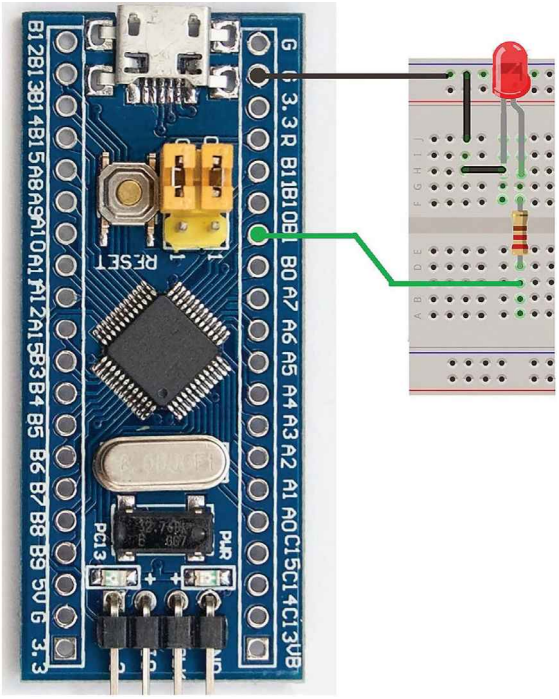


Figura 11.13

El código del ejemplo es el siguiente:

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"

// Lista de funciones empleadas
void GPIO_Config(void);
void TIM2_Config(void);
void TIM3_Config(void);
void TIM2_IRQHandler(void);
void TIM3_IRQHandler(void);

/*       Modulo principal                                    */
//=============================================-==========
int main(void){
  GPIO_Config();  // Inicializamos los GPIO
  TIM2_Config();  // Inicializa el Timer 2
  TIM3_Config();  // Inicializa el Timer 3

  while (1)
  {
   ;;
  }
}

/*       Funcion que configura el GPIO para los LED      */
//-------------------------------------------------------  -
void GPIO_Config(void)
{
  GPIO_InitTypeDef GPIO_InitStruct;
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB |
                         RCC_APB2Periph_GPIOC, ENABLE);

  // Configura el pin PB1 como salida para LED1
  GPIO_InitStruct.GPIO_Pin = GPIO_Pin_1;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_2MHz;
  GPIO_Init(GPIOB, &GPIO_InitStruct);

  // Configura el pin PC13 como salida para LED2
  GPIO_InitStruct.GPIO_Pin = GPIO_Pin_13;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_2MHz;
  GPIO_Init(GPIOC, &GPIO_InitStruct);
}

/*         Funcion que configura el TIM2            */
//-----------------------------------------------------
void TIM2_Config(void)
{
  // TIM2   ------------------------------------------
  // Crea la estructura del TIMER 2
  TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

  /* Configuramos los parametros del TIMER_2 para
     detectar un desbordamiento cada vez que cuente
     un periodo de tiempo de 2 HZ (0,5 seg = 500 ms)
    ------------------------------------------------------
    Formula para calcular el periodo de tiempo:
     Update_event (Hz) = 72 000 000  / 2 = 36 000 000 Hz
       TIM_Prescaler(ARR) = (3600 - 1)
       TIM_Period    (PSC) = (10 000 - 1)                  */

  TIM_TimeBaseInitStruct.TIM_Prescaler = 3599;
  TIM_TimeBaseInitStruct.TIM_Period = 9999;
  TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV1;
```

```
68    TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;
69    TIM_TimeBaseInit(TIM2, &TIM_TimeBaseInitStruct);
70
71    // Creamos la interrupcion para TIM2
72    TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
73
74    // Iniciamos el TIM2 programado
75    TIM_Cmd(TIM2, ENABLE);
76
77    // NVIC  ------------------------------------------------
78    // Nombramos la estructura para configurar la interrupcion
79    NVIC_InitTypeDef NVIC_InitStruct;
80    // Configuramos los parametros de la interrupcion
81    NVIC_InitStruct.NVIC_IRQChannel = TIM2_IRQn;
82    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0x00;
83    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0x00;
84    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
85    NVIC_Init(&NVIC_InitStruct);
86  }
87
88  /*      Funcion que configura el TIM3             */
89  //-----------------------------------------------
90  void TIM3_Config(void)
91  {
92    // TIM3  ------------------------------------------------
93    // Crea la estructura del TIMER 3
94    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
95    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
96
97    /* Configuramos los parametros del TIMER_3 para
98       detectar un desbordamiento cada vez que cuente
99       un periodo de tiempo de 10 Hz  (0,1 seg) 100ms
100    ------------------------------------------------------
101     Formula para calcular el periodo de tiempo:
102      Update Event(Hz) = Timer_CLK (Hz) / Frec_event (Hz)
103                       =    72 000 000  /  10    = 7 200 000 Hz
104     TIM_Prescaler(ARR)= (7 200 - 1)
105     TIM_Period    (PSC)= (1 000 - 1)                    */
106
107    TIM_TimeBaseInitStruct.TIM_Prescaler = 7199;
108    TIM_TimeBaseInitStruct.TIM_Period = 999;
109    TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV1;
110    TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;
111    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseInitStruct);
112
113    // Creamos la interrupcion para TIM3
114    TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);
115
116    // Iniciamos el TIM3 programado
117    TIM_Cmd(TIM3, ENABLE);
118
119    // NVIC ------------------------------------------------
120    // Nombramos la estructura para configurar la interrupcion
121    NVIC_InitTypeDef NVIC_InitStruct;
122    // Configuramos los parametros de la interrupcion
123    NVIC_InitStruct.NVIC_IRQChannel = TIM3_IRQn;
124    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0x00;
125    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0x00;
126    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
127    NVIC_Init(&NVIC_InitStruct);
128 }
129
130 /*  Funcion que se ejecutara cuando se detecta el desbordamiento
131     en TIMER 2                                                  */
132 //-------------------------------------------------------------
133 void TIM2_IRQHandler()
134 {
135       // Comprueba si se produce la interrupcion del TIM2
136       if (TIM_GetITStatus(TIM2, TIM_IT_Update))
```

```
137       {
138             // Enciende y apaga (Toggle) LED en PB1
139             GPIOB->ODR ^= GPIO_Pin_1;
140
141             // Limpia la bandera de la interrupcion de TIM2 para reiniciarla
142             TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
143       }
144 }
145
146 /*  Funcion que se ejecutara cuando se detecta el desbordamiento
147     en TIMER 3                                                    */
148 //-------------------------------------------------------------------
149 void TIM3_IRQHandler()
150 {
151    // Comprueba si se produce la interrupcion del TIM3
152    if (TIM_GetITStatus(TIM3, TIM_IT_Update))
153    {
154       // Enciende y apaga (Toggle) LED en PC13
155       GPIOC->ODR ^= GPIO_Pin_13;
156
157       // Limpia la bandera de la interrupcion de TIM3 para reiniciarla
158       TIM_ClearITPendingBit(TIM3, TIM_IT_Update);
159    }
160 }
161
```

Figura 11.14

11.2 EJEMPLO DEL TIMER COMO CONTADOR

11.2.1 EJEMPLO DE MEDICIÓN DE TIEMPOS ENTRE DOS EVENTOS

Como vimos al principio de este apartado, los Timer pueden servir también para contar determinados eventos que se produzcan dentro o fuera de nuestro microcontrolador.

En nuestro siguiente ejemplo, empleamos el timer para medir la longitud de un pulso o el tiempo transcurrido entre dos eventos.

Mediante el Timer, calcularemos el tiempo transcurrido entre que se pulsa un pulsador y luego otro, conectados en los pines PB0 y PB1, enviando luego por el puerto serial USART el valor del tiempo medido.

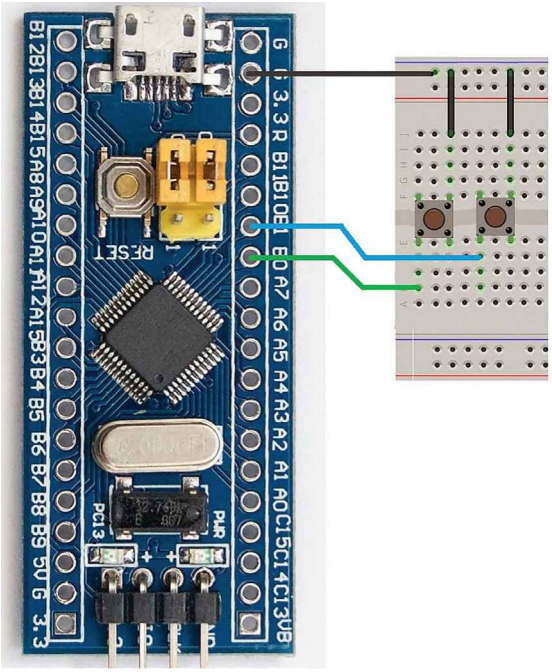


Figura 11.15

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_usart.h"

// Libreria para manejo del USART1
#include "usart1.h"

// Librería complementaria para manejo del comando Sprintf
#include <stdarg.h>

// Lista de funciones utilizadas
void GPIO_Config(void);
void TIM_Config(void);
void TIM4_IRQHandler(void);

// Lista de variables empleadas
volatile int TimeElapsed;     // Variable que guarda el tiempo final
transcurrido
volatile int TimeSec;
volatile uint8_t TimeState = 0;
float TimeSEGUNDOS = 0;


```

```
26 /*            Modulo principal                                    */
27 //-----------------------------------------------------------------
28 int main(void)
29 {
30     char buff...[30] = {'\0'};
31
32     GPIO_Config(); //Configuramos el GPIO de Pulsadores y LED testigo
33
34     TIM_Config();  // Configuramos e inicializamos el TIMER
35
36     USART1_Config(); // Configurtamos la comunicacion por USART1
37
38     while(1)
39     {
40         if (TimeState == 0) {
41             // Comprueba si se ha pulsado el boton PB0
42             if (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0) == 0)
43             {
44                 TIM_Cmd(TIM4, ENABLE);
45                 TIM_SetCounter(TIM4, 0);
46                 TimeSec = 0;
47                 // Establecemos Bandera de Tiempo inicial = 1 (On)
48                 TimeState = 1;
49                 // Apagamos el LED testigo en PC13
50                 GPIO_ResetBits(GPIOC, GPIO_Pin_13);
51                 // Imprimimos un mensaje de inicio por el USART1
52                 USART1_printf(USART1, " Inicio...");
53             }
54         }
55
56         if (TimeState == 1) {
57             // Comprueba si se ha pulsado el boton PB1
58             if (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_1) == 0) {
59                 // Sumamos en la variable 'TimeElapsed' el tiempo
60                 // transcurrido mediante la formula
61                 // Time in en ms
62                 TimeElapsed = TIM_GetCounter(TIM4)/10 + TimeSec * 1000;
63                 TIM_Cmd(TIM4, DISABLE);
64                 // Bandera de tiempo transcurrido finalizado = 0
65                 TimeState = 0;
66                 // Encendemos el LED testigo en PC13
67                 GPIO_SetBits(GPIOC, GPIO_Pin_13);
68                 // Imprimimos el tiempo transcurrido por USART1
69                 TimeSEGUNDOS = TimeElapsed * 0.001;
70                 USART1_printf(USART1, "Tiempo transcurrido: %d ms ",
71 TimeElapsed);
72                 printf( "-> %4.2f Seg. \r\n", TimeSEGUNDOS);
73
74             }
75         }
76     }
77 }
78
79 /*  Funcion donde configuramos los GPIO     */
80 //-----------------------------------------
81 void GPIO_Config(void)
82 {
83     /* Configuramos GPIO para el LED en PC13 ------------*/
84     GPIO_InitTypeDef  GPIO_InitStructure;
85
86     /* Iniciamos el reloj para GPIOC  -------------------*/
87     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
88
89     /* Configuramos GPIO del pin PC13 para el LED  ------*/
90     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
91     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
92     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
93     GPIO_Init(GPIOC, &GPIO_InitStructure);
94
```

```
95      GPIO_SetBits(GPIOC, GPIO_Pin_13);
96
97      /* Configuramos el GPIO para PB0 y PB1 --------------*/
98      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
99      /* Configuramos los pines para los pulsadores en PB0 y PB1  */
100     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
101     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
102     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
103     GPIO_Init(GPIOB, &GPIO_InitStructure);
104 }
105
106 /*      Funcion que configura el TIMER4      */
107 //-------------------------------------------------
108 void TIM_Config(void)
109 {
110     /*   Crea la estructura del TIMER_4 ----------------------*/
111     TIM_TimeBaseInitTypeDef TIMER_InitStructure;
112     // Inicializa el reloj para el TIMER4
113     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
114
115     /* Configuramos los parametros del TIMER4   --------------*/
116     TIM_TimeBaseStructInit(&TIMER_InitStructure);
117     TIMER_InitStructure.TIM_Prescaler = 7200;
118     TIMER_InitStructure.TIM_Period = 10000;
119     TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
120     TIM_TimeBaseInit(TIM4, &TIMER_InitStructure);
121
122     /* Creamos la interrupcion para TIM4  ---------------*/
123     TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
124     TIM_Cmd(TIM4, ENABLE);
125
126     /* Configuracion del NVIC ---------------------------*/
127     NVIC_InitTypeDef NVIC_InitStructure;
128     NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
129     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
130     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
131     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
132     NVIC_Init(&NVIC_InitStructure);
133 }
134
135 /*   Funcion que se ejecuta cuando se detecta la primera interrupcion
136 //-------------------------------------------------------------------
137 void TIM4_IRQHandler(void)
138 {
139         if (TIM_GetITStatus(TIM4, TIM_IT_Update) != RESET)
140         {
141           TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
142           // Se va incrementando la variable de tiempo
143           TimeSec++;
144         }
145 }
```

Figura 11.16

La ejecución de nuestro código, mostrará en la terminal de nuestro ordenador el siguiente mensaje:

```
Inicio...Tiempo transcurrido: 2588 ms -> 2.59 Seg.
Inicio...Tiempo transcurrido: 6072 ms -> 6.07 Seg.
Inicio...Tiempo transcurrido: 1301 ms -> 1.30 Seg.
```

Figura 11.17

11.2.2 EJEMPLO DE USO DEL SENSOR HC-SR04

Vamos a explicar ahora, un ejemplo práctico de cómo emplear lo que aprendimos en nuestro proyecto anterior. A continuación trabajaremos con el detector de movimiento **HC-SR04**, que utiliza un sistema parecido al de un sonar para detectar las distancias que existen desde el dispositivo y los objetos que tiene delante.



Figura 11.18

Importante señalar que este dispositivo se **alimenta con 5Vcc**, aunque no existirá ningún problema al conectarlo al pin de salida de 5Vcc de nuestra placa -si la tenemos alimentada por el USB- además de poder conectar las señales TTL directamente a los pines digitales que son tolerantes a esta tensión en nuestra placa.

Mediante el control de interrupciones externas de nuestro microcontrolador, utilizaremos una señal o pulso que se enviará a un pin del dispositivo **HC-SR04**, que luego este emite y tras chocar con el objeto, es recibida de nuevo, generando una señal que nos remite el dispositivo informando de su recepción.

El microcontrolador puede calcular el tiempo transcurrido entre la señal que se envió (**Trigger**) y la señal que nos envía el dispositivo (**Echo**) cuando recibe la señal rebotada por el objeto. Ese tiempo transcurrido podemos convertirlo mediante una tabla, en distancia hasta el objeto. Igual que hacíamos con los pulsadores en el ejemplo anterior.

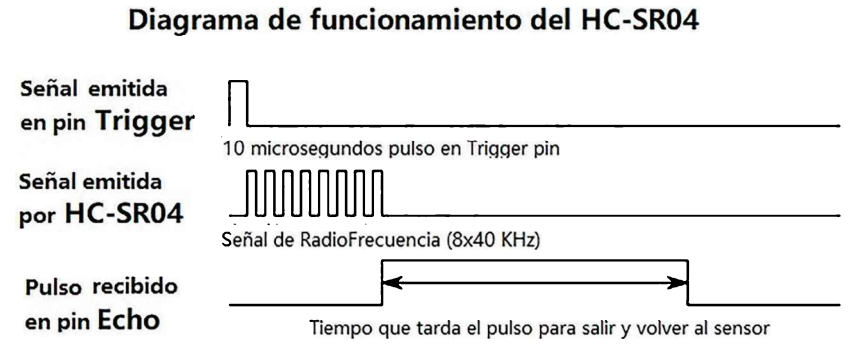


Figura 11.19

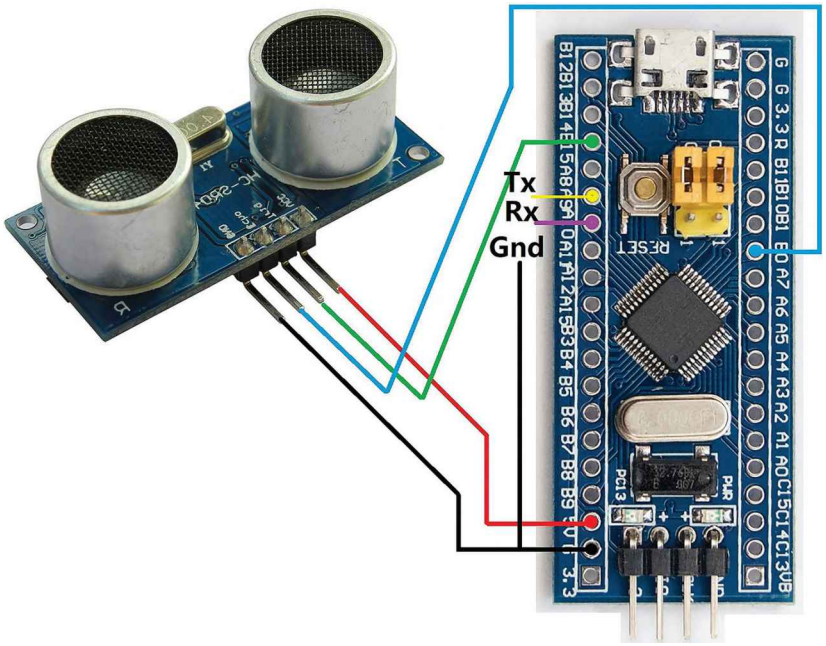


Figura 11.20

Importante: si estamos empleando el adaptador ST-Link para programar nuestra placa, debemos conectar la alimentación del dispositivo HC-SR04 a una alimentación externa, por ejemplo a la salida de +5 Vcc del adaptador.

Código del ejemplo:

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_usart.h"

// Libreria para control de Interrupciones Externas
#include "stm32f10x_exti.h"

// Librerria para manejo del USART1
#include "usart1.h"

// Libreria para manejo de variables
#include "stdio.h"

// Lista de funciones utilizadas
void GPIO_Config(void);
void TIM_Config(void);
void EXTI0_IRQHandler(void);

// Lista de varianles empleadas
volatile uint8_t Senal_ECHO = 0;  // Variable que se usa com bandera de evento
volatile uint16_t Valor_Escanner; // Variable guarda el valor obtenido

/*  Funcion que genera la señal de Trigger   */
//-----------------------------------------------
void HCSR04_start() {
      int i;
      GPIO_SetBits(GPIOB, GPIO_Pin_15);   // Activamos PB15
      for(i=0;i<0x7200;i++);
      GPIO_ResetBits(GPIOB, GPIO_Pin_15); // Apagamos PB15
}

/*      Funcion que cuenta el tiempo transcurrido  */
//-------------------------------------------------------------
unsigned int HCSR04_get() {
      unsigned long Escanner;
      Escanner = (unsigned long)Valor_Escanner /75;
      return (unsigned int)Escanner;
}

/*                  Modulo principal                        */
//==========================================================
int main(void)
{
  GPIO_Config();     // Configuramos el GPIO de señales Trigger y Echo
  TIM_Config();      // Configuramos e inicializamos el TIMER
  USART1_Config();  // Configurtamos la comunicacion por USART1
  while(1)
  {
      // Comprobamos si la señal ECHO se ha recibido
      if (Senal_ECHO == 1) {
        // Imprimimos por el puerto USART1 el valor obtenido
        USART1_printf(USART1, "  Distancia %d cm\r\n", HCSR04_get());
        // Desactivamos la bandera de deteccion de la señal ECHO
            Senal_ECHO = 0;
      }
    }
}

/*    Funcion que configura el GPIO  */
//-----------------------------------------------
void GPIO_Config(void)
{
      /* Configuramos GPIO para el LED en PC13  */
      GPIO_InitTypeDef GPIO_InitStructure;

      // Iniciamos el reloj para PORTC
      RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

      GPIO_StructInit(&GPIO_InitStructure);

      /* Creamos señal Trigger en Pin PB15 como salida  */
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
      GPIO_Init(GPIOB, &GPIO_InitStructure);

      /* Creamos señal Echo en pin PB0 como entrada */

```

```
81        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
82        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
83        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
84        GPIO_Init(GPIOB, &GPIO_InitStructure);
85
86        //EXTI en PB0
87        EXTI_InitTypeDef EXTI_InitStructure;
88        NVIC_InitTypeDef NVIC_InitStructure;
89
90        /* Inicamos el reloj para AFIO */
91        RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
92
93        /* Creamos la IRQ del NVIC    */
94        /* COnfiguramos el pin PB0 conectado como EXTI_Line0 */
95        NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
96        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
97        NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
98        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
99        NVIC_Init(&NVIC_InitStructure);
100
101       /* Llamamos el pin PB0 para la interrupcion EXTI_Line0 */
102       GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource0);
103
104       /* Configuramos PD0 para EXTI_Line0 */
105       EXTI_InitStructure.EXTI_Line = EXTI_Line0;
106       /* Iniciamos interrupcion */
107       EXTI_InitStructure.EXTI_LineCmd = ENABLE;
108       /* Configuramos el modo de la interrupcion */
109       EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
110       /* Configuramos para detectar flanco desendente */
111       EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising_Falling
112       /* Iniciamos la interrupcion */
113       EXTI_Init(&EXTI_InitStructure);
114 }
115
116 /*     Funcion que configura el TIMER_3   */
117 //-----------------------------------------------
118 void TIM_Config(void)
119 {
120     /* Creamos la configuracion para el Timer TIM3 */
121     TIM_TimeBaseInitTypeDef timer_base;
122     TIM_TimeBaseStructInit(&timer_base);
123     timer_base.TIM_CounterMode = TIM_CounterMode_Up;
124     timer_base.TIM_Prescaler = 72;
125     TIM_TimeBaseInit(TIM3, &timer_base);
126     TIM_Cmd(TIM3, ENABLE);
127
128     TIM_TimeBaseInitTypeDef TIMER_InitStructure;
129
130     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
131
132     TIM_TimeBaseStructInit(&TIMER_InitStructure);
133     TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
134     TIMER_InitStructure.TIM_Prescaler = 7200;
135     TIMER_InitStructure.TIM_Period = 5000;
136     TIM_TimeBaseInit(TIM4, &TIMER_InitStructure);
137
138     TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
139
140     TIM_Cmd(TIM4, ENABLE);
141
142     /* Creamos la interrupcion para la interrupcion del Timer */
143     /* TIM4_IRQn Interrupt */
144     NVIC_InitTypeDef NVIC_InitStructure;
145     NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
146     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
147     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
148     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
149     NVIC_Init(&NVIC_InitStructure);
150 }
151
152 /*  Funcion que se ejecuta cuando la interrupcion es detectada */
153 //-----------------------------------------------------------------
154 void TIM4_IRQHandler(void)
155 {
156         if (TIM_GetITStatus(TIM4, TIM_IT_Update) != RESET)
157         {
158             HCSR04_start();
159             Senal_ECHO = 1;      // Activamos la bandera del evento
160             TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
161         }
162 }
163
164 /*  Funcion que se ejecuta cuando se detecta la Interrupcion EXTI_0
165 //-----------------------------------------------------------------
```

```
166 void EXTI0_IRQHandler(void) {
167       // Comprueba si la bandera de estado esta activa
168       if (EXTI_GetITStatus(EXTI_Line0) != RESET) {
169       // Comprueba si PB0 esta activo
170             if (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0) != 0) {
171                   // Inicia el conteo del tiempo
172                   TIM_SetCounter(TIM3, 0);
173             }
174     // COmprueba si el PB0 se ha desactivado
175             if (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0) == 0) {
176                   // Pasa el valor del conteo obtenido a la variabl
177                   Valor_Escanner = TIM_GetCounter(TIM3);
178             }
179             // Reinicia la Interrupcion EXT_0
180             EXTI_ClearITPendingBit(EXTI_Line0);
181       }
182 }
183
184
```

Figura 11.21

En nuestro código de ejemplo, como siempre, necesitamos establecer los parámetros que tendrán los pines que vamos a utilizar. PB15 será un pin de salida con el que generaremos una señal "Trigger" que desencadenará la señal que debe emitir el dispositivo HC-SR04 como radar para buscar el objeto; y este luego activará su salida "Echo" que hemos conectado a nuestro pin PB0 que configuraremos como pin de entrada.

También hemos creado las estructuras necesarias para configurar el NVIC que contendrá la configuración de nuestra interrupción externa EXTI. Interrupción que, internamente, hemos configurado para que se conecte con el canal 0 en el pin PB0, donde detectaremos dicha interrupción.

Además, creamos otra estructura que configura el TIMER 3. Lo utilizaremos para generar una señal a una frecuencia que usaremos como sincronización de la señal de inicio del proceso de escaneo mediante la función en **HCSR04_start**() -contiene la activación del pin PB15, la señal de "Trigger"-.

Mediante la función **EXTI0_IRQHandler**() detectamos cuándo se produce la interrupción que el dispositivo provoca al detectar el objeto y nos activa la señal "Echo" en nuestro pin PB0.

Por último, el proceso termina cuando comparamos el valor que tiene la variable "Valor_Escanner" al desactivar el pin PB0 y se guarda el valor del contador del tiempo transcurrido con el comando **TIM_GetCounter**(*TIMx*).

Luego, mediante la fórmula mostrada, convertimos ese contador de tiempo en un valor en centímetros de distancia.

11.2.3 OTRO EJEMPLO DE MEDICIÓN DE TIEMPOS ENTRE EVENTOS

En nuestro siguiente ejemplo, configuramos dos canales de entrada de un timer, de forma que lee la entrada y cambia su valor detectando el sentido de giro de un encoder.



Figura 11.22

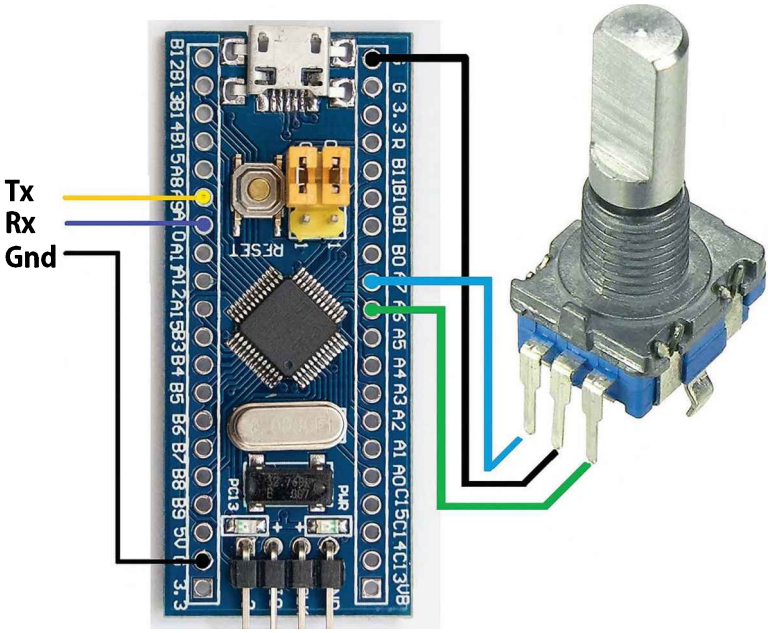


Figura 11.23

Inicializaremos el Timer con un valor 300, utilizando para ello el comando "***TIM_Period = 300***", que será el valor con el que se inicia el programa y que después aumentará o disminuirá dependiendo del sentido de giro en el que movamos el encoder.

A la vez, comparando el valor obtenido con el valor que se produjo anteriormente, se indicará si el giro del encoder es hacia la derecha o izquierda.

Por último, el valor obtenido y la dirección del giro se enviarán por el puerto USART1 hacia nuestro ordenador.

A continuación mostramos el código del ejemplo:

```
// Libreria principal del microcontrolador
#include "stm32f10x.h"

// Las librerias para los perifericos
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"

// Lista de funciones empleadas
void TIM4_Config(void);
void TIM4_IRQHandler(void);
void GPIO_Config(void);

/*             Modulo principal                    */
//=================================================
int main(void)
{
    TIM4_Config();  // Inicializamos el Timer4 y la interrupcion
    GPIO_Config();  // Configuramos el GPIO del LED

    while(1)
    {
        // No es necesario lineas de codigo adicionales
        // ya que la funcion que se ejecuta esta configurada
        // dentro de la funcion TIM4_IRQHandler
    }
}

/*  Funcion que configura el GPIO del LED en PC13    */
//-----------------------------------------------------------------
void GPIO_Config(void)
{
    /* Inicializamos pin conecctado LED PC13 */
    GPIO_InitTypeDef GPIO_InitStructure;

    // Iniciamos reloj para PORTC
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

    // Configura el pin PC13 como salida en push-pull para LED
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_ResetBits(GPIOC, GPIO_Pin_13); // Establecemos pin LED apagado

}

/*   Función que se ejecuta cuando se detecta el fin del periodo  */
//-----------------------------------------------------------------
void TIM4_Config(void)
{
    // Crea la estructura del TIMER_4
    TIM_TimeBaseInitTypeDef TIM_InitStructure;

    // Inicializa el reloj para el TIMER4
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

     detectar un desbordamiento cada vez que cuente
     un periodo de tiempo que será cada 1000 veces por segundo. */
    TIM_TimeBaseStructInit(&TIM_InitStructure);
    TIM_InitStructure.TIM_Prescaler = 8000;
    TIM_InitStructure.TIM_Period = 500;
    TIM_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM4, &TIM_InitStructure);

```

```
68      // Creamos la interrupcion para TIM4
69      TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
70
71      // Iniciamos el TIM4 programado
72      TIM_Cmd(TIM4, ENABLE);
73
74      /* Creamos la configu...n de la interrupcion TIM4_IRQn */
75      NVIC_InitTypeDef NVIC_InitStructure;
76      NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
77      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
78      NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
79      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
80      NVIC_Init(&NVIC_InitStructure);
81 }
82
83 /*   Funcion que se ejcuta cuando se detecta la interrupcion   */
84 void TIM4_IRQHandler(void)
85 {
86         if (TIM_GetITStatus(TIM4, TIM_IT_Update) != RESET)
87         {   // Resetea la bandera de la interrupcion
88             TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
89             // Blink el LED en PC13
90             GPIOC->ODR ^= GPIO_Pin_13;
91         }
92 }
93
```

Figura 11.24

11.3 CONTROL DEL WATCHDOG TIMER

WATCHDOG TIMER

El WDT (*WatchDog Timer*), literalmente "timer perro guardián", es un módulo encargado del procedimiento de seguridad de nuestro microcontrolador que permite que, cuando este se quede colgado en algún punto de la programación durante un determinado tiempo que se puede configurar, pueda detectarlo y resetear el sistema de forma automática e independiente para que se reinicie.

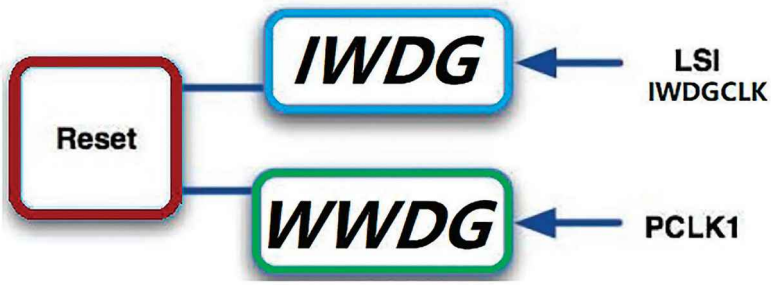


Figura 11.25

Nuestro microcontrolador posee dos: el **IWDG** (*Independent Watchdog*) y el **WWDG** (*Window Watchdog*). El IWDG está completamente aislado de la parte principal del microcontrolador. Integrado en un área con alimentación de respaldo y sincronizado con su propio reloj oscilador de baja frecuencia, el **LSI** (*low-speed internal RC*) de 32,768 kHz, que cuando el reloj principal falla, sigue funcionando, pudiendo trabajar incluso estando el microcontrolador en modo de suspensión. El WWDG está integrado con el sistema principal del microcontrolador y se sincroniza desde el bus del resto de periféricos internos, posee una funcionalidad mayor que el otro. Los dos se activan individualmente y pueden utilizarse simultáneamente.

El WDT es un temporizador de 12 bit con un divisor de 8 bits. Al iniciarse un programa, el watchdog timer está activo y configurado por defecto con un intervalo de reset de aproximadamente 32 ms. Para evitar esto, el usuario debe configurar o detener el WDT antes de la expiración del intervalo inicial.

El **WWDG**, es configurable como interrupción especial antes de que el sistema se restablezca, permitiendo su configuración con un intervalo de tiempo y un prescaler desde el reloj APB1, con el fin de detectar un comportamiento anómalo, temprano o tardío, de la aplicación que esté ejecutándose en ese momento.

Si el watchdog timer no se emplea en ninguna subrutina, puede ser configurado como un temporizador de intervalos y generar las interrupciones de tiempo que queramos, permitiendo su configuración en dos modos posibles: parada y espera.

11.3.1 EJEMPLO DE EMPLEO DEL IWDG

El **IWDG**, -modo independiente- utiliza un contador que podemos configurar, al que se le va restando un valor, utilizando para ello la frecuencia de reloj dividido por un prescaler y cuya cuenta se debe actualizar periódicamente para evitar que se envíe la orden de reiniciar el microcontrolador.

Para controlar este proceso posee dos registros. Si escribimos el valor 0xCCCC en el registro **IWDG_KR**, se iniciará una cuenta en el watchdog comenzando desde su valor de restablecimiento 0xFFF. Cuando dicho contador llegue a '0x000', se generará una señal de reposición **IWDG_RESET**. Siempre que escribamos en el registro **IWDG_KR** el valor 0xAAAA, el registro **IWDG_RLR** se volverá a cargar evitándose así el restablecimiento del watchdog.

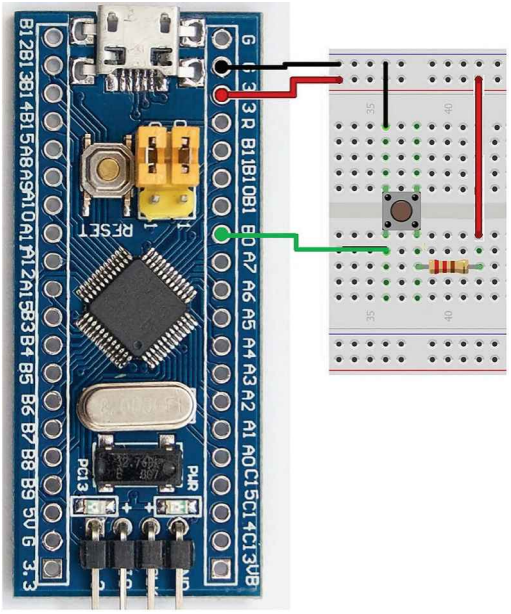


Figura 11.26

En el siguiente ejemplo, mostraremos el empleo del Watchdog en el modo **IWDG.**

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_iwdg.h"

/*    Funcion que configura los GPIO   */
//-----------------------------------------------
void GPIO_Config(void)
{
      /*  Nombramos la estructura que contendra los GPIOx  */
      GPIO_InitTypeDef GPIO_StructureInit;

      /* Activamos el clock para GPIOA y GPIOC    */
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

      /* Configuramos el pin PC13 para el LED  */
      GPIO_StructureInit.GPIO_Pin = GPIO_Pin_13;
      GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_StructureInit.GPIO_Mode = GPIO_Mode_Out_PP;
      GPIO_Init(GPIOC, &GPIO_StructureInit);

      /* Configuramos el pin PB0 para el pulsador  */
      GPIO_StructureInit.GPIO_Pin = GPIO_Pin_0;
      GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_StructureInit.GPIO_Mode = GPIO_Mode_IPU;
      GPIO_Init(GPIOB, &GPIO_StructureInit);
}

/*  Funcion que configura inicialmente el IWDG para que
    se reinicie el sistema cada 0,5 segundos
```

```
32      ------------------------------------------------------
33        T_out = 500ms / (32/32) = 500                      */
34  //------------------------------------------------------------
35  void IWDG_setup(){
36      IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
37      IWDG_SetPrescaler(IWDG_Prescaler_32); // Prescaler a 32(32 KHz/32 = 1 KHz)
38      IWDG_SetReload(500);                  // Recarga cada 500 ms -> 0,5 seg
39      IWDG_ReloadCounter();
40      IWDG_Enable();
41  }
42
43  /*  Funcion que permite reconfigurar el IWDG en un tiempo
44      determinado -en el ejemplo pre = 32, rlr = 4000
45      ------------------------------------------------------
46       T_out = Tiempo(ms) / (Frc_LSI * IWDG_Prescaler)
47       P.ej. 4000ms (4seg) Presc = 32
48       T_out = 4000ms / (32 * 32) = 4000                  */
49  //------------------------------------------------------------
50  void IWDG_ReInit(u8 prer, u16 rlr)
51  {
52      IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
53      IWDG_SetPrescaler(prer);
54      IWDG_SetReload(rlr);
55      IWDG_ReloadCounter();
56      IWDG_Enable();
57  }
58
59  /*                  Modulo Principal                    */
60  //**********************************************************
61  int main(void){
62
63    GPIO_Config();
64
65    GPIO_ResetBits(GPIOC, GPIO_Pin_13);  // Encendemos el LED con '0'
66
67    // Introducimos un retardo al iniciar el sistema */
68    for(int i=0;i<0x100000;i++);
69
70    /* Iniciamos el reloj LSI */
71    RCC_LSICmd(ENABLE);
72
73    /* Esperamos a que la bandera del reloj LSI este operativo */
74    while (RCC_GetFlagStatus(RCC_FLAG_LSIRDY) == RESET)
75    {}
76
77    /*  Configuramos el WatchDog para provocar un reinicio
78       del sistema cada 1 Segundo                         */
79    IWDG_setup();
80
81    /* Apagamos el LED para mostrar cuando se ha reiniciado
82       el sistema poniendo el pin a valor '1'             */
83    GPIO_SetBits(GPIOC, GPIO_Pin_13);
84
85        while(1){
86
87              // Comprueba cuando el pin PB1 esta a nivel bajo '0'
88              if(GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_0) == 0 )
89              {
90                  // Encendemos el LED
91                  GPIO_ResetBits(GPIOC, GPIO_Pin_13); // Desactiva el pin PC13
92
93                 // Reconfiguramos el IWDG para un retardo de unos 4 segundos
94                 IWDG_ReInit(IWDG_Prescaler_32,4000);
95              }
96              // Mientras el botón no se pulsa
97              else
98              {
99                 // Apagamos el led si no se ha pulsado el boton
100                GPIO_SetBits(GPIOC, GPIO_Pin_13);     // Activamos el pin PC13
101             }
102       }
103 }
104
```

Figura 11.27

Nuestro código se configura como siempre. Primero los pines que vamos a utilizar, creando la estructura para el pin PC13 como salida para el LED de pruebas y el pin PB0 como entrada en donde conectaremos un botón. A continuación, creamos dos funciones que contendrán la configuración de nuestro timer **IWDG**; donde la primera, **IWDG_setup()**, se cargará por defecto cada vez que se inicia el sistema con un "Time_out" de 500 ms (0,5 segundos), provocando que se reinicie automáticamente nuestro microcontrolador transcurrido ese tiempo; y donde la segunda función, **IWDG_ReInit** (*u8 prer*, *u16 rlr*), cargará un determinado valor en el contador (*rlr*) y en el prescaler (*prep*) cuando pulsemos el botón.

```
1  /*   Funcion que inicializa el IWDG  ------------------------*/
2  void IWDG_setup(){
3    IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
4    IWDG_SetPrescaler(IWDG_Prescaler_32);
5    IWDG_SetReload(500);      // Recarga cada 500ms (0,5 seg)
6    IWDG_ReloadCounter();
7    IWDG_Enable();
8  }
9
10 /*  Funcion que configura el IWDG con parametros que le pasemos */
11 void IWDG_ReInit(uint8_t prer, uint16_t rlr)
12 {
13     IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
14     IWDG_SetPrescaler(prer);
15     IWDG_SetReload(rlr);
16     IWDG_ReloadCounter();
17     IWDG_Enable();
18 }
```

Figura 11.28

Los registros **IWDG_PR** y **IWDG_RLR** están protegidos contra escritura. Por lo que la primera línea de la estructura que configura sus parámetros, debe ser un comando que nos permita manipular estos registros como el siguiente:

```
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
```

Figura 11.29

A continuación debemos establecer el factor de división o prescaler que corresponde al valor "*prer*", y que puede ser:

```
IWDG_SetPrescaler(IWDG_Prescaler_32)
```

Figura 11.30

Y en la siguiente línea establecemos el valor que deberá tener la recarga del contador del watchdog, que será el valor "*rlr*":

```
IWDG_SetReload(500);
```

Figura 11.31

Supongamos que necesitamos generar un tiempo de restablecimiento transcurridos unos 0.8 segundos -800 ms-; para saber el valor que deberá tener el contador, será necesario calcular el tiempo con la siguiente fórmula:

Valor del contador a recargar en IWDG = 800 ms / (Frec_LSI kHz / IWDG_ Prescaler)

```
Frec_LSI = 32 kHz  ;   IWDG_Prescaler = 32
                Valor del recarga IWDG = 800 ms / ( 32 / 32)
                          ReloadValue  = 800 / 1
                                       = 800
```

Figura 11.32

Por último se debe dar la orden de recargar los valores nuevos programados y la de iniciar el watchdog, introduciendo las dos líneas siguientes en nuestro código:

```
IWDG_ReloadCounter();
IWDG_Enable();
```

Figura 11.33

En nuestro programa de ejemplo, al iniciarse automáticamente el microcontrolador, se carga un contador de reseteo mediante el IWDG a los 0,5 segundos (500 ms), encendiéndose el LED en PC13 y apagándose inmediatamente para avisar de que se ha iniciado. Transcurrido ese tiempo y si no se ha recargado el contador, se reiniciará constantemente cada vez que el contador llegue a '0'.

Pero, si pulsamos el botón que está en el pin PB0, el sistema llamará a la segunda función, **IWDG_ReInit**(IWDG_Prescaler_32, 4000). Esta reconfigura el IWDG, donde mandamos que se configure ahora con 4000 ms lo que produce que el sistema esperará durante 4 segundos para reiniciarse, tras lo cual, se reconfigurará con el tiempo establecido por defecto en nuestra programación inicial de 0,5 segundos.

Verificación del motivo del reinicio

Cuando se produce un reinicio es posible saber el motivo que lo ha causado; mediante el comando **RCC_GetFlagStatus**() se puede leer el valor de alguno de los marcadores o 'banderas' del registro de estado de control **CSR** (*Control and Status Register*) y saber el motivo.

Los siguientes marcadores pueden usarse para conocer el motivo, utilizando la siguiente forma de código:

```
if (RCC_GetFlagStatus (RCC_FLAG_IWDGRSTF)) ...
```

RCC_FALG_LPWRRSTF – Reinicio por un fallo de alimentación del sistema.
RCC_FLAG_WWDGRSTF – A través del sistema de vigilancia de ventanas.
RCC_FLAG_IWDGRSTF – A través del sistema de vigilancia independiente.
RCC_FLAG_SFTRSTF – Reinicio activado por el programa
RCC_FLAG_PORRSTF – Al encender o apagar la fuente de alimentación.
RCC_FLAG_PINRSTF – Por llamada del hardware.

Figura 11.34

Todos los bits de este registro, cuando no reciben una señal, tienen un valor predeterminado de '0'. Si reciben una señal de reinicio, el bit correspondiente cambiará a 1. Por lo tanto, después de leer el marcador y reconocer el motivo del reinicio, debe reiniciarse mediante la función **RCC_ClearFlag**().

11.3.2 EJEMPLO DE EMPLEO DEL WWDG

El módulo **WWDG** o watchdog, se utiliza a menudo para supervisar fallos de nuestro software o fallos de condiciones lógicas inesperadas que desvían o alteran la secuencia de ejecución normal; a menos que el valor del contador del módulo WWDG se actualice antes que se ponga a '0'. El circuito de vigilancia generará un reinicio del sistema cuando se alcance un periodo de tiempo preestablecido.

EL contador del **WWDG** opera de modo que necesita ser actualizado, ni antes ni después de los márgenes que tiene establecido, que va desde el valor 127 (0x7F) al 64 (0x40). El sistema realiza una cuenta atrás del 127 al 64 y genera un restablecimiento cuando el valor del contador desciende del valor 64 (0x40) a 63 (0x3F); en ese restablecimiento se genera una interrupción que se puede utilizar mediante una subrutina o función de servicio, para realizar algún proceso que queramos antes de que se reinicie el sistema. También se puede generar un restablecimiento por software, si el valor del contador se vuelve a cargar antes de que dicho contador descienda del valor límite o el valor establecido como ventana. Este

valor del contador, cargado por el software, se deberá cargar a intervalos regulares durante el funcionamiento del sistema para evitar un reinicio y siempre y cuando el valor del contador sea menor que el valor de ventana establecido. Este valor deberá estar entre 255 (0xFF) y 192 (0xC0).

Para establecer la configuración del WWDG son necesarios calcular tres valores, que nombraremos *'tr'*, *'wr'* y *'presc'*:

El valor *'tr'*, es el valor que deberá tener el contador del WWDG (*WWDG_counter_code*), que se configura con el comando **WWDG_SetCounter**(*Counter*) y se calcula mediante la división de la frecuencia de entrada al reloj del WWDG, que en nuestro microcontrolador es de 36 MHz de máximo, entre su resolución 4096 para 16 bits y el resultado se divide nuevamente por el prescaler que escojamos: [(PCLK1(36 MHz) / 4096) / WWDG_Prescaler]. Aunque, se suele dejar con el valor máximo establecido por las especificaciones de nuestro microcontrolador, 127 (0x7F).

El otro valor a calcular 'wr', es el tiempo de espera del WWDG, que se configura mediante el comando **WWDG_SetWindowValue**(*WindowValue)* y que será el valor de la ventana WWDG que se comparará con el contador descendente y cuyo valor debe ser superior a 64 (0x40) e inferior a 128 (0x80). Este valor es el umbral que se establecerá en nuestra programación para que el sistema de watchdog se actualice solo cuando el contador descendente se encuentre por debajo de ese valor.

El *'presc'* es el valor de prescaler que vamos a utilizar, se configura con el comando **WWDG_SetPrescaler**(*WWDG_Prescaler*) y además, deberá corresponder con el valor de prescaler que hayamos utilizado en los cálculos previos.

En el ejemplo siguiente veremos cómo configurar y programar el uso del modo ventana o **WWDG** (*Window Watchdog)*.

Configuraremos la estructura de GPIO para establecer el pin de salida para el LED de prueba en PC13, que se encenderá y apagará cada vez que se detecte la interrupción que produce al llegar el contador WWDG al umbral que hayamos establecido. Provocándose así el reseteo que hemos establecido mediante la configuración en **NVIC** y del **WWDG**.

Existen unos valores mínimos y máximos para los tiempos a establecer del WWDG que se indican en el manual de referencia de nuestro microcontrolador y que se muestran en la tabla de la Figura 11.35.

Table 98. Minimum and maximum timeout values @36 MHz (f_{PCLK1})

Prescaler	WDGTB	Min timeout value	Max timeout value
1	0	113 µs	7.28 ms
2	1	227 µs	14.56 ms
4	2	455 µs	29.12 ms
8	3	910 µs	58.25 ms

Figura 11.35

Supongamos que necesitamos configurar nuestra interrupción del WWDG para que se restablezca transcurridos un tiempo de espera de 45 microsegundos:

Lo primero será calcular el valor del contador (*WWDG_clock_counter)* mediante las siguientes ecuaciones que se indican en la Figura 11.36:

a)

$$WWDG_clock_counter_{(\mathrm{Hz})} = \left((PCLK1 = 36\,MHz)/4096\right)/Prescaler$$

b)

$$\frac{1}{WWDG_clock_counter} = WWDG_clock_counter_{(\mu s)}$$

Figura 11.36

Donde en la ecuación a) calculamos el valor en Hercios (Hz) para a continuación, con la ecuación b) obtener el valor en microsegundos (µs) de nuestro microcontrolador; es decir, el valor de 36 MHz dividido entre 4096 y dividido todo entre un prescaler seleccionado:

a)

$$WWDG_clock_counter_{(\mathrm{Hz})} = \left(\frac{36\,000\,000}{4096}\right)/8 = 1\,098{,}6\,Hz$$

b)

$$WWDG\ counter\ code = \frac{1}{1\,098{,}6\,Hz} = 910\ (\mu s)$$

Figura 11.37

Si establecemos el valor de ventana en 80, estamos estableciendo que nuestro watchdog debe ser actualizado cuando el contador esté por debajo de 80 y por encima de 64, porque de lo contrario, se generará un restablecimiento y con ello un reseteo del sistema.

Si establecemos el valor de recarga del contador en 127, realizaremos el siguiente cálculo para averiguar en cuánto tiempo se generará el restablecimiento.

$$WWDG_c_c \sim 910\,\mu s * 64 = 58{,}2\,ms$$

Figura 11.38

El resultado nos indica que nuestro Watchdog se restablecerá cada 58,2 ms como tiempo máximo.

A continuación, detallamos en la Figura 11.39, el grupo de comandos que constituyen la estructura con la configuración del WWDG.

También creamos la estructura del NVIC que inicializa y configura los parámetros de la interrupción "WWDG_IRQn" que habilitamos con el comando "**WWDG_EnableIT**()".

```
1   /*  Inicializamos el reloj del Watchdog WDG    */
2   RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG,ENABLE);
3
4   WWDG_DeInit();    // Inicializamos el Watchdog
5
6   WWDG_SetPrescaler(WWDG_Prescaler_8); // Valores de Prescaler 1, 2, 4, 8
7
8   WWDG_SetWindowValue(0x50);           // (80) valor debe ser < 127
9
10  WWDG_Enable(0x7F);                   // (127) Valor entre 64 <--> 127
11
12  WWDG_ClearFlag();   // Borramos y reiniciamos la bandera del WWDG
13
14  WWDG_EnableIT();    // Habilitamos la interrupcion WWDB
15
16  /*  Iniciamos el NVIC para la interrupcion   */
17  NVIC_InitTypeDef NVIC_InitStructure;
18  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
19  NVIC_InitStructure.NVIC_IRQChannel = WWDG_IRQn;    /*Interrupcion WWDG */
20  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
21  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
22  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
24  NVIC_Init(&NVIC_InitStructure);
```

Figura 11.39

Como indicamos, el WWDG posee su propio reloj, por lo que, con la primera línea lo activamos:

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);
```

Figura 11.40

Establecemos el valor de prescaler o divisor que queremos.

```
WWDG_SetPrescaler (uint32_t WWDG_Prescaler) ;
```

Figura 11.41

Y, seguidamente, introducimos el valor que deberá tener el contador que lo inicializa.

```
WWDG_SetWindowValue (uint8_t WindowValue) ;
```

Figura 11.42

La siguiente línea configurará el valor inicial del contador y habilitará el WWDG.

```
WWDG_Enable(uint8_t Counter);
```

Figura 11.43

También necesitaremos en nuestra configuración del watchdog, una función de servicio que detecte la interrupción y produzca la ejecución del código que queramos. Figura 11.44.

```
1  void WWDG_IRQHandler(void) {
2
3    WWDG_ClearFlag();       // Reseteamos la bandera del WWDG
4    WWDG_SetCounter(100);   // (101) Valor entre 64 <--> 127
5
6    /* Toggle LED en PC13*/
7    GPIOC->ODR ^= GPIO_Pin_13;
8  }
```

Figura 11.44

Se resetea la bandera del WWDG mediante el comando "**WWDG_ClearFlag()**" y recargamos el contador nuevamente para que se vuelva a reproducir

con el comando "**WWDG_SetCounter**()", creando además una línea que encienda y apague el LED de pruebas de nuestra placa, para indicar que la interrupción ha sido detectada.

En nuestro código de ejemplo, en el módulo principal "*main*", añadimos además las siguientes líneas en el bucle infinito, que muestran: si se restablece el valor del contador antes de que este se restablezca, no se producirá el reseteo del sistema.

```
1        .
2        .
3        .
4   Delay_ms(91);            // < 91 Se produce el reseteo
5   WWDG_SetCounter(100);  // Valor debe estar entre 0x40(64) y 0x7F(127)
```

Figura 11.45

A modo de ejercicio de comprobación, podemos deshabilitar esas líneas, volver a compilar y cargar el firmware en nuestra placa. Observaremos cómo ahora, el watchdog se ejecutará reseteando nuestro sistema cada cierto periodo de tiempo y mostrando cómo se enciende y apaga el LED de pruebas indicándolo.

Se muestra a continuación el código completo del ejercicio explicado.

```
1   #include "stm32f10x.h"
2   #include "stm32f10x_gpio.h"
3   #include "stm32f10x_rcc.h"
4   #include "stm32f10x_wwdg.h"
5   #include "misc.h"
6
7   #include "delay.h"   // Llama a la libreria de 'delay'
8
9   void LED_Init(void);
10
11  /*   Funcion de servicio que se ejecuta cuando se produce
12       el restablecimiento                                    */
13  //-------------------------------------------------------------
14  void WWDG_IRQHandler(void) {
15
16    WWDG_ClearFlag();       // Reseteamos la bandera del WWDG
17    WWDG_SetCounter(100);  // (101) Valor entre 64 <--> 127
18    /* Toggle LED en PC13*/
19    GPIOC->ODR ^= GPIO_Pin_13;
20  }
21
22  /*                  Modulo Principal                  */
24  //-------------------------------------------------------
25  int main(void)
26  {
27    LED_Init();      // Inicializamos el GPIO del LED de pruebas
28
29    DelayInit();     // Inicializamos la libreria 'delay'
30
31    /*  Inicializamos el reloj del Watchdog WDG    */
32    RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG,ENABLE);
33
34    WWDG_DeInit();  // Inicializamos el Watchdog
35
36    /* Donde configuramos el WWDG con los siguientes valores:
37      WWDG_clock_counter = (PCLK1 (36MHz)) /4096)/8 = 1098,6 Hz (~910 us)
38      ---------------------------------------------------------------*/
```

```
  WWDG_SetPrescaler(WWDG_Prescaler_8); // Valores de Prescaler 1, 2, 4, 8

  /* Valor de Ventana en 0x50 (80 decimal), que significa que el
     contador WWDG debe actualizarse solo cuando el contador
     este por debajo de 0x50 (80) y por encima de 0x40 (64),
     de lo contrario se generara un restablecimiento y un
     reseteo del sistema.
   ------------------------------------------------------------*/
  WWDG_SetWindowValue(0x50); // (80) valor debe ser < 127

  /* Establecemos el tiempo de restablecimiento:
     El contador del WWDG  se establece en 0x7F (127) los
     que se genera un timeout de = ~910 us * 64 = 58.2 ms
     ------------------------------------------------------*/
  WWDG_Enable(0x7F);   // (127) Valor entre 64 <--> 127

  WWDG_ClearFlag();    // Borramos y reiniciamos la bandera del WWDG

  WWDG_EnableIT();    // Habilitamos la interrupcion WWDB

  /*  Iniciamos el NVIC para la interrupcion   */
  NVIC_InitTypeDef NVIC_InitStructure;
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
  NVIC_InitStructure.NVIC_IRQChannel = WWDG_IRQn;    /*Interrupcion WWDG */
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);


while (1)
  {
    /*  Mientras se restablezca el contador en las siguientes
        lineas, antes de que se produzca el restablecimiento
        no se producirá el reseteo del sistema.

        Si desabilitamos las dos líneas, o bajamos el valor
        del retraso generado con el delay a un valor que deje
        que se restablezca el contador WWDG, se producira
        el restablecimiento del WWDG automáticamente y veremos
        encenderse el LED en PC13 indicando que se esta produciendo
        el restablecimiento.
        ------------------------------------------------------------*/
    DelayMs(91);           // < 91 Se produce el reseteo
    WWDG_SetCounter(100);  // Valor debe estar entre 0x40(64) y 0x7F(127)
  }
}

/*     Funcion que configura el GPIO del LED de pruebas         */
//-----------------------------------------------------------------
void LED_Init(void)
{
   /* Creamos la estructura GPIO */
   GPIO_InitTypeDef  GPIO_InitStructure;

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC , ENABLE);

   /* Configuramos el pin PC13 para el LED  */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
   GPIO_Init(GPIOC, &GPIO_InitStructure);

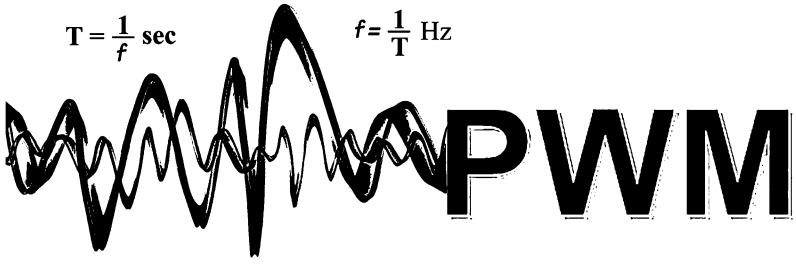
   // Apagamos el LED al inicio
   GPIO_SetBits(GPIOC, GPIO_Pin_13);
}

```

Figura 11.46

12

PROGRAMACIÓN PWM



Nuestro microcontrolador también puede generar frecuencias de salida para ser utilizadas como control de determinados dispositivos o simplemente generar sonidos.

La Modulación de Ancho de Pulso (**PWM** -*Pulse Width Modulation*) es el uso de una salida digital de nuestro microprocesador para simular una señal analógica. Esta señal se produce mediante el control del tiempo que ese pulso permanece activo –ciclo de trabajo (*Duty cycle*)- y la cantidad de veces que se activa.

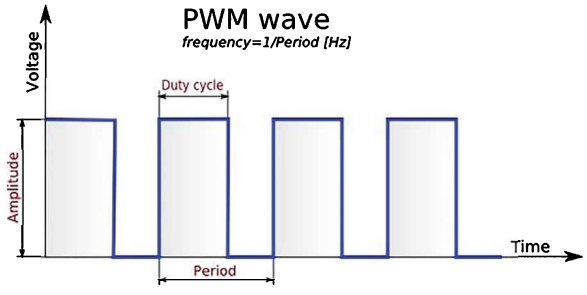


Figura 12.1

Para ello se utiliza alguno de los timer de propósito general, que se explicaron anteriormente, configurando el tiempo que permanece activa la señal del pulso generado (*Duty_cycle*) expresándolo en valores de porcentaje: donde el valor '0' significa sin señal y el valor '100' nivel alto todo el tiempo.

Por otro lado, se establece el valor en que se repite esa señal en un determinado periodo de tiempo (TIM_Period).

Por ejemplo, si enviamos una señal positiva durante 1 segundo, y a continuación desconectamos la señal durante 99 segundos, el ciclo de trabajo que tenemos será: (1 / 99 *100)= 1%, con lo que el ciclo de trabajo será de 1 % del tiempo encendido en positivo.

Supongamos que queremos configurar una señal PWM con una frecuencia de 75 kHz:

Lo primero que necesitamos para configurar nuestro Timer es encontrar un valor del **TIM_Period** para generar la frecuencia de esos 75 kHz; para lo que aplicamos la siguiente fórmula que vimos anteriormente, en el capítulo 11.

$$TIMER_Period = (Timer_clock \ / \ FREC_Evento)$$

$$TIMER_Period = (72\ 000\ 000 \ / \ 75\ 000) = 960$$

Figura 12.2

Del resultado, extraemos los valores necesarios para configurar el **TIMER_ Period** (ARR) = (960 - 1) y el **TIM_Prescaler** (PSC) = 0, que serán los valores que colocaremos en la configuración de nuestro **TIMER**.

```
1  /*     Se crea la estructura del TIMER_3                    */
2  //-------------------------------------------------------------
3  TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
4  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
5
6  TIM_TimeBaseStructure.TIM_Period = 959;
7  TIM_TimeBaseStructure.TIM_Prescaler = 0;
8  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
9  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
10 TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
```

Figura 12.3

El siguiente paso, será establecer la estructura con los parámetros necesarios para generar la señal **PWM**; para ello la librería CMSIS nos establece el formato que se muestra en la Figura 12.4.

```
1  /*     Configuracion pulso PWM TIMER3_Canal4                    */
2  //-------------------------------------------------------------------
3  TIM_OCInitTypeDef TIM_PWM_InitStructure;
4
5  TIM_PWM_InitStructure.TIM_Pulse = 10;
6  TIM_PWM_InitStructure.TIM_OCMode = TIM_OCMode_PWM1;
7  TIM_PWM_InitStructure.TIM_OutputState = TIM_OutputState_Enable;
8  TIM_PWM_InitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
9  TIM_OC4Init(TIM3, &TIM_PWM_InitStructure);   // TIM_OC4 = TIM3_CH4
10
11 TIM_Cmd(TIM3, ENABLE);                       // Timer 3
```

Figura 12.4

Mediante el comando **TIM_OCInitTypeDef** especificamos el nombre que le daremos a nuestra estructura, que en nuestro ejemplo nombramos como *TIM_PWM_InitStructure*.

Posee los siguientes parámetros:

TIM_OCIdleState	TIM_OCMode	TIM_OCNidleState
TIM_OCNPolarity	TIM_OCPolarity	TIM_OutputNSate
TIM_OutputState	TIM_Pulse	

El primer parámetro a configurar es el **TIM_Pulse**, valor del ciclo de trabajo en porcentaje; cuyo valor deberá estar entre 0 y 65535 (0x0 y 0xFFFF). Para calcularlo usamos la siguiente fórmula:

$$TIM_Pulse = \left(\frac{(TIM_Period + 1) \times PWM\ DutyCycle}{100} \right) - 1$$

Figura 12.5

En nuestro ejemplo, si queremos que el ciclo de trabajo (*Duty_cycle*) sea un 25%:

TIM_Pulse = (((959+1) * 25) / 100) **-1** = 239

Figura 12.6

En el caso de querer un ciclo de trabajo del 75%:

TIM_Pulse = (((959 +1) * 75) / 100) -1 = 719

Figura 12.7

12.1 EJEMPLO DE SEÑAL PWM

En nuestro siguiente código de ejemplo, configuramos el canal 1 del timer 1, en el pin PA8, para que cree señales PWM seleccionando diferentes frecuencias y a la vez variando el ciclo de trabajo (*Duty cycle*) de dicha señal, generando así un aumento o disminución del brillo de un led conectado a dicha salida.

Si conectamos un osciloscopio o un frecuencímetro, podremos medir dicho pin de salida y comprobar las diferentes frecuencias.

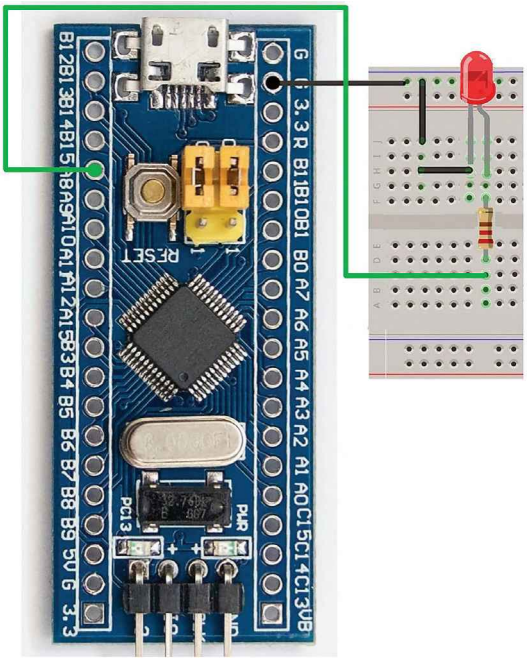


Figura 12.8

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"

/*       Funcion que configura el TIMER y el pulso PWM    */
//-------------------------------------------------------------
void TIM1_PWM_Config(u16 arr,u16 psc, u16 pulse)
{
   /*  Creamos las estructuras para GPIO, TIM y PWM      */
   GPIO_InitTypeDef GPIO_InitStructure;
   TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
   TIM_OCInitTypeDef  TIM_OCInitStructure;

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
                          RCC_APB2Periph_TIM1, ENABLE);
   //Configuracion del pin PA8 canal 1 del TIM1
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_Init(GPIOA, &GPIO_InitStructure);

   //Configuracion TIM1 ------------------------------------
   TIM_TimeBaseStructure.TIM_Period = arr;
   TIM_TimeBaseStructure.TIM_Prescaler = psc;
   TIM_TimeBaseStructure.TIM_ClockDivision = 0;
   TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
   TIM TimeBaseInit(TIM1, &TIM TimeBaseStructure);

   // Configuracion PWM ------------------------------------
   TIM_OCInitStructure.TIM_Pulse = (uint16_t)(((arr+1)*pulse)/100)-1;
   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;
   TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
   TIM_OCInitt(TIM1, &TIM_OCInitStructure); // Canal 1 TIM1


   // Iniciamos el modulo de salida PWM
   TIM_CtrlPWMOutputs(TIM1,ENABLE);

   //
   TIM_OC1PreloadConfig(TIM1, TIM_OCPreload_Enable);

   TIM_ARRPreloadConfig(TIM1, ENABLE);

   // Iniciamos el TIM1
   TIM_Cmd(TIM1, ENABLE);
}

/*          Modulo   Principal           */
//=============================-------=====
int main(void)
{
  // Diferentes configuraciones para probar diferentes frecuenci--
  // ------------------------------------------------------------------
  //TIM1_PWM_Config(7199, 999, 50); // 10 Hz 50%
  //TIM1_PWM_Config(2399, 999, 25); // 30 Hz 25%
  TIM1_PWM_Config(3599, 0, 15);     // 20 kHz 15%
  //TIM1_PWM_Config(959, 0, 25);    // 75 kHz 25%


   while(1) {
    ;;
   }
}
```

Figura 12.9

En el código del ejemplo, vemos la configuración básica para establecer una señal PWM con nuestra placa.

Dentro de la función **TIM1_PWM_Config()** creamos previamente la estructura para el GPIO del pin PA8 que corresponde al canal 1 del Timer 1, que es el que vamos a emplear.

El siguiente módulo será crear la estructura que contendrá los parámetros que ya conocemos para el Timer y en donde utilizamos las variables arr y psc que le pasamos al llamar la función; y que asignamos al **TIM_Period** = *arr* y al **TIM_Prescaler** = *psc*.

En nuestro siguiente módulo, establecemos los parámetros que deberá tener nuestra señal PWM y que su creación se inicia con el comando "**TIM_OCInitTypeDef**".

Contiene los siguientes parámetros:

TIM_OCIDleState	TIM_OCMode	TIM_OCNIdleState
TIM_OCPolarity	TIM_OutputNState	TIM_OutputState
TIM_Pulse		

EL primero a configurar es el ciclo de trabajo de la señal PWM, para lo que se usa el comando "**TIM_Pulse**", que explicamos al principio de este capítulo.

Luego, mediante el comando "**TIM_OCMode**" establecemos el modo de funcionamiento del registro de captura/comparación del timer (**TIMx_CCER** – *Capture/compare Enable Register*); que podemos establecer en el modo PWM1 en forma de activo a inactivo y en el modo PW2 de inactivo a activo.

En la línea siguiente establecemos, mediante el comando "**TIM_OutputState**", el estado del comparador de salida; que puede establecerse como activo con "**TIM_OutputState_Enable**" y desactivo con "**TIM_OutputState_Disable**".

Con el comando "**TIM_OCPolarity**" indicamos la polaridad de salida que podrá ser positiva "**TIM_OCPolarity_High**" o negativa "**TIM_OCPolarity_Low**".

A continuación, iniciamos el módulo PWM con el comando "**TIM_CtrlMOutpust**(*TIMx*, ENABLE).

También el comando "**TIM_OC[n]PreloadConfig**(*TIMx*, **TIM_OCPreload_Enable**)", con el que activamos el registro [n] de precarga del TIMx.

Cargamos el registro de precarga ARR del timer seleccionado con el comando "**TIM_ARRPreloadConfig**(*TIMx*, ENABLE)".

De esta manera comenzará a funcionar nuestro Timer con la frecuencia que hemos configurado generando una señal PWM.

Podemos utilizar un medidor de frecuencias, osciloscopio o simplemente un altavoz para comprobar la señal generada.

12.2 EJEMPLO DE SEÑAL PWM CONTROLANDO EL BRILLO DE UN LED

En nuestro siguiente ejemplo, vamos a controlar el nivel de brillo de un LED conectado al pin PB0, a través de dos pulsadores que controlan su aumento o disminución mediante el control del ciclo de trabajo (*Duty cycle*), conectados a los pines PA0 y PA1.

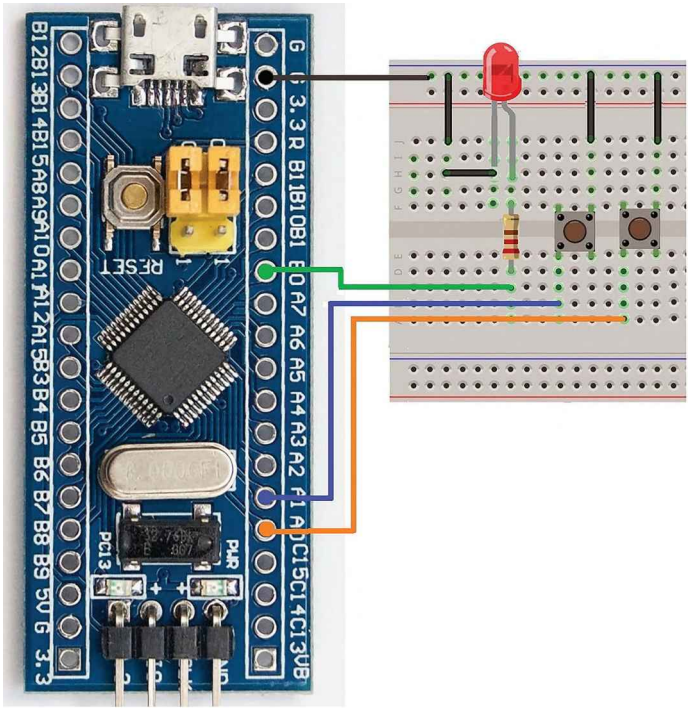


Figura 12.10

Código del ejemplo:

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"

// Definimos limite max. PERIODO
#define PERIOD 1000

/*  Modulo principal incluidas las configuraciones GPIO y PWM  */
//=====================================================================
int main(void)
{
  int TIM_Pulse = 10;
  int i;

  /* Configuramos GPIO para el LED y Pulsadores  */
  GPIO_InitTypeDef GPIO_InitStructure;

  // Iniciamos los clock para cada puerto
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);   // TIM3 Clock

  /*    BOTONES  Pines de entrada PA0 & PA1   */
  GPIO_StructInit(&GPIO_InitStructure);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  /*     LED     Pin de salida en PB1          */
  GPIO_StructInit(&GPIO_InitStructure);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;          // PB0 -> TIM3_CH3
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  /*     Configuracion del TIM                */
  TIM_TimeBaseInitTypeDef TIMER_InitStructure;
  TIM_TimeBaseStructInit(&TIMER_InitStructure);
  TIMER_InitStructure.TIM_Prescaler = 800;
  TIMER_InitStructure.TIM_Period = PERIOD;
  TIMER_InitStructure.TIM_ClockDivision = 0;
  TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM3, &TIMER_InitStructure);

  /*      Configuracion pulso PWM  CH4      */
  TIM_OCInitTypeDef TIM_PWM_InitStructure;
  TIM_OCStructInit(&TIM_PWM_InitStructure);
  TIM_PWM_InitStructure.TIM_Pulse = TIM_Pulse;
  TIM_PWM_InitStructure.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_PWM_InitStructure.TIM_OutputState = TIM_OutputState_Enable;
  TIM_PWM_InitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OC3Init(TIM3, &TIM_PWM_InitStructure);  //  TIM_OC3 = TIMx_CH3

  TIM_Cmd(TIM3, ENABLE);      // Timer 3

  while(1)
  {
      /* Comprueba pulsacion de los botones */
      // Boton PA0
      if (GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0) == 0) {
```

```
65              // Comprueba si se alcanza el limite superior
66              // PERIOD=1000 si no aumenta uno
67              if (TIM_Pulse < PERIOD)
68                 TIM_Pulse++;
69                 TIM3->CCR3 = TIM_Pulse;      //  TIMx->CCR3 = TIMx_
70        }
71        // Boton PA1
72        if (GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_1) == 0) {
73              // Comprueba si se alcanza el 0
74              // Si es mayor > 0 disminuye uno
75              if (TIM_Pulse > 0)
76                 TIM_Pulse--;
77                 TIM3->CCR3 = TIM_Pulse;      //  TIMx->CCR3 = TIMx_
78        }
79
80        /* Linea que genera un retardo delay */
81        for(i=0;i<0x10000;i++);
82      }
83  }
84
```

Figura 12.11

Definimos primero la variable "PERIOD" con un valor fijo de 1000 que nos procurará un valor límite en su parte superior para el valor **TIM_Period**.

Después, creamos la variable TIM_Pulse que inicializamos con 100, y que luego empleamos para incrementar o disminuir mediante la pulsación de los pulsadores.

En nuestro código, tenemos un primer módulo donde creamos la estructura GPIO necesaria para los parámetros de los pines y puertos a emplear, correspondientes a los puertos GPIOB, GPIOA y el TIM3 que hemos seleccionado.

Por último, en nuestro módulo principal, creamos unas líneas de comando que comprueban si se ha pulsado alguno de los botones; si se detecta que se ha pulsado el botón del pin PA0, el valor de la variable TIM_Pulse se aumenta y si se pulsa el botón del pin PA1 el valor disminuye.

Para pasar el valor de la variable TIM_Pulse, al valor real del contador, utilizamos el acceso directo al registro mediante la instrucción TIMx->CCR[n] donde especificamos que [n] es el canal 4 del TIM3.

Por último, introducimos un simple contador con el comando C++ "for…" para producir un pequeño retardo en la ejecución del programa.

12.3 EJEMPLO PWM CONTROL DE BRILLO DE UN LED TRICOLOR

En el siguiente ejemplo emplearemos tres canales para generar las señales PWM a los que estarán conectados los pines de un LED tricolor.

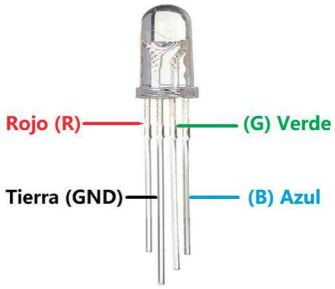


Figura 12.12

Como ya seguramente conocen, el funcionamiento de un LED Tricolor o LED RGB es algo distinto al de los led normales con un color ya definido. Estos en función del nivel de tensión que se entregue en cada una de las patas de este tipo de LED, producirán diferentes combinaciones de tonalidades de color, mezcla de los tres colores básicos (rojo, verde y azul).

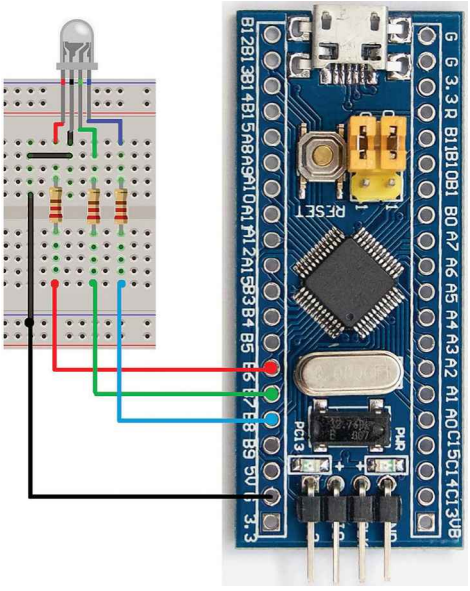


Figura 12.13

Si solo aplicamos tensiones de 0 y 5 Vcc, las combinaciones posibles serán las siguientes.

- R = 0 | G = 0 | B = 0 – Negro (apagado)
- R = 0 | G = 0 | B = 1 – Azul
- R = 0 | G = 1 | B = 0 – Verde
- R = 0 | G = 1 | B = 1 – Turquesa
- R = 1 | G = 0 | B = 0 – Rojo
- R = 1 | G = 0 | B = 1 – Morado
- R = 1 | G = 1 | B = 0 – Amarillo verdoso
- R = 1 | G = 1 | B = 1 – Blanco

Mediante la generación de señales PWM, simulamos una señal analógica en las salidas digitales de nuestro microcontrolador. De modo que vamos variando el ancho del pulso generado mediante el aumento o disminución del valor del TIM_ Pulse en la configuración de una señal PWM, dando valores distintos a cada uno de los pines que conectaremos a los pines de color de nuestro LED RGB.

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"

void espera(unsigned int nCount);

#define PERIOD 1000

/*        Modulo principal y configuracion de GPIO y TIM    */
//=============================================================
int main(void)
{
      int TIM_Pulse_R = 0;
      int TIM_Pulse_G = 0;
      int TIM_Pulse_B = 0;

      /*  Creamos las estructuras GPIO y TIM  */
      GPIO_InitTypeDef GPIO_InitStructure;

      /* Activamos los clock correspondientes */
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
      RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

      /* Configuramos los GPIO de los pines de salida
            PB6_Red  PB7_Green  PB8_Blue  */
      GPIO_StructInit(&GPIO_InitStructure);
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
      GPIO_Init(GPIOB, &GPIO_InitStructure);

      /*   Configuracion del TIMER_4   */
      TIM_TimeBaseInitTypeDef TIMER_InitStructure;
      TIM_TimeBaseStructInit(&TIMER_InitStructure);
      TIMER_InitStructure.TIM_Prescaler = 720;
      TIMER_InitStructure.TIM_Period = PERIOD;        // Periodo(1000) Max
      TIMER_InitStructure.TIM_ClockDivision = 0;
      TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
      TIM_TimeBaseInit(TIM4, &TIMER_InitStructure);

```

```
42        /*  Configuracion del PWM   */
43        TIM_OCInitTypeDef TIM_PWM_InitStructure;
44        TIM_OCStructInit(&TIM_PWM_InitStructure);
45        TIM_PWM_InitStructure.TIM_Pulse = 0;  // 000% de Duty cycle (Apagado)
46        TIM_PWM_InitStructure.TIM_OCMode = TIM_OCMode_PWM1;
47        TIM_PWM_InitStructure.TIM_OutputState = TIM_OutputState_Enable;
48        TIM_PWM_InitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
49        // Cargamos en cada canal del TIM4 el valor TIM_Pulse = 0 inicial
50        TIM_OC1Init(TIM4, &TIM_PWM_InitStructure);   // TIM4_CH1 PB6
51        TIM_OC2Init(TIM4, &TIM_PWM_InitStructure);   // TIM4_CH2 PB7
52        TIM_OC3Init(TIM4, &TIM_PWM_InitStructure);   // TIM4_CH3 PB8
53
54      TIM_Cmd(TIM4, ENABLE);
55
56      while(1)
57      {
58        // Incrementamos TIM_Pulse de la señal Rojo
59        TIM_Pulse_R++;
60        // Comprobamos que no se supera el umbral de la variable 'PERIOD'
61        if (TIM_Pulse_R > PERIOD)
62            TIM_Pulse_R = 0;
63        // Incrementamos TIM_Pulse de la señal Verde
64        TIM_Pulse_G+=2;
65        if (TIM_Pulse_G > PERIOD)
66            TIM_Pulse_G = 0;
67        // Incrementamos TIM_Pulse de la señal Azul
68        TIM_Pulse_B+=4;
69        if (TIM_Pulse_B > PERIOD)
70           TIM_Pulse_B = 0;
71
72        // Enviamos los valores a los registros de cada color
73        // TIM_PWM_InitStructure.TIM_Pulse = TIM_Pulse_RGB
74        // TIM_OCxInit(TIM3, &TIM_PWM_InitStructure)
75        TIM4->CCR1 = TIM_Pulse_R; // TIM4_CH1 PB6
76        TIM4->CCR2 = TIM_Pulse_G; // TIM4_CH2 PB7
77        TIM4->CCR3 = TIM_Pulse_B; // TIM4_CH3 PB8
78
79        // Realizamos un 'delay'
80        espera(200);
81      }
82  }
83
84  // Funcion de espera
85  void espera(unsigned int nCount)
86  {
87        unsigned int i, j;
88        for (i = 0; i < nCount; i++)
89              for (j = 0; j < 0x2AFF; j++);
90  }
91
```

Figura 12.14

En el ejemplo, lo primero que hacemos es crear las variables TIM_Pulse_R, TIM_Pulse_G y TIM_Pulse_B, que contendrán los valores del TIM_Pulse que luego enviaremos a cada pin correspondiente.

Creamos con el primer módulo la configuración GPIO de los pines de salida, donde conectaremos cada una de las entradas a los colores del LED: PB6 para rojo, PB7 para el verde y PB8 para el azul.

En el siguiente módulo, configuramos los parámetros del TIM4 para que trabaje a una frecuencia de 100 Hz y para que su valor de TIM_Period se inicialice con el valor inicial de la variable PERIOD = 1000.

A continuación, creamos la estructura con la configuración de una señal PWM, donde establecemos inicialmente TIM_Pulse = 0 para que los tres colores de nuestro LED se inicien apagados.

Por último, programamos unas líneas en donde los valores de las variables TIM_Pulse_R/G/B se incrementen automáticamente y se escriban directamente en los registros de los canales 1, 2 y 3 del timer 4.

Cuando los valores de las variables lleguen a 1000, se resetean a valor 0 y comienza de nuevo el proceso.

12.4 EJEMPLO PWM CONTROLANDO UN SERVO MOTOR

Los servo motores son un tipo específico de motores CC, cuya característica principal es que utilizan precisamente el ancho de pulso de una señal PWM para controlar la posición de su cabezal.



Figura 12.15

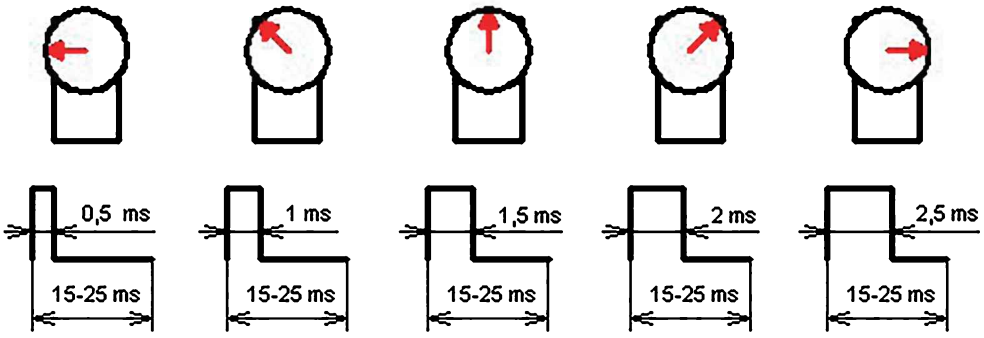


Figura 12.16

En nuestro siguiente ejemplo, controlaremos el sentido de giro del cabezal de un servo motor mediante dos pulsadores que modificarán el valor del TIM_Pulse y con ello se originará que el cabezal se mueva en un sentido u otro.

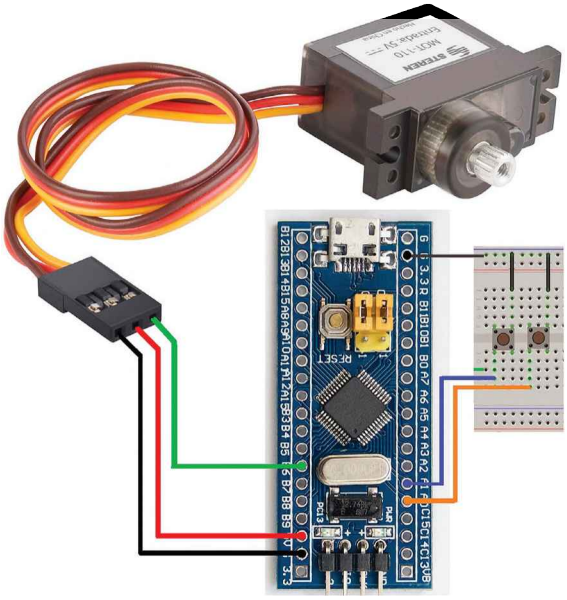


Figura 12.17

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"

/*            Modulo principal                */
//============================================
int main(void)
{
      int TIM_Pulse = 1000;   // Variable para el TIM_Pulse


       /*  Creamos las estructuras para GPIO   */
       GPIO_InitTypeDef GPIO_InitStructure;

      /*  Configuramos GPIO del pin de Salida PB6  */
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
      GPIO_StructInit(&GPIO_InitStructure);
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
      GPIO_Init(GPIOB, &GPIO_InitStructure);

      /*  Configuramos GPIO de los pines PA0 y PA1 de los botones  */
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
      GPIO_StructInit(&GPIO_InitStructure);
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
      GPIO_Init(GPIOA, &GPIO_InitStructure);
```

```
31
32      /*   Creamos la estructura para TIMER4   */
33      TIM_TimeBaseInitTypeDef TIMER_InitStructure;
34      RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
35      TIM_TimeBaseStructInit(&TIMER_InitStructure);
36      TIMER_InitStructure.TIM_Prescaler = 72;
37      TIMER_InitStructure.TIM_Period = 20000;
38      TIMER_InitStructure.TIM_ClockDivision = 0;
39      TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
40      TIM_TimeBaseInit(TIM4, &TIMER_InitStructure);
41
42      /*  Configuracion del PWM TIM4_CH1  */
43      TIM_OCInitTypeDef TIM_PWM_InitStructure;
44      TIM_OCStructInit(&TIM_PWM_InitStructure);
45      TIM_PWM_InitStructure.TIM_Pulse = TIM_pulse;
46      TIM_PWM_InitStructure.TIM_OCMode = TIM_OCMode_PWM1;
47      TIM_PWM_InitStructure.TIM_OutputState = TIM_OutputState_Enable;
48      TIM_PWM_InitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
49      TIM_OC1Init(TIM4, &TIM_PWM_InitStructure);
50
51      TIM_Cmd(TIM4, ENABLE);
52
53      TIM_Pulse = TIM_PWM_InitStructure.TIM_Pulse;
54
55      while(1)
56      {
57          /*  Comprobamos la pulsacion de los botones    */
58
59          // Boton PA0
60          if (GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0) == 0)
61          {
62          // Se comprueba si se sobrepasa un tope de 3000
63          if (TIM_Pulse < 3000)
64            TIM_Pulse++;     // Incrementamos la variable de TIM_Pulse
65          }
66
67          // Boton PA1
68          if (GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_1) == 0)
69          {
70          // Se comprueba si se sobrepasa un tope de 100
71          if (TIM_Pulse > 100)
72            TIM_Pulse--;    // Disminuimos la variable TIM_Pulse
73          }
74
75          // Igualamos el valor de la variable 'TIM_Pulse'
76          TIM4->CCR1 = TIM_Pulse;
77
78          // Se reproduce un delay
79          for(int i=0;i<0x1000;i++);
80      }
81  }
```

Figura 12.18

En este ejemplo, vemos que es muy similar al proyecto anterior con el que controlábamos el valor de TIM_Pulse mediante unas variables que se incrementan o disminuyen de acuerdo al control de las pulsaciones de dos botones conectados a los pines PA0 y PA1.

El control del TIM_Pulse produce el movimiento del cabezal del servomotor.

12.5 EJEMPLO PWM GENERANDO SONIDOS EN UN ALTAVOZ

Otras de las funcionalidades que nos ofrece la generación de señales PWM es la de poder generar sonidos en dispositivos como altavoces o zumbadores.



Figura 12.19

Existen dos tipos de dispositivos que pueden generar sonido con la señal que obtienen: un tipo es el **zumbador piezoeléctrico,** que utiliza la propiedad de los cristales como el poliéster o la cerámica, que se deforman cuando se les aplica una tensión eléctrica, y el otro tipo, son los **electromagnéticos** que emplean un imán y una bobina para transmitir la señal eléctrica a una membrana.

Y generan diferentes sonidos cuando se les aplica una señal de tensión a una frecuencia determinada.

En nuestro siguiente ejemplo, reproduciremos una melodía con un zumbador conectado al pin PB6 de nuestro microcontrolador utilizando la generación de señales PWM a diferentes frecuencias y tiempos de duración, para generar notas musicales.

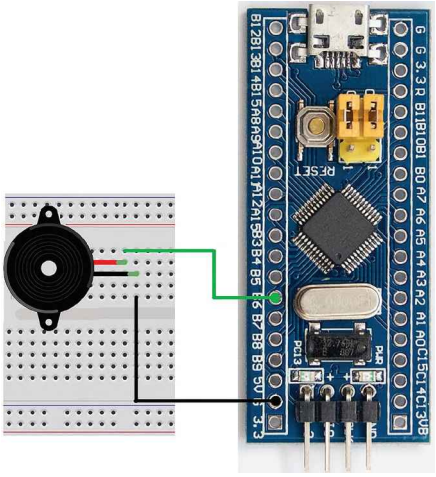


Figura 12.20

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_tim.h"

// Definimos la frecuencia del reloj y el prescaler
#define SYSCLK 72000000
#define PRESCALER 72

// Creamos las variables con las notas musicales y sus frecuencias
#define Do   261 //Do
#define Do1  277 //Do#
#define Re   293 //Re
#define Re1  311 //Re#
#define Mi   239 //Mi
#define Fa   349 //Fa
#define Fa1  370 //Fa#
#define Sol  392 //Sol
#define Sol1 415 //Sol#
#define La   440 //La
#define La1  466 //La#
#define Si   494 //Si

// Creamos las variables con los tiempos musicales
#define t1  2000
#define t3  1250
#define t2  1000
#define t4  500
#define t8  250
#define t16 125

// Definimos una estructura para entrar la musica
typedef struct
{
      uint16_t freq;
      uint16_t time;
}SoundTypeDef;


// Creamos la variable macro Music con la partitura
/*************************************/
/*       HARRY POTTER                */
/*************************************/
#define MUSICSIZE 32
const SoundTypeDef Music[MUSICSIZE] ={
  {Si, t4},
  {Mi, t1},
  {Sol, t8},
  {Fa1, t8},
  {Mi, t2},
  {Si, t4},
  {La, t3},
  {Fa1, t2},
  {Si, t4},
  {Mi, t2},
  {Sol, t8},
  {Fa1, t8},
  {Re, t2},
  {Mi, t4},
  {Si, t2},
  {0, t2},
  {Si, t4},
  {Mi, t2},
  {Sol, t8},
  {Fa1, t8},
  {Mi, t2},
  {Si, t4},
  {Re, t2},
  {Do1, t4},
  {Do*2, t2},
  {La, t4},
```

```
  {Do*2, t2},
  {Si, t8},
  {La1, t8},
  {Si, t2},
  {Sol, t4},
  {Mi, t2},
};

// Definimos las variables a emplear
int Paso_sonido = 0;
char Play_Musica = 0;
int TIMER_Sounido;
int sound_counter;

/*    Funcion que configura los parámetros GPIO, TIMER_4 y NVIC  */
//-------------------------------------------------------------------
void SONIDO_Config(void)
{
      /*  Creamos la estructura para GPIO  */
      GPIO_InitTypeDef GPIO_InitStructure;

      /* Establecemos el reloj pra GPIO  */
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

      /*  Configuamos la salida del altavoz en pin PB6 */
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
      GPIO_Init(GPIOB, &GPIO_InitStructure);
      GPIO_StructInit(&GPIO_InitStructure);

      /*   Configuramos el TIM  */
      // Creamos la estructura para TIM
      TIM_TimeBaseInitTypeDef TIMER_InitStructure;
      RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
      TIM_TimeBaseStructInit(&TIMER_InitStructure);
      TIMER_InitStructure.TIM_Prescaler = 72;
      TIMER_InitStructure.TIM_Period = 0xFFFF;
      TIMER_InitStructure.TIM_ClockDivision = 0;
      TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
      TIM_TimeBaseInit(TIM4, &TIMER_InitStructure);

      /*  Configuramos el PWM  */
      // Creamos la estructura para PWM
      TIM_OCInitTypeDef TIM_PWM_InitStructure;
      TIM_OCStructInit(&TIM_PWM_InitStructure);
      TIM_PWM_InitStructure.TIM_Pulse = 0;
      TIM_PWM_InitStructure.TIM_OCMode = TIM_OCMode_PWM1;
      TIM_PWM_InitStructure.TIM_OutputState = TIM_OutputState_Enable;
      TIM_PWM_InitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
      TIM_OC1Init(TIM4, &TIM_PWM_InitStructure);

      /* Iniciamos la interrupcion para TIM4   */
      TIM_ITConfig(TIM4, TIM_IT_CC4, ENABLE);

      /* Iniciamos la interrupcion para TIM2   */
      NVIC_EnableIRQ(TIM4_IRQn);
}

/*   Funcion que genera la frecuencia en el Altavoz     */
//-------------------------------------------------------
void sound (int freq, int time_ms) {
      if (freq > 0) {
            TIM4->ARR = SYSCLK / PRESCALER / freq; //Cargamos el perido en TIM4
            TIM4->CCR1 = TIM4->ARR / 2;
      }
      else {
            TIM4->ARR = 1000;
            TIM4->CCR1 = 0;
      }
      TIM_SetCounter(TIM4, 0);

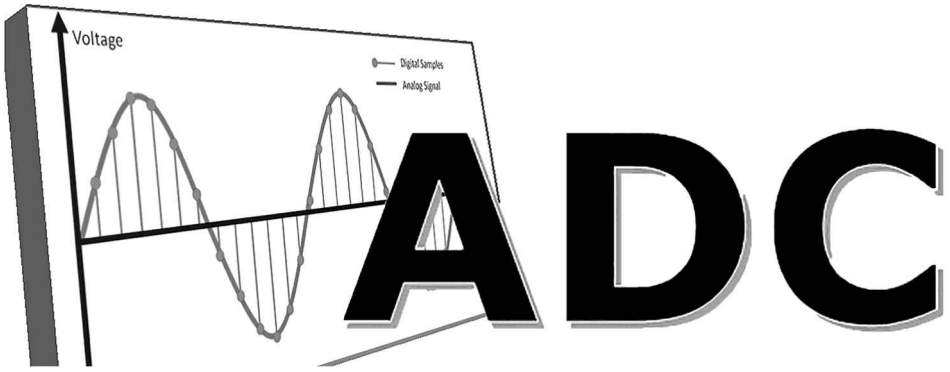
```

```
144       TIMER_Sounido = ((SYSCLK / PRESCALER / TIM4->ARR) * time_ms ) / 1000;
145       sound_counter = 0;
146       TIM_Cmd(TIM4, ENABLE);
147 }
148
149 /*    Funcion que inicia la musica en el altavoz       */
150 //-------------------------------------------------------------
151 void Sonido_Init(void) {
152       Paso_sonido = 0;
153       Play_Musica = 1;
154       sound(Music[Paso_sonido].freq, Music[Paso_sonido].time);
155 }
156
157 /*  Funcion que detecta la interrupcion y produce el sonido    */
158 //-------------------------------------------------------------------
159 void TIM4_IRQHandler(void){
160
161       if (TIM_GetITStatus(TIM4, TIM_IT_CC4) != RESET)
162         {
163
164           TIM_ClearITPendingBit(TIM4, TIM_IT_CC4);
165           sound_counter++;
166
167           if (sound_counter > TIMER_Sounido) {
168             if (Play_Musica == 0) {
169                TIM_Cmd(TIM4, DISABLE);
170             }
171             else {
172                // Mientras el índice 'Paso_sonido' sea menor del
173                // total 'MUSICSIZE' de notas de la partitura
174                if (Paso_sonido < MUSICSIZE-1) {
175                    if (TIM4->CCR1 == 0){
176                        // Pasamos a la siguiente nota a reproducir
177                        Paso_sonido++;
178                        // Reproducimos la frecuencia de la nota de la partitura
179                        sound(Music[Paso_sonido].freq, Music[Paso_sonido].time);
180                        }
181                        else{
182                            sound(0, 30);  //Silencio durante 30 de tiempo
183                        }
184                   }
185                   else {
186                      Play_Musica = 0;
187                      TIM_Cmd(TIM4, DISABLE);
188                   }
189             }
190           }
191
192           if (TIM_GetFlagStatus(TIM4, TIM_FLAG_CC4OF) != RESET)
193           {
194             TIM_ClearFlag(TIM4, TIM_FLAG_CC4OF);
195           }
196         }
197 }
198
199 /*                Modulo principal             */
200 //=============================================
201 int main(void)
202 {
203     SONIDO_Config();  // Inicializamos la configuracion del GPIO y TIMER
204
205     Sonido_Init();    // Iniciamos los parametros que inician la meolodia
206
207     while(1)
208     {
209
210     }
211 ;
```

Figura 12.21

13

PROGRAMACIÓN ADC



ADC (*Analog to Digital Converter*) es el módulo del microcontrolador encargado de convertir señales analógicas del exterior, en valores digitales que pueden ser interpretados por otros periféricos. Los microcontroladores poseen varios pines que pueden realizar esta función de conversión ADC.

Vemos en la representación de la Figura 13.1, que cada uno de los puntos significativos de una señal analógica (raya verde) lo podemos representar por un valor digital que representará dicho valor en un punto de esa cuadrícula.

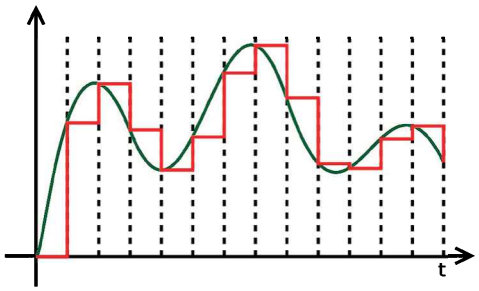


Figura 13.1

En el STM32F103, contamos con 2 módulos de 10 canales externos cada uno, que se nombran desde **ADC1/2_IN0** al **ADC1/2_IN9**; además, dos canales internos, un sensor interno de temperatura y una entrada de alimentación separada que se utiliza para obtener la tensión de referencia necesaria para el cálculo de la señal analógica. Pueden configurarse con una resolución de 8 bits o 12 bits. En el modo de 12 bits, los datos obtenidos pueden justificarse a la izquierda o alinearse a la derecha. Los módulos pueden tardar unos 14 ciclos de reloj en realizar el muestreo de la señal analógica obtenida; por lo que tarda aproximadamente 1 µseg en obtener la representación digital de un valor analógico capturado.

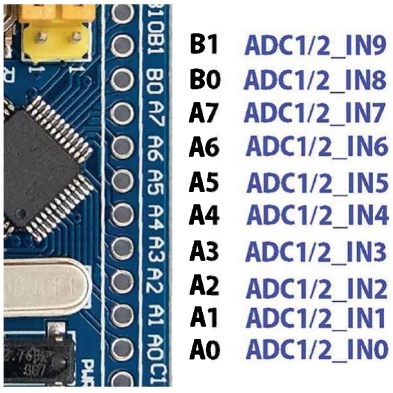


Figura 13.2

Todos estos canales pueden configurarse para su lectura en **modo individual**, en **modo aleatorio** o en **modo doble,** comparando la lectura de dos canales a la vez. También podemos programar su lectura para ir guardando los valores entregados o establecer un "*Watchdog*" que nos avise, cuando se supere un valor determinado.

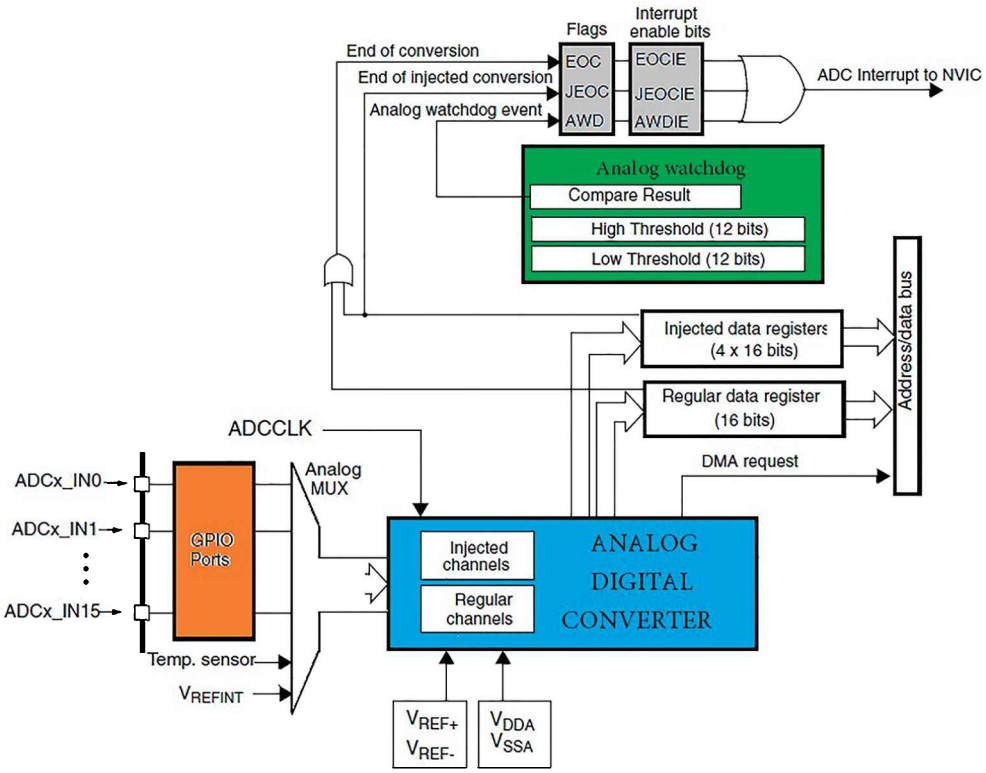


Figura 13.3

Estos canales se organizan en dos grupos: de los 20, 16 son canales regulares (normales) de 16 bits y 4 son canales "*injected*" de 16 bits que pueden tratar eventos del exterior y que poseen prioridad ante los canales normales. Cuando se utiliza la conversión con un canal inyectado, todas las conversiones de canales regulares se detienen temporalmente, comportándose por tanto como si se trataran de interrupciones.

MODO INDEPENDIENTE DE ADC

El ADC es configurado utilizándolo de forma independiente de otro canal, en este modo, podemos establecer otros subtipos de configuraciones posibles:

1. **Mono canal** (*Single Channel*)

   El ADC realiza una conversión de un canal, graba el valor resultante en el registro de salida y se detiene.

2. **Individual continúo** (*Single-channel continuous conversion mode*)

   Este modo es similar al primero, pero el ADC no se detiene y sigue trabajando con el canal seleccionado. El resultado se sobrescribe de forma continua con el valor más actual.

3. **Scan** (*Multichannel (scan) continuous conversion mode*)

   Este modo puede configurarse para realizar transformaciones sucesivas en múltiples canales en una secuencia dada. El tiempo de conversión se ajusta por separado para cada canal. Después de procesar el número especificado de canales, el ADC se detiene.

4. **Escaneo continuo** (*Multichannel continuous*)

   Igual que en el modo "*Scan*", el ADC realizará conversiones en varios canales programados y una vez procesados todos, el ciclo vuelve a comenzar, sobrescribiéndose el valor final resultante cuando se reinicia el proceso.

5. **Discontinuo** (*Intermitente*)

   En este modo, no se escanearán todos los canales a la vez, sino solo unos pocos predefinidos. Cuando se produce el evento que desencadena la transformación, será escaneado y se pasará al siguiente grupo de canales y así sucesivamente.

MODO DOBLE DE ADC

Se utiliza para realizar las lecturas en dos o más canales ADC, con el fin de mejorar la velocidad de captura de datos. Puede tener estos subtipos:

1. **Simultáneas** (*Simultáneamente a los canales regulares o inyectados*).

   Primero el ADC explorará los canales que van desde 0 a 15 para canales regulares, o de 0 a 3 para canales inyectados. En segundo lugar, hacia atrás desde 15 a 0 o de 0 a 3 para canales inyectados. Por lo tanto, cada canal en el mismo periodo de tiempo, será procesado dos veces.

2. **Intercalada rápida** (*Quick intercambiables*)

   Está disponible solo para un canal regular. Cuando se ejecuta, comienza primero la conversión ATSP2 y luego se ejecuta en ADC –que produce un retraso de siete ciclos-. Si se establece en escaneos continuos, esta secuencia se repetirá y se permitirá el doble de la entrada de medición de frecuencia.

3. **Intercalada lenta** (*Slow intercambiables*)

   Igual que en el modo anterior, solamente el ADC se inicia con un retraso de 14 ciclos y después de la primera conversión, el ADC soportará un retardo de 14 ciclos antes de volver a empezar.

4. **Disparo alternativo** (*Alternativa para desencadenar*)

   Disponibles solo para canales inyectados. El ADC comienza secuencialmente a convertir todos los canales. Cuando se produce un evento que desencadena la próxima conversión. Si se establece en discontinuo, se puede convertir solo un canal para cada evento.

5. **Modo combinado Regular / inyectado simultáneo** (*Modo simultáneo combinado para los canales regulares e inyectados*)

   No es una transformación de los dos grupos de canales. Pero los canales del grupo inyectado pueden interrumpir canales regulares en su conversión. Es decir, si un evento desencadena la conversión del canal inyectado, entonces el ADC, que se ocupa de los canales regulares, cambiará a canales de procesamiento inyectados.

6. **Modo de activación simultánea + alternativo normal combinado.**

   Lanzamiento por grupo de canales regulares que se puede interrumpir por señal inyectada en canales alternos. Una vez completada la conversión de todos los grupos, debe detenerse la conversión de los dos grupos, y los resultados del procesamiento de datos, serán almacenados en los registros de cada ADC.

7. **En combinación modo de activación simultánea + alternancia regular.**

   En la conversión de los canales regulares, producidos después de generado el evento, y que no se hubiera producido la conversión simultánea de dos grupos de canales.

El módulo ADC utiliza una tensión de referencia para realizar la conversión. En nuestra placa tenemos señales **VDDA** y **VSSA** que provienen de la fuente de alimentación y que se usan también para alimentar el periférico ADC, mientras que **Vref+**, es una señal de tensión de referencia para el módulo ADC.

13.1 EJEMPLO ADC EN MODO CONTINUO

En nuestro primer ejemplo utilizaremos el modo continuo (*Scan*), conectando un potenciómetro a uno de los canales del ADC, que nos irá entregando la lectura de un voltaje, cuyo valor analógico, será convertido a un valor digital, que enviaremos por el puerto USART1 hacia nuestro ordenador

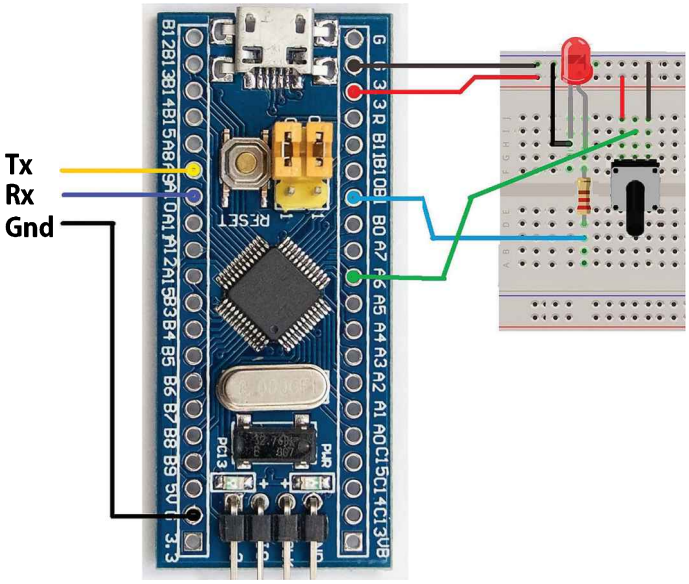


Figura 13.4

Código de ejemplo:

```
// Libreria principal del microcontrolador
#include "stm32f10x.h"

// Las librerias para los perifericos
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_adc.h"

// Libreria de contro del puerto USART1
#include "usart1.h"
#include <stdio.h>
#include <string.h>

/* Lista de funciones empleadas    */
int getPot(void);
void GPIO_config(void);
void adc_config(void);
void Delay_ms(unsigned int nCount);
```

```
/*  Modulo principal   */
//=====================================================
int main(void){

    // Creamos la variable para el contador
    float x;
    // Llamamo a las funciones que inicializan los perifericos
    USART1_Init(); // Inicializamos el USART1
    GPIO_config(); //Inicializamos el GPIO del LED y potenciometro
    adc_config();  // Inicializamos el ADC

    while(1)    // Bucle infinito
    {
       // La formula empleada es:
       //     [Vcc_out = Val_ADC / MaxADC_bits(4096) * Vcc_in ]
       // Calculamos el valor leido del PIN PA6 de la siguiente forma:
       // lo multiplicamos por 3.3 Vcc/dividio por 4095 -que es la resolucion
       //                                                de 12 bits-
       // el resultado 'x' es el valor en voltios
       x = getPot()*3.3/4095;

       // Comparamos el valor obtenido con un limite
       if(x > 1.6)  // Si x es mayor de 1.6
                   //if(x > 1.3)   // Si x es mayor de 1.3
        {
          GPIO_WriteBit(GPIOB, GPIO_Pin_1, Bit_SET);//Enciende LED
        }
        else
        {
          GPIO_WriteBit(GPIOB, GPIO_Pin_1, Bit_RESET);//Apaga LED
        }

        // Enivamos el valor obtenido por el puerto USART1
        printf("Valor = %4.2f Vcc\n\r", x);
        Delay_ms(100);
    }
}

/* Funcion que lee el valor analogico obtenido en el pin   */
//-------------------------------------------------
int getPot(void)
{
    return ADC_GetConversionValue(ADC1);
}


/* Funcion que configura e inicializa el ADC    */
//-------------------------------------------------
void adc_config(void)
{
      /*  Nombramos la estructura de configuracion del ADC   */
      ADC_InitTypeDef ADC_StructureInit;

      /*  Inicializamos el Reloj de control del ADC   */
      RCC_ADCCLKConfig(RCC_PCLK2_Div6); //Clock para el ADC (max 14MHz,
                                        72/6=12MHz)
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); // Inicializa el
                                                             clock ADC

      /* Configuramos los parametros del ADC        */
      ADC_StructureInit.ADC_Mode = ADC_Mode_Independent;
      ADC_StructureInit.ADC_ScanConvMode = DISABLE;
      ADC_StructureInit.ADC_ContinuousConvMode = ENABLE;
      ADC_StructureInit.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
      ADC_StructureInit.ADC_DataAlign = ADC_DataAlign_Right;
      ADC_StructureInit.ADC_NbrOfChannel  = 1;
      ADC_RegularChannelConfig(ADC1, ADC_Channel_6, 1,
ADC_SampleTime_55Cycles5); //PA6 as Input
      ADC_Init(ADC1, &ADC_StructureInit);
```

```
88
89          /*  Iniciamos el ADC          */
90          ADC_Cmd(ADC1, ENABLE);
91
92          /* Opciones de libreria para calibracion del modulo ADC   */
93          ADC_ResetCalibration(ADC1);
94          while(ADC_GetResetCalibrationStatus(ADC1));
95          ADC_StartCalibration(ADC1);
96          while(ADC_GetCalibrationStatus(ADC1));
97
98          // Iniciamos el ADC1
99          ADC_Cmd(ADC1, ENABLE);
100 }
101
102 /*  Funcion que configura el GPIO para LED en PB1    */
103 //-----------------------------------------------------
104 void GPIO_config(void)
105 {
106         /*  Creamos la estructura para GPIO   */
107         GPIO_InitTypeDef GPIO_StructureInit;
108
109         /* Activamos el reloj para GPIOB       */
110         RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
111         RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
112
113         /* Configuramos el pin PC13 para el LED  */
114         GPIO_StructureInit.GPIO_Pin = GPIO_Pin_1; // Asignamos el Pin PC13 para
115 configuracion
116         GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;// Establecemos la
117 velocidad a 50MHz (Speed)
118         GPIO_StructureInit.GPIO_Mode = GPIO_Mode_Out_PP; // Establecemos el pin
119 como Salida(output) y push-pull resistor
120         GPIO_Init(GPIOB, &GPIO_StructureInit);     // Iniciamos la estructura en
121 el compilador
122
123         /* Configuramos el pin PA1 para el pulsador  */
124         GPIO_StructureInit.GPIO_Pin = GPIO_Pin_6;       // Asignamos el Pin PCA1
125 para configuracion
126         GPIO_StructureInit.GPIO_Speed = GPIO_Speed_50MHz;// Establecemos la
127 velocidad a 50MHz (Speed)
128         GPIO_StructureInit.GPIO_Mode = GPIO_Mode_AIN;    // Establecemos el pin
129 como Entrada (Imput) y pull-up resitor
130         GPIO_Init(GPIOA, &GPIO_StructureInit);
131
132 }
133
134 /* Modulo que genera un retardo (ms)     */
135 //--------------------------------------
136 void Delay_ms(unsigned int nCount){
137             unsigned int i, j;
138             for(i = 0; i < nCount; i++)
139             {
140                         for(j = 0; j < 0x2AFF; j++){;}
141             }
142 }
```

Figura 13.5

En nuestro ejemplo utilizamos el canal 1 del ADC, en el pin PA1; creando la estructura GPIO y activando su reloj correspondiente.

En la Figura 13.6 enumeramos los canales de los dos ADC de que dispone nuestro microcontrolador STM32 y mostramos la tabla de correspondencia de pines GPIO:

CANAL	ADC 1 / ADC 2
Canal 0	PA0
Canal 1	PA1
Canal 2	PA2
Canal 3	PA3
Canal 4	PA4
Canal 5	PA5
Canal 6	PA6
Canal 7	PA7
Canal 8	PB0
Canal 9	PB1

Figura 13.6

Creamos una función **ADC_Config()** que engloba toda la configuración necesaria con los parámetros que inicializan el ADC, tal como aparecen en la Figura 13.7.

```
/*  Creamos la estructura del ADC   */
ADC_InitTypeDef ADC_StructureInit;

/*  Inicializamos el Reloj de control del ADC   */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
RCC_ADCCLKConfig(RCC_PCLK2_Div6); // ADC (max 14MHz, 72/6=12MHz)

/* Configuramos los parametros del ADC        */
ADC_StructureInit.ADC_Mode = ADC_Mode_Independent;
ADC_StructureInit.ADC_ScanConvMode = DISABLE;
ADC_StructureInit.ADC_ContinuousConvMode = ENABLE;
ADC_StructureInit.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
ADC_StructureInit.ADC_DataAlign = ADC_DataAlign_Right;
ADC_StructureInit.ADC_NbrOfChannel  = 1;
ADC_RegularChannelConfig(ADC1, ADC_Channel_6, 1, ADC_SampleTime_55Cycles5);
ADC_Init(ADC1, &ADC_StructureInit);

/*  Iniciamos el ADC          */
ADC_Cmd(ADC1, ENABLE);

/* Opciones de libreria para calibracion del modulo ADC   */
ADC_ResetCalibration(ADC1);
while(ADC_GetResetCalibrationStatus(ADC1));
ADC_StartCalibration(ADC1);
while(ADC_GetCalibrationStatus(ADC1));
```

Figura 13.7

La estructura, la llamaremos "ADC_StructureInit", donde el primer parámetro será habilitar los relojes correspondientes: el APB2 para el pin del Canal 1 del ADC1 y el reloj **ADCCLK** que es específico del ADC, indicándose también su divisor de la frecuencia. Se le puede dar cualquiera de estos valores:

- **RCC_PCLK2_Div2**: ADC clock = PCLK2/2
- **RCC_PCLK2_Div4**: ADC clock = PCLK2/4
- **RCC_PCLK2_Div6**: ADC clock = PCLK2/6
- **RCC_PCLK2_Div8**: ADC clock = PCLK2/8

Figura 13.8

A continuación, introducimos los siguientes parámetros necesarios para nuestro ADC:

ADC_Mode con el que configuramos el modo de funcionamiento del canal o canales que se programan en nuestro código:

- **ADC_Mode_ComRegSimulAltTrig** - Modo múltiple canales regulares, modo disparo.
- **ADC_Mode_CombRegSimulInjSimul** – Modo simultaneo canales inyectados.
- **ADC_Mode_Independent** – Modo canal simple independiente.
- **ADC_Mode_InjSimul** – Modo múltiple simultaneo.
- **ADC_Mode_Interleave** – Modo múltiple intercalado.
- **ADC_Mode_AltTrig** – Modo múltiple de disparo alternativo.

Figura 13.9

ADC_ScanConvMode, donde especificamos si la conversión será en modo multicanal (DISABLE) o de un solo canal (ENABLE).

ADC_ContinuousConvMode con el que indicamos si la conversión a realizar será de modo continuo (ENABLE) o individual de una sola vez (DISABLE).

ADC_ExternalTrigConv, define el modo de disparo externo que se utilizará para comenzar la conversión de canales regulares. Los modos posibles podrán ser:

- **ADC_ExternalTrigConv_T1_CC1** - Para ADC1 y ADC2
- **ADC_ExternalTrigConv_T1_CC2** - Para ADC1 y ADC2
- **ADC_ExternalTrigConv_T1_CC3** - Para ADC1, ADC2 y ADC3
- **ADC_ExternalTrigConv_T2_CC2** - Para ADC1 y ADC2
- **ADC_ExternalTrigConv_T2_CC3** - Solo para ADC3
- **ADC_ExternalTrigConv_T3_CC1** - Solo para ADC3
- **ADC_ExternalTrigConv_T3_TRGO** – Para ADC1 y ADC2
- **ADC_ExternalTrigConv_T4_CC4** - Para ADC1 y ADC2
- **ADC_ExternalTrigConv_Ext_IT11_TIM8_TRGO** - Para ADC1 y ADC2
- **ADC_ExternalTrigConv_None** - Para ADC1, ADC2 y ADC3
- **ADC_ExternalTrigConv_T8_CC1** - Solo para ADC3
- **ADC_ExternalTrigConv_T8_TRGO** – Solo para ADC3
- **ADC_ExternalTrigConv_T5_CC1** - Solo para ADC3
- **ADC_ExternalTrigConv_T5_CC3** - Solo para ADC3

Figura 13.10

ADC_DataAlign, permite especificar si la alineación de datos ADC será a la izquierda (***ADC_DataAlign_Right***) o a la derecha *(**ADC_DataAlign_Left**)*.

ADC_NbrOfChannel, indica el número de canales ADC a utilizar, que serán del grupo de canales normales y cuyo valor a especificar va de 1 a 16.

ADC_RegularChannelConfig(*ADC_TypeDef *ADCx, uint8_t ADC_ Channel, uint8_t Rank, uint8_t ADC_SampleTime)*, donde definimos qué ADC**x** vamos a usar (x = 1 o 2), qué canal vamos a utilizar (CH0~CH9), el rango en el secuenciador de canales regulares (1 a 16), y el valor del tiempo de muestra que se establecerá para el canal indicado según las siguientes opciones:

- **ADC_SampleTime_3Cycles**, tiempo de muestreo igual a 3 ciclos.
- **ADC_SampleTime_15Cycles**, tiempo de muestreo igual a 15 ciclos.
- **ADC_SampleTime_28Cycles**, tiempo de muestreo igual a 28 ciclos.
- **ADC_SampleTime_56Cycles**, tiempo de muestreo igual a 56 ciclos.
- **ADC_SampleTime_84Cycles**, tiempo de muestreo igual a 84 ciclos.
- **ADC_SampleTime_112Cycles**, tiempo de muestreo igual a 112 ciclos.
- **ADC_SampleTime_144Cycles**, tiempo de muestreo igual a 144 ciclos.
- **ADC_SampleTime_480Cycles**, tiempo de muestreo igual a 480 ciclos.

Figura 13.11

Una vez configurados todos los parámetros que queremos para nuestro ADC, lo siguiente será activarlo mediante la instrucción **ADC_Cmd(***ADCx***,** *ENABLE***);**

Después necesitamos realizar una calibración interna que restablecerá los valores iniciales; para lo que se utilizarán las líneas con el comando **ADC_ ResetCalibration**(*ADCx*); posteriormente, se espera a que se produzca la recalibración del ADC con la instrucción **ADC_StartCalibration**(*ADCx*)**.**

Luego, en el módulo principal "int main ()", bastará con introducir una línea de código que llame a la función **ADC_config** que hemos creado, para que se inicialice el ADC con la configuración que hemos establecido.

Para obtener el valor de la conversión que se produce en el canal 1 del ADC1 mediante el comando **ADC_GetConversionValue**(*ADCx*), que hemos encerrado en nuestro código de ejemplo en la función **getPot**(), y que esta nos devuelve el último valor que fue obtenido en la conversión, que pasamos a nuestra variable 'x', realizándole el promedio respecto al voltaje de 3.3 Vcc que utilizamos como referencia y dividiéndolo por 4096 que es la resolución de 16 bits que posee el registro.

En las siguientes líneas, programamos que si dicho valor 'x' es mayor de un tope, que nosotros establecemos en 1,6 voltios, se encienda el LED conectado al pin PB1 y si no, se apague.

Todo ello, enviando los valores obtenidos por el puerto serial hacia nuestro ordenador.

13.2 EJEMPLO CON EL SENSOR DE TEMPERATURA INTERNO

Nuestro microcontrolador posee un sensor interno que se utiliza para medir la temperatura ambiente y de la propia CPU. Posee su propio canal ADC, está internamente conectado al canal 16 (**ADC**x_**IN16)**, que convierte la entrada de tensión que está entre 2 Vcc y 3,6 Vcc de VDDA, en un valor digital. El tiempo de muestreo recomendado es de 17,1 µs. Soporta un rango de temperatura de entre -40 a 125 grados Celsius. La precisión es de aproximadamente ± 1,5 ºC.

Para su configuración, bastará con utilizar los mismos comandos y funciones descritas en el ejemplo anterior de ADC, necesitando solo añadir el comando que lo inicializa, **ADC_TempSensorVrefintCmd**(ENABLE).

```
1   /*  Creamos la estructura para ADC    */
2   ADC_InitTypeDef ADC_InitStructure;
3
4   // Iniciamos el reloj APB2 para ADC1
5   RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
6
7   /*  Configuramos los parámetros del ADC1  */
8       - NbrOfChannel = 1 - ADC_Channel_16
9       - Mode = Single ConversionMode(ContinuousConvMode disabled)
10      - Resolution = 12Bits
11      - Prescaler = /1
12      - sampling time 192 */
13
14  ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
15  ADC_InitStructure.ADC_ScanConvMode = DISABLE;
16  ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
17  ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
18  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
19  ADC_InitStructure.ADC_NbrOfChannel = 1;
20  ADC_Init(ADC1, &ADC_InitStructure);
21
22  // Iniciamos el Sensor de Temperatura Interno
23  ADC_TempSensorVrefintCmd(ENABLE);
24
25  // Configuramos el canal 16 del ADC1
26  // Establecemos unos 41.5 cycles de conversión
27  // y el rango en el secuenciador =1
28  ADC_RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_41Cycles5);
29
30  /*  Iniciamos el ADC          */
31  ADC_Cmd(ADC1, ENABLE);
32
```

```
33 /* Opciones de libreria para calibracion del modulo ADC   */
34 ADC_ResetCalibration(ADC1);
35
36 While (ADC_GetResetCalibrationStatus(ADC1));
37    ADC_StartCalibration(ADC1);
38 While (ADC_GetCalibrationStatus(ADC1));
39
40 //Start ADC1 Software Conversion
41 ADC_SoftwareStartConvCmd(ADC1, ENABLE);
42
43 //Esperamos a que se complete la conversion
44 while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC))
45 {}
46
47 ///  Leemos el valor leido
48 AD_value = ADC_GetConversionValue(ADC1);
49
50 // Reseteamos la bandera de conversion
51 ADC_ClearFlag(ADC1, ADC_FLAG_EOC);
52
```

Figura 13.12

Los pasos son sencillos; primero debemos activar el canal interno del ADC, que en este caso es el canal **CH16**, después ajustamos los parámetros. El sensor nos devolverá un valor de voltaje medido, que debemos convertir a un valor de acuerdo al voltaje de referencia que estemos empleando. Este, lo dividimos por la resolución que tiene, en el modo de 12 bits será 4096 y en el de 8 bits, 255. Después, al valor obtenido, le aplicamos la siguiente fórmula que nos permite obtener la temperatura en grados Celsius.

$$T\,(℃) = \{\,(V25 - Vsense)\,/Avg_Slope\} + 25$$

Figura 13.13

Donde tenemos que:

V25 = Vsense -> Valor típico de grados (1.43)
Avg_Slope -> EL valor típico de (mv / ºC) o (uv/ºC)
Típicamente = (4.3 Mv / ºC)

Figura 13.14

Ejemplo del fichero “main.c”:

```
#include "stm32f10x.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"

// Llamada a la librería de control del puerto USARTx
#include "usart1.h"

// Llamada a la librería especial del Keil para manejo del comando 'printf'
#include <stdio.h>

// Lista de funciones empleadas
void IntTemp_Config(void);
void Delay_ms(unsigned int nCount);

// Lista de variables empleadas
const uint16_t V25 = 1.43;      // valor de V25=1.41V para 3.3Vcc de REF
const uint16_t Avg_Slope = 4.3; // valor de avg_slope=4.3mV/C para 3.3Vcc de REF
uint16_t AD_value, TemperatureC;

/*              Modulo principal                                */
//=============================================================
int main(void)
{
  USART1_Config(115200);   // Inicializamos el USART1

  IntTemp_Config();        // Inicializamos el Sensor Interno


  while (1)
  {
    // Esperamos a que se termine la conversion
    while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);

    //  Leemos el valor leido
    float AD_value=ADC_GetConversionValue(ADC1);

    /*  Calculamos el valor obtenido en el ADC según VRef
        Valor representado sera = ADC_val * (3.3 Vcc / 4096)  */
    AD_value = AD_value * (3.3/4096);
    printf(" Valor ADC  : %4.2f \r\n", (float) AD_value);

    // Reseteamos la bandera de fin de conversion
    ADC_ClearFlag(ADC1, ADC_FLAG_EOC);

    // Convertimos el valor obtenido en un valor en grados Celsius
    TemperatureC = (uint16_t) ((V25-AD_value)/ Avg_Slope+25);

    // Imprimimos por el puerto USART el valor obtenido
    printf(" Temperatura: %d C\r\n", TemperatureC);

    ADC_SoftwareStartConvCmd(ADC1, ENABLE);

    Delay_ms(1000);
  }
}

/*       Funcion que configura el ADC del sensor interno   */
//-------------------------------------------------------------------
void IntTemp_Config(void)
{
    /*  Creamos la estructura para ADC    */
    ADC_InitTypeDef ADC_InitStructure;

    // Iniciamos el reloj APB2 para ADC1
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
    RCC_ADCCLKConfig(RCC_PCLK2_Div6);

    /*  Configuramos los parámetros del ADC1  */
    /*
```

```
70      - NbrOfChannel = 1 - ADC_Channel_16 (ADC_Channel_TempSensor)
71      - Mode = Single ConversionMode (ContinuousConvMode disabled)
72      - Resolution = 12Bits
73      - Prescaler = /1
74      - sampling time 192 */
75
76      ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
77      ADC_InitStructure.ADC_ScanConvMode = DISABLE;
78      ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;  // Continuo sampler
79      ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
80      ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
81      ADC_InitStructure.ADC_NbrOfChannel = 1;
82      ADC_Init(ADC1, &ADC_InitStructure);
83
84      /* Configuramos el canal 16 del ADC1
85         Establecemos unos 41.5 cycles de conversión
86         y el rango en el secuenciador =1          */
87      ADC_RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_41Cycles5);
88
89      // También existe esta forma de entrar estos parámetros
90      //ADC_RegularChannelConfig(ADC1, ADC Channel TempSensor, 1,
91  ADC_SampleTime_239Cycles5);
92
93      // Iniciamos el Sensor de Temperatura Interno
94      ADC_TempSensorVrefintCmd(ENABLE);
95
96      // Iniciamos el ADC
97      ADC_Cmd(ADC1, ENABLE);
98
99      /* Opciones de libreria para calibracion del modulo ADC   */
100     ADC ResetCalibration(ADC1);
101
102     while(ADC_GetResetCalibrationStatus(ADC1));
103
104     ADC_StartCalibration(ADC1);
105
106     while(ADC_GetCalibrationStatus(ADC1));
107
108     ADC_SoftwareStartConvCmd(ADC1, ENABLE);
109 }
110
111 /*   Modulo que genera un retardo (ms)     */
112 //--------------------------------------
113 void Delay_ms(unsigned int nCount){
114       unsigned int i, j;
115       for(i = 0; i < nCount; i++)
116       {
117             for(j = 0; j < 0x2AFF; j++){;}
118       }
119 }
```

Figura 13.15

13.3 EJEMPLO ADC EN MODO MÚLTIPLES CANALES. (DMA)

En el siguiente ejemplo, empleamos varios canales para conseguir capturas continuas. Necesitaremos acceso al **DMA** –que explicaremos más adelante- para poder obtener los datos convertidos que han sido almacenados en la memoria, mientras se emplea un tiempo en recoger los siguientes datos recogidos en otros canales.

También utilizaremos el diseño de conexiones del ejemplo del potenciómetro, para así poder conectar la salida de este, a cada uno de los pines PA4, PA5, PA6 o PA7 y así comprobar el funcionamiento de conversión de las señales de modo independiente en cada uno de los canales que vamos a utilizar. De manera que, cada vez que queramos obtener un valor de conversión en uno de los canales, modificamos el nivel de nuestro potenciómetro para obtener un valor diferente y cambiamos de canal de entrada para simular así el funcionamiento en los 4 canales.

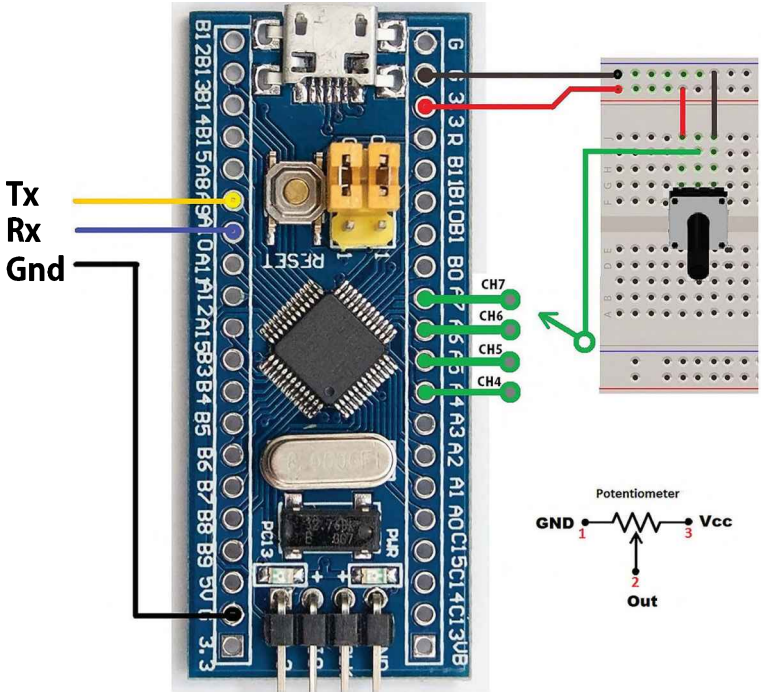


Figura 13.16

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_usart.h"
#include "stm32f10x_adc.h"
#include "stm32f10x_dma.h"
#include "stm32f10x_tim.h"

/* Llama a la libreria para el puerto USART1  ------------------------*/
#include "usart1.h"

#include <stdio.h>

/* Lista de funciones  -----------------------------------------------*/
void ADC_DMA_Config(void);
void TIM_Config(void);
```

```
17
18 /* Variables empleadas en nuestro programa  -------------------------*/
19 volatile short FLAG_ECHO = 0;
20 volatile uint16_t ADCBuffer_value[] = {0xAAAA, 0xAAAA, 0xAAAA, 0xAAAA};
21 volatile uint32_t status = 0;
22
23 /*   Funcion que detecta la interrupcion IRQ en TIMER4 -------------*/
24 //-----------------------------------------------------------------------
25 void TIM4_IRQHandler(void)
26 {
27       if (TIM_GetITStatus(TIM4, TIM_IT_Update) != RESET)
28       {
29           FLAG_ECHO = 1;
30           TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
31       }
32 }
33
34 /*             Modulo principal                                      */
35 //=======================================================================
36 int main(void)
37 {
38     USART1_Config(115200);  // Inicializamos USART1
39
40     ADC_DMA_Config();       // Inicializamos ADC y GPIO
41
42     TIM_Config();           // Configuramos TIMER y NVIC
43
44     printf("  PRUEBA DE ADC Multiple  \r\n");
45
46     while (1)
47     {
48         /* Habilita los canales 1 y 2 de DMA  ----------------------*/
49         DMA_Cmd(DMA1_Channel1, ENABLE);
50         DMA_Cmd(DMA1_Channel2, ENABLE);
51
52         /* Inicia la conversion de canales  ------------------------*/
53         ADC_SoftwareStartConvCmd(ADC1, ENABLE);
54
55         /* Imprime los valores por el puerto USART  ----------------*/
56         if (FLAG_ECHO == 1) {
57             printf (" Lecturas de ADC: ");
58
59             /*  Muestra el valor obtenido de cada canal -----------*/
60             for (int index = 0; index <5; index++){
61                 printf("[%d] %4.2fVc ", index,
62                    (float)ADCBuffer_value[index+16]*(3.3/4096));
63             }
64             printf ("\r\n");
65             /*  Resetea la bandera FLAG_ECHO ----------------------*/
66             FLAG_ECHO = 0;
67         }
68     ADC_SoftwareStartConvCmd(ADC1, DISABLE);
69     }
70 }
71
72 /*            Funcion que configura los canales ADC               */
73 //-----------------------------------------------------------------
74 void ADC_DMA_Config(void)
75 {
76   /*  Creamos las estrucutras para el GPIO, ADC y el DMA ---------*/
77   GPIO_InitTypeDef GPIO_InitStructure;
78   ADC_InitTypeDef ADC_InitStructure;
79   DMA_InitTypeDef DMA_InitStructure;
80
81   /*  Habilitamos los relejos para RCC, GPIO, ADC1, AFIO del ADC y DMA1
82   RCC_ADCCLKConfig(RCC_PCLK2_Div6);
83   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|
84                          RCC_APB2Periph_ADC1 |
85                          RCC_APB2Periph_AFIO , ENABLE);
86   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE );
```

```
87
88    /*  Crea la estructura del GPIO de los pines ---------------------*/
89    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | // ADC1_CH4 en PA4
90                                  GPIO_Pin_5 | // ADC1_CH5 en PA5
91                                  GPIO_Pin_6 | // ADC1_CH6 en PA6
92                                  GPIO_Pin_7;  // ADC1_CH7 en PA7
93    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
94    GPIO_Init(GPIOA, &GPIO_InitStructure);
95
96    /*   Configuracion de los parametros DMA ------------------------*/
97    DMA_InitStructure.DMA_BufferSize = 4;
98    DMA_InitStructure.DMA_DIR        = DMA_DIR_PeripheralSRC;
99    DMA_InitStructure.DMA_M2M        = DMA_M2M_Disable;
100
101   DMA_InitStructure.DMA_MemoryBaseAddr     = (uint32_t)ADCBuffer_value;
102
103   DMA_InitStructure.DMA_MemoryDataSize     = DMA_MemoryDataSize_HalfWord;
104   DMA_InitStructure.DMA_MemoryInc          = DMA_MemoryInc_Enable;
105   DMA_InitStructure.DMA_Mode               = DMA_Mode_Circular;
106   DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&ADC1->DR;
107   DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
108   DMA_InitStructure.DMA_PeripheralInc      = DMA_PeripheralInc_Disable;
109   DMA_InitStructure.DMA_Priority           = DMA_Priority_High;
110   DMA_Init(DMA1_Channel1, &DMA_InitStructure);
111   DMA_Cmd(DMA1_Channel1, ENABLE) ;
112
113   ADC_InitStructure.ADC_Mode               = ADC_Mode_Independent;
114   ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
115   ADC_InitStructure.ADC_ScanConvMode       = ENABLE;
116   ADC_InitStructure.ADC_ExternalTrigConv   = ADC_ExternalTrigConv_None;
117   ADC_InitStructure.ADC_DataAlign          = ADC_DataAlign_Right;
118   ADC_InitStructure.ADC_NbrOfChannel       = 4;
119   ADC_Init(ADC1, &ADC_InitStructure);
120
121   ADC_RegularChannelConfig(ADC1, ADC_Channel_4, 1, ADC_SampleTime_41Cycles5);
122   ADC_RegularChannelConfig(ADC1, ADC_Channel_5, 2, ADC_SampleTime_41Cycles5);
123   ADC_RegularChannelConfig(ADC1, ADC_Channel_6, 3, ADC_SampleTime_41Cycles5);
124   ADC_RegularChannelConfig(ADC1, ADC_Channel_7, 4, ADC_SampleTime_41Cycles5);
125
126   ADC_Cmd(ADC1, ENABLE) ;
127   ADC_DMACmd(ADC1, ENABLE) ;
128   ADC_ResetCalibration(ADC1);
129
130   while(ADC_GetResetCalibrationStatus(ADC1));
131   ADC_StartCalibration(ADC1);
132
133   while(ADC_GetCalibrationStatus(ADC1));
134   ADC_SoftwareStartConvCmd(ADC1 , ENABLE ) ;
135 }
136
137 /*   Funcion que configura el TIMER 4                              */
138 //---------------------------------------------------------------
139 void TIM_Config(void)
140 {
141     /*  Establece la estructrua TIMER y NVIC -------------*/
142     TIM_TimeBaseInitTypeDef TIMER_InitStructure;
143     NVIC_InitTypeDef NVIC_InitStructure;
144
145     /*  Habilita el reloj para el TIMER 4  ---------------*/
146     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
147
148     /*   Configura los parametros del TIMER --------------*/
149     TIM_TimeBaseStructInit(&TIMER_InitStructure);
150     TIMER_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
151     TIMER_InitStructure.TIM_Prescaler   = 7200;
152     TIMER_InitStructure.TIM_Period      = 5000;
153     TIM_TimeBaseInit(TIM4, &TIMER_InitStructure);
154     TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
155     TIM_Cmd(TIM4, ENABLE);
156
```

```
157     /* Configura el NVIC  para el IRQ del TIMER 4 --------*/
158     NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
159     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
160     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
161     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
162     NVIC_Init(&NVIC_InitStructure);
163 }
164
165
```

Figura 13.17

13.4 EJEMPLO ADC EN MODO MÚLTIPLES CANALES INYECTADO

El modo de conversión con canales inyectados, está diseñado para ser utilizado cuando la conversión se desencadena por un evento externo o por software. La diferencia principal entre los canales ADC normales y los inyectados radica, en que para obtener una secuencia de conversión, los canales normales realizan ese proceso de forma automática; mientras que los canales inyectados necesitan de un evento externo o mandato configurado por software para que se realice la conversión. El grupo de canales inyectados tiene prioridad sobre el grupo de canales regulares.

Este sistema, nos permite retrasar la conversión hasta después que se produzca un evento que necesitemos.

Como vimos al principio de este capítulo sobre la programación del ADC, en el proceso de conversión, el módulo almacena en el registro ADC_DR –registro de datos- el resultado de la conversión. Activa la bandera de fin de conversión y se produce la interrupción si la hemos configurado. El funcionamiento es similar para los canales inyectados, aunque con la diferencia de que, el resultado se almacenará en el registro ADC_DRJx correspondiente y se activará la bandera ADC_IT_JEOC de la interrupción, que solo se genera, si previamente se ha habilitado mediante el comando **ADC_ITConfig** (*ADC1, ADC_IT_JEOC, ENABLE*).

Para configurar un canal del grupo de inyectado, debemos realizar la siguiente estructura, muy similar a la que ya conocemos del ejemplo anterior.

```
1  /*  Creamos las estructuras para GPIO, ADC y NVIC   */
2  GPIO_InitTypeDef GPIO_InitStructure;
3  ADC_InitTypeDef ADC_InitStructure;
4  NVIC_InitTypeDef NVIC_InitStructre;
5
6  /* Habilitamos los reloj para GPIOA y ADC1    */
7  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_ADC1,ENABLE);
8
9  /* Habilitamos el reloj del ADC y establecemos su divisor */
10 RCC_ADCCLKConfig (RCC_PCLK2_Div6);
11
12 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
13 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
14 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
15 GPIO_Init(GPIOA, &GPIO_InitStructure);
16
17 /*  Configuramos los parametros del ADC    */
18 ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
19 ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
20 ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
21 ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
22 ADC_InitStructure.ADC_NbrOfChannel = 2;
23 ADC_InitStructure.ADC_ScanConvMode = ENABLE;
24 ADC_Init(ADC1, &ADC_InitStructure);
25
26 /* Establecemos los canales del grupo de inyectado a utilizar  */
27 ADC_InjectedSequencerLengthConfig(ADC1, 2);
28 ADC_InjectedChannelConfig(ADC1,ADC_Channel_0, 1,ADC_SampleTime_7Cycles5);
29 ADC_InjectedChannelConfig(ADC1,ADC_Channel_1, 2,ADC_SampleTime_7Cycles5);
30
31 ADC_ExternalTrigInjectedConvConfig(ADC1, ADC_ExternalTrigInjecConv_None);
32
33 ADC_Cmd(ADC1, ENABLE ) ;
34
35 ADC_ResetCalibration(ADC1);
36 while(ADC_GetResetCalibrationStatus(ADC1));
37 ADC_StartCalibration(ADC1);
38 while(ADC_GetCalibrationStatus(ADC1));
39
40 ADC_AutoInjectedConvCmd( ADC1, ENABLE );
41 ADC_SoftwareStartInjectedConvCmd ( ADC1 , ENABLE ) ;
42
```

Figura 13.18

En un primer bloque, creamos la estructura con la configuración GPIO del pin del canal o canales que vamos a emplear, también habilitamos los relojes correspondientes: el APB2 -del Puerto A y el ADC1-, y el ADCCLKC del módulo ADC.

Luego, en el siguiente bloque de nuestra estructura, configuramos normalmente los parámetros del ADC que ya conocemos. También necesitaremos habilitar y configurar los canales del grupo de inyectados que vamos a utilizar en nuestro proyecto.

Para ello, el comando **ADC_InjectedSequencerLnegthConfig**(*ADCx, uint8_t Largo*), nos configura la longitud del secuenciador para los canales seleccionados. Cuyo valor puede ser de 1 a 4.

A continuación, con el comando **ADC_InjectedChannelConfig**(*ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime*), se configuran: el número del ADC que queremos utilizar, el canal, su rango de secuenciador y el

tiempo de muestra que queremos. El rango podrá ser un valor entre 1 y 4. Los modos de sampleado podrán ser:

- **ADC_SampleTime_1Cycles5**: Tiempo de muestra de 1.5 ciclos
- **ADC_SampleTime_7Cycles5**: Tiempo de muestra de 7.5 ciclos
- **ADC_SampleTime_13Cycles5**: Tiempo de muestra de 13.5 ciclos
- **ADC_SampleTime_28Cycles5**: Tiempo de muestra de 28.5 ciclos
- **ADC_SampleTime_41Cycles5**: Tiempo de muestra de 41.5 ciclos
- **ADC_SampleTime_55Cycles5**: Tiempo de muestra de 55.5 ciclos
- **ADC_SampleTime_71Cycles5**: Tiempo de muestra de 71.5 ciclos
- **ADC_SampleTime_239Cycles5**: Tiempo de muestra de 239.5 ciclos

Figura 13.19

Después, con el comando **ADC_ExternalTrigInjectedConvConfig**(*ADCx, uint32_t ADC_ExternalTrigInjecConv*) configuramos el activador –*trigger*- externo que producirá el inicio de la conversión de los canales inyectados. Los cuales pueden activarse por alguno de los siguientes eventos:

- **ADC_ExternalTrigInjecConv_T1_TRGO**: Seleccionamos evento trigger en Timer1 (para ADC1, ADC2 y ADC3).
- **ADC_ExternalTrigInjecConv_T1_CC4**: Seleccionamos captura comparación4 en Timer1 (para ADC1, ADC2 y ADC3).
- **ADC_ExternalTrigInjecConv_T2_TRGO**: Seleccionamos evento trigger en Timer2 (para ADC1 y ADC2).
- **ADC_ExternalTrigInjecConv_T2_CC1**: Seleccionamos captura comparación1 en Timer2 (para ADC1 y ADC2).
- **ADC_ExternalTrigInjecConv_T3_CC4**: Seleccionamos captura comparación4 en Timer3 (para ADC1 y ADC2).
- **ADC_ExternalTrigInjecConv_T4_TRGO**: Seleccionamos evento trigger en Timer4 (para ADC1 y ADC2).
- **ADC_ExternalTrigInjecConv_Ext_IT15_TIM8_CC4**: : Seleccionamos una interrupción en línea 15 o captura comparación4 en Timer8 (para ADC1 y ADC2).
- **ADC_ExternalTrigInjecConv_T4_CC3**: Seleccionamos captura comparación3 en Timer4 (sólo ADC3).
- **ADC_ExternalTrigInjecConv_T8_CC2**: Seleccionamos captura comparación2 en Timer8 (sólo ADC3).
- **ADC_ExternalTrigInjecConv_T8_CC4**: Timer8 capture compare4 selected (for ADC3 only)
- **ADC_ExternalTrigInjecConv_T5_TRGO**: Seleccionamos evento trigger en Timer5 (sólo ADC3).
- **ADC_ExternalTrigInjecConv_T5_CC4**: Seleccionamos captura comparación4 en Timer5 (sólo ADC3).
- **ADC_ExternalTrigInjecConv_None**: Seleccionamos la captura se iniciará por software y no por activador evento externo (para ADC1, ADC2 y ADC3).

Figura 13.20

Los siguientes comandos son los de configuración y calibración del ADC, necesarios en todo inicio del módulo ADC.

En la parte del módulo principal "int main()", para obtener los valores de la conversión, y si hemos escogido el modo "**ADC_ExternalTrigInjecConv_None**", bastará con pasar a una variable el valor que nos es devuelto por el comando "**ADC_GetInjectedConversionValue** (*ADCx, ADC_InjectedChannel_x*).

A continuación mostramos el programa completo:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_usart.h"
#include "stm32f10x_adc.h"

#include "usart1.h"

/*        Funcion que configure el GPIO y ADC                */
//============================================================
void ADC_Injected_Config(void)
{
    /*  Creamos las estructuras para GPIO, ADC y NVIC    */
    GPIO_InitTypeDef GPIO_InitStructure;
    ADC_InitTypeDef ADC_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructre;

    /* Habilitamos los reloj para GPIOA y ADC1     */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_ADC1,ENABLE);

    /* Habilitamos el reloj del ADC y establecemos su divisor */
    RCC_ADCCLKConfig (RCC_PCLK2_Div6);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /*  Configuramos los parametros del ADC     */
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_InitStructure.ADC_NbrOfChannel = 2;
    ADC_InitStructure.ADC_ScanConvMode = ENABLE;
    ADC_Init(ADC1, &ADC_InitStructure);

    /* Establecemos los canales del grupo de inyectado a utilizar  */
    ADC_InjectedSequencerLengthConfig(ADC1, 2);
    ADC_InjectedChannelConfig(ADC1,ADC_Channel_0, 1,ADC_SampleTime_7Cycles5);
    ADC_InjectedChannelConfig(ADC1,ADC_Channel_1, 2,ADC_SampleTime_7Cycles5);

    ADC_ExternalTrigInjectedConvConfig(ADC1, ADC_ExternalTrigInjecConv_None);

    ADC_Cmd(ADC1, ENABLE ) ;

    ADC_ResetCalibration(ADC1);
    while(ADC_GetResetCalibrationStatus(ADC1));
    ADC_StartCalibration(ADC1);
    while(ADC_GetCalibrationStatus(ADC1));

    ADC_AutoInjectedConvCmd(ADC1, ENABLE );
    ADC_SoftwareStartInjectedConvCmd(ADC1, ENABLE);
}

/*              Modulo principal                          */
//============================================================
int main(void)
{
    uint16_t ADC_valor0, ADC_valor1;

    USART1_Config(115200);   // Inicializamos el USART1

    ADC_Injected_Config();   // Inicializamos el ADC inyectado

    while (1)
    {
        ADC_valor0 = ADC_GetInjectedConversionValue(ADC1,
ADC_InjectedChannel_1);
        ADC_valor1 = ADC_GetInjectedConversionValue(ADC1,
ADC_InjectedChannel_2);
        printf("Valores ADC [1]%d [2]%d \r\n", ADC_valor0, ADC_valor1);
    }
}
```

Figura 13.21

13.5 EJEMPLO ADC CON WATCHDOG. (AWD)

La función Watchdog nos es útil si necesitamos saber cuándo una señal analógica es igual a un determinado valor. Una vez que se produzca ese evento, podremos configurar que se realice un proceso determinado, para ello, podemos utilizar algunos de los canales ADC.

El *Watchdog* analógico **AWD**, es un medio para producir una interrupción cuando algún voltaje externo, que es supervisado por algún canal ADC, excede de un límite de umbral programable, ya sea por arriba o por abajo. Esta, se realizará sin intervención del sistema que solo la atenderá cuando se produzca.

El siguiente código es un ejemplo de este funcionamiento:

```
#include
    .
    .

// Creamos las variables de forma 'volatile'
volatile char buffer[50] = {'\0'};
volatile char ADC_IT_AWD_FLAG = 0;
    .
    .
    .

/*    Funcion que configura el watchdog ADC        */
//-------------------------------------------------
void ADC_Config(void)
{
    // Activamos el RCC de los canales ADC y del ADC
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
                           RCC_APB2Periph_ADC1, ENABLE);

    // Activamos el reloj 'ADCCLK' del ADC con divisor 6 (72/6 =12MHz)
    RCC_ADCCLKConfig(RCC_PCLK2_Div6);

    // Creamos la estructura GPIO para los canales ADC
    GPIO_InitTypeDef  GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AIN;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* Configuracion del NVIC Configuration */
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_InitStructure.NVIC_IRQChannel = ADC1_2_IRQn;   //IRQn ADC 1/2
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    /* Configuración del ADC                */
    ADC_InitTypeDef ADC_InitStructure;
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;

    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfChannel = 1;
    ADC_Init (ADC1, &ADC_InitStructure);
```

```
    // Configuramos el canal 0 del ADC1     */
    ADC_RegularChannelConfig(ADC1, ADC_Channel_0, 1, ADC_SampleTime_28Cycles5);

    /* Configuramos los umbrales mayor y menor del watchdog */
    ADC_AnalogWatchdogThresholdsConfig(ADC1, 3000, 1000);
    /* Configure channel14 as the single analog watchdog guarded channel */
    /* Configuamos el canal 0 como canal de vigilancia watchdog  */
    ADC_AnalogWatchdogSingleChannelConfig(ADC1, ADC_Channel_0);
    /* Enable analog watchdog on one regular channel */
    /* Habilitamos el canal del watchdog ADC1 como simple canal regular  */
    ADC_AnalogWatchdogCmd(ADC1, ADC_AnalogWatchdog_SingleRegEnable);

    /* Habilita la bandera de la interrupcion AWD */
    ADC_ITConfig(ADC1, ADC_IT_AWD, ENABLE);

    /* Inicializa el ADC1
    ADC_Cmd(ADC1, ENABLE);

    // Restablecemos el ADC para reiniciar los parámetros configurados
    ADC_ResetCalibration(ADC1);
    while(ADC_GetResetCalibrationStatus(ADC1));
    ADC_StartCalibration(ADC1);
    while(ADC_GetCalibrationStatus(ADC1));

    // Iniciamos el ADC1
    ADC_Cmd (ADC1,ENABLE);
    // Iniciamos el procesos de conversion en ADC1 en modo continuo
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);
}

/*   Funcion que se ejecutará cuando se detecte la interrupcion  */
//---------------------------------------------------------------
void ADC1_2_IRQHandler(void)
{
    if(ADC_GetITStatus(ADC1,ADC_IT_AWD))
    {
        ADC_IT_AWD_FLAG = 1;
        ADC_ClearITPendingBit(ADC1,ADC_IT_AWD);
    }
}

/*              Modulo  Principal                */
//-----------------------------------------------
int main(void)
{
    int adc_value;   // Creamos la variable que contendra el valor convertido

    AWD_Config();    // Iniciamos el watchdog del ADC

    USART1_Config(); // Inicializamos el USART1


    while (1)
    {
        // Comprobamos el estado de la bandera del AWD
        // Si el watchdog se ha activado, se lee el valor convertido
        if (ADC_IT_AWD_FLAG == 1) {
            // Pasamos a la variable el valor de conversion
            adc_value = ADC_GetConversionValue(ADC1);
            printf(" Valor ADC = %d\r\n", adc_value);
            ADC_IT_AWD_FLAG = 0;
        }
    }
}
```

Figura 13.22

Como vemos, la estructura de módulos a crear es muy similar a la que mostramos en nuestro primer ejemplo; donde para configurar el ADC, se comienza creando las estructuras necesarias: las del GPIO de los pines de los canales ADC, la del NVIC con la configuración de la interrupción que vamos usar y la del ADC; añadiendo solo las líneas siguientes que configuran el *watchdog* del ADC que necesitamos.

En la primera línea, con el comando "**ADC_AnalogWatchdogTheresholdsConfig**(ADCx, uint16_t *Umbral alto*, uint16_t *Umbral bajo*) es en la que especificamos los valores de voltaje mayor y menor que estableceremos como límites a controlar durante la conversión; que podrán contener valores de 12 bits.

En la siguiente línea, el comando "**ADC_AnalogWatchdogSingleChannelConfig** (*ADCx, ADC_Channel*[n])", nos permite especificar qué canal queremos que sea el watchdog que estamos configurando.

A continuación, con el comando "**ADC_AnalogWatchdogCmd**(*ADCx, ADC_AnalogWatchdog*) activamos o desactivamos el canal watchdog que hemos configurado y establecemos de qué tipo queremos que sea, de acuerdo a los siguientes tipos:

- ADC_AnalogWatchdog_**SingleRegEnable**: En un solo canal regular.
- ADC_AnalogWatchdog_**SingleInjecEnable**: En un solo canal inyectado.
- ADC_AnalogWatchdog_**SingleRegOrInjecEnable**: En un solo canal regular o inyectado.
- ADC_AnalogWatchdog_**AllRegEnable**: En todos los canales regulare.
- ADC_AnalogWatchdog_**AllInjecEnable**: En todos los canales inyectados.
- ADC_AnalogWatchdog_**AllRegAllInjecEnable**: En todos los canales regulares e inyectados.
- ADC_AnalogWatchdog_**None**: Ningun canal controlado por el watchdog.

Figura 13.23

Por último, con el comando "**ADC_ITConfig**(ADCx, **ADC_IT_AWD**, ENABLE) habilitamos o deshabilitamos la bandera de la interrupción que queremos controlar, que en nuestro caso, es la interrupción del watchdog ADC: "**ADC_IT_AWD**".

Más adelante en nuestra programación, necesitaremos crear la función de servicio que se ejecutará cuando el sistema detecte la interrupción, como en la Figura 13.24.

```
1  /*   Funcion que se ejecutará cuando se detecte la interrupcion  */
2  //-----------------------------------------------------------------
3  void ADC1_2_IRQHandler(void)
4  {
5      if(ADC_GetITStatus(ADC1, ADC_IT_AWD))
6      {
7          ....
8          ADC_ClearITPendingBit(ADC1, ADC_IT_AWD);
9          ....
10     }
11 }
```

Figura 13.24

La función de servicio que se ejecuta cuando se detecta esta interrupción, se nombra "**ADC1_2_IRQHandler**" en ella podemos establecer si se ha producido la interrupción o no comprobando el estado de la bandera mediante el comando "**ADC_GetITStatus**(ADCx, **ADC_IT_AWD**)", que nos devolverá un valor '1' si se ha producido, o un valor '0' si no.

Con el comando "**ADC_ClearITPendingBit** (ADCx, ADC_IT_AWD)" reseteamos a '0' la bandera de la interrupción para reiniciarla y que pueda ser detectada de nuevo.

13.6 MÓDULO DAC

Aunque incluimos la explicación de este módulo en el capítulo del ADC, realmente este es un periférico más e independiente dentro de nuestro microcontrolador.

El **DAC**, convertidor de digital a analógico (*Digital to Analog Converter*), es un dispositivo que realiza la función opuesta del ADC, ya que convierte un valor digital a una señal analógica de salida de voltaje correspondiente. Puede configurarse para trabajar a 8 o 12 bits, y al igual que el ADC, en el modo de 12 bits, los datos obtenidos pueden alinearse a la izquierda o a la derecha. Posee dos canales de salida (DAC_OUTx) cada uno con su propio convertidor, que están asignados a los pines **PA4** para el canal 1 y al pin **PA5** el canal 2. Pueden trabajar en modo dual, en el que sus conversores podrán configurarse de forma independiente cada uno o simultáneamente de forma síncrona. Utiliza también un pin de referencia de voltaje de entrada VREF+, que comparte con el ADC. También puede conectarse a una señal de disparo externa -*trigger*- de algunos de los Timer (TIMx_TRGO) para sincronizar la conversión.

Tiene asignado el reloj APB1.

El funcionamiento de este módulo se basa en la tecnología DDS (*Direct Digital Synthesis*) o Síntesis Digital Directa, donde un convertidor digital-analógico, convierte una señal de referencia a una señal sinusoidal. Las entradas digitales se convierten a voltajes de salida en una conversión lineal entre el valor '0' y el voltaje de referencia VREF+ (3.3Vcc en nuestra placa). Los voltajes de salida analógica en cada pin de canal DAC son determinados por la siguiente ecuación:

$$DAC_{Output}\ (Vcc) = \frac{DAC_DORx\ *VREF}{4096}$$

Figura 13.25

Cada canal posee una lógica de control independiente que se configuran a través de un único registro de control (CR). Los datos convertidos en el canal 1 o 2 se escriben en el registro temporal (DHRx – *Data Holding Register*), cuando se detecta el evento desencadenante programado, el contenido del registro (DHRx) pasa al registro de salida de datos (DORx – *Data Output Register*) y después de un tiempo de procesado, el valor analógico obtenido, sale por la salida correspondiente. El registro DORx se actualiza cada tres ciclos del reloj APB1 después de que se produce el desencadenante.

El evento desencadenante, puede producirse: cuando el DAC detecta un flanco ascendente en la salida *trigger* de alguno de los Timer (TIMx_TRGO) seleccionado, o, utilizando la línea 9 de una interrupción EXTI_9, en el que los datos almacenados en el registro temporal (DHRx) pasan al registro (DORx).

También es posible configurar el desencadenante o disparador del DAC mediante software; que se iniciará mediante el establecimiento del bit SWTRIG.

A continuación, vemos un ejemplo de la configuración de un módulo DAC:

```
1  /*  Funcion que configura el DAC     */
2  //--------------------------------------
3  void DAC_Config(void)
4  {
5    /* Configuracion del GPIO de PA4 y PA5  */
6    GPIO_InitTypeDef GPIO_InitStructure;
7    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
8    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_4 | GPIO_Pin_5;
9    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AIN;  // Salida analógica
10   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
11   GPIO_Init(GPIOA, &GPIO_InitStructure);
12
13   /* Configuracion del DAC          */
14   RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
15   DAC_InitTypeDef DAC_InitStructure;
16   DAC_InitStructure.DAC_OutputBuffer  = DAC_OutputBuffer_Enable;
17   DAC_InitStructure.DAC_Trigger       = DAC_Trigger_None;
18   DAC_InitStructure.DAC_WaveGeneration= DAC_WaveGeneration_None;
19   DAC_Init(DAC_Channel_1, &DAC_InitStructure);
20   DAC_Init(DAC_Channel_2, &DAC_InitStructure);
21
22   // habilitamos canales del DAC
23   DAC_Cmd(DAC_Channel_1, ENABLE);
24   DAC_Cmd(DAC_Channel_2, ENABLE);
25 }
```

Figura 13.26

Debemos señalar que, en la configuración GPIO de los pines de los canales DAC, se deben configurar como pines analógicos de entrada, mediante el "GPIO_Mode = **GPIO_Mode_AIN**", aunque para generar la salida analógica sabemos

que dicho pin se comportará como de salida. El DAC en sí saldrá por ese pin, lo que sucede es que, al habilitar un canal DACx, el pin correspondiente se conectará automáticamente a la salida del DAC de forma interna. Recordemos que el PA4 es para el canal 1 y PA5 para el canal 2.

El primer comando, "**DAC_InitTypeDef**" nos inicia para poder crear la estructura con los parámetros del DAC.

Mediante el comando "**DAC_OutputBuffer**", podemos habilitar el buffer de salida del canal DAC con el parámetro "**DAC_OutputBuffer_Enable**", o inhabilitarlo con el parámetro "**DAC_OutputBuffer_Disable**".

Después, con el comando "**DAC_Trigger**", definimos qué evento o desencadenante queremos que sea el que inicie la conversión. Podemos escoger cualquiera de los siguientes:

- **DAC_Trigger_None** – Conversión automática una vez se ha cargado el registro DHRx, sin dispador externo alguno.
- **DAC_Trigger_T3_TRGO** - Evento TRIG del TIM3.
- **DAC_Trigger_T2_TRGO** - Evento TRIG del TIM2.
- **DAC_Trigger_T4_TRGO** - Evento TZRIG del TIM3.
- **DAC_Trigger_Ext_IT9** - Evento por interrupción externa en línea 19.
- **DAC_Trigger_Software** - Conversión iniciada por disparado por software.

Figura 13.27

Mediante el comando "**DAC_WaveGeneration**" configuramos el tipo de onda que deberá generarse a la salida y estos tipos pueden ser:

- **DAC_WaveGeneration_None** – No se generan ondas de ruido en el canal.
- **DAC_WaveGeneration_Noise** – Especifica que se generan ondas de ruido en el canal.
- **DAC_WaveGeneration_Triangle** – Se generan ondas triangulares.

Figura 13.28

Hay otro parámetro a introducir en la configuración del módulo DAC: el que especifica la máscara LFSR para generar ondas de ruido o crear la amplitud máxima de las ondas triangulares para el canal DAC, mediante el comando "**DAC_LFSRUnmask_TriangleAmplitude**". No incluimos los detalles de los modos posibles para la configuración de este parámetro, ya que no se usarán en este libro.

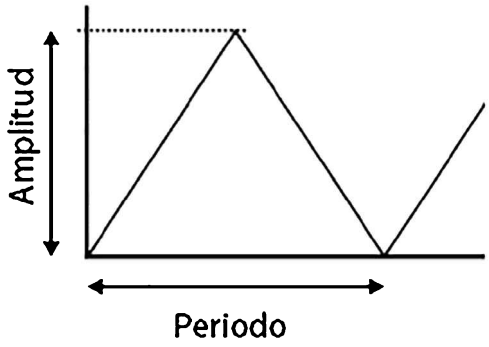


Figura 13.29

Con el comando "**DAC_Init** (*DAC_Channel*[n], & [NombreEstructureDAC]) definimos qué canal será [n] el seleccionado para la configuración que hemos establecido.

Y por último, con "**DAC_Cmd**(*DAC_Channe*l[n])" habilitamos e iniciamos el canal DAC programado.

Para escribir datos en los registros del DAC se utiliza el comando "**DAC_ SetChannel[n]Data** (*CH_alineación*, uint16_t *datos*)" donde [n], será 1 o 2 para especificar el canal correspondiente; el valor "*datos*" será el que queremos cargar en el registro de retención, y el parámetro "*CH_alineación*" corresponderá al tipo de alineación de los datos introducidos de acuerdo a los siguientes modos:

- **DAC_Align_8b_R** - 8bit con alineación de datos a la derecha
- **DAC_Align_12b_L** - 12bit con alineación de datos a la izquierda.
- **DAC_Align_12b_R** - 12bit con alineación de datos a la derecha.

Figura 13.30

Mediante el comando "**DAT_GetDataOutputValue** (DAC_Channel[n]), leemos el último valor de salida de datos del canal DAC seleccionado por [n].

A continuación incluimos un código de ejemplo, donde configuramos el canal 1 que generará una señal de salida analógica de diente de sierra.

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_dac.h"

/*  Funcion que configura el DAC     */
//--------------------------------------
void DAC_Config (void)
{
  /* Configuracion del GPIO de PA4  */
  GPIO_InitTypeDef GPIO_InitStructure;
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_4; // DAC_CH1
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AIN;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  /* Configuracion del DAC          */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
  DAC_InitTypeDef DAC_InitStructure;
  DAC_InitStructure.DAC_OutputBuffer  = DAC_OutputBuffer_Enable;
  DAC_InitStructure.DAC_Trigger       = DAC_Trigger_None;
  DAC_InitStructure.DAC_WaveGeneration= DAC_WaveGeneration_None;
  DAC_Init(DAC_Channel_1, &DAC_InitStructure);

  // Habilitamos canales del DAC
  DAC_Cmd(DAC_Channel_1, ENABLE);
}


/*          Modulo principal      */
//=======================================
int main ()
{
  DAC_Config();   // Inicializamos el DAC

  while (1) {

    for (int valor=0; valor< 4000; valor++)
      {

      DAC_SetChannel1Data(DAC_Align_12b_R, valor);
      }
  }
}
```

Figura 13.31

14

PROGRAMACIÓN DMA



DMA (*Direct Memory Access*) es el controlador interno de acceso a la memoria del microcontrolador. Su principal función es la de transferir datos desde los periféricos a la memoria interna del microcontrolador.

Nuestro microcontrolador el STM32F103C8 posee dos controladores **DMA** con 12 canales (7 para el **DMA1** y 5 para el **DMA2**); cada uno de ellos está dedicado a gestionar solicitudes de acceso a memoria desde uno o más periféricos. Podemos verlos con detalle en las siguientes Figuras 14-1 y 14-2.

Tabla 78. Lista de periféricos y canales del DMA1

Periférico	canal 1	canal 2	canal 3	canal 4	canal 5	canal 6	canal 7
ADC1	ADC1	-	-	-	-	-	-
SPI / I²S	-	SPI1_RX	SPI1_TX	SPI2/I2S2_RX	SPI2/I2S2_TX	-	-
USART	-	USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I²C	-	-	-	I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1	-	TIM1_CH1	-	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	-
TIM2	TIM2_CH3	TIM2_UP	-	-	TIM2_CH1	-	TIM2_CH2 TIM2_CH4
TIM3	-	TIM3_CH3	TIM3_CH4 TIM3_UP	-	-	TIM3_CH1 TIM3_TRIG	-
TIM4	TIM4_CH1	-	-	TIM4_CH2	TIM4_CH3	-	TIM4_UP

Figura 14.1

Tabla 79. Lista de periféricos y canales del DMA2

Periférico	canal 1	canal 2	canal 3	canal 4	canal 5
ADC3(1)					ADC3
SPI/I2S3	SPI/I2S3_RX	SPI/I2S3_TX			
UART4			UART4_RX		UART4_TX
SDIO(1)				SDIO	
TIM5	TIM5_CH4 TIM5_TRIG	TIM5_CH3 TIM5_UP		TIM5_CH2	TIM5_CH1
TIM6/ DAC_Channel1			TIM6_UP/ DAC_Channel1		
TIM7				TIM7_UP/ DAC_Channel2	
TIM8	TIM8_CH3 TIM8_UP	TIM8_CH4 TIM8_TRIG TIM8_COM	TIM8_CH1		TIM8_CH2

Figura 14.2

Cada canal puede configurarse de modo independiente, activando o desactivando el bit de control DMA en los registros del periférico correspondiente. Se puede acceder a los registros DMA de los periféricos por medio de valores de 8, 16 y 32 bits, aunque solo un dispositivo puede acceder al canal DMA en un determinado momento.

Las prioridades entre las solicitudes de los canales, que son programables también por software, poseen 4 niveles que priorizan desde muy alto, alto, medio o bajo. Las direcciones de origen o destino deben estar alineadas con el tamaño de los datos que vayan a manejar.

Existe un módulo que realiza las funciones de árbitro que maneja la prioridad entre las solicitudes de DMA.

Al igual que con otros periféricos, es necesario crear previamente la estructura del DMA para configurar sus parámetros. Según se muestra en la Figura 14.3.

```
1  /* Creamos la estructura el DMA   ---------------------------------*/
2  DMA_InitTypeDef DMA_InitStructure;
3
4  /* Habilitamos reloj para DMA1 -----------------------------------*/
5  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
6
7  /* Configuracion del canal 4 del DMA1 para Tx USART1 -------------*/
8  DMA_DeInit(DMA1_Channel4);
9  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&USART1->DR;
10 DMA_InitStructure.DMA_MemoryBaseAddr     = (uint32_t)TxBuffer1;
11 DMA_InitStructure.DMA_DIR               = DMA_DIR_PeripheralDST;
12 DMA_InitStructure.DMA_BufferSize        = TxBufferSize1;
13 DMA_InitStructure.DMA_PeripheralInc     = DMA_PeripheralInc_Disable;
14 DMA_InitStructure.DMA_MemoryInc         = DMA_MemoryInc_Enable;
15 DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
16 DMA_InitStructure.DMA_MemoryDataSize     = DMA_MemoryDataSize_Byte;
17 DMA_InitStructure.DMA_Mode              = DMA_Mode_Normal;
18 DMA_InitStructure.DMA_Priority          = DMA_Priority_VeryHigh;
19 DMA_InitStructure.DMA_M2M               = DMA_M2M_Disable;
20
21 DMA_Init(DMA1_Channel4, &DMA_InitStructure);
22
23 /* Habilitamos el canal 4 del DMA1 para Tx ----------------------*/
24 DMA_Cmd(DMA1_Channel4, ENABLE);
```

Figura 14.3

Utilizamos el comando **DMA_InitTypeDef,** que inicia la creación de la estructura y que en nuestro ejemplo nombramos como "*DMA_InitStructure*", procediendo a configurar cada uno de los parámetros necesarios.

El primer parámetro debe ser para habilitar el reloj correspondiente mediante el comando **RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA[x]**,*ENABLE*); en el que observamos que el reloj que controla el bus DMA es diferente a los que hasta ahora hemos seleccionado para otros periféricos, y que es el **AHB** (*Advanced High-performance Bus*).

La mayoría de periféricos internos de nuestro microcontrolador utilizan los relojes de sincronización APBx (*Advanced Peripheral Bus*), sin embargo, existe un reducido número de módulos internos que utilizan este reloj de sincronización

directa. Cuando se realiza una transferencia de datos por medio del DMA desde una fuente a un destino, este reloj, el **AHB**, detiene el resto de accesos al bus DMA e inserta el acceso, permitiendo que la transferencia se lleve a cabo.

Con el comando, "**DMA_DeInit(***DMA[x]_ChannelNN***)**", damos la orden al sistema para que se resetee la configuración que inicialmente posean los registros del módulo DMA seleccionado con el valor de [x], que puede ser 1 o 2; y del canal que estemos programando indicado en el valor NN.

El siguiente comando es "**DMA_PeripheralBaseAddr** = (unit32_t) *Periferico*–>**DR"**, que especifica la dirección de memoria que contiene el registro de datos del periférico del que se va a recibir la información. Cuyo formato para especificar la dirección del registro de datos del periférico será: especificando el nombre del periférico seguido del símbolo de una flecha hacia la derecha, seguido de las letras DR para indicar "registro de datos", todo ello, expresado en una variable del tipo 32 bits.

Podemos ver algunos ejemplos de sintaxis para diferentes periféricos en las siguientes líneas:

```
// Ejemplo de configuración para USART1
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) & USART1->DR;

// Ejemplo de configuración para canal Rx del SPI
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) SPIx_DR_ADDRESS;

// Ejemplo de configuración del canal 1 del ADC1
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) & ADC1->DR;

// Ejemplo de configuración del canal 1 del TIMER_3
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) TIM3_CCR1_Address;

// Ejemplo en el que se especifica la dirección de memoria
// de forma directa a una dirección específica del DAC
DMA_PeripheralBaseAddr = (uint32_t)(DAC_BASE+0x10);
```

Figura 14.4

El siguiente comando "**DMA_MemoryBaseAddr** = (uint32_t) destination", configura la dirección base de la memoria origen de los datos.

El comando "**DMA_DIR"**, es donde vamos a indicar si el periférico será la fuente, con el parámetro "**DMA_DIR_PeripheralSRC**" o el destino, con "**DMA_DIR_PeripheralDST**". Donde los datos pueden ser transferidos desde el periférico a la memoria y viceversa.

Con el comando "**DMA_BufferSize**" especificamos el tamaño del buffer que recogerá los datos. Y que dependerá también, del valor que se establezca en los otros comandos *DMA_PeripheralDataSize o DMA_MemoryDataSize.*

Con "**DMA_PeripheralInc**", indicamos si queremos que se incremente la dirección del registro de datos que configuramos y de donde obtendremos la información. Podremos indicar que sí se incrementa, mediante el parámetro "**DMA_PeripheralInc_Enable**", o que no, con "**DMA_PeripheralInc_Disable**".

Después, con el comando "**DMA_MemoryInc**", le indicamos si queremos que la dirección de memoria de datos se incremente o no, con "**DMA_MemoryInc_Enable**", o con "**DMA_MemoryInc_Disable**".

Mediante el comando "**DMA_PeripheralDataSize**", configuramos el tamaño de los datos del periférico y con el comando "**DMA_MemoryDataSize**", el tamaño de la memoria de datos; en donde estos comandos pueden ser cualquiera de las siguientes tres opciones:

- **DMA_Peripheral/Memory...DataSize_Byte**, tamaño de 8 bit (-128 ... +127).
- **DMA_Peripheral/Memory...DataSize_HalfWord**, tamaño de 16 bit (-32768 ... +32767).
- **DMA_Peripheral/Memory...DataSize_Word**, tamaño de 32 bit (-2147483648 ... + 2147483647) (0xFFFF FFFF).

Figura 14.5

Otro comando necesario en nuestra configuración será el "**DMA_Mode**", donde establecemos el modo en que queremos que funcione el canal DMA que configuramos.

En modo normal, que se establece con el parámetro "**DMA_Mode_Normal**", en el que solo se accede a la información una vez, y, a continuación el flujo es desactivado hasta que se solicite otra transferencia.

Modo cíclico, establecido con "**DMA_Mode_Circular"**, donde se leen o escriben datos en una memoria buffer de determinado tamaño, y en la que una vez llegados al final se regresará al principio para continuar el proceso. Se recomienda utilizar este modo si se necesita actualizar la información para cada ciclo y poder utilizar mejor la lectura de los datos.

Otro parámetro será el "**DMA_Priority**", con el que configuramos la prioridad por software del canal DMA establecido, que podrá ser:

- **DMA_Priority_VeryHigh.**
- **DMA_Priority_High.**
- **DMA_Priority_Medium.**
- **DMA_Priority_Low.**

Figura 14.6

Y el comando "**DMA_M2M**", donde establecemos si la transferencia se realizará o no de memoria a memoria, se configura con las siguientes opciones: "**DMA_M2M_Enable**" o "**DMA_M2M_Disable**".

El último comando, "**DMA_Init** (*DMA[x]_ChannelNN*, &DMA_ InitStructure), en donde inicializamos el DMA y el canal con los parámetros que hemos configurado; que luego con el comando "**DMA_Cmd**(DMA[x], ENABLE)" iniciamos.

14.1 EJEMPLO DE COMUNICACIÓN SERIAL CON EL USART EMPLEANDO EL DMA

Una de las situaciones más comunes que se producen en la mayoría de proyectos, donde necesitamos leer o escribir datos desde una dirección de memoria o un periférico es, aquella en la que durante la ejecución de un programa, el sistema se queda esperando a que la transferencia se produzca, impidiendo que se puedan realizar otras tareas. Ese tiempo de espera será crítico en determinados procesos donde la velocidad de transferencia es muy importante, como en los módulo de comunicaciones USARTx, SPI, CAN, etc…

Es por ello que el uso del DMA en procesos con dispositivos de hardware específico, donde se comparten el uso de la memoria y los buses, produce importantes resultados.

En nuestro siguiente ejemplo, utilizamos el DMA para enviar de nuevo los datos que recibimos por el puerto USART; cuando el sistema detecta que se han recibidos 40 caracteres los reenvía de nuevo por el puerto USART.

```
1  #include "stm32f10x.h"
2  #include "stm32f10x_gpio.h"
3  #include "stm32f10x_rcc.h"
4  #include "stm32f10x_dma.h"
5  #include "stm32f10x_usart.h"
6
7  // Creamos la variable estructura 'TestSatus"
8  // conteniendo los valores 'FAILED' y 'PASSED'
9  typedef enum {FAILED = 0, PASSED = !FAILED} TestStatus;
10 // Variable que guarda el tamano del Buffer1
11 #define TxBufferSize1 (cuentaFinal(TxBuffer1) - 1)
12 // Variable que guarda el tamano del Buffer2
```

```
13 #define TxBufferSize2 (cuentaFinal(TxBuffer2) - 1)
14 // variable que guarda el valor de comparacion de tamano Buffer
15 #define cuentaFinal(a) (sizeof(a) / sizeof(*(a)))
16
17 uint8_t TxBuffer1[] = "Prueba de DMA en USART1 ---------------";
18 uint8_t RxBuffer1[TxBufferSize1];
19
20 //  Lista de funciones
21 void USART1_Config(void);
22 void DMA_Configuration(void);
23 TestStatus Buffercmp(uint8_t* pBuffer1, uint8_t* pBuffer2, uint16_t BufferLength);
24 void uartStartTxDMA(DMA_Channel_TypeDef *txDMAChannel);
25
26 /*        Modulo Principal                        */
27 //================================================
28 int main()
29 {
30   USART1_Config();       // Inicializamos el USART1
31
32   DMA_Configuration();  // Inicializamos el DMA
33
34   while(1)
35   {
36     // Esperamos hasta que  la transferencia se complete en canal 4 DMA
37     while (DMA_GetFlagStatus(DMA1_FLAG_TC4) == RESET)
38     {
39     }
40
41     // Esperamos hasta que la recepción se complete en canal 5 DMA
42     while (DMA_GetFlagStatus(DMA1_FLAG_TC5) == RESET)
43     {
44     }
45
46     // Se va añadiendo los datos recibidos a la variable de datos
47     // para transmitidos
48     for (int i=0; i < TxBufferSize1; ++i)
49     {
50       TxBuffer1[ i] = RxBuffer1[ i];
51     }
52
53     // Reiniciamos la bandera de datos pendiente en los dos canales
54     DMA_ClearITPendingBit(DMA1_IT_TC4 | DMA1_IT_TC5);
55
56     // Desabilitamos el canal 4 del DMA1
57     DMA_Cmd(DMA1_Channel4, DISABLE);
58
59     // Llamada a la funcion que transmite los datos recibidos
60     uartStartTxDMA(DMA1_Channel4);
61   }
62 }
63
64 /*           Funcion que configura el USART1              */
65 //-------------------------------------------------------------
66 void USART1_Config(void)
67 {
68
69   GPIO_InitTypeDef GPIO_InitStructure;
70
71   // Habilitamos los relojes para GPIO, USART y AFIO
72   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
73                          RCC_APB2Periph_USART1|
74                          RCC_APB2Periph_AFIO, ENABLE);
75
76   // Configuramos el pin 10 para Rx USART1
77   GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_10;
```

```
78    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
79    GPIO_Init(GPIOA, &GPIO_InitStructure);
80
81    // Configuramos el pin 9 para Tx USART1
82    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_9;
83    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
84    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF_PP;
85    GPIO_Init(GPIOA, &GPIO_InitStructure);
86
87    // Configuracion del USART1
88    USART_InitTypeDef USART_InitStructure;
89    USART_InitStructure.USART_BaudRate     = 115200;
90    USART_InitStructure.USART_WordLength   = USART_WordLength_8b;
91    USART_InitStructure.USART_StopBits     = USART_StopBits_1;
92    USART_InitStructure.USART_Parity       = USART_Parity_No;
93    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
94    USART_InitStructure.USART_Mode         = USART_Mode_Rx | USART_Mode_Tx;
95    USART_Init(USART1, &USART_InitStructure);
96
97    // Habilitamos las baderas Tx y Rx USART para el DMA
98    USART_DMACmd(USART1, USART_DMAReq_Tx | USART_DMAReq_Rx, ENABLE);
99
100   // Habilitamos el USART1
101   USART_Cmd(USART1, ENABLE);
102}
103
104/*    Funcion que transmite los datos recibidos por USART1 */
105//-------------------------------------------------------------
106void uartStartTxDMA(DMA_Channel_TypeDef *txDMAChannel)
107{
108   // Escribimos los datos en los registros CMAR y CNDTR del modulo DMA
109   txDMAChannel->CMAR = (uint32_t)TxBuffer1; // Registro Memory address de DMA
110   txDMAChannel->CNDTR = TxBufferSize1;      // Registro Numero de datos a
111                                                transferir por DMA
112   DMA_Cmd(txDMAChannel, ENABLE);
113}
114
115/*     Funcion que configura el DMA y canales 4 y 5        */
116//-------------------------------------------------------------
117void DMA_Configuration(void)
118{
119   DMA_InitTypeDef DMA_InitStructure;
120
121   // Habilitamos reloj para DMA1
122   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
123
124   // Configuracion del canal 4 del DMA1 para Tx USART1
125   DMA_DeInit(DMA1_Channel4);
126   DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&USART1->DR;
127   DMA_InitStructure.DMA_MemoryBaseAddr     = (uint32_t)TxBuffer1;
128   DMA_InitStructure.DMA_DIR                = DMA_DIR_PeripheralDST;
129   DMA_InitStructure.DMA_BufferSize         = TxBufferSize1;
130   DMA_InitStructure.DMA_PeripheralInc      = DMA_PeripheralInc_Disable;
131   DMA_InitStructure.DMA_MemoryInc          = DMA_MemoryInc_Enable;
132   DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
133   DMA_InitStructure.DMA_MemoryDataSize     = DMA_MemoryDataSize_Byte;
134   DMA_InitStructure.DMA_Mode               = DMA_Mode_Normal;
135   DMA_InitStructure.DMA_Priority           = DMA_Priority_VeryHigh;
136   DMA_InitStructure.DMA_M2M                = DMA_M2M_Disable;
137   DMA_Init(DMA1_Channel4, &DMA_InitStructure);
138
139   // Habilitamos el canal 4 del DMA1 para Tx
140   DMA_Cmd(DMA1_Channel4, ENABLE);
141
```

```
142    // Configuracion del canal 5 del DMA1 para Rx USART1
143    DMA_DeInit(DMA1_Channel5);
144    DMA_InitStructure.DMA_PeripheralBaseAddr   = (uint32_t)&USART1->DR;
145    DMA_InitStructure.DMA_MemoryBaseAddr       = (uint32_t)RxBuffer1;
146    DMA_InitStructure.DMA_DIR                  = DMA_DIR_PeripheralSRC;
147    DMA_InitStructure.DMA_BufferSize           = TxBufferSize1;
148    DMA_InitStructure.DMA_Mode                 = DMA_Mode_Circular;
149    DMA_Init(DMA1_Channel5, &DMA_InitStructure);
150
151    //Habilitamos el canal 5 del DMA1 para Rx
152    DMA_Cmd(DMA1_Channel5, ENABLE);
153}
154
155/*   Funcion quee realiza la comparación de los buffers1 y 2                */
156//---------------------------------------------------------------------------
157TestStatus Buffercmp(uint8_t* pBuffer1, uint8_t* pBuffer2, uint16_t BufferLength)
158{
159  // Espera a que el tamano del 'BufferLength' llegue a su limite
160  while(BufferLength--)
161  {
162    // Compara el tamano de cada 'pBufferN' sea distinto
163    if(*pBuffer1 != *pBuffer2)
164    {
165      return FAILED;
166    }
167
168    // Incrementamos cada 'pBuff...'
169    pBuffer1++;
170    pBuffer2++;
171  }
172  return PASSED;
173}
```

Figura 14.7

En nuestro ejemplo, comenzamos creando una serie de variables que utilizaremos en nuestra configuración y ejecución posterior.

A continuación, se crean la función, "**USART1_Config**()" que contendrá la estructura GPIO de los pines y USART con la configuración de comunicación RS232 que utilizamos para realizar la transferencia de los datos desde nuestro ordenador.

Creamos también, la función "**DMA_Configuration**()" en la que establecemos los parámetros con la configuración de nuestros canales DMA que vamos a utilizar. Donde creamos una estructura con la configuración para el canal 14 del DMA, y que establecemos para la escritura de los datos que enviaremos por el pin Tx del USART1. También establecemos el canal 15 del DMA, para leer los datos recibidos en el Rx del USART1.

Además, en nuestro código, debemos crear dos funciones más, una en la que se compare el tamaño del buffer de entrada de Rx y así calcular el número de dígitos que hemos establecido como límite de datos recibidos en 40, la función "**TestSatus Buffercmp**()" y la otra, "**uartStartTxDMA**()" que utilizamos para

enviar el contenido del buffer Tx que hemos obtenido de nuevo por el mismo puerto USART hacia nuestro ordenador.

En esta última función, que utilizamos para escribir los datos recibidos en el canal 4 (Tx) del DMA, para ser nuevamente enviados por el puerto USART; realizamos la transmisión mediante el acceso a los registros del DMA; donde primero cargamos la dirección de memoria del periférico destino donde enviamos los datos en el registro CMAR del DMA, mediante la instrucción siguiente:

txDMAChannel->**CMAR** = (uint32_t)TxBuffer1;

Después, cargamos en el registro DMA, **CNDTR,** la cantidad de bytes que vamos a enviar, mediante la instrucción:

txDMAChannel->**CNDTR** = TxBufferSize1;

Una vez cargados los dos registros, se habilita la transferencia de datos DMA, mediante el comando "**DMA_Cmd**(*DMAChannel[nn]*, ENABLE" para que se produzca.

15

PROGRAMACIÓN RTC



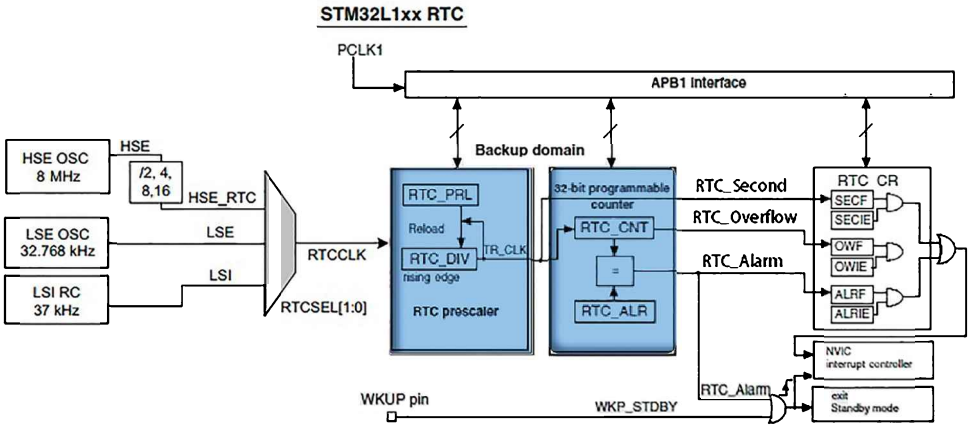


Figura 15.1

El **RTC** (*Real-Time clock*) o reloj en tiempo real, es un módulo de reloj interno que posee nuestro microcontrolador que actúa como temporizador y contador continuo en formato BCD. Puede programarse para ser utilizado como: cronómetro, alarma, generar calendarios, marcar tiempos y como referencia de calibración. Funciona de forma independiente y la información de sus valores de configuración y sus registros de tiempo, se pueden mantener alimentados con una conexión externa a una batería de 3 Vcc del tipo CR2025 o CR2032, que se conecta al pin VBAT y que no se reinician con un restablecimiento del sistema.

Puede utilizar como fuente de reloj cualquiera de los tres relojes que posee nuestro microcontrolador: HSE, LSE o LSI. Aunque el único que tiene la propiedad de mantener su información mediante el sistema de alimentación con batería es el LSE.

Se le puede configurar hasta tres líneas de interrupción que pueden generar eventos de alarma; una que se dispare cuando el contador del reloj llegue a un periodo de 1 segundo; otra que se produzca cuando el contador llegue a un determinado valor inicializado como valor de alarma; y una tercera interrupción, que se produzca cuando el contador llegue a un desbordamiento (*Overflow*) establecido.

Básicamente su funcionamiento es el siguiente:

El módulo RTC utiliza unos registros especiales (*backup registers*) del módulo BKP, los cuales sabemos, que conectando a nuestra placa una batería externa de 3.3 Vcc se mantienen aunque la alimentación principal no esté o se produzca un reinicio del sistema. Estos se protegen contra cualquier escritura al iniciarse el sistema, por lo que, para poder escribir o modificar sus valores, es necesario deshabilitar esta protección.

También es necesario establecer alguno de los tres relojes como fuente de sincronización RTC_CLK; estableciendo un valor determinado de prescaler para que dicho reloj funcione a una frecuencia de 1 Hz, que permite que el contador RTC se incremente cada 1 segundo. Para ello, establecemos también, que se genere una interrupción transcurrido ese periodo de 1 segundo.

A continuación, es necesario que el valor inicial a cargar en el reloj RTC de horas, minutos y segundos, se convierta todo a un valor único en segundos. Este, será el que se carga en los registros del RTC para que se inicie el contador interno, que se incrementará cada 1 segundo. Este valor deberá estar en formato BCD, que es la forma de funcionamiento interno.

Después, para leer y saber el valor acumulado en los registros del contador RTC, debemos obtener el valor del contador, que está en segundos totales, y

convertirlos de nuevo a horas, minutos y segundos, que son los que indicarán la hora actual.

Los pasos para establecer la configuración de un RTC con la fuente del reloj LSI se explican a continuación:

```
   /* Iniciamos el reloj APB1 para PWR y BKP          */
1  RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR |
                          RCC_APB1Periph_BKP, ENABLE);

   /* Permitir acceso a los registros BKP             */
2  PWR_BackupAccessCmd(ENABLE);

   /* Iniciar los registros BKP                       */
3  BKP_DeInit();

   /* Habilitar reloj LSI                             */
4  RCC_LSICmd(ENABLE);

   /* Espere hasta que el LSI esté listo              */
5  while(RCC_GetFlagStatus(RCC_FLAG_LSIRDY)==RESET)
   {;}

   /* Seleccionamos LSI como fuente de reloj del RTC */
6  RCC_RTCCLKConfig(RCC_RTCCLKSource_LSI);

   /*  Habilita el reloj RTC                          */
7  RCC_RTCCLKCmd(ENABLE);

   /* Espera a que se sincronice los registros RTC    */
8  RTC_WaitForSynchro();

   /* Espera hasta que finalice la última operación en RTC          */
9  RTC_WaitForLastTask();

   /* Habilitar la Interrupcio RTC por Segundos                     */
10 RTC_ITConfig(RTC_IT_SEC, ENABLE);

   /* Establecemos el Preescaler del RTC para período de 1 segundo */
   /* Peridodo RTC = RTCCLK/RTC_Prescal = (40 000 Hz)/39999+1)      */
11 RTC_SetPrescaler(39999);

   /* Espere hasta que finalice la última operación en RTC          */
12 RTC_WaitForLastTask();
```

Figura 15.2

En primer lugar, debemos habilitar los relojes para los módulos PWR y BKP, a los que les corresponde el reloj APB1.

En segundo lugar, necesitamos deshabilitar la protección de sobrescritura en los registros de *backup*; para lo que emplearemos el comando **PWR_BackupAccessCmd**(*ENABLE*), que nos habilitará su acceso.

El tercer punto, será iniciar los registros Backup para establecer un nuevo valor, para lo que se emplea el comando **BKP_DeInit**().

En el siguiente punto, iniciaremos con el comando **RCC_LSICmd** (ENABLE), el reloj LSE y en la siguiente línea, esperaremos a que este se inicie mediante el empleo de un comando '*while*...' y la comprobación de la bandera "**RCC_FLAG_LSIRDY**", que nos avisará cuando esté operativo.

Como hemos habilitado el reloj LSI, ahora en la siguiente línea, la número 6, establecemos que este sea el reloj origen para nuestro módulo RTC mediante el comando **RCC_RTCCLKConfig**(RCC_RTCCLKSource_LSI). A continuación, iniciamos el reloj del RTC en nuestra siguiente línea con el comando **RCC_ RTCCLKCmd**(ENABLE).

En la línea 8, realizamos una pausa para que los registros se configuren, para lo que empleamos el comando **RTC_WaitForSynchro**().

Y en la línea 9, volvemos a realizar una espera para que se ajusten los registros con el comando **RTC_WaitForLastTask**().

En la línea 10, mediante el comando **RTC_ITConfig(RTC_IT_xxx**, *ENABLE*), habilitamos (ENABLE) o deshabilitamos (DISABLE) la interrupción del RTC y establecemos cualquiera de las tres posibilidades:

- **RTC_IT_SEC** (*Second interrupt)* – Interrupción por contador de segundos.
- **RTC_IT_ALR** (*Alarm interrupt*) – Interrupción por alarma en contador.
- **RTC_IT_OW** (*Overflow interrupt*) – Interrupción por desbordamiento.

Figura 15.3

En nuestro ejemplo anterior, hemos habilitado la interrupción del RTC por segundos.

En nuestra siguiente línea, la número 11, configuramos el valor del Prescaler del RTC, del que dependerá con qué frecuencia se incrementará nuestro contador principal mediante la siguiente fórmula:

$$RTC_Clock = \frac{Frec_RTC_CLK}{RTC_Presc + 1}$$

Figura 15.4

En nuestro ejemplo establecemos que el reloj del RTC se alimente del reloj LSI, que tiene una frecuencia de 40 kHz; que si la dividimos por 40000, obtendremos una frecuencia de 1 Hz que equivale a que se incremente el contador del RTC cada 1 segundo. Para ello, establecemos un valor de prescaler de 39999 mediante el comando **RTC_SetPrescaler**(39999).

Por último, en la línea 12, volvemos a realizar una espera para que se ajusten los registros con el comando ya utilizado anteriormente, **RTC_WaitForLastTask**().

Bien, hasta aquí hemos establecido la configuración inicial de nuestro RTC; lo siguiente será indicar a partir de qué tiempo queremos que se inicie nuestro reloj. Para ello es importante recordar, que el contador que guarda el tiempo transcurrido en nuestro microcontrolador, lo almacena con valores en formato de decimales codificados en binario (**BCD**- *Binary Coded Decimal*) en un registro de 32 bits.

El formato BCD, guarda en formato de 4 bits en binario, los datos de cada uno de los dígitos del valor a representar; no al total del número en sí, sino de cada dígito por separado, como en la Figura 15.5.

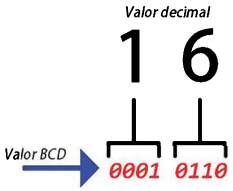


Figura 15.5

Para convertir, por ejemplo, el decimal 16 a BCD según lo explicado anteriormente; tenemos que tomar cada dígito decimal y transformarlo a su equivalente BCD según la tabla siguiente:

	Valor BCD	
DECIMAL	8 4 2 1	Dígito BCD
0	0 0 0 0	
1	0 0 0 1	
2	0 0 1 0	
3	0 0 1 1	
4	0 1 0 0	
5	0 1 0 1	
6	0 1 1 0	
7	0 1 1 1	
8	1 0 0 0	
9	1 0 0 1	

Figura 15.6

Crearemos una función con la que podremos realizar este proceso de pasar los datos de horas, minutos y segundos a BCD y convertirlos a segundos, que es lo que cargamos en nuestro RTC.

```
/*     Funcion que carga la hora nueva en el RTC        */
//-------------------------------------------------------
Void Inicializa_Hora(uint8_t Hora,uint8_t Minutos,uint8_t Segundos)
{
  // Creamos la variable que contendrá la suma total
  uint32_t CounterValue;

  // Deshabilitar la Interrupcio RTC por Segundos
  RTC_ITConfig(RTC_IT_SEC, DISABLE);

  // Habilitamos el acceso a los registros Backup
  PWR_BackupAccessCmd(ENABLE);

  // Convertimos Horas+Minutos+Segundos a segundos totales
  CounterValue=((Hora * 3600) + (Minutos * 60) + Segundos);

  // Grabamos el horario nuevo en los registros RTC
  RTC_SetCounter(CounterValue);

  // Esperamos a que se termine el acceso a los registros
  RTC_WaitForLastTask();

  // Rehabilitar la Interrupcio RTC por Segundos
  RTC_ITConfig(RTC_IT_SEC, ENABLE);
}
```

Figura 15.7

En la función que realiza este proceso, observamos que es muy fácil que nuestro entorno de trabajo realice la conversión, él mismo, de los valores a BCD.

Primero, al crear en la función **Inicializa_Hora**(10, 22, 00), tres valores donde pasarle el tiempo en 10 horas, 22 minutos y 00 segundos; que serán convertidos en su interior en un valor total a segundos, y poder así pasarlo a la variable de 32 bits '**CounterValue**', y cargarlos en el registro contador con el comando **RTC_SetCounter**(CounterValue).

Será necesario deshabilitar primero la interrupción y el bloqueo de acceso a los registros, mediante los comandos **RTC_ITConfig(RTC_IT_SEC**, *DISABLE*) y el comando **PWR_BackupAccessCmd**(ENABLE); para una vez cargado el valor en el registro, volver a habilitar estas opciones.

La otra función que necesitaremos a continuación, es la que lee el valor actual del contador RTC y convertirla a un valor de hora, minutos y segundos para poder reflejarla.

```
1  /*        Funcion que Lee la Hora en el RTC                    */
2  //-------------------------------------------  ---------------
3  void RTC_LeeHora(void)
4  {
5   uint32_t TimeValor;
6
7   TimeValor = RTC_GetCounter();  // Leemos contador del RTC
8
9   // Muestra la hora obtenida
10  Muestra_Hora(TimeValor);
11 }
12
13 /*  Funcion que extrae la hora, minutos y segundos
14     del valor obtenido                                          */
15 //-------------------------------------------------------------
16 void Muestra_Hora(uint32_t TimeValor)
17 {
18   uint8_t THor_High, THor_Low, TMin_High, TMin_Low,
19           TSeg_High, TSeg_Low;
20   uint8_t THor, TMin, TSeg;
21
22   // Extraemos los valores
23   THor_High = (uint8_t)(TimeValor/3600) / 10;
24   THor_Low  = (uint8_t)(TimeValor/3600) % 10;
25   TMin_High = (uint8_t)((TimeValor%3600)/60) /10;
26   TMin_Low  = (uint8_t)((TimeValor%3600)/60) %10;
27   TSeg_High = (uint8_t)((TimeValor%3600)%60) /10;
28   TSeg_Low  = (uint8_t)((TimeValor%3600)%60) %10;
29
30   // Pasamos los valores de NibbleHIGH y NibbleLOW
31   // a su valor Decimal
32   THor = (THor_High*10) + THor_Low;
33   TMin = (TMin_High*10) + TMin_Low;
34   TSeg = (TSeg_High*10) + TSeg_Low;
35
36   // Imprimimos los valores obtenidos por USART1
37   printf("\r\n  HORA RTC:  %d:%d:%d\r\n" ,THor, TMin, TSeg);
38 }
```

Figura 15.8

Estructuramos nuestro ejemplo en dos módulos o funciones separadas: en la primera "**RTC_LeeHora**()", creamos una variable con formato uint32_t que llamamos "**TimeValor**", en el que guardamos el valor obtenido del contador RTC mediante el comando "**RTC_GetCounter**()" ; siendo pasado ese valor a la segunda función "**Muestra_Hora**", en la que hemos creado dos grupos de variables. Primero, vamos extrayendo del valor del contador RTC los valores BCD de los *nibbles* superior e inferior de los valores que se guardaron en el formato 'HHMMSS' pasados a un valor de segundos. Y, tras realizar la extracción, los pasamos a las variables finales de Thor, TMin y TSeg que contendrá la hora acumulada en los registros RTC para posteriormente enviarlo por el puerto USART1 hacia nuestro ordenador.

Otra función que será importante e imprescindible, es la que configura e inicializa la interrupción mediante la estructura NVIC; ya que nuestra configuración

RTC se basa en el contaje que este realizará con base 1 segundo; para lo que se usará la forma que se muestra en la Figura 15.9.

```
/*    Funcion que configura el NVIC          */
//-----------------------------------------------
void NVIC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  // Configuramos y habilitamos la interrupcion del RTC
  NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

Figura 15.9

Como hemos creado la interrupción "**RTC_IRQn"**, necesitaremos también la función de servicio que la detecte y que es imprescindible para que se incremente el contador del reloj.

```
/* Funcion que controla cuando se produce la IRQ del RTC     */
//-------------------------------------------------------------
void RTC_IRQHandler(void)
{
  // Comprobamos el estado de la Interrupcion RTC por Segundos
  if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
  {
    // Receteamos la bandera de la Interrupcio RTC por Segundos
    RTC_ClearITPendingBit(RTC_IT_SEC);

    // Esperamos a que se termine el acceso a los registros
    RTC_WaitForLastTask();

    //  Reseteamos el contador del RTC cuando se detecte
    // que se ha llegado a la hora 23:59:59
    if (RTC_GetCounter() == 0x00015180)
    {
      // Reinicia el contador a '0'
      RTC_SetCounter(0x0);
      // Esperamos a que se termine el acceso a los registros
      RTC_WaitForLastTask();
    }
  }
}
```

Figura 15.10

Mediante el comando "**RTC_GetITStatus**(RTC_IT_SEC)", comprobaremos el estado de la bandera de la interrupción por segundos "RTC_IT_SEC"; si se ha producido, realizándose un reseteo de dicha bandera para restablecerla y que pueda ser detectada de nuevo.

También hemos añadido unas líneas que controlan cuando nuestro contador del RTC llegue a 24 horas y se reinicie a '0'.

CONTROL DE LA MEMORIA DEL RTC

Como vimos al principio de este capítulo, nuestro módulo RTC utiliza para salvaguardar el contaje transcurrido, una serie de registros del módulo backup, que pueden ser hasta 42 registros que se enumeran del BKP_DR1 al BKP_DR42. Estos son registros de memoria, que conservan su valor después de que se apague la alimentación principal del microcontrolador. No son una memoria flash o EEPROM, son una memoria RAM común que funciona con alimentación o batería.

Cuando nuestro microcontrolador STM32 se inicia desde un reinicio, existen varias fuentes posibles que han producido ese reinicio y para indicarlo, existe un registro de 32 bits del módulo de reloj RCC, el RCC_CSR, que se carga con un valor para indicar a qué se ha debido el reinicio, conteniendo así banderas que indican el posible motivo.

- Bit31: **LPWRRSTF** – (*Low-power reset flag)*
  Establecido por hardware cuando ocurre un reinicio de administración de bajo consumo
- Bit30: **WWDGRSTF** – (*Window watchdog reset flag*)
  Establecido por hardware cuando se produce un reinicio de vigilancia de ventana.
- Bit29: **IWDGRSTF** – (*Independent watchdog reset flag*)
  Establecido por el hardware cuando se produce un reinicio del perro guardián independiente del dominio VDD.
- Bit28: **SFTRSTF** – (*Software reset flag*)
  Establecido por hardware cuando ocurre un restablecimiento de software
- Bit27: **PORRSTF** – (POR/PDR *reset flag*)
  Establecido por hardware cuando se produce un reinicio (POR-*Power_on reset*) / (PDR-*Power_down reset*).
- Bit26: **PINRSTF** – (PIN *reset flag*)
  Establecido por hardware cuando ocurre un reinicio desde el pin de reset externo NRST.
- Bit25: **BORRSTF** – (BOR *reset flag*)
  Establecido por hardware cuando ocurre un reinicio por un encendido (POR-*Power_on reset*) / (PDR-*Power_down reset*) o (BOR-*Brownout reset*) donde el sistema permanece bajo reinicio durante el tiempo en que los voltajes de suministro (VDD y VDDIO) se estabilizan.

Figura 15.11

En nuestro código de ejemplo, al principio del módulo principal "*main*", podemos introducir unas líneas que nos permitan controlar, en caso de reinicio del sistema, si se ha perdido completamente la información de nuestro reloj y por ello, necesita reiniciarse de nuevo o solo se ha producido un reinicio de nuestro

microcontrolador, sin que haya existido pérdida de la alimentación y por tanto, de los datos almacenados en los registros RTC.

Mediante la lectura del primer registro de la memoria backup, la BKP_DR1, podemos saber si los registros del RTC permanecen configurados y con algún valor o, están en mal estado o borrados, para ello usaremos la línea de código siguiente:

```
// Comprobamos si los registros Backup no están configurados
 if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)
```

Figura 15.12

EL valor '0xA5A5', es un valor aleatorio que podemos utilizar para saber si existen valores en este registro.

Si no existe ese valor, entonces sabemos que no están los valores correctos del contador del RTC y por tanto, se necesita reiniciar el RTC con los valores de tiempo de comienzo. Y si existe el valor, es que el RTC sigue corriendo y su contador permanece activo.

Para ello, las siguientes líneas de nuestro código sirven para detectar el motivo por el que nuestro microcontrolador se ha reiniciado. Figura 15.13.

```
1    /* Check if the Power On Reset flag is set */
2  if (RCC_GetFlagStatus(RCC_FLAG_PORRST) != RESET)
3  {
4      printf(" Un Reinicio a ocurrido....\r\n\");
5  }
6    /* Check if the Pin Reset flag is set */
7  else if (RCC_GetFlagStatus(RCC_FLAG_PINRST) != RESET)
8  {
9      printf(" Un Reset externo se a detectado....\r\n");
10 }
```

Figura 15.13

Existe un registro de 32 bits del módulo RCC, el RCC_CSR (RCC *clock control & status register*), que permite saber por qué se ha debido el reinicio de nuestro microprocesador, estos se muestran en el listado siguiente:

- Bit 31: **LPWRRSTF** – (*Low-power reset flag*)
  Establecido por hardware cuando ocurre un reinicio de administración de bajo consumo
- Bit 30: **WWDGRSTF** – (*Window watchdog reset flag*)
  Establecido por hardware cuando se produce un reinicio de vigilancia de ventana.
- Bit 29: **IWDGRSTF** – (*Independent watchdog reset flag*)
  Establecido por el hardware cuando se produce un reinicio del perro guardián independiente del dominio VDD.
- Bit 28: **SFTRSTF** – (*Software reset flag*)
  Establecido por hardware cuando ocurre un restablecimiento de software
- Bit 27: **PORRSTF** – (*POR/PDR reset flag*)
  Establecido por hardware cuando se produce un reinicio (POR-*Power_on reset*) / (PDR-*Power_down reset*).
- Bit 26: **PINRSTF** – (*PIN reset flag*)
  Establecido por hardware cuando ocurre un reinicio desde el pin de reset externo NRST.
- Bit 25: **BORRSTF** – (*BOR reset flag*)
  Establecido por hardware cuando ocurre un reinicio por un encendido (POR-*Power_on reset*) / (PDR-*Power_down reset*) o (BOR-*Brownout reset*) donde el sistema permanece bajo reinicio durante el tiempo en que los voltajes de suministro (VDD y VDDIO) se estabilizan.

Figura 15.14

Hay que tener en cuenta que el estado de estas banderas se mantiene a pesar de un nuevo reinicio; donde por ejemplo, si este se ha producido por un encendido y después se pulsa el botón de reinicio, la bandera se mantendrá a pesar de que hayamos reiniciado ya que será la que registra el hecho de que hubo un reinicio por un encendido. Por ello se debe restablecer el indicador una vez que se ha utilizado mediante el comando "**RCC_ClearFlag()**".

Es importante también, que para que se mantenga el contador acumulado que llevaba el RTC hasta el momento en que se ha producido el reinicio, se evite que dicho contador sea reiniciado con el empleo del comando "RTC_DeInit()". Por lo que hemos añadido en nuestro código de ejemplo, la variable "**RTC_Estado**" que se inicia a valor '0' cuando es la primera vez que se inicia el sistema, pero que, cuando se ha producido un reinicio y se detecta que los registros del RTC todavía están operativos y con un acumulado de un contaje anterior, esta variable pasa a valor '1' para que en la función RTC_Init() no se produzca el restablecimiento del RTC.

15.1 EJEMPLO DE CONTROL DE HORARIO CON EL RTC

Incluimos a continuación todo el código del ejemplo:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rtc.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_bkp.h"
#include "stm32f10x_usart.h"

#include <stdio.h>    // Funcion que permite usar el 'printf'

//  Lista de funciones ------------------------
void RTC_Init(void);
void NVIC_Configuration(void);
void Inicializa_Hora(uint8_t Horas, uint8_t Minutos, uint8_t Segundos);
void RTC_LeeHora(void);
void Muestra_Hora(uint32_t TimeValor);
void USART1_Config(void);
void USARTSend(char *pucBuffer);

/*       Permite configurar el reloj LCE o LSI       */
//-----------------------------------------------------
//#define RTCClockSource_LSE  // Externo
#define RTCClockSource_LSI    //(*) Interno OK

uint32_t RTC_CountSecond = 0; // Variable donde se guarda seg. transcurridos

static int RTC_Estado = 0;   // Creamos la variable de estado del RTC


/*       Modulo Principal                            */
//=====================================================
int main(void)
{
  USART1_Config();     // Inicializamos el USART1

  NVIC_Configuration(); // Inicializamos el NVIC para IRQ del RTC

  printf("\r\n================================================\r\n");
  printf("      PROGRAMA DEMO DE UTILIZACION DEL RTC  \r\n");
  printf("================================================\r\n");

  // Comprobamos si los registros Bakcup no estan configurados
  if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)
  {
     printf("\r\n  RTC NO CONFIGURADO....\r\n");

     printf("\r\n  Reiniciando RTC, espere....\r\n");
     RTC_Init();   // Inicializamos el RTC

     printf("\r\n  Configurando RTC, espere....\r\n");
     // Inicializamos Horas, Minutos y Segundos
     Inicializa_Hora(14, 23, 00);

     printf("\r\n  --- RTC CONFIGURADO --- !!! \r\n");

     // Cargamos un valor en el registro 1 del Backup
     BKP_WriteBackupRegister(BKP_DR1, 0xA5A5);
  }
  else
  {
```

```
60       /* Comprobamos si el reinicio es POR o PDR */
61
62     if (RCC_GetFlagStatus(RCC_FLAG_PORRST) != RESET)
63     {
64       printf("\r\n  Un Reinicio a ocurrido....\r\n\r\n");
65     }
66     /* Comprobamos si el reinicio es por pulsar el pin NRST */
67     else if (RCC_GetFlagStatus(RCC_FLAG_PINRST) != RESET)
68     {
69       printf("\r\n  Un Reset externo se a detectado....\r\n\r\n");
70     }
71
72     printf("  NO es necesario configurar de nuevo el RTC....\r\n",;
73     printf("  ... Iniciando el RTC, espere ...\r\n");
74
75     // Asignamos '1' para indicar un reinicio sin reseteo del RTC
76     RTC_Estado = 1;
77     RTC_Init();   // Inicializamos el RTC
78
79     /* Clear reset flags */
80     RCC_ClearFlag();
81   }
82
83   while(1)
84   {
85       // Muestra la hora cada 20 segundos
86       if (RTC_GetCounter() - RTC_CountSecond > 20U)
87       {
88       printf("\r\n  Total segundos: %d", (int)RTC_CountSecond);
89       RTC_LeeHora();
90
91       // Restablece el valor del contador
92       RTC_CountSecond = RTC_GetCounter();
93       }
94   }
95 }
96
97 /*   Funcion que configura el RTC              */
98 //-----------------------------------------------
99 void RTC_Init(void)
100 {
101   /* Iniciamos el reloj APB1 para PWR y BKP    */
102   RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR |
103                          RCC_APB1Periph_BKP, ENABLE);
104
105   /* Permitir acceso a los registros BKP */
106   PWR_BackupAccessCmd(ENABLE);
107
108   /* Comprobamos si es un reinicio de un nuevo RTC  */
109   if (RTC_Estado != 1)
110   {
111     printf("  Reinicio del RTC \r\n");
112     /* Reseteo de los registros Backup  */
113     BKP_DeInit();
114     // Reseteando registros backup del RTC
115     RCC_BackupResetCmd(ENABLE);
116     RCC_BackupResetCmd(DISABLE);
117     printf("    - Reseteando backup de registros RTC\r\n");
118   }
119
120   ////////////////////////////////////////////////
121   // ----   Si se habilita el  reloj LSI    -------
122   #ifdef RTCClockSource_LSI
123   ////////////////////////////////////////////////
124   /* Habilitar reloj LSI */
```

```
125  RCC_LSICmd(ENABLE);
126  printf("    - Habilitando fuente LSI \r\n");
127  //
128  /* Espere hasta que el LSI esté listo */
129  while(RCC_GetFlagStatus(RCC_FLAG_LSIRDY)==RESET)
130  {;}
131  //
132  /* Seleccionamos LSI como fuente de reloj del RTC */
133  RCC_RTCCLKConfig(RCC_RTCCLKSource_LSI);
134  printf("    - Habilitando LSI en RTCCLK \r\n");
135  ////////////////////////////////////////////////
136  // ----  Si se habilita el reloj LSE     --------
137  #elif defined RTCClockSource_LSE
138  ////////////////////////////////////////////////
139  /* Habilitar reloj LSE */
140  RCC_LSEConfig(RCC_LSE_ON);
141  //
142  // Deshabilitamos el reloj LSI
143  RCC_LSICmd(DISABLE);
144  printf("    - Deshabilitando LSI \r\n");
145  //
146  /* Espere hasta que el LSE esté listo */
147  while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
148  {;}
149  //
150  /* Seleccionamos LSE como fuente de reloj del RTC */
151  RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);
152  printf("    - Habilitando LSE para RCC_RCT \r\n");
153  #endif
154  ////////////////////////////////////////////////
155
156  /*  Habilita el reloj RTC              */
157  RCC_RTCCLKCmd(ENABLE);
158  printf("    - Habilitando RTCCLK\r\n");
159
160  /* Espera a que se sincronice los registros RTC  */
161  printf("    - Esperando sincronizacion \r\n");
162  RTC_WaitForSynchro();
163
164  /* Espera hasta que finalice última operación de escritura en registros "T
165  RTC_WaitForLastTask();
166
167  /* Habilitar la Interrupcio RTC por Segundos */
168  RTC_ITConfig(RTC_IT_SEC, ENABLE);
169  printf("    - Habilitando segundo RTC \r\n");
170
171  // Establecer el periodo del contador RTC en 1 segundo
172  // Para LSE = RTCCLK/RTC period = 32768Hz/1Hz = 32767+1
173  // para LSI = RTCCLK/RTC period = 40000Hz/1Hz = 39999+1
174  ////////////////////////////////////////////////
175   #ifdef RTCClockSource_LSI
176  ////////////////////////////////////////////////
177  /* Establecemos el Preescaler del RTC para un período de 1 segundo */
178  /* Peridodo RTC = RTCCLK/RTC_Prescal = (40 000 Hz)/39999+1) */
179  RTC_SetPrescaler(39999);
180  printf("    - Configurado Prescaler LSI = 40 kHz \r\n");
181
182  ////////////////////////////////////////////////
183  #elif defined  RTCClockSource_LSE
184  ////////////////////////////////////////////////
185  /* Establecemos el Preescaler del RTC para un período de 1 segundo */
186  /* Periodo RTC = RTCCLK/RTC_Prescal = (32.768KHz)/(32767+1) */
187  RTC_SetPrescaler(32767);
188  printf("    - Configurado Prescaler LSE = 32,768 kHz \r\n");
189  #endif
190
```

```
192   /* Espere hasta que finalice última operación de escritura en registros RT
193   RTC_WaitForLastTask();
194 }
195
196 /*       Funcion que Lee la Hora en el RTC    */
197 //-----------------------------------------------
198 void RTC_LeeHora(void)
199 {
201  uint32_t TimeValor;
202
203  TimeValor = RTC_GetCounter();  // Leemos contador del RTC
204  TimeValor = TimeValor % 86400;
205
206  // Muestra la hora obtenida
207  Muestra_Hora(TimeValor);
208 }
209
210 /*  Funcion que extrae la hora, minutos y segundos del valor obtenido */
211 //-------------------------------------------------------------------------
212 void Muestra_Hora(uint32_t TimeValor)
213 {
214   uint8_t THor_High, THor_Low, TMin_High, TMin_Low, TSeg_High, TSeg_Low;
215   uint8_t THor, TMin, TSeg;
216
217   // Extraemos los valores
218   THor_High = (uint8_t) (TimeValor/3600) / 10;
219   THor_Low  = (uint8_t) (TimeValor/3600) % 10;
220   TMin_High  = (uint8_t) ((TimeValor%3600)/60)/10;
221   TMin_Low   = (uint8_t) ((TimeValor%3600)/60)%10;
222   TSeg_High  = (uint8_t) ((TimeValor%3600) %60)/10;
223   TSeg_Low   = (uint8_t) ((TimeValor%3600) %60)%10;
224
225   // Pasamos los valores de NibbleHIGH y NibbleLOW a su valor Decimal
226   THor = (uint8_t) (THor_High*10) + THor_Low;
227   TMin = (uint8_t) (TMin_High*10) + TMin_Low;
228   TSeg = (uint8_t) (TSeg_High*10) + TSeg_Low;
229
230   // Imprimimos los valores obtenidos por USART1
231   printf("\r\n  HORA RTC:  %d:%d:%d\r\n" ,THor, TMin, TSeg);
232 }
233
234 /*     Funcion que carga la hora nueva en el RTC              */
235 //-----------------------------------------------------------------
236 void Inicializa_Hora(uint8_t Horas, uint8_t Minutos, uint8_t Segundos)
237 {
238   uint32_t CounterValue;
239   // Deshabilitar la Interrupcio RTC por Segundos
240   RTC_ITConfig(RTC_IT_SEC, DISABLE);
241   // Habilitamos el acceso a los registros BacKup
242   PWR_BackupAccessCmd(ENABLE);
243   // Convertimos segundos totales el horario nuevo
244   CounterValue=((Horas * 3600) + (Minutos * 60) + Segundos);
245
246   // Grabamos el horario nuevo en los registros RTC
247   RTC_SetCounter(CounterValue);
248
249   // Esperamos a que se termine el acceso a los registros
250   RTC_WaitForLastTask();
251
252   // Rehabilitar la Interrupcio RTC por Segundos */
253   RTC_ITConfig(RTC_IT_SEC, ENABLE);
254 }
255
256
```

```
/*    Funcion que configura el NVIC                    */
//-------------------------------------------------------
void NVIC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  // Configuramos y habilitamos la interrupcion del RTC
  NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}


/*          Funcion que configura el USART1                */
//-----------------------------------------------------------
void USART1_Config(void)
{
      /* Configuramos el GPIO para USART1 -----------------*/
      GPIO_InitTypeDef GPIO_InitStructure;

      /* Habilitamos los reloj para GPIO y el USART1     */
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA,
ENABLE);

      /* Configuramos PA9 para Tx USART1 ------------------*/
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_Init(GPIOA, &GPIO_InitStructure);

      /* Configuramos Rx para Rx USART1 -------------------*/
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
      GPIO_Init(GPIOA, &GPIO_InitStructure);

      /* Configuramos el USART1  --------------------------*/
      USART_InitTypeDef USART_InitStructure;
      USART_InitStructure.USART_BaudRate = 115200;
      USART_InitStructure.USART_WordLength = USART_WordLength_8b;
      USART_InitStructure.USART_StopBits = USART_StopBits_1;
      USART_InitStructure.USART_Parity = USART_Parity_No;
      USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;
      USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
      USART_Init(USART1, &USART_InitStructure);
      USART_Cmd(USART1, ENABLE);
}

// Funncion que nos permite utilizar el comando 'printf'
int fputc(int ch, FILE *f)
{
      /* Envia dato */
      USART_SendData(USART1, (unsigned char) ch);

```

Figura 15.15

15.2 EJEMPLO DE CONFIGURACIÓN DE UNA ALARMA CON EL RTC

En nuestro siguiente ejemplo, configuramos la opción de alarma del RTC, para establecer una interrupción que se genere cada segundo.

La configuración es prácticamente igual a la de nuestro primer ejemplo, necesitando tan solo añadir algunos elementos a nuestra programación anterior.

```
  void RTC_Init(void)
  {
1   /* Iniciamos el reloj APB1 para PWR y BKP    */
     . . .
2   /* Permitir acceso a los registros BKP */
     . . .
3   /* Resetea de los registros Backup  */
     . . .
4   /* Habilitar reloj LSI */
     . . .
5   /* Espere hasta que el LSI esté listo */
     . . .
6   /* Seleccionamos LSI como fuente de reloj del RTC */
     . . .
7   /*  Habilita el reloj RTC             */
     . . .
8   /* Espera hasta que finalice la última operación en RTC */
     . . .
9   /* Espera a que se sincronice los registros RTC  */
     . . .
10  /* Espera hasta que finalice la última operación en RTC */
     . . .
11  /* Habilitar la Interrupcio RTC por Segundos */
     . . .
12  /* Habilitar la Interrupcion ALARMA del RTC   */
    RTC_ITConfig(RTC_IT_ALR, ENABLE);
13  /* Establecemos el Preescaler del RTC para período de 1 seg */
     . . .
14  /* Espere hasta que finalice la última operación en RTC */
  }
```

Figura 15.16

La función de inicio y configuración del RTC es casi igual, solo tenemos que añadir una línea habilitando también la interrupción de alarma del RTC, como vemos en la Figura 15.16.

Además, en el módulo que configura el **NVIC** de control de interrupciones, necesitamos crear también la "**RTCAlarm_IRQn**" correspondiente.

```
1   void NVIC_Configuration(void)
2   {
3     NVIC_InitTypeDef NVIC_InitStructure;
4     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
5      . . .
6
7     /* Configuramos la Interrupcion IRQ de la Alarma     */
8     NVIC_InitStructure.NVIC_IRQChannel = RTCAlarm_IRQn;
9     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 15;
10    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
11    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
12    NVIC_Init(&NVIC_InitStructure);}
13     . . .
14     . . .
```

Figura 15.17

También debemos añadir una función de servicio para controlar cuando se produzca la interrupción, y que al mismo tiempo compruebe el valor de la bandera "**RTC_IT_ALR**", como se muestra en la siguiente Figura 15.18.

```
1   void RTC_IRQHandler(void)
2   {
3     . . .
4     // Comprobamos si se ha producido la IRQ de la Alarma
5     if(RTC_GetITStatus(RTC_IT_ALR) != RESET)
6       {
7         // Borramos la bandera para volver a utilizarla
8         RTC_ClearITPendingBit(RTC_IT_ALR);
9         . . .
10        printf("  ALARMA ACTIVADA !!!!!!!\r\n");
11
12        // Repetimos la alarma a los 3 minutos
13        RTC_SetAlarm(RTC_GetCounter()+180);
14        RTC_WaitForLastTask();
15      }
16    . . .
17  }
```

Figura 15.18

Necesitaremos establecer, el horario de la alarma; para lo que podemos crear una función que además de convertir los valores de HH, MM, SS de la alarma, se establezca en el registro correspondiente con el comando "**RTC_SetAlarm**(*TimeValor*)", tal como se muestra en la Figura 15.19.

```
1   /*  Función que establece la hora de la Alarma en el RTC  */
2   //-------------------------------------------------------------
3   void Establece_Alarma(int8_t Horas,uint8_t Minutos,uint8_t Segundos)
4   {
5     uint32_t TimeValor;
6
7     // Convertimos el horario indicado a segundos totales
8     TimeValor=((Horas * 3600) + (Minutos * 60) + Segundos);
9
10    // Establecemos la Hora de la Alarma en el RTC
11    RTC_SetAlarm(TimeValor);
12    RTC_WaitForLastTask();
```

Figura 15.19

A continuación se presenta el código completo del proyecto del ejemplo de Alarma RTC. Donde, además de la gestión que hemos explicado de la interrupción de una alarma para nuestro RTC, añadimos la gestión de la pulsación de un botón controlado con la interrupción EXT_5 que permite deshabilitar la alarma.

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rtc.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_bkp.h"
#include "stm32f10x_exti.h"

#include "stm32f10x_usart.h"
#include "misc.h"

#include <stdio.h>    // Funcion que permite usar el 'printf'

//  Lista de funciones ------------------------
void RTC_Init(void);
void NVIC_Configuration(void);
void Inicializa_Hora(uint8_t Horas,uint8_t Minutos,uint8_t Segundos);
void RTC_LeeHora(void);
void Muestra_Hora(uint32_t TimeValor);
void Establece_Alarma(int8_t Horas,uint8_t Minutos,uint8_t Segundos)   ;
void EXT5_Config(void);
void RTC_IRQHandler(void);
void EXTI9_5_IRQHandler(void) ;
void USART1_Config(void);
void USARTSend(char *pucBuffer);

/*       Permite configurar el reloj LCE o LSI       */
//-----------------------------------------------------
//  Selecciones una de las dos líneas
//-----------------------------------------------------
//#define RTCClockSource_LSE  // Externo
#define RTCClockSource_LSI //(*) Interno OK

uint32_t RTC_CountSecond = 0; // Variable donde se guarda seg. transcu...

static int RTC_Estado = 0;   // Creamos la variable estado del RTC
static int ALARMA_Estado = 1;// Variable que indica Alarma activada o no

/*       Modulo Principal                            */
//=====================================================
int main(void)
{
  USART1_Config();      // Inicializamos el USART1
  NVIC_Configuration(); // Inicializamos el NVIC para IRQ del RTC
  EXT5_Config();        // Inicializa la interrupcion del Boton

  printf("\r\n=====================================================\r\n");
  printf("  PROGRAMA DEMO DE UTILIZACION DE LA ALARMA DEL RTC  \r\n");
  printf("=====================================================\r\n");

  // Comprobamos si los registros Bakcup no estan configurados
  if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)
  {
     printf("\r\n  RTC NO CONFIGURADO....\r\n");

     printf("\r\n  Reiniciando RTC, espere....\r\n");
     RTC_Init();   // Inicializamos el RTC

     printf("\r\n  Configurando RTC, espere....\r\n");
     // Inicializamos Horas, Minutos y Segundos

```

```
61        Inicializa_Hora(15, 00, 00);
62
63        // Establece la Alarma
64        Establece_Alarma(15, 02, 00);
65
66        printf ("\r\n  --- RTC CONFIGURADO --- !!! \r\n");
67
68        // Cargamos un valor en el registro 1 del Backup
69        BKP_WriteBackupRegister(BKP_DR1, 0xA5A5);
70    }
71     else
72    {
73      /* Comprobamos si el reinicio es POR o PDR */
74      if (RCC_GetFlagStatus(RCC_FLAG_PORRST) != RESET)
75      {
76        printf("\r\n  Un Reinicio a ocurrido....\r\n\r\n");
77      }
78      /* Comprobamos si el reinicio es por pulsar el pin NRST */
79      else if (RCC_GetFlagStatus(RCC_FLAG_PINRST) != RESET)
80      {
81        printf("\r\n  Un Reset externo se a detectado....\r\n\r\n");
82      }
83
84    printf("  NO es necesario configurar de nuevo el RTC....\r\n");
85    printf("  ... Iniciando el RTC, espere ...\r\n");
86
87    // Asignamos '1' para indicar un reinicio sin reseteo del RTC
88    RTC_Estado = 1;
89    RTC_Init();      // Inicializamos el RTC
90)
91    /* Clear reset flags */
92        RCC_ClearFlag();
93    }
94
95    while(1)
96    {
97        // Muestra la hora cada 20 segundos
98        if (RTC_GetCounter() - RTC_CountSecond > 10U)
99        {
100       printf("\r\n  Total segundos: %d", (int)RTC_CountSecond);
101       RTC_LeeHora();
102
103       // Restablece el valor del contador
104       RTC_CountSecond = RTC_GetCounter();
105       }
106   }
107 }
108
109 /*   Funcion que configura el RTC                    */
110 //-----------------------------------------------------
111 void RTC_Init(void)
112 {
113   /* Iniciamos el reloj APB1 para PWR y BKP    */
114   RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR |
115                          RCC_APB1Periph_BKP, ENABLE);
116   /* Permitir acceso a los registros BKP */
117   PWR_BackupAccessCmd(ENABLE);
118   /* Comprobamos si es un reinicio de un nuevo RTC  */
119   if (RTC_Estado != 1)
120   {
121     printf("  Reinicio del RTC \r\n");
122     /* Reseteo de los registros Backup  */
123     BKP_DeInit();
124     // Reseteando registros backup del RTC
125     RCC_BackupResetCmd(ENABLE);
126     RCC_BackupResetCmd(DISABLE);
```

```
127     printf("    - Reseteando backup de registros RTC\r\n");
128   }
129
130   ////////////////////////////////////////////////
131   // ----   Si se habilita el  reloj LSI    -------
132   #ifdef RTCClockSource_LSI
133   ////////////////////////////////////////////////
134   /* Habilitar reloj LSI */
135   RCC_LSICmd(ENABLE);
136   printf("    - Habilitando fuente LSI \r\n");
137   //
138   /* Espere hasta que el LSI esté listo */
139   while(RCC_GetFlagStatus(RCC_FLAG_LSIRDY)==RESET)
140   {;}
141   //
142   /* Seleccionamos LSI como fuente de reloj del RTC */
143   RCC_RTCCLKConfig(RCC_RTCCLKSource_LSI);
144   printf("    - Habilitando LSI en RTCCLK \r\n");
145   ////////////////////////////////////////////////
146   // ----  Si se habilita el reloj LSE     --------
147   #elif defined RTCClockSource_LSE
148   ////////////////////////////////////////////////
149   /* Habilitar reloj LSE */
150   RCC_LSEConfig(RCC_LSE_ON);
151   printf("    - Habilitando fuente LSE \r\n");
152   //
153   /* Espere hasta que el LSE esté listo */
154   while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
155   {;}
156   // Deshabilitamos el reloj Lsi
157   RCC_LSICmd(DISABLE);
158   printf("    - Deshabilitando LSI \r\n");
159   //
160   /* Seleccionamos LSE como fuente de reloj del RTC */
161   RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);
162   printf("    - Habilitando LSE para RTCCLK \r\n");
163   #endif
164   ////////////////////////////////////////////////
165   /*  Habilita el reloj RTC             */
166   RCC_RTCCLKCmd(ENABLE);
167   printf("    - Habilitando RTCCLK\r\n");
168   /* Espera hasta que finalice la última operación en registros RTC */
169   RTC_WaitForLastTask();
170   /* Espera a que se sincronice los registros RTC  */
171   printf("    - Esperando sincronizacion \r\n");
172   RTC_WaitForSynchro();
173   /* Espera hasta que finalice la última operación en registros RTC */
174   RTC_WaitForLastTask();
178   /* Habilitar la Interrupcio RTC por Segundos */
176   RTC_ITConfig(RTC_IT_SEC, ENABLE);
177   printf("    - Habilitando Interrupcion general RTC \r\n");
178
179   /* Habilitar la Interrupcion ALARMA del RTC   */
180   RTC_ITConfig(RTC_IT_ALR, ENABLE);
181   printf("    - Habilitando Interrupcion de Alarma RTC \r\n");
182
186   /* Espera hasta que finalice la última operación en registros RTC */
187   RTC_WaitForLastTask();
188   ////////////////////////////////////////////////////////////////////
189   // Establecer el periodo del contador RTC en 1 segundo
190   // Para LSE = RTCCLK/RTC period = 32768Hz/1Hz = 32767+1
191   // para LSI = RTCCLK/RTC period = 40000Hz/1Hz = 39999+1
192   ////////////////////////////////////////////////////////////////////
193   #ifdef RTCClockSource_LSI
194   ////////////////////////////////////////////////////////////////////
195   /* Establecemos el Preescaler del RTC para un período de 1 segundo */
```

```
  /* Peridodo RTC = RTCCLK/RTC_Prescal = (40 000 Hz)/39999+1) */
  RTC_SetPrescaler(39999);
  printf("    - Configurado Prescaler LSI (40 kHz) \r\n");
  ////////////////////////////////////////////////////////////////////
  #elif defined  RTCClockSource_LSE
  ////////////////////////////////////////////////////////////////////
  /* Establecemos el Preescaler del RTC para un período de 1 segundo */
  /* Periodo RTC = RTCCLK/RTC_Prescal = (32.768KHz)/(32767+1) */
  RTC_SetPrescaler(32767);
  printf("    - Configurado Prescaler LSE (32,768 kHz) \r\n");
  #endif
  ////////////////////////////////////////////////////////////////////

  /* Espere hasta que finalice la última operación en registros RTC */
  RTC_WaitForLastTask();
}

/*       Funcion que Lee la Hora en el RTC           */
//-----------------------------------------------------
void RTC_LeeHora(void)
{
  uint32_t TimeValor;
  TimeValor = RTC_GetCounter();  // Leemos contador del RTC
  // Muestra la hora obtenida
  Muestra_Hora(TimeValor);
}

/*  Funcion que extrae la hora, minutos y segundos del valor obtenido */
//-------------------------------------------------------------------------
void Muestra_Hora(uint32_t TimeValor)
{
  uint8_t THor_High, THor_Low, TMin_High, TMin_Low, TSeg_High, TSeg_Low;
  uint8_t THor, TMin, TSeg;
  // Extraemos los valores
  THor_High = (uint8_t) (TimeValor/3600) / 10;
  THor_Low  = (uint8_t) (TimeValor/3600) % 10;
  TMin_High  = (uint8_t) ((TimeValor%3600)/60)/10;
  TMin_Low   = (uint8_t) ((TimeValor%3600)/60)%10;
  TSeg_High  = (uint8_t) ((TimeValor%3600) %60)/10;
  TSeg_Low   = (uint8_t) ((TimeValor%3600) %60)%10;
  // Pasamos los valores de NibbleHIGH y NibbleLOW a su valor Decimal
  THor = (THor_High*10) + THor_Low;
  TMin = (TMin_High*10) + TMin_Low;
  TSeg = (TSeg_High*10) + TSeg_Low;
  // Imprimimos los valores obtenidos por USART1
  printf("\r\n  HORA RTC:  %d:%d:%d " ,THor, TMin, TSeg);
  // Indica si la alarma esta activada
  if (ALARMA_Estado == 1)
  {
    printf (" (A) \r\n");
  }  else  {
    printf("\r\n");
  }
}

/*     Funcion que carga la hora nueva en el RTC                */
//--------------------------------------------------------------------
void Inicializa_Hora(uint8_t Horas,uint8_t Minutos,uint8_t Segundos)
{
  uint32_t CounterValue;
  // Deshabilitar la Interrupcio RTC por Segundos
  RTC_ITConfig(RTC_IT_SEC, DISABLE);
  // Habilitamos el acceso a los registros BacKup
  PWR_BackupAccessCmd(ENABLE);
  //RTC_WaitForLastTask();
  // Convertimos segundos totales el horario nuevo
```

```
262   CounterValue=((Horas * 3600) + (Minutos * 60) + Segundos);
263   // Grabamos el horario nuevo en los registros RTC
264   RTC_SetCounter(CounterValue);
265   // Esperamos a que se termine el acceso a los registros
266   RTC_WaitForLastTask();
267   // Rehabilitar la Interrupcio RTC por Segundos */
268   RTC_ITConfig(RTC_IT_SEC, ENABLE);
269 }
270
271 /*    Funcion que configura NVIC para IRQ del RTC      */
272 //-------------------------------------------------------
273 void NVIC_Configuration(void)
274 {
275   NVIC_InitTypeDef NVIC_InitStructure;
276   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
277   /* Configuramos la interrupcion general del RTC      */
278   NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQn;
279   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
280   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
281   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
282   NVIC_Init(&NVIC_InitStructure);
283
284   /* Configuramos la Interrupcion IRQ de la Alarma     */
285   NVIC_InitStructure.NVIC_IRQChannel = RTCAlarm_IRQn;
286   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 15;
287   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
288   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
289   NVIC_Init(&NVIC_InitStructure);
290
291   /* Condiguracion NVIC para EXT5 canal 5 IRQ en PA5   */
292   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
293   NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;
294   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
295   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
296   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
297   NVIC_Init(&NVIC_InitStructure);
298 }
299
300 /*   Funcion que configura la interrupcion EXT5_IRQ  */
301 //-------------------------------------------------------
302 void EXT5_Config(void){
303
304   // Creamos la estructrua para GPIO Boton en PA5
305   GPIO_InitTypeDef GPIO_InitStructure;
306   // Activamos el reloj para GPIO en puerto A
307   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
308
309   // Configura pin PA5 como entrada del boton
310   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
311   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
312   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
313   GPIO_Init(GPIOA, &GPIO_InitStructure);
314
315   // Creamos la estructura para EXTI
316   EXTI_InitTypeDef EXTI_InitStructure;
317   //Configura la interrupcion EXTI5 en linea 5 de PA5
318   EXTI_InitStructure.EXTI_Line = EXTI_Line5;
319   EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
320   EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
321   EXTI_InitStructure.EXTI_LineCmd = ENABLE;
322   EXTI_Init(&EXTI_InitStructure);
323 }
324
325 /*  Funcion que establece la hora de la Alarma en el RTC  */
326 //-------------------------------------------------------
327 void Establece_Alarma(int8_t Horas,uint8_t Minutos,uint8_t Segundos)
```

```
328 {
329   uint32_t TimeValor;
330
331   // Convertimos el horario indicado a segundos totales
332   TimeValor=((Horas * 3600) + (Minutos * 60) + Segundos);
333
334   // Establecemos la Hora de la Alarma en el RTC
335   RTC_SetAlarm(TimeValor);
336   RTC_WaitForLastTask();
337 }
338
339
340 /*     Funcion que se genera cada vez que se produce la IRQ del RTC  */
341 //---------------------------------------------------------------------------
342 void RTC_IRQHandler(void)
343 {
344   // Comprobamos si se ha producido la IRQ general del RTC
345   if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
346   {
347     // Receteamos la bandera de la Interrupcio RTC por Segundos
348     RTC_ClearITPendingBit(RTC_IT_SEC);
349
350     // Esperamos a que se termine el acceso a los registros
351     RTC_WaitForLastTask();
352
353     //  Reseteamos el contador del RTC cuando se detecte la hora 23:59:59
354     if (RTC_GetCounter() == 0x00015180)
355     {
356       // Reinicia el contador a '0'
357       RTC_SetCounter(0x0);
358       // Esperamos a que se termine el acceso a los registros
359       RTC_WaitForLastTask();
360     }
361   }
362
363   // Comprobamos si se ha producido la IRQ de la Alarma
364   if(RTC_GetITStatus(RTC_IT_ALR) != RESET)
365     {
366       RTC_ClearITPendingBit(RTC_IT_ALR);
367       RTC_LeeHora();
368       printf("  ALARMA ACTIVADA !!!!!!!\r\n");
369
370       // Repetimos la alarma a los 3 minutos
371       RTC_SetAlarm(RTC_GetCounter()+180);
372       RTC_WaitForLastTask();
373     }
374 }
375
376 /*    Funcion que controla si se pulsa el boton      */
377 //-----------------------------------------------------
378 void EXTI9_5_IRQHandler(void)
379 {
380       // Comprueba si se ha producido la interrupcion
381       if(EXTI_GetITStatus(EXTI_Line5) != RESET)
382       {
383          // Desactiva la interrupcion para que se vuelva a comprobar
384          EXTI_ClearITPendingBit(EXTI_Line5);
385
386          // Reseteamos la hora de la alarma
387          RTC_SetAlarm(0x0);
388          RTC_WaitForLastTask();
389
390         RTC_ITConfig(RTC_IT_ALR, DISABLE);
391         printf("  ALARMA DESABILITADA !!!!!! \r\n");
392         ALARMA_Estado = 0;  // Alarma desactivada
393     }
```

```
394 }
395
396 /*           Funcion que configura el USART1           */
397 //-------------------------------------------------------
398 void USART1_Config(void)
399 {
400     /* Configuramos el GPIO para USART1 -----------------*/
401     GPIO_InitTypeDef GPIO_InitStructure;
402
403     /* Habilitamos los reloj para GPIO y el USART1     */
404     RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2P
405
406     /* Configuramos PA9 para Tx USART1 ------------------*/
407     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
408     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
409     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
410     GPIO_Init(GPIOA, &GPIO_InitStructure);
411
412     /* Configuramos Rx para Rx USART1 -------------------*/
413     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
414     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
415     GPIO_Init(GPIOA, &GPIO_InitStructure);
416
417     /* Configuramos el USART1  --------------------------*/
418     USART_InitTypeDef USART_InitStructure;
419     USART_InitStructure.USART_BaudRate = 115200;
420     USART_InitStructure.USART_WordLength = USART_WordLength_
421     USART_InitStructure.USART_StopBits = USART_StopBits_1;
422     USART_InitStructure.USART_Parity = USART_Parity_No;
423     USART_InitStructure.USART_HardwareFlowControl =
424 USART_HardwareFlowControl_None;
425     USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_M
426     USART_Init(USART1, &USART_InitStructure);
427     USART_Cmd(USART1, ENABLE);
428 }
429
430 // Funncion que nos permite utilizar el comando 'printf'
431 int fputc(int ch, FILE *f)
432 {
433       /* Envia dato */
434       USART_SendData(USART1, (unsigned char) ch);
435
436       while( USART_GetFlagStatus(USART1,USART_FLAG_TC)!= SET)
437       return (ch);
438 }
439
```

Figura 15.20

15.3 EJEMPLO DE CONFIGURACIÓN DE UN CALENDARIO CON EL RTC

En nuestro siguiente ejemplo, configuramos nuestro reloj para que nos muestre además de la hora, un calendario en el que se calcule los días, meses y años. También se debe tener en cuenta: los años bisiestos; que un año es de 365,242190 días y además, que se debe agregar un día bisiesto aproximadamente una vez cada 4 años (4 × 0,242190 = 0,968760) -por tanto, cada cuatro años, el mes de febrero cuenta 29 días en lugar de 28-.

Utilizaremos fórmulas matemáticas ya establecidas para pasar toda la fecha del formato (HH-MM-SS-DD-MM-AAAA) a su valor total en segundos, que es lo que grabamos en el RTC.

Para ello, debemos realizar la conversión para que cada parámetro del tiempo inicial se convierta a segundos equivalentes. Por ejemplo: 3600 segundos equivalen a una hora y 86400 segundos equivalen a un día completo de 24 horas y, como dijimos antes, durante los cálculos, debemos tener en cuenta los años bisiestos o, de lo contrario, los cálculos de tiempo serán imperfectos después de algunos años.

Para pasar la fecha y la hora conocidas a un valor único de segundos, se utilizan las ecuaciones que existen para establecer y calcular los días del calendario Juliano.

Calendario Juliano... Un poco de historia...

Inicialmente el calendario que se utilizaba en la antigüedad contaba ya con 12 meses de 30 días; pero con el paso del tiempo, se observó que las fechas del mismo no coincidían con las estaciones del año, añadiéndosele 5 días más para intentar que cuadraran. Ante este problema, el emperador romano Julio Cesar, realizó en el año 45 A.C. una reforma global del calendario que se usaba hasta ese momento que se basaba en uno de origen egipcio. Estableció la duración del año en 365,25 días, insertando un día más en el mes de febrero cada cuatro años, haciendo bisiestos todos los años cuyo número sea divisible por 4....

Ahora un mucho de matemáticas...

Este calendario, el Juliano, cuenta los días a partir del día 1 de enero del año 4713 A.C. que tendrá un final ya establecido en el 31 de diciembre del 3267, que corresponde con el fin del primer periodo Juliano de 7980 años.

Por ello, para calcular el valor total de días del Calendario Juliano que representa nuestra fecha inicial, con la que queremos establecer nuestro reloj RTC, realizamos la siguiente conversión:

$$a = (14 - Mes) / 12 \;;\; y = Año + 4800 - a \;;\; m = (Mes + 12) * 4 - 3 \;;$$

$$JDN = \left[\left((153 * m + 2) / 5 \right) + (365 * y) + (y/4) - (y/100) + (y/400) \right) - 32045 \right]$$

Figura 15.21

El resultado "*JDN*" será el número de días totales del calendario Juliano que constituye nuestra fecha inicial; valor que al multiplicarlo por el número de segundos que hay en un día 86400, obtendremos un valor total en segundos de nuestra fecha inicial.

Luego realizaremos el mismo proceso con la hora inicial, para obtener el total de segundos y poder sumarlos a los de la fecha inicial.

```
HoraTSegundos = Horas * 3 600; MinutosTSegundos = Minutos * 60 ; SegundosT = Segundos;
TSegundos = HoraTSegundos + MinutosTSegundos + SegundosT
RTC_Contador = SegundosJDN + Tsegundos
```

Figura 15.22

Ese valor, lo cargamos en la variable que hemos creado la "**RTC_Contador"**, y lo grabamos en nuestro RTC mediante el comando "**RTC_SetCounter**()".

Después, cuando queramos leer el valor transcurrido en el contador acumulado del RTC, utilizamos el comando "**RTC_GetCounter**()", que nos devolverá el tiempo transcurrido en segundos; por lo que, deberemos ahora, realizar a la inversa el procedimiento de conversión de extraer una fecha y hora válidos del valor obtenido.

El primer paso para realizar la conversión será: el convertir esos segundos totales a un valor de días totales, en base al Calendario Juliano, para lo que aplicamos la siguiente ecuación:

```
TotalDias = [ (RTC_Contador + 43200) / (86400 >> 1)] + (2440587 <<1) +1;
TotalJDN = TotalDias >> 1;
```

Figura 15.23

Ahora, de este valor de total de días obtenidos, "TotalJDN", extraeremos la fecha acumulada que nos ha devuelto el RTC, utilizando esta otra ecuación:

```
a = TotalJDN + 32044;
b = (4 * a+3)/146097;
c = a - (146097 * b)/4;
d = (4 * c+3)/1461;
e = c - (1461 * d)/4;
m = (5 * e+2)/153;
Temp_Dia = e - (153 * m+2)/5 + 1;
Temp_Mes = m + 3 – 12 * (m/10);
Temp_Anno = 100 * b + d - 4800 + (m/10);
```

Figura 15.24

De la que obtenemos ya, un valor para los días, los meses y el año.

Procederemos igual ahora para obtener la hora, mediante las siguientes operaciones:

```
tiempo = RTC_Counter;
t1 = tiempo /60;
Temp_Segund = tiempo - t1*60;

tiempo = t1;
t1 = tiempo /60;
Temp_Minutos = tiempo - t1*60;

tiempo = t1;
t1 = tiempo /24;
Temp_Horas = tiempo - t1*24;
```

Figura 15.25

Ahora, ya tenemos los valores de fecha y hora que representan el tiempo acumulado que hemos obtenido de nuestro RTC. Lo siguiente, será crear unas líneas que nos permitan enviar y mostrarlas por el puerto serie USART1 de nuestro microcontrolador.

A continuación mostramos el código completo de nuestro proyecto:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rtc.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_pwr.h"
#include "stm32f10x_usart.h"
#include "misc.h"

#include <stdio.h> // Funcion que permite usar el 'printf'

/* Definicion de variables  ----------------------------------------------*/
#define JULIANO_BASE_DATE 2440588 // Dia inicial del Calendario Juliano
1/1/1970
#define TSegundosenDia    86400   // (3600*24) Segundos en un dia

/*  Se crea una matriz para guardar los valores de HH-MM-SS-DD-DS-MM-AAAA---*/
typedef struct
{
      uint8_t RTC_Horas;
      uint8_t RTC_Minutos;
      uint8_t RTC_Segundos;
      uint8_t RTC_Dia;
      uint8_t RTC_DSem;
      uint8_t RTC_Mes;
```

```
26        uint16_t RTC_Anno;
27  } RTC_FechaHoraTypeDef;
28
29  // Creamos la estructura para la Fecha y la Hora
30  //-------------------------------------------------
31  RTC_FechaHoraTypeDef  RTC_DateTimeStructure;
32
33  // Se seleccion que la fuente del reloj tendrá el RTC
34  //-------------------------------------------------
35  //#define RTCClockSource_LSE  // Reloj Externo Low
36  #define RTCClockSource_LSI    //(*) Reloj Interno
37
38  // Creamos una variable indexada para los dias de la semana
39  //-------------------------------------------------------------
40  uint8_t const *DiasSemana[] = {"Lunes","Martes","Miercoles",
41                                 "Jueves","Viernes","Sabado","Domingo"};
42
43  // Creamos una variable indexada para los meses
44  //-------------------------------------------------------------
45  uint8_t const *Meses[] = {"Enero","Febrero","Marzo","Abril","Mayo",
46                            "Junio","Julio","Agosto","Septiempre","Octubre",
47                            "Noviembre","Diciembre"};
48
49  uint8_t RTC_Estado = 0;   // Creamos la variable estado del RTC
50
51  // Lista de Funciones   ------------------------
52  void RTC_Init(void);
53  void RTC_IRQHandler(void);
54  void USART1_Config(void);
55
56  /*     Funcion que lee el contador RTC y lo convierte a Fecha y Hora       */
57  //-------------------------------------------------------------------------
58  void Lee_RTC(uint32_t RTC_Contador, RTC_FechaHoraTypeDef* RTC_DateTimeStruct)
59  {
60    unsigned long Tiempo;
61    unsigned long t1, a, b, c, d, e, m;
62    uint16_t Temp_Anno;
63    uint8_t Temp_Mes, Temp_DSem, Temp_Dia, Temp_Horas,
64            Temp_Minutos, Temp_Segund;
65    uint64_t jd = 0;;
66    uint64_t TDiasJulianos = 0;
67
68    /* Averiaguamos el Total de dias del Calendario Juliano  ---------------*/
69    //            43200=seg*12horas / 86400 Segundos en un dia
70    //a = ((counter + 43200)/(SECOND_A_DAY>>1)) + (2440587<<1) + 1;
71    //-----------------------------------------------------------------------
72    jd = ((RTC_Contador+43200)/(86400>>1)) + (2440587<<1) + 1;
73    TDiasJulianos = jd>>1;
74
75    Tiempo = RTC_Contador;
76    t1 = Tiempo/60;
77    Temp_Segund = Tiempo - t1*60;
78
79    Tiempo = t1;
80    t1 = Tiempo/60;
81    Temp_Minutos = Tiempo - t1*60;
82
83    Tiempo = t1;
84    t1 = Tiempo/24;
85    Temp_Horas = Tiempo - t1*24;
86
87    /* Extraemos el el resto [%] con ... de día de la ...  ---------------*/
88    Temp_DSem = TDiasJulianos%7;
89
```

```
90    /* Extraemos los dias, meses y anos de los dias del Calendario Juliano --*/
91    a = TDiasJulianos + 32044;
92    b = (4*a+3)/146097; // (400x365)+100-3=146097 = Total dias de 400
93                                                    anos(CuadCentury)
94    c = a - (146097*b)/4;
95    d = (4*c+3)/1461;       // 1461 => 4 anos * 365.25 = 1460,8 ~ 1461
96    e = c - (1461*d)/4;
97    m = (5*e+2)/153;
98    Temp_Dia = e - (153*m+2)/5 + 1;
99    Temp_Mes = m + 3 - 12*(m/10);
100   Temp_Anno = 100*b + d - 4800 + (m/10);
101
102   /* Pasamos los datos obtenidos a la variable matriz ----------------------*/
103   RTC_DateTimeStruct->RTC_Anno = Temp_Anno;
104   RTC_DateTimeStruct->RTC_Mes = Temp_Mes;
105   RTC_DateTimeStruct->RTC_Dia = Temp_Dia;
106   RTC_DateTimeStruct->RTC_DSem = Temp_DSem;
107   RTC_DateTimeStruct->RTC_Horas = Temp_Horas;
108   RTC_DateTimeStruct->RTC_Minutos = Temp_Minutos;
109   RTC_DateTimeStruct->RTC_Segundos = Temp_Segund;
110 }
111
112 /*  Funcion que pasa la Nueva fecha a Segundos                          */
113 //-------------------------------------------------------------------------
114 uint32_t Convierte_FechaHora(RTC_FechaHoraTypeDef* RTC_DateTimeStruct) {
115   uint8_t a;
116   uint16_t y;
117   uint8_t m;
118   uint32_t TDiasJulianos, TSegDJulianos;
119
120   /* Calcular los dias del Calendario Juliano  ------------------------*/
121   // a = (14 - Mes) /12
122   // y = Ano + 4800 - a
123   // m = Mes + 12 * a -3
124   // JDN = Dia+(((153*m +2)/5)+(365*y) + (y/4) - (y/100) + (y/400)) - 32045;
125   //-------------------------------------------------------------------------
126   A = (14-RTC_DateTimeStruct->RTC_Mes) / 12;
127   y = RTC_DateTimeStruct->RTC_Anno + 4800 - a;
128   m = RTC_DateTimeStruct->RTC_Mes+(12*a)-3;
129
130   /* Calculamos el total de Dias de la Fecha  -------------------------*/
131   TDiasJulianos = RTC_DateTimeStruct->RTC_Dia;
132   TDiasJulianos+= (((153*m+2)/5)+(365*y)+(y/4) - (y/100)+ (y/400)) - 32045;
133   TDiasJulianos = TDiasJulianos - JULIANO_BASE_DATE;
134
135   /* Pasamos los días a segundos  -------------------------------------*/
136   TSegDJulianos*= 86400;  // x 86400 Segundos en un dia
137
138   /* Sumamos al total de Segundos los segundos de la hora -------------*/
139   TSegDJulianos+= (RTC_DateTimeStruct->RTC_Horas*3600); // + TSegundos Horas
140   TSegDJulianos+= (RTC_DateTimeStruct->RTC_Minutos*60); // + TSegundos Minutos
141   TSegDJulianos+= (RTC_DateTimeStruct->RTC_Segundos);   // + TSegundos
142
143   /* Devolvemos el total de segundos de la fecha inicial al RTC -------*/
144   return TSegDJulianos;
145 }
146
147 /*       Modulo Principal                                               */
148 //=========================================================================
149 int main(void)
150 {
151   uint32_t RTC_Contador = 0;    // Creamos la variable contador
152
153   USART1_Config();    // Inicializamos el USART1
154
```

```
155  printf("\r\n ======================================================\r\n");
156  printf("  PROGRAMA DEMO DE UTILIZACION DEL CALENDARIO DEL RTC  \r\n");
157  printf(" ======================================================\r\n");
158
159  /* Comprobamos si los registros Bakcup no estan configurados  ---*/
160  if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)
161  {
162     printf("\r\n  RTC NO CONFIGURADO....\r\n");
163     printf("\r\n  Reiniciando RTC, espere....\r\n");
164     printf("\r\n  Configurando RTC, espere....\r\n");
165     RTC_Estado = 0;
166     RTC_Init();
167
168     /* Establecemos el Calendario Inicial  ----------------------*/
169     // 02-03-2018
170     RTC_DateTimeStructure.RTC_Dia  = 04;
171     RTC_DateTimeStructure.RTC_Mes  = 03;
172     RTC_DateTimeStructure.RTC_Anno = 2018;
173     // 20:03:10
174     RTC_DateTimeStructure.RTC_Horas    = 06;
175     RTC_DateTimeStructure.RTC_Minutos  = 03;
176     RTC_DateTimeStructure.RTC_Segundos = 10;
177
178     RTC_SetCounter(Convierte_FechaHora(&RTC_DateTimeStructure));
179
180     printf("\r\n  --- RTC CONFIGURADO --- !!! \\n\r\n");
181
182     /* Cargamos un valor en el registro 1 del Backup ------------*/
183     BKP_WriteBackupRegister(BKP_DR1, 0xA5A5);
184     }
185     else
186     {
187        /* Comprobamos si el reinicio es POR o PDR ----------------*/
188        if (RCC_GetFlagStatus(RCC_FLAG_PORRST) != RESET)
189        {
190            printf("\r\n  Un Reinicio a ocurrido....\r\n\r\n");
191        }
192        /* Comprobamos si el reinicio es por pulsar el pin RST ---*/
193        else if (RCC_GetFlagStatus(RCC_FLAG_PINRST) != RESET)
194        {
195            printf("\r\n  Un Reset externo se a detectado....\r\n\r\n");
196        }
197
198      printf("  NO es necesario configurar de nuevo el RTC....\r\n");
199      printf("  ... Iniciando el RTC, espere ...\r\n\r\n");
200      RTC_Estado = 1;
201      RTC_Init();
202      }
203
204      /* Borramos banderas RCC para volver a usarlas ------------*/
205      RCC_ClearFlag();
206
207      while(1)
208      {
209         if (RTC_GetCounter() - RTC_Contador > 10U)
210      {
211      RTC_Contador = RTC_GetCounter();
212      printf("\r\n\r\n  Contador segundos RTC : %d\r\n", (int)RTC_Contador);
213
214      /* Lee el valor del RTC  ----------------------------------*/
215      Lee_RTC(RTC_Contador, &RTC_DateTimeStructure);
216
217      /* Muestra la fecha y hora leida  -------------------------*/
218      printf("  Fecha: %02d/%02d/%04d Hora: %02d:%02d:%02d\r\n",
```

```
219               RTC_DateTimeStructure.RTC_Dia,
220               RTC_DateTimeStructure.RTC_Mes,
221               RTC_DateTimeStructure.RTC_Anno,
222               RTC_DateTimeStructure.RTC_Horas,
223               RTC_DateTimeStructure.RTC_Minutos,
224               RTC_DateTimeStructure.RTC_Segundos);
225
226       printf("  Dia %s, %d de %s de %04d \r\n",
227               DiasSemana[RTC_DateTimeStructure.RTC_DSem],
228               RTC_DateTimeStructure.RTC_Dia,
229               Meses[RTC_DateTimeStructure.RTC_Mes -1],
230               RTC_DateTimeStructure.RTC_Anno);
231     }
232   }
233 }
234
235 /*   Funcion que configura el RTC                    */
236 //-----------------------------------------------------
237 void RTC_Init(void)
238 {
239   /* Habilitamos el reloj APB1 para PWR y BKP ---------*/
240   RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR |
241                          RCC_APB1Periph_BKP, ENABLE);
242
243   /* Permitir acceso a los registros BKP ------------*/
244   PWR_BackupAccessCmd(ENABLE);
245
246   /* Comprobamos si es un reinicio de un nuevo RTC --*/
247   if (RTC_Estado != 1)
248   {
249     printf("  Reinicio del RTC \r\n");
250     /* Reseteo de los registros Backup  -------------*/
251     BKP_DeInit();
252
253     /* Reseteando registros backup del RTC  ---------*/
254     RCC_BackupResetCmd(ENABLE);
255     RCC_BackupResetCmd(DISABLE);
256     printf("    - Reseteando backup de registros RTC\r\n");
257   }
258
259   ////////////////////////////////////////////////////
260   // ----   Si se habilita el  reloj LSI    --------//
261   #ifdef RTCClockSource_LSI
262   ////////////////////////////////////////////////////
263   /* Habilitar reloj LSI ----------------------------*/
264   RCC_LSICmd(ENABLE);
265   printf("    - Habilitando fuente LSI \r\n");
266   //
267   /* Espere hasta que el LSI esté listo ------------*/
268   while(RCC_GetFlagStatus(RCC_FLAG_LSIRDY)==RESET)
269   {;}
270   //
271   /* Seleccionamos LSI como fuente de reloj del RTC */
272   RCC_RTCCLKConfig(RCC_RTCCLKSource_LSI);
273   printf("    - Habilitando LSI en RTCCLK \r\n");
274   ////////////////////////////////////////////////////
275   // ----  Si se habilita el reloj LSE     ---------//
276   #elif defined RTCClockSource_LSE
277   ////////////////////////////////////////////////////
278   /* Habilitar reloj LSE ----------------------------*/
279   RCC_LSEConfig(RCC_LSE_ON);
280   printf("    - Habilitando fuente LSE \r\n");
281   //
282   /* Espere hasta que el LSE esté listo ------------*/
```

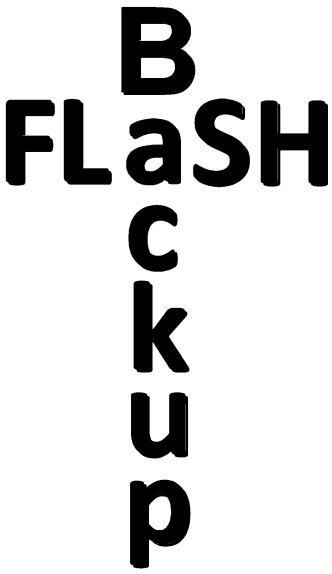
```
283   while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
284   {;}
285   /* Deshabilitamos el reloj Lsi  ------------------*/
286   RCC_LSICmd(DISABLE);
287   printf("    - Deshabilitando LSI \r\n");
288   //
289   /* Seleccionamos LSE como fuente de reloj del RTC */
290   RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);
291   printf("    - Habilitando LSE para RTCCLK \r\n");
292   #endif
293   ////////////////////////////////////////////////////
294   /*  Habilita el reloj RTC   ----------------------*/
295   RCC_RTCCLKCmd(ENABLE);
296   printf("    - Habilitando RTCCLK\r\n");
297   //
298   /* Espera hasta que finalice la última operación
299      de escritura en registros RTC ------------------*/
300   RTC_WaitForLastTask();
301   //
302   /* Espera a que se sincronice los registros RTC --*/
303   printf("    - Esperando sincronizacion \r\n");
304   RTC_WaitForSynchro();
305   //
306   /* Espera hasta que finalice la última operación
307      de escritura en registros RTC ------------------*/
308   RTC_WaitForLastTask();
309   //
310   /* Habilitar la Interrupcio RTC por Segundos -----*/
311   RTC_ITConfig(RTC_IT_SEC, ENABLE);
312   printf("    - Habilitando Interrupcion general RTC \r\n");
313   //
314   /* Habilitar la Interrupcion ALARMA del RTC  -----*/
315   RTC_ITConfig(RTC_IT_ALR, ENABLE);
316   printf("    - Habilitando Interrupcion de Alarma RTC \r\n");
317   //
318   /* Espera hasta que finalice la última operación
319      de escritura en registros RTC ------------------*/
320   RTC_WaitForLastTask();
321   //
322   ////////////////////////////////////////////////////////////
323   /* Establece el período del contador RTC en 1 segundo -----*/
324   // Para LSE = RTCCLK/RTC period = 32768Hz/1Hz = 32767+1
325   // para LSI = RTCCLK/RTC period = 40000Hz/1Hz = 39999+1
326   ////////////////////////////////////////////////////////////
327   #ifdef RTCClockSource_LSI
328   ////////////////////////////////////////////////////////////////////
329   /* Establecemos el Preescaler del RTC para un período de 1 segundo*/
330   RTC_SetPrescaler(39999); /* Peridodo RTC = RTCCLK/RTC_Prescal =
331                                                  (40000Hz)/39999+1)*/
332   printf("    - Configurado Prescaler LSI (40 kHz) \r\n");
333   ////////////////////////////////////////////////////////////////////
334   #elif defined  RTCClockSource_LSE
335   ////////////////////////////////////////////////////////////////////
336   /* Establecemos el Preescaler del RTC para un período de 1 segundo*/
337   RTC_SetPrescaler(32767); /* Periodo RTC = RTCCLK/RTC_Prescal =
338                                            (32.768KHz)/(32767+1) */
339   printf("    - Configurado Prescaler LSE (32,768 kHz) \r\n");
340   #endif
341   ////////////////////////////////////////////////////////////////////
342   //
343   /* Espere hasta que finalice la última operación
344      de escritura en registros RTC ----------------------------------*/
345   RTC_WaitForLastTask();
346 }
```

```
347
348 /*           Funcion que configura el USART1                    */
349 //--------------------------------------------------------------
350 void USART1_Config(void)
351 {
352   /* Configuramos el GPIO para USART1 ----------------------*/
353   GPIO_InitTypeDef GPIO_InitStructure;
354
355   /* Habilitamos los reloj para GPIO y el USART1 -----------*/
356   RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 |
357                          RCC_APB2Periph_GPIOA, ENABLE);
358
359   /* Configuramos PA9 para Tx USART1 -----------------------*/
360   GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_9;
361   GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF_PP;
362   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
363   GPIO_Init(GPIOA, &GPIO_InitStructure);
364
365   /* Configuramos PA10 para Rx USART1 ----------------------*/
366   GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_10;
367   GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN_FLOATING;
368   GPIO_Init(GPIOA, &GPIO_InitStructure);
369
370   /* Configuramos el USART1  -------------------------------*/
371   USART_InitTypeDef USART_InitStructure;
372   USART_InitStructure.USART_BaudRate   = 115200;
373   USART_InitStructure.USART_WordLength = USART_WordLength_8b;
374   USART_InitStructure.USART_StopBits   = USART_StopBits_1;
375   USART_InitStructure.USART_Parity     = USART_Parity_No;
376   USART_InitStructure.USART_HardwareFlowControl =
377                              USART_HardwareFlowControl_None;
378   USART_InitStructure.USART_Mode       = USART_Mode_Rx | USART_Mode_Tx;
379   USART_Init(USART1, &USART_InitStructure);
380   USART_Cmd(USART1, ENABLE);
381 }
382
383 /* Funcion que nos permite utilizar el comando 'printf'    */
384 int fputc(int ch, FILE *f)
385 {
386       /* Envia dato ----------------------------------------*/
387       USART_SendData(USART1, (unsigned char) ch);
388
389       while( USART_GetFlagStatus(USART1, USART_FLAG_TC)!= SET);
390       return (ch);
391 }
392
393 /* Funcion que se genera cada vez que se produce la IRQ del RTC  */
394 //--------------------------------------------------------------
395 void RTC_IRQHandler(void)
396 {
397   /* Comprobamos si se ha producido la IRQ general del RTC -------*/
398   if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
399   {
400     /* Receteamos la bandera de la Interrupcio RTC por Segundos --*/
401     RTC_ClearITPendingBit(RTC_IT_SEC);
402
403     /* Esperamos a que se termine el acceso a los registros  -----*/
404     RTC_WaitForLastTask();
405   }
406 }
407
```

Figura 15.26

16

PROGRAMACIÓN BKP Y FLASH



Como pudimos comprobar en nuestro anterior capítulo, donde explicamos el reloj RTC, existe un módulo que nos permite grabar en unos determinados registros de nuestro microcontrolador información o valores de nuestra programación que se mantendrá a pesar de que se resetee o se desconecte la alimentación principal, pero manteniendo la alimentación con una batería externa.

En los microcontroladores STM32, el módulo **BKP** (*Backup Register*) consta de hasta 42 registros de 16 bits que pueden almacenar 84 bytes de datos, que componen el área de datos de respaldo y los registros del RTC que vimos antes.

Se pueden alimentar por VBAT y permanecer operativo cuando se apaga el VDD. No se restablece cuando el sistema se reinicia. Aunque debemos señalar aquí, que cuando el microcontrolador se encuentra alimentado con VBA y no se encuentra la alimentación VDD, no está funcionando.

También que, cuando se restablece el sistema, a estos registros se les protege de posibles reescrituras producidos por señales parásitas en la alimentación; siendo necesario deshabilitar esta protección para poder escribir algún dato en estos registros.

16.1 EJEMPLO DE UTILIZACIÓN DEL BKP

En el siguiente proyecto de ejemplo, establecemos un sencillo programa en el que se crea un contador cuyo valor se graba en el primer registro del área de backup, el BKP_DR1. Este se incrementará en '1' cada vez que se pulse el botón de reinicio de nuestra placa –el switch Boot0 deberá estar en el modo de su posición '0' -. Al iniciar el sistema, primero se leerá el valor grabado anteriormente en la memoria BKP y luego se incrementará en '1'; mostrándonos el sistema el valor actual del contador por el puerto USART1. Y produciendo que, cada vez que reiniciemos el microcontrolador, se incremente el contador al valor que tenía almacenado en el registro backup más uno; demostrando que se guarda la información cuando se resetea el sistema o se desconecta la alimentación, manteniéndose la conexión a Vbat con una batería de 3.3 Vcc.

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_pwr.h"
#include "stm32f10x_bkp.h"
#include "stm32f10x_usart.h"
#include <stdio.h>

//  Lista de funciones ----------------------------------------
void USART1_Config(void);

/-                  Funcion          Principal                */
//*************************************************************
int main(void)
{
   uint16_t Contador = 0; // Creamos la variables 'Contador' de 16 bits

   USART1_Config();       // Inicializamos el USART1

   // Habilitamos los relojes para modulos BKP y PWM
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

   // Deshabilitamos la protección de escritura en los registros BKP
   PWR_BackupAccessCmd(ENABLE);

   // Leemos el valor del 'Contador' guardado en el registro 1 del BKP
   Contador = BKP_ReadBackupRegister(BKP_DR1);

```

```
29    // Aumentamos en 1 el 'Contador' tras el reinicio
30    Contador++;
31
32    // Volvemos a grabar el valor actual del 'Contador'
33    BKP_WriteBackupRegister(BKP_DR1, Contador);
34
35    // Mostramos por el puerto USART1 el valor actual del 'Contador'
36    printf("\r\n Valor del BKP_DR1: %d\r\n", Contador);
37
38    while(1)
39    {;;}
40 }
41
42 //          Funcion que configura el USART1                    */
43 //-------------------------------------------------------------
44 void USART1_Config(void)
45 {
46       RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 |
47                              RCC_APB2Periph_GPIOA, ENABLE);
48       GPIO_InitTypeDef GPIO_InitStructure;
49       GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
50       GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
51       GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
52       GPIO_Init(GPIOA, &GPIO_InitStructure);
53       GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
54       GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
55       GPIO_Init(GPIOA, &GPIO_InitStructure);
56       USART_InitTypeDef USART_InitStructure;
57       USART_InitStructure.USART_BaudRate = 115200;
58       USART_InitStructure.USART_WordLength = USART_WordLength_8b;
59       USART_InitStructure.USART_StopBits = USART_StopBits_1;
60       USART_InitStructure.USART_Parity = USART_Parity_No;
61       USART_InitStructure.USART_HardwareFlowControl =
62                                       USART_HardwareFlowControl_No
63       USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_T
64       USART_Init(USART1, &USART_InitStructure);
65       USART_Cmd(USART1, ENABLE);
66 }
67
68 // Funcion que nos permite utilizar el comando 'printf'
69 int fputc(int ch, FILE *f)
70 {
71       /* Envia dato */
72       USART_SendData(USART1, (unsigned char) ch);
73
74       while( USART_GetFlagStatus(USART1,USART_FLAG_TC)!= SET);
75       return (ch);
76 }
```

Figura 16.1

En nuestro ejemplo, vemos que los pasos son muy sencillos.

En el módulo principal "*main*", creamos primero una variable de 16 bits, que será la que lleve el contador que vamos a usar.

En la siguiente línea, habilitamos los relojes para sincronizar los módulos que vamos a utilizar de nuestro microcontrolador, el módulo **BKP** y el módulo de control de alimentación de los registros **PWR** (*Power control register*).

A continuación, habilitamos el acceso a los registros BKP, mediante el comando "**PWR_BackupAccessCmd** (ENABLE)" que deshabilita la protección de escritura.

Después con el comando "**BKP_ReadBackupRegister**(BKP_DR1)", leemos el contenido del registro 1 del área de memoria backup y pasamos el valor obtenido a nuestra variable 'Contador' que se incrementa en '1' cada vez que se inicie el sistema.

Luego, ese valor lo enviamos a través del módulo USART1 hacia nuestro ordenador.

16.2 EJEMPLO DE UTILIZACIÓN DE VARIOS REGISTROS DEL BKP

En nuestro siguiente proyecto, utilizaremos varios de los registros del módulo BKP para almacenar una serie de números y comprobar así su uso con mayor cantidad de datos.

Mostramos a continuación el código del ejemplo:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_pwr.h"
#include "stm32f10x_bkp.h"
#include "stm32f10x_usart.h"
#include "misc.h"

#include <stdio.h>

// Definimos el maximo de registros de nuestro microcntrolador
#define NUM_Max_RBKP 10

// Creamos una variable indexada para su manejo
uint16_t RegistroBKP[NUM_Max_RBKP] = { BKP_DR1, BKP_DR2, BKP_DR3, BKP_DR4,
        BKP_DR5, BKP_DR6, BKP_DR7, BKP_DR8, BKP_DR9, BKP_DR10};


/*          Funcion que configura modulo BKP             */
//-------------------------------------------------------
void BKP_Init()
{
  // Habilitamos los relojes para modulos BKP y PWM
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

  // Deshabilitamos la protección de escritura en los registros BKP
  PWR_BackupAccessCmd(ENABLE);
}

/*        Funcion que graba datos en los registros BKP    */
//-------------------------------------------------------
void Graba_Registro(uint16_t index, uint16_t valor)
{
  // Grabamos en el registro BKP_DR(index) un dato
  BKP_WriteBackupRegister(RegistroBKP[index], valor);
}

```

```
/*        Funcion que lee datos de los registros BKP       */
//-------------------------------------------------------------
uint16_t Lee_Registro(uint16_t index)
{
  // Leemos el valor del 'Contador' guardado en el registro 1 del BKP
  uint16_t valor = BKP_ReadBackupRegister(RegistroBKP[index]);

  return valor;
}

//  Lista de funciones ----------------------------------------
void USART1_Config(void);

/*                  Funcion  Principal                      */
//*************************************************************
int main(void)
{
   uint16_t Contador = 0;// Creamos la variables 'Contador'

   USART1_Config();      // Inicializamos el USART1

   BKP_Init();           // Inicializamos el modulo BKP

   printf("\r\n =====================================\r\n");
   printf("  PROGRAMA DEMO DE UTILIZACION DE BKP  \r\n");
   printf(" =====================================\r\n");

   // Leemos el valor del 'Contador' guardado en el registro 1 del BKP
   Contador = Lee_Registro(0);

   // Aumentamos en 1 el 'Contador' tras el reinicio
   Contador++;

  // Volvemos a grabar el valor actual del 'Contador'
  Graba_Registro(0, Contador);

  printf(" Registro BKP_DR1 : %d", Lee_Registro(0));

  // Comprobamos si los registros contienen informacion
  if (Lee_Registro(2) != 102)
  {
     printf("\r\n  Registro BKP vacios, espere...\r\n");

     // Si estan vacios los llenamos con datos
     for (uint32_t Indice = 1; Indice < NUM_Max_RBKP; Indice++)
     {
      Graba_Registro(Indice, 100 + (Indice));
     }
     printf("\r\n  ... Registros BKP rellenados\n\r\n");
  }
  // Si los registros contienen datos
  else {
     // Comprobamos si el reinicio es POR o PDR o pin reset
     if (RCC_GetFlagStatus(RCC_FLAG_SFTRST))
     {
        printf("\r\n  Reinicio a ocurrido....\r\n\r\n");
     }
     // Comprobamos si el reinicio es por pulsar el pin NRST
     else if (RCC_GetFlagStatus(RCC_FLAG_PINRST))
     {
        printf("\r\n  Boton Reset pulsado detectado....\r\n\r\n");
     }

     for (uint32_t Indice = 1; Indice < NUM_Max_RBKP; Indice++)
     {
       // Leemos el valor del 'Contador' guardado en el registro 1 del BKP
       printf(" Valor Registro BKP_DR%2d : %d \n\r", Indice+1, Lee_Registro
     }

  }
  // Reseamos las banderas RCC para reutilizarlas
```

```
109   RCC_ClearFlag();
110
111   // Mostramos por el puerto USART1 el valor actual del 'Contador'
112   printf("\r\n Valor Contador del BKP_DR1: %d\r\n", Contador);
113
114   while(1)
115     {
116     }
117 }
118
119 /*          Funcion que configura el USART1                */
120 //-------------------------------------------------------------
121 void USART1_Config(void)
122 {
123       RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);
124       GPIO_InitTypeDef GPIO_InitStructure;
125       GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
126       GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
127       GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
128       GPIO_Init(GPIOA, &GPIO_InitStructure);
129       GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
130       GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
131       GPIO_Init(GPIOA, &GPIO_InitStructure);
132       USART_InitTypeDef USART_InitStructure;
133       USART_InitStructure.USART_BaudRate = 115200;
134       USART_InitStructure.USART_WordLength = USART_WordLength_8b;
135       USART_InitStructure.USART_StopBits = USART_StopBits_1;
136       USART_InitStructure.USART_Parity = USART_Parity_No;
137       USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
138       USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
139       USART_Init(USART1, &USART_InitStructure);
140       USART_Cmd(USART1, ENABLE);
141 }
142
143 // Funcion que nos permite utilizar el comando 'printf'
144 int fputc(int ch, FILE *f)
145 {
146       /* Envia dato */
147       USART_SendData(USART1, (unsigned char) ch);
148
149       while( USART_GetFlagStatus(USART1,USART_FLAG_TC)!= SET);
150       return (ch);
151 }
```

Figura 16.2

16.3 EMPLEO DE LA MEMORIA FLASH

Los microcontroladores STM32 poseen también una memoria flash interna de alta velocidad y que, dependiendo del modelo de microcontrolador puede ser de hasta 1 Mbyte (ver tabla de la Figura 16.3). En el modelo de microcontrolador que posee nuestra placa de pruebas el STM32F103C8 es de 64 Kilobytes. Se trata de una memoria no-volátil, que mantiene la información grabada en ella sin necesidad de alimentación y por lo tanto, se utiliza como si se tratara de una memoria EEPROM.

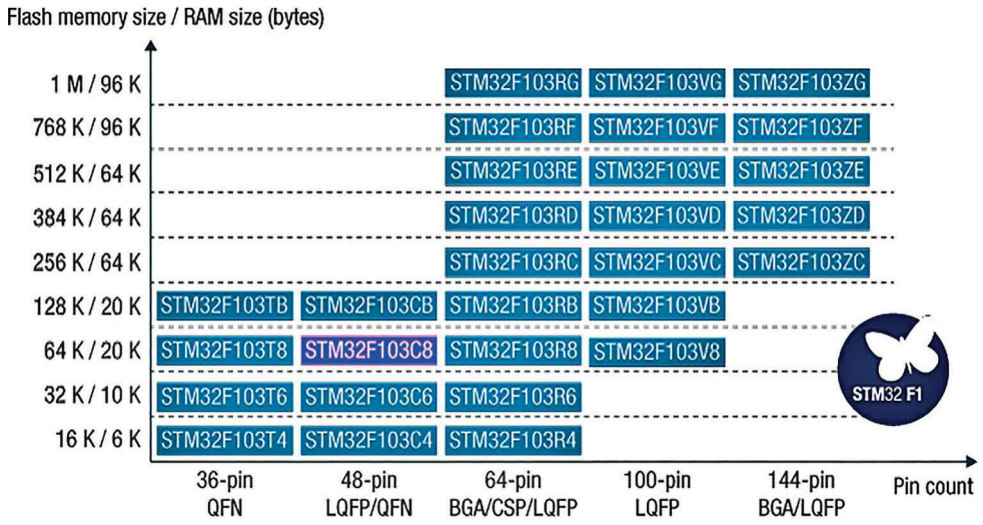


Figura 16.3

Esta memoria flash, se encuentra organizada en celdas de memoria de 32 bits que pueden ser utilizadas para almacenar valores de código o datos y está ubicada a partir de una dirección de memoria específica. En nuestro microcontrolador, esta comienza en la dirección "0x08000000", con un largo de 64 Kilobytes y configurada en páginas de 1024 Kilobytes.

La memoria flash total de nuestro microcontrolador se haya organizada en tres grandes bloques: memoria principal (*Main memory*), un bloque de memoria de información (*Information block*) y un tercer bloque reservado para la memoria flash de los periféricos.

El bloque de información, posee a su vez dos divisiones más.

- *System memory* Zona de memoria del sistema, área de 2 Kilobytes que está reservada al fabricante STMicroelectronics y contiene el gestor de arranque (*bootloader*) que permite programar el chip utilizando la propia interfaz serial del microcontrolador (USART1). Esta área es programada por ST en el momento de su fabricación y es protegida contra operaciones ilegítimas de escritura / borrado.

- *Option bytes* Una zona tolerante a operaciones de escritura y borrado gestionadas por el módulo FPEC (*Flash program/erase Controller*).

La memoria flash puede protegerse contra diferentes tipos de acceso no deseado (lectura / escritura / borrado). Hay dos tipos de protección: protección contra escritura de página y protección de lectura.

Por ejemplo, durante una operación de escritura en la memoria flash, cualquier intento de leer la memoria detendrá el bus. La operación de lectura se realizará correctamente una vez que se haya completado la operación de escritura. Esto significa que las capturas o lecturas de código o datos no se pueden realizar mientras está en curso una operación de escritura o borrado.

Para las operaciones de escritura y borrado en la memoria flash, el oscilador RC interno (HSI) debe estar ENCENDIDO. La operación de fin de escritura en la programación o el borrado, puede disparar una interrupción que podemos configurar.

El manejo de esta memoria tiene algunas especificaciones que no se pueden vulnerar:

- Es necesario desbloquear la memoria Flash antes de poder operar en ella con operaciones de borrado o escritura.
- Se puede escribir memoria de solo pre-borrado. Cuando la memoria tiene borrada la información, todos sus bits estarán a valor '0xFF'.
- Solo puede borrarse la página entera, no es posible borrar bytes específicos.
- La escritura en dicha memoria debe hacerse en palabras de 32 bits.

A continuación, exponemos un ejemplo de código en el que mostraremos el uso de esta memoria como sistema de almacenamiento para nuestros datos:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_usart.h"
#include "stm32f10x_exti.h"
#include "stm32f10x_flash.h"   //(*)

#include <stdio.h>

// Definimos una variable que contendrá el inicio de nuestra pagina
#define DIRECC_MI_PAGINA 0x800FC00

// Definimos una estructura para manejar los datos en bloques de 32 bits
typedef struct
{
  uint8_t Dato1;  // Dato1 +1 byte  (8 bits)
  uint8_t Dato2;  // Dato2 +1 byte  (8 bits)
  uint16_t Dato3; // Dato3 +2 byte  (16 bits)
  uint32_t Dato4; // Dato4 +4 bytes (32 bits)
                  // = 8 bytes totales = 32 bits words_leng
} WordByte;

// Nombramos la estructura de nuestra variable de array
```

```
24 WordByte WordByteStructure;
25
26 // CREAMOS UNA VARIABLE QUE GUARDA EL TAMAÑO DE CADA CAMPO DE NUESTRA ESTRUCTURA
27 #define SIZE_DATO_WORDBYTE sizeof(WordByteStructure)/4
28
29 // Lista de funciones ---------------------------------
30 void USART1_Config(void);
31 void GPIO_Config(void);
32 void NVIC_Configuration(void);
33 void EXT5_Config(void);
34 void EXTI9_5_IRQHandler(void);
35
36 /*   Funcion que inicializa el modulo FLASH        */
37 //-------------------------------------------------
38 void FLASH_Init(void)
39 {
40     /* Habilitar buffer de precarga */
41     FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
42
43     /* Establecemos el Estado de latencia a 2 */
44     FLASH_SetLatency(FLASH_Latency_2);
45 }
46
47 /*     Funcion que extrae lo datos guardados en la memoria Flash   */
48 //-------------------------------------------------------------------
49 void Lee_Flash(void) {
50     // Establece las variables de
51     // Direccion Origen utilizando el puntero de los datos en la Flash
52       uint32_t *direcc_origen = (uint32_t *)DIRECC_MI_PAGINA;
53
54     // y de Direccion Destino de nuestros datos para el array
55       uint32_t *direcc_destino = (void *)&WordByteStructure;
56
57     // Crea un bucle que va extrallendo cada dato de las
58     // direcciones origenes del puntero de nuestro array
59       for (uint16_t i=0; i<SIZE_DATO_WORDBYTE; i++)
60       {
61        // Pasa cada dato a su posicion de direccion del array
62          *direcc_destino = *(__IO uint32_t*)direcc_origen;
63
64        // incrementamos los valores de las direcciones
65          direcc_origen++; direcc_destino++;
66       }
67 }
68
69 /*   Funcion que graba los datos del array en direcciones de memoria Flash  */
70 //-------------------------------------------------------------------------
71 void Graba_Flash(void) {
72     // Desbloquea memoria flash para poder manipularla
73     FLASH_Unlock();
74
75     // Borrar toda la pagina de nuestros datos
76     FLASH_ErasePage(DIRECC_MI_PAGINA);
77
78     // Establece las variables de
79     // Direccion Origen utilizando el puntero de los datos en la Flash
80       uint32_t *direcc_origen = (void*) (&WordByteStructure);
81
82     // y de Direccion Destino la Primera pagina de nuestro ejemplo
83       uint32_t *direcc_destino = (uint32_t *) DIRECC_MI_PAGINA;
84
85    // Crea un bucle que va grabando en la direccion de la primera pagina
86    // de nuestro proyecto los datos extraidos del puntero de nuestro array
87       for (uint16_t i=0; i<SIZE_DATO_WORDBYTE; i++)
88       {
89          // Graba en la memoria Flash en direccion origen los datos del array
90          FLASH_ProgramWord((uint32_t)direcc_destino, *direcc_origen);
91
92          // incrementamos los valores de las direcciones
93           direcc_origen++; direcc_destino++;
94       }
95
```

```
96    // Bloquea de nuevo la memoria FLASH
97    FLASH_Lock();
98  }
99
100 /*              Modulo Principal                          */
101 //*********************************************************
102 int main(void)
103 {
104     USART1_Config();     // Inicializamos el USART1
105     GPIO_Config();       // Inicializamos el GPIO para el LED
106     NVIC_Configuration(); // Inicializamos el NVIC para IRQ del RTC
107     EXT5_Config();       // Inicializa la interrupcion del Boton
108
109     printf("\r\n ======================================================\r\n");
110     printf("  PROGRAMA EJEMPLO DE UTILIZACION DE LA MEMORIA FLASH  \r\n");
111     printf(" ======================================================\r\n");
112
113     FLASH_Init();        // Inicializamos el modulo FLASH
114
115     // Tras un reinicio del sistema Comprueba si el reinicio
116     // es por apagado (POR o PDR) o por pulsar el botón de reset
117     if (RCC_GetFlagStatus(RCC_FLAG_SFTRST))
118     {
119        printf("\r\n  Un Reinicio por software ocurrido....\r\n\r\n");
120     }
121     else if (RCC_GetFlagStatus(RCC_FLAG_PORRST))
122     {
123        printf("\r\n  Reinicio por Power On/Off detectado....\r\n\r\n");
124     }
125
126     // Comproba si el reinicio es por pulsar el botón de reset NRST
127     else if (RCC_GetFlagStatus(RCC_FLAG_PINRST))
128     {
129        printf("\r\n  Boton Reset pulsado detectado....\r\n\r\n");
130     }
131
132     // Resea las banderas RCC para reutilizarlas
133     RCC_ClearFlag();
134
135     // Lee si existen datos almacenados y los pasa al array
136     Lee_Flash();
137
138     // Incrementa los valores en '1' del array
139     WordByteStructure.Dato1 = WordByteStructure.Dato1+1;
140     WordByteStructure.Dato2 = WordByteStructure.Dato2+1;
141     WordByteStructure.Dato3 = WordByteStructure.Dato3+1;
142     WordByteStructure.Dato4 = WordByteStructure.Dato4+1;
143
144     // Graba de nuevo los datos en la memoria FLASH
145     Graba_Flash();
146
147     // Enviamos la informacion por el puerto USART1
148     printf("  Datos leidos [1] %d [2] %d [3] %d [4] %d \n\r",
149         WordByteStructure.Dato1,
150         WordByteStructure.Dato2,
151         WordByteStructure.Dato3,
152         WordByteStructure.Dato4);
153
154     while (1)
155     {
156       // Enciende y apaga (Toggle) LED en PC13
157             GPIOC->ODR ^= GPIO_Pin_13;
158       // delay
159       for(int i=0;i<0x100000;i++);
160     }
161 }
162
163 //       Funcion que configura el USART1                  */
165 void USART1_Config(void)
166 {
167       RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 |
```

```
168                              RCC_APB2Periph_GPIOA, ENABLE);
169       GPIO_InitTypeDef GPIO_InitStructure;
170       GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
171       GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
172       GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
173       GPIO_Init(GPIOA, &GPIO_InitStructure);
174       GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
175       GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
176       GPIO_Init(GPIOA, &GPIO_InitStructure);
177       USART_InitTypeDef USART_InitStructure;
178       USART_InitStructure.USART_BaudRate = 115200;
179       USART_InitStructure.USART_WordLength = USART_WordLength_8b;
180       USART_InitStructure.USART_StopBits = USART_StopBits_1;
181       USART_InitStructure.USART_Parity = USART_Parity_No;
182       USART_InitStructure.USART_HardwareFlowControl =
183 USART_HardwareFlowControl_None;
184       USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
185       USART_Init(USART1, &USART_InitStructure);
186       USART_Cmd(USART1, ENABLE);
187 }
188
189 // Funcion que nos permite utilizar el comando 'printf'
190 int fputc(int ch, FILE *f)
191 {
192       // Envia dato por el USART1
193       USART_SendData(USART1, (unsigned char) ch);
194
195       while( USART_GetFlagStatus(USART1,USART_FLAG_TC)!= SET);
196       return (ch);
197 }
198
199 /*     Funcion que configura el NVIC para EXTI_5_IRQ    */
200 //-------------------------------------------------------
201 void NVIC_Configuration(void)
202 {
203     NVIC_InitTypeDef NVIC_InitStructure;
204     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
205
206     // Condiguracion NVIC para EXT5 canal 5 IRQ en PA5
207     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
208     NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;
209     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
210     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
211    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
212    NVIC_Init(&NVIC_InitStructure);
213 }
214
215 /*    Funcion que configura el GPIO para LED y PULSADOR     */
216 //-------------------------------------------------------------
217 void GPIO_Config(void)
218 {
219     GPIO_InitTypeDef  GPIO_InitStructure;
220
221     // Activamos el reloj para el GPIO
222     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC |
223                            RCC_APB2Periph_GPIOA, ENABLE);
224
225     // Configuramos los parametros para el pin PC13
226     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
227     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
228     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
229     GPIO_Init(GPIOC, &GPIO_InitStructure);
230
231     /* Configuracion para el pin PA5 entrada pulsador */
232     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
233     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
234     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
235     GPIO_Init(GPIOA, &GPIO_InitStructure);
236 }
237
238 /*  Funcion que configura la Interrupcion IRQ_5  en PA5   */
```

```
239 //-------------------------------------------------------------
240 void EXT5_Config(void){
241
242       // Activamos el reloj para GPIO en puerto A
243       RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
244       EXTI_InitTypeDef EXTI_InitStructure;
245       //Configura la interrupcion EXTI5 en linea 5 de PA5
246       EXTI_InitStructure.EXTI_Line = EXTI_Line5;
247       EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
248       EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
249       EXTI_InitStructure.EXTI_LineCmd = ENABLE;
250       EXTI_Init(&EXTI_InitStructure);
251 }
252
253 /*  Funcion que controla si se pulsa el boton en PA5  */
254 //-------------------------------------------------------
255 void EXTI9_5_IRQHandler(void)
256 {
257       // Comprueba si se ha producido la interrupcion
258       if(EXTI_GetITStatus(EXTI_Line5) != RESET)
259       {
260         // Desactiva la bandera de interrupcion para que se vuelva a comprobar
261         EXTI_ClearITPendingBit(EXTI_Line5);
262
263         // Desbloquea memoria flash para poder manipularla
264         FLASH_Unlock();
265
266         // Borrar toda la pagina de nuestros datos
267         FLASH_ErasePage(DIRECC_MI_PAGINA);
268
269         // Mostramos el mensaje de Borrado
270         printf("\n\r  MEMORIA FLASH BORRADA !!!! \r\n");
271
272         // delay
273         for(int i=0;i<0x2000000;i++);
274
275       // Realizamos un reset del sistema (Software reset)
276       SCB->AIRCR = 0x05fa0004;
277       }
278 }
```

Figura 16.4

En nuestro código de ejemplo comenzamos por definir una serie de parámetros en variables globales que emplearemos contantemente en el resto del programa.

Usamos una variable para indicar a partir de qué dirección de la memoria flash queremos que se guarden nuestros datos. Hay que recordar que la memoria flash también es compartida por nuestro microcontrolador para según qué opciones hayamos establecido de inicio "*Boot*" –vean al comienzo del libro cuando hablábamos de la configuración de los jumper de nuestra placa de pruebas para programarla-. La programación de arranque "*bootloader*" utilizará parte de la memoria flash para guardarse y también para guardar nuestro firmware o programación propia.

En nuestro ejemplo, definimos la variable 'DIRECC_MI_PAGINA" a partir de la dirección '0x0800FC001" que sabemos estará libre.

A continuación, para facilitar el manejo de los datos que necesitamos que estén agrupados en una palabra completa de 32 bits, creamos una estructura que contendrá cuatro datos, dos de 1 byte, uno de 2 bytes y un último de 4 bytes; en total los 8 bytes que necesitamos para crear un array de datos de 32 bits. A la que le damos el nombre de "*WordByte*".

También definimos una variable con el nombre 'SIZE_DAT_WORDBYTE" a la que le damos el valor que tiene el tamaño de los campos de nuestros datos.

La primera función que creamos es la que inicializa el módulo, la "**FLASH_Init**()", en la que establecemos los pasos estándar que son utilizados para ello. En primer lugar el comando "**FLASH_PrefetchBufferCmd**(*FLASH_PrefetchBuffer_*Enable)", es el que habilita (ENABLE) o desabilita (DISABLE) el "*buffer Prefetch*" de nuestro módulo. En segundo lugar el comando "**FLASH_SetLatency**(*FLASH_Latency_*x)", con el que establecemos el valor de latencia máxima que queremos y que se pueden producir cuando se accede a la memoria debido a problemas de voltaje, frecuencias de reloj o a estados de exceso de potencia. Podemos establecer tres modos de latencia:

- **FLASH_Latency_0** - Cero ciclos de latencia si se cumple [0 < SYSCLK ≤ 24 MHz].
- **FLASH_Latency_1** - Un ciclo de lantencia si se cumple [24MH < SYSCLK ≤ 48 MHZ].
- **FLASH_Latency_2** - 2 ciclos de latencia si se cumple [48MH < SYSCLK ≤ 72 MHZ].

Figura 16.5

Bastan estos dos parámetros para que se inicie el acceso a nuestra memoria Flash.

La siguiente función que creamos es la que nos permite leer la memoria Flash, "**Lee_Flash**()". Primero creamos dos variables donde guardamos las direcciones origen y destino que vamos a utilizar en la siguiente instrucción y en la que usamos los punteros que representan las direcciones de memoria:

```
*direcc_destino = *(__IO uint2_t *) direcc_origen
```

Figura 16.6

También crearemos un bucle mediante el comando de C++ "for…" para que se incrementen automáticamente cada una de las direcciones y poder leer así los cuatro datos para pasarlos a los punteros de nuestro array "**(void *)** ***&WordByteStructure***".

A continuación, creamos la función que nos permite grabar nuestros datos en la memoria Flash: la función "**Graba_Flash**()"; en donde lo primero que necesitamos es habilitar el acceso a dicha memoria, que recordemos que por defecto está protegida contra escritura; que deshabilitamos con el comando "**FLASH_Unlock**()".

Volvemos a utilizar los punteros a nuestro array y la variable que representa el inicio de nuestra página para cargarlos a las variables que indican las direcciones de memoria de origen y destino de los datos.

Ahora, mediante el comando "**FLASH_ProgramWord**(*dirección destino*, *dirección origen*)", podemos grabar nuestros datos en las direcciones de memoria de nuestra flash.

Creamos también un bucle para que se incrementen dichos valores y grabar cada campo de nuestro array.

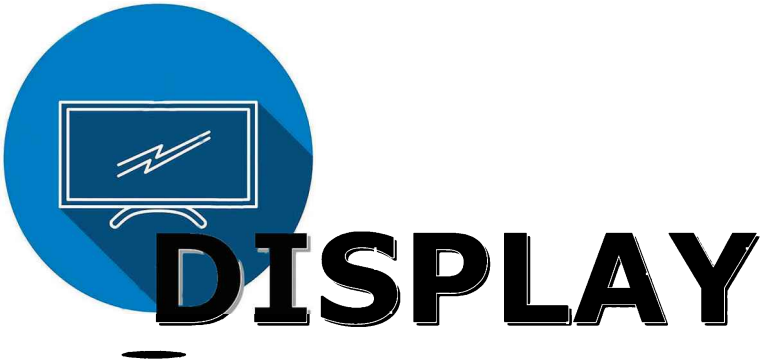
Al final del proceso de escritura, debemos volver a colocar la protección de acceso a la memoria Flash mediante el comando "**FLASH_Lock**()".

También hemos creado otras funciones: como las que configuran los parámetros del USART1 y las que configuran el GPIO para utilizar los pines PC13 con el led de pruebas (que se encenderá y apagará indicando que el programa está ejecutándose y el del pulsador que colocamos en el pin PA5 y al que le configuramos la interrupción IEXT_IRQ_5 para que, cuando se detecte su pulsación, se borre el contenido de nuestra página y se reinicie el sistema).

Igualmente, añadimos unas líneas al comienzo del módulo "*main*" para que nos informe si se produce un reinicio y de qué tipo ha sido. Con la finalidad de que podamos comprobar que, cuando se pulsa el botón reset de nuestra placa o incluso cuando desconectamos la alimentación de la misma – no es necesaria la conexión de una batería externa para que se mantenga la información grabada- y volvemos a conectarla, al reiniciar nuestra placa, la información que ha sido grabada en la memoria flash permanece, creando un contador que se graba en nuestros datos de prueba para demostrarlo. Todo ello, enviando por el puerto serie USART los mensajes de cada proceso.

17

PROGRAMACIÓN CON PANTALLAS



Nuestro microcontrolador también puede comunicarse con nosotros mediante el empleo de pantallas o displays.

Estableceremos la programación de pantallas del tipo LCD, de las que existen una gran variedad de tipos y modelos.

En nuestro primer ejemplo, utilizaremos las más comunes, las pantallas LCD de 16x2 caracteres.

Como repaso diremos que, este tipo de dispositivo tiene hasta 16 pines de interconexión que se utilizan para alimentar la circuitería que contiene y encender un fondo de led que retro-ilumina la pantalla LCD y un bus de datos de 8 bits que sirve para comunicarse con el chip que posee. Aunque existen diferentes modelos y tipos, nosotros nos centraremos en las que poseen un circuito compatible con el Hitachi HD44780 que son estándar.

Este tipo de controlador posee una memoria RAM de 80 posiciones de 8 bits, donde cada posición se corresponde con una posición de la pantalla de fila 'x' y columna 'y'. El sistema revisa constantemente el contenido de esa memoria para leer un código ASCII que contiene, y lo representa en la pantalla.

Mediante un protocolo que utiliza el nivel lógico de tres de sus pines, EN, RS y RW, se comanda la pantalla para indicarle si el paquete de datos que vamos a enviarle por el bus de 8 bits, será un carácter o un comando.

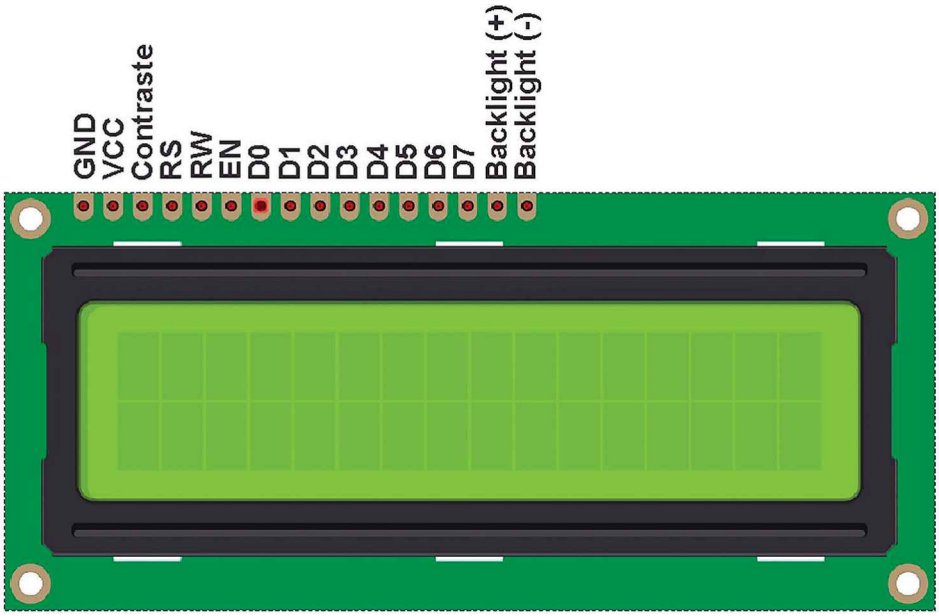


Figura 17.1

Lo más indicado es utilizar una librería para implementarla en nuestro código y así poder reutilizarla en otros proyectos.

17.1 EJEMPLO DE EMPLEO DE PANTALLAS LCD 16X2

En nuestro siguiente ejemplo, utilizamos una librería para mostrar simples mensajes en una pantalla LCD de este tipo.

No explicaremos aquí todo el funcionamiento de la librería que se muestra en el código siguiente, ya que no concierne a este libro el tratar los conceptos técnicos del funcionamiento de las pantallas LCD.

```
#include "stm32f10x.h"
#include "lcd4bit.h"
#include "delay.h"

/* Caracter definido por usu... ... ... CGRAM del LCD */
const char UserFont[8][8] =
{
{ 0x11,0x0A,0x04,0x1B,0x11,0x11,0x11,0x0E },
{ 0x10,0x10,0x10,0x10,0x10,0x10,0x10,0x10 },
{ 0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18 },
{ 0x1C,0x1C,0x1C,0x1C,0x1C,0x1C,0x1C,0x1C },
{ 0x1E,0x1E,0x1E,0x1E,0x1E,0x1E,0x1E,0x1E },
{ 0x1F,0x1F,0x1F,0x1F,0x1F,0x1F,0x1F,0x1F },
{ 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 },
{ 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 }
};

/*    Funcion que inicializa un LCD a 4 Bits    */
//-----------------------------------------------
void LCD_Init(void)
{
  int i;
  char const *p;

  // Configuracion de los pines de salida hacia el LCD
  GPIO_InitTypeDef GPIO_InitStructure;
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_EN, ENABLE);
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Pin   = LCD_EN_PIN;
  GPIO_Init(LCD_EN_PORT, &GPIO_InitStructure);

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_RW, ENABLE);
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Pin   = LCD_RW_PIN;
  GPIO_Init(LCD_RW_PORT, &GPIO_InitStructure);

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_RS, ENABLE);
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Pin   = LCD_RS_PIN;
  GPIO_Init(LCD_RS_PORT, &GPIO_InitStructure);

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_DATA, ENABLE);
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Pin = LCD_D4_PIN |
                                LCD_D5_PIN |
                                LCD_D6_PIN |
                                LCD_D7_PIN;
  GPIO_Init(LCD_DATA_PORT, &GPIO_InitStructure);

  delay_init(72);

  LCD_D4_HI();
  LCD_D5_HI();
  LCD_D6_LO();
  LCD_D7_LO();
  delay_ms(15);  // Delay de 15 ms

  LCD_D4_HI();
  LCD_D5_HI();
  LCD_D6_LO();
  LCD_D7_LO();
  LCD_pulse_EN();
  delay_us(4100); // Delay de 4.1 ms

  LCD_D4_HI();
  LCD_D5_HI();
  LCD_D6_LO();
  LCD_D7_LO();
  LCD_pulse_EN();
  delay_us(100); // Delay de 100 us

  LCD_D4_HI();
  LCD_D5_HI();
```

```
78    LCD_D6_LO();
79    LCD_D7_LO();
80    LCD_pulse_EN();
81
82    while(LCD_Busy()); // Espera hasta que el LCD
83    LCD_D4_LO();
84    LCD_D5_HI();
85    LCD_D6_LO();
86    LCD_D7_LO();
87    LCD_pulse_EN();
88
89    while(LCD_Busy());   // Espera hasta que se complete
90    LCD_Write_Cmd(0x28); // Establecemos LCD como (Datos 4-Bit,
91                                                N_lineas=2, Font=0 5X7)
92    LCD_Write_Cmd(0x0C); // Enciende el LCD sin cursor.
93    LCD_Write_Cmd(0x06); // Inicializa cursor
94
95    /* Cargamos el caracter definidos por usuario en la CGRRAM */
96    LCD_Write_Cmd(0x40); // Establece la direccion CGRAM desde 0
97    p = &UserFont[0][0];
98
99    for (i = 0; i < sizeof(UserFont); i++, p++)
100   LCD_Put_Char(*p);
101
102   LCD_Write_Cmd(0x80);
103 }
104
105 /*  Funcion que genera un Strobe en el LCD   */
106 //-------------------------------------------------
107 void LCD_Out_Data4(unsigned char val)
108 {
109   if((val&0x01) == 0x01) // Bit[0]
110   {
111     LCD_D4_HI();
112   }
113   else
114   {
115     LCD_D4_LO();
116   }
117
118   if((val&0x02) == 0x02) // Bit[1]
119   {
120     LCD_D5_HI();
121   }
122   else
123   {
124     LCD_D5_LO();
125   }
126
127   if((val&0x04)==0x04) // Bit[2]
128   {
129     LCD_D6_HI();
130   }
131   else
132   {
133     LCD_D6_LO();
134   }
135
136   if((val&0x08)==0x08) // Bit[3]
137   {
138     LCD_D7_HI();
139   }
140   else
141   {
142     LCD_D7_LO();
143   }
144
145 }
146
147 /*  Funcion que escribe 1 byte de datos en el LCD  */
148 //-------------------------------------------------
149 void LCD_Write_Byte(unsigned char val)
150 {
151   LCD_Out_Data4((val>>4)&0x0F);
152   LCD_pulse_EN();
153
154   LCD_Out_Data4(val&0x0F);
```

```
155   LCD_pulse_EN();
156
157   while(LCD_Busy());
158 }
159
160 /*  Funcion que escribe un comando en el LCD   */
161 //-----------------------------------------------
162 void LCD_Write_Cmd(unsigned char val)
163 {
164   LCD_RS_LO();          // pin RS = 0 (Seleccion de comando)
165   LCD_Write_Byte(val);
166 }
167
168 /*  Escribe un Caracter (ASCII) en el LCD      */
169 //-----------------------------------------------
170 void LCD_Put_Char(unsigned char c)
171 {
172   LCD_RS_HI();         // pin RS = 1 (Seleccion de caracteres)
173   LCD_Write_Byte(c);
174 }
175
176 /* Funcion que establece el cursor en una posicion de la pantalla LCD */
177 //-----------------------------------------------------------------------
178 void LCD_Set_Cursor(unsigned char line, unsigned char column)
179 {
180   unsigned char address;
181   column--;line--;
182   address = (line * 40) + column;
183   address = 0x80 + (address & 0x7F);
184   LCD_Write_Cmd(address);
185 }
186
187 /*  Funcion que envia una cadena de caracteres (ASCII) al LCD  */
188 //-------------------------------------------------------------
189 void LCD_Put_Str(char* str)
190 {
191   int i;
192
193   for (i=0;i<16 && str[i]!=0;i++)
194   {
195     LCD_Put_Char(str[i]);    // Envia un Byte al LCD
196   }
197 }
198
199 /*   Funcion que envia un caracter numerico al LCD  */
200 //-------------------------------------------------
201 void LCD_Put_Num(int num)
202 {
203   int i,j;
204   int p,f=0;
205   char ch[5];
206
207   for(i=0;i<5;i++)
208   {
209     p=1;
210     for(j=4-i;j>0;j--)
211       p = p*10;
212       ch[i] = (num/p);
213       if (num>=p && !f){
214         f=1;
215       }
216    num =num - ch[i] *p ;
217    ch[i] = ch[i] +48;
218    if(f) LCD_Put_Char(ch[i]);
219   }
220 }
221
222 /*  Funcion que provoca tiempos de espera en el LCD */
223 //-------------------------------------------------
224 char LCD_Busy(void)
225 {
226   GPIO_InitTypeDef GPIO_InitStructure;
227   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
228   GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN_FLOATING;
229   GPIO_InitStructure.GPIO_Pin   = LCD_D7_PIN;
230   GPIO_Init(LCD_DATA_PORT, &GPIO_InitStructure);
231
```

```
232   LCD_RS_LO();
233   LCD_RW_HI();
234   LCD_EN_HI();
235
236   delay_us(100);
237   if (GPIO_ReadInputDataBit(LCD_DATA_PORT, LCD_D7_PIN) == Bit_SET)
238   {
239     LCD_EN_LO();
240     LCD_RW_LO();
241
242     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
243     GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
244     GPIO_InitStructure.GPIO_Pin   = LCD_D7_PIN;
245     GPIO_Init(LCD_DATA_PORT, &GPIO_InitStructure);
246
247     return 1;
248   }
249   else
250   {
251     LCD_EN_LO();
252     LCD_RW_LO();
253
254     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
255     GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
256     GPIO_InitStructure.GPIO_Pin   = LCD_D7_PIN;
257     GPIO_Init(LCD_DATA_PORT, &GPIO_InitStructure);
258
259     return 0;
260   }
261 }
262
263 /*  Funcion que genera un pulso en el pin EN del LCD      */
264 //-------------------------------------------------------------
265 void LCD_pulse_EN(void)
266 {
267   delay_init(72);
268   LCD_EN_HI(); // Habilita pin EN ON
269   delay_us(50);
270   LCD_EN_LO(); // Habilita pin EN Off
271 }
272
273 /*   Funcion que muestra un caracter gráfico en el LCD
274      en 'value' el valor en su posicion CGRAM y en
275      'size' especificamos su tamaño                        */
276 //-------------------------------------------------------------
277 void LCD_BarGraphic (int value, int size)
278 {
279   int i;
280
281   value = value * size / 20; /* Divide de 5 en 5 8 pixels */
282   for (i = 0; i < size; i++)
283   {
284     if (value > 5) {
285       LCD_Put_Char (0x05);
286       value -= 5;
287     }
288   else {
289     LCD_Put_Char (value);
290
291     break;
292   }
293   }
294 }
295
296 /*    Funcion que muestra un caracter grafico en la pantalla LCD
297       especificando la posicion pos_x horizontal de inicio y la
298       posicion pos_y vertical de la pantalla LCD                */
299 //-------------------------------------------------------------
300 void LCD_BarGraphicXY (int pos_x, int pos_y, int value)
301 {
302   int i;
303
304   LCD_Set_Cursor(pos_x, pos_y);
305   for (i = 0; i < 16; i++)
306   {
307     if (value > 5) {
308       LCD_Put_Char (0x05);
309       value -= 5;
```

```
310     } else {
311       LCD_Put_Char (value);
312       while (i++ < 16) LCD_Put_Char (0);
313     }
314   }
315 }
316
```

Figura 17.2. Fichero de la librería lcd.c

A continuación se muestra el fichero de cabecera, de la librería anterior de manejo de un LCD:

```
1  ##ifndef __lcd_H
2  #define  __lcd_H
3
4  /*   Definimos los pines donde conectaremos la pantalla LCD   */
5  //--------------------------------------------------------------
6  // Definimos los pines a conectar con la pantalla LCD
7  #define LCD_RS_PIN GPIO_Pin_0   // Pin RS_LCD PB0
8  #define LCD_RW_PIN GPIO_Pin_1   // Pin RW_LCD PB1
9  #define LCD_EN_PIN GPIO_Pin_10  // Pin EN_LCD PB10
10
11 #define LCD_D4_PIN GPIO_Pin_12  // Pin DATA4_LCD PB12
12 #define LCD_D5_PIN GPIO_Pin_13  // Pin DATA5_LCD PB13
13 #define LCD_D6_PIN GPIO_Pin_14  // Pin DATA6_LCD PB14
14 #define LCD_D7_PIN GPIO_Pin_15  // Pin DATA7_LCD PB15
15
16 #define LCD_RS_PORT     GPIOB       // Puerto Pin RS GPIOB
17 #define LCD_RW_PORT     GPIOB       // Puerto Pin RW GPIOB
18 #define LCD_EN_PORT     GPIOB       // Puerto Pin EN GPIOB
19 #define LCD_DATA_PORT   GPIOB     // Puerto Pines DATAx GPIOB
20
21 #define RCC_APB2Periph_GPIO_RS RCC_APB2Periph_GPIOB // Reloj para pin RS_LCD APB2
22 #define RCC_APB2Periph_GPIO_RW RCC_APB2Periph_GPIOB // Reloj para pin RW_LCD APB2
23 #define RCC_APB2Periph_GPIO_EN RCC_APB2Periph_GPIOB // Reloj para pin EN_LCD APB2
24 #define RCC_APB2Periph_GPIO_DATA RCC_APB2Periph_GPIOB//Reloj para pines Data APB2
25
26 // Definimos que nombres para activar los pines a nivel 'High' y 'Low'
27 #define LCD_EN_HI() GPIO_WriteBit(LCD_EN_PORT,LCD_EN_PIN,Bit_SET)
28 #define LCD_EN_LO()     GPIO_WriteBit(LCD_EN_PORT,LCD_EN_PIN,Bit_RESET)
29 #define LCD_RW_HI() GPIO_WriteBit(LCD_RW_PORT,LCD_RW_PIN,Bit_SET)
30 #define LCD_RW_LO() GPIO_WriteBit(LCD_RW_PORT,LCD_RW_PIN,Bit_RESET)
31 #define LCD_RS_HI() GPIO_WriteBit(LCD_RS_PORT,LCD_RS_PIN,Bit_SET)
32 #define LCD_RS_LO() GPIO_WriteBit(LCD_RS_PORT,LCD_RS_PIN,Bit_RESET)
33 #define LCD_D4_HI() GPIO_WriteBit(LCD_DATA_PORT,LCD_D4_PIN,Bit_SET)
34 #define LCD_D4_LO() GPIO_WriteBit(LCD_DATA_PORT,LCD_D4_PIN,Bit_RESET)
35 #define LCD_D5_HI() GPIO_WriteBit(LCD_DATA_PORT,LCD_D5_PIN,Bit_SET)
36 #define LCD_D5_LO() GPIO_WriteBit(LCD_DATA_PORT,LCD_D5_PIN,Bit_RESET)
37 #define LCD_D6_HI() GPIO_WriteBit(LCD_DATA_PORT,LCD_D6_PIN,Bit_SET)
38 #define LCD_D6_LO() GPIO_WriteBit(LCD_DATA_PORT,LCD_D6_PIN,Bit_RESET)
39 #define LCD_D7_HI() GPIO_WriteBit(LCD_DATA_PORT,LCD_D7_PIN,Bit_SET)
40 #define LCD_D7_LO() GPIO_WriteBit(LCD_DATA_PORT,LCD_D7_PIN,Bit_RESET)
41
42 /*   Definimos los nombres de los comandos para el LCD                  */
43 //-------------------------------------------------------------------------
44 #define lcd_clear() lcd_write_cmd(0x01)          // Borra la pantalla
45 #define lcd_cursor_home() lcd_write_cmd(0x02)    // Establecer el cursor a 'Home'
46 #define lcd_display_on() lcd_write_cmd(0x0E)     // Pantalla LCD Activa
47 #define lcd_display_off() lcd_write_cmd(0x08)    // Pantalla LCD Inactiva
48 #define lcd_cursor_blink() lcd_write_cmd(0x0F)// Cursor intermitente
49 #define lcd_cursor_on() lcd_write_cmd(0x0E)    // Cursor visible activo
50 #define lcd_cursor_off() lcd_write_cmd(0x0C)   // Cursor inactivo
51 #define lcd_cursor_left() lcd_write_cmd(0x10) // Movimiento hacia la izquierda
52                                                  del cursor
53 #define lcd_cursor_right() lcd_write_cmd(0x14)// Movimiento hacia la derecha
54                                                  del cursor
55 #define lcd_display_sleft() lcd_write_cmd(0x18)// Movimiento a la izquierda
56                                                  de la pantalla
57 #define lcd_display_sright() lcd_write_cmd(0x1C)// Movimiento a la derecha
```

```
58                                                    d  la pantalla
59
60  // List  d  funciones ----------------------------------------------------
61  void lcd_out_data4(unsigned char val);
62  void lcd_write_byte(unsigned char val);
63  void lcd_write_cmd(unsigned char val);
64  void lcd_putchar(unsigned char c);
65  void lcd_init(void);
66  void set_cursor(unsigned char column, unsigned char line);
67  void lcd_putsf(char* str);
68  void lcd_puts(int num);
69  char busy_lcd(void);
70  void enable_lcd(void);
71  void lcd_bargraph (int value, int size);
72  void lcd_bargraphXY (int pos_x, int pos_y, int value);
73
74  endif /* __lcd_H */
```

Figura 17.3. Fichero lcd.h

Seguidamente, mostramos el contenido del fichero "delay.c" que contiene las funciones para producir los retrasos en nuestra programación.

```
1   #include "stm32f10x.h"
2   #include "delay.h"
3
4   static u8 fac_us=0;
5   static u16 fac_ms=0;
6
7   /*  Funcion  i  cializa el reloj  l  sistema */
8   //-------------------------------------------------
9   void delay_init(u8 SYSCLK)
10  {
11    SysTick->CTRL&=0xfffffffb;
12    fac_us=SYSCLK/8;
13    fac_ms=(u16)fac_us*1000;
14  }
15
16  /*  Funcion que genera un retraso en ms   */
17  //-----------------------------------------
18  void  _  _  (u16 nms)
19  {
20    u32 temp;
21    SysTick->LOAD=(u32)nms*fac_ms;
22    SysTick->VAL =0x00;
23    SysTick->CTRL=0x01 ;
24    do
25    {
26       temp=SysTick->CTRL;
27    }
28    while(temp&0x01&&!(temp&(1<<16)));
29    SysTick->CTRL=0x00;
30    SysTick->VAL =0X00;
31  }
32
33  /*  Funcion que genera un retraso en us   */
34  //-----------------------------------------
35  void delay_us(u32 nus)
36  {
37    u32 temp;
38    SysTick->LOAD=nus*fac_us;
39    SysTick->VAL=0x00;
40    SysTick->CTRL=0x01 ;
41    do
42    {
43      temp=SysTick->CTRL;
44    }
45    while(temp&0x01&&!(temp&(1<<16)));
46    SysTick->CTRL=0x00;
47    SysTick->VAL =0X00;
48  }
```

Figura 17.4. Fichero delay.c

Por último, mostramos el contenido del fichero principal "main.c" de nuestro ejemplo para el uso de una pantalla LCD:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "misc.h"

#include "delay.h"
#include "lcd.h"

/*            Funcion principal            */
//=========================================
int main(void)
{
  lcd_init();   // Inicializamos la libreria LCD

  lcd_cursor_on();    // Cursor visible activo
  lcd_cursor_blink(); // Cursor intermitente
  lcd_clear();        // Borra la pantalla

  set_cursor(1,0);
  delay_ms(100);
  lcd_putsf("PRUEBA DE LCD");

  set_cursor(2,0);
  delay_ms(100);
  lcd_putsf(" Esta es Linea 2");


while(1)
      {
      }
}
```

Figura 17.5. Fichero main.c

En este ejemplo, las conexiones de nuestra pantalla y el microcontrolador son las especificadas en el fichero de configuración "lcd.h" utilizando solo 4 bits del bus de datos; no obstante, son las indicadas a continuación:

PINES DEL LCD	Pines del STM32
Señal LCD RS	PB0
Señal LCD RW	PB1
Señal LCD EN	PB10
Señal LCD D4	PB12
Señal LCD D5	PB13
Señal LCD D6	PB14
Señal LCD D7	PB15

Hemos seleccionado aquí, los pines que corresponden a un lateral de nuestra placa de pruebas buscando la sencillez de conexiones y proximidad. Aunque se puede utilizar cualquier puerto o pin para su conexión; la única salvedad al respecto, es que los pines del puerto de datos deben ser del mismo puerto, ya que el programa utiliza la escritura en paralelo en todos los pines de ese puerto para enviar los datos hacia el display.

Con respecto a cada una de las funciones que se utilizan en el código, están sobradamente explicadas en el propio código.

17.2 EJEMPLOS DE EMPLEO DE PANTALLAS MEDIANTE ADAPTADOR I2C

Existe un adaptador con el que podemos utilizar nuestra pantalla LCD con solo 2 cables, mediante la comunicación I2C –que explicaremos más adelante-; lo cual reduce muchísimo el uso de todos los pines que suelen ser necesarios para el manejo de este tipo de pantallas.

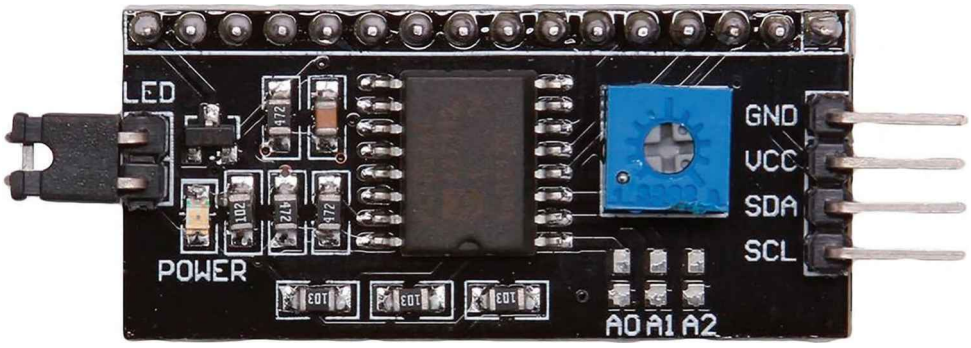


Figura 17.6

El adaptador I2C posee un integrado que permite comunicarnos con nuestra pantalla LCD, el PCF8574, que mediante comandos en I2C activan o desactivan cada uno de los pines de salida que posee. Estos, se conectan perfectamente con los pines de entrada de cualquier pantalla con el controlador HD44780. También tiene un potenciómetro con el que es posible adaptar el nivel de contraste del display, un jumper (LED) con el que se puede activar o desactivar los led de retroiluminación del display. Por defecto viene con un jumper puenteado para dejar activada la retroiluminación del display. Además, mediante una tira de 6 pines nos permite configurar la dirección I2C del dispositivo, que configura los tres bits de dirección del controlador, A0-A1-A2; en los que por defecto, sin soldar, todos los pines estarán a nivel alto; mientras que cuando se suelda alguno, se pasa a nivel bajo.

En la Figura 17.7 vemos un esquema común de conexión de este integrado con una pantalla LCD.

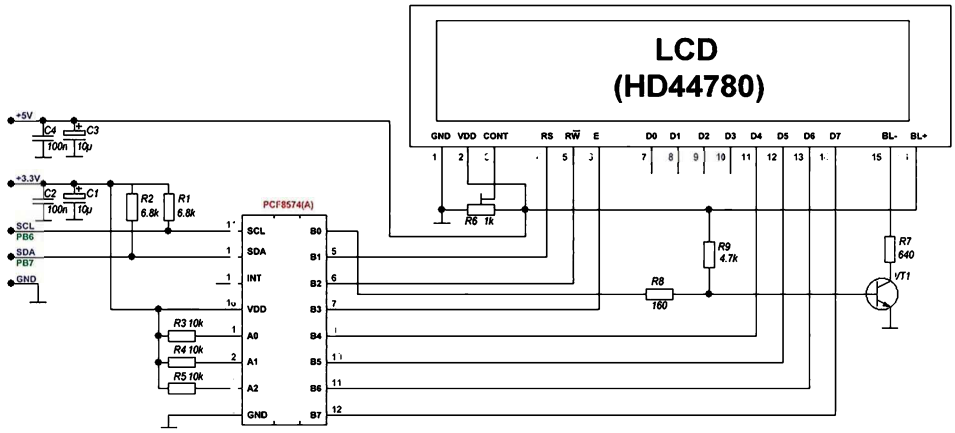


Figura 17.7

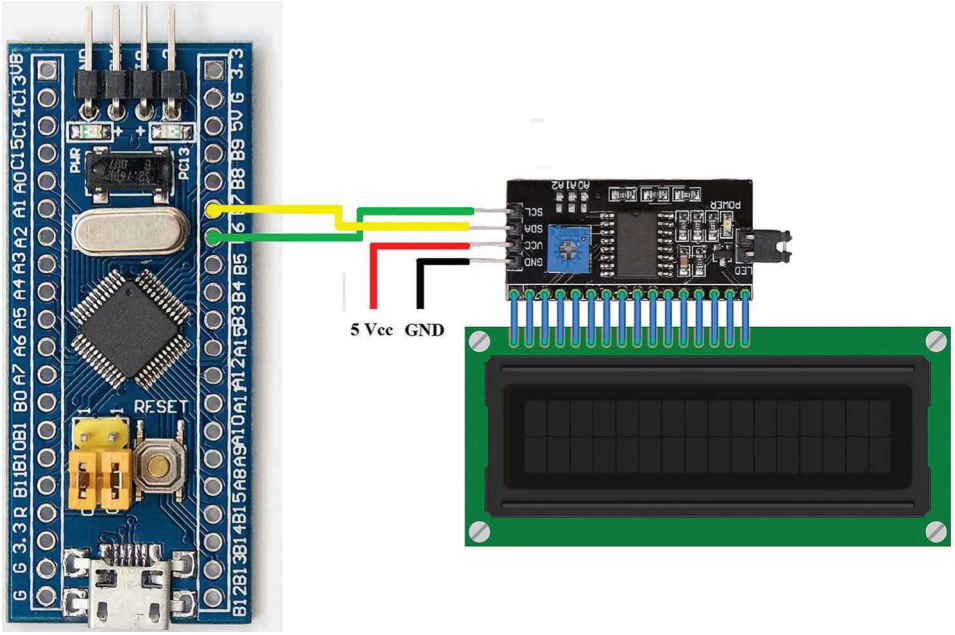


Figura 17.8

Al igual que con nuestro primer proyecto de ejemplo, utilizaremos una librería para el manejo de este tipo de adaptador con una pantalla LCD de las más

comunes; logrando utilizarla con solo dos pines de la comunicación I2C, el SDA y el SCL de nuestro microcontrolador.

La librería "**LiquidCristal**" para STM32, es un desarrollo original publicado en diciembre del 2017 en el portal "*https://habrahabr.ru/post/322184/*", que se puede descargar de la siguiente dirección en internet "*https://github.com/Vendict/STM32_LCD_I2C*".

Nuestro fichero "main.c" con el manejo de la pantalla mediante el adaptador I2C, se reproduce a continuación:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_i2c.h"

#include "delay.h"                // Cargamos la librería 'delay'
#include "I2C.h"                  // Cargamos la librería del 'I2C'
#include "LiquidCrystal_I2C.h"    // Cargamos la libreia del LCD por I2c

#include <string.h>

#define LCD_I2C_ADDR 0x38    // Establecemos la dirección del modulo I2c
#define LCD_I2C_Long   16    // Establecemos longitud de caracteres del LCD
#define LCD_I2C_Lines   4    // Establecemos el número de líneas del LCD

// Creamos una serie de caracteres para representar en el LCD
uint8_t note[8]  = {0x2,0x3,0x2,0xe,0x1e,0xc,0x0};
uint8_t clock[8] = {0x0,0xe,0x15,0x17,0x11,0xe,0x0};
uint8_t face[8]  = {0x0,0xa,0xa,0xa,0x0,0x11,0xe,0x0};
uint8_t heart[8] = {0x0,0xa,0x1f,0x1f,0xe,0x4,0x0};
uint8_t duck[8]  = {0x0,0xc,0x1d,0xf,0xf,0x6,0x0};
uint8_t check[8] = {0x0,0x1,0x3,0x16,0x1c,0x8,0x0};
uint8_t cross[8] = {0x0,0x1b,0xe,0x4,0xe,0x1b,0x0};
uint8_t retarrow[8] = {0x1,0x1,0x5,0x9,0x1f,0x8,0x4};

uint8_t batt000[8]  = {0x0E,0x1B,0x11,0x11,0x11,0x11,0x11,0x1F};
uint8_t batt010[8]  = {0x0E,0x1B,0x11,0x11,0x11,0x11,0x1F,0x1F};
uint8_t batt030[8]  = {0x0E,0x1B,0x11,0x11,0x11,0x1F,0x1F,0x1F};
uint8_t batt050[8]  = {0x0E,0x1B,0x11,0x11,0x1F,0x1F,0x1F,0x1F};
uint8_t batt070[8]  = {0x0E,0x1B,0x11,0x1F,0x1F,0x1F,0x1F,0x1F};
uint8_t batt090[8]  = {0x0E,0x1B,0x1F,0x1F,0x1F,0x1F,0x1F,0x1F};
uint8_t batt100[8]  = {0x0E,0x1F,0x1F,0x1F,0x1F,0x1F,0x1F,0x1F};
uint8_t battcrg[8]  = {0x0E,0x1F,0x1B,0x17,0x1D,0x1B,0x1F,0x1F};

/* Creamos una matiz que contendrá los nombres de los
   Símbolos creados para carga en la memoria del LCD   */
uint8_t const *simbol[] = {"bateria:","nota :  ","reloj : ","cara :
","corazon:","pato :  ","control:","cruz :  ","retorno:"};

/* Definimos una varible con el abecedario */
static char caracteres[] = "0123456789abcdefghijklmnopqrstuvwxyz";

/*              Modulo principal                                    */
//-----------------------------------------------------------------
int main(void)
{
  /*  Configuramos la Direccion y el tipo de pantalla LCD   */
  //---------------------------------------------------------------
  LCDI2C_init(LCD_I2C_ADDR, LCD_I2C_Long, LCD_I2C_Lines);

  LCDI2C_backlight(); // Encendemos la retroiluminacion en la pantalla
  LCDI2C_clear();     // Borramos el contenido de la pantalla

  LCDI2C_setCursor(0, 0);  // Cursor en columna 0 linea 1 de la pantalla
```

```
55   LCDI2C_write_String("- PRUEBA LCD I2C-");
56
57   LCDI2C_setCursor(0,1);   // Cursor en columna 0 linea 2 de la pantalla
58   LCDI2C_write_String("Linea 2 -ABCDEFGH");
59
60   // Cuando especificamos al principio una pantalla con mas líneas
61   if (LCD_I2C_Lines == 4)
62   {
63     LCDI2C_setCursor(0,2);    // Cursor en columna 0 linea 3 de la pantalla
64     LCDI2C_write_String("01234567890123456");
65     LCDI2C_setCursor(0,3);    // Cursor en columna 0 linea 4 de la pantalla
66     LCDI2C_write_String("abcdefghijklmnopq");
67   }
68   delay_ms(1000);
69   LCDI2C_clear();
70
71   //LLenamos la pantalla de caracteres de prueba
72   //--------------------------------------------------------------------------
73   char caracter[33];
74   for (int i=0; i < LCD_I2C_Lines; i++)
75   {
76     LCDI2C_setCursor(0,i);
77     LCDI2C_write_String(memcpy(caracter, caracteres, LCD_I2C_Long));
78   }
79   delay_ms(1000);
80
81   /* Ejemplo de desplazamiento de texto hacia la izquierda */
82   //--------------------------------------------------------------------------
83   LCDI2C_setCursor(0,1);
84   LCDI2C_write_String("MOVER A IZQUIERD");
85
86   for (int i=0; i < 20;i++)
87   {
88     LCDI2C_scrollDisplayLeft();
89     delay_ms(500);
90   }
91   delay_ms(1000);
92
93   /* Ejemplo de desplazamiento de texto hacia la derecha */
94   //--------------------------------------------------------------------------
95   LCDI2C_setCursor(0,1);
96   LCDI2C_write_String("MOVER A DERECHA ");
97
98   for (int i=0; i < 20;i++)
99   {
100    LCDI2C_scrollDisplayRight();
101    delay_ms(500);
102  }
103
104  /* LLenamos la pantalla de caracteres                    */
105  //--------------------------------------------------------------------------
106  delay_ms(1000);
107  LCDI2C_clear();
108  for (int i=0; i < LCD_I2C_Lines; i++)
109  {
110    LCDI2C_setCursor(0,i);
111    LCDI2C_write_String(memcpy(caracter, caracteres, LCD_I2C_Long));
112  }
113  delay_ms(1000);
114  LCDI2C_clear();
115
116  /*  Mostrar caracteres predefinidos por el usuario */
117  // Guardamos los simbolos de Bateria en la memoria de la pantalla LCD
118  LCDI2C_createChar(0, batt000);
119  LCDI2C_createChar(1, batt010);
120  LCDI2C_createChar(2, batt030);
121  LCDI2C_createChar(3, batt050);
122  LCDI2C_createChar(4, batt070);
123  LCDI2C_createChar(5, batt090);
124  LCDI2C_createChar(6, batt100);
125  LCDI2C_createChar(7, battcrg);
126
```

```
127   // Los mostramos en la pantalla
128   for (int i=0; i <= 7; i++)
129   {
130     LCDI2C_setCursor(0,0);
131     LCDI2C_write_String((char*) simbol[0]);
132
133     LCDI2C_setCursor(10,1);
134     LCDI2C_write(i);
135     delay_ms(1000);
136   }
137   delay_ms(1500);
138   LCDI2C_clear();
139
140   // Guardamos los otros simbolos
141   LCDI2C_createChar(0, note);
142   LCDI2C_createChar(1, clock);
143   LCDI2C_createChar(2, face);
144   LCDI2C_createChar(3, heart);
145   LCDI2C_createChar(4, duck);
146   LCDI2C_createChar(5, check);
147   LCDI2C_createChar(6, cross);
148   LCDI2C_createChar(7, retarrow);
149
150   // Los mostramos en la pantalla
151   for (int i=0; i < 8; i++)
152   {
153     LCDI2C_setCursor(0,0);
154     LCDI2C_write_String((char*) simbol[i+1]);
155     LCDI2C_setCursor(10,1);
156     LCDI2C_write(i);
157     delay_ms(1000);
158   }
159   LCDI2C_clear();
160
161 }
```

Figura 17.9

17.3 EJEMPLO DE EMPLEO DE PANTALLAS OLED

Otro tipo de pantallas son las de tipo **OLED** (*Organic Light-Emitting Diode*) con un desarrollo notable en nuestros días, ya que su principal novedad respecto a las pantallas TFT, es que están fabricadas con un tipo de diodo OLED -Diodo Orgánico de Emisión de Luz-, que como su nombre indica son de base orgánica y que al aplicarles corriente eléctrica generan su propia luz.

Desde sus comienzos en 1970, su tecnología se ha desarrollado mucho ya que, además de la propiedad que le da su nombre tiene una estructura maleable, lo que permitirá muy pronto que contemos con pantallas flexibles. Algunos fabricantes ya disponen de proyectos que han mostrado en algunas ferias.

Gracias a su proliferación y fuerte comercialización, es posible utilizarlas también para los proyectos con microcontrolador.

Existen diferentes modelos y tamaños, nosotros nos ocuparemos de las del controlador CMOS SSD1306 y matriz de RAM de 128x64 bits o puntos, para pantallas orgánicas del tipo de 0.96 pulgadas monocromo y con conexión I2c, como las mostradas en la Figura 17.10.

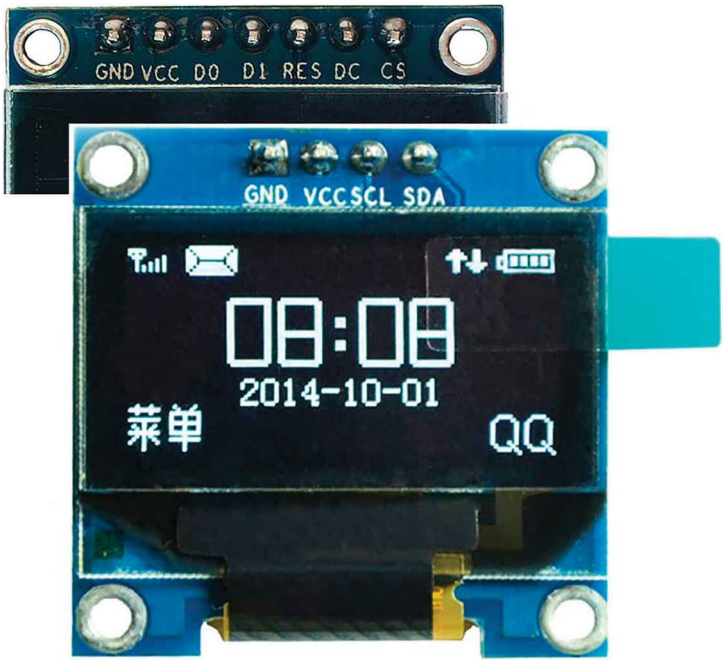


Figura 17.10

Como se observa en la Figura 17.10, existen pantallas con dos tipos de conexiones: de 7 y de 4 pines; lo que obliga a utilizar diferentes configuraciones para el manejo de la señal I2C, y por tanto, también diferentes librerías para usarlas en nuestra programación. En el libro presentamos un ejemplo de las de 4 pines.

Para el empleo de este tipo de pantallas, podemos utilizar una librería que se basa en el proyecto que se presentó en el entorno "https://habrahabr.ru/post/313490/" y cuya librería podemos descargar de la siguiente dirección: "*https://github.com/SL-RU/stm32libs*".

Aunque en este caso, la librería está creada bajo el entorno del STMCubeMx, que utiliza las librerías STM32F1xx_HAL_Driver para el manejo de los diferentes periféricos de nuestro microcontrolador.

El STMCubeMX, también es un entorno que suministra la casa STMicroelectronics; se trata de una herramienta que genera automáticamente el código en C utilizando asistentes y pantallas gráficas, que al final crea un proyecto completo en Keil que puede ser gestionado, editado y compilado como un programa más del Keil, aunque utilizando una estructura diferente con respecto a las librerías y comandos a utilizar.

Casi todos los proyectos creados con esta utilidad, crean una serie de carpetas automáticamente, tal como mostramos en la Figura 17.11.

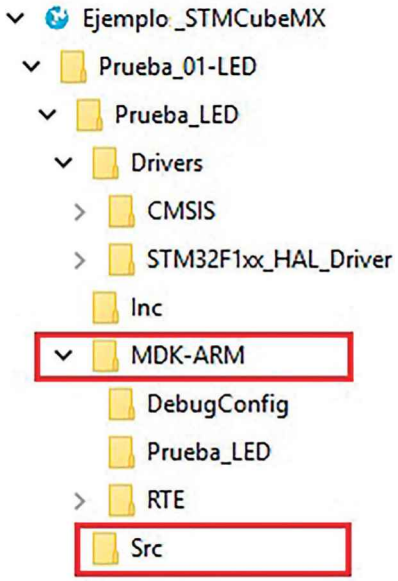


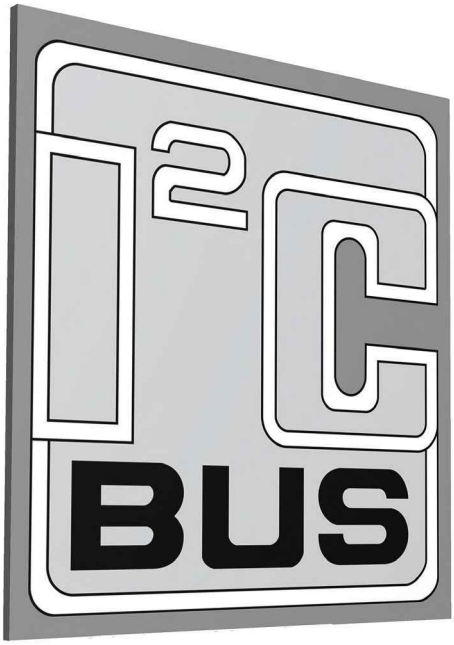
Figura 17.11.

En esta estructura, en la carpeta "MDK-RM" encontraremos el ejecutable del proyecto para iniciarlo con nuestro Keil MDK y en la carpeta "Src", algunos de los ficheros *.c con el código fuente.

Una vez nos descargamos el fichero "stm32libs-master.zip" y descomprimimos su contenido, se nos crean una serie de carpetas, donde en el interior de la carpeta "\HAL\ssd1306\ssd1306\MDK-ARM", encontraremos el proyecto "ssd1306" que utilizaremos como base para comprobar el funcionamiento de este tipo de pantalla.

18

PROGRAMACIÓN I2C



El **I2C** (*Inter-Integrated Circuit*), es un sistema de comunicación que se emplea en una gran cantidad de dispositivos como memorias, sensores y otros circuitos integrados. Se basa en un protocolo de comunicación serial, ya que solo utiliza un cable para transmitir la información bit a bit y de forma síncrona -la información se envía y se recibe por el mismo cable-. Posee dos líneas, el **SCL** y el **SDA** que deben estar conectadas en todos los dispositivos que se van a comunicar.

Permite interconectar hasta 112 dispositivos a la vez, que serán reconocidos por medio de un número de dirección independiente. Debe existir un dispositivo **maestro** y uno o más **esclavos**. Su velocidad puede variar, aunque está entre 100 y 400 kHz.

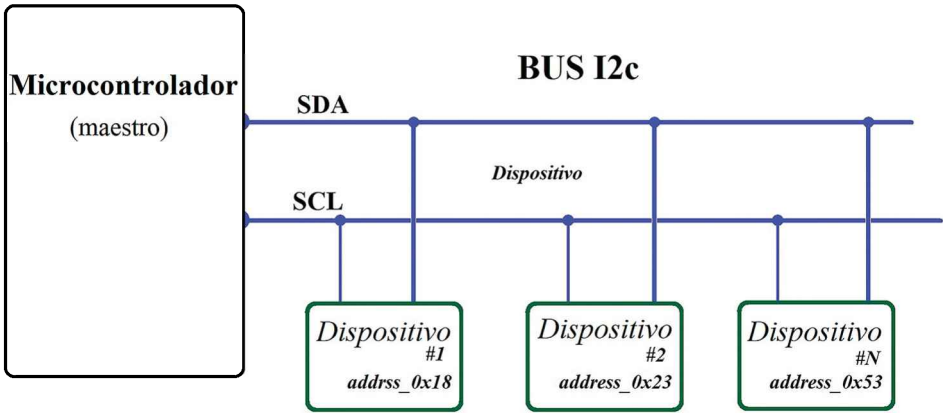


Figura 18.1

Las tramas de información a enviar deben cumplir un protocolo; en el que debe existir primero una condición de inicio de la comunicación (Start), el envío de la información en sí y una condición de finalización (Stop).

La Condición de Inicio (***Start***), se produce cuando la línea del bus **SDA** se pasa de un nivel alto a un nivel bajo, antes de que la otra línea en el bus, la **SCL,** cambie de nivel alto a nivel bajo.

La Condición de Paro (**Stop**) se origina cuando tras enviar los datos de la comunicación, la línea **SDA** cambia de un nivel bajo a nivel alto después de que la línea **SCL** cambie de nivel bajo a nivel alto.

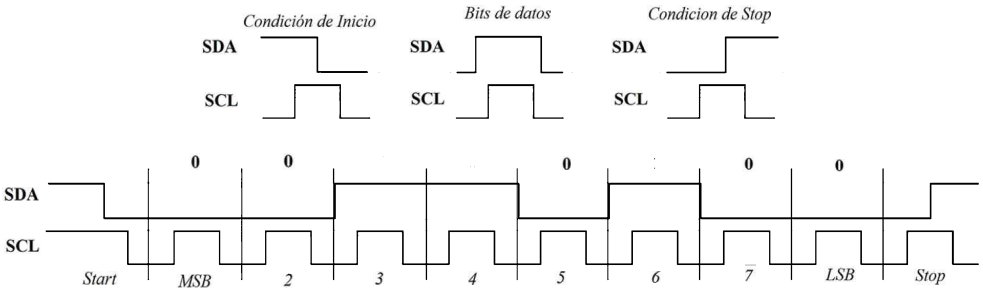


Figura 18.2

La información a transmitir dentro de las tramas irá en paquetes que contienen la dirección del dispositivo al que va dirigida la información y paquetes que configuran o comandan el dispositivo de acuerdo con sus especificaciones. La primera trama de información que se transmite, contiene solo siete bits y corresponderá a la dirección del periférico "esclavo" que debe recibirla; en el que su octavo bit, le indicará qué operación se requiere realizar con él: un '1' requiere lectura y un '0', escritura.

Cuando las dos líneas del bus están en nivel alto durante todo el tiempo, indica que dicho bus está libre para que cualquier dispositivo "maestro" pueda ocuparlo, estableciendo este, previamente la condición de inicio que indicamos antes.

El dispositivo al que corresponda la dirección que se ha transmitido, enviará un bit en nivel bajo inmediatamente después del octavo bit (R/W) que se ha enviado por el bus; este bit de reconocimiento (**ACK-** *acknowledge*), lo recibirá el dispositivo "maestro" como confirmación de que se reconoce el dispositivo destino y que está en condiciones de recibir la información. Cuando la transferencia es del "esclavo" al "maestro", el octavo bit en la trasmisión estará en nivel alto. El dispositivo maestro generará una señal de frecuencia o pulsos para que el dispositivo "esclavo" pueda enviar los datos y será entonces, cuando el dispositivo "maestro" genera la señal de reconocimiento (ACK).

En nuestros siguientes códigos de ejemplos, veremos cómo la librería CMSIS de manejo del I2C de los microcontroladores STM32, la "stm32f10x_i2c.h", nos permite trabajar con este periférico de forma simple.

También señalar que, el microcontrolador que posee nuestra placa de pruebas tiene dos puertos especializados: **I2C1** con los pines **SCL** en **PB6** y **SDA** en **PB7**; y el puerto **I2C2** con los pines **SCL** en **PB10** y **SDA** en **PB11**.

Aunque también es posible re-mapear los pines PB8 como SCL y PB9 como SDA del I2C1.

No obstante, como hemos visto al principio de este apartado, básicamente la comunicación I2C se basa en el funcionamiento y control de niveles altos y bajos, de dos líneas del bus durante unos tiempos determinados, para que se reconozca la información por parte de un dispositivo. Veremos, que existen librerías específicas para el control de determinados dispositivos que, utilizando tan solo los niveles lógicos en otro pines -los predefinidos para estos puertos- se reproduce el protocolo mediante software.

18.1 EJEMPLO DE EMPLEO DE UNA EEPROM I2C

Las memorias EEPROM (*Erasable Programmable Read Only Memory*), son dispositivos que nos permiten almacenar datos de forma permanente sin que se borren cuando se les retira la alimentación. Pueden grabarse o borrarse solo de forma eléctrica hasta 4 millones de ciclos de escritura o borrado y pueden almacenar la información durante 200 años.

No todos los chips de este tipo se manejan igual respecto a su direccionamiento y control.

Por ejemplo, cuando empleamos eeprom del tipo 24C00, la trama inicial que indicará su direccionamiento en el bus I2C será del formato "1|0|1|0|X|X|X|(R/W)"; donde como vimos al principio, el octavo bit le indica si queremos realizar un proceso de lectura valor '1' o de escritura, valor '0'. Los valores de los bit que están en la posición de la 'X' no tienen relevancia, ya que este tipo de eeprom no posee pines para establecerles una dirección en un bus de datos.

Mientras que en las eeprom del tipo 24C01/02/04/08/16, el formato de direccionamiento será de "1|0|1|0|**A2**|**A1**|**A0**|(R/W)", donde los cuatro bits iniciales son los mismos para todas ellas y los otros tres (**A2**|**A1**|**A0**), corresponden a la dirección de la eeprom que ya se ha establecido en los pines del propio chip. El octavo bit se utilizará para indicar el proceso que queremos realizar con el dispositivo.

Se pueden conectar varias eeprom en un mismo bus para sumar su capacidad. Aunque en el caso de los dos últimos tipos, las 24C08 y 24C16, solo se puede conectar y direccionar una sola en un solo bus; sí es posible mediante esos bits A2-A1-A0, indicar la dirección de la página o bloque a utilizar.

Respecto a su capacidades, tenemos que una eeprom de 24C08 posee una estructura de cuatro páginas de 256 Bytes; lo que nos da 4x256x8 bits = 1 Kbyte = 8Kbit; y la 24C32, contiene 16 páginas de 256 Bytes, de la que obtendremos 16x256x8 = 32768 bits ~ 36 Kilobytes.

Todo ello deberemos tenerlo en cuenta cuando programemos estos dispositivos.

En el siguiente ejemplo, utilizamos los comandos establecidos en la librería CMSIS para su utilización.

Primero, como siempre, creamos la estructura con la configuración de pines y puertos que vamos a emplear. Creamos una función que contenga todos los parámetros de inicialización necesarios para nuestro puerto I2C1.

```
1   /*   Funcion que inicializa GPIO e I2C1    */
2   //-----------------------------------------
3   void I2C1_EE_Init(void)
4   {
5      GPIO_InitTypeDef GPIO_InitStructure;
6
7      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE);
8
9      // Configuracion de GPIO para SCL y SDA
10     GPIO_InitStructure.GPIO_Pin        = GPIO_Pin_6 | GPIO_Pin_7;
11     GPIO_InitStructure.GPIO_Speed      = GPIO_Speed_50MHz;
12     GPIO_InitStructure.GPIO_Mode       = GPIO_Mode_AF_OD;
13     GPIO_InitStructure.GPIO_Speed      = GPIO_Speed_50MHz;
14     GPIO_Init(GPIOB, &GPIO_InitStructure);
15         .
16         .
17         .
```

Figura 18.3

En donde configuramos los pines PB6 y PB7 que corresponden a las líneas SCL y SDA del puerto I2C1.

Dentro de la misma función, creamos también la configuración para nuestro puerto I2C.

```
1          .
2          .
3          .
4      // Configuracion de parametros del I2C_1
5      I2C_DeInit(I2C1);
6      I2C_InitStructure.I2C_Mode          = I2C_Mode_I2C;
7      I2C_InitStructure.I2C_DutyCycle     = I2C_DutyCycle_2;
8      I2C_InitStructure.I2C_OwnAddress1   = 0x00;
9      I2C_InitStructure.I2C_Ack           = I2C_Ack_Enable;
10     I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
11     I2C_InitStructure.I2C_ClockSpeed    = 100000;
12
13     I2C_Init(I2C1, &I2C_InitStructure);
14     I2C_Cmd(I2C1, ENABLE);
15 }
```

Figura 18.4

Los parámetros que debemos configurar son:

Primero, mediante el comando "**I2C_DeInit**(I2Cx)", damos la orden de resetear los registros del periférico I2C de sus valores predeterminados al inicio.

Con el comando "**I2C_Mode**", establecemos el modo de funcionamiento de la comunicación del bus. El módulo principal de nuestro microcontrolador soporta múltiples modos de comunicación serie, los cuales podrán ser: __I2c, __SMBusDevice o __SMBusHost.

Después, el comando "**I2C_DutyCycle**", establece el ciclo de trabajo del bus respecto a los tiempos en que permanece la señal en modo alto y bajo, para que se cumpla la especificación del bus I2C. Puede ser:

- **I2C_DutyCycle_16_9** – Bus I2C modo rápido en tLOW / tHIGH = 16/9.
- **I2C_DutyCycle_2** – Bus I2C modo rápido en tLOW/tHIGH = 2/2.

Con el comando "**I2C_OwnAddress1**", especificamos la dirección que tendrá nuestro microcontrolador en el bus I2C.

Con el comando "**I2C_Ack**", activamos o desactivamos la generación de la señal de acuse de recibo (*Acknowledgement*) de la información transmitida.

El comando "**I2C_ AcknowledgedAddress**", indica si las direcciones a manejar en la comunicación I2C serán de 7 o 10 bits.

El último parámetro será establecer la velocidad de transferencia de la comunicación I2C, que se configura con el comando "**I2C_ ClockSpeed**", y en el que establecemos un valor en hercios (Hz) que será inferior al límite máximo de 400 kHz; que dependerá del reloj principal de nuestra placa y del tipo de microcontrolador.

A continuación con el comando "**I2C_Init**(I2Cx)", inicializamos el módulo I2C 'x' con los parámetros configurados en la estructura creada; y, con el comando "**I2C_Cmd**(I2Cx, ENABLE)", damos la orden de iniciarlo.

El siguiente paso, será establecer las funciones que generen la lectura y escritura en nuestra eeprom.

La función que creamos con los comandos que generan la escritura, en nuestro ejemplo la llamamos "**I2C_eprom_Write**", será:

```
1  /*  Funcion que escribe un byte en la eeprom                */
2  //-------------------------------------------------------------
3  int I2C_eeprom_Write(I2C_TypeDef* I2Cx, uint8_t HWaddress, uint8_t DataByte,
4                                                              uint8_t WRAddress)
5  {
6      /* Esperamos a que el bus I2c este operativo*/
7  (1) while(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));
8
9      /* Establecemos condicion de START                                  */
10 (2) I2C_GenerateSTART(I2Cx, ENABLE);
11
12     /* Comprobamos si el registro EV5 confirma el bus libre             */
13 (3) while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));
14
15     /* Enviamos la direccion de la EEPROM para escribir                 */
16 (4) I2C_Send7bitAddress(I2C1, HWaddress << 1, I2C_Direction_Transmitter);
17
18     /* Comprobamos que el dispositivo 'esclavo' confirma la comunicacion */
19 (5) while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED)),
20
21     /* Enviamos la direccion interna de la EEPROM para escribir         */
22 (6) I2C_SendData(I2Cx, WRAddress);
23
24     /* Comprobamos que el dispositivo 'esclavo' confirma la comunicacion */
25 (7) while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
26
27     /* Enviamos un bytye de datos para escribir en la EEPROM            */
28 (8) I2C_SendData(I2Cx, DataByte);
29
30     /* Comprobamos que el dispositivo 'esclavo' confirma la comunicacion */
31 (9) while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
32
33     /* Send STOP condition */
34 (10)I2C_GenerateSTOP(I2Cx, ENABLE);
35
36 }
```

Figura 18.5

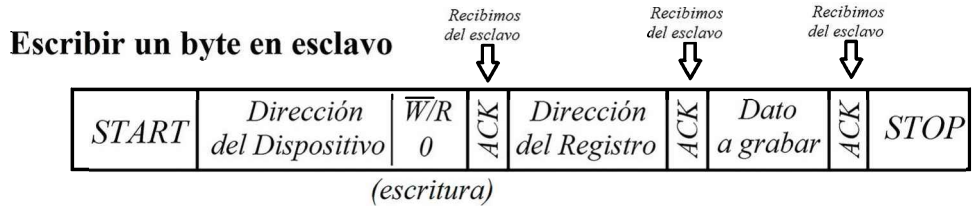


Figura 18.6

En el formato de un código para escribir un valor en un periférico I2C, se sigue el protocolo que aparece en la Figura 18.5; donde los pasos a seguir serán:

1. Comprobar si el bus I2C está libre para utilizarlo. (**I2C_GetFlagStatus**())
2. Establecer una condición inicial de START. (**I2C_GenerateStart**())
3. Comprobar si el bus I2C está libre. (**I2C_CheckEvent**())
4. Enviar la dirección del periférico esclavo. (**I2C_Send7bitAddress**())
5. Comprobar si el periférico esclavo confirma la selección y ha enviado un ACK. (**I2C_CheckEvent**())
6. Enviar la dirección del registro o memoria del periférico esclavo al que queremos escribir. (**I2C_SendData**())
7. Comprobar que el periférico confirma la recepción del dato de dirección enviada. (**I2C_CheckEvent**())
8. Enviar el dato a guardar por el periférico esclavo. (**I2C_SendData**())
9. Comprobar que el periférico confirma la recepción del dato enviado. (**I2C_CheckEvent**())
10. Establecer una condición de final o STOP de la transmisión. (**I2C_GenerateStop**())

Para crear una función que lea un dato determinado en nuestro periférico esclavo, se seguirán los pasos de la Figura 18.7.

```
1   /*  Funcion que Lee un byte de la eeprom                          */
2   //--------------------------------------------------------------
3   int I2C_eprom_Read(I2C_TypeDef* I2Cx, uint8_t HWaddress, uint8_t WRAddress)
4   {
5       /* Creamos la variable que contendrá el dato leído          */
6       uint8_t DataByte = 0;
7
8       /* Esperamos a que el bus I2c este operativo*/
9   (1) while(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));
10
11      /* Establecemos condicion de START */
12  (2) I2C_GenerateSTART(I2Cx, ENABLE);
13
14      /* Comprobamos si el registro EV5 confirma el bus libre */
15  (3) while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));
16
17      /* Enviamos direccion del dispotivo 'esclavo'               */
18  (4) I2C_Send7bitAddress(I2Cx, HWaddress <<1, I2C_Direction_Transmitter);
19
20      /* Comprobamos que el dispositivo 'esclavo' confirma la comunicacion */
21  (5) while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
22
23      /* Enviamos la direccion interna de la EEPROM para leer          */
24  (6) I2C_SendData(I2Cx, WRAddress);
25
26      /* Test on EV8 and clear it */
27  (7) while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
28
29      /* Regeneramos una nueva condicion de START */
30  (8) I2C_GenerateSTART(I2Cx, ENABLE);
31
32      /* Comprobamos si el registro EV5 confirma el bus libre */
33  (9) while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));
34
35      /* Enviamos direccion del dispositivo 'esclavo'                */
36  (10)I2C_Send7bitAddress(I2Cx, HWaddress <<1, I2C_Direction_Receiver);
37
38      /* Comprobamos que el dispositivo 'esclavo' confirma la comunicacion */
39  (11)while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
40
41      /* Desabilitamos la señal de ACK (Acknowledgement) */
42  (12)I2C_AcknowledgeConfig(I2Cx, DISABLE);
43
44      /* Esperamos a que se genere el evento EV7 para leer los datos */
45  (13)while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_RECEIVED));
46
47      /* Leemos un byte de la EEPROM */
48  (14)DataByte = I2C_ReceiveData(I2Cx);
49
50      /* Habilitar ACK para estar listo para otra recepción */
51      //I2C_AcknowledgeConfig(I2C1, ENABLE);
52
53      /* Establecemos condicion de STOP */
54  (15)I2C_GenerateSTOP(I2Cx, ENABLE);
55
56     return DataByte;
57  }
58
```

Figura 18.7

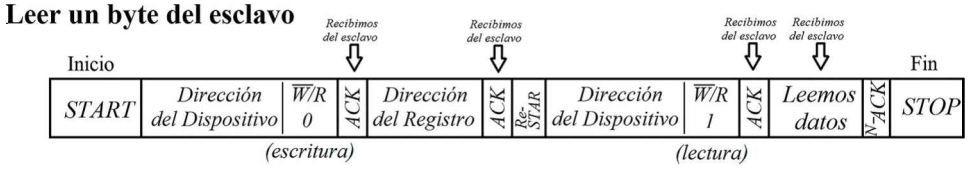


Figura 18.8

En la lectura de un valor de un periférico I2C se sigue el protocolo que aparece en la Figura 18.7; los pasos a seguir son:

1. Comprobar si el bus I2C está libre para utilizarlo. (**I2C_GetFlagStatus**())
2. Establecer una condición inicial de START. (**I2C_GenerateStart**())
3. Comprobar si el bus I2C está libre. (**I2C_CheckEvent**())
4. Enviar la dirección del periférico esclavo. (**I2C_Send7bitAddress**())
5. Comprobar si el periférico esclavo confirma la selección y ha enviado un ACK. (**I2C_CheckEvent**())
6. Enviar la dirección del registro o memoria del periférico esclavo del que queremos leer. (**I2C_SendData**())
7. Comprobar si el periférico esclavo confirma la selección y ha enviado un ACK. (**I2C_CheckEvent**())
8. Enviar la dirección del periférico esclavo. (**I2C_Send7bitAddress**())
9. Comprobar si el periférico esclavo confirma la selección y ha enviado un ACK. (**I2C_CheckEvent**())
10. Deshabilitar la señal de ACK. (**I2C_AcknowledgeConfig**())
11. Esperar a que se genere la condición de leer datos del esclavo. (**I2C_CheckEvent**())
12. Leer los datos y guardar en una variable que previamente habremos creado. (**I2C_ReceiveData**())
13. Establecer una condición de final o STOP de la transmisión. (**I2C_GenerateStop**())

Por último creamos la opción en la que se devuelva el valor leído con el comando *return*.

Reproducimos a continuación el contenido completo de la librería que contendrán las funciones de lectura y escritura de un dato de prueba en un módulo eeprom externo.

```
#include "i2c_eeprom.h"

/*   Funcion que inicializa GPIO e I2C1    */
//-----------------------------------------
void I2C1_EE_Init(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;
   I2C_InitTypeDef I2C_InitStructure;

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE);
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);

   // Configuracion de GPIO para SCL y SDA
   GPIO_InitStructure.GPIO_Pin        = GPIO_Pin_6 | GPIO_Pin_7;
   GPIO_InitStructure.GPIO_Speed      = GPIO_Speed_50MHz;
   GPIO_InitStructure.GPIO_Mode       = GPIO_Mode_AF_OD;
   GPIO_InitStructure.GPIO_Speed      = GPIO_Speed_50MHz;
   GPIO_Init(GPIOB, &GPIO_InitStructure);

   // Configuracion de parametros del I2C_1
   I2C_DeInit(I2C1);
   I2C_InitStructure.I2C_Mode         = I2C_Mode_I2C;
   I2C_InitStructure.I2C_DutyCycle    = I2C_DutyCycle_2;
   I2C_InitStructure.I2C_OwnAddress1  = 0x00;
   I2C_InitStructure.I2C_Ack          = I2C_Ack_Enable;
   I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
   I2C_InitStructure.I2C_ClockSpeed   = 100000;

   I2C_Init(I2C1, &I2C_InitStructure);
   I2C_Cmd(I2C1, ENABLE);
}

/*  Funcion que Lee un byte d[illegible]n                          */
//-----------------------------------------------------------------
int I2C_eeprom_Read(I2C_TypeDef* I2Cx, uint8_t HWAddress, uint8_t WRAddress)
{
    // Creamos la variable que contendrá el dato leido
    uint8_t DataByte = 0;

    /* Esperamos a que el bus I2c este operativo*/
    while(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));

    /* Establecemos condicion de START */
    I2C_GenerateSTART(I2Cx, ENABLE);

    /* Compro[illegible] si el registro EV5 confirma el bus libre */
    while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

    /* Enviamos direccio[illegible] del dispotivo 'esclavo'          */
    I2C_Send7bitAddress(I2Cx, HWAddress<<1, I2C_Direction_Transmitter);

    /* Comprobamos que el dispositivo 'esclavo' confirma la comunicacion */
    while (!I2C_CheckEvent(I2Cx,
I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

    /*  Borramos la bandera del registro EV6 para reiniciar el registro [illegible] */
    //I2C_Cmd(I2C1, ENABLE);

    /* Envia[illegible] la direccion interna de la [illegible] para leer      */
    I2C_SendData(I2Cx, WRAddress);

    /* Test on EV8 and clear it */
    while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));

    /* Regeneramos una nueva condicion de START */
```

```
66      I2C_GenerateSTART(I2Cx, ENABLE);
67
68      /* Comprobamos si el registro EV5 confirma el bus libre */
69      while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));
70
71      /* Enviamos direccion del dispositivo 'esclavo' */
72      I2C_Send7bitAddress(I2Cx, HWAddress<<1, I2C_Direction_Receiver);
73
74      /* Comprobamos que el dispositivo 'esclavo' confirma la comunicacion */
75      while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
76
77      /* Disabilitamos la señal de ACK (Acknowledgement) */
78      I2C_AcknowledgeConfig(I2Cx, DISABLE);
79
80      /* Esperamos a que se genere el evento EV7 para leer los datos */
81      while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_RECEIVED));
82
83      /* Leemos un byte de la EEPROM */
84      DataByte = I2C_ReceiveData(I2Cx);
85
86      /* Establecemos condicion de STOP */
87      I2C_GenerateSTOP(I2Cx, ENABLE);
88
89    return DataByte;
90  }
```

Figura 18.9

En caso de que necesitemos utilizar el segundo puerto de comunicación I2C, reproducimos a continuación una configuración de la función con la opción de utilizar uno u otro puerto y la velocidad de funcionamiento del puerto seleccionado:

```
1   void I2C_init(I2C_TypeDef* I2Cx, uint32_t speed)
2   {
3      GPIO_InitTypeDef GPIO_InitStructure;
4      I2C_InitTypeDef I2C_InitStructure;
5
6      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE);
7      RCC_APB1PeriphClockCmd(I2Cx == I2C1 ? RCC_APB1Periph_I2C1 :
8                                            RCC_APB1Periph_I2C2, ENABLE);
9
10     // Configuracion de GPIO para SCL y SDA dependiendo del puerto
11     if (I2Cx == I2C1)
12     {//  ----------------------------      SCL_Pin      SDA_Pin
13        GPIO_InitStructure.GPIO_Pin     = GPIO_Pin_6 | GPIO_Pin_7;
14     } else {
15        GPIO_InitStructure.GPIO_Pin     = GPIO_Pin_10 | GPIO_Pin_11;
16     }
17
18     GPIO_InitStructure.GPIO_Speed      = GPIO_Speed_50MHz;
19     GPIO_InitStructure.GPIO_Mode       = GPIO_Mode_AF_OD;
20     GPIO_Init(GPIOB, &GPIO_InitStructure);
21
22     I2C_DeInit(I2Cx);
23     I2C_InitStructure.I2C_Mode         = I2C_Mode_I2C;
24     I2C_InitStructure.I2C_DutyCycle    = I2C_DutyCycle_2;
25     I2C_InitStructure.I2C_OwnAddress1  = 0x00;
26     I2C_InitStructure.I2C_Ack = I2C_Ack_Disable;
27     //I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
28     I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
29     I2C_InitStructure.I2C_ClockSpeed   = speed;
30
31     I2C_Init(I2Cx, &I2C_InitStructure);
32     I2C_Cmd(I2Cx, ENABLE);
33  }
```

Figura 18.10

Como repaso a lo que hemos visto con el ejemplo de la eeprom, si nos fijamos ahora en el anterior código de ejemplo del capítulo anterior -donde configurábamos el puerto I2C de nuestro microcontrolador para utilizar una pantalla LCD con el adaptador de I2C- vemos que, cuando queríamos enviar un comando a la pantalla a través del adaptador, los pasos que se producían, por ejemplo, en la función "**LCDI2C_expanderWrite()**" eran los de la Figura 18.11.

```
void LCDI2C_expanderWrite(uint8_t _data){
  I2C_StartTransmission(I2C1, I2C_Direction_Transmitter, LCD2c_InitStructure.Addr);
  I2C_WriteData(I2C1, (int)_data)|LCD2c_InitStructure.backlightval);
  I2C_GenerateSTOP(I2C1, ENABLE);
}
```

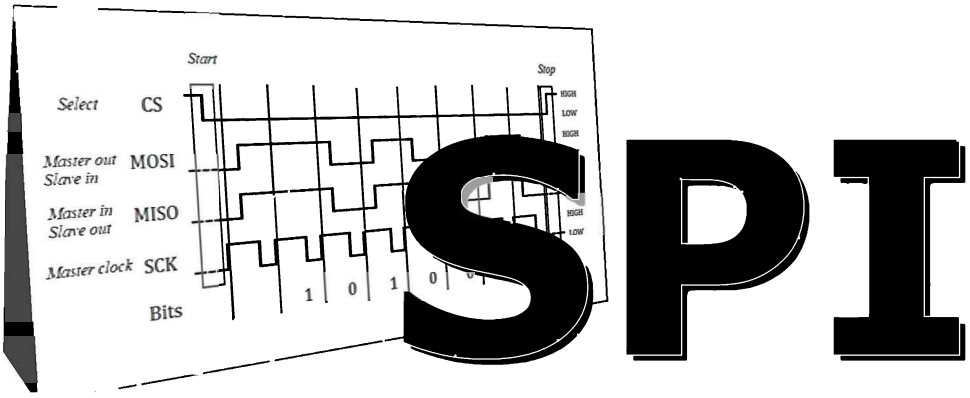
Figura 18.11

En la que podemos ver, que los pasos y comandos empleados cumplen el protocolo que se ha explicado anteriormente:

1. Esperar a que el bus I2C este operativo. (I2C_GetFlagStatus())
2. Generar una condición de inicio de transmisión. (I2C_GenerateStart())
3. Esperar a que el bus esté libre. (I2C_CheckEvent())
4. Enviar la dirección del adaptador. (I2C_Send7bitAddress())
5. Esperar la confirmación del adaptador. (I2C_CheckEvent())

19

PROGRAMACIÓN SPI



Otro puerto que puede utilizar nuestro microcontrolador es el de **SPI** (*Serial Peripheral Interface*), que es un tipo de comunicaciones usado principalmente para la transferencia de información entre circuitos integrados. Emplea también, la arquitectura de dispositivo maestro (*master*) que puede comunicarse con uno o varios dispositivos esclavos (*slave*). Los dispositivos esclavos no pueden iniciar la comunicación, ni comunicarse entre ellos.

Utiliza un bus de cuatro líneas, de las que existen dos líneas para la comunicación: una entre el dispositivo maestro y los esclavos y la otra para la comunicación de los dispositivos esclavos hacia el maestro; donde el dispositivo maestro puede enviar y recibir datos simultáneamente en una comunicación síncrona. Hay otra línea por la que el maestro proporciona una señal de reloj, que sincroniza a todos los dispositivos esclavos.

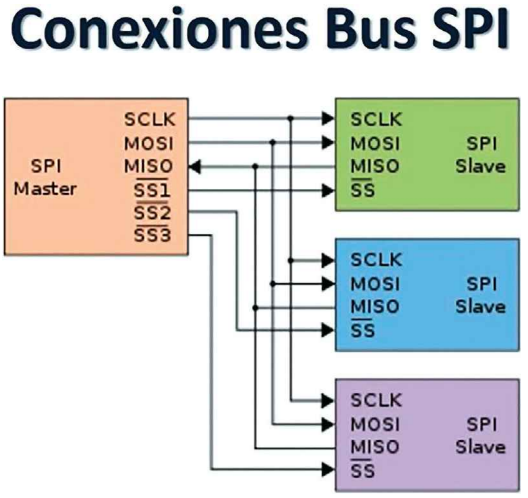


Figura 19.1

La línea **MOSI** (*Master-out, slave-in*), es la línea que proporciona comunicación entre el maestro y el esclavo.

La línea **MISO** (*Master-in, slave-out*), la que permite la comunicación desde el esclavo hacia el maestro.

La línea **SCK** (*Clock*), es la de sincronización de reloj, es generada por el maestro y sincroniza las comunicaciones, evitando tener que pactar una velocidad de transmisión entre los periféricos, aunque sí existe un límite máximo según el dispositivo que se emplee.

Según qué casos, se requerirá de una línea adicional **SS** (*Slave Select*) para cada dispositivo esclavo conectado, esto permitirá seleccionar cuál de los dispositivos será con el que se comunicará el maestro.

El protocolo que se utiliza para generar la comunicación es sencillo. Por defecto el dispositivo maestro mantiene las líneas **SS** en estado alto (*High*). Cuando necesita establecer comunicación con un determinado dispositivo esclavo, su línea **SS** se pondrá en nivel bajo (*Low*) y, en cada pulso de señal de reloj del maestro, este envía un bit hacia el esclavo por su línea **MOSI.** El esclavo a su vez, cuando quiere transmitir, envía hacia el maestro información utilizando también los pulsos de reloj de la línea **SCK** para sincronizar la transferencia, ahora por la línea **MISO,** hacia el maestro. Mientras ha durado la transferencia, la línea **SS** ha permanecido a nivel bajo (*Low*) y cuando se terminado se pasa a nivel alto (*High*) de nuevo, generando así el estado de inicio (*Start*) y paro (*Stop*) en la comunicación.

La información enviada no obedece a ningún protocolo o regla. Se puede enviar cualquier trama de bits; pero previamente, es necesario, que tanto el maestro como el esclavo tengan una pre-configuración respecto a la longitud y significado de dichas tramas para que se comprendan.

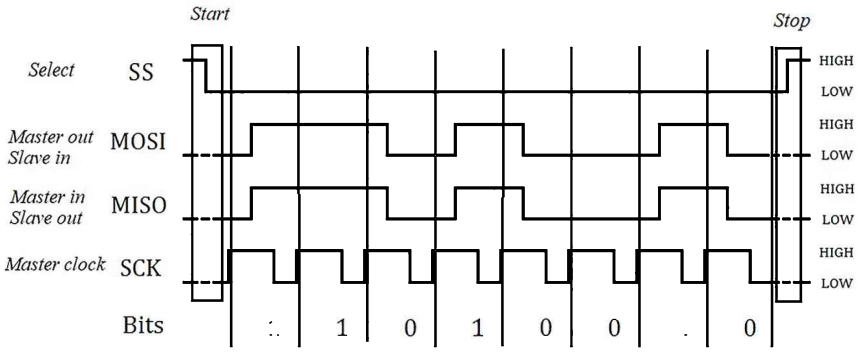


Figura 19.2

19.1 EJEMPLO DE PROGRAMACIÓN SPI DE UN MAX7912

El circuito **MAX7219** es un dispositivo driver para el control de display de matriz de led de 8 x 8 que posee un bus SPI para comunicarse con él.

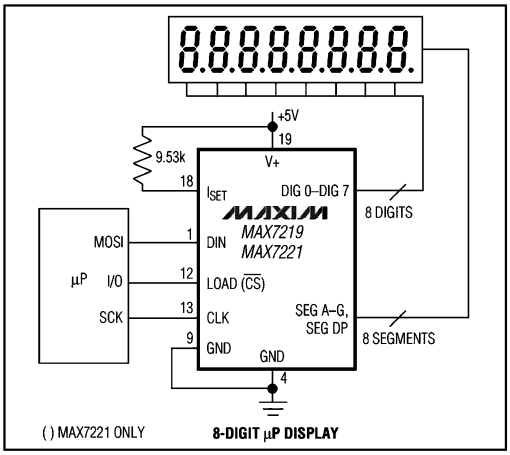


Figura 19.3. (Propiedad de Maxim Integrated, extraída del Datasheet)

El **MAX7219** trabaja con un voltaje de 5 Vcc y en su interior posee una SRAM con 14 registros direccionables para seleccionar los dígitos de una matriz de 8x8 mediante las columnas de la 1 a la 8, y cinco registros más para establecer los valores de inicialización y control.

Trabaja con paquetes de información que se le envían a su pin DIN y que son leídos en cada flanco ascendente de la señal de reloj **CLK**. La señal de selección **SS** (en alguno sitios establecida como **NSS**), debe estar a nivel bajo para activarlo. Estos paquetes de datos serán de 16 bits. Deben contener la dirección de los registros internos (los bits D8 al D11) y los datos a guardar en los bits del D0 al D7. Se descartarán los bits del D12-D15; siendo el primer valor que recibe el integrado a partir del bit D15 que es el más significativo (MSB).

Es necesario establecer los parámetros de inicialización del integrado antes de enviarle datos a mostrar en la matriz; ya que, cuando se produce el encendido de este, todos los registros de control se reinician, quedando la matriz en blanco y estableciéndose el modo de ahorro de energía automáticamente. Esto provoca que inicialmente, el integrado se configura por sí solo a escanear datos entrantes y no obedecerá a tramas con registros de control.

La codificación de tramas puede enviarse por defecto en el modo de códigos BCD, aunque es posible configurar el Registro de Método de Decodificación (*Decode Mode*) en el modo sin decodificación, donde cada bit transmitido (dígito del D7 al D0) se corresponderá a un digito de la matriz.

El MAX7219, contiene un Registro de Control del brillo (*Intensity*) con el que es posible regular el nivel de brillo de la pantalla de leds, donde se le puede dar un valor entre 1 y 15.

Mediante el Registro de Límite de Escaneo (*Scan Limit*), es posible conectar entre sí hasta 8 matrices diferentes de led que mostrarán de manera multiplexada la información que se le envía, en una frecuencia de 800 Hz. Aunque, debemos señalar, que el número de dígitos escaneados y conectados afecta al brillo de las pantallas por verse afectada también la potencia del consumo de su alimentación.

También posee un Registro de Prueba de pantalla (*Display Test*), el cual puede ser inicializado en dos formas: en el modo '0', o modo normal, se encienden todos los LED de la pantalla al inicio, pero sin resetearse los controles y registros de dígitos; en el modo '1', no se produce el encendido de los LED al inicio.

En nuestro ejemplo, emplearemos el siguiente módulo que posee ya: un display de matriz 8x8 led del tipo 1088AS, el integrado MAX7219 y una circuitería que interconecta ya estos elementos.

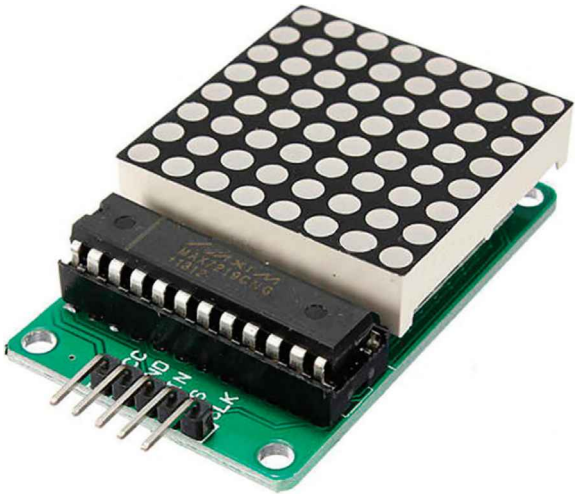


Figura 19.4

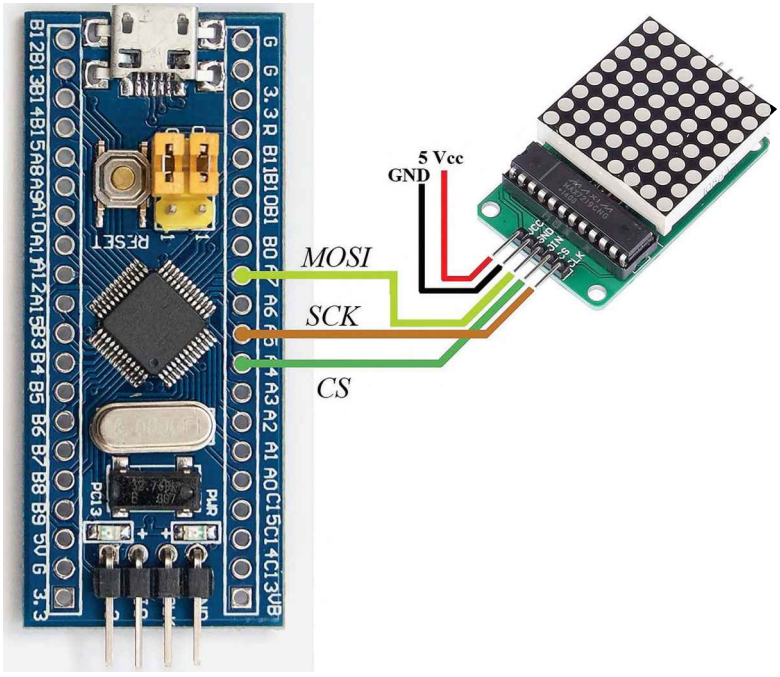


Figura 19.5

Reproducimos a continuación el código del ejemplo:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_spi.h"

// Listado de caracteres obtenidos de la dirección
// web siguiente https://xantorohara.github.io/led-matrix-editor/
 unsigned char caracter[38][8]={
{0x3C,0x42,0x42,0x42,0x42,0x42,0x42,0x3C},  //0
{0x10,0x18,0x14,0x10,0x10,0x10,0x10,0x10},  //1
{0x7E,0x2,0x2,0x7E,0x40,0x40,0x40,0x7E},    //2
{0x3E,0x2,0x2,0x3E,0x2,0x2,0x3E,0x0},       //3
{0x8,0x18,0x28,0x48,0xFE,0x8,0x8,0x8},      //4
{0x3C,0x20,0x20,0x3C,0x4,0x4,0x3C,0x0},     //5
{0x3C,0x20,0x20,0x3C,0x24,0x24,0x3C,0x0},   //6
{0x3E,0x22,0x4,0x8,0x8,0x8,0x8,0x8},        //7
{0x0,0x3E,0x22,0x22,0x3E,0x22,0x22,0x3E},   //8
{0x3E,0x22,0x22,0x3E,0x2,0x2,0x2,0x3E},     //9
{0x8,0x14,0x22,0x3E,0x22,0x22,0x22,0x22},   //A
{0x3C,0x22,0x22,0x3E,0x22,0x22,0x3C,0x0},   //B
{0x3C,0x40,0x40,0x40,0x40,0x40,0x3C,0x0},   //C
{0x7C,0x42,0x42,0x42,0x42,0x42,0x7C,0x0},   //D
{0x7C,0x40,0x40,0x7C,0x40,0x40,0x40,0x7C},  //E
{0x7C,0x40,0x40,0x7C,0x40,0x40,0x40,0x40},  //F
{0x3C,0x40,0x40,0x40,0x40,0x44,0x44,0x3C},  //G
{0x44,0x44,0x44,0x7C,0x44,0x44,0x44,0x44},  //H
{0x7C,0x10,0x10,0x10,0x10,0x10,0x10,0x7C},  //I
{0x3C,0x8,0x8,0x8,0x8,0x8,0x48,0x30},       //J
{0x0,0x24,0x28,0x30,0x20,0x30,0x28,0x24},   //K
{0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x7C},  //L
{0x81,0xC3,0xA5,0x99,0x81,0x81,0x81,0x81},  //M
{0x0,0x42,0x62,0x52,0x4A,0x46,0x42,0x0},    //N
{0x3C,0x42,0x42,0x42,0x42,0x42,0x42,0x3C},  //O
{0x3C,0x22,0x22,0x22,0x3C,0x20,0x20,0x20},  //P
{0x1C,0x22,0x22,0x22,0x22,0x26,0x22,0x1D},  //Q
{0x3C,0x22,0x22,0x22,0x3C,0x24,0x22,0x21},  //R
{0x0,0x1E,0x20,0x20,0x3E,0x2,0x2,0x3C},     //S
{0x0,0x3E,0x8,0x8,0x8,0x8,0x8,0x8},         //T
{0x42,0x42,0x42,0x42,0x42,0x42,0x22,0x1C},  //U
{0x42,0x42,0x42,0x42,0x42,0x42,0x24,0x18},  //V
{0x0,0x49,0x49,0x49,0x49,0x2A,0x1C,0x0},    //W
{0x0,0x41,0x22,0x14,0x8,0x14,0x22,0x41},    //X
{0x41,0x22,0x14,0x8,0x8,0x8,0x8,0x8},       //Y
{0x0,0x7F,0x2,0x4,0x8,0x10,0x20,0x7F},      //Z
};

// Modulo que genera retardo
//---------------------------------------
void delay_us(unsigned int nCount){
            unsigned int i, j;
            for(i = 0; i < nCount; i++)
            {
                        for(i = 0; i < 0x2AFF; i++){;}
            }
}

/*  Funcion que inicializa el Bus SPI      */
//-----------------------------------------
void SPI1_Init(void)
{
  SPI_InitTypeDef  SPI1_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;

  // Configuramos los reloj de los modulos
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
                         RCC_APB2Periph_SPI1, ENABLE);
  // Pin pa4 (CS PA4)-----------------------------------------------
  GPIO_InitStructure.GPIO_Pin       = GPIO_Pin_4;
  GPIO_InitStructure.GPIO_Speed     = GPIO_Speed_50MHz;
```

```
70    GPIO_InitStructure.GPIO_Mode      = GPIO_Mode_Out_PP;
71    GPIO_Init(GPIOA,&GPIO_InitStructure);
72
73    // Pines de PA5(SCK),  PA6 (MISO)   y   PA7 (MOSI)---------------
74    GPIO_InitStructure.GPIO_Pin       = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
75    GPIO_InitStructure.GPIO_Mode      = GPIO_Mode_AF_PP;
76    GPIO_Init(GPIOA,&GPIO_InitStructure);
77
78    // Inicializamos el bus ISP (1)---------------------------------
79    SPI1_InitStructure.SPI_Direction |  = SPI_Direction_2Lines_FullDuplex;
80    SPI1_InitStructure.SPI_BaudRatePrescaler  = SPI_BaudRatePrescaler_64;
81    SPI1_InitStructure.SPI_DataSize           = SPI_DataSize_8b;
82    SPI1_InitStructure.SPI_Mode               = SPI_Mode_Master;
83    SPI1_InitStructure.SPI_FirstBit           = SPI_FirstBit_MSB;
84    SPI1_InitStructure.SPI_CPOL               = SPI_CPOL_High;
85    SPI1_InitStructure.SPI_CPHA               = SPI_CPHA_2Edge;
86    SPI1_InitStructure.SPI_NSS                = SPI_NSS_Soft;
87    SPI1_InitStructure.SPI_CRCPolynomial      = 7;
88
89    SPI_I2S_DeInit(SPI1);
90    SPI_Init(SPI1, &SPI1_InitStructure);
91    SPI_Cmd(SPI1, ENABLE);
92  }
93
94  /*     Funcion que escribe un byte en el bus ISP              */
95  //-------------------------------------------------------------
96  void Send_Max7219(unsigned char address,unsigned char data)
97  {
98      GPIO_ResetBits(GPIOA, GPIO_Pin_4);
99
100     SPI_I2S_SendData(SPI1, address); //Escribir la dirección del
101 registro
102     delay_us(10);
103
104     SPI_I2S_SendData(SPI1, data);    //Escribir datos
105     delay_us(10);
106
107     GPIO_SetBits(GPIOA, GPIO_Pin_4);
108 }
109
110 /*      Funcion que inicializa los registros del MAX7219      */
111 //-------------------------------------------------------------
112 void Init_MAX7219(void)
113 {  /*  Registros de control: ----------------------------------
114       0x9 -- Registro de modo de codificacion (Decode Mode)
115       0xA -- Registro de brillo de pantalla (Intensity)
116       0xB -- Registro de límite de escaneo (Scan Limit)
117       0xC -- Registro de modo de apagado (Shutdwn)
118       0xF -- Registro de detección de pantalla (Display Test)
119    ------------------------------------------------------------*/
120    //Send_Max7219(0x09, 0xFF); // Método de decodificación BCD
121    Send_Max7219(0x09, 0x00); // Método de decodificación desabilita BCD
122    Send_Max7219(0x0A, 0x03); // Luminosidad
123    Send_Max7219(0x0B, 0x07); // Limite de escaneo; Pantalla de 8 LEDs
124    Send_Max7219(0x0C, 0x01); // Modo de apagado: 0, modo normal 1
125    Send_Max7219(0x0F, 0x00); // Test de pantalla: 1
126 }
127
128 /*            Modulo Principal                                 */
129 //-------------------------------------------------------------
130 int main(void)
131  {
132    SPI1_Init();      // Inicializamos el bus ISP
133
134    delay_us(800);    // Retardo para que se reinicie
135
136    Init_MAX7219();   // Enviamos la secuencia de configuracion del
137                         MAX7219
138
139    delay_us(200);     // Retardo para que se reinicie
140
```

```
141    while(1)  // Bucle infinito para que se repita automatica.....e
142    {
143       // Bucle que va mostrando los caracteres predefinidos
144       for(int NLetra=0; NLetra<38; NLetra++)
145       {
146         // Selecciona cada registro de la columna a escribir
147         for(int NRow=1; NRow<9; NRow++)
148         Send_Max7219(NRow, caracter[NLetra][NRow-1]);
149         delay_us(500);
150       }
151    }
152 }
153
```

Figura 19.6

Como vemos, su estructura de funcionamiento y comandos empleados es muy similar a la del bus I2C que vimos en el apartado anterior.

Se crea una función que encierra todos los parámetros necesarios para los GPIO de los puertos y pines a emplear y la configuración del bus SPI.

Nuestro microcontrolador posee también dos puertos SPI, donde el número 1 puede ser remapeado hacia otros pines del microcontrolador.

	Puerto SPI_1 (remap)		Puerto SPI_2
SPI1_NSS	PA4	PA15	PB12
SPI_SCK	PA5	PB3	PS13
SPI_MISO	PA6	PB4	PB14
SPI_MOSI	PA7	PB5	PB15

Figura 19.7

Creamos la estructura previa de configuración de los pines, según se muestra en la Figura 19.8.

```
1   void SPI1_Init(void)
2   {
3     SPI_InitTypeDef  SPI1_InitStructure;
4     GPIO_InitTypeDef GPIO_InitStructure;
5
6     // Configuramos los reloj de los modulos
7     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
8                            RCC_APB2Periph_SPI1, ENABLE);
9     // Pin pa4 (CS PA4)-----------------------------------------
10    GPIO_InitStructure.GPIO_Pin       = GPIO_Pin_4;
11    GPIO_InitStructure.GPIO_Speed     = GPIO_Speed_50MHz;
12    GPIO_InitStructure.GPIO_Mode      = GPIO_Mode_Out_PP;
13    GPIO_Init(GPIOA,&GPIO_InitStructure);
14
15    // Pines de PA5(SCK),  PA6 (MISO)   y   PA7 (MOSI)----------
16    GPIO_InitStructure.GPIO_Pin       = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
17    GPIO_InitStructure.GPIO_Mode      = GPIO_Mode_AF_PP;
18    GPIO_Init(GPIOA,&GPIO_InitStructure);
19     .
20     .
21     .
```

Figura 19.8

En esta configuración establecemos un pin de salida en **PA4** para la señal **SS** y tres pines más con las señales de sincronización **SCK** en **PA5**, la señal **MISO** en **PA6** y la de salida de tramas **MOSI** en **PA7**. Como no establecemos una comunicación con lectura de datos desde el dispositivo, no se empleará la señal MISO en la conexión con este.

En el siguiente grupo de nuestro código se establece la configuración que tendrá el bus SPI respecto a la comunicación con el dispositivo.

```
1      .
2      .
3      .
4     // Inicializamos el bus ISP (1)---------------------------------
5     SPI1_InitStructure.SPI_Direction      = SPI_Direction_2Lines_FullDuplex;
6     SPI1_InitStructure.SPI_BaudRatePrescaler  = SPI_BaudRatePrescaler_64;
7     SPI1_InitStructure.SPI_DataSize           = SPI_DataSize_8b;
8     SPI1_InitStructure.SPI_Mode               = SPI_Mode_Master;
9     SPI1_InitStructure.SPI_FirstBit           = SPI_FirstBit_MSB;
10    SPI1_InitStructure.SPI_CPOL               = SPI_CPOL_High;
11    SPI1_InitStructure.SPI_CPHA               = SPI_CPHA_2Edge;
12    SPI1_InitStructure.SPI_NSS                = SPI_NSS_Soft;
13    SPI1_InitStructure.SPI_CRCPolynomial      = 7;
14
15    SPI_I2S_DeInit(SPI1);
16    SPI_Init(SPI1, &SPI1_InitStructure);
17    SPI_Cmd(SPI1, ENABLE);
18  }
```

Figura 19.9

Comenzamos habilitando el reloj correspondiente al módulo SPI, mediante el comando RCC siguiente "RCC_APB2PeriphClockCmd(**RCC_APB2Periph_SPI1**, ENABLE)".

Después configuramos el tipo de comunicación que utilizaremos en nuestro modelo unidireccional o bidireccional, y el número de líneas que usaremos, mediante el comando "**SPI_Direction**", que podemos establecer como:

- **SPI_Direction_2Lines_FullDuplex** – Dos líneas en modo full dupplex.
- **SPI_Direction_2Lines_RxOnly** – Dos líneas modo recepción sólo.
- **SPI_Direction_1Line_Rx** – Una línea modo recepción sólo.
- **SPI_Direction_1Line_Tx** – Una línea modo transmisión sólo.

Figura 19.10

Con el siguiente comando "**SPI_DataSize**", establecemos el tamaño que tendrán los datos transmitidos, que podrán ser "**SPI_DataSize_8b**" para 8 bits o "**SPI_DataSize_16b**" para 16.

El comando "**SPI_Mode**" configura qué dispositivo será nuestro microcontrolador, el "**SPI_Mode_Master**" en modo maestro, o "**SPI_Mode_Slave**" para indicar que será un dispositivo esclavo de otro.

Mediante el comando "**SPI_ FirstBit**", indicamos cual será el orden en que se transmitirán los datos, si se enviarán primero los bits más significativos con "**SPI_ FirstBit_MSB**", o los bits menos significativos con "**SPI_FirstBit_LSB**".

Con el comando "**SPI_CPOL**", implantamos en qué estado estará inactivo el reloj de sincronización; que podrá ser en nivel alto con "**SPI_CPOL_High**" o en nivel bajo con "**SPI_CPOL_Low**".

Configuraremos con el comando "**SPI_CPHA**", en qué nivel se dispara la lectura de los datos en la señal de reloj CLK: en el primer activador de señal de borde con "**SPI_CPHA_1Edge**", o, en el segundo activador de señal de borde "**SPI_CPHA_2Edge**".

Como se muestra en la Figura 19.11.

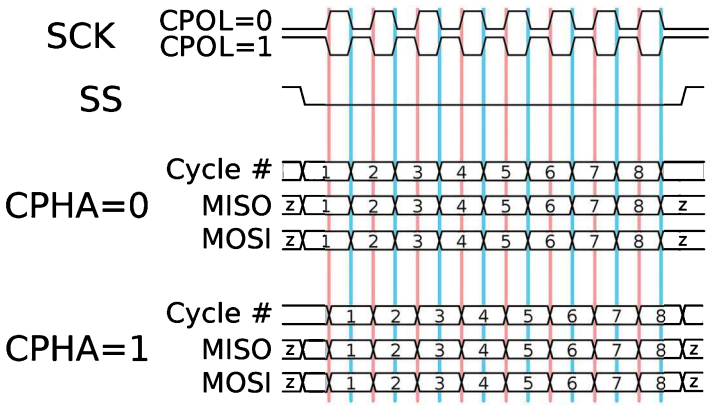


Figura 19.11

Si CPOL y CPHA son ambos '0' (Modo 0), los datos se muestrean en el borde ascendente principal del reloj. El modo 0 es el modo más común para la comunicación esclava de bus SPI. Si CPOL = '1' y CPHA = '0' (Modo 2), los datos se muestrean en el borde descendente principal del reloj. Del mismo modo, si CPOL = '0' y CPHA = '1' (Modo 1) da como resultado datos muestreados en el borde descendente posterior y, si CPOL = '1' y CPHA = '1' (Modo 3) dará como resultado datos muestreados al final del flanco ascendente.

Otro parámetro a establecer con el comando "**SPI_NSS**", es el que nos permite indicar cómo queremos que se active la señal del pin SS, si mediante un control por software con "**SPI_NSS_Soft**" o por hardware con "**SPI_NSS_Hard**"; en este último modo, el estado del pin es administrado por el propio periférico; lo que obliga a configurar el GPIO de dicho pin como de salida si el dispositivo es un maestro y de entrada si es un esclavo. Además en este modo por hardware, el dispositivo a usar deberá contar específicamente con esa cualidad.

Con el comando "**SPI_CRCPolynomial**", se configura el CRC (*cyclic redundancy check*). Se basa en la aritmética polinómica para calcular el resto de la división de un polinomio que representa los datos transmitidos; esto produce una suma que comprueba si los datos han sido transmitidos y recibidos correctamente. Con este comando, lo que estableceremos será cuál debe ser el bit que contendrá dicha suma de comprobación.

Con el comando "**SPI_I2S_DeInit**(*SPIx*)", se resetean los registros del periférico SPI 1 o 2 a sus valores predeterminados.

Con el comando "**SPI_Init**(*SPIx*, &InitTypeDef)" se cargan los parámetros que hemos establecido en nuestra estructura.

Y por último, mediante el comando "**SPI_Cmd**(*SPIx*, ENABLE)", se inicia el módulo SPI que hemos configurado.

A continuación, necesitamos establecer los valores de inicialización de nuestro MAX7219.

```
1   void Init_MAX7219(void)
2   {  /*  Registros de control: ------------------------------------
3         0x9 -- Registro de modo de codificacion (Decode Mode)
4         0xA -- Registro de brillo de pantalla (Intensity)
5         0xB -- Registro de límite de escaneo (Scan Limit)
6         0xC -- Registro de modo de apagado (Shutdwn)
7         0xF -- Registro de detección de pantalla (Display Test)
8       ------------------------------------------------------------*/
9       //Send_Max7219(0x09, 0xFF); // Método de decodificación BCD
10      Send_Max7219(0x09, 0x00); // Método de decodificación desabilita BCD
11      Send_Max7219(0x0A, 0x03); // Luminosidad
12      Send_Max7219(0x0B, 0x07); // Limite de escaneo; Pantalla de 8 LEDs
13      Send_Max7219(0x0C, 0x01); // Modo de apagado: 0, modo normal 1
14      Send_Max7219(0x0F, 0x00); // Test de pantalla: 1
15  }
```

Figura 19.12

Lo siguiente será crear la función que envía los datos hacia el dispositivo.

```
1  void Send_Max7219(unsigned char address,unsigned char data)
2  {
3      GPIO_ResetBits(GPIOA, GPIO_Pin_4);
4
5      //Escribir la dirección del registro
6      SPI_I2S_SendData(SPI1, address);
7      delay_us(10);
8
9      //Escribir datos
10     SPI_I2S_SendData(SPI1, data);
11     delay_us(10);
12     GPIO_SetBits(GPIOA, GPIO_Pin_4);
13 }
```

Figura 19.13

Observamos que, primero pasamos a nivel bajo el pin PA4, que corresponde a la señal de selección del dispositivo esclavo, para que así este sepa que vamos a comunicarnos con él. A continuación, utilizamos el comando "**SPI_I2C_SendData**(SPIx, uint8/16_t dato)" con el que enviamos un byte por el pin de transmisión MOSI hacia el pin DIN del MAX7219.

El primer byte que enviamos será el de la dirección del registro que queremos escribir en la memoria SRAM del integrado, que podrá ser un registro de control o de direccionamiento de una de las columnas del display.

El siguiente byte que transmitimos inmediatamente será el dato que queremos grabar.

Por último se pasa a nivel alto, el pin PA4 para indicar en la línea NSS o SS que ya no transmitimos más.

Veremos que, dado que el dispositivo que utilizamos para comunicarnos, no nos suministra ningún tipo de trama que necesitemos leer, no creamos ninguna programación que realice un control del bus cuando se nos envía algún dato desde el dispositivo.

Incluimos a continuación el código completo del ejemplo:

```
1  #include "stm32f10x.h"
2  #include "stm32f10x_gpio.h"
3  #include "stm32f10x_rcc.h"
4  #include "stm32f10x_spi.h"
5
6  // Listado de caracteres que se pueden definir en la
7  // siguiente pagina de internet:
8  // https://xantorohara.github.io/led-matrix-editor/
9   unsigned char caracter[38][8]={
```

```
{0x3C,0x42,0x42,0x42,0x42,0x42,0x42,0x3C},  //0
{0x10,0x18,0x14,0x10,0x10,0x10,0x10,0x10},  //1
{0x7E,0x2,0x2,0x7E,0x40,0x40,0x40,0x7E},    //2
{0x3E,0x2,0x2,0x3E,0x2,0x2,0x3E,0x0},       //3
{0x8,0x18,0x28,0x48,0xFE,0x8,0x8,0x8},      //4
{0x3C,0x20,0x20,0x3C,0x4,0x4,0x3C,0x0},     //5
{0x3C,0x20,0x20,0x3C,0x24,0x24,0x3C,0x0},   //6
{0x3E,0x22,0x4,0x8,0x8,0x8,0x8,0x8},        //7
{0x0,0x3E,0x22,0x22,0x3E,0x22,0x22,0x3E},   //8
{0x3E,0x22,0x22,0x3E,0x2,0x2,0x2,0x3E},     //9
{0x8,0x14,0x22,0x3E,0x22,0x22,0x22,0x22},   //A
{0x3C,0x22,0x22,0x3E,0x22,0x22,0x3C,0x0},   //B
{0x3C,0x40,0x40,0x40,0x40,0x40,0x3C,0x0},   //C
{0x7C,0x42,0x42,0x42,0x42,0x42,0x7C,0x0},   //D
{0x7C,0x40,0x40,0x7C,0x40,0x40,0x40,0x7C},  //E
{0x7C,0x40,0x40,0x7C,0x40,0x40,0x40,0x40},  //F
{0x3C,0x40,0x40,0x40,0x40,0x44,0x44,0x3C},  //G
{0x44,0x44,0x44,0x7C,0x44,0x44,0x44,0x44},  //H
{0x7C,0x10,0x10,0x10,0x10,0x10,0x10,0x7C},  //I
{0x3C,0x8,0x8,0x8,0x8,0x8,0x48,0x30},       //J
{0x0,0x24,0x28,0x30,0x20,0x30,0x28,0x24},   //K
{0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x7C},  //L
{0x81,0xC3,0xA5,0x99,0x81,0x81,0x81,0x81},  //M
{0x0,0x42,0x62,0x52,0x4A,0x46,0x42,0x0},    //N
{0x3C,0x42,0x42,0x42,0x42,0x42,0x42,0x3C},  //O
{0x3C,0x22,0x22,0x22,0x3C,0x20,0x20,0x20},  //P
{0x1C,0x22,0x22,0x22,0x22,0x26,0x22,0x1D},  //Q
{0x3C,0x22,0x22,0x22,0x3C,0x24,0x22,0x21},  //R
{0x0,0x1E,0x20,0x20,0x3E,0x2,0x2,0x3C},     //S
{0x0,0x3E,0x8,0x8,0x8,0x8,0x8,0x8},         //T
{0x42,0x42,0x42,0x42,0x42,0x42,0x22,0x1C},  //U
{0x42,0x42,0x42,0x42,0x42,0x42,0x24,0x18},  //V
{0x0,0x49,0x49,0x49,0x49,0x2A,0x1C,0x0},    //W
{0x0,0x41,0x22,0x14,0x8,0x14,0x22,0x41},    //X
{0x41,0x22,0x14,0x8,0x8,0x8,0x8,0x8},       //Y
{0x0,0x7F,0x2,0x4,0x8,0x10,0x20,0x7F},      //Z
};

/* Modulo que genera un simple retardo */
//-------------------------------------
void delay_us(unsigned int nCount){
            unsigned int i, j;
            for(i = 0; i < nCount; i++)
            {
                        for(j = 0; j < 0x2AFF; j++){;}
            }
}

/*  Funcion que inicializa el Bus SPI                                    */
//-----------------------------------------------------------------------
void SPI1_Init(void)
{
   /*   Creamos las estructura para el GPIO y el SPI ------------*/
   SPI_InitTypeDef  SPI1_InitStructure;
   GPIO_InitTypeDef GPIO_InitStructure;

   /* Habilitamos los reloj del GPIO y el SPI---------------------*/
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
                             RCC_APB2Periph_SPI1, ENABLE);

   /* Creacion del GPIO del pin PA4 para la señal SS  -----------*/
   GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_4;
   GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_Init(GPIOA,&GPIO_InitStructure);
```

```
75
76      /* Creacion del GPIO de los pines PA5(SCK), PA6(MISO), PA7(MOSI)*/
77      GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
78      GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF_PP;
79      GPIO_Init(GPIOA,&GPIO_InitStructure);
80
81      /* Configuracion del puerto ISP (1)    -------------------------*/
82      SPI1_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
83      SPI1_InitStructure.SPI_BaudRatePrescaler= SPI_BaudRatePrescaler_64;
84      SPI1_InitStructure.SPI_DataSize         = SPI_DataSize_8b;
85      SPI1_InitStructure.SPI_Mode             = SPI_Mode_Master;
86      SPI1_InitStructure.SPI_FirstBit         = SPI_FirstBit_MSB;
87      SPI1_InitStructure.SPI_CPOL             = SPI_CPOL_High;
88      SPI1_InitStructure.SPI_CPHA             = SPI_CPHA_2Edge;
89      SPI1_InitStructure.SPI_NSS              = SPI_NSS_Soft;
90      SPI1_InitStructure.SPI_CRCPolynomial    = 7;
91
92      /*  Resetea el modulo SPI1  ----------------------------------*/
93      SPI_I2S_DeInit(SPI1);
94
95      /* Reconfigura el modulo SPI1  ------------------------------*/
96      SPI_Init(SPI1, &SPI1_InitStructure);
97
98      /* Iniciamos el modulo SPI1  --------------------------------*/
99      SPI_Cmd(SPI1, ENABLE);
100 }
101
102 /*      Funcion que escribe un byte en el bus ISP             */
103 //------------------------------------------------------------
104 void Send_Max7219(unsigned char address,unsigned char data)
105 {
106     GPIO_ResetBits(GPIOA, GPIO_Pin_4);
107
108     /* Enviamos la direccion del registro a grabar ----------*/
109     SPI_I2S_SendData(SPI1, address);
110     delay_us(10);
111
112     /* Enviamos el dato a grabar en el registro -------------*/
113     SPI_I2S_SendData(SPI1, data);         //Escribir datos
114     delay_us(10);
115
116     GPIO_SetBits(GPIOA, GPIO_Pin_4);
117 }
118
119 /*      Funcion que inicializa los registros del MAX7219      */
120 //------------------------------------------------------------
121 void Init_MAX7219(void)
122 {  /*  Registros de control: ----------------------------------
123       0x9 -- Registro de modo de codificacion (Decode Mode)
124       0xA -- Registro de brillo de pantalla (Intensity)
125       0xB -- Registro de límite de escaneo (Scan Limit)
126       0xC -- Registro de modo de apagado (Shutdwn)
127       0xF -- Registro de detección de pantalla (Display Test)
128    ------------------------------------------------------------*/
129     //Send_Max7219(0x09, 0xFF); // Método de decodificación BCD
130   Send_Max7219(0x09, 0x00); // Método decodificación: desabilita BCD
131   Send_Max7219(0x0A, 0x03); // Luminosidad
132   Send_Max7219(0x0B, 0x07); // Limite de escaneo; Pantalla de 8 LEDs
133   Send_Max7219(0x0C, 0x01); // Modo de apagado: 0, modo normal 1
134   Send_Max7219(0x0F, 0x00); // Test de pantalla: 1
135 }
136
137 /*            Modulo Principal                                */
138 //------------------------------------------------------------
139 int main(void)
```

```
140 {
141     SPI1_Init();     // Inicializamos el bus ISP
142     delay_us(800);  // Retardo para que se reinicie
143
144     Init_MAX7219(); // Enviamos secuencia de configuracion del MAX7219
145     delay_us(200);  // Retardo para que se reinicie
146
147     while(1)  // Bucle infinito para que se repita automaticamente
.48     {
149       // Bucle que va mostrando los caracteres predefinidos
150       for(int NLetra=0; NLetra<38; NLetra++)
151       {
152         // Selecciona cada registro de la columna a escribir
153         for(int NRow=1; NRow<9; NRow++)
154            Send_Max7219(NRow, caracter[NLetra][NRow-1]);
155            delay_us(500);
156       }
157   }
158 }
```

Figura 19.14

20

PROGRAMACIÓN USB



Nuestro microcontrolador STM32 también posee un puerto **USB** (*Universal Serial Bus*) que puede ser utilizado para comunicarnos con él.

Este tipo de conexión hoy en día es utilizada por una gran cantidad de dispositivos y por ello, es muy importante saber cómo se programa y cómo podemos aplicarlo para nuestros proyectos.

La empresa STMicroelectronics también nos ofrece una librería CMSIS que nos facilita la programación de estos dispositivos para todos los microcontroladores de la familia STM32.

Esta librería se puede descargar gratuitamente desde la página de STMicroelectronics en la siguiente dirección:

http://www.st.com/en/embedded-software/stsw-stm32121.html#

También ofrece información detallada sobre la librería en formato PDF en inglés en la siguiente dirección de internet:

http://www.st.com/en/embedded-software/stsw-stm32121.html

El paquete con la librería ofrece también códigos de ejemplo para la creación de firmware en nuestro microcontrolador que puede ser utilizado como ocon dispositivos HID, dispositivos de interfaz humana (*Human Interface Device*); implementación con dispositivos joysticks; transmisión de audio; dispositivos CDC de Configuración de Dispositivos Conectados (*Connected Device Configuration*); conexión USB a RS-232 como dispositivo bridge de puerto COM virtual; emulación de un dispositivo de almacenamiento masivo como tarjetas microSD y, como dispositivo DFU de actualización de dispositivos (*Device Firmware Update*).

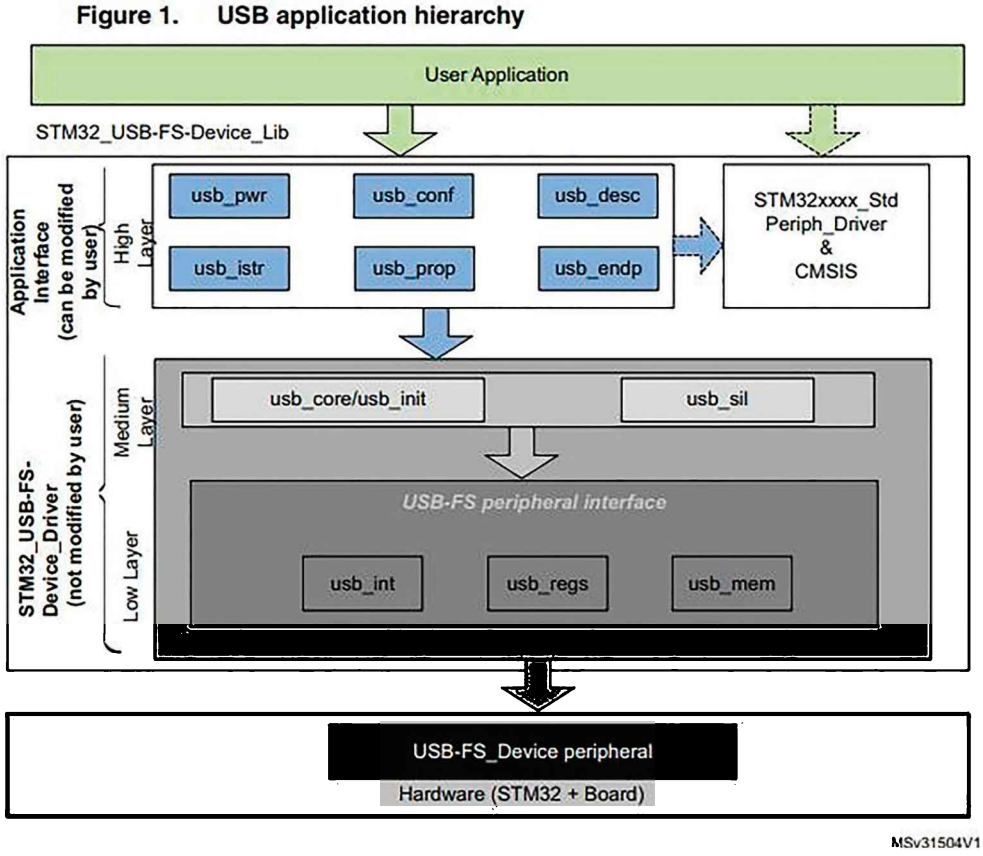


Figura 20.1. (Propiedad de STMicroelectronics del Manual de referencia de la librería "STM32 USB-FS-Device firmware library")

Una vez dominadas un poco más estas librerías, nuestra imaginación nos permitirá utilizar nuestro microcontrolador en otros muchos campos con el bus USB.

20.1 EJEMPLO DE CONEXIÓN USB COMO PUERTO COM VIRTUAL

De los ejemplos que contiene la librería, reproducimos uno en el que convertimos nuestro microcontrolador en un dispositivo que se conectará directamente a nuestro ordenador y será reconocido como un dispositivo estándar USB.



Figura 20.2

Una vez creado y cargado el firmware en la placa, nuestro ordenador la reconocerá como un dispositivo de COM virtual, tal como se muestra en la Figura 20.3.

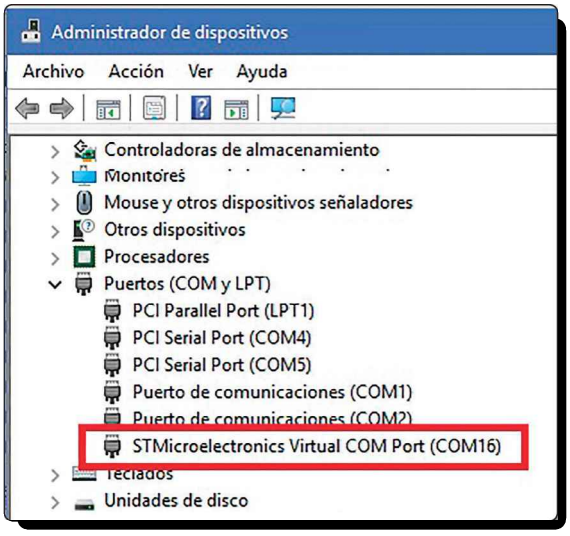


Figura 20.3

El código utiliza las librerías de STMicroelectronics para su gestión, por lo que es importante que guardemos dentro de una carpeta de nuestro proyecto el contenido de dicha librería para poder llamarla y poder así configurar el dispositivo adecuadamente.

También configuraremos el Keil, para que vaya a buscar en dicha carpeta los ficheros necesarios, para lo que tendremos que añadir el camino hacia esa carpeta en la configuración del proyecto. Los pasos a seguir son: una vez iniciado el Keil, seleccionamos el botón de "***Options for Target***" y la pestaña de "**C/C++**", luego pulsamos sobre el botón en la opción de "***Include Paths***" tal como se muestra en la Figura 20.4.

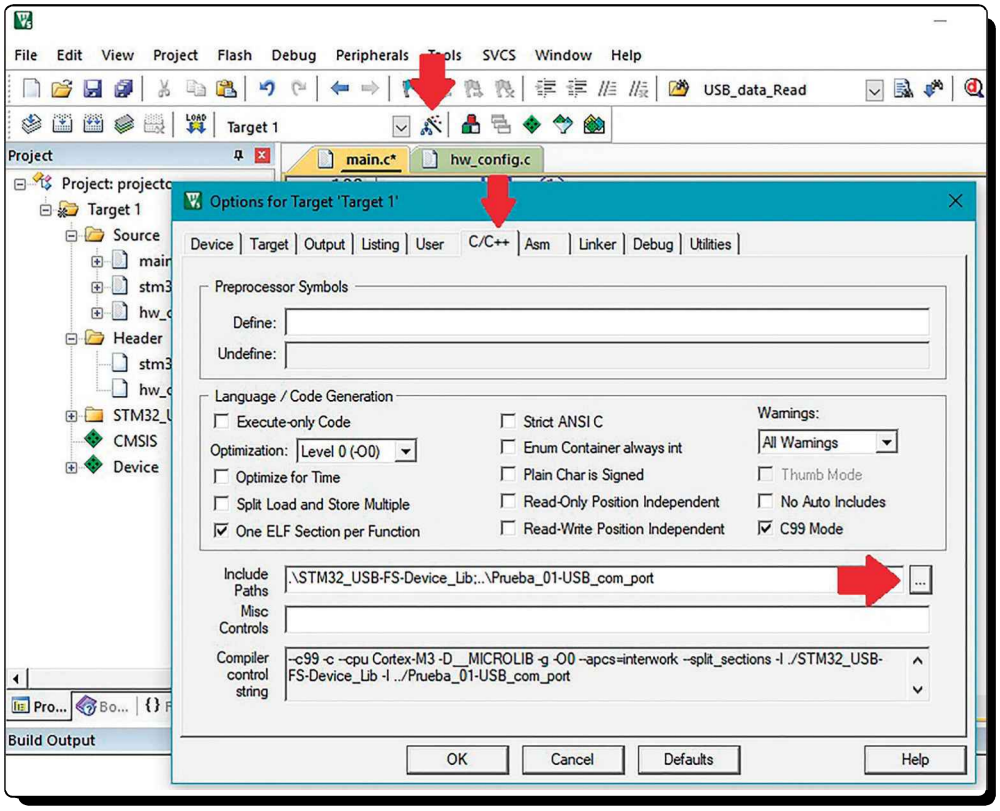


Figura 20.4

Se nos abrirá una ventana, como la de la Figura 20.5, donde podremos seleccionar la carpeta.

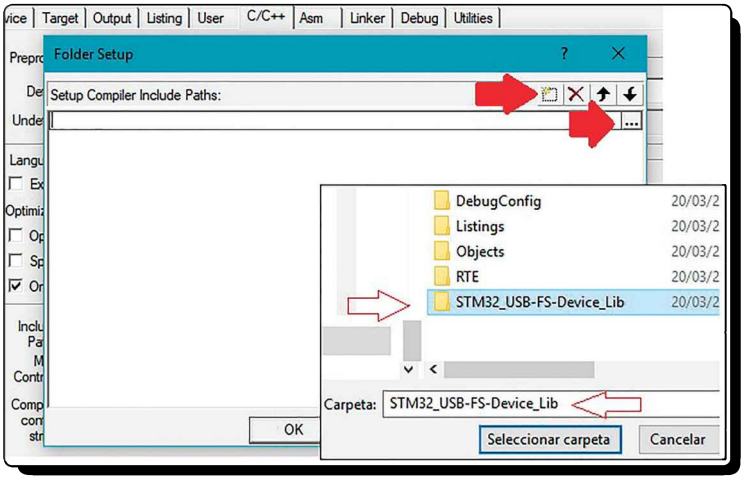


Figura 20.5

Deberemos añadir también los ficheros de esa carpeta en la compilación, para lo que seguiremos los pasos señalados en la Figura 20.6.

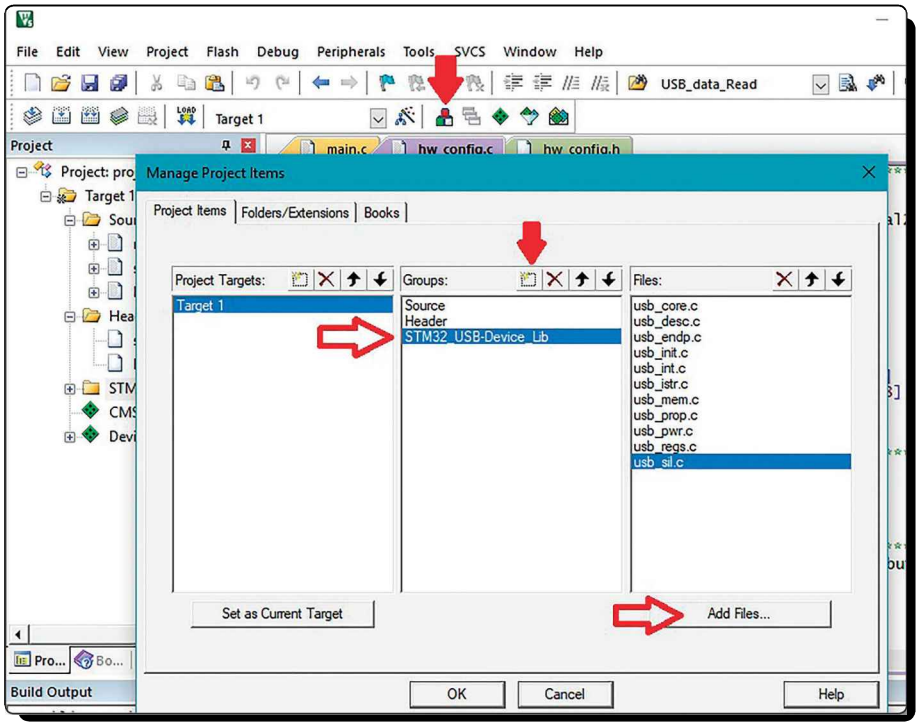


Figura 20.6

Reproducimos el contenido del fichero principal main:

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"

#include "hw_config.h"
#include "usb_lib.h"
#include "usb_pwr.h"

extern volatile uint8_t Receive_Buffer[64];
extern volatile uint32_t Receive_length ;
extern volatile uint32_t length ;
uint32_t Data_send  = 1;
uint32_t packet_sent  = 1;
uint32_t packet_receive = 1;

/*                Modulo Principal                    */
//-----------------------------------------------------
int main(void)
{
  Set_System();      // Inicializamos el GPIO del USB

  USB_Init();        // Inicializamos la libreria USB

  while (1)
  {
    // Comprobamos si el puerto USB esta operativo
    if (bDeviceState == CONFIGURED)
    {
      CDC_Receive_DATA();     // Leemos la entrada del USB

      // Comprobamos si se ha recibido un dato
      if (Receive_length  != 0)
      {
        // Enciende el LED de pruebas al recibir un '1'
        if (Receive_Buffer[0]=='1') {
             GPIO_ResetBits(GPIOC, GPIO_Pin_13);
        }
        else {
          // Apaga el LED de pruebas al recibir cualquier otro caracter
             GPIO_SetBits(GPIOC, GPIO_Pin_13);
        }

        // Reproducimos eco hacia el PC de los datos recibidos por el USB
        if (Data_send == 1) {
             USB_data_Send ((uint8_t*)Receive_Buffer, Receive_length);
        }

        Receive_length = 0;
      }
    }
  }
}
```

Figura 20.7

Como podemos ver, además de los "*include*" correspondientes al tipo de microcontrolador utilizado y las librerías propias de la gestión de los módulos estándar, se incluye la llamada a librerías específicas del puerto USB; y en especial, el fichero "hw_config.h", donde se configuran los parámetros del GPIO y la

interrupción que manejará el puerto USB en nuestro microcontrolador en la función "**Set_System**()", que se muestra en la Figura 20.8.

```
1   void Set_System(void)
2   {
3     GPIO_InitTypeDef  GPIO_InitStructure;
4
5      /* Habilita los reloj para el puerto USB */
6     RCC_APB1PeriphClockCmd(RCC_APB1Periph_USB, ENABLE);
7     RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);
8
9     /* Habilita el reloj para el GPIO pin de desconexión del puerto USB   */
10    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_DISCONNECT, ENABLE);
11
12    /* Se crea el GPIO para el pin USB_DISCONNNECT del puerto USB   */
13    GPIO_InitStructure.GPIO_Pin = USB_DISCONNECT_PIN;
14    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
15    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
16    GPIO_Init(USB_DISCONNECT, &GPIO_InitStructure);
17
18    /* Crea el GPIO de los Pines PA11 y PA12 que son las líneas DM y DP del USB */
19    RCC_AHBPeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
20    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12;
21    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_AF_PP;
22    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
23    GPIO_Init(GPIOA, &GPIO_InitStructure);
24
25    /* Para nuestra prueba con el LED de pruebas PC13
26       Se crea el GPIO del prin PC13                                    */
27    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
28    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
29    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
30    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
31    GPIO_Init(GPIOC, &GPIO_InitStructure);
32    GPIO_SetBits(GPIOC, GPIO_Pin_13);
33
34    /* Configura la línea 18 EXT para el control de la interrupción USB IP*/
35    EXTI_InitTypeDef EXTI_InitStructure;
36    EXTI_ClearITPendingBit(EXTI_Line18);
37    EXTI_InitStructure.EXTI_Line = EXTI_Line18;
38    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
39    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
40    EXTI_Init(&EXTI_InitStructure);
41
42    NVIC_InitTypeDef NVIC_InitStructure;
43
44    // Se establece el Grupo 2 para la configuración de prioridad */
45    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
46
47    /* Configuracion del NVIC para la interrupción USB             */
48    NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
49    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
50    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
51    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
52    NVIC_Init(&NVIC_InitStructure);
53
54    /* Habilitamos la bandera de la interrupción USBWakeUp          */
55    NVIC_InitStructure.NVIC_IRQChannel = USBWakeUp_IRQn;
56    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
57    NVIC_Init(&NVIC_InitStructure);
58
59  }
```

Figura 20.8

El contenido de este fichero y su función principal "**Set_System**()", son por defecto, los que vienen incluidos y se establecen en la propia librería cuando se descarga, no necesitando cambiar nada del mismo; ya que, los pines que se indican PA11 y PA12, son los que corresponden con la salida y la entrada de datos de nuestro microcontrolador para el módulo USB. No obstante, sí es posible añadir algún otro pin u opción que queramos en nuestro proyecto, como en nuestro caso, que añadimos la configuración del pin del LED de pruebas de nuestra placa.

20.2 EJEMPLO DE EMULACIÓN DE UN RATÓN Y UN TECLADO USB

En nuestro siguiente ejemplo, convertiremos nuestra placa en un ratón y un teclado USB; que al ser conectada a nuestro PC, este la detectará como tales y reproducirá no solo el movimiento de nuestro puntero en la pantalla sino también se introducirá un texto de prueba simulando las pulsaciones de un teclado desde nuestro microcontrolador.

El contenido del fichero main.c del ejemplo se muestra a continuación:

```
1   // Libreria principal del micro
2   #include "stm32f10x.h"
3
4   // Librerias de getion del puerto USB
5   #include "hw_config.h"
6   #include "usb_lib.h"
7   #include "usb_pwr.h"
8   #include "keycodes.h"
9
10  __IO uint8_t PrevXferComplete = 1;
11
12  extern uint8_t key_buf[9];
13
14  void MOUSE_move(int8_t x, int8_t y){
15        /*
16         * buf[0]: 1 - report ID
17         * buf[1]: bit2 - middle button, bit1 - right, bit0 - left
18         * buf[2]: move X
19         * buf[3]: move Y
20         * buf[4]: wheel
21         */
22        uint8_t buf[5] = {1,0,0,0,0};
23        buf[2] = x; buf[3] = y;
24        USB_SIL_Write(EP1_IN, buf, 5);
25        PrevXferComplete = 0;
26        SetEPTxValid(ENDP1);
27  }
28
```

```
29 void KEYBOARD_SEND_key_buf(void) {
30     USB_SIL_Write(EP1_IN, key_buf, 9);
31     PrevXferComplete = 0;
32     SetEPTxValid(ENDP1);
33     while (PrevXferComplete == 0)
34     {}
35 }
36
37 void KEYBOARD_SEND_Char(char ch) {
38     press_key(ch);
39     KEYBOARD_SEND_key_buf();
40     release_key();
41     KEYBOARD_SEND_key_buf();
42 }
43
44 void KEYBOARD_SEND_word(char *wrd){
45       do {
46             KEYBOARD_SEND_Char(*wrd);
47       } while(*(++wrd));
48 }
49
50
51 /*   Modulo principal                          */
52 //---------------------------------------------
53 int main(void)
54 {
55   Set_System();          // Inicializa la configuracion de periferi
56   USB_Interrupts_Config();  // Crea la interrupcion USB
57   Set_USBClock();        // Establece el reloj del USB
58   USB_Init();            // Inicializa el modulo USB
59
60   while (1)
61   {
62     // Comprueba si el USB esta operativo
63     if (bDeviceState == CONFIGURED)
64     {
65
66       if (PrevXferComplete)
67       {
68         // Reproduce la pulsacion de una frase con el teclado
69         KEYBOARD_SEND_word ("Hola desde STM32!!!");
70         MOUSE_move(1,-1);  // Reproduce un movimiento de raton
71
72       }
73     }
74   }
75 }
76
```

Figura 20.9

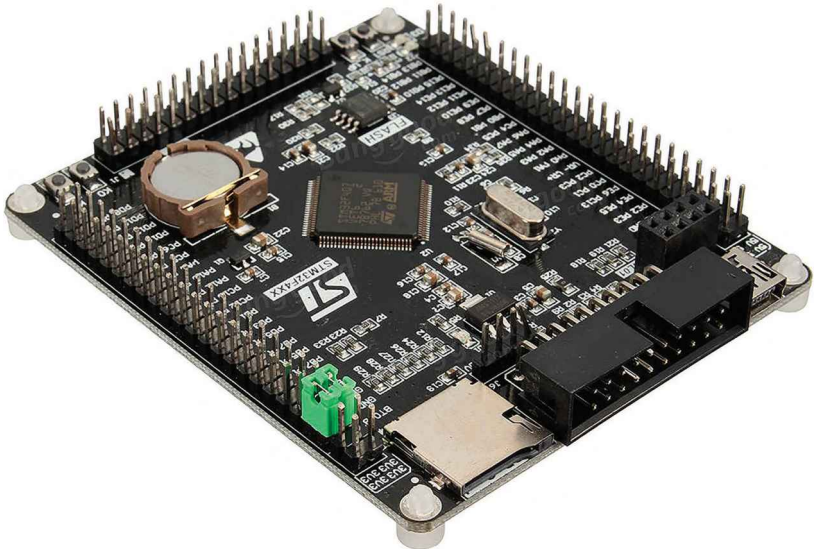


Figura 20.10

Existen innumerables posibilidades de conexiones que podemos realizar en nuestro microcontrolador STM32 pero hemos visto en este libro las más significativas. No hemos añadido en él, el puerto CAN que posee nuestro microcontrolador y que permite comunicarnos con los ordenadores de a bordo de los vehículos de hoy en día; ya que, este tipo de módulo precisa conectarnos con un vehículo para poder probarlo y ver así el funcionamiento de un código de ejemplo. Tampoco hemos añadido ninguna explicación de utilización del puerto WiFi que poseen algunos modelos de placas STM32 o de conexión a dispositivos como memorias SD, porque la placa que hemos usado de modelo para este libro no disponía de esas opciones.

Esperamos poder incluir todas estas opciones y explicarlas más detalladamente en una futura publicación, en la que utilizaremos microcontroladores de la familia STM32 superiores como el STM32F407VET6 que mostramos en la Figura 20.10.

Esta placa posee:

- Microcontrolador STM32F407VET6, Core-M4 de 32 bit RISC.
- Frecuencia máxima de funcionamiento a 168 MHz.
- Memoria Flash de 512 Kb y 192 SRAM.
- Puertos: 3 SPI, 3 USART, 2 UART, 2 I2C, 2 I2S, 2 CAN y 1 USB.

MATERIAL ADICIONAL

El material adicional de este libro puede descargarlo en nuestro portal web: *http://www.ra-ma.es*.

Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga. Dicha descarga consiste en un fichero ZIP con una contraseña de este tipo: XXX-XX-XXXX-XXX-X la cual se corresponde con el ISBN de este libro.

Podrá localizar el número de ISBN en la página IV (página de créditos). Para su correcta descompresión deberá introducir los dígitos y los guiones.

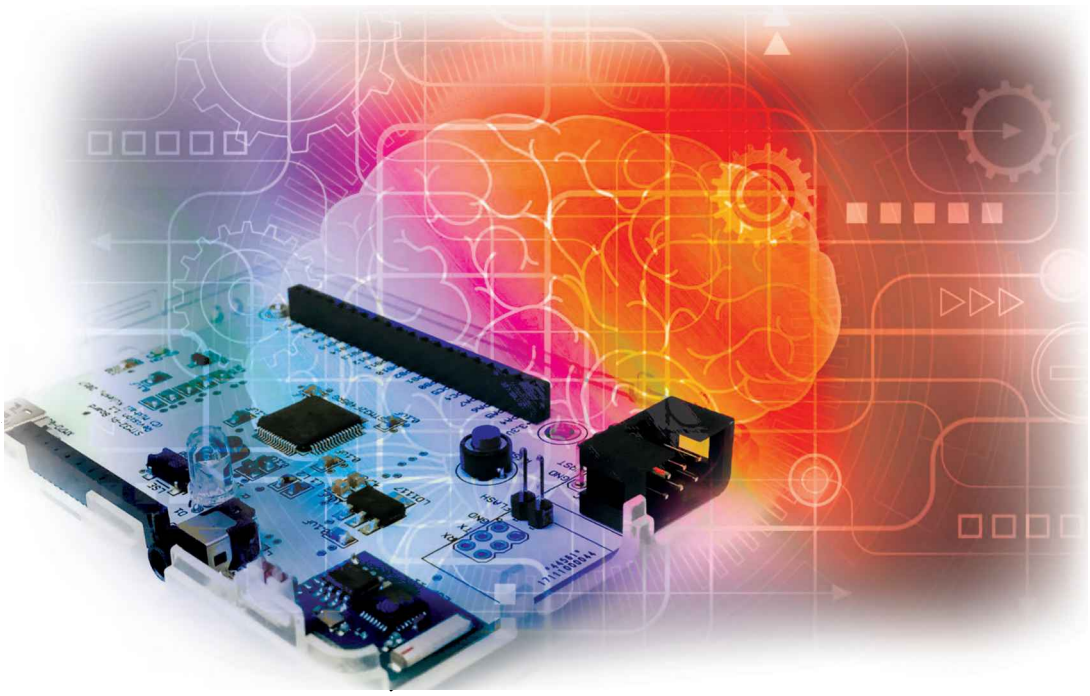
Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.

Microcontrolador STM32

Programación y Desarrollo



Jesús María Pestano Herrera



Desde **www.ra-ma.es** podrá descargar material adicional.

Ra-Ma®