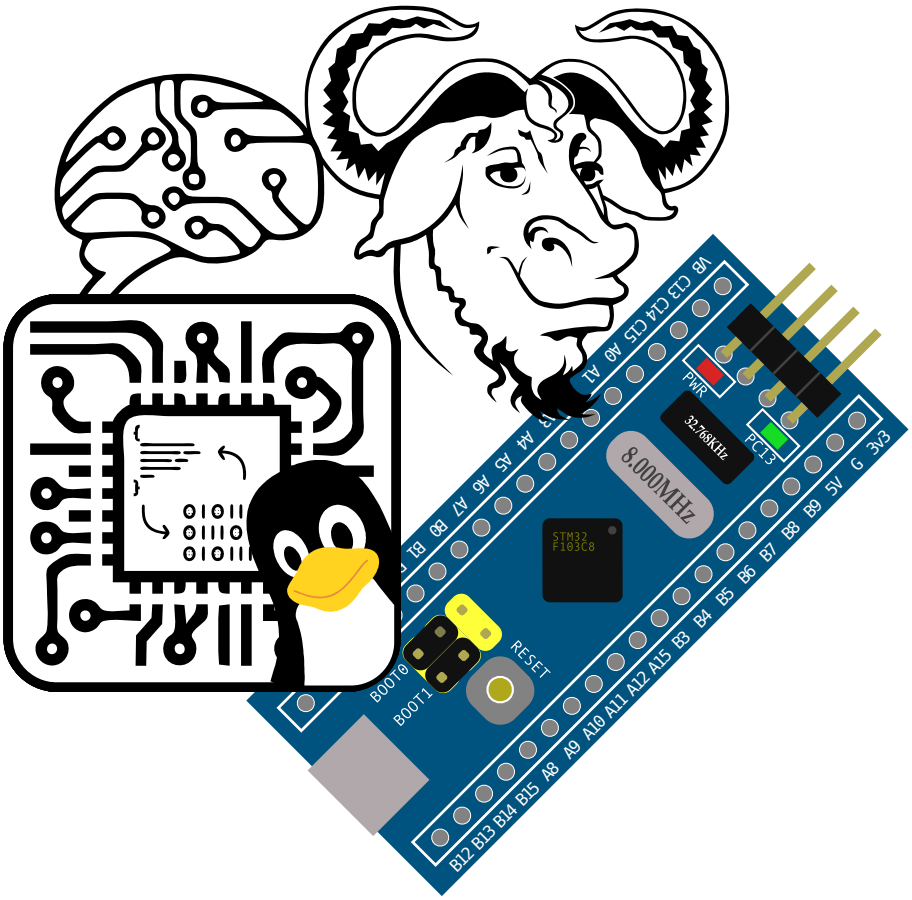
Programación de ARM® Cortex™-M con Software Libre



Prof. Matías S. Ávalos

Índice

Esta página fue dejada en blanco deliberadamente.

Unidad 0

Instalación y configuración de las herramientas

0.1. Sobre la materia y su objetivo

0.1.1. ¿De qué se trata?

En este curso se darán herramientas y recursos para programar, depurar y entender un microcontrolador (μC) de 32-bits ARM Cortex-M y la correcta forma de administrar sus recursos utilizando únicamente herramientas de **Software Libre**. Los estudiantes deberán aplicar conocimientos básicos de programación en C y Técnicas Digitales, como mínimo de lógica digital y estructuras básicas de control. Entre más conocimientos se tengan en esta área, será más sencillo abarcar con el contenido.

0.1.2. ¿Por qué *únicamente* con Software Libre?

Cada vez son más los que estamos en *educación, ciencia y tecnología* y adherimos a esta filosofía. Seguramente han escuchado sobre el Software Libre, y quizás, también de Hardware Open Source/Libre, un icono popular de eso es el proyecto Arduino (originalmente el *proyecto Wiring*). Pero, ¿qué significa que un Software sea Libre? Básicamente que respeta las *famosas* 4 libertades:

- **Libertad 0:** Usar el programa con cualquier propósito.
- **Libertad 1:** Estudiar cómo funciona el programa y poder modificarlo.
- **Libertad 2:** Distribuir copias del programa a cualquier persona u organización.
- **Libertad 3:** Mejorar el programa y compartir las mejoras en beneficio de todos.

Es entendible el porqué es deseable implementar Software Libre en educación, ya que nos da la posibilidad de estudiar, comprender y mejorar las mismas herramientas que utilizamos. Esto es un beneficio, incluso para las empresas que distribuyen *software privativo*, que ven, aprenden y mejoran sus propios programas gracias a esto. Incluso muchas grandes empresas de software desarrollan y colaboran en proyectos libres (Google, Amazon, Microsoft, Facebook, etc.).

En educación los motivos son varios, no solo el factor económico, ya que gran parte del SL es gratuito, también motivos éticos y legales (evitar el robo/pirateo de software). También hay motivos operativos, ya que suele aplicarse protocolos abiertos y cumplir con estándares libres de imposiciones o intereses de empresas. Esto asegura el acceeso a un conocimiento *agnóstico* con respecto de fabricantes que están interesados en que nos entrenemos en herramientas propias (y privativas) de uso exclusivo.

Ahora, ¿cómo funciona para el Hardware? Hay mucho debate al respecto, la definición más aceptada en materia de dispositivos electrónicos es la de **Hardware Open Source** que debe cumplir con los siguiente requisitos:

- Acceso al diseño esquemático/circuito.
- Acceso al diseño del PCB.
- Acceso al BOM (*build of materials*), listado de materiales.
- Documentación de armado y puesta a punto.

Es decir que todo lo empleado en los talleres de nuestras escuelas podría denominarse **Hardware Open Source**, ya que le damos a nuestros estudiantes todos estas herramientas. Aunque a veces el diseño del PCB es la práctica en cuestión. Para denominar al Hardware como **Libre**, algunos acordamos que debe cumplirse todo lo anterior y que además:

- Los archivos de CAD y BOM deben estar generados con Software Libre.

Este último requerimiento quiebra el hecho de que debamos recurrir de programas privativos para su elaboración (Altium, Autodesk EAGLE, etc.). Sin importar que estos sean gratuitos o que posean una versión *freeware*, licencias educativas, etc. Solo así podemos determinar que un proyecto de Hardware sea realmente *Libre*.

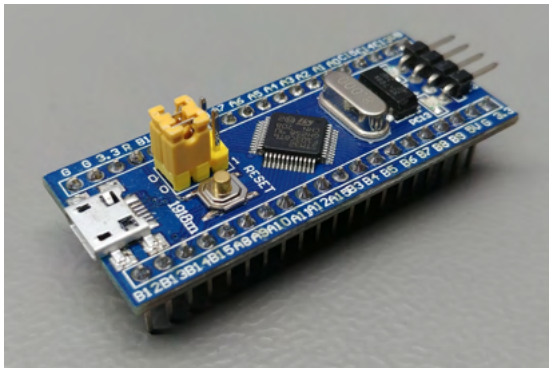
0.2. Hardware

A continuación se expondrá una rápida guía sobre el Hardware a utilizar en el curso. Se buscaron de forma tal que sean asequibles y favorecer y garantizar la sostenibilidad, escalabilidad y adaptación o migración a otras plataformas iguales o superiores.

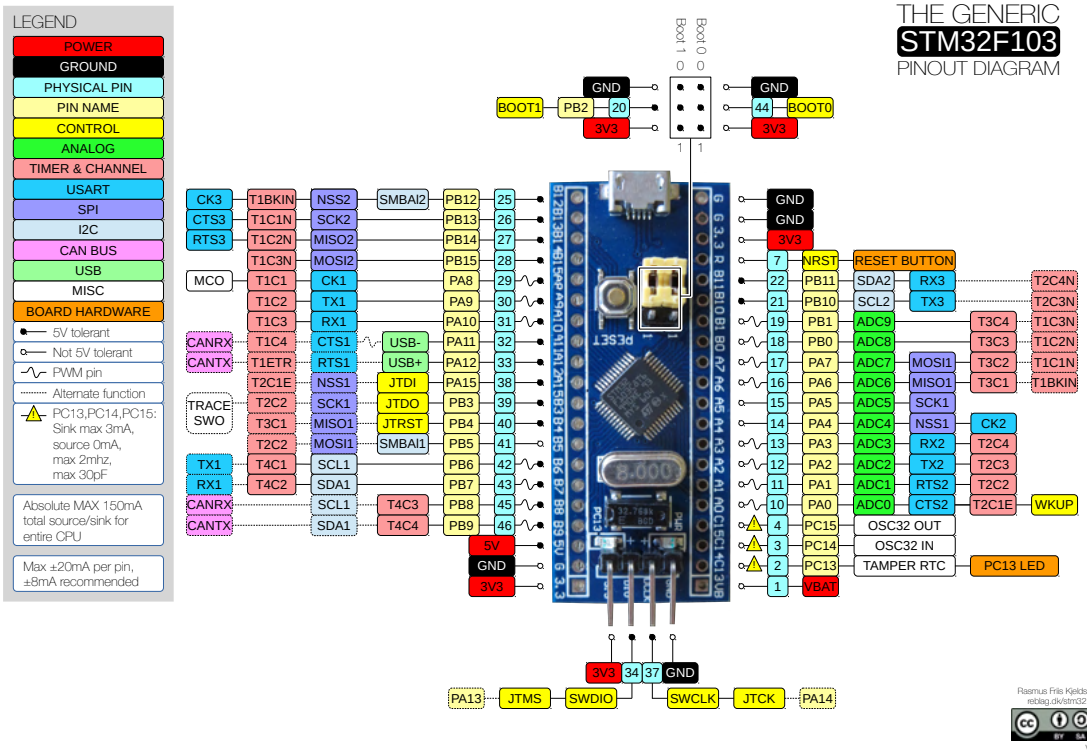
0.2.1. BluePill (STM32F103Cx)

Esta placa tiene cierta popularidad, es un diseño de origen chino que consta de un µC STM32F103C8/B. En la comunidad, se desarrolló un firmware compatible con le framework de Arduino, lo cual le garantizó una estabilidad en el mercado. Tiene un precio muy accesible y sus prestaciones básicas son las siguientes:

- **CPU:** ARM Cortex-M3
- **F.Max.:** 72Mhz
- **FLASH:** 64K/128K
- **SRAM:** 20K
- **E/S:** 31 pines
- **Timers:** 4 + 2 Watchdogs + SysTick
- **Comunicación:** 3 USART, 2 SPI, 2 I²C, CAN y USB2.0
- **Analógico:** 2 ADC (12 bits) y Temperatura
- **Otros:** RTC, Backup, 2 DMA, CRC



A continuación se deja la disposición una referencia de los pines y sus funciones:



Cabe destacar la mayoría de estos dispositivos viene con 128K de FLASH, pero debemos verificarlo (más adelante veremos cómo).

Atención

Algunas placas vienen con un circuito integrado clon CKS32/CS32, los cuales tienen una firma diferente para poder programarlos.

0.2.2. ST-Link V2 (opcional)

Este programador/depurador (debugger) utiliza el protocolo SWD (*Single Wire Debug*) que necesita tan solo 2 señales (SWDIO y SWCLK) para funcionar. Lo cual lo hace muy práctico, además de contar con la ventaja de ser económico. De más está decir que no hace falta tener uno por cada BluePill aunque a veces se obtienen en combos promocionales. La ventaja que nos da este dispositivo es poder ejecutar nuestro programa, correrlo paso a paso, ver el cambio en los registros y variables.

El dispositivo del que estamos hablando no es la herramienta oficial del fabricante, sino una versión *"clon"* por una empresa china que lleva por dentro un STM32F103C8.

0.2.3. Otras alternativas para grabar

De no contar con un ST-Link, aun podemos programarlo utilizando un **Serial USB/TLL**, y la señal `BOOT0` activa a través del jumper presente en la BluePill. Pero no podríamos utilizar la poderosa herramienta de debug.

Alternativamente como programador/depurador podemos grabar en una BluePill un firmware conocido como *Blackmagic*. Entonces debemos contar con 2 BluePills, una remplaza al ST-Link como depurador y la otra será en la que programaremos nuestra aplicación.

Otra opción es grabar en la BluePill un *bootloader* para poder utilizarla directamente a través del USB que viene soldado en la placa. Recomendamos utilizar dap boot (*DFU Bootloader for STM32 chips*). Aquí también perderemos la posibilidad del debugger.

Para más información sobre estos métodos consulte la lista de links y materiales (Subsección 0.6).

0.3. Software

La premisa de este curso es la utilización de Software Libre, por lo tanto a continuación se especificarán las herramientas las elegidas. La selección fue a partir de su simpleza, versatilidad y posibilidad multiplataforma (GNU/Linux, OSX y Windows).

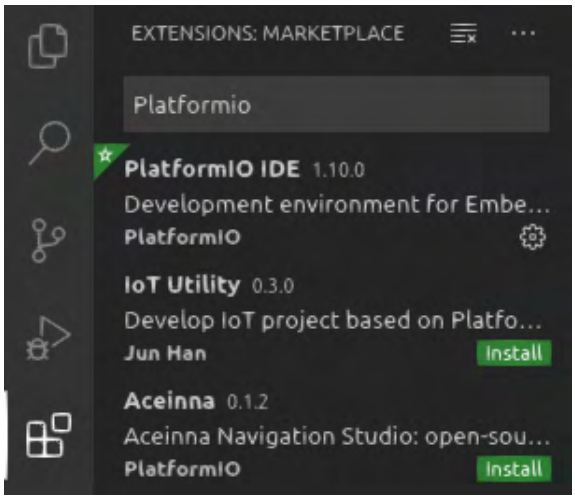
0.3.1. Visual Studio Code + PlatformIO

El IDE o Entorno de Desarrollo Integrado elegido fue PlatformIO, es un proyecto de Software Libre que se utiliza para programar diferentes tipos de hardware utilizando diversas herramientas de desarrollo con tan solo hacer un clic. Se encarga de descargar las mismas, para no tener que armar el *workflow* uno mismo (descargar el compilador, las librerías, etc.).

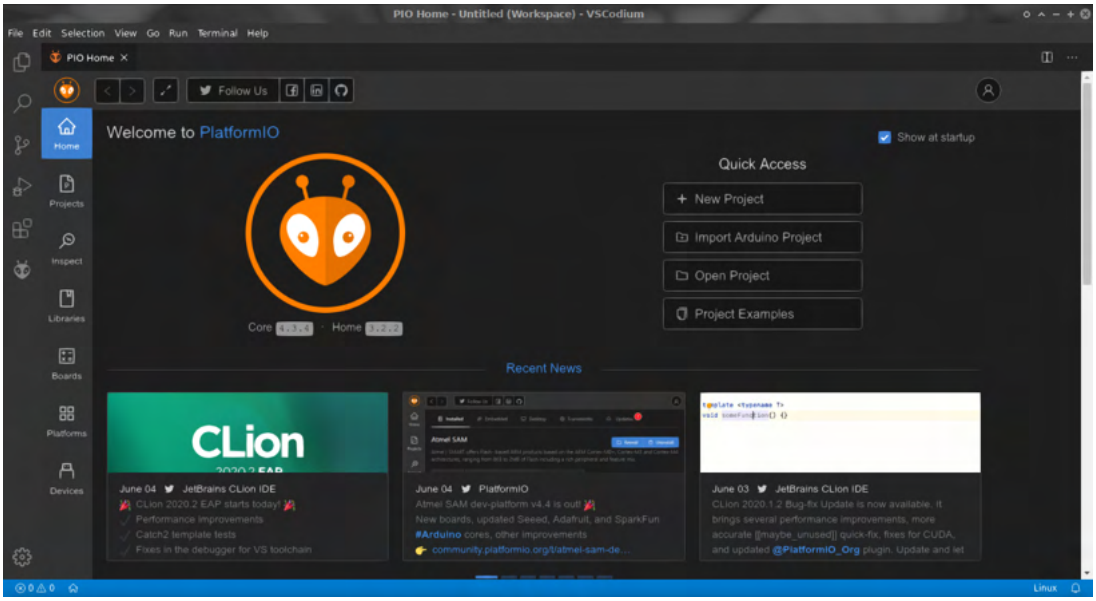
PlatformIO en sí es un software independiente que tiene la facilidad de ser instalado como plug-in de otros editores o IDEs. Las opciones son varias, por ejemplo: Atom, Micorsoft VSCode, el IDE CLion de JetBrains (Google), Eclipse, entre otros. Se tuvo presente utilizar la menor cantidad de recursos, es decir, que las herramientas de Software sean lo más livianas posibles para asegurar el funcionamiento en cualquier hardware de PC, por eso se opto por VSCode. El cual es muy sencillo de instalar desde su página.

Debe adicionalmente tener Python instalado en su sistema operativo y agregado al PATH. Que también cuenta con una instalación muy sencilla.

Una vez instalado debe ir a la parte de Extensiones y escribir *platformio* en el buscador e instalarlo haciendo clic en el botón verde:



Una vez instalado se reiniciará y se verá la siguiente página de inicio:



Para tener en cuenta

Si bien, la licencia del código fuente de VSCode es libre (MIT), sus binarios no son libres y tienen la telemetría habilitada por default. Como alternativa puede optar por **VSCodium** que es un binario libre de VSCode, pero requerirá una configuración extra del *marketplace* para poder instalar PlatformIO.

0.3.2. libOpenCM3

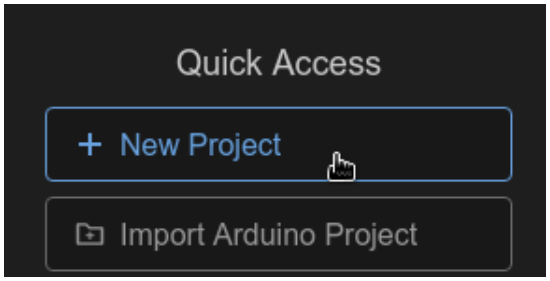
Antes de empezar a programar tenemos que definir las herramientas de desarrollo. El abanico es amplio, aunque PlatformIO tiene un segmento limitado de estas. Podemos optar por utilizar Arduino, CMSIS (la plataforma oficial de ARM), libOpenCM3, el HAL oficial de ST (STMCube) o un RTOS denominado Zephyr de Software Libre mantenido por Intel. Como uno de los objetivos es utilizar Software Libre, vamos a optar por libOpenCM3, pero no solo por ello.

Es un proyecto en desarrollo, que nació para proporcionar una API (*Application Programming Interface*) para µC STM32, pero hoy en día abarca muchas plataformas y fabricantes. Nos permite analizar el funcionamiento del µC y de cada uno de sus módulos, sin la necesidad de programar a nivel de registros (*bare-metal*) y, a su vez, muy cerca de la hoja de datos. Cuenta con documentación hecha en Doxygen con ejemplos de código. Lo cual, a futuro, nos podría llegar a brindar una interfaz extensible a otros dispositivos.

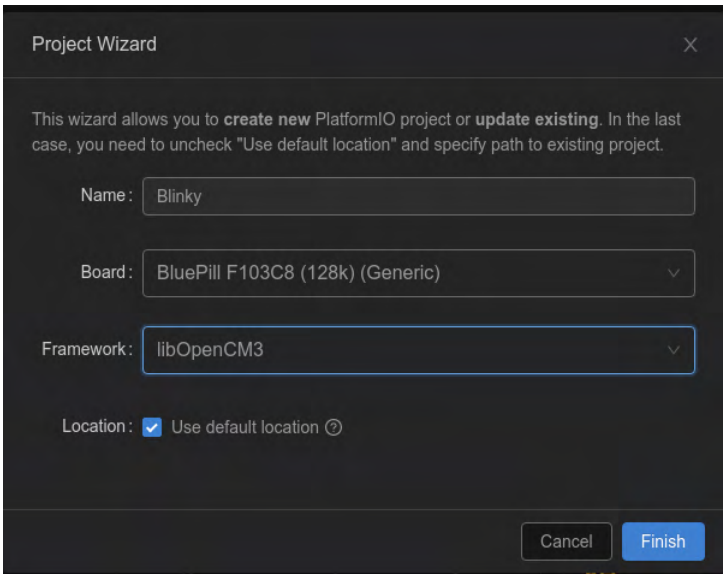
Ya que el objetivo es comprender el funcionamiento del dispositivo y sus periféricos internos (algo que queda muy lejos si utilizamos un HAL como el del Arduino o el oficial de ST) lo hace ideal para su uso pedagógico. Como alternativa recomendada se puede utilizar CMSIS, la interfaz oficial de ARM, pero esto es a nivel de registros.

0.4. Un primer ejemplo

Veremos cómo hacer nuestro primer proyecto con PlatformIO y libOpenCM3. Lo primero que debemos hacer es crearlo, haciendo clic en `+ New Project`:

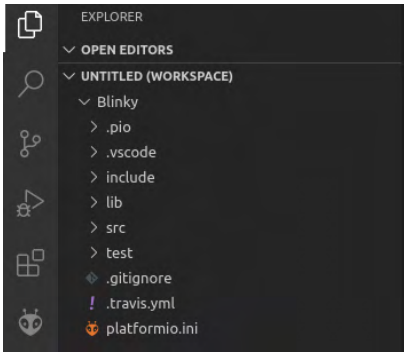


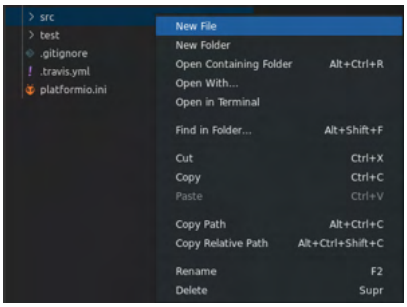
En la venta emergente debemos seleccionar un nombre para nuestro proyecto, la placa o µC a utilizar y la plataforma de desarrollo (*framework*) con el que deseamos trabajar:

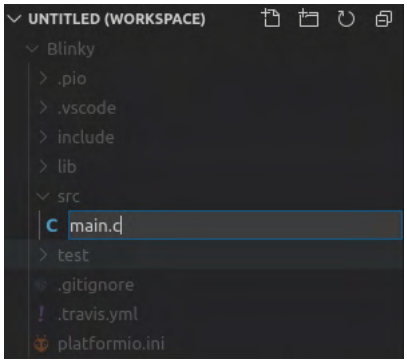


Cabe destacar que la primera vez, quizás tarde un poco porque debe descargar la base de datos de todas las placas que soporta PlatformIO, y luego de darle a Finish, debe descargar el compilador, programas necesarios para grabar el firmware y libOpenCM3. Pero esta demora será solamente la primera vez que lo hagamos.

Los proyectos en VSCode con PlatformIO tiene la siguiente estructura. Nosotros haremos clic con el derecho sobre la carpeta `src` y crearemos un nuevo archivo llamada `main.c`. Aunque cabe destacar que si quisiéramos podríamos nombrar el archivo con extensión `.cpp` si se deseara programar en C++ y *aprovechar* el paradigma de objetos.







Completaremos el `main.c` con el siguiente código para hacer el famoso *"¡Hola mundo!"* de los microcontroladores, el Blink (hacer parpadear un LED). La BluePill posee un LED integrado en el pin 13 del GPIOC (PC13):

main.c

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>

int main()
{
    // Se configura el clock del sistema a 72Mhz
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    // Se habilita el clock en el periférico GPIOC
    rcc_periph_clock_enable(RCC_GPIOC);

    // Se configura el PC13 como salida push-pull (2Mhz max)
    gpio_set_mode(
        GPIOC,
        GPIO_MODE_OUTPUT_2_MHZ,
        GPIO_CNF_OUTPUT_PUSHPULL,
        GPIO13);

    while (true)
    {
        // Se invierte el PC13
        gpio_toggle(GPIOC, GPIO13);

        // Demora a ojo
        for(uint32_t i=0; i < 7200000;++i)
            __asm__("nop");
    }
}
```

Para poder compilar y grabar este sencillo ejemplo podemos utilizar los botones que nos proporciona PlatformIO en una pequeña barra de tareas ubicada abajo a la izquierda donde veremos los siguientes iconos:

El ST-Link/V2 es la herramienta por omisión (default) que PlatformIO buscará para grabar el firmware. Y si todo salió bien, deberíamos poder ver titilar el PC13 en la BluePill. Las funciones utilizadas serán analizadas en la próxima unidad para su comprensión.

Diremos rápidamente que se configuró el clock del sistema para utilizar el cristal presente en la placa de 8MHz y por medio del PLL llegar al máximo de 72Mhz. Luego se habilitó el `GPIOC` y se configuro el pin 13 (`GPIO13`) como salida Push-Pull y por último en el bucle principal se alterna el mismo haciendo demoras con un repetición de operaciones `NOP` en assembler.

0.5. Troubleshooting

En los casos de no poseer el ST-Link, entonces debemos especificar en el archivo `platformio.ini` la herramienta elegida:

```
; Para Blackmagic:
upload_protocol = blackmagic

; Para grabar por medio de la USART1=> PA9-PA10 (BOOT0=1, BOOT1=0)
upload_protocol = serial

; Para dap boot por medio del USB (BOOT0=0, BOOT1=1)
upload_protocol = dfu
upload_command = $PROJECT_PACKAGES_DIR/tool-dfuutil/bin/dfu-util -D $SOURCE
```

En caso de tener el ST-Link y un chip *clon* con la firma `0x2ba01477` podemos especificar que al grabar se utilice esta firma añadiendo en el `platformio.ini` el siguiente comando:

```
upload_flags = -c set CPUTAPID 0x2ba01477
```

Es recomendable en sistemas GNU/Linux agregar el archivo de reglas udev para dispositivos USB. La forma recomendada para hacer esto es ejecutando el comando `curl`:

```
curl -fsSL https://raw.githubusercontent.com/platformio/platformio-core/master/scripts/99-
    platformio-udev.rules | sudo tee /etc/udev/rules.d/99-platformio-udev.rules
```

Es posible que el dispositivo al que le grabemos el firmware de *dap-boot* no funcione al menos que agreguemos al archivo anterior o a un nuevo `.rules` lo siguiente:

```
# dap boot DFU
ATTRS{idVendor}=="1209", ATTRS{idProduct}=="db42", MODE:="0666", ENV{ID_MM_DEVICE_IGNORE}=
    "1", ENV{ID_MM_PORT_IGNORE}="1"
```

0.6. Links y materiales de la unidad

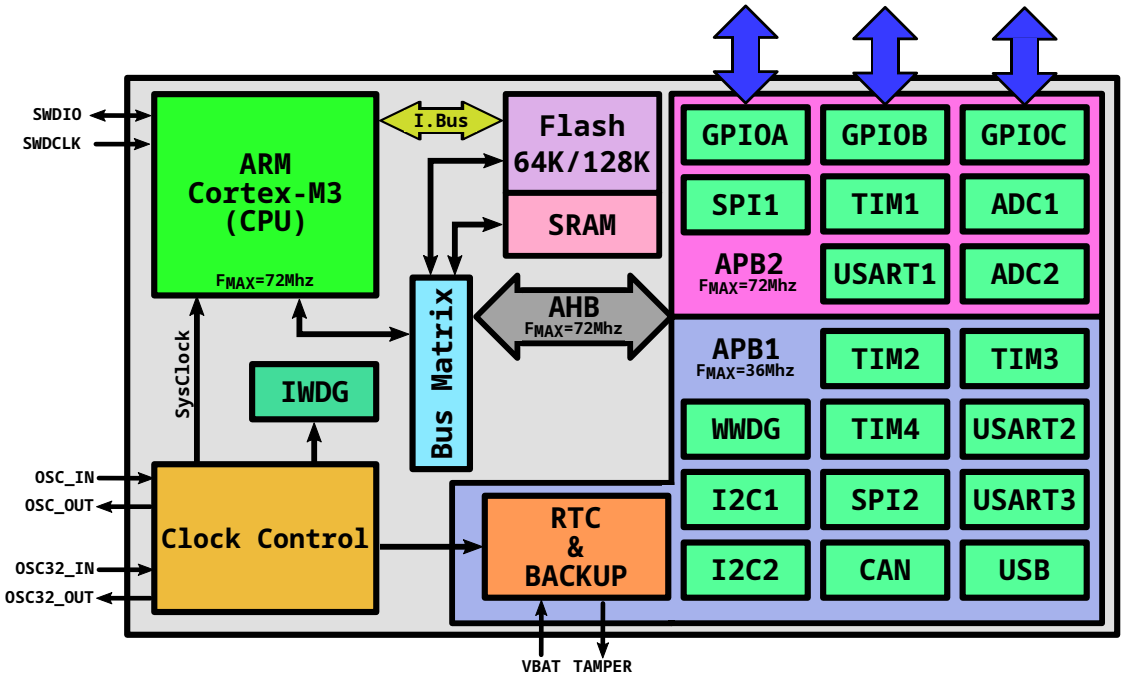
- **PlatformIO**
  https://platformio.org/
- **VSCode**
  https://code.visualstudio.com/
- **Python**
  https://www.python.org/downloads/
- **VSCodium**
  https://vscodium.com/
- **Referencias sobre la BluePill**
  https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill
- **Referencias sobre el ST-Link/V2**
  https://stm32-base.org/boards/Debugger-STM32F101C8T6-STLINKV2
- **Documentación de libOpenCM3**
  http://libopencm3.org/docs/latest/stm32f1/html/modules.html
- **Black Magic Probe**
  https://github.com/blacksphere/blackmagic
- **dap boot**
  https://github.com/devanlai/dapboot
- **Datasheet STM32F103C8/B**
  https://www.st.com/resource/en/datasheet/stm32f103c8.pdf
- **Manual de referencia STM32F1xx**
  https://tinyurl.com/yxv9m5b2
- **Proyecto Wiring**
  http://wiring.org.co/
- **Troubleshooting oficial de PlatformIO**
  https://docs.platformio.org/en/latest/faq.html#troubleshooting

Unidad 1

Programación del Clock pincipal y GPIOs

1.1. El diagrama interno en bloques

Toda hoja de datos de un microcontrolador (µC) debe poseer un diagrama en bloques de su estructura general y de sus módulos y periféricos internos. Aquí se expondrá una versión simplificada, pero es recomendable observar la original para tener una visión más detallada y precisa.

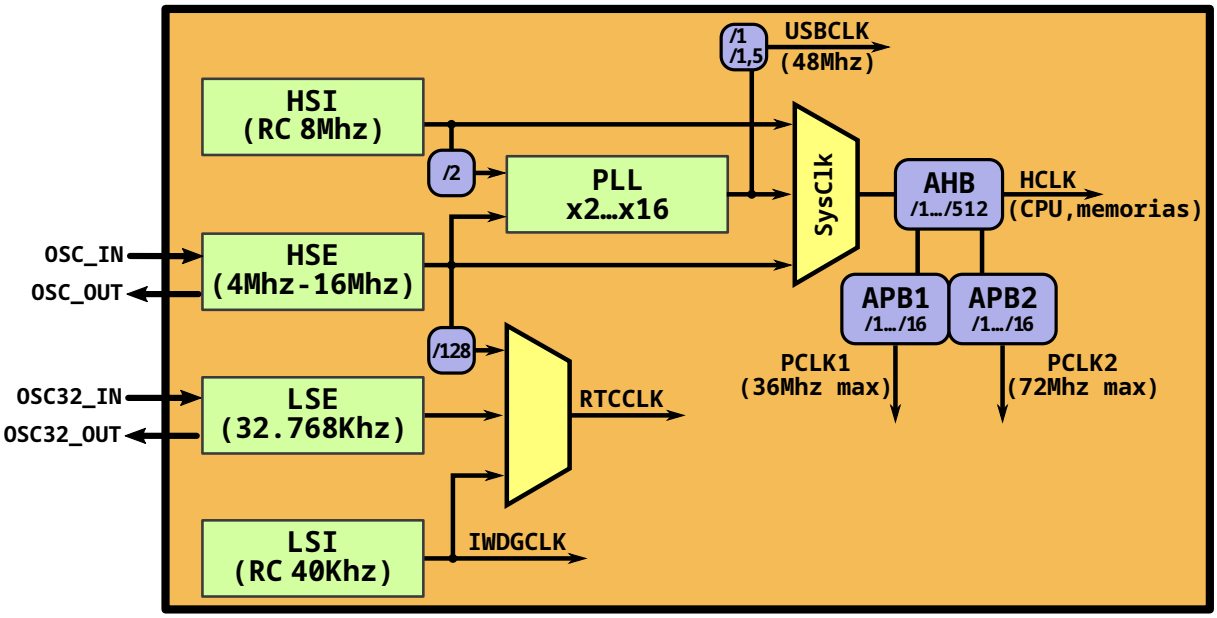


En principio se debe destacar que los buses de comunicación ya no son simples hilos que transportan la información, sino que tienen una complejidad superior debido a la cantidad de periféricos y posibilidades que nos brindan los **ARM Cortex-M** en general. Así es que la comunicación se hace a través de un *Bus Matrix* el cual se conecta al CPU, las memorias y a un **AHB** *(Advanced High-performance Bus)*, el cual puede funcionar a un máximo de 72Mhz en este µC. Luego los periféricos internos están separados en dos **APB** (*Advanced Peripheral Bus*), de los cuales el *APB1* puede funcionar a una F_{MAX} de 36Mhz y en el *APB2*, donde se encuentran los **Puertos/GPIOs** de E/S, que puede funcionar a la misma frecuencia que el *AHB*.

El *controlador de clocks* es un módulo de hardware que es altamente complejo, permite manejar las frecuencias del CPU, Buses, ADCs, RTC, USB y *WatchDogs* con prescalers separados, además de contar con un **PLL** (*Phase-Locked Loop*) para multiplicar la frecuencia del clock. Todo esto lo convierte en un tema de análisis que llevaría una unidad completa, pero solo focalizaremos en algunas características y profundizaremos de ser necesario a medida que lo necesitemos en otras unidades.

1.1.1. Fuentes de reloj (clock)

Como se adelantó, el *controlador de clocks* es un tema bastante complejo de abarcar. Aquí se hará foco en lo que respecta a el origen de donde se toman las señales para repartirlas a través de todo el µC y sus periféricos.



Como se observa en la imagen hay 3 fuentes para el **Clock del Sistema** (*SysClk*):

- **HSI (*High-Speed Internal*):** Es un resonador RC interno fijo a 8Mhz. Si no se hace ninguna configuración previa, esta es la fuente de clock después de un **reset**.
- **HSE (*High-Speed External*):** Para utilizar esta fuente de clock es necesario proveer por medio de los pines `OSC_IN` y `OSC_OUT` un resonador/cristal entre 4Mhz y 16Mhz o una fuente externa por `OSC_IN` hasta 25Mhz.
- **PLL:** Este multiplicador puede elevar tanto el **HSI/2** como el **HSE** para alcanzar frecuencias superiores. Si el **PLL** se sirve del **HSI/2** solo puede alcanzar 64Mhz y si la fuente es el **HSE** llega a los 72Mhz (F_{MAX}). Sólo es posible utilizar el **USB** si el **PLL** se sirve del **HSE** y el `USBCLK` debe ser 48Mhz lo cual se logra a través de un *prescaler* que se configura por 1 o 1,5.

Del *SysClk* se desprenden los clocks para el **AHB** y los respectivos **APB** que habilitan a los diferentes periféricos (`PCLK1` y `PCLK2`). Es importante entender que éstos **no están habilitados después de un reset**, eso quiere decir que *debemos habilitar el clock para cada uno de ellos a medida que se requiera.*

Por otro lado, hay otro oscilador que sirve al Reloj de Tiempo Real (**RTC** - *Real-Time Clock*) que viene integrado dentro del µC. Éste tiene 3 fuentes diferentes:

- **LSI (*Low-Speed Internal*):** Es un resonador RC interno fijo a 40Khz. La ventaja de este método es que no necesitamos elementos externos para utilizar el **RTC**.
- **LSE (*Low-Speed External*):** Para utilizar esta fuente de clock es necesario proveer por medio de los pines `OSC32_IN` y `OSC32_OUT` un cristal de 32.768Khz. Este método es el más recomendable y fiable cuando necesitamos precisión en nuestro reloj.
- **HSE/128:** Tras un divisor por 128 podemos tomar el valor del **HSE** para la fuente del **RTC**. La ventaja sería tener la precisión de un cristal (aunque no tan bueno como el del **LSE**) y no ocupar los pines de `OSC32_IN` y `OSC32_OUT`.

Por último, en esta figura se observa como el **IWDG** (*Independent WatchDog*) puede servirse del **LSI**. Un *WatchDog* es un temporizador, el cual protege nuestra aplicación evitando que *"se cuelgue"*. La ventaja de que tenga una fuente de clock independiente es que este puede seguir funcionando por más que el *SysClk* no esté habilitado. Abordaremos el tema de los *WatchDogs* y otras mecánicas de seguridad más adelante.

1.2. Reset & Clock Control (*RCC*)

El **RCC** es el módulo de hardware que se encarga de controlar todo lo analizado en la sección anterior. Como todo módulo de hardware o periférico interno de cualquier µC, estas configuraciones se hacen a través de *Registros* que ocupan direcciones de memoria específicos, que suelen ser nombrados como **SFR** (*Special Function Registers*).

Por el momento mencionaremos que estos registros serán nuestra fuente de información principal para poder configurar cualquier característica que tenga el µC. La información que escribamos y leamos de estos espacios de memoria activarán los diferentes mecanismos que hacen al funcionamiento del módulo específico a programar.

1.2.1. Fuentes de Reset

En este caso el módulo está encargado de informarnos las fuentes del **Reset**, es decir qué causó que el µC se haya *reseteado*. Para el denominado *Reset del Sistema* existen estas posibilidades:

1. **Reset Externo:** una señal en bajo (LOW) en el pin externo `nRST`.
2. **WWDG reset:** que el *WatchDog de ventana* (**WWDG** - *Window WatchDog*) haya llegado al final de la cuenta.
3. **IWDG reset:** que el *WatchDog independiente* (**IWDG**) llegue al final de la cuenta.

4. **Software Reset:** hay mecanismos para causar un reset con nuestro programa.

5. **Reset de gestión de baja energía:** este tipo de reset es generado especialmente al entrar al modo de bajo consumo si fue configurado previamente.

Hay otros tipos de *reset*, por ejemplo el de **BackUp**, pero no se analizarán aquí, puede buscar más información en el *Reference Manual* de ST.

1.2.2. Selección de SysClk

El RCC también nos ayuda a configurar y seleccionar las diversas fuentes de clock que ya fueron analizadas. Para este propósito existen diversos registros dependiendo de la tarea y hay procedimientos de cómo configurar y calibrar los distintos osciladores.

En libOpenCM3 para poder utilizar estas características debemos incluir la cabecera *header*:

```
#include <libopencm3/stm32/rcc.h>
```

El cual nos provee de las siguientes funciones para la configuración de los clocks en modos estándar:

```
void rcc_clock_setup_in_hsi_out_64mhz(void);
void rcc_clock_setup_in_hsi_out_48mhz(void);
void rcc_clock_setup_in_hsi_out_24mhz(void);
void rcc_clock_setup_in_hse_8mhz_out_24mhz(void);
void rcc_clock_setup_in_hse_8mhz_out_72mhz(void);
void rcc_clock_setup_in_hse_12mhz_out_72mhz(void);
void rcc_clock_setup_in_hse_16mhz_out_72mhz(void);
void rcc_clock_setup_in_hse_25mhz_out_72mhz(void); // solo con clock externo (en OSC_IN)
```

En estas configuraciones se deja al **AHB**, **APB1** y **APB2** en las máximas frecuencias posibles. Estas pueden ser consultadas en 3 variables globales denominadas `rcc_ahb_frequency`, `rcc_apb1_frequency` y `rcc_apb2_frequency` respectivamente. Si desea puede ver la implementación de cada una de ellas para darse una idea de cómo es el procedimiento de activar el **PLL** y demás configuraciones necesarias y crear una propia en caso de que no haya una función que tenga las características deseadas.

En general nosotros siempre utilizaremos la que indica usar el **HSE** como entrada con el cristal de 8Mhz soldado a la placa de la BluePill (`in_hse_8mhz`) y genera un *SysClk* de 72Mhz (`out_72mhz`).

1.2.3. Habilitación de periféricos

Como se adelantó anteriormente, los periféricos están inicialmente con el clock inhabilitado. Por lo tanto debemos habilitar el clock correspondiente. Para ello se utiliza la función:

```
void rcc_periph_clock_enable(enum rcc_periph_clken clken);
```

Que como observamos recibe como parametro un tipo `enum rcc_periph_clken`, los cuales todos inician con `RCC_`, por ejemplo:

```
RCC_GPIOA,RCC_GPIOB,RCC_ADC1,RCC_SPI1,RCC_USART1 // etc...
```

1.3. GPIOs & AFIOs

Las Entradas/Salidas de Propósito General (**GPIOs** - *General-Purpose I/O*) representan los pines del integrado, los cuales están agrupados en los `GPIOA`, `GPIOB` , `GPIOC` y `GPIOD`. Estos grupos son de 16 pines, del 0 al 15, siendo por ejemplo `PA0` el pin 0 del `GPIOA`, `PB8` el pin 8 del `GPIOB`, `PC13` el pin 13 del `GPIOC`, etc. Como la familia del STM32F103 viene en varios encapsulados, con diversas cantidades de pines, es normal que no encontremos todos disponibles en el encapsulado de 48 pines que tiene la BluePill.

1.3.1. Modos y configuraciones

Los pines de los **GPIOs** pueden tener las siguientes modos:

- **Entrada (`INPUT`)**
- **Salida (`OUTPUT`):**
  - 10Mhz max
  - 2Mhz max
  - 50Mhz max

Las diferencias de frecuencia estructuran el circuito de salida diferente, lo cual permite regular el consumo eléctrico del µC. Además Tanto de entrada como de salida, tenemos diferentes configuraciones:

- **Como entrada:**
  - **Analógica:** Funciona como canal para el `ADC1` y `ADC2`
  - **Flotante:** estado por (*default*) en flotante/alta impedancia (**Hi-Z**)
  - **Pull-Up/Down:** podemos configurar internamente un *pull-up* o *pull-down*.
- **Como salida:**
  - **GPIO Push-Pull:** salida de propósito general que entrega corriente desde el mismo µC.
  - **GPIO Open-Drain:** salida de propósito general con *drenador abierto*.
  - **AFIO Push-Pull:** salida alternativa que entrega corriente desde el mismo µC.
  - **AFIO Open-Drain:** salida alternativa con *drenador abierto*

1.3.2. Funciones alternativas

Como se acaba de exponer, existen *modos alternativos* para los pines denominados **AFIOs** (*Alternate-Function I/O*). Todos los pines del integrado están asociados a **GPIOs**, pero otros periféricos también necesitan acceso al exterior para poder funcionar. Por ejemplo, una USART utiliza como mínimo 2 pines: uno de transmisión (T_X) y otro de recepción (R_X) para enviar y recibir datos. Para el receptor basta con dejar la entrada en su estado por *default* (**entrada flotante**) para que el periférico pueda tomar posesión, pero en cuanto a T_X, debemos indicar explícitamente que deseamos usar la *función alternativa push-pull* correspondiente a este periférico.

Los pines utilizados por estos periféricos de funciones alternativas están indicados en varias tablas en el *Manual de Referencia* de ST. A medida que vayamos analizando los diferentes periféricos iremos indicando los pines correspondientes a cada uno de ellos.

Como dato adicional, con algunos periféricos es posible *"remapear"* estos pines alternativos. Es decir, se puede cambiar de lugar los pines a otros preestablecidos.

1.3.3. Programación

Para poder utilizar los **GPIOs** (y **AFIOs**) debemos incluir la siguiente cabecera:

```
#include <libopencm3/stm32/gpio.h>
```

Las funciones y defines que nos provee libOpenCM3 para la configuración y utilización de los **GPIOs** es bastante intuitiva si se conoce todo lo anteriormente mencionado. Por ejemplo, para configurar utilizaremos:

```
void gpio_set_mode(uint32_t gpioport, uint8_t mode, uint8_t cnf, uint16_t gpios);
```

En cuyos parámetros utilizaremos alguna combinación de los siguientes argumentos:

```
/* Argumentos para 'gpioport' */
GPIOA,GPIOB,GPIOC,GPIOD

/* Argumentos para 'mode' */
GPIO_MODE_INPUT
GPIO_MODE_OUTPUT_10_MHZ
GPIO_MODE_OUTPUT_2_MHZ
GPIO_MODE_OUTPUT_50_MHZ

/* Argumentos para 'cnf' */
GPIO_CNF_INPUT_ANALOG
GPIO_CNF_INPUT_FLOAT
GPIO_CNF_INPUT_PULL_UPDOWN
GPIO_CNF_OUTPUT_PUSHPULL
GPIO_CNF_OUTPUT_OPENDRAIN
GPIO_CNF_OUTPUT_ALTFN_PUSHPULL
GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN

/* Argumentos para 'gpios'. Puede ser 1 o varios separados por | */
GPIO0,GPIO1,GPIO2,GPIO3,GPIO4,GPIO5,GPIO6,GPIO7,
GPIO8,GPIO9,GPIO10,GPIO11,GPIO12,GPIO13,GPIO14,GPIO15,
GPIO_ALL
```

Las funciones que tenemos a disposición tienen un comportamiento **de grupo**, es decir que podemos escribir y leer varios pines al mismo tiempo, un ejemplo de `gpio_set_mode()` sería:

```
// Se configura el PB8 y PB9 como Salidas Push-Pull a 2Mhz max:
gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8|GPIO9);
```

En cuanto a la lectura y escritura o sensado y control, tenemos las siguientes funciones listadas a continuación:

```
/* Funciones de control/escritura */
void gpio_set(uint32_t gpioport, uint16_t gpios);   // escribe un 1 en los gpios
void gpio_clear(uint32_t gpioport, uint16_t gpios); // escribe un 0 en los gpios
void gpio_toggle(uint32_t gpioport, uint16_t gpios);// alterna el estado de los gpios

/* Función de sensado/lectura */
uint16_t gpio_get(uint32_t gpioport, uint16_t gpios);// obtiene el valor de los gpios
```

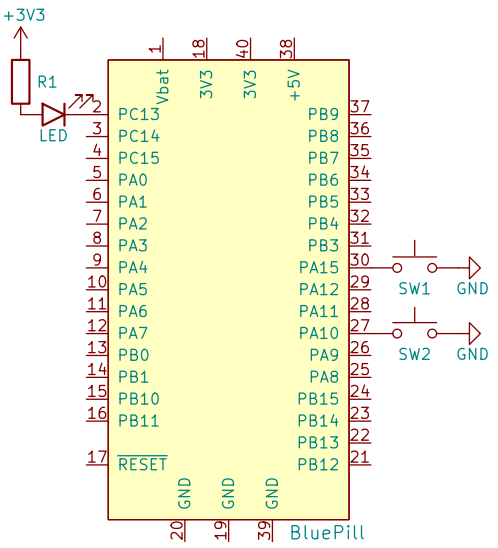
Si quisiéramos utilizar una entrada con Pull-Up o Pull-Down, esto dependerá del valor en que se encuentre el registro de salida (**ODR** - *Output Data Register*), eso quiere decir que si tenemos un pin configurado como entrada y escribimos un 1 con `gpio_set()` habilitaremos el Pull-Up, en caso contrario, si escribimos un 0 en ese pin con `gpio_clear()` habilitaremos el Pull-Down.

```
// Se configura el PA5 como Entrada con Pull-up/down.
gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_PULL_UPDOWN, GPIO5);
// Se habilita el Pull-up
gpio_set(GPIOA, GPIO5);
```

Entonces si se quisiera saber el estado del PA7, utilizaremos la función `gpio_get`, la cual nos devolverá un entero sin signo de 16 bits (`uint16_t`), en el cual debemos analizar el bit en la posición 7 (los demás estarán en 0):

```
uint16_t estado_pa7 = gpio_get(GPIOA, GPIO7);
// La variable estado_pa7 solo puede tener 2 valores posibles:
//  ESTADO          BINARIO          HEX      DEC
//  PA7=1  => '0b0000000010000000' = 0x0080 =  127
//  PA7=0  => '0b0000000000000000' = 0x0000 =   0
```

En el siguiente ejemplo se conectan 2 pulsadores al **GPIOA**, aprovechando los Pull-Ups internos para evitar utilizar componentes externos. Si bien en el circuito figura un LED externo es el que ya está conectado en la BluePill, el cual tiene *lógica activa en BAJO*, es decir que con un estado BAJO (0) enciende y con un estado ALTO (1) estará apagado. Los pulsadores también utilizan lógica activa en BAJO, ya que si no están presionados estará el Pull-Up interno (a 1) y al presionar el pulsador el pin estará conectado a GND (en 0).



2puls.c

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>

int main(void)
{
    // AHB=72Mhz; APB1=36Mhz; APB2=72Mhz;
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    /********** CONFIGURACIÓN DE SALIDAS  **********/
    // Habilitación del PCLK para el GPIOC
    rcc_periph_clock_enable(RCC_GPIOC);
    // PC13 como salida PUSH-PULL
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);

    /********** CONFIGURACIÓN DE ENTRADAS **********/
    // Habilitación del PCLK para el GPIOA
    rcc_periph_clock_enable(RCC_GPIOA);
    // PA10 y PA15 como entrada con Pull-up/down
    gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_PULL_UPDOWN, GPIO10 | GPIO15);
    // Se habilitan los Pull-Ups en PA10 y PA15
    gpio_set(GPIOA, GPIO10 | GPIO15);

    while (true)
    {
        if (gpio_get(GPIOA, GPIO10) == 0)      // Si el pulsador en PA10 está presionado:
            gpio_clear(GPIOC, GPIO13);         //    se prender el LED en PC13.
        else if (gpio_get(GPIOA, GPIO15) == 0) // Sino-Si el pulsador en PA15:
            gpio_set(GPIOC, GPIO13);           //    se apaga el LED en PC13.
    }
}
```

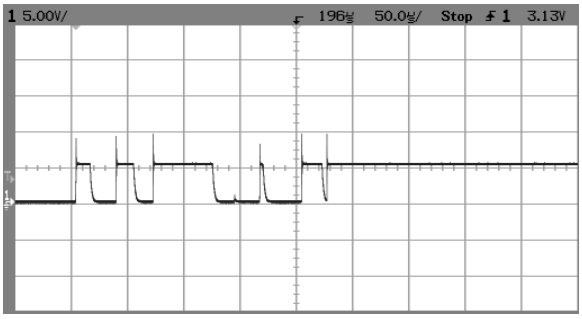
Se debe tener cuidado y no confundir que la función `gpio_get` devuelva simplemente un 0 o un 1, recordemos que devuelve un `uint16_t` donde solo debemos verificar los bits correspondientes a la posición del pin (los demás estarán en 0). Por ejemplo, reutilizando el circuito del programa anterior, se quisiera modificar para que el LED prendiera si ambos pulsadores estuvieran **sin presionar** y apagado en caso contrario:

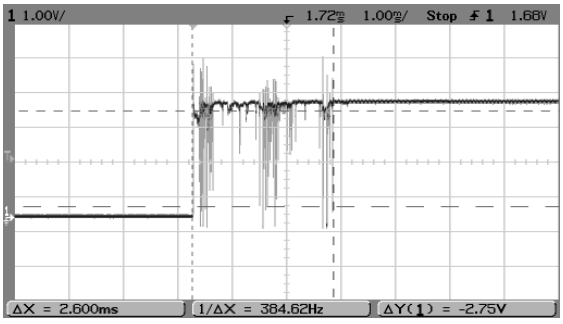
```c
while (true)
{                       // eqivalencia en hex: 0x8400
    if (gpio_get(GPIOA, GPIO10 | GPIO15) == 0b1000010000000000)
    {
        gpio_clear(GPIOC, GPIO13);      //  bit15 y bit10 en 1
    }
    else
    {
        gpio_set(GPIOC, GPIO13);
    }
}
```

En el ejemplo anterior, se utilizó un número constante para explicitar las posiciones de los pines con respecto a los bits, pero es una práctica desaconsejada. Lo mejor sería plantear la pregunta como:

```c
if (gpio_get(GPIOA, GPIO10 | GPIO15) == (GPIO10 | GPIO15) )
```

Al momento de sensar pulsadores mecánicos existen ruidos asociados a la calidad de los mismos, conocidos como **rebote** (*bounce*).



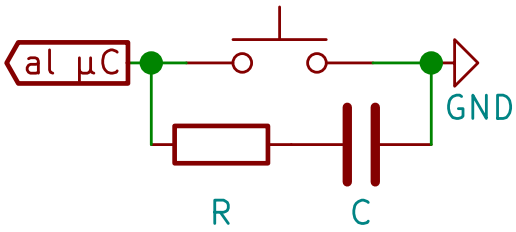


Imágenes proporcionadas por Super Rad!

Para evitar esto suelen haber varias técnicas, la más difundida (*y peor opción*) es hacer una demora una vez detectado el cambio en el pin asociado al pulsador. Lo cual no asegura que no tengamos falsas pulsaciones detectadas. Por otro lado podría utilizar un capacitor y una resistencia a modo de filtro en paralelo con el pulsador. A lo largo de los ejemplos veremos más técnicas para evitar este problema.

```
if(gpio_get(GPIOB, GPIO8) == 0)
{
    // demora anti-rebote ("debounce")
    for(uint16_t i=0; i < 1000000 ; i++)
        __asm__("nop");

    // ...
}
```

Anti-rebote (*debounce*) por software.

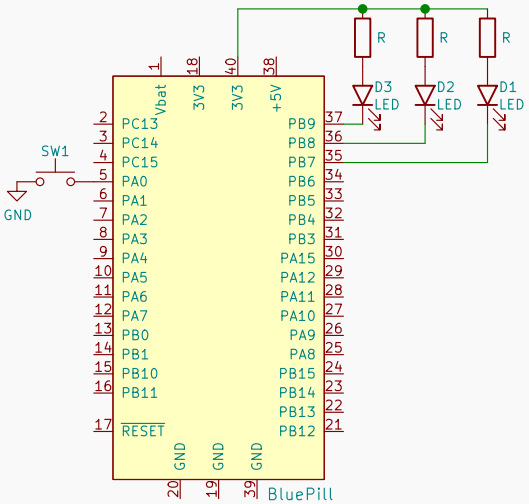


Anti-rebote (*debounce*) por hardware.

1.4. Ejercicios

EJERCICIO N°1

Dado el siguiente circuito, desarrolle los programas solicitados a continuación.



1. Enciendan los LEDs en secuencia: D1, luego el D2 y finalmente el D3 hasta que todos estén prendidos. Luego se apaguen en el mismo orden. Utilizar demoras bloqueantes entre cada uno de los estados con bucles y operadores nop.

2. Idem anterior, pero que luego de encender apaguen en el orden inverso: primero el D3, luego D2 y finalmente D1.
3. Haga nuevamente el primer punto, pero que la secuencia se ejecute al presionar el SW1.
4. Que el LED D1 se encienda si el SW1 está presionado y el D2 al revés. El D3 alterne cada vez que se presiona el SW1.

1.5. Links y materiales de la unidad

- **Datasheet STM32F103C8/B**
  https://www.st.com/resource/en/datasheet/stm32f103c8.pdf
- **Manual de referencia STM32F1xx**
  https://tinyurl.com/yxv9m5b2
- **Documentación de libOpenCM3**
  http://libopencm3.org/docs/latest/stm32f1/html/modules.html
- **Especificación AMBA**
  https://es.wikipedia.org/wiki/Especificaci%C3%B3n_AMBA
- **PLL - Bucle de enganche de fase**
  https://es.wikipedia.org/wiki/Lazo_de_seguimiento_de_fase
- **Salida Push-Pull**
  https://es.qaz.wiki/wiki/Push%E2%80%93pull_output
- **Open-Drain (Open-Collector)**
  https://es.qaz.wiki/wiki/Open_collector
- **Super Rad!**
  https://commons.wikimedia.org/wiki/User:Super_Rad!

Unidad 2

Análisis del funcionamiento de los GPIOs

En la unidad anterior se hizo un paneo general de los GPIOs del *STM32F103* de la BluePill. Ahora analizaremos cómo es su funcionamiento a nivel de registros y las diferentes características al momento de programarlos.

2.1. Los registros del SFR de los GPIOs

Como se hizo mención anteriormente, todos los módulos internos y periféricos se configuran a través de registros que tienen direcciones de memoria específicas determinadas, en parte, por el diseñador del **CPU** (*ARM*), y por el fabricante (*ST*). ARM exige que los **SFR** se ubiquen en un rango específico de direcciones de memoria y luego los fabricantes hacen uso de ese rango a disposición.

Cada grupo de registros hace referencia a un único módulo de hardware, donde cada bit –o pequeño grupo de bits– tendrá su influencia en el comportamiento del periférico en cuestión. Para los **GPIOs** (y **AFIOs**). Cuando nosotros invocamos una de las funciones de la API de libOpcenCM3, en realidad se está escribiendo algún valor en estas posiciones de memoria.

2.1.1. Registros de Control Alto y Bajo (CRH y CRL)

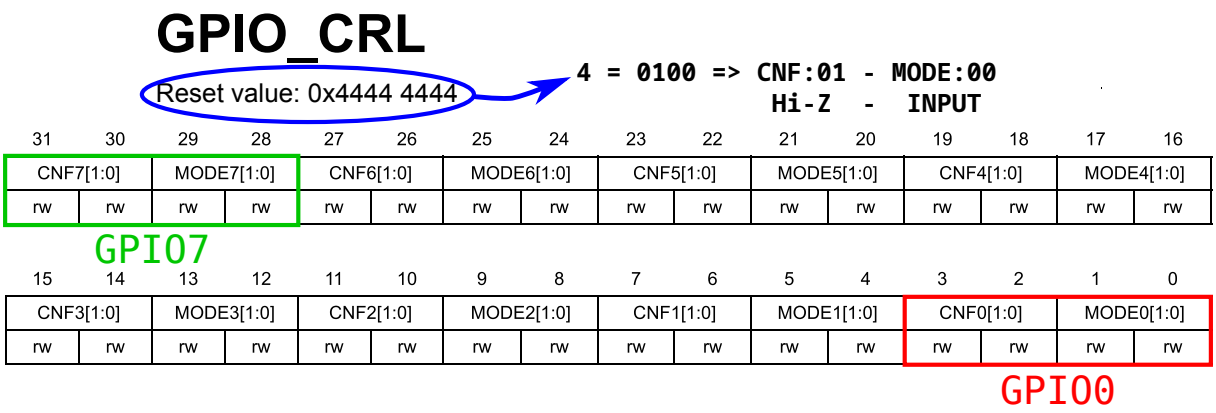
Cada vez que invocamos a `gpio_set_mode()` en realidad estamos escribiendo en alguno de los **GPIO_CRH** (*GPIO Control Register High*) o **GPIO_CRL** (*GPIO Control Register Low*) donde cada pin independiente se configura con 4 bits, 2 para `MODE` y 2 para `CNF`:

MODE		**Descripción**	
bit1	**bit0**	**Modo**	**F$_{MAX}$**
0	0	Entrada	—
0	1	Salida	10MHz
1	0		2MHz
1	1		50MHz

MODE	**CNF**		**Descripción**	
	bit1	**bit0**		
Entrada	0	0	Analógica	
		1	Flotante (Hi-Z)	
	1	0	Pull-up/down	
		1	*Reservado*	
Salida	0	0	**GPIO**	Push-Pull
		1		Open-Drain
	1	0	**AFIO**	Push-Pull
		1		Open-Drain

Tabla 1: Configuraciones de E/S.

A modo ilustrativo, veamos cómo se ve la descripción del registro del **SFR** del **GPIO_CRL** (el *CRH* es igual, pero con los GPIOs del 8-15). Allí observaremos información importante como el valor ante un *reset* que en este caso es el hexadecimal `0x4444 4444`, y que todos los bits son `rw`, es decir, de *lectura/escritura*.



En libOpenCM3 podemos acceder a esta programación de *bajo nivel* a través de macros, que son parecidas a funciones, que toman las posiciones de memoria de estos registros y podemos así escribir directamente el valor deseado:

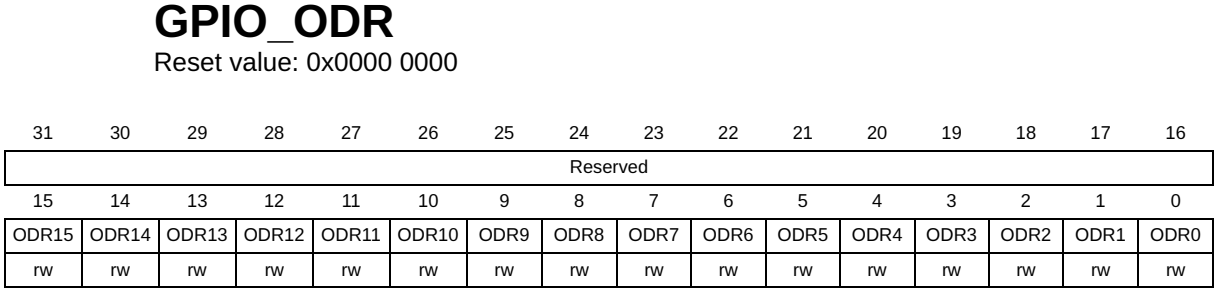
```
                // GPIO: 07  06  05  04  03  02  01  00
                //       CCMMCCMMCCMMCCMMCCMMCCMMCCMMCCMM
    GPIO_CRL(GPIOA) = 0b01000100010001000100001001000100;
                //                             ^^^^
                //            PA2: Salida 2MHz Push-Pull
```

2.1.2. Registro de Salida de Datos (ODR)

Cuando configuramos un **GPIO** como salida (`MODE≠00`) podemos establecer el valor lógico (`ALTO/BAJO`) que queremos en él a través del **ODR** (***O**utput **D**ata **R**egister*). Este tipo de programación es accesible a través de la macro `GPIO_ODR`, y solo se utilizan los bits del 0 al 15 representando los pines correspondientes. Esto es lo que hicimos en la Unidad 1 con `gpio_set` y `gpio_clear`.

```
    GPIO_ODR(GPIOA) = 0b0000000000001000; // GPIO3 en 1 los demás en 0
```

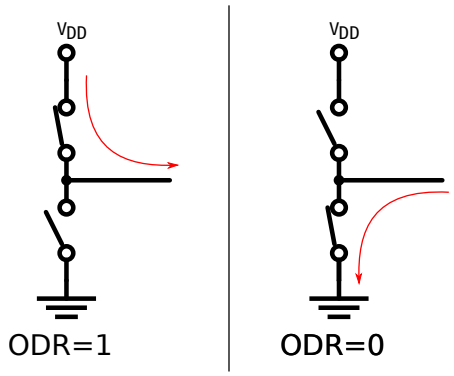
Los bits de este registro son `rw` lo cual indica que sobre el **ODR** se permiten acciones de lectura y escritura. Por lo tanto, se podrá consultar el estado del mismo de ser requerido, por ejemplo para alternar su estado (`gpio_toggle`). Cabe mencionar que a pesar de tener 32 bits, como los **GPIOs** tan solo tiene hasta 16 pines externos cada uno, los bytes superiores simplemente no se utilizan, lo cual es indicado con la palabra `Reserved` (*Reservado*).



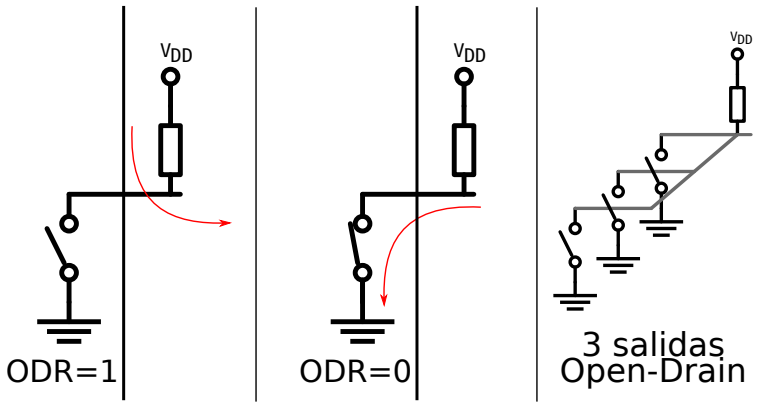
Push-Pull vs Open-Drain

Como observamos en la Tabla 1, cuando el **GPIO** está en *modo salida* este puede ser configurado (`CNF`) como **Push-Pull** u **Open-Drain**. Esto implica diferentes comportamientos.

Cuando el **GPIO** es configurado como salida *Push-Pull* es capaz de entregar corriente hacia el dispositivo el cual es conectado. Ya que la salida es manejada por dos transistores (representados como llaves) los cuales están conectados uno a la alimentación (V_{DD}) y el otro a tierra (GND), los cuales funcionan alternadamente. Esto nos indica que cuando uno está activo el otro no.



En cambio cuando se trata de una configuración *Open-Drain*, tan solo hay un transistor que se conecta a GND lo cual implica que debemos colocar una resistencia de *Pull-up* externa (no necesariamente a V_{DD}). En esta configuración el integrado ya no es el que suministra la corriente sino la tensión aplicada en conjunto con el *Pull-up*. Además de poder hacer una *AND por conexión*, es decir que podemos conectar muchas salidas Open-Drain juntas, con que una esté en 0 el estado de toda la linea será 0.



2.1.3. Registro de Entrada de Datos (IDR)

El **IDR** (***I**nput **D**ata **R**egister*) es la posición de memoria donde se podrá consultar por el estado de los **GPIOs**. Este es utilizado por la función `gpio_get()`. Solo se permiten operaciones de *lectura* (`r`) sobre este registro y el estado en que se encuentre el bit (0 ó 1) será el estado actual del pin en ese instante. Al igual que en el **ODR**, solo se utilizan los bits del 0 al 15.

GPIO_IDR

Reset value: 0x0000 XXXX

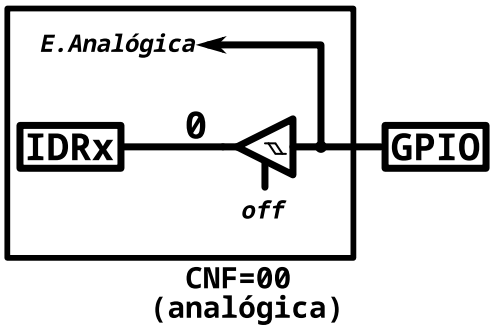
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Con libOpenCM3 podemos acceder a la lectura de este registro a través de la macro `GPIO_IDR()`:

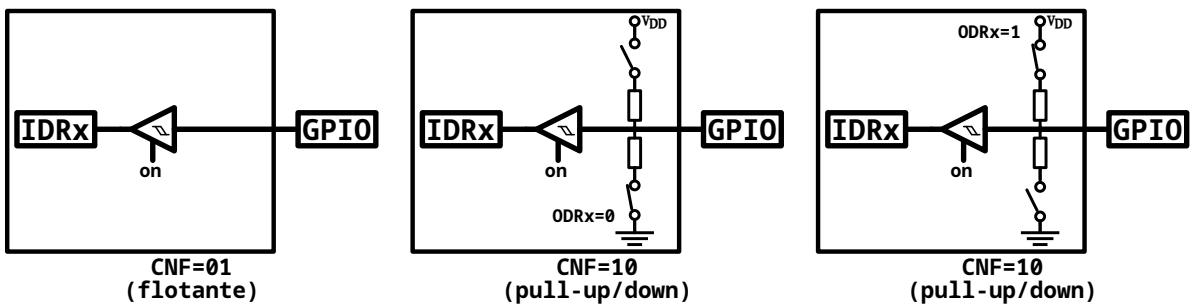
```
uint16_t estado_gpioa = GPIO_IDR(GPIOA); // Se lee el estado del GPIOA completo.
```

Analógica vs Flotante vs Pull-up/down

Como vimos en Tabla 1, cuando el modo es entrada (`MODE=00`) existen 3 tipos de configuraciones posibles: analógica (`CNF=00`), flotante (`CNF=01`) y con pull-up/down (`CNF=10`). La diferencia principal entre la analógica y las otras es desde qué punto es tomada la señal de entrada. Además, siempre leeremos un 0 en el bit correspondiente del **IDR** cuando esté configurado de esta manera.



La entrada flotante es la que se conecta directamente a través de un buffer y es conveniente utilizarla de esta forma siempre que se pueda, considerando que es necesario controlar el nivel lógico externamente. En el último caso las resistencias internas son controladas por el registro **ODR**: cuando está en 1 indicará un *pull-up* y 0 *pull-down*. Es por ello que anteriormente fue configurado utilizando `gpio_set()` y `gpio_clear()`.



Hay otros registros que han sido omitidos por simplificación, los cuales son **BSRR** (***Bit Set/Reset Register***), **BRR** (***Bit Reset Register***) y **LCKR** (***Lock Register***) y puede analizar su comportamiento en el Manual de Referencia de ST (2.3).

2.2. Funciones Alternativas

Como mencionamos en la unidad anterior, los pines tienen funciones alternativas. Un caso particular son el `PB4` y `PB3`, ya que estos por *default* no están mapeados como **GPIOs**, sino que al energizar el STM32F103 estos están destinados al protocolo de programación y debuggin **JTAG**. Estos pines no podrán ser utilizados como **GPIOs** "normales" al menos que activemos el periférico de los **AFIO** y explicitemos que deseamos utilizar unicamente el protocolo **SWD** (del ST-Link) con la siguiente porción de código:

```
rcc_periph_clock_enable(RCC_AFIO);
gpio_primary_remap(AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_ON, 0); // JTAG off - SW on
```

Esta función recibe dos parámetros, uno es si queremos **JTAG**, **SWD**, ambos o ninguno como interfaz de programación y el segundo para reubicar los pines de cada periférico en caso de ser necesario. Esto no será tratado durante el curso, pero es bueno tenerlo presente al momento de un proyecto. La hoja de datos indica qué pines pueden ser remapeados por periférico.

Atención

Si pone ambos protocolos en *off*, esto hará que el ST-Link no pueda programar más el microcontrolador por medio de SWD. Deberá utilizar el método por Serie (`BOOT0 = 1`, `BOOT1 = 0`) para poder volver a grabarlo y sacarlo de ese estado.

2.3. Links y materiales de la unidad

- **Datasheet STM32F103C8/B**
  https://www.st.com/resource/en/datasheet/stm32f103c8.pdf
- **Manual de referencia STM32F1xx**
  https://tinyurl.com/yxv9m5b2
- **Documentación de libOpenCM3**
  http://libopencm3.org/docs/latest/stm32f1/html/modules.html
- **Salida Push-Pull**
  https://es.qaz.wiki/wiki/Push%E2%80%93pull_output
- **Open-Drain (Open-Collector)**
  https://es.qaz.wiki/wiki/Open_collector
- **JTAG en Wikipedia**
  https://es.wikipedia.org/wiki/JTAG

Esta página fue dejada en blanco deliberadamente.

Unidad 3
Interrupciones

Toda computadora posee un mecanismo de hardware conocido como **interrupción**, el cual puede cambiar el flujo del programa en un momento no determinístico. Es una solución práctica a eventos externos e internos que deben ser atendidos pero no sabemos cuando ocurrirán, pero debemos estar preparados para ellos.

3.1. ¿Cómo funciona una interrupción?

Las interrupciones **son eventos** que pueden sucintarse en cualquier momento mientras ejecutamos nuestro programa. Cuando esto ocurre, la señal llega al CPU e *interrumpe* el flujo, lo que quiere decir que al terminar la instrucción en curso, se salva el valor actual del **IP/PC** y se carga en él un nuevo valor prefijado por el fabricante. A esa posición de memoria se la conoce como **vector de interrupción**.

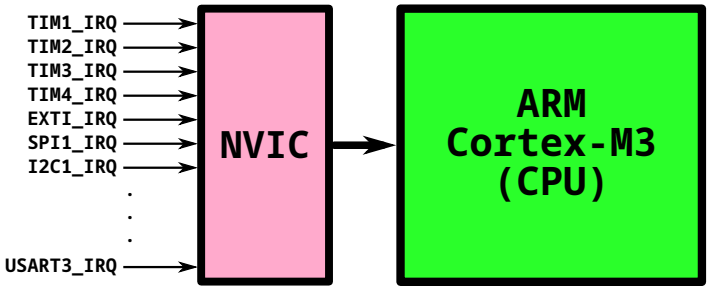
Es trabajo del programador completar las instrucciones en dicho vector, es decir *crear la rutina que atienda la interrupción*. Ese pequeño algoritmo que debe estar preparado es generalmente denominado como *Sub-Rutina de Interrupción* (**ISR** - ***I**nterrupt **S**ub-**R**outine*).

Es habitual que haya más de una *fuente de interrupción*, lo que nos indica que existen muchos y variados eventos que podrían ocasionar una interrupción. No todas tendrán su propio vector, algunas compartirán la misma dirección de memoria para atender diferentes causas. Es por eso que existen bits de estado conocidos como **banderas** (*flags*) que podremos consultar y nos indicarán cual fue el causante de la interrupción en curso.

Si bien este mecanismo de hardware brinda una herramienta poderosa y relativamente sencilla, debemos tener ciertos recaudos al utilizarla. Por ejemplo, existe la posibilidad de que 2 interrupciones lleguen prácticamente al mismo tiempo y debemos estar preparados para ello. También podría ocurrir que una interrupción llegue en medio de un calculo o proceso crítico dentro de nuestro sistema entre muchas otras. Además, es aconsejable que las *ISR* sean lo más cortas posible para retornar al flujo normal del programa rápidamente. Todo esto será detallado a medida que se desarrolle la unidad.

3.2. Control de Vectores de Interrupción Anidados

Todos los ARM Cortex-M poseen un módulo de hardware conocido como **NVIC** (***N**ested **V**ector **I**nterrupt **C**ontrol*) que es el mecanismo encargado de atender los *pedidos de interrupción* (**IRQ** - ***I**nterrupt **Req**uest*) al CPU y definir sus **prioridades**.



Todas las peticiones de interrupción están inhabilitadas del **NVIC** por *defualt* excepto las que pertenecen al CPU mismo, p.e. el *Reset* que es la interrupción con mayor prioridad y es ineludible. Cuando el **µC** se *resetea* el flujo del programa vuelve al inicio inexorablemente. Es por ello que debemos ir indicando los eventos que deseamos atender.

Para las demás interrupciones que corren por cuenta del fabricante (en este caso ST) debemos explicitar cuando habilitarlas e inhabilitarlas utilizando las funciones:

```
void nvic_enable_irq(uint8_t irqn);
void nvic_disable_irq(uint8_t irqn);
```

3.2.1. Vectores de interrupción

Prácticamente cada periférico presente en el STM32F103 posee una o más **IRQs**, es por ello que optaremos por analizarlas en la medida que se describan cada uno de ellos. Pero a modo general, los vectores están asociados a funciones en C que tienen nombres específicos y únicos que deben ser respetados. En libOpenCM3, las rutinas que manejan las interrupciones propias del ARM Cortex-M terminan con la palabra `handler` y son las listadas a continuación (más adelante en esta misma unidad utilizaremos `sys_tick_handler`).

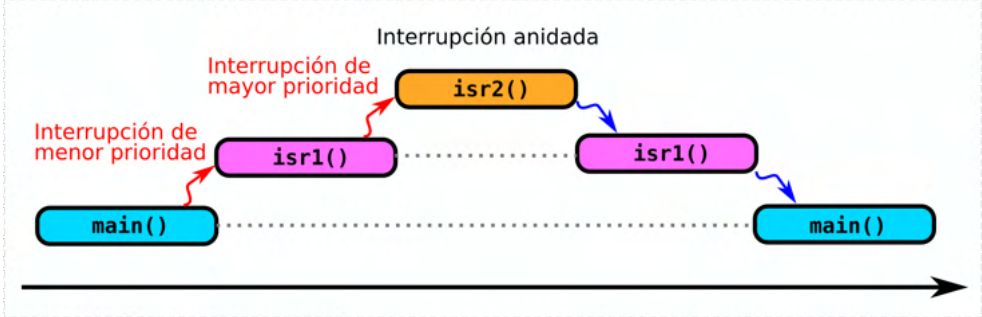
```
void reset_handler(void);
void nmi_handler(void);
void hard_fault_handler(void);
void sv_call_handler(void);
void pend_sv_handler(void);
void sys_tick_handler(void);
void mem_manage_handler(void);
void bus_fault_handler(void);
void usage_fault_handler(void);
void debug_monitor_handler(void);
```

En cambio, las **IRQs** provenientes de módulos de hardware implementados por el fabricante terminan con la palabra `isr`, p.e. `exti0_isr()` que analizaremos sobre el final de la unidad.

Estas funciones solamente están declaradas, y es el programador el que debe implementarlas. Y estás serán ejecutadas cuando la **IRQ** ocurra. Si no se implementan por el programador estás están definidas como un bucle infinito que no deja continuar la ejecución del programa. En otras palabras, si no se implementa la *ISR* el STM32 simplemente *"se colgará"*.

3.2.2. Prioridades

Así como se mencionó durante el apartado anterior, podría ocurrir que 2 interrupciones ocurran *prácticamente* al mismo tiempo o que una segunda llegue mientras se está en la *ISR* de la primera. Para ello existen dos políticas, una sería inhabilitar las interrupciones como primera acción dentro de la *ISR* que primero se atienda. La segunda opción, más robusta y flexible es *establecer prioridades*.



En esta familia de microcontroladores hay 16 niveles de prioridades siendo el 0 la mayor prioridad y el 15 la menor. Y con libOpenCM3 podemos configurar utilizando `nvic_set_priority()`, donde `irqn` es la **IRQ** correspondiente y `priority` un número entre 0 y 15.

```
void nvic_set_priority(uint8_t irqn, uint8_t priority);
```

3.3. SysTick Timer

En todos los ARM Cortex-M existe un temporizador de 24-bits conocido como **SysTick**, el cual está pensado para llevar la referencia de tiempo cuando se utiliza un *Sistema Operativo de Tiempo Real* (**RTOS**). Como nosotros no utilizaremos esa tecnología, podemos utilizarlo como un simple temporizador más.

Analizaremos en una próxima unidad cómo funcionan los *Timers* en general. Aquí simplemente expondremos que es un módulo que cuenta hasta un valor determinado entre 1 y $2^{24} - 1$. Al terminar, vuelve a empezar generando una **IRQ** periódica a una frecuencia X de tiempo la cual depende del clock del AHB. Para poder programar este módulo podemos utilizar la función `systick_set_frequency` que recibe una frecuencia en Hz (`freq`) y el valor de clock del `ahb` el cual podemos obtener de la variable global `rcc_ahb_frequency`.

```
bool systick_set_frequency(uint32_t freq, uint32_t ahb);
```

Una vez configurada la cuenta debe inicializar la cuenta y habilitar la **IRQ** para que funcione correctamente con las siguientes funciones:

```
void systick_counter_enable(void);
void systick_interrupt_enable(void);
```

Como esta interrupción es interna del CPU no necesitamos invocar al **NVIC** para habilitarla, pero sí debemos agregar `nvic.h`, debido a que en este archivo están declaradas las *ISRs* de todas las interrupciones.

delay.c

```
1  #include<libopencm3/stm32/rcc.h>
2  #include<libopencm3/stm32/gpio.h>
3  #include<libopencm3/cm3/systick.h>
4  #include<libopencm3/cm3/nvic.h>
5
6  volatile uint32_t millis; // variable que se incrementa cada 1ms
7
8  /**
9   * Delay bloqueante utilizando el SysTick
10  */
11 void delay_ms(uint32_t ms);
12
13 int main(void)
14 {
15     // SYSCLK a 72Mhz
16     rcc_clock_setup_in_hse_8mhz_out_72mhz();
17
18     // PC13 (LED de la BluePill) como salida:
19     rcc_periph_clock_enable(RCC_GPIOC);
20     gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
21
22     /*** Configuración del SysTick ***/
23     // frecuencia 1KHz => T = 1ms
24     systick_set_frequency(1000, rcc_ahb_frequency);
25     // Se habilita la interrupción del SysTick:
26     systick_interrupt_enable();
27     // El SysTick empieza a contar:
28     systick_counter_enable();
29
30     while(true)
31     {
32         // Se alterna el LED en PC13
33         gpio_toggle(GPIOC,GPIO13);
34         // Esperamos 500ms
35         delay_ms(500);
36     }
37 }
38
39 void delay_ms(uint32_t ms)
40 {
41     uint32_t tm = millis + ms;
42     while(millis < tm);
43 }
44
45 /**
46  * Sub-Rutina de interrupción del SysTick (ejecutada cada 1ms)
47  */
48 void sys_tick_handler(void)
49 {
50     millis++; // Se incrementan los millis
51 }
```

Notese que en las lineas 3 y 4 de `delay.c` los `#include` agregan `cm3` y no `stm32` lo cual reafirma que es algo independiente al fabricante y es propio del CPU. En cuanto a la función `delay_ms()`, ésta recibe un valor (`ms`) que es sumado al valor actual en `millis` la cual es una variable global que se incrementa en la *ISR* del **SysTick** cada $1ms$. El resultado de esa suma es almacenado en una variable (`tm`) para ser comparada en un bucle `while` de forma bloqueante. Es importante entender que si no fuera porque `millis` es incrementada en una rutina de interrupción el bucle sería infinito y el programa quedaría *"colgado"*.

3.4. Interrupciones externas (EXTIs)

Las interrupciones externas **EXTIs** son un mecanismo de hardware que detectan los cambios de estado en pines habilitados y esto origina una **IRQ**. En el *STM32F103* podemos utilizar cualquier pin de cualquier **GPIO** para esta tarea, generando una petición de interrupción por flancos ascendentes, descendentes o ambos.

Si bien cualquier pin puede ser utilizado como **EXTI** debemos tener en cuenta que son 16 (de *EXTI0* a *EXTI15*) coincidiendo con el número de pin del **GPIO** lo cual nos indica que si configuramos, por ejemplo, la *EXTI13* este podrá estar asociado al PA13, PB13 o PC13, pero no a los tres al mismo tiempo, sino que a uno solo de ellos. Para determinar este vinculo entre **EXTI** y **GPIO**, elegir qué evento disparará la **IRQ** y habilitar la interrupción utilizaremos las siguientes funciones de libOpenCM3 incluidas en `exti.h`.

```
// Include necesario para utilizar las funciones:
#include <libopencm3/stm32/exti.h>

// Vincula la EXTIx con el GPIOx:
void exti_select_source(uint32_t exti, uint32_t gpioport);
// p.e.: exti_select_source(EXTI3, GPIOB);

// Configura la EXTIx por el flanco correspondiente:
void exti_set_trigger(uint32_t extis, enum exti_trigger_type trig);
// Las opciones de 'exti_trigger_type' son:
EXTI_TRIGGER_RISING     // flanco ascendente genera la IRQ
EXTI_TRIGGER_FALLING    // flanco descendente genera la IRQ
EXTI_TRIGGER_BOTH       // ambos flancos generan la IRQ

// Habilita la interrupción EXTIx:
void exti_enable_request(uint32_t extis);

// Inhabilita la interrupción EXTIx:
void exti_disable_request(uint32_t extis);
```

Los vectores de interrupción son independientes del *EXTI0* al *EXIT4*, pero del *EXTI5* al *EXTI9* comparten el vector al igual que los *EXTI10* al *EXTI15*. Esto quiere decir que si utilizáramos el *EXTI11* y *EXTI15* deberíamos preguntar durante el *ISR* cuál de ellos provocó el **IRQ**. Independientemente de si se comparte o no vector de interrupción debemos siempre *"bajar la bandera"* indicando que ya hemos atendido dicha interrupción.

```
// Indica que se atendió la IRQ generada por la EXTIx
void exti_reset_request(uint32_t extis);

// Valor de la bandera que indica si la EXTIx fue la que generó el IRQ
uint32_t exti_get_flag_status(uint32_t exti);

// Debe incluirse nvic.h donde están declaradas las
// Sub-Rutinas de Interrupción de los EXTIs:
void exti0_isr(void);       // ISR de la EXTI0
void exti1_isr(void);       // ISR de la EXTI1
void exti2_isr(void);       // ISR de la EXTI2
void exti3_isr(void);       // ISR de la EXTI3
void exti4_isr(void);       // ISR de la EXTI4
void exti9_5_isr(void);     // ISR de la EXTI5 a EXTI9
void exti15_10_isr(void);   // ISR de la EXTI10 a EXTI15
```

Atención

Es importante que además de habilitar y configurar correctamente el pin del **GPIO** correspondiente como ENTRADA, debemos también habilitar los **AFIO**, ya que justamente es una *función alternativa* utilizando:

```
rcc_periph_clock_enable(RCC_AFIO);
```

Observemos el siguiente ejemplo donde configuraremos y utilizaremos el *EXTI0* conectado al *PB0* por flanco descendente para alternar el *PC13*.

exti0.c

```
1  #include <libopencm3/stm32/rcc.h>
2  #include <libopencm3/stm32/gpio.h>
3  #include <libopencm3/stm32/exti.h>
4  #include <libopencm3/cm3/nvic.h>
5
6  int main(void)
7  {
8      // Configuración del SysClock a 72MHz
9      rcc_clock_setup_in_hse_8mhz_out_72mhz();
10
11     // Se habilita el PC13 (LED de la BluePill como salida Push-Pull)
12     rcc_periph_clock_enable(RCC_GPIOC);
13     gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
14
15     // Se configura el PB0 como entrada
16     rcc_periph_clock_enable(RCC_GPIOB);
17     gpio_set_mode(GPIOB, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO0);
18
19     /** Configuración de la EXTI0 **/
20     rcc_periph_clock_enable(RCC_AFIO);              // Habilitar el AFIO
21     exti_select_source(EXTI0, GPIOB);               // EXTI0 vinculado al GPIOB (PB0)
22     exti_set_trigger(EXTI0, EXTI_TRIGGER_FALLING); // Flanco descendente
23     exti_enable_request(EXTI0);                     // Se habilita el EXTI0
24
25     // Se habilita la IRQ para el EXTI0 en el NVIC
26     nvic_enable_irq(NVIC_EXTI0_IRQ);
27
28     while (true)
29         __asm__("nop"); // Esperar la interrupción...
30 }
31
32 /**
33  * Sub-Rutina de Interrupción de EXTI0
34  */
35 void exti0_isr(void)
36 {
37     gpio_toggle(GPIOC, GPIO13); // Se alterna el PC13
38     exti_reset_request(EXTI0);  // Se marca la EXTI0 como atendida. (baja el flag)
39 }
```

Es importante habilitar la **IRQ** en el **NVIC** como se observa en la *linea 26*. Y en particular con las **EXTIs** no debemos olvidarnos de *resetear* el pedido de interrupción como se observa en la *linea 38*.

En el caso de utilizar las **EXTIs** que comparten el vector debemos consultar los flags como ya se anticipo. Esto es visible en las *lineas 44 y 50* en el siguiente ejemplo donde la *EXTI12* y *EXTI13* son utilizadas para prender y apagar el LED en *PC13* respectivamente. Es importante recordar siempre bajar el *flag* como se observa en las *lineas 47 y 53*.

exti12-13.c

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/exti.h>
#include <libopencm3/cm3/nvic.h>

int main(void)
{
    // Configuración del SysClk a 72Mhz
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    // Configuración del PC13 como salida (LED de la Bluepill activo en BAJO)
    rcc_periph_clock_enable(RCC_GPIOC);
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);

    // Configuración como entrada de los pines PB12 y PB13
    rcc_periph_clock_enable(RCC_GPIOB);
    gpio_set_mode(GPIOB, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO12 | GPIO13);

    /* Configuración de las EXTIs 12 y 13 */
    // Se habilitan las AFIO
    rcc_periph_clock_enable(RCC_AFIO);
    // EXTI12 en PB12 por flanco descendente
    exti_select_source(EXTI12, GPIOB);
    exti_set_trigger(EXTI12, EXTI_TRIGGER_FALLING);
    exti_enable_request(EXTI12);
    // EXTI13 en PB13 por flanco descendente
    exti_select_source(EXTI13, GPIOB);
    exti_set_trigger(EXTI13, EXTI_TRIGGER_FALLING);
    exti_enable_request(EXTI13);

    // Se habilita la IRQ de las EXTI10 a EXTI15 (mismo vector)
    nvic_enable_irq(NVIC_EXTI15_10_IRQ);

    while (true)
        __asm__("nop"); // esperando las interrupciones
}

/**
 * Sub-Rutina de Interrupción de las EXTI12 y EXTI13
 */
void exti15_10_isr(void)
{
    // Si la IRQ fue generada por EXTI12
    if (exti_get_flag_status(EXTI12))
    {
        gpio_clear(GPIOC, GPIO13);  // Se enciende el LED en PC13
        exti_reset_request(EXTI12); // Se baja el flag de la EXTI12
    }
    // Si la IRQ fue generada por EXTI13
    if (exti_get_flag_status(EXTI13))
    {
        gpio_set(GPIOC, GPIO13);    // Se apaga el LED en PC13
        exti_reset_request(EXTI13); // Se baja el flag de la EXTI13
    }
}
```

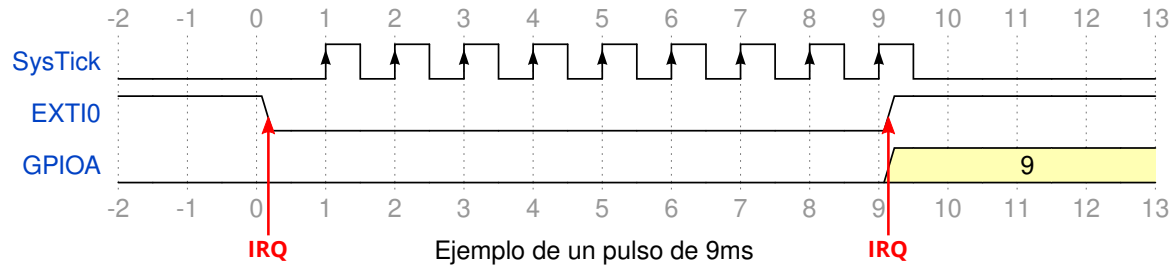
En el siguiente ejemplo utilizaremos una **EXTI** y el **SysTick** para medir la longitud de un pulso *activo en BAJO* y luego el valor en milisegundos será enviado al **GPIOA** para su visualización. Note que **no se utiliza la interrupción** del **SysTick**, simplemente tomamos su valor y lo enviamos al puerto borrando su contenido (`gpio_clear()`) y luego escribiéndolo y a través de las funciones `gpio_set()` y `systick_get_value()`. La interrupción del *EXTI0* borra e inicia el **SysTick** al detectar un flanco descendente y cambia el flanco que dispara la **IRQ**. Al detectar el flanco ascendente la **IRQ** detiene el **SysTick** y se indica que hay una nueva medición disponible volviendo a configurar la *EXTI0* por flanco descendente.

pulse-width.c

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/exti.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/cm3/systick.h>

volatile bool flanco_descendente = true; // Configuración actual de la EXTI0
volatile bool hay_medicion = false;      // Indica que hay una medición nueva

int main(void)
{
    // Configuración del SysClk a 72Mhz
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    // Se configura el PB0 como entrada
    rcc_periph_clock_enable(RCC_GPIOB);
    gpio_set_mode(GPIOB, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO0);

    // Se configura el GPIOA como salida (todo los pines)
    rcc_periph_clock_enable(RCC_GPIOA);
    gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO_ALL);

    // Se configura el SysTick para contar cada 1ms
    systick_set_frequency(1000, rcc_ahb_frequency);

    /* Configuración de la EXTI0 */
    rcc_periph_clock_enable(RCC_AFIO);              // Habilitación del AFIO
    exti_select_source(EXTI0, GPIOB);               // Vinculación del EXTI0 al PB0
    exti_set_trigger(EXTI0, EXTI_TRIGGER_FALLING); // Inicialmente por flanco descendete
    exti_enable_request(EXTI0);                     // Se habilita el EXTI0

    // Se habilita la IRQ para la EXTI0 en el NVIC
    nvic_enable_irq(NVIC_EXTI0_IRQ);

    while (true)
    {
        if (hay_medicion) // Si hay una nueva medición:
        {
            // Se envía el valor actual en el SysTick a los pines del GPIOA
            gpio_clear(GPIOA, GPIO_ALL);
            gpio_set(GPIOA, systick_get_value());
            hay_medicion = false; // Ya se procesó la medición...
        }
    }
}

/**
 * Sub-Rutina de Interrupción del EXTI0
 */
void exti0_isr(void)
{
    // Si el EXTI0 está configurado por flanco descendente:
    if (flanco_descendente)
    {
        systick_clear();                            // Se resetea la cuenta del SysTick
        systick_counter_enable();                   // Se habilita la cuenta del SysTick
        exti_set_trigger(EXTI0, EXTI_TRIGGER_RISING); // Reconfigurar flanco
        flanco_descendente = false;
    }
    else // Si el EXTI0 está configurado por flanco ascendente:
    {
        systick_counter_disable();                   // Se inhabilita el SysTick
        exti_set_trigger(EXTI0, EXTI_TRIGGER_FALLING); // Reconfigurar flanco
        hay_medicion = true;                         // Hay una nueva medición
        flanco_descendente = true;
    }
    exti_reset_request(EXTI0); // La IRQ fue atendida (bajada del flag)
}
```

En la siguiente grafica se expone el funcionamiento del ejemplo para una me-

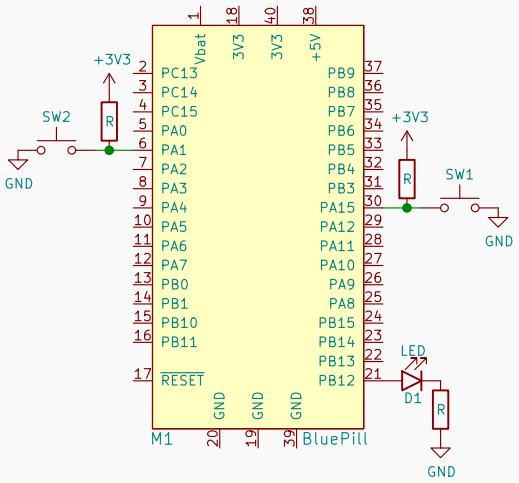
jor comprensión.



3.5. Ejercicios

EJERCICIO N°2

Programe utilizando las interrupciones externas un sistema que al detectar el flanco en *EXTI1* (PA1) prenda el LED D1 (PB12) y al detectar el flanco en *EXTI15* (PA15) lo apague. Verifique el funcionamiento con el siguiente circuito.



EJERCICIO N°3

Agregue un LED D2 al circuito del ejercicio anterior en cualquier pin disponible. Utilizando el **SysTick** haga que el LED D2 titile con un periodo de 1 segundo sin alterar el funcionamiento del sistema anterior.

EJERCICIO N°4

Utilizando el circuito del **EJERCICIO N°2** con el LED D2 agregado del **EJERCICIO N°3** y diseñe un sistema que calcule la duración del pulso en *PA1* y el LED D1 titile con un periodo igual a la duración del mismo. Haga lo mismo para *PA15* y el LED D2.

3.6. Links y materiales de la unidad

- **Documentación de libOpenCM3**
  http://libopencm3.org/docs/latest/stm32f1/html/modules.html
- **Datasheet STM32F103C8/B**
  https://www.st.com/resource/en/datasheet/stm32f103c8.pdf
- **Manual de referencia STM32F1xx**
  https://tinyurl.com/yxv9m5b2
- **Interrupción**
  https://es.wikipedia.org/wiki/Interrupci %C3 %B3n

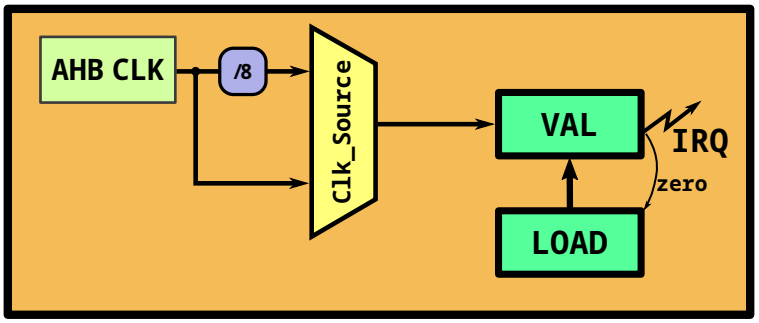
Unidad 4

Temporizadores y Contadores

En la unidad anterior se hizo uso del **SysTick** que es un temporizador, pero no se ha hecho foco en su funcionamiento. Como ya se mencionó, es un simple módulo que ya viene integrado en todo los **CPU** *ARM Cortex-M*, por lo tanto es independiente de los distintos fabricantes. A su vez, cada uno de estos agregan más de estos periféricos internos. En el caso del *STM32F103Cx* tiene 4 de ellos denominados **TIM**, siendo numerados como TIM1, TIM2, TIM3 y TIM4, sin contar los *WatchDogs* que serán cubiertos más adelante, ya que su propósito es especial.

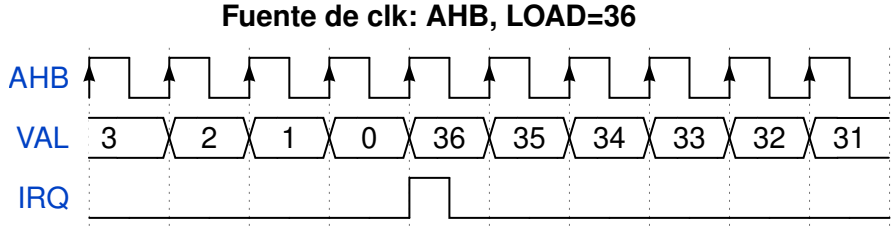
4.1. Breve análisis sobre el SysTick

Como se mencionó en el apartado 3.3, el *SysTick* es un temporizador descendente, es decir que del valor inicial decrementa con cada pulso hasta llegar al *underflow* un pulso después del 0. Este posee 24 bits, es decir que puede llevar cuentas desde `0x00FFFFFF` a `0` (*hasta 16.777.216 cuentas*).



Cuenta con 4 registros, uno de **Control** donde se configura la entrada de clock (`AHB` o `AHB/8`) y se habilita la cuenta (arranca el temporizador) e interrupción. El registro `LOAD` con el valor al cual debe ser recargado cada vez que llega el *underflow* (cero) y el registro `VAL` donde podemos obtener el valor actual del SysTick. Además tiene un registro con el cual se puede calibrar para tener un ajuste fino para los sistemas que son críticos con respecto al tiempo.

En el siguiente diagrama temporal podemos observar cómo funcionaría con `LOAD=36` y la fuente de clock del `AHB` sin dividir:



Con libOpenCM3 disponemos de las siguientes funciones para tener acceso al SysTick incluidas en `libopencm3/cm3/systick.h`:

```
void systick_set_reload(uint32_t value);
bool systick_set_frequency(uint32_t freq, uint32_t ahb);
uint32_t systick_get_reload(void);
uint32_t systick_get_value(void);
void systick_set_clocksource(uint8_t clocksource);
void systick_interrupt_enable(void);
void systick_interrupt_disable(void);
void systick_counter_enable(void);
void systick_counter_disable(void);
uint8_t systick_get_countflag(void);
void systick_clear(void);
uint32_t systick_get_calib(void);
```

Una configuración básica sería la siguiente:

```
// Se toma la velocidad del AHB (SysClk) y se activa el divisor por 8:
systick_set_clocksource(STK_CSR_CLKSOURCE_AHB_DIV8); // 72M/8 = 9M

// Se establece el valor a cargar cuando el SysTick pasa el 0 (underflow):
systick_set_reload(8999); // 9M / 9000 = 1k => T = 1ms

// Se habilita la interrupción del SysTick:
systick_interrupt_enable();

// El SysTick empieza a contar:
systick_counter_enable();

// (...)

void sys_tick_handler(void)
{
    // interrupción periódica del SysTick
}
```

La manera más sencilla de configuración es como lo utilizamos anteriormente:

```
// 'rcc_ahb_frequency' es una variable global accesible desde rcc.h
systick_set_frequency(xxxx, rcc_ahb_frequency); // xxxx: valor en Hz.
systick_interrupt_enable();                     // Se habilita la interrupción.
systick_counter_enable();                       // Se habilita el SysTick.

// (...)

void sys_tick_handler(void)
{
    // interrupción periódica del SysTick
}
```

La función `systick_set_frequency()` calcula automáticamente si necesita dividir o no el `AHB/8` para llegar a la frecuencia requerida. En caso de no poder alcanzar la frecuencia deseada retorna un `false`.

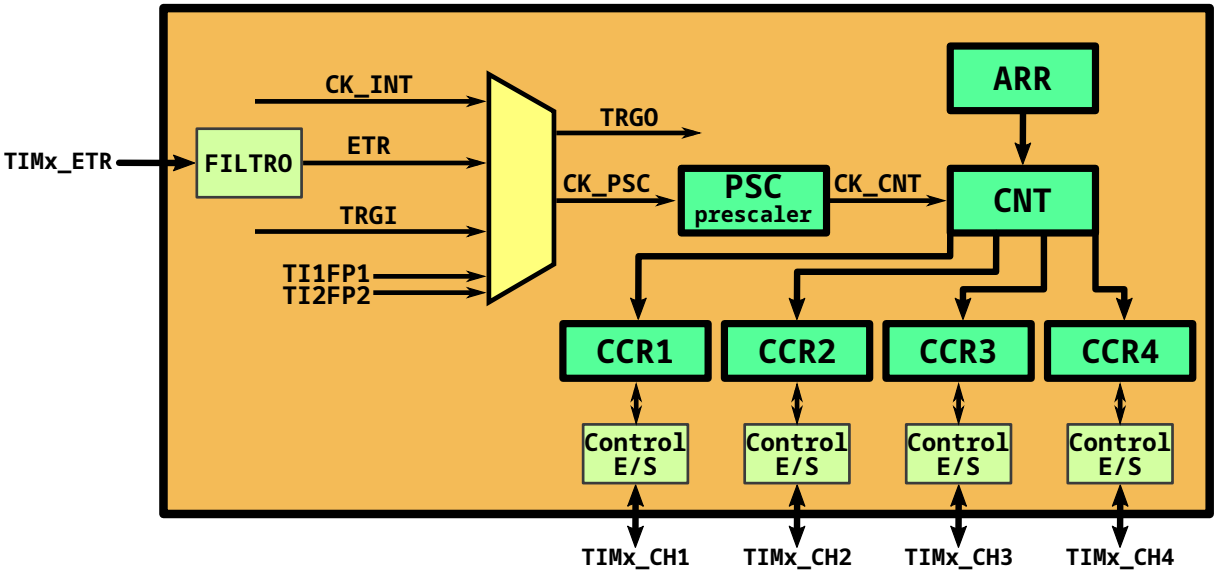
Como ya estuvimos haciendo uso de este módulo no se ahondará mucho más en el para poder abarcar a los **TIM**.

4.2. Temporizadores del fabricante (TIMx)

Hay 2 tipos de TIMx disponibles en el *STM32F103Cx* los denominados por el fabricante como **Temporizadores de Control Avanzado** (*Advanced-Control Timers* - **TIM1**) y los **Temporizadores de Propósito General** (*General-Purpose Timers* - **TIM2**, **TIM3** y **TIM4**). Como el primero extiende las funcionalidades de los generales, solo cubriremos estos que darán una idea de lo que se puede hacer.

A su vez, no se cubrirán todas las características que poseen ya que son demasiado extensas y se explicarán los conceptos más básicos y sólo algunas particulares que son de utilidad.

4.3. Aspectos generales de los TIM



Como se observa en el esquema anterior, los **TIMs** tienen más elementos internos que el *SysTick*. Todos ellos poseen al menos 4 canales (`TIMx_CHx`) que pueden funcionar como entrada para contar o medir diferentes pulsos o utilizarlos como salidas (*PWM*, pulsos únicos, entre otros). Las fuentes para el clock podemos separarlas también en 4: Clock interno, pulso externo desde `TIMx_ETR`, *triggers* (eventos) internos (*Modo Esclavo)* y la interfaz de lectura para *encoders* (`TI1FP1` y `TI2FP2`).

Además, posee un prescaler ajustable entre 1 y 65536 que dividirá el clock entrante. El contador (`CNT`) que es análogo al `VAL` del *SysTick* puede contar en ambas direcciones y generar interrupciones por *underflow*, *overflow* o por alcanzar el valor en el Registro de *Auto-Reload* (`ARR`) que hace a su vez de `LOAD` cuando cuenta de forma descendente.

Los comparadores (`CCRx`) pueden generar interrupciones al igualar su valor con el contador o eventos de petición al **DMA**.

Por último, se visualiza como en *Modo Maestro* puede enviar señales a otros módulos como el **ADC** u otros *timers* con la señal *Trigger Output* (`TRGO`).

4.4. Temporizador clásico

Este es el modo más común en el que se puede utilizar exactamente igual al **SysTick**. En este modo la fuente de clock será el `CK_INT`, el cual está asociado al `PCLKx` correspondiente. Si bien el **TIM2**, **TIM3** y **TIM4** están en el **APB1** ($F_{MAX} = 36Mhz$), en la configuración estándar del clock esta es multiplicada x2 obteniendo $72Mhz$

como máximo. La función `timer_set_mode()` configura los aspectos básicos de funcionamiento que encontramos en el primer registro de configuración **TIMx control register 1** (`TIMx_CR1`).

Podemos setear el divisor para los filtros de entrada (`TIMx_ETR` y de los canales `TIMx_CHx`) sin división, x2 o x4; la alineación al flanco o centrado; y la dirección del contador (ascendente o descendente). Todas las funciones serán provistas por `libopencm3/stm32/timer.h`.

```
/* Función de configuración del TIM como temporizador */
void timer_set_mode(uint32_t timer_peripheral, uint32_t clock_div, uint32_t alignment,
    uint32_t direction);

// defines para 'timer_peripheral'
TIM1, TIM2, TIM3, TIM4

// defines para 'clock_div' (para los filtros digitales de entrada)
TIM_CR1_CKD_CK_INT        // tiempo de sampleo Tdts = Tck_int
TIM_CR1_CKD_CK_INT_MUL_2  // tiempo de sampleo Tdts = 2 x Tck_int
TIM_CR1_CKD_CK_INT_MUL_4  // tiempo de sampleo Tdts = 4 x Tck_int

// defines para 'alignment'
TIM_CR1_CMS_EDGE          // Alineado al flanco
TIM_CR1_CMS_CENTER_1      // Modo centrado 1
TIM_CR1_CMS_CENTER_2      // Modo centrado 2
TIM_CR1_CMS_CENTER_3      // Modo centrado 3

// defines para 'direction'
TIM_CR1_DIR_UP            // Cuenta ascendente
TIM_CR1_DIR_DOWN          // Cuenta descendente
```

El **prescaler** es de 16-bits, pudiendo dividir la frecuencia entrante por `CK_PSC` entre 1 y 65536, teniendo en cuenta que el registro toma valores de 0 a 65535, es decir que el valor a dividir será `PSC`+1. Para modificar este utilizamos la función `timer_set_prescaler()` indicando el **TIMx** y el valor a utilizar. La salida es la señal `CK_CNT` que hace que el contador del timer incremente o decremente según su configuración. En caso de contar descendentemente se cargará nuevamente el valor en el Registro de Auto-Reload (*Underflow*) el cual se configura con la función `timer_set_period()`. En caso de contar ascendentemente el contador volverá a 0 (cero) tras el valor en dicho registro (*Overflow*). Una vez lista la configuración, es necesario habilitar la cuenta con la función `timer_enable_counter()`.

```
void timer_set_prescaler(uint32_t timer_peripheral, uint32_t value);
void timer_set_period(uint32_t timer_peripheral, uint32_t period);
void timer_enable_counter(uint32_t timer_peripheral)
```

4.4.1. Contador ascendente

Analicemos el siguiente ejemplo donde configuraremos el **TIM2** para contar de 0 al valor del `ARR` de tal manera que el timer interrumpa cada vez que alcance dicho valor.

cnt-up.c

```
1 #include <libopencm3/stm32/rcc.h>
2 #include <libopencm3/stm32/gpio.h>
3 #include <libopencm3/stm32/timer.h>
```

```
4  #include <libopencm3/cm3/nvic.h>
5
6  int main(void)
7  {
8      // Se configura el clock del sistema a 72Mhz
9      rcc_clock_setup_in_hse_8mhz_out_72mhz();
10
11     // Se configura el LED de la BluePill (PC13) como Salida Push-Pull
12     rcc_periph_clock_enable(RCC_GPIOC);
13     gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
14
15     // Se habilita el clock del periférico del TIM2
16     rcc_periph_clock_enable(RCC_TIM2);
17
18     // Se configura el modo de funcionamiento del Timer2
19     timer_set_mode(
20         TIM2,               // Timer2
21         TIM_CR1_CKD_CK_INT, // Clk de sampleo == CK_INT, (no se utiliza)
22         TIM_CR1_CMS_EDGE,   // Alineado al flanco
23         TIM_CR1_DIR_UP);    // Cuenta ascendente
24
25     timer_set_prescaler(TIM2, 35999); // 72MHz / 36000 => 2KHz
26
27     // Se setea el valor hasta donde se desea contar (ARR):
28     timer_set_period(TIM2, 999); // 2KHz / 1000 = 2Hz => T = 0.5s
29
30     // Se habilita al interrupción por UpdateInterrupt (overflow/underflow)
31     timer_enable_irq(TIM2, TIM_DIER_UIE);
32
33     // Empieza a contar
34     timer_enable_counter(TIM2);
35
36     // Se habilita la interrupción en el NVIC
37     nvic_enable_irq(NVIC_TIM2_IRQ);
38
39     while (1)
40         ; // Nada por hacer, esperando la interrupción
41 }
42
43 void tim2_isr(void)
44 {                                       // Cada 0.5s:
45     timer_clear_flag(TIM2, TIM_SR_UIF); // Se baja el flag de la interrupción
46     gpio_toggle(GPIOC, GPIO13);         // Se alterna el LED
47 }
```

Ahora analicemos la función `timer_enable_irq()`. Como es posible generar varios tipos de interrupciones y eventos, debemos especificar cual es la que queremos implementar. Esto está asociado al **DMA/Interrupt enable register** (`DIER`), el `DMA` es un módulo avanzado que analizaremos hacia el final del apunte. Para más información consulte la hoja de datos.

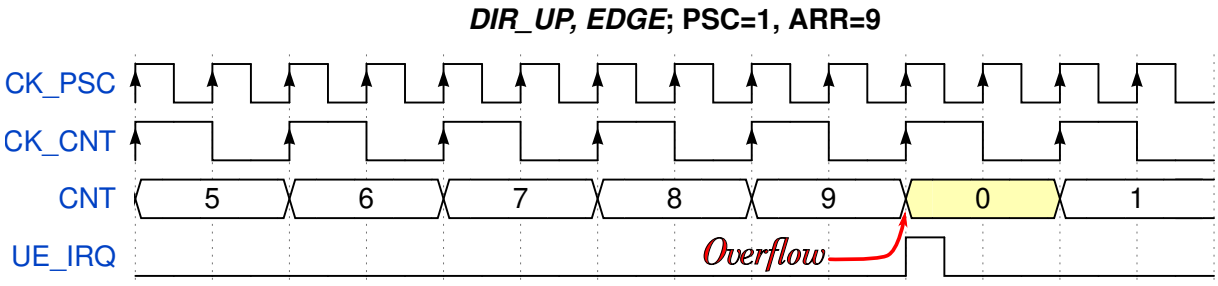
```
/* Habilitación de interrupciones y eventos de los timers */
void timer_enable_irq(uint32_t timer_peripheral, uint32_t irq)

// defines para 'irq' en un TIM de Propósito general
// -- Interrupciones --
TIM_DIER_UIE    // UIE:   Update interrupt enable
TIM_DIER_CC1IE  // CC1IE: Capture/compare 1 interrupt enable
TIM_DIER_CC2IE  // CC2IE: Capture/compare 2 interrupt enable
TIM_DIER_CC3IE  // CC3IE: Capture/compare 3 interrupt enable
TIM_DIER_CC4IE  // CC4IE: Capture/compare 4 interrupt enable
TIM_DIER_TIE    // TIE:   Trigger interrupt enable
// -- DMA --
TIM_DIER_UDE    // UDE:   Update DMA request enable
TIM_DIER_CC1DE  // CC1DE: Capture/Compare 1 DMA request enable
TIM_DIER_CC2DE  // CC2DE: Capture/Compare 2 DMA request enable
TIM_DIER_CC3DE  // CC3DE: Capture/Compare 3 DMA request enable
TIM_DIER_CC4DE  // CC4DE: Capture/Compare 4 DMA request enable
TIM_DIER_TDE    // TDE:   Trigger DMA request enable
```

Para poder utilizarlo como temporizador utilizaremos la interrupción que se ejecuta al sobrepasar (*Overflow*) el valor en el registro de Auto-Recarga (ARR) conocida como *Update Interrupt*. Como se observa en la linea 31 del ejemplo. Luego habilitamos la interrupción en el NVIC como lo discutimos en la Unidad 3. Como ya vimos, las interrupciones podrían ser causadas por varias situaciones, es por ello que debemos consultar el *flag* correspondiente para luego bajarlo, las cuales se encuentran en el **Status Register** (SR). Para ello utilizamos las funciones:

```
bool timer_get_flag(uint32_t timer_peripheral, uint32_t flag);
void timer_clear_flag(uint32_t timer_peripheral, uint32_t flag);

// defines para 'flag' en un TIM de propósito general
TIM_SR_UIF      // UIF:   Update Interrupt Flag
TIM_SR_CC1IF    // CC1IF: Capture/compare 1 interrupt flag
TIM_SR_CC2IF    // CC2IF: Capture/compare 2 interrupt flag
TIM_SR_CC3IF    // CC3IF: Capture/compare 3 interrupt flag
TIM_SR_CC4IF    // CC4IF: Capture/compare 4 interrupt flag
TIM_SR_TIF      // TIF:   Trigger interrupt flag
TIM_SR_CC1OF    // CC1OF: Capture/compare 1 overcapture flag
TIM_SR_CC2OF    // CC2OF: Capture/compare 2 overcapture flag
TIM_SR_CC3OF    // CC3OF: Capture/compare 3 overcapture flag
TIM_SR_CC4OF    // CC4OF: Capture/compare 4 overcapture flag
```

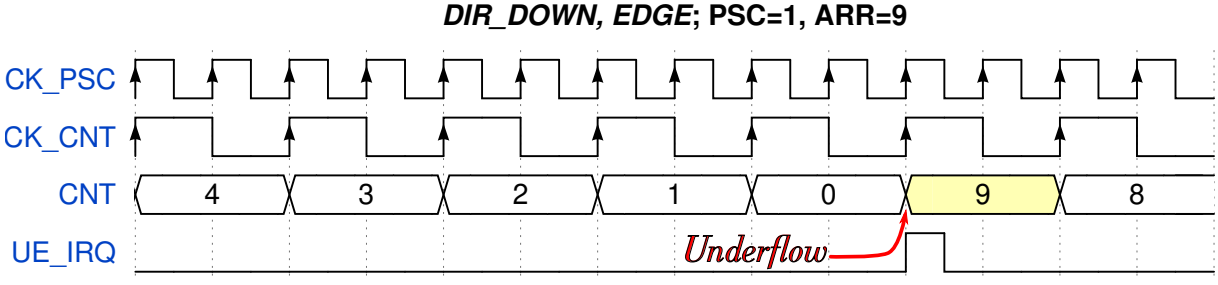
En nuestro ejemplo sólo habilitamos una fuente de interrupción, por ello no utilizamos `timer_get_flag()` y simplemente bajamos el *flag* del *Update Interrupt* en la linea 45 dentro de la sub-rutina de interrupción del **TIM2** llamada `tim2_isr()`.

Aquí analizaremos el comportamiento de este modo en un diagrama temporal, suponiendo valores de PSC=1 (CK_CNT = CK_INT/2) y ARR=9:



4.4.2. Contador descendente

Para transformar nuestro ejemplo en modo descendente debemos cambiar simplemente la linea 23 del ejemplo en la pág. 40, reemplazando por el define TIM_CR1_DIR_DOWN. A los efectos prácticos de nuestro programa no notaremos ninguna diferencia. Este modo el comportamiento es prácticamente igual al del *SysTick*.



4.4.3. Modo centrado (cuenta ascendente/descendente)

En este modo el contador va desde 0 y genera un *overflow* al pasar el valor de `ARR-1`, para luego contar desde `ARR` hasta 1 generando un *underflow* al llegar a 0 y allí vuelve a empezar la cuenta. Cada uno de esos eventos genera un *Update Interrupt*. En este caso se ignora la configuración de dirección (`UP/DOWN`), ya que es escrita por hardware, pero podemos consultar si está subiendo o bajando directamente del registro utilizando la macro:

```
TIM_CR1(timer_peripheral) // donde 'timer_peripheral' => TIM1, TIM2, etc.
```

Veamos cómo modificar el ejemplo y leer el bit de dirección del registro:

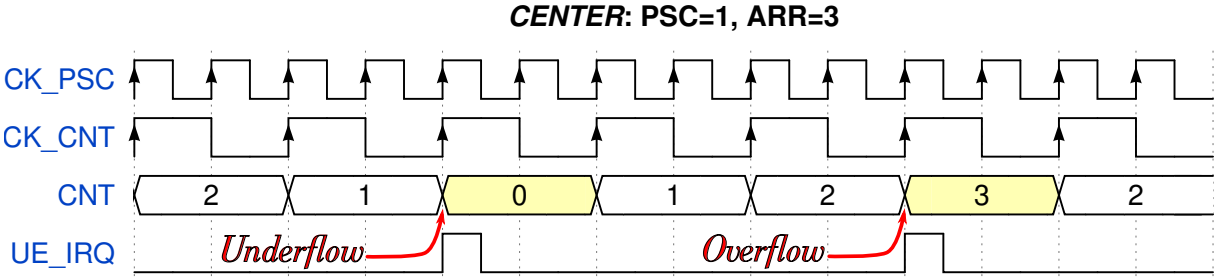
cnt-cent.c

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/nvic.h>

int main(void)
{
    // Se configura el clock del sistema a 72Mhz
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    // Se configura el LED de la BluePill (PC13) como Salida Push-Pull
    rcc_periph_clock_enable(RCC_GPIOC);
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);

    // Se habilita el clock del periférico del TIM2
    rcc_periph_clock_enable(RCC_TIM2);

    // Se configura el modo de funcionamiento del Timer2
    timer_set_mode(
        TIM2,                // Timer2
        TIM_CR1_CKD_CK_INT,  // Clk de sampleo == CK_INT, (no se utiliza)
        TIM_CR1_CMS_CENTER_1, // Alineado al centro MODO 1 (indistinto para este ejemplo)
        TIM_CR1_DIR_UP);     // Es ignorado, será escrito por hardware

    timer_set_prescaler(TIM2, 35999); // 72MHz / 36000 => 2KHz

    // Se setea el valor hasta donde se cuenta:
    timer_set_period(TIM2, 1000); // 2KHz / 1000 = 2Hz => T = 0.5s

    // Se habilita al interrupción por Update Interrupt (underflow & overflow)
    timer_enable_irq(TIM2, TIM_DIER_UIE);

    // Empieza a contar el Timer2
    timer_enable_counter(TIM2);

    // Se habilita la interrupción desde el NVIC
    nvic_enable_irq(NVIC_TIM2_IRQ);

    while (1)
        ; // Nada por hacer, esperando la interrupción
}

void tim2_isr(void)
{                                       // Cada 0.5s:
    timer_clear_flag(TIM2, TIM_SR_UIF);   // Se baja el flag de la interrupción
    if (TIM_CR1(TIM2) & TIM_CR1_DIR_DOWN) // Si la cuenta es descendente:
        gpio_toggle(GPIOC, GPIO13);       // Se alterna el LED
}
```

Además existen 3 modos en las cuentas alineadas al centro, pero esto lo cubriremos cuando hablemos de los comparadores, ya que es allí donde influye esta

configuración.

Para poder hacer una comparación, analicemos un diagrama temporal similar a los anteriores:



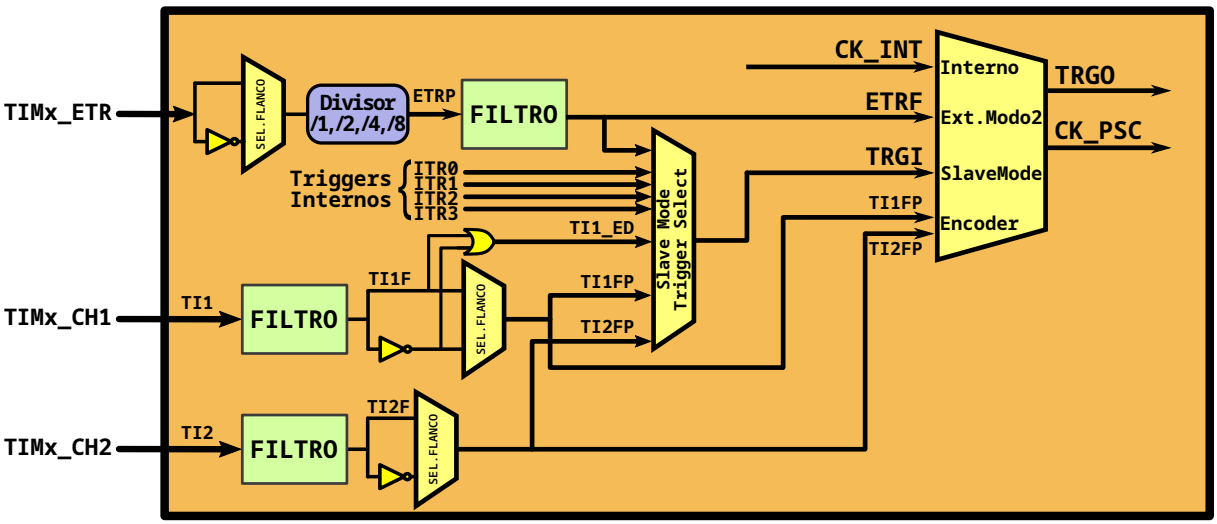
4.5. Modo contador (ETR & TIx)

El segundo modo clásico de utilizar un timer en cualquier microcontrolador es como contador. Es decir a través de pulsos que entran externamente por un pin. En este sentido tenemos dos posibilidades con los STM32, una es utilizar el pin `TIMx_ETR` y la otra es utilizando el *Modo Esclavo*, el cual incluye como fuentes el `TI1` (`TIMx_CH1`), `TI2` (`TIMx_CH2`) o incluso varias fuentes internas.

No todos lo timers tienen la entrada **ETR**, en el caso del *STM32F103Cx* solo el **TIM1** y el **TIM2** tienen esta entrada. Siendo el pin `PA12` el `TIM1_ETR` y el `PA0` corresponde al `TIM2_ETR`, aunque este último comparte su ubicación con el `TIM2_CH1`.

4.5.1. Fuentes de clock

Antes de proseguir, debemos analizar con un poco más de profundidad las posibles fuentes de clock para los **TIM**. De esta manera se entenderá mejor las configuraciones posibles.



Como se observa en la figura anterior, podemos tener básicamente 4 fuentes de clock (como ya habíamos observado anteriormente). Pero aquí con un poco más

de detalle en la entrada ETR y las funtes para TRGI, el cual está asociado con el *Modo Esclavo*. En este modo puede recibir pulsos desde otro **TIM** que haya sido configurado como *Maestro* (salida TRGO) y entrando por algunas de las ITRx. O bien, puede utilizarse la misma entrada ETR, ambos flancos de TI1 (TI1_ED) o seleccionando un flanco (ascendente o descendente) de TI1 o TI2.

4.5.2. Modo externo 2 (utilizando ETR)

Como se ha detallado anteriormente, es el modo externo más clásico, un pulso entra por el pin ETR asociado al **TIM**, seleccionando el flanco correspondiente. Este cuenta con un divisor, ya que la frecuencia máxima de entrada no puede superar a la cuarta parte del CK_INT ($\frac{CK\_INT}{4}$). En el supuesto caso de 72Mhz, no podremos superar los 18Mhz para ETRP.

Este modo no está soportado plenamente en libOpenCM3, ya que no cuenta con funciones específicas para este modo. Por lo tanto, debemos utilizar las macros para configurar los bits del registro correspondiente. De cierta manera está ligado al *Modo Esclavo*, es por ello que los bits de configuración están en el SMCR (*Slave Mode Control Register*) siendo el byte más alto (del bit 8 al 15) los que controlan el ETR.

TIMx slave mode control register (TIMx_SMCR)

Address offset: 0x08

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			Res.	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw

- **ETP:** *External trigger polarity*
  0: Cuenta flancos ascendentes entrantes por ETR.
  1: Cuenta flancos descendentes entrantes por ETR.
- **ECE:** *External clock enable*
  0: Modo Externo 2 inhabilitado.
  1: Modo Externo 2 habilitado. Cuenta pulsos entrantes por ETR.
- **ETPS:** *External trigger prescaler*
  00: /1 (sin prescaler)
  01: /2
  10: /4
  11: /8
- **ETF:** *External trigger filter*. Valida el estado luego de N lecturas consecutivas.
  0000: sin filtro
  0001: $f_{SAMPLEO} = f_{CK\_INT}, N = 2$
  ...
  0111: $f_{SAMPLEO} = f_{DTS}/4, N = 8$
  ...
  1111: $f_{SAMPLEO} = f_{DTS}/32, N = 8$

Para poder probar este modo y los demás, haremos un generador de 80Khz utilizando el **TIM2**. Por el `PA10` se envía la señal para a contar por el otro dispositivo.

generador.c

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/cm3/systick.h>

volatile uint32_t millis;

void delay_ms(uint32_t ms);

int main(void)
{
    /* Se configura el SYSCLK en 72Mhz */
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    /* Se configura el pin que envía los pulsos a la otra bluepill */
    rcc_periph_clock_enable(RCC_GPIOA);
    gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO10);

    /* Se configura el LED de la bluepill para constatar la frecuencia */
    rcc_periph_clock_enable(RCC_GPIOC);
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);

    /* Se configura el TIM2 para generar una señal de 80Khz */
    rcc_periph_clock_enable(RCC_TIM2); // Se habilita el clock
    timer_set_mode(
        TIM2,               // Timer 2
        TIM_CR1_CKD_CK_INT, // No usamos el filtro, es indistinto
        TIM_CR1_CMS_EDGE,   // Alineado al flanco
        TIM_CR1_DIR_UP);    // Cuenta ascendente

    timer_set_prescaler(TIM2, 0);         // PSC: 0   => 72Mhz/1 (PSC+1)
    timer_set_period(TIM2, 450 - 1);      // ARR: 449 => 72Mhz/450 = 160Khz
    timer_enable_irq(TIM2, TIM_DIER_UIE); // Se habilita la interrupción
    timer_enable_counter(TIM2);           // Empíeza la cuenta

    /* Se configura el SysTick para usar delay_ms() */
    systick_set_frequency(1000, rcc_ahb_frequency);
    systick_interrupt_enable();
    systick_counter_enable();

    /* Se habilita la interrupción del TIM2 en el NVIC */
    nvic_enable_irq(NVIC_TIM2_IRQ);

    while (true)
    {
        gpio_toggle(GPIOC, GPIO13); // LED de verificación
        delay_ms(500);
    }
}

void tim2_isr(void)
{                                       // cada 6.25µs:
    timer_clear_flag(TIM2, TIM_SR_UIF); // Se baja el flag
    gpio_toggle(GPIOA, GPIO10);         // Salida de 80Khz
}

void delay_ms(uint32_t ms)
{
    uint32_t tm = millis + ms;
    while (millis < tm);
}

void sys_tick_handler(void)
{ // cada 1ms:
    millis++;
}
```

Y aquí se expone el código de la bluepill que recibe los pulsos por el TIM2_ETR (PA0), donde vemos la configuración correspondiente en las lineas 28 a 32.

ext_mode_2.c

```
1  #include <libopencm3/stm32/rcc.h>
2  #include <libopencm3/stm32/gpio.h>
3  #include <libopencm3/stm32/timer.h>
4  #include <libopencm3/cm3/nvic.h>
5
6  int main(void)
7  {
8      /* SysClk a 72Mhz */
9      rcc_clock_setup_in_hse_8mhz_out_72mhz();
10
11     /* Se configura como salida el LED de la bluepill */
12     rcc_periph_clock_enable(RCC_GPIOC);
13     gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
14
15     /* Se configura el TIM2 como modo externo 2 (entrada por ETR) */
16     rcc_periph_clock_enable(RCC_TIM2);
17     timer_set_mode(
18         TIM2,                    // Timer 2
19         TIM_CR1_CKD_CK_INT_MUL_4, // Tdts = Tck_int x 4 => Fdts = 72Mhz/4 = 18Mhz
20         TIM_CR1_CMS_EDGE,        // Alineado al flanco
21         TIM_CR1_DIR_UP);         // Contador ascendente
22
23     // Se esperan pulsos a 80Khz
24     timer_set_prescaler(TIM2, 2000 - 1); // 80Khz/2000 => 40Hz
25     timer_set_period(TIM2, 20 - 1);      // 40Hz/20 => 2Hz
26
27     // ETP = 0 (flanco ascendente)
28     TIM_SMCR(TIM2) &= ~TIM_SMCR_ETP;
29     // Fsampling = fdts/8 = 18Mhz/8 = 2.25Mhz y 8 lecturas consecutivas para validar
30     TIM_SMCR(TIM2) |= TIM_SMCR_ETF_DTS_DIV_8_N_8;
31     // Se habilita el External Counter (External Mode 2)
32     TIM_SMCR(TIM2) |= (TIM_SMCR_ECE);
33
34     timer_enable_irq(TIM2, TIM_DIER_UIE); // Se configura la Update Interrupt
35     timer_enable_counter(TIM2);           // Empieza a contar pulsos
36
37     // Se habilita la interrupción del TIM2 en el NVIC
38     nvic_enable_irq(NVIC_TIM2_IRQ);
39
40     while (true); // Esperando pulsos
41 }
42
43 void tim2_isr(void)
44 {                                      // cada 3.200M de pulsos entrantes por ETR:
45     timer_clear_flag(TIM2, TIM_SR_UIF); // Se baja el flag
46     gpio_toggle(GPIOC, GPIO13);         // Se alterna el LED
47 }
```

En este ejemplo, utilizamos los defines correspondientes al registro SMCR asociados al ETR. No se utilizó el prescaler, pero aquí se enumeran todos los defines utilizables para configurar todos los bits correspondientes.

```
/* ETP: External trigger polarity */
#define TIM_SMCR_ETP                (1 << 15)

/* ECE: External clock enable */
#define TIM_SMCR_ECE                (1 << 14)

/* ETPS: External trigger prescaler */
#define TIM_SMCR_ETPS_OFF           (0x0 << 12)
#define TIM_SMCR_ETPS_ETRP_DIV_2    (0x1 << 12)
#define TIM_SMCR_ETPS_ETRP_DIV_4    (0x2 << 12)
#define TIM_SMCR_ETPS_ETRP_DIV_8    (0x3 << 12)
#define TIM_SMCR_ETPS_MASK          (0X3 << 12)
```

```
/* ETF: External trigger filter */
#define TIM_SMCR_ETF_OFF            (0x0 << 8)
#define TIM_SMCR_ETF_CK_INT_N_2     (0x1 << 8)
#define TIM_SMCR_ETF_CK_INT_N_4     (0x2 << 8)
#define TIM_SMCR_ETF_CK_INT_N_8     (0x3 << 8)
#define TIM_SMCR_ETF_DTS_DIV_2_N_6  (0x4 << 8)
#define TIM_SMCR_ETF_DTS_DIV_2_N_8  (0x5 << 8)
#define TIM_SMCR_ETF_DTS_DIV_4_N_6  (0x6 << 8)
#define TIM_SMCR_ETF_DTS_DIV_4_N_8  (0x7 << 8)
#define TIM_SMCR_ETF_DTS_DIV_8_N_6  (0x8 << 8)
#define TIM_SMCR_ETF_DTS_DIV_8_N_8  (0x9 << 8)
#define TIM_SMCR_ETF_DTS_DIV_16_N_5 (0xA << 8)
#define TIM_SMCR_ETF_DTS_DIV_16_N_6 (0xB << 8)
#define TIM_SMCR_ETF_DTS_DIV_16_N_8 (0xC << 8)
#define TIM_SMCR_ETF_DTS_DIV_32_N_5 (0xD << 8)
#define TIM_SMCR_ETF_DTS_DIV_32_N_6 (0xE << 8)
#define TIM_SMCR_ETF_DTS_DIV_32_N_8 (0xF << 8)
#define TIM_SMCR_ETF_MASK           (0xF << 8)
```

4.5.3. Modo Externo 1

Para poder acceder a este funcionamiento debemos configurar el Modo Esclavo utilizando la función `timer_slave_set_mode()`:

```
void timer_slave_set_mode(uint32_t timer, uint8_t mode);

// defines para 'mode'
TIM_SMCR_SMS_OFF    // Modo Esclavo desactivado
TIM_SMCR_SMS_EM1    // Modo Encoder 1
TIM_SMCR_SMS_EM2    // Modo Encoder 2
TIM_SMCR_SMS_EM3    // Modo Encoder 3
TIM_SMCR_SMS_RM     // Modo Reset
TIM_SMCR_SMS_GM     // Modo Gate
TIM_SMCR_SMS_TM     // Modo Trigger
TIM_SMCR_SMS_ECM1   // Modo Externo 1
```

En este modo se puede seleccionar varias fuentes como se observo en la imagen detallada del clock en el apartado 4.5.1. Serán seleccionadas con la función `timer_slave_set_trigger()`. A continuación se detallan las posibles selecciones:

1. **Triggers Internos (`ITRx`):** Estos provienen de otros TIMs configurados en *Modo Maestro*.
2. **Detector de Flancos de TI1 (`TI1F_ED`)**: Este genera un pulso por cada flanco ascendente y descendente proveniente de TI1.
3. **TI1 Filtrado (`TI1FP`):** Entrada por el `TIMx_CH1`.
4. **TI2 Filtrado (`TI2FP`):** Entrada por el `TIMx_CH2`.
5. **Trigger Externo (`ETRF`):** Entrada por el pin `TIMx_ETR`.

```
void timer_slave_set_trigger(uint32_t timer, uint8_t trigger);

// defines para 'trigger'
TIM_SMCR_TS_ITR0    // Entrada interna 0 (desde otro timer)
TIM_SMCR_TS_ITR1    // Entrada interna 1 (desde otro timer)
TIM_SMCR_TS_ITR2    // Entrada interna 2 (desde otro timer)
TIM_SMCR_TS_ITR3    // Entrada interna 3 (desde otro timer)
TIM_SMCR_TS_TI1F_ED // Flancos asdentes y descendetes de TI1
TIM_SMCR_TS_TI1FP1  // Flancos (filtrados) desde TI1
TIM_SMCR_TS_TI2FP2  // Flancos (filtrados) desde TI2
TIM_SMCR_TS_ETRF    // Flancos (filtrados) desde ETR
```

Para configurar los filtros, polaridad y prescaler del ETR y los TI1 y TI2 tenemos las siguientes funciones:

```
// Para el ETR:
void timer_slave_set_filter(uint32_t timer, enum tim_ic_filter flt);
void timer_slave_set_prescaler(uint32_t timer, enum tim_ic_psc psc);
void timer_slave_set_polarity(uint32_t timer, enum tim_et_pol pol);
// Para el TIx (Entrada IC = Input Capture):
void timer_ic_set_filter(uint32_t timer, enum tim_ic_id ic, enum tim_ic_filter flt);
void timer_ic_set_polarity(uint32_t timer, enum tim_ic_id ic, enum tim_ic_pol pol);

// valores del enum tim_ic_id 'ic' (solo para TIx)
TIM_IC1, TIM_IC2, TIM_IC3, TIM_IC4

// valores para el enum tim_ic_filter 'ftl'
TIM_IC_OFF
TIM_IC_CK_INT_N_2, TIM_IC_CK_INT_N_4,TIM_IC_CK_INT_N_8
TIM_IC_DTF_DIV_2_N_6, TIM_IC_DTF_DIV_2_N_8
TIM_IC_DTF_DIV_4_N_6, TIM_IC_DTF_DIV_4_N_8
TIM_IC_DTF_DIV_8_N_6, TIM_IC_DTF_DIV_8_N_8
TIM_IC_DTF_DIV_16_N_5, TIM_IC_DTF_DIV_16_N_6, TIM_IC_DTF_DIV_16_N_8
TIM_IC_DTF_DIV_32_N_5, TIM_IC_DTF_DIV_32_N_6, TIM_IC_DTF_DIV_32_N_8

// valores para el enum tim_ic_psc 'psc' (solo para ETR)
TIM_IC_PSC_OFF
TIM_IC_PSC_2
TIM_IC_PSC_4
TIM_IC_PSC_8

// valores para enum tim_et_pol 'pol'
TIM_ET_RISING
TIM_ET_FALLING

// valores para enum tim_ic_pol 'pol'
TIM_IC_RISING
TIM_IC_FALLING
```

En el siguiente ejemplo utilizaremos el `TI2` para recibir los pulsos de 80Khz del `generador.c` (apartado 4.5.2) modificando la configuración del **TIM2** entre las lineas de la 27 a la 30, para utilizar el `TIM2_CH2` (`PA1`).

ext_mode_1.c

```
1  #include <libopencm3/stm32/rcc.h>
2  #include <libopencm3/stm32/gpio.h>
3  #include <libopencm3/stm32/timer.h>
4  #include <libopencm3/cm3/nvic.h>
5
6  int main(void)
7  {
8      /* SysClk a 72Mhz */
9      rcc_clock_setup_in_hse_8mhz_out_72mhz();
10
11     /* Se configura como salida el LED de la bluepill */
12     rcc_periph_clock_enable(RCC_GPIOC);
13     gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
14
15     /* Se configura el TIM2 como modo externo 2 (entrada por ETR) */
16     rcc_periph_clock_enable(RCC_TIM2);
17     timer_set_mode(
18         TIM2,                     // Timer 2
19         TIM_CR1_CKD_CK_INT_MUL_4, // Tdts = CK_INT/4 => Fdts = 72Mhz/4 = 18Mhz
20         TIM_CR1_CMS_EDGE,         // Alineado al flanco
21         TIM_CR1_DIR_UP);          // Contador ascendente
22
23     // Se esperan pulsos a 80Khz
24     timer_set_prescaler(TIM2, 2000 - 1); // 80Khz/2000 => 40Hz
25     timer_set_period(TIM2, 20 - 1);      // 40Hz/20 => 2Hz
26
```

```
27      timer_slave_set_mode(TIM2, TIM_SMCR_SMS_ECM1);            // External Counter Mode 1
28      timer_slave_set_trigger(TIM2, TIM_SMCR_TS_TI2FP2);        // Trigger: TI2FP
29      timer_ic_set_filter(TIM2, TIM_IC2, TIM_IC_DTF_DIV_8_N_8); // Fsamp: 2.25Mh 8 lecturas
30      timer_ic_set_polarity(TIM2, TIM_IC2, TIM_IC_RISING);      // flanco ascendete
31
32      // Se configura el Update Interrupt
33      timer_enable_irq(TIM2, TIM_DIER_UIE);
34      timer_enable_counter(TIM2);
35
36      // Se habilita la interrupción del TIM2 en el NVIC
37      nvic_enable_irq(NVIC_TIM2_IRQ);
38
39      while (true);
40  }
41
42  void tim2_isr(void)
43  {                                       // cada 3.200M de pulsos entrantes por TI2FP:
44      timer_clear_flag(TIM2, TIM_SR_UIF); // Se baja el flag
45      gpio_toggle(GPIOC, GPIO13);         // Se alterna el LED
46  }
```

En el caso de querer utilizar el `ETR` nuevamente, pero en el modo externo 1 (a través del Controlador del *Modo Esclavo*), debemos cambiar las lineas de la 28, 29 y 30 con las siguientes funciones. Recordando que el pin correspondiente es (`PA0`).

```
timer_slave_set_trigger(TIM2, TIM_SMCR_TS_ETRF);
timer_slave_set_filter(TIM2, TIM_IC_DTF_DIV_8_N_8);
timer_slave_set_polarity(TIM2, TIM_ET_RISING);
```

4.6. Canales de Captura/Comparadores

Como ya se ha expuesto los **TIMs** tienen 4 canales de Entrada/Salida. Estos están asociados a registros de captura y comparación de valores con respecto al contador. Con ellos podemos generar señales de salida o capturar eventos de entrada, generando interrupciones o controlando el timer a través del *Modo Esclavo*.

4.6.1. Modo Entrada de Captura (ICx)

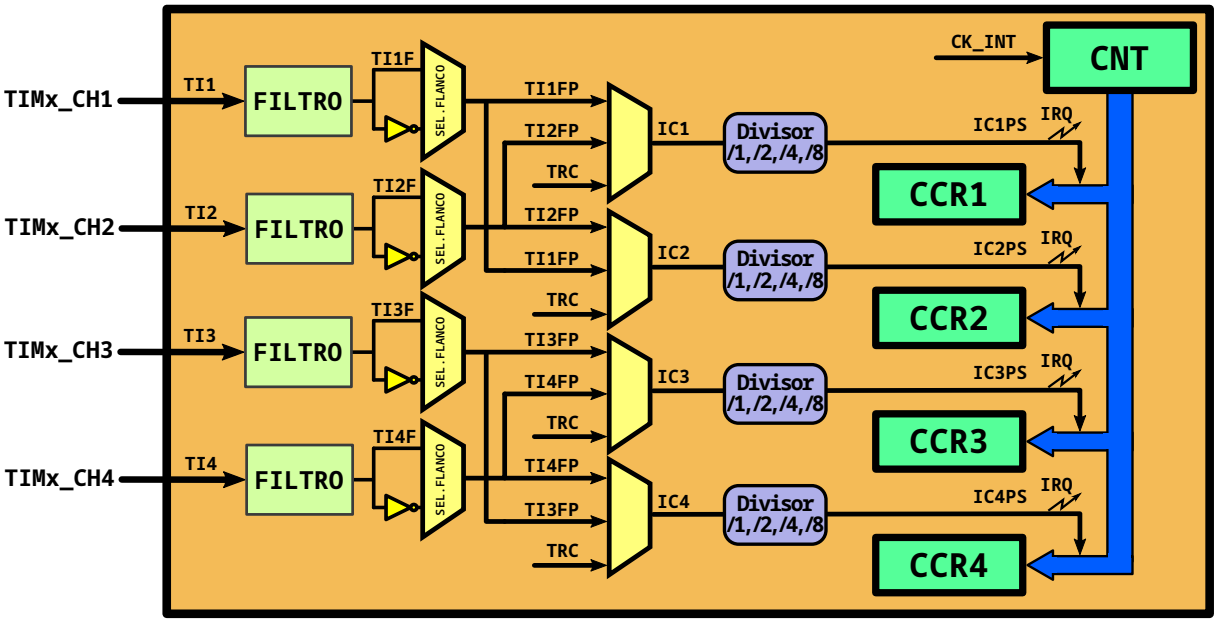
Los canales pueden ser utilizados como entradas y *capturar* valores del timer ante eventos externos. Esto podría ser útil en caso de querer contabilizar o hacer mediciones de tiempo. Para tal fin, son utilizadas señales internas llamadas `ICx` (*Input Capture*), asociados a los `TIx`.

Como podemos observar en la siguiente figura, las entradas `TI1` y `TI2` como las `TI3` y `TI4` pueden cruzarse para generar la señal `ICx` correspondientes a cada **Registro de Captura/Comparación** (`CCRx`). Incluso podemos utilizar la misma entrada `TIx` para 2 comparadores. La entrada `TRC` es generada por el *Controlador del Modo Esclavo* y la fuente puede ser un *Trigger Interno* (`ITRx`) o `TI1_ED`.

El pulso generado en `ICxPS` (luego del prescaler/divisor) habilita a que se guarde el valor actual del contador del **TIM** en el `CCRx` al instante. Esto puede generar una interrupción y/o una petición al **DMA**.

En el siguiente ejemplo se demuestra cómo capturar el valor del `CNT` a través del `TI1` asociado al `IC1`. Este se almacenará en `CCR1`, el cual será accedido a través

de la macro `TIM_CCR1()`, indicando el **TIM** correspondiente. Se utilizarán LEDs para observar el cambio en el registro.



ic_toggle.c

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/nvic.h>

int main(void)
{
    /* Configuración del SysClk a 721mhz */
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    /* Configuración de LEDs para visualizar el comportamiento */
    rcc_periph_clock_enable(RCC_GPIOC);
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
    gpio_set(GPIOC, GPIO13);
    rcc_periph_clock_enable(RCC_GPIOB);
    gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO6);
    gpio_set(GPIOB, GPIO6);

    /* Se configura el TIM2 para capturar flancos descendetes del CH1 (PA0) */
    rcc_periph_clock_enable(RCC_TIM2);
    timer_set_mode(TIM2, TIM_CR1_CKD_CK_INT_MUL_4, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
    timer_set_prescaler(TIM2, 1440 - 1);  // 72Mhz / 1440 = 50Khz
    timer_set_period(TIM2, 50000 - 1);    // 50Khz / 50K  = 1Hz

    // Se configura el filto, polaridad y fuente de entrada para IC1
    timer_ic_set_filter(TIM2, TIM_IC1, TIM_IC_DTF_DIV_8_N_8); // Fsamp = Fdts/8, N=8
    timer_ic_set_polarity(TIM2, TIM_IC1, TIM_IC_FALLING);     // Flanco descendente
    timer_ic_set_input(TIM2, TIM_IC1, TIM_IC_IN_TI1);         // IC1 <- TI1 (CH1)

    // Sin prescaler. Apenas se detecte el flanco, se captura el valor del TIM2
    timer_ic_set_prescaler(TIM2, TIM_IC1, TIM_IC_PSC_OFF);
    timer_ic_enable(TIM2, TIM_IC1);       // Se habilita el la entrada IC1

    // Se habilita la Interrupción por Update Event (overflow)
    timer_enable_irq(TIM2, TIM_DIER_UIE);

    // Empieza a contar el TIM2
    timer_enable_counter(TIM2);

    // Se habilita la interrupción en el NVIC
```

```
41      nvic_enable_irq(NVIC_TIM2_IRQ);
42
43      while (true)
44      {
45          // Si el valor capturado es menor o igual al valor actual se apaga LED
46          if (TIM_CCR1(TIM2) <= timer_get_counter(TIM2))
47          {
48              gpio_set(GPIOB, GPIO6);
49          }
50          else // sino se prende el LED
51          {
52              gpio_clear(GPIOB, GPIO6);
53          }
54      }
55  }
56
57  void tim2_isr(void)
58  { // Se alterna el LED para verificar el ciclo capturado.
59      timer_clear_flag(TIM2, TIM_SR_UIF);
60      gpio_toggle(GPIOC, GPIO13); // Toggle del PC13 en 2Hz
61  }
```

Podemos hacer un seguimiento de la configuración observando las lineas de la 26 a la 28 y la figura de la pág.51. Donde se configuran el filtro, el flanco a detectar y por último la selección para la señal `IC1`. Y luego en las lineas 31 y 32 se selecciona el divisor `/1` (sin prescaler), y finalmente se habilita la captura. Todas estas funciones empiezan con `timer_ic` y se enumeran a continuación junto a los posibles atributos para sus parámetros.

```
// Valores del enum tim_ic_id 'ic':
TIM_IC1, TIM_IC2, TIM_IC3, TIM_IC4

void timer_ic_set_filter(uint32_t timer, enum tim_ic_id ic, enum tim_ic_filter flt);
// Valores del enum tim_ic_filter 'flt'
TIM_IC_OFF
TIM_IC_CK_INT_N_2, TIM_IC_CK_INT_N_4, TIM_IC_CK_INT_N_8
TIM_IC_DTF_DIV_2_N_6, TIM_IC_DTF_DIV_2_N_8
TIM_IC_DTF_DIV_4_N_6, TIM_IC_DTF_DIV_4_N_8
TIM_IC_DTF_DIV_8_N_6, TIM_IC_DTF_DIV_8_N_8
TIM_IC_DTF_DIV_16_N_5, TIM_IC_DTF_DIV_16_N_6, TIM_IC_DTF_DIV_16_N_8
TIM_IC_DTF_DIV_32_N_5, TIM_IC_DTF_DIV_32_N_6, TIM_IC_DTF_DIV_32_N_8

void timer_ic_set_polarity(uint32_t timer, enum tim_ic_id ic, enum tim_ic_pol pol);
// Valores del enum tim_ic_pol 'pol'
TIM_IC_RISING
TIM_IC_FALLING

void timer_ic_set_input(uint32_t timer, enum tim_ic_id ic, enum tim_ic_input in);
// Valores del enum tim_ic_input 'in'
TIM_IC_OUT    // Canal configurado como salida
TIM_IC_IN_TI1 // valido para el IC1 o IC2
TIM_IC_IN_TI2 // valido para el IC1 o IC2
TIM_IC_IN_TRC
TIM_IC_IN_TI3 // valido para el IC3 o IC4
TIM_IC_IN_TI4 // valido para el IC3 o IC4

void timer_ic_set_prescaler(uint32_t timer, enum tim_ic_id ic, enum tim_ic_psc psc);
// Valores del enum tim_ic_psc 'psc'
TIM_IC_PSC_OFF
TIM_IC_PSC_2
TIM_IC_PSC_4
TIM_IC_PSC_8

void timer_ic_enable(uint32_t timer, enum tim_ic_id ic);
void timer_ic_disable(uint32_t timer, enum tim_ic_id ic);
```

En caso de haber querido utilizar la interrupción del comparador podríamos

haberlo hecho en la linea 35 agregando el `CC1IE`:

```
    timer_enable_irq(TIM2, TIM_DIER_UIE | TIM_DIER_CC1IE);
```

Dentro de la interrupción debemos consultar con `timer_get_flag()` si fue el `UIF` o el `CC1IF` la fuente de interrupción. El flag de interrupción de los comparadores puede bajarse por software utilizando `timer_clear_flag()` o por hardware simplemente al leer el contenido del `CCRx` correspondiente.

```
    void tim2_isr(void)
    {
        if(timer_get_flag(TIM2, TIM_SR_UIF)) // fue el overflow
        {
            timer_clear_flag(TIM2, TIM_SR_UIF);
            gpio_toggle(GPIOC, GPIO13);
        }

        if(timer_get_flag(TIM2, TIM_SR_CC1IF)) // fue el comparador
        {
            // Se lee el valor no hace falta el timer_clear_flag().
            n = TIM_CCR1(TIM2);
        }
    }
```

4.6.2. Modulación por Ancho de Pulso (PWM)

A continuación se expondrá cómo utilizar los **OCx** (***Output Compare*** - *Comparadores de Salida*) específicamente para generar señales **PWM**. No es su único modo de funcionamiento, pero es uno de los más habituales. Esta técnicas es muy conocida por su amplia difusión. Consta con 2 parámetros básicos:

- **Frecuencia:** Es la frecuencia base para el PWM, determina la duración de un ciclo completo, es decir el **Período** (**T**).
- **Ciclo de trabajo (*duty cycle*):** Es la porción de tiempo en que la señal se mantiene activa en un **Período** (**T**).

Para más información puede acceder a la Subsección 4.10, donde encontrará un link a un artículo muy completo al respecto.

La señal PWM puede ser activa en ALTO o BAJO, para ello debemos utilizar la función:

```
    void timer_set_oc_mode(uint32_t timer_peripheral, enum tim_oc_id oc_id, enum tim_oc_mode
        oc_mode);

    /* Valores para el enum tim_oc_mode: */
    TIM_OCM_PWM1    // PWM activo en ALTO
    TIM_OCM_PWM2    // PWM activo en BAJO

    // Existen más opciones, pero no serán abarcadas aquí
```

Por último debemos habilitar el canal, de lo contrario no saldrá ninguna señal de el pin:

```
void timer_enable_oc_output(uint32_t timer_peripheral, enum tim_oc_id oc_id);
```

Para poner un valor en el registro del **OCx** (*Output Compare*) para definir el *duty cycle* tenemos la función:

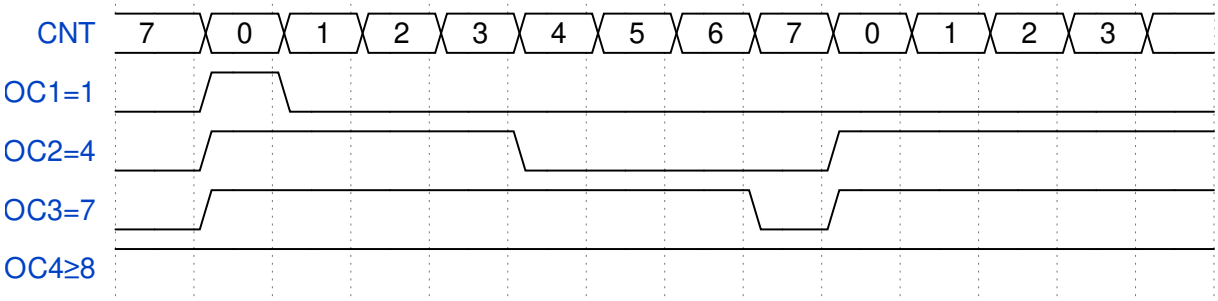
```
void timer_set_oc_value(uint32_t timer_peripheral, enum tim_oc_id oc_id, uint32_t value);

// Donde 'value' es un número entre 0 y el valor del ARR+1 para flanco y ARR para centrado
    .
```

Existen varias configuraciones posibles para el **PWM**, pero solo analizaremos 2 de ellas. Una más *clásica*, que es alineada al flanco (modo por default en la mayoría de los µC) y otra conocida como alineada al centro o con *corrección de fase*.

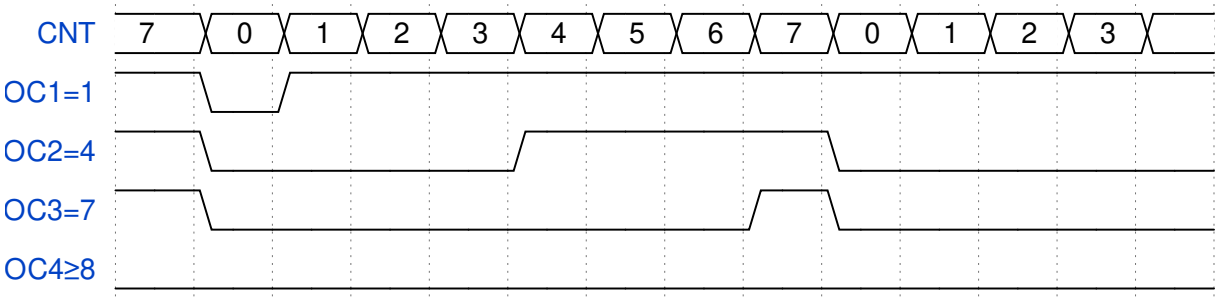
Alineado al flanco

En el PWM alineado al flanco, las señales arrancan siempre al reiniciar el timer, es decir, cuando la cuenta llega a 0 el *duty cycle* empieza, permanece activa el tiempo configurado en el **OCx** correspondiente y luego permanece inactiva hasta que el periodo vuelve a empezar como se observa en los gráficos correspondientes.

***Alineado al Flanco*: PWM1 (A. HIGH), ARR=7**



***Alineado al Flanco*: PWM2 (A. LOW), ARR=7**



EL **OCx** permanece activo hasta alcanzar el valor de **ARR**, es por ello que si se desea que la señal permanezca activa durante todo el ciclo se debe poner un valor de *al menos* `ARR+1` como se había anticipado (**OC4** en los gráficos).

En el siguiente ejemplo se conecta un LED al `PB6` (`TIM4_CH1`) activo en BAJO y se lo hará prender y apagar linealmente utilizando PWM a 1Khz. Con una variable de control (`sentido`) se enviarán los valores del 0 % al 100 % y luego en el sentido opuesto indefinidamente, haciendo una demora bloqueante entre medio para poder observar como el LED prender lentamente y luego se apaga al mismo ritmo.

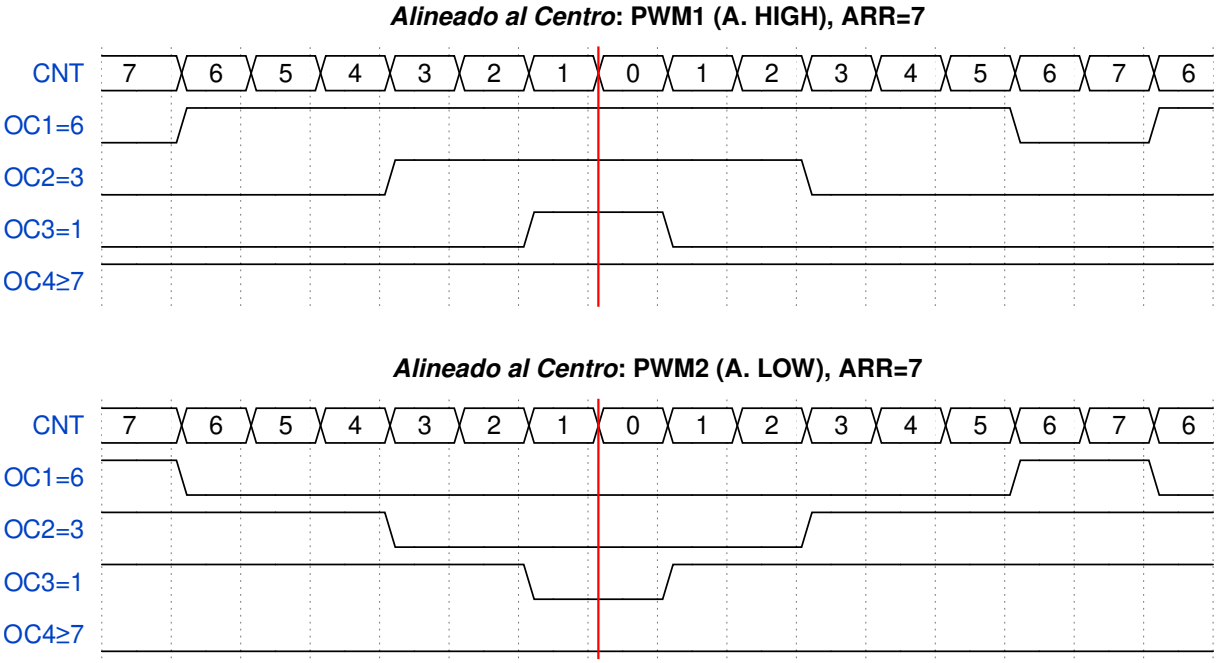
pwm_flanco.c

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/systick.h>
#include <libopencm3/cm3/nvic.h>

volatile uint32_t millis = 0;

void delay_ms(uint32_t ms);

int main(void)
{
    /* Configuración del SysClk a 72Mhz */
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    /* Se configura la salida alternativa para el TIM4_CH1 */
    rcc_periph_clock_enable(RCC_GPIOB);
    gpio_set_mode(
        GPIO_BANK_TIM4_CH1,             // GPIOB
        GPIO_MODE_OUTPUT_2_MHZ,         // Salida Max 2Mhz
        GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, // Función alternativa Push-Pull
        GPIO_TIM4_CH1);                 // GPIO6

    /* Se configura el TIM4 para una frecuencia de 1Khz */
    rcc_periph_clock_enable(RCC_TIM4);                // vvvv Tim alineado al flanco!
    timer_set_mode(TIM4, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
    timer_set_prescaler(TIM4, 72 - 1); // 72M/72 = 1Mhz
    timer_set_period(TIM4, 1000 - 1);  // 1M / 1000 => f = 1Khz (ARR=999).

    // Se configura el modo del OC1 para PWM activo en BAJO
    timer_set_oc_mode(TIM4, TIM_OC1, TIM_OCM_PWM2);
    timer_enable_oc_output(TIM4, TIM_OC1);

    // Valores iniciales del PWM:
    timer_set_oc_value(TIM4, TIM_OC1, 0);

    // Se habilita el TIM4
    timer_enable_counter(TIM4);

    // Se configura el SysTick para realizar un delay
    systick_set_frequency(1000, rcc_ahb_frequency);
    systick_interrupt_enable();
    systick_counter_enable();

    bool sentido = true;  // true: incrementa; false: decrementa
    uint16_t pwm_val = 0; // Duty para el PWM
    while (true)
    {
        // Se envía el valor del duty al canal PWM
        timer_set_oc_value(TIM4, TIM_OC1, sentido ? pwm_val : 1000 - pwm_val);
        pwm_val++;
        if (pwm_val > 1000) // Si el ciclo llegó al 100%
        {
            pwm_val = 0;        // Se pone el valor en 0
            sentido = !sentido; // se inverte el sentido
        }
        delay_ms(1); // demora para observar el efecto
    }
}

void delay_ms(uint32_t ms)
{
    uint32_t tm = ms + millis;
    while (millis < tm);
}

void sys_tick_handler(void)
{ // cada 1ms
    millis++;
}
```

Este modo de trabajo es ideal, por ejemplo, para utilizar *sevo-motores*, los cuales tiene un modo de funcionamiento muy característico y podremos tenerlos sincronizados a todos ellos.

Alineado al centro

En este modo el comportamiento es similar, pero el **TIMx** se configura con alineación al centro. Por lo tanto las señales cambian (*toggle*) cuando el **OCx** iguala al **ARR**. Pero teniendo en cuenta que esto ocurre 2 veces en un ciclo, ya que el **TIMx** cuenta para arriba y abajo. En los siguientes gráficos se ilustra el comportamiento del mismo.



Debemos considerar, que el ciclo ahora es el doble que alineado al flanco y que los *dutys* durarán consecuentemente 2 cuentas del **CNT**. Se observa el centro de la señal en el *underflow* (cuando pasa de 1 a 0). El valor máximo de la cuenta será ahora **ARR**, esto se explica en el modo centrado del **TIMx** (4.4.3).

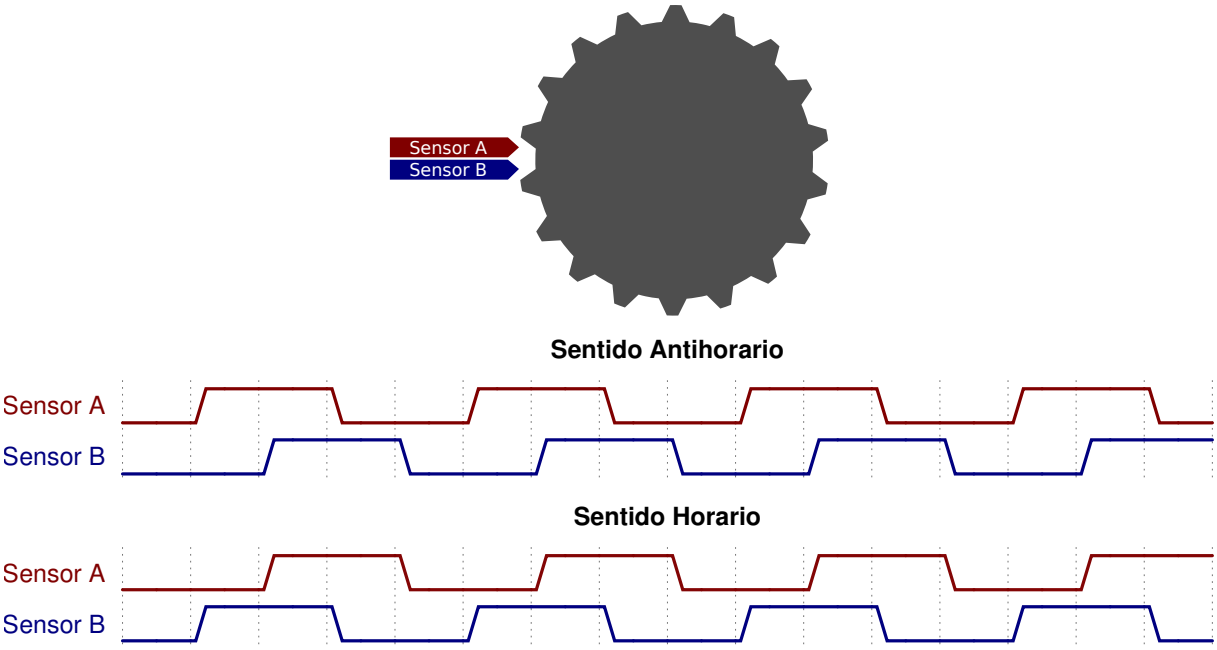
Este modo es útil para manejar motores o incluso para mejorar el consumo, es el modo más recomendado si se quiere utilizar control. Por ejemplo, si tuviéramos tiras de LEDs controladas con transistores y PWM, sería más sensato utilizar la forma centrada. De esta manera, no hay un pico de consumo al inicio del período. En contraparte, es mejor utilizar el alineado al flanco para aplicaciones de audio, ya que generan menos armónicos en el rango audible, o cuando se necesiten frecuencias más altas.

4.7. Interfaz para *encoders*

Los *encoders* de cuadratura son dispositivos digitales de posicionamiento. Son ampliamente utilizados, por ejemplo, en los desplazamientos verticales de los *mouses* (“la ruedita”), en equipos de audio, controles de vehículos, posicionamiento de

discos ópticos y magnéticos, etc. Es tan común contar estos dispositivos que el fabricante ST agregó un módulo de hardware a los **TIMx** para poder obtener su posición.

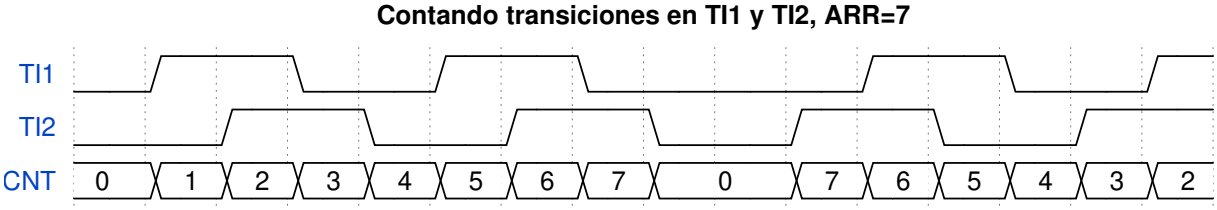
Analicemos el funcionamiento básico de un *encoder*, este parte un principio muy sencillo el cual puede ser mecánico, óptico o magnético dependiendo la situación. Pero básicamente tenemos al menos dos sensores ubicados de tal manera que nunca puedan activarse al mismo tiempo, de tal forma cuando el dispositivo gira activa uno u otro sensor con un desfazaje de manera que podamos detectar si este es en sentido horario o antihorario.



En el **TIMx** este es un modo esclavo y debemos conectar las salidas del *encoder* en las entradas **TI1** y **TI2**. Y luego tenemos 3 modos de funcionamiento:

- Contar solamente los pulsos de TI1.
- Contar solamente los pulsos de TI2.
- Contar los pulsos de TI1 y TI2.

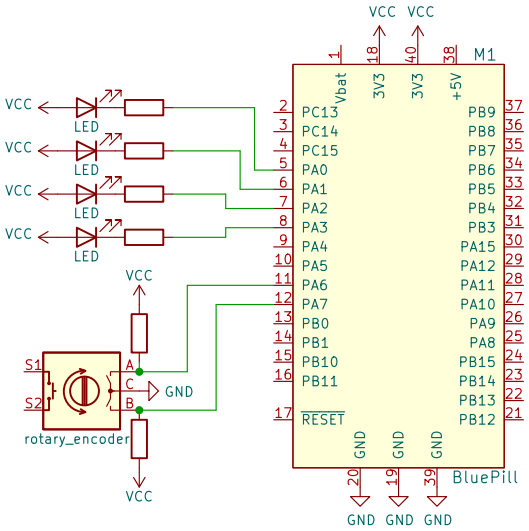
El valor en el **CNT** aumentará o disminuirá dependiendo del sentido de giro del *encoder* y contará hasta el valor de **ARR**. Esto puede generar una interrupción si se habilita la misma, en caso de ser necesario simplemente para contar, p.e., vueltas completas.



Como se indicó, el modo encoder es una configuración del modo esclavo, por lo tanto para configurarlo debemos utilizar:

```
void timer_slave_set_mode(uint32_t timer_peripheral, uint8_t mode);
// Modos para Encoder
TIM_SMCR_SMS_EM1    // Cuenta TI2 dependiendo del estado en TI1
TIM_SMCR_SMS_EM2    // Cuenta TI1 dependiendo del estado en TI2
TIM_SMCR_SMS_EM3    // Cuenta cambios en TI1 y TI2
```

En el siguiente ejemplo se utilizarán 4 LEDs conectados del `PA0` al `PA3` y un *encoder* de cuadratura mecánico muy asequible el EC11 de 20 pasos en el `PA6` (`TIM3_CH1`) el Sensor A y en `PA7` (`TIM3_CH2`) el Sensor B. El mismo hace 4 transiciones por paso, es decir que si quisiéramos contar vueltas, debemos contar hasta 80 (20×4). Pero en esta ocación simplemente utilizaremos los LEDs para contar en binario del 0 al 16 cada vez que demos un paso.



encoder_mode.c

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>

int main(void)
{
    /*        Configuración del SysClk          */
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    /*        Configuración de los LEDs         */
    rcc_periph_clock_enable(RCC_GPIOA);
    gpio_set_mode(
        GPIOA,
        GPIO_MODE_OUTPUT_2_MHZ,
        GPIO_CNF_OUTPUT_PUSHPULL,
        GPIO0 | GPIO1 | GPIO2 | GPIO3);
    // Inicialmente apagados
    gpio_set(GPIOA, GPIO0 | GPIO1 | GPIO2 | GPIO3);

    /* Configuración del TIM3 como modo Encoder */
    rcc_periph_clock_enable(RCC_TIM3);
    timer_set_period(TIM3, 16 * 4 - 1); // Cada 4 pulsos una posición
    timer_slave_set_mode(TIM3, TIM_SMCR_SMS_EM3); // Contar transisciones en TI1 y TI2
    timer_ic_set_input(TIM3, TIM_IC1, TIM_IC_IN_TI1);
    timer_ic_set_input(TIM3, TIM_IC2, TIM_IC_IN_TI2);
    timer_enable_counter(TIM3);

```

```
28      /*  Variables de control para la posición   */
29      uint16_t old_pos = 0, new_pos;
30      while (true)
31      {   // Cada 4 cuentas es una posición:
32          new_pos = timer_get_counter(TIM3) / 4;
33          if (old_pos != new_pos)
34          {                                                   // Ante un cambio:
35              gpio_set(GPIOA, GPIO0 | GPIO1 | GPIO2 | GPIO3);// Apagamos los LEDs
36              gpio_clear(GPIOA, new_pos);                     // Encendemos según la posición
37              old_pos = new_pos;                              // Guaramos la posición actual
38          }
39      }
40  }
```

El efecto que veremos es que a medida que rotemos en sentido horario el valor en binario incrementará visualmente en los LEDs hasta prenderlos todos y luego volver a cero. Contrariamente, en sentido antihorario el valor ira decrementando hasta llegar a 0 y luego hacer un *underflow* mostrando el 15 (1111).

4.8. Modo Maestro

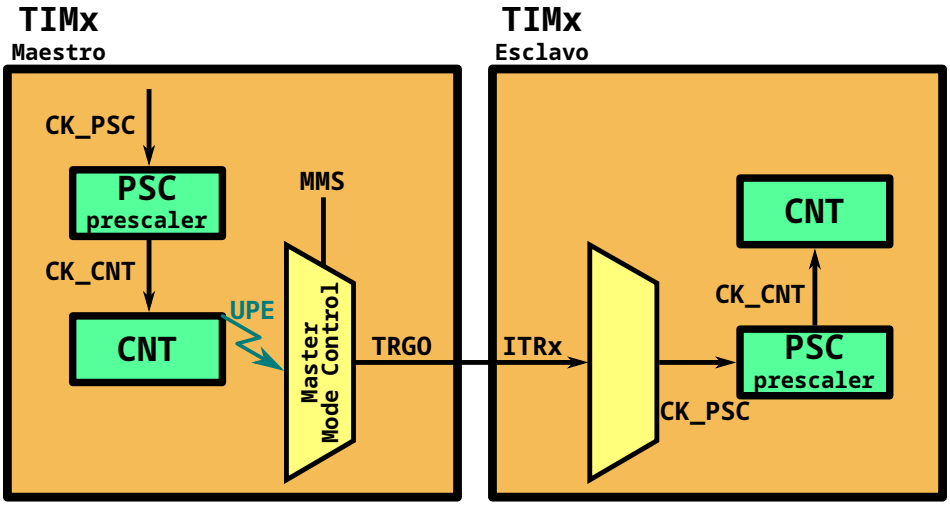
Un **TIM** en modo esclavo puede conectarse a otro en modo Maestro de diferentes formas. Puede resetear, sincronizar, habilitar, etc. Además como ya vimos en el Modo Externo 1 (4.5.3), podemos seleccionar otro **TIM** como fuente de clock, es decir utilizarlo como *prescaller* gracias a las entradas **ITRx** que están mapeadas a la salida **TRGO** de la siguiente manera:

TIM Esclavo	ITR0	ITR1	ITR2	ITR3
TIM2	TIM1	—	TIM3	TIM4
TIM3	TIM1	TIM2	—	TIM4
TIM4	TIM1	TIM2	TIM3	—

En el siguiente ejemplo utilizaremos un **TIM** como Maestro y el *Update Event* para disparar la señal **TRGO** y el **TIM** esclavo recibirá esa señal por su correspondiente **ITRx** según la tabla anterior. Para ello utilizaremos las funciones:

```
void timer_set_master_mode(uint32_t timer_peripheral, uint32_t mode);
// Defines de los posibles valores para 'mode'
TIM_CR2_MMS_RESET          // Cuando el bit UG del registro EGR se activa
TIM_CR2_MMS_ENABLE         // Cuando habilitamos el timer
TIM_CR2_MMS_UPDATE         // Cada vez que hay un Update Event
TIM_CR2_MMS_COMPARE_PULSE  // Flag de comparación CCR1IF
TIM_CR2_MMS_COMPARE_OC1REF // Output Compare Event Flag en OC1
TIM_CR2_MMS_COMPARE_OC2REF // Output Compare Event Flag en OC2
TIM_CR2_MMS_COMPARE_OC3REF // Output Compare Event Flag en OC3
TIM_CR2_MMS_COMPARE_OC4REF // Output Compare Event Flag en OC4
```

Así, si activamos las interrupciones al momento del *Update Event* en ambos **TIMs** podemos hacer, p.e., un contador binario de dos bits, poniendo dos LEDs y que estos alternen su estado en las interrupciones. El Maestro alternará el bit menos significativo (PB5) y el esclavo cada 2 **UPE** alterna el más significativo (PB6).



master-slave.c

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/cm3/systick.h>

int main(void)
{
    /* Configuración del SysClk */
    rcc_clock_setup_in_hse_8mhz_out_72mhz();

    /* Configuración de los LEDs que funcionarán para contar */
    rcc_periph_clock_enable(RCC_GPIOB);
    gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO5 | GPIO6);
    gpio_set(GPIOB, GPIO5 | GPIO6);

    /*   TIM2 - MAESTRO   */
    rcc_periph_clock_enable(RCC_TIM2);
    timer_set_mode(TIM2, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
    timer_set_prescaler(TIM2, 36000 - 1);            // 72M / 36K = 2K
    timer_set_period(TIM2, 1000 - 1);                // 2K / 1K = 2Hz => 0.5s
    timer_enable_irq(TIM2, TIM_DIER_UIE);            // Update Interrupt
    timer_set_master_mode(TIM2, TIM_CR2_MMS_UPDATE); // UPE -> TRGO

    /*   TIM3 - ESCLAVO   */
    rcc_periph_clock_enable(RCC_TIM3);
    timer_set_mode(TIM3, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
    timer_set_period(TIM3, 2 - 1);                   // Cuenta 2 UPE del TIM2
    timer_enable_irq(TIM3, TIM_DIER_UIE);            // Update Interrupt
    timer_slave_set_trigger(TIM3, TIM_SMCR_TS_ITR1); // ITR1 (TRGO del TIM2)
    timer_slave_set_mode(TIM3, TIM_SMCR_SMS_ECM1);   // Modo Externo 1

    /*   Habilitación de Interrupciones   */
    nvic_enable_irq(NVIC_TIM3_IRQ);
    nvic_enable_irq(NVIC_TIM2_IRQ);

    /*   Se habilitan los TIMs   */
    timer_enable_counter(TIM3);
    timer_enable_counter(TIM2);

    while (true)
        __asm__("nop");

}

void tim2_isr(void)
{ // Cada 0.5s
    timer_clear_flag(TIM2, TIM_SR_UIF);
    gpio_toggle(GPIOB, GPIO5);
}
```

```
51
52 void tim3_isr(void)
53 { // Cada 2 UPE del TIM2:
54     timer_clear_flag(TIM3, TIM_SR_UIF);
55     gpio_toggle(GPIOB, GPIO6);
56 }
```

Se pueden seleccionar otras funcionalidades, algunas no se mencionarán y otras que están relacionadas con otros periféricos se abarcarán en futuras unidades.

4.9. Ejercicios

EJERCICIO N°5

Conecte un *encoder* de cuadratura y un LED en una salida PWM a su elección. Utilice el *encoder* para modular el PWM de 0 % a 100 % en 5 vueltas.

EJERCICIO N°6

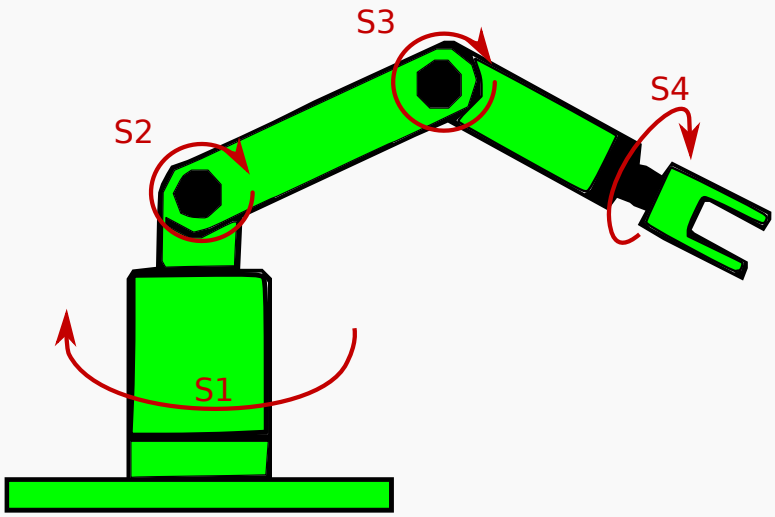
Agregue al ejercicio anterior dos displays 7 segmentos multiplexados. Cuente del 0 al 99 y −− (para representar el 100) en el display sumando o restando 1 por cada paso del *encoder*. Si el *encoder* gira "rápido" incremente las decenas en vez de las unidades.

EJERCICIO N°7

Utilizando 4 displays de 7 segmentos multiplexados programe un frecuencímetro para señales digitales lógicas. Puede conectar pulsadores para cambiar los múltiplos, recuerde que puede utilizar el punto de los displays para correr la coma de ser necesario.

EJERCICIO N°8

Un brazo robot posee 4 servo-motores, conéctelos a los canales PWM de un TIM para manejar dicho brazo utilizando dos pulsadores para cada servo.



4.10. Links y materiales de la unidad

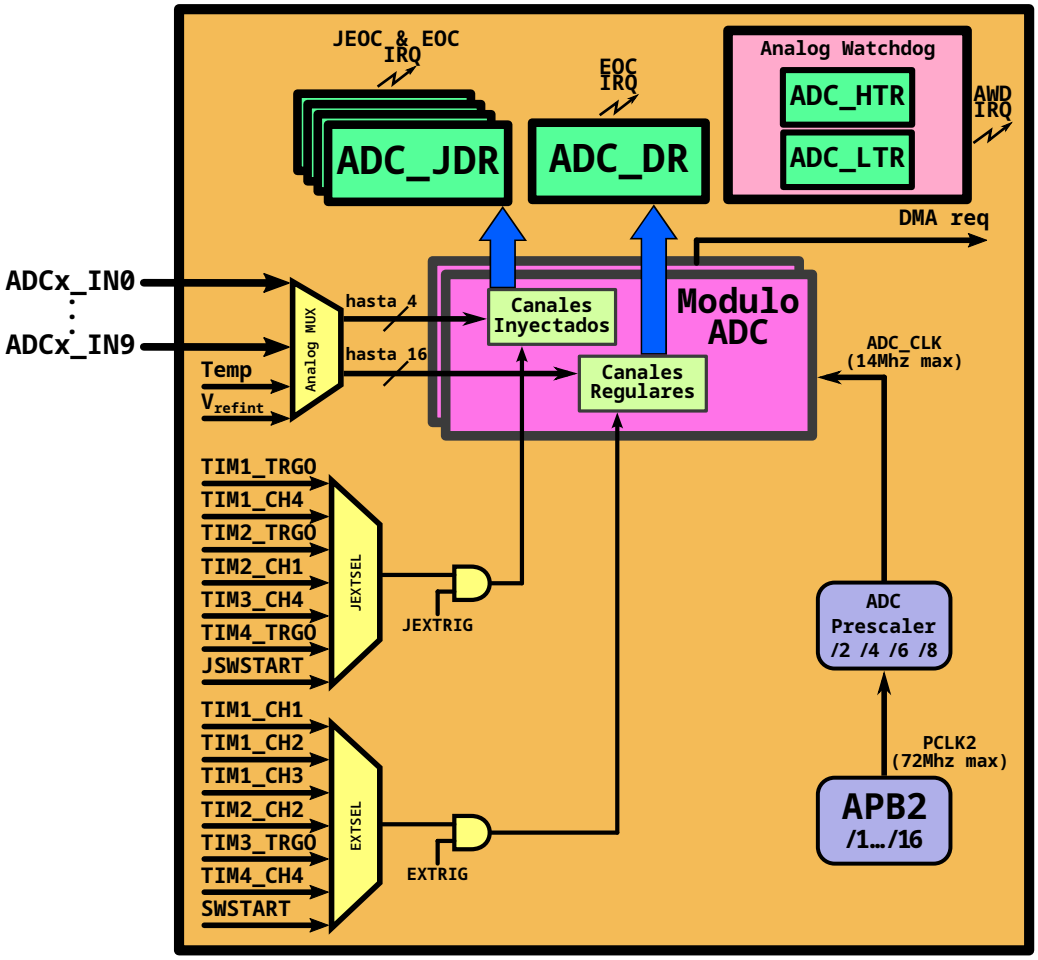
- **Documentación de libOpenCM3**
  http://libopencm3.org/docs/latest/stm32f1/html/modules.html
- **Datasheet STM32F103C8/B**
  https://www.st.com/resource/en/datasheet/stm32f103c8.pdf
- **Manual de referencia STM32F1xx**
  https://tinyurl.com/yxv9m5b2
- **¿Qué es PWM y cómo usarlo?**
  https://solectroshop.com/es/blog/que-es-pwm-y-como-usarlo–n38
- **Datasheet *encoder* EC11 11mm Size Metal Shaft Type**
  https://www.farnell.com/datasheets/1837001.pdf

Unidad 5
Conversor Analógico-Digital

5.1. Principios básicos

Los ADC (conversores analógico-digitales) son módulos que convierten una señal analógica en un valor digital. Se trata de un periférico imprescindible en cualquier microcontrolador moderno. En el caso del *STM32F103Cx*, el fabricante ha integrado dos *conversores de aproximaciones sucesivas*[1] independientes, de 12 bits de resolución. En el encapsulado de la BluePill, se dispone de 10 canales externos (desde `ADCx_IN0` hasta `ADCx_IN9`)[2], más dos canales internos: uno para medir la temperatura del silicio y otro para la tensión de referencia interna (V_{refint}).

Internamente el módulo del ADC está compuesto de varias etapas y elementos que lo convierten en un periférico muy versátil. En el siguiente diagrama simplificado, podemos observar algunas de sus características principales.



El diagrama puede leerse de izquierda a derecha siguiendo el recorrido de la señal: primero entra por uno de los canales analógicos, luego es convertida por el

[1] Este tipo de conversor utiliza una búsqueda binaria con un comparador de tensión, lo que supone una eficiencia de $O(\log n)$

[2] En otros encapsulados de la familia puede haber hasta el `ADCx_IN15`, es decir, que como máximo podremos tener 16 canales externos.

módulo **ADC** y finalmente el resultado queda disponible en los registros de datos (`ADC_DR` y/o `ADC_JDRx`) para que el programa lo lea o para que el **DMA** lo transfiera automáticamente. A continuación se detalla cada bloque.

5.1.1. Entradas analógicas (*Analog MUX*)

El primer bloque del diagrama es el **multiplexor analógico** (*Analog MUX*). Su función es seleccionar, de entre todas las fuentes disponibles, cuál señal analógica ingresará al módulo de conversión en cada momento. Las fuentes son:

- **Canales externos** (`ADCx_IN0` a `ADCx_IN9`): son los pines físicos del integrado configurados en modo analógico. En la *BluePill* hay 10 disponibles, aunque en encapsulados más grandes la familia STM32F103 puede ofrecer hasta 16 (`ADCx_IN15`).
- **Sensor de temperatura interno**: el propio silicio del *STM32F103* integra un sensor de temperatura cuya salida analógica puede medirse a través del **ADC**. Es útil para aplicaciones de monitoreo térmico, aunque su precisión absoluta es limitada.
- **Tensión de referencia interna** (V_{refint}): es una tensión interna de referencia nominal de 1,2 V. Puede aprovecharse para calibrar la lectura o para conocer la tensión de alimentación real del microcontrolador.

5.1.2. Canales regulares e inyectados

Una vez dentro del módulo **ADC**, las conversiones se organizan en dos grupos independientes, lo que otorga una gran flexibilidad al periférico.

- **Canales regulares:** son el caso de uso habitual. Se pueden encadenar hasta 16 canales en una *secuencia de conversión* y el resultado de cada conversión se almacena en el registro `ADC_DR`. Si se activa el **DMA**, cada resultado se transfiere automáticamente a memoria sin intervención del CPU, lo cual es especialmente valioso cuando se muestrean varios canales de forma continua.
- **Canales inyectados:** permiten realizar hasta 4 conversiones con alta prioridad, *interrumpiendo* una secuencia regular si fuera necesario, de manera análoga a como una **ISR** interrumpe el programa principal. Los resultados se guardan en registros propios (`ADC_JDR1` a `ADC_JDR4`), por lo que no sobreescriben los datos de los canales regulares. Son útiles, por ejemplo, para medir una señal crítica en un instante preciso disparado por un evento externo.

5.1.3. Fuentes de disparo (*triggers*)

El **ADC** no tiene por qué iniciar una conversión únicamente por software; puede ser disparado (*triggered*) por señales provenientes de otros periféricos. Esto

es clave para lograr conversiones sincronizadas con otros eventos del sistema sin ocupar al CPU.

En el diagrama se distinguen dos grupos de disparos:

- **Para canales regulares** (`EXTSEL` / `EXTRIG`): pueden provenir de varios timers (`TIM1_TRGO`, `TIM1_CH4`, `TIM2_TRGO`, `TIM2_CH1`, `TIM3_CH4`, `TIM4_TRGO`) o iniciarse por software (`SWSTART`).
- **Para canales inyectados** (`JEXTSEL` / `JEXTRIG`): cuentan con su propio conjunto de disparos (`TIM1_CH1`, `TIM1_CH2`, `TIM1_CH3`, `TIM2_CH2`, `TIM3_TRGO`, `TIM4_CH4`) y también admiten inicio por software (`JSWSTART`).

¿Por qué sincronizar el ADC con un timer?

Imaginemos que queremos muestrear una señal de audio a 8 kHz. Si dejáramos que el CPU disparara cada conversión mediante un bucle, cualquier interrupción desincronizaría el muestreo. En cambio, configurando un **TIM** para que genere un evento cada 125 µs y conectando su salida `TRGO` al **ADC**, el muestreo ocurrirá siempre en el instante correcto, independientemente de lo que esté haciendo el programa.

5.1.4. Prescaler y clock del ADC

El **ADC** tiene su propio prescaler alimentado desde el bus **APB2** (`PCLK2`, hasta 72 MHz). El divisor puede configurarse en `/2`, `/4`, `/6` u `/8` mediante la función `rcc_set_adcpre()` de libOpenCM3, y **la frecuencia resultante (`ADC_CLK`) no debe superar los 14 MHz**, ya que así lo exige el fabricante para garantizar la precisión de la conversión.

```
void rcc_set_adcpre(uint32_t adcpre);

RCC_CFGR_ADCPRE_PCLK2_DIV2 // -> PCLK2 /2
RCC_CFGR_ADCPRE_PCLK2_DIV4 // -> PCLK2 /4
RCC_CFGR_ADCPRE_PCLK2_DIV6 // -> PCLK2 /6
RCC_CFGR_ADCPRE_PCLK2_DIV8 // -> PCLK2 /8
```

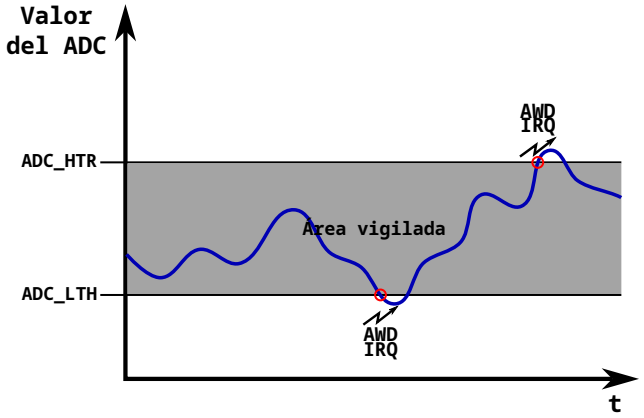
Atención

Si el `ADC_CLK` supera los 14 MHz, las conversiones pueden ser incorrectas. Con `PCLK2` = 72 MHz, el divisor mínimo válido es `/6` (12 MHz). El divisor `/4` daría 18 MHz y quedaría fuera de especificación.

5.1.5. Watchdog analógico y fuentes de interrupción

El bloque **Analog Watchdog** (`AWD`) permite vigilar que la señal convertida se mantenga dentro de un rango definido por dos registros: `ADC_HTR` (*High Threshold Register*) y `ADC_LTR` (*Low Threshold Register*). Si el valor convertido cae fuera de la ventana [`LTR`, `HTR`], se genera una interrupción (`AWD IRQ`). Esto resulta muy práctico

para detectar condiciones fuera de rango sin necesidad de que el programa compruebe cada muestra manualmente.



El diagrama en bloques inicial muestra en total tres fuentes de interrupción provenientes del **ADC**:

- `EOC IRQ` (*End Of Conversion*): se activa al completarse la conversión de un canal regular.
- `JEOC IRQ` (*Injected End Of Conversion*): ídem para canales inyectados, al mismo tiempo se dispara una `EOC IRQ`.
- `AWD IRQ`: activada por el *Analog Watchdog* al detectar una muestra fuera del rango configurado.

5.1.6. Registros de resultado

El resultado de cada conversión es un valor entero de hasta 12 bits que queda almacenado en:

- `ADC_DR` (*Data Register*): contiene el resultado de los canales regulares. Se puede leer mediante `adc_read_regular(ADC1|ADC2)`. En modo de conversión continua de múltiples canales, los resultados se sobreescriben a medida que llegan, por lo que es fundamental usar el **DMA** para no perder datos.
- `ADC_JDR1`–`ADC_JDR4`: contienen los resultados de los canales inyectados, uno por canal, por lo que no existe riesgo de sobreescritura. Y son accedidos a través de la función `adc_read_injected(ADC1|ADC2, 1..4)` de **libOpenCM3**.

Dada la complejidad de este módulo y que su mayor potencialidad está dada en conjunto con el **DMA**. Solo se expondrá la parte más básica de su funcionamiento, y en la unidad dedicada al **DMA** se retomará para completar las posibilidades que ofrece este periférico.

5.2. Modos de funcionamiento

El módulo de **ADC** de los *STM32* es muy potente y, por ende, complejo. Antes de escribir una sola línea de código conviene tener claro qué decisiones hay que tomar para configurarlo correctamente. Estas decisiones se organizan en tres ejes independientes: el **modo de operación**, el **tipo de conversión** y la **fuente de disparo**.

5.2.1. Modo de operación

El primer eje define la relación entre los dos **ADC** disponibles en el *STM32F103*:

- **Modo independiente:** es la manera más habitual. Cada **ADC** funciona por su cuenta, convirtiendo los canales de su propia secuencia sin ninguna coordinación con el otro. Es el punto de partida natural y el que desarrollaremos en esta unidad.
- **Modo dual:** ambos **ADC** trabajan en conjunto, coordinados por hardware, para realizar conversiones simultáneas o intercaladas. Se configura con la función `adc_set_dual_mode()`. Esto permite, por ejemplo, muestrear dos señales exactamente al mismo tiempo o duplicar la tasa de conversión efectiva. Dado que su mayor utilidad aparece en combinación con el **DMA**, quedará para una unidad posterior.

5.2.2. Tipo de conversión

Una vez elegido el modo de operación, hay que decidir *cómo* se realizarán las conversiones dentro de la secuencia de canales configurada:

- **Conversión única (*single*):** se convierte un solo canal o secuencia (depende del modo de escaneo), una sola vez, cada vez que se produce un disparo. Se activa con `adc_set_single_conversion_mode(ADC1|ADC2)`. Es el modo más sencillo y el más adecuado para empezar.
- **Conversión continua (*continuous*):** al terminar la secuencia completa, el **ADC** vuelve a empezar automáticamente sin necesidad de un nuevo disparo. Se activa con `adc_set_continuous_conversion_mode(ADC1|ADC2)`. Resulta útil cuando se necesita tener siempre disponible el valor más reciente de una señal.
- **Conversión por escaneo (*scan*):** recorre automáticamente todos los canales de la secuencia configurada, uno tras otro, en el orden definido. Se activa con `adc_enable_scan_mode(ADC1|ADC2)`. Puede combinarse con el modo continuo para un muestreo permanente de múltiples canales.
- **Conversión discontinua (*discontinuous*):** divide la secuencia en subgrupos y convierte un subgrupo por disparo. Para canales regulares se activa con `adc_enable_discontinuous_mode_regular(ADC1|ADC2, 1..8)`, donde el segundo parámetro indica el tamaño del subgrupo (máximo 8 canales). Es útil cuando se necesita espaciar las conversiones de un grupo de canales.

Estos modos pueden combinarse entre sí, con una única excepción: los modos **conversión única** y **conversión continua** son mutuamente excluyentes, ya que se controlan mediante el bit `CONT` del registro `ADC_CR2`. El modo de escaneo y el modo discontinuo pueden activarse o desactivarse de forma independiente sobre cualquiera de los dos modos principales.

A modo de resumen, la siguiente tabla (*machete*) muestra el comportamiento del **ADC** según la combinación de bits:

1. **Repetición de la conversión**
   - Modo conversión única (`CONT = 0`):
     - El ADC no se reinicia solo.
     - Necesita un disparo (*trigger*) cada vez (externo o `SWSTART`).
   - Modo conversión continua (`CONT = 1`):
     - El ADC se auto-dispara internamente.
     - Un solo disparo inicial es suficiente.
2. **Recorrido dentro del disparo**
   - Sin escaneo (`SCAN = 0`): solo se convierte el primer canal de la secuencia.
   - Con escaneo (`SCAN = 1`): se recorren todos los canales de la secuencia.
3. **Modo discontinuo**
   - Sin subgrupos (`DISC = 0`): la secuencia se recorre de forma normal (canal por canal o todos según el escaneo).
   - Con subgrupos (`DISC = 1`, `DISCONT = n`, con n entre 1 y 8): cada disparo convierte un subgrupo de n canales. Una vez finalizados todos los subgrupos, el siguiente disparo vuelve a empezar desde el subgrupo inicial.

5.2.3. Fuente de disparo

Por último, hay que definir *cuándo* arranca cada conversión:

- **Disparo por software (*software trigger*):** la conversión se inicia explícitamente desde el programa. Es la opción más directa y la que utilizaremos en los primeros ejemplos y dependiendo la configuración se puede hacer con `adc_start_conversion_direct(ADC1|ADC2)` o con `adc_start_conversion_regular(ADC1|ADC2)`[3].
- **Disparo por hardware (*hardware trigger*):** la conversión arranca automáticamente ante un evento externo, como el desbordamiento de un **TIM** o una señal en un pin **EXTI**. Como se describió en el diagrama, los canales regulares e inyectados tienen cada uno su propio conjunto de fuentes de disparo. Esta modalidad es la que permite sincronizar el muestreo con otros periféricos sin intervención del **CPU**.

[3] Esto se cubrirá con más detalle en la siguiente subsección.

```
void adc_enable_external_trigger_regular(uint32_t adc, uint32_t trigger);
// Fuentes de disparo para el ADC en modo regular:
ADC_CR2_EXTSEL_TIM1_CC1    // trigger -> TIM1_CH1
ADC_CR2_EXTSEL_TIM1_CC2    // trigger -> TIM1_CH2
ADC_CR2_EXTSEL_TIM1_CC3    // trigger -> TIM1_CH3
ADC_CR2_EXTSEL_TIM2_CC2    // trigger -> TIM2_CH2
ADC_CR2_EXTSEL_TIM3_TRGO   // trigger -> TIM3_TRGO
ADC_CR2_EXTSEL_TIM4_CC4    // trigger -> TIM4_CH4
ADC_CR2_EXTSEL_SWSTART     // trigger -> por Software

void adc_enable_external_trigger_injected(uint32_t adc, uint32_t trigger);
// Fuentes de disparo para el ADC en modo inyectado:
ADC_CR2_JEXTSEL_TIM1_TRGO  // trigger -> TIM1_TRGO
ADC_CR2_JEXTSEL_TIM1_CC4   // trigger -> TIM1_CH4
ADC_CR2_JEXTSEL_TIM2_TRGO  // trigger -> TIM2_TRGO
ADC_CR2_JEXTSEL_TIM2_CC1   // trigger -> TIM2_CH1
ADC_CR2_JEXTSEL_TIM3_CC4   // trigger -> TIM3_CH4
ADC_CR2_JEXTSEL_TIM4_TRGO  // trigger -> TIM4_TRGO
ADC_CR2_JEXTSEL_JSWSTART   // trigger -> por Software
```

Combinando los tres ejes

La configuración final del **ADC** surge de elegir una opción en cada eje. Por ejemplo: *modo independiente* + *conversión única* + *disparo por software* es la combinación más simple posible y la que implementaremos a continuación. Más adelante, *modo independiente* + *escaneo continuo* + *disparo por timer* + **DMA** representará el caso de uso más potente para adquisición de datos.

5.2.4. Tiempo de muestreo

Hay un último parámetro que merece mención antes de pasar al código: el **tiempo de muestreo** (*sample time*). Antes de que el conversor realice la aproximación sucesiva, el capacitor interno de *sample-and-hold* debe cargarse lo suficiente como para representar fielmente la tensión de entrada. El tiempo necesario depende de la impedancia de la fuente conectada al pin: cuanto mayor sea esa impedancia, más tiempo se necesita.

El *STM32F103* permite configurar el tiempo de muestreo canal por canal, con los siguientes valores expresados en ciclos de ADC_CLK: 1,5 – 7,5 – 13,5 – 28,5 – 41,5 – 55,5 – 71,5 – 239,5 ciclos. A estos se suma siempre un tiempo fijo de 12,5 ciclos para la conversión propiamente dicha. Así, la conversión más rápida posible toma $1{,}5 + 12{,}5 = 14$ ciclos, lo que con un ADC_CLK de 14 MHz resulta en exactamente 1 µs por muestra (1 Msps). Para esto, **libOpenCM3** nos ofrece la siguientes funciones y macros.

```
void adc_set_sample_time_on_all_channels(uint32_t adc, uint8_t time);
void adc_set_sample_time(uint32_t adc, uint8_t channel, uint8_t time);
// Defines para el parámetro time:
ADC_SMPR_SMP_1DOT5CYC   // 1,5 ciclos
ADC_SMPR_SMP_7DOT5CYC   // 7.5 ciclos
ADC_SMPR_SMP_13DOT5CYC  // 13.5 ciclos
ADC_SMPR_SMP_28DOT5CYC  // 28.5 ciclos
ADC_SMPR_SMP_41DOT5CYC  // 41.5 ciclos
ADC_SMPR_SMP_55DOT5CYC  // 55.5 ciclos
ADC_SMPR_SMP_71DOT5CYC  // 71.5 ciclos
ADC_SMPR_SMP_239DOT5CYC // 239.5 ciclos
```

Atención

Para los canales internos (sensor de temperatura y *Vrefint*) el fabricante exige un tiempo de muestreo mínimo de 17,1 µs, lo que equivale a seleccionar al menos 239,5 ciclos con `ADC_CLK = 14 MHz`. Utilizar un tiempo menor producirá lecturas erróneas en esos canales.

5.3. Programación del ADC

Una vez comprendidos los conceptos básicos y las diferentes opciones de configuración, es momento de escribir código. A continuación se presentan dos ejemplos progresivos que muestran cómo utilizar el **ADC1** en modo independiente, con disparo por software y conversión única. El primero es una lectura bloqueante, ideal para entender el flujo de trabajo. El segundo incorpora interrupciones, lo que permite aprovechar el **CPU** mientras el ADC trabaja.

En todos los casos se debe seguir una secuencia de inicialización común que incluye: habilitación de relojes, configuración del pin analógico, ajuste del prescaler, configuración del **ADC**, calibración y definición de la secuencia de canales regulares.

5.3.1. Calibración

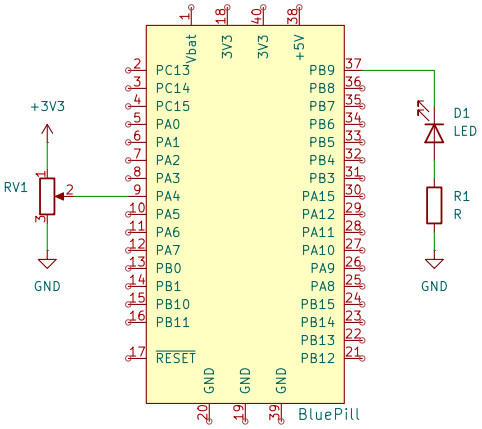
El fabricante recomienda calibrar el **ADC** después de cada reset. La calibración ajusta internamente los condensadores de muestreo para compensar variaciones de proceso y temperatura. El procedimiento es obligatorio si se desea obtener lecturas precisas; en libOpenCM3 se implementa de la siguiente manera:

```
    adc_power_off(ADC1);           // apagar antes de resetear
    rcc_periph_reset_pulse(RST_ADC1);
    rcc_set_adcpre(RCC_CFGR_ADCPRE_PCLK2_DIV6);
    // ... configuración de modo, canales, sample time ...
    adc_power_on(ADC1);            // encender
    adc_reset_calibration(ADC1);   // resetear registros de calibración
    adc_calibrate(ADC1);           // calibrar (bloqueante)
```

5.3.2. Conversiones sin escaneo (1 canal a la vez)

A continuación se presentan algunos ejemplos prácticos. Los dos primeros controlan la intensidad de un LED (`PB9`, `TIM4_CH4`) mediante un potenciómetro en `PA4` (`ADC_CHANNEL4`). El tercero agrega un segundo potenciómetro y un segundo LED para mostrar la lectura de múltiples canales sin **DMA**. El esquema de conexiones del primer circuito se muestra a continuación.

Lectura bloqueante, conversión única disparada por software



adc_to_pwm.c

```c
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/cm3/systick.h>
#include <libopencm3/stm32/adc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/timer.h>

#define MAX_COUNT (4096)    // cuenta máxima para el PWM (12bits)
volatile uint32_t millis = 0;
void config_adc(void);
void config_pwm(void);
void delay_ms(uint32_t ms);

int main(void)
{   /* Configuración del clock principal */
    rcc_clock_setup_in_hse_8mhz_out_72mhz();
    /* Configuración del ADC */
    config_adc();
    /* Configuración PWM */
    config_pwm();
    // Inicialización del SysTick para el delay_ms()
    systick_set_frequency(1000, rcc_ahb_frequency);
    systick_interrupt_enable();
    systick_counter_enable();
    uint16_t lastadc = 0;
    while (true)
    {   // Inicia la conversión
        adc_start_conversion_regular(ADC1);
        // Esperamos a que termine
        while (!adc_get_flag(ADC1, ADC_SR_EOC));
        // Ponemos el valor leido en el canal PWM
        uint16_t adcdata = adc_read_regular(ADC1);
        adcdata = (lastadc * 7 + adcdata) / 8;
        lastadc = adcdata;
        timer_set_oc_value(TIM4, TIM_OC4, adcdata);
        delay_ms(5);
    }
}
void config_adc(void)
{   // Se habilita el Clock del periférico
    rcc_periph_clock_enable(RCC_ADC1);
    adc_power_off(ADC1); // apagar antes de resetear
    rcc_periph_reset_pulse(RST_ADC1);
    // 72Mhz/6 = 12Mhz (14Mhz es el máximo permitido)
    rcc_set_adcpre(RCC_CFGR_ADCPRE_PCLK2_DIV6);
    // Se utilizará el CH04 del ADC, que es PA4
    rcc_periph_clock_enable(RCC_GPIOA);
    gpio_set_mode(
        GPIOA,                  // Canales del 0-7 GPIOA; 8 (PB0) y 9 (PB1)
        GPIO_MODE_INPUT,        // Modo Entrda
        GPIO_CNF_INPUT_ANALOG,  // Canal Analógico
        GPIO4);                 // GPIO4 (CHANNEL 4)

```

```
54      // configurar el trigger por software:
55      adc_enable_external_trigger_regular(ADC1, ADC_CR2_EXTSEL_SWSTART);
56      //modo de conversión única
57      adc_set_single_conversion_mode(ADC1);
58      // Se configura el canal a leer en el sequencer (como single conversion)
59      uint8_t ch[] = {ADC_CHANNEL4};
60      adc_set_regular_sequence(ADC1, 1, ch);
61      // Sampleo por default: 1.5cyclos
62      adc_set_sample_time(ADC1, ADC_CHANNEL4, ADC_SMPR_SMP_1DOT5CYC);
63      // Calibración:
64      adc_power_on(ADC1);          // encender
65      adc_reset_calibration(ADC1); // resetear registros de calibración
66      adc_calibrate(ADC1);         // calibrar (bloqueante)
67  }
68
69  void config_pwm(void)
70  {
71      // Se utilizará el PWM en el CH4 del TIM4 (PB9)
72      rcc_periph_clock_enable(RCC_GPIOB);
73      gpio_set_mode(GPIO_BANK_TIM4_CH4,
74                    GPIO_MODE_OUTPUT_2_MHZ,
75                    GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
76                    GPIO9);
77      rcc_periph_clock_enable(RCC_TIM4);
78      timer_set_mode(TIM4, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
79      timer_set_period(TIM4, MAX_COUNT - 1); // 72Mhz/4096= 17.578125Khz
80      timer_set_oc_mode(TIM4, TIM_OC4, TIM_OCM_PWM2);
81      timer_enable_oc_output(TIM4, TIM_OC4);
82      timer_enable_counter(TIM4);
83  }
84
85  void delay_ms(uint32_t ms)
86  {
87      uint32_t tm = ms + millis;
88      while (millis < tm);
89  }
90
91  void sys_tick_handler(void)
92  { // cada 1ms
93      millis++;
94  }
```

La configuración del **ADC** está en la función `config_adc()`, entre las *líneas 40 y 67*. Lo primero, como siempre, es habilitar el clock del periférico con la función del **RCC** `rcc_periph_clock_enable(RCC_ADC1)`. A partir de ahí es aconsejable seguir la secuencia recomendada por el fabricante:

1. Apagar el ADC con `adc_power_off(ADC1)`.
2. Resetear el periférico con `rcc_periph_reset_pulse(RST_ADC1)`.
3. Establecer el prescaler con `rcc_set_adcpre()`. Con `PCLK2` a 72 MHz, el divisor mínimo válido es `/6` (`RCC_CFGR_ADCPRE_PCLK2_DIV6`), que entrega 12 MHz, dentro del límite de 14 MHz exigido por el fabricante.
4. Configurar el pin analógico con `gpio_set_mode()` usando `GPIO_CNF_INPUT_ANALOG`. En este caso, `PA4` corresponde al `ADC_CHANNEL4`.
5. Seleccionar el modo de conversión. Aquí usamos conversión única con la función `adc_set_single_conversion_mode(ADC1)`, que activa el flag `EOC` al finalizar cada conversión.

6. Configurar la fuente de disparo. Aunque en este ejemplo el disparo es por software, el ADC del STM32F1 igualmente requiere indicar el origen explícitamente con `adc_enable_external_trigger_regular(ADC1, ADC_CR2_EXTSEL_SWSTART)`.

7. Definir la secuencia de canales. El ADC siempre opera sobre un arreglo de canales, aunque solo haya uno. Se declara el arreglo y se pasa a la función `adc_set_regular_sequence()`:

```
    uint8_t ch[] = {ADC_CHANNEL4};
    adc_set_regular_sequence(ADC1, 1, ch);
```

8. Configurar el tiempo de muestreo por canal con `adc_set_sample_time()`, utilizando alguno de los valores de la Subsubsección 5.2.4.

9. Finalmente, encender el ADC y calibrarlo. La calibración es bloqueante: el programa espera hasta que el hardware confirme que finalizó.

```
    adc_power_on(ADC1);          // encender
    adc_reset_calibration(ADC1); // resetear registros de calibración
    adc_calibrate(ADC1);         // calibrar (bloqueante)
```

Una vez configurado, el bucle principal inicia cada conversión con la función `adc_start_conversion_regular(ADC1)`, espera a que el flag `ADC_SR_EOC` se active mediante `adc_get_flag()` y luego lee el resultado con `adc_read_regular(ADC1)`. Como aclaración adicional, se aplica un filtro pasa bajos de primer orden por software en la *línea 35* para estabilizar los bits menos significativos. Es un promedio ponderado sencillo que no tiene relación directa con el módulo ADC pero es útil en la práctica.

Disparo directo del ADC

En el ejemplo anterior se configuró el disparo por software mediante `adc_enable_external_trigger_regular()` junto con `ADC_CR2_EXTSEL_SWSTART`. Otra alternativa hubiera sido usar `adc_start_conversion_direct()`, que inicia la conversión escribiendo directamente en el bit `ADON` del registro `ADC_CR2`. Se eligió el método del trigger externo por software porque es compatible con otras familias **STM32F** y unifica la forma de configurar el **ADC** en todos los modos.

Lectura con interrupciones, conversión única disparada por hardware

adc_to_pwm_isr.c

```
1  #include <libopencm3/cm3/nvic.h>
2  #include <libopencm3/cm3/systick.h>
3  #include <libopencm3/stm32/adc.h>
4  #include <libopencm3/stm32/gpio.h>
5  #include <libopencm3/stm32/rcc.h>
6  #include <libopencm3/stm32/timer.h>
7
8  #define MAX_COUNT (4096)
9  void config_adc(void);
10 void config_pwm(void);
11 void config_tim_trig(void);
12
13 int main(void)
14 {
15     /* Configuración del clock principal */
16     rcc_clock_setup_in_hse_8mhz_out_72mhz();
```

```
17      /* Configuración del ADC */
18      config_adc();
19      /* Configuración PWM */
20      config_pwm();
21      while (true);
22  }
23
24  void config_adc(void)
25  {
26      // Se configura el prescaler del ADC:
27      rcc_periph_clock_enable(RCC_ADC1);
28      adc_power_off(ADC1); // apagar antes de resetear
29      rcc_periph_reset_pulse(RST_ADC1);
30      // 72Mhz/6 = 12Mhz (14Mhz es el máximo permitido)
31      rcc_set_adcpre(RCC_CFGR_ADCPRE_PCLK2_DIV6);
32      // Se utilizará el CH04 del ADC, que es PA4
33      rcc_periph_clock_enable(RCC_GPIOA);
34      gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO4);
35      // configurar el trigger por el TRGO del TIM3 (cada 5ms):
36      adc_enable_external_trigger_regular(ADC1, ADC_CR2_EXTSEL_TIM3_TRGO);
37      //modo de conversión única
38      adc_set_single_conversion_mode(ADC1);
39      // Se configura el canal a leer en el sequencer (como single conversion)
40      uint8_t ch[] = {ADC_CHANNEL4};
41      adc_set_regular_sequence(ADC1, 1, ch);
42      // Sampleo por default: 1.5cyclos
43      adc_set_sample_time(ADC1, ADC_CHANNEL4, ADC_SMPR_SMP_239DOT5CYC);
44      // Calibración:
45      adc_power_on(ADC1);               // encender
46      adc_reset_calibration(ADC1);      // resetear registros de calibración
47      adc_calibrate(ADC1);              // calibrar (bloqueante)
48      adc_enable_eoc_interrupt(ADC1);   // habilitación del End-of-Conversion
49      nvic_enable_irq(NVIC_ADC1_2_IRQ); // habilita la ISR
50      config_tim_trig(); // habilitación del TIM3 para disparar el ADC1
51  }
52
53  void config_pwm(void)
54  {
55      // Se utilizará el PWM en el CH4 del TIM4 (PB9)
56      rcc_periph_clock_enable(RCC_GPIOB);
57      gpio_set_mode(GPIO_BANK_TIM4_CH4, GPIO_MODE_OUTPUT_2_MHZ,
58                    GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO9);
59      rcc_periph_clock_enable(RCC_TIM4);
60      timer_set_mode(TIM4, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
61      timer_set_period(TIM4, MAX_COUNT - 1); // 72Mhz/4096= 17.578125Khz
62      timer_set_oc_mode(TIM4, TIM_OC4, TIM_OCM_PWM2);
63      timer_enable_oc_output(TIM4, TIM_OC4);
64      timer_enable_counter(TIM4);
65  }
66
67  void config_tim_trig(void)
68  {
69      rcc_periph_clock_enable(RCC_TIM3);
70      timer_set_mode(TIM3, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
71      timer_set_prescaler(TIM3, 359); // 72MHz/360 = 200KHz
72      timer_set_period(TIM3, 999);    // 200KHz / 1000 = 200Hz => T = 5ms
73      timer_set_master_mode(TIM3, TIM_CR2_MMS_UPDATE); // TRGO en cada update
74      timer_enable_counter(TIM3);                      // Arranca el timer
75  }
76
77  void adc1_2_isr(void)
78  {
79      // virificamos si fue el ADC1 o ADC2 (en este ejemplo no es necesario)
80      if (adc_get_flag(ADC1, ADC_SR_EOC)) {
81        static uint16_t lastadc = 0;
82        uint16_t adcdata = adc_read_regular(ADC1);
83        adcdata = (lastadc * 7 + adcdata) / 8;
84        lastadc = adcdata;
85        timer_set_oc_value(TIM4, TIM_OC4, adcdata);
86      }
87  }
```

En este ejemplo se reemplaza la espera bloqueante por una interrupción de fin de conversión, y el disparo por software por un disparo periódico generado por el **TIM3** en modo maestro. De esta manera el **CPU** queda libre entre conversiones. Los cambios con respecto al ejemplo anterior son tres:

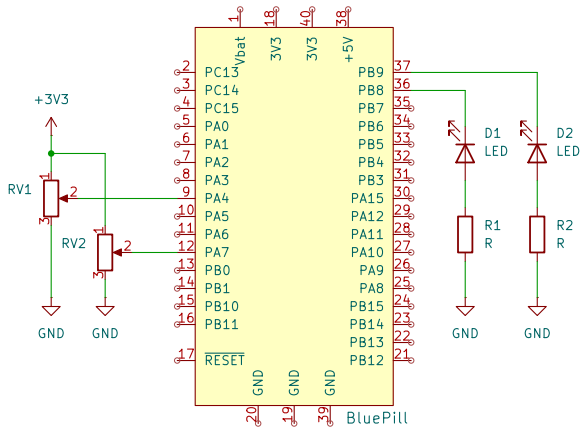
- **Fuente de disparo** (*línea 36*): se cambia `ADC_CR2_EXTSEL_SWSTART` por la siguiente fuente: `ADC_CR2_EXTSEL_TIM3_TRGO`. De manera que el **TIM3** dispara una conversión cada 5 ms a través de su salida **TRGO**.
- **Habilitación de la interrupción** (*línea 48*): se agrega `adc_enable_eoc_interrupt(ADC1)` y se habilita la **IRQ** en el **NVIC** con `nvic_enable_irq(NVIC_ADC1_2_IRQ)`.
- **Configuración del TIM3** (*líneas 69–74*): se inicializa el timer en modo maestro con `timer_set_master_mode(TIM3, TIM_CR2_MMS_UPDATE)`, de manera que genera un pulso **TRGO** en cada *Update Event* (Subsección 4.8).

La **ISR** `adc1_2_isr()` es compartida por el **ADC1** y el **ADC2**. Por eso, aunque en este ejemplo solo se usa el **ADC1**, se muestra el uso de `adc_get_flag(ADC1, ADC_SR_EOC)` para verificar que fue efectivamente el **ADC1** quien generó la interrupción. La lectura del registro `ADC_DR` mediante `adc_read_regular()` baja automáticamente el flag `EOC`, por lo que no es necesario llamar a `adc_clear_flag()` en este caso.

Lectura de más de un canal sin DMA

Como se mencionó en varias oportunidades, el **ADC** está especialmente pensado para trabajar junto con el **DMA** cuando se necesita leer múltiples canales (*scan mode*). Sin embargo, es posible hacerlo sin él utilizando el **modo discontinuo**: en lugar de convertir todos los canales de la secuencia de una vez, el ADC convierte un subgrupo por cada disparo. Así, cada **IRQ** entrega exactamente un resultado, y el programa puede identificar a qué canal corresponde por orden de llegada.

En el siguiente circuito se agrega un segundo potenciómetro en `PA7` (`ADC_CHANNEL7`) y un segundo LED en `PB8` (`TIM4_CH3`).



adc_discontinus_mode.c

```
1  #include <libopencm3/cm3/nvic.h>
2  #include <libopencm3/cm3/systick.h>
3  #include <libopencm3/stm32/adc.h>
```

```
4  #include <libopencm3/stm32/gpio.h>
5  #include <libopencm3/stm32/rcc.h>
6  #include <libopencm3/stm32/timer.h>
7
8  #define MAX_COUNT (4096)
9  void config_adc(void);
10 void config_pwm(void);
11 void config_tim_trig(void);
12
13 int main(void)
14 {
15     /* Configuración del clock principal */
16     rcc_clock_setup_in_hse_8mhz_out_72mhz();
17     /* Configuración del ADC */
18     config_adc();
19     /* Configuración PWM */
20     config_pwm();
21     while (true);
22 }
23
24 void config_adc(void)
25 {
26     // Se configura el prescaler del ADC:
27     rcc_periph_clock_enable(RCC_ADC1);
28     adc_power_off(ADC1); // apagar antes de resetear
29     rcc_periph_reset_pulse(RST_ADC1);
30     // 72Mhz/6 = 12Mhz (14Mhz es el máximo permitido)
31     rcc_set_adcpre(RCC_CFGR_ADCPRE_PCLK2_DIV6);
32     // Se utilizará el CH04 y CH7 del ADC, que es PA4 y PA7
33     rcc_periph_clock_enable(RCC_GPIOA);
34     gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO4 | GPIO7);
35     // configurar el trigger por el TRGO del TIM3 (cada 2.5ms):
36     adc_enable_external_trigger_regular(ADC1, ADC_CR2_EXTSEL_TIM3_TRGO);
37     // modo de conversión única
38     adc_set_single_conversion_mode(ADC1);
39     // modo discontinuo con subgrupos de 1 canal:
40     adc_enable_discontinuous_mode_regular(ADC1, 1);
41     uint8_t ch[] = {ADC_CHANNEL4, ADC_CHANNEL7};
42     adc_set_regular_sequence(ADC1, 2, ch);
43     // Sampleo por default: 1.5cyclos (por canal)
44     adc_set_sample_time(ADC1, ADC_CHANNEL4, ADC_SMPR_SMP_1DOT5CYC);
45     adc_set_sample_time(ADC1, ADC_CHANNEL7, ADC_SMPR_SMP_1DOT5CYC);
46     // Calibración:
47     adc_power_on(ADC1);                // encender
48     adc_reset_calibration(ADC1);       // resetear registros de calibración
49     adc_calibrate(ADC1);               // calibrar (bloqueante)
50     adc_enable_eoc_interrupt(ADC1);    // habilitación del End-of-Conversion
51     nvic_enable_irq(NVIC_ADC1_2_IRQ); // habilita la ISR
52     config_tim_trig(); // habilitación del TIM3 para disparar el ADC1
53 }
54
55 void config_pwm(void)
56 {
57     // Se utilizará el PWM en el CH3 y CH4 del TIM4 (PB8 y PB9)
58     rcc_periph_clock_enable(RCC_GPIOB);
59     gpio_set_mode(GPIO_BANK_TIM4_CH4, GPIO_MODE_OUTPUT_2_MHZ,
60                   GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO8 | GPIO9);
61     rcc_periph_clock_enable(RCC_TIM4);
62     timer_set_mode(TIM4, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
63     timer_set_period(TIM4, MAX_COUNT - 1); // 72Mhz/4096= 17.578125Khz
64     timer_set_oc_mode(TIM4, TIM_OC3, TIM_OCM_PWM2);
65     timer_enable_oc_output(TIM4, TIM_OC3);
66     timer_set_oc_mode(TIM4, TIM_OC4, TIM_OCM_PWM2);
67     timer_enable_oc_output(TIM4, TIM_OC4);
68     timer_enable_counter(TIM4);
69 }
70
71 void config_tim_trig(void)
72 {
73     rcc_periph_clock_enable(RCC_TIM3);
74     timer_set_mode(TIM3, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
75     timer_set_prescaler(TIM3, 359); // 72MHz/360 = 200KHz
76     timer_set_period(TIM3, 499);    // 200KHz/500 => T = 2.5ms (5ms entre canales)
```

```
77      timer_set_master_mode(TIM3, TIM_CR2_MMS_UPDATE); // TRGO en cada update
78      timer_enable_counter(TIM3);                      // Arranca el timer
79  }
80
81  void adc1_2_isr(void)
82  {   // virificamos si fue el ADC1 o ADC2 (en este ejemplo no es necesario)
83      if (adc_get_flag(ADC1, ADC_SR_EOC)) {
84          static uint16_t lastadc[2] = {0, 0};
85          static uint8_t i_ch = 0;
86          uint16_t adcdata = adc_read_regular(ADC1);
87          adcdata = (lastadc[i_ch] * 7 + adcdata) / 8;
88          lastadc[i_ch] = adcdata;
89          timer_set_oc_value(TIM4, (i_ch == 0) ? TIM_OC4 : TIM_OC3, adcdata);
90          if (++i_ch >= 2) i_ch = 0;
91      }
92  }
```

Los cambios clave respecto al ejemplo anterior son los siguientes. En la *línea 34* se agrega `GPIO7` al pin analógico. En la *línea 40* se activa el modo discontinuo con subgrupos de un canal: `adc_enable_discontinuous_mode_regular(ADC1, 1)`; esto hace que cada disparo del **TIM3** convierta solo el siguiente canal de la secuencia, alternando entre `ADC_CHANNEL4` y `ADC_CHANNEL7` (*líneas 41–42*). El tiempo de muestreo se configura por separado para cada canal (*líneas 44–45*).

Para mantener una tasa de actualización de 5 ms por canal, la frecuencia de disparo del **TIM3** se duplica a 2,5 ms (*línea 76*). Dentro de la **ISR**, la variable estática `i_ch` actúa como índice que alterna entre 0 y 1 en cada llamado, indicando a qué canal pertenece el dato recién convertido. Con ese índice se actualiza el canal de **PWM** correspondiente y se aplica el mismo filtro pasa bajos de primer orden que en los ejemplos anteriores (*líneas 84–90*).

Watchdog analógico con conversión continua

El **Analog Watchdog** (**AWD**) permite que el hardware vigile automáticamente si el valor convertido cae fuera de un rango predefinido, generando una **IRQ** sin que el programa tenga que comparar cada muestra por software. En este ejemplo se utilizan ambos **ADC** de forma independiente, aprovechando el circuito del ejemplo anterior: el **ADC1** monitorea el potenciómetro en `PA4` y enciende el LED en `PB8` si la tensión baja de 1 V; el **ADC2** monitorea el potenciómetro en `PA7` y apaga el LED si supera los 2,5 V.

Las funciones que introduce este ejemplo son las siguientes:

```
// Habilita el AWD sobre los canales regulares del ADC:
void adc_enable_analog_watchdog_regular(uint32_t adc);

// Restringe la vigilancia a un único canal (en lugar de todos):
void adc_enable_analog_watchdog_on_selected_channel(uint32_t adc, uint8_t channel);

// Configura los umbrales de la ventana (valores de 12 bits, 0..4095):
void adc_set_watchdog_low_threshold(uint32_t adc, uint16_t threshold);
void adc_set_watchdog_high_threshold(uint32_t adc, uint16_t threshold);

// Habilita la IRQ del AWD (el flag en la ISR es ADC_SR_AWD):
void adc_enable_awd_interrupt(uint32_t adc);

// Configura el modo de operación dual (se aplica solo al ADC1):
void adc_set_dual_mode(uint32_t mode);
ADC_CR1_DUALMOD_IND     // Modo independiente: cada ADC opera por su cuenta. (default)
ADC_CR1_DUALMOD_CRSISM  // Combinado: regulares simultáneos + inyectados simultáneos.
```

```
    ADC_CR1_DUALMOD_CRSATM  // Combinado: regulares simultáneos + trigger alternado.
    ADC_CR1_DUALMOD_CISFIM  // Combinado: inyectados simultáneos + intercalado rápido.
    ADC_CR1_DUALMOD_CISSIM  // Combinado: inyectados simultáneos + intercalado lento.
    ADC_CR1_DUALMOD_ISM     // Solo inyectados simultáneos.
    ADC_CR1_DUALMOD_RSM     // Solo regulares simultáneos.
    ADC_CR1_DUALMOD_FIM     // Solo intercalado rápido.
    ADC_CR1_DUALMOD_SIM     // Solo intercalado lento.
    ADC_CR1_DUALMOD_ATM     // Solo trigger alternado.
```

awd_adc1_adc2.c

```
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/adc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/rcc.h>

void config_gpios(void);
void config_adc(void);

int main(void)
{   /* Configuraciónd el clock principal */
    rcc_clock_setup_in_hse_8mhz_out_72mhz();
    /* Se configuran los canales para el ADC y salida del LED */
    config_gpios();
    /* Se configuran los ADC1 y ADC2 de modo independiente
       e interrupciones de AWD por umbral bajo y alto.        */
    config_adc();

    while (true);
}

void config_gpios(void)
{   // habilitación de los periféricos:
    rcc_periph_clock_enable(RCC_GPIOA);
    rcc_periph_clock_enable(RCC_GPIOB);
    // Canales analógicos a utilizar (CHANNEL4 y CHANNEL7):
    gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO4 | GPIO7);
    // LED en PB8:
    gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_50_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8);
    // Inicialmente apagar el LED:
    gpio_set(GPIOB, GPIO8);
}

void config_adc(void)
{
    // Se establece el prescaler para ambos ADCs (12MHz):
    rcc_set_adcpre(RCC_CFGR_ADCPRE_PCLK2_DIV6);

    /********** Configuración y calibración del  ADC1 **********/
    // Habilitación y reseteo del módulo:
    rcc_periph_clock_enable(RCC_ADC1);
    adc_power_off(ADC1);
    rcc_periph_reset_pulse(RST_ADC1);
    // Se disparará por software
    adc_enable_external_trigger_regular(ADC1, ADC_CR2_EXTSEL_SWSTART);
    // Se setea el modo continuo (se auto dispara al terminar la conversión)
    adc_set_continuous_conversion_mode(ADC1);
    // Para el ADC1 se leera el Canal 4
    uint8_t ch[] = {ADC_CHANNEL4};
    adc_set_regular_sequence(ADC1, 1, ch);
    // No se necesita mucha velocidad (es un potenciometro)
    adc_set_sample_time(ADC1, ADC_CHANNEL4, ADC_SMPR_SMP_239DOT5CYC);
    // Se configura el Analog WatchDog (AWD) para lo canales regulares
    adc_enable_analog_watchdog_regular(ADC1);
    // Solo un canal se va a comparar:
    adc_enable_analog_watchdog_on_selected_channel(ADC1, ADC_CHANNEL4);
    // Se configura el umbral bajo, si la señal baja de 1V
    adc_set_watchdog_low_threshold(ADC1, 1242); // ~1V
    // modo dual independiente (default)
    adc_set_dual_mode(ADC_CR1_DUALMOD_IND);
    // Calibración del ADC1:
```

```
61      adc_power_on(ADC1);
62      adc_reset_calibration(ADC1);
63      adc_calibrate(ADC1);
64      // Se habilita la interrupción del AWD:
65      adc_enable_awd_interrupt(ADC1);
66
67      /********** Configuración y calibración del  ADC2 **********/
68      // Idem ADC1:
69      rcc_periph_clock_enable(RCC_ADC2);
70      adc_power_off(ADC2);
71      rcc_periph_reset_pulse(RST_ADC2);
72      adc_enable_external_trigger_regular(ADC2, ADC_CR2_EXTSEL_SWSTART);
73      adc_set_continuous_conversion_mode(ADC2);
74      // Se va a comparar el canal 7
75      ch[0] = ADC_CHANNEL7;
76      adc_set_regular_sequence(ADC2, 1, ch);
77      adc_set_sample_time(ADC2, ADC_CHANNEL7, ADC_SMPR_SMP_239DOT5CYC);
78      adc_enable_analog_watchdog_regular(ADC2);
79      adc_enable_analog_watchdog_on_selected_channel(ADC2, ADC_CHANNEL7);
80      // Se configura el AWD del ADC2 por umbral alto, si supera 2.5V
81      adc_set_watchdog_high_threshold(ADC2, 3105); // ~2.5V
82      // Calibración del ADC2
83      adc_power_on(ADC2);
84      adc_reset_calibration(ADC2);
85      adc_calibrate(ADC2);
86      // Se habilita la interrupción del AWD:
87      adc_enable_awd_interrupt(ADC2);
88
89      // se habilita la IRQ del NVIC
90      nvic_enable_irq(NVIC_ADC1_2_IRQ);
91
92      // Se disparan ambos ADC, las próximas conversiones son automáticas
93      adc_start_conversion_regular(ADC1);
94      adc_start_conversion_regular(ADC2);
95  }
96
97  void adc1_2_isr(void)
98  {   // Si fue el AWD del ADC1 (canal 4 por debajo de 1V)
99      if (adc_get_flag(ADC1, ADC_SR_AWD)) {
100         adc_clear_flag(ADC1, ADC_SR_AWD); // Se baja el flag
101         gpio_clear(GPIOB, GPIO8);         // prende el LED
102     }
103     // Si fue el AWD del ADC2 (canal 7 por arriba de 2.5V)
104     if (adc_get_flag(ADC2, ADC_SR_AWD)) {
105         adc_clear_flag(ADC2, ADC_SR_AWD); // Se baja el flag
106         gpio_set(GPIOB, GPIO8);           // apaga el LED
107     }
108 }
```

La configuración del GPIO se separó en `config_gpios()` para mayor claridad. Allí se configuran los pines analógicos (`PA4` y `PA7`) y la salida del LED en `PB8`, que se inicializa apagado con `gpio_set()` dado que el LED es activo en bajo (*línea 30*).

El prescaler se establece una sola vez al inicio de `config_adc()` (*línea 36*), ya que ambos **ADC** comparten el mismo clock. A continuación se configuran los dos conversores de forma análoga, por lo que conviene seguir la secuencia del **ADC1** y notar las diferencias en el **ADC2**.

Para el **ADC1** se selecciona el modo continuo con la función de **libOpenCM3** `adc_set_continuous_conversion_mode(ADC1)` (*línea 46*): el ADC se auto-dispara al terminar cada conversión, por lo que un único disparo inicial es suficiente para mantenerlo corriendo indefinidamente. El **AWD** se habilita sobre los canales regulares (*línea 53*) y se restringe al `ADC_CHANNEL4` (*línea 55*). El umbral bajo se establece en 1242 cuentas (*línea 57*), que corresponde aproximadamente a 1 V sobre una referencia de 3,3 V: $1242/4095 \times 3{,}3\,\text{V} \approx 1\,\text{V}$. La llamada a `adc_set_dual_mode(ADC_CR1_DUALMOD_IND)` (*línea 59*)

es en este caso redundante —es el estado por defecto tras un reset— pero se incluye para explicitar la intención.

El **ADC2** se configura de forma idéntica, con dos diferencias: vigila el canal `ADC_CHANNEL7` y utiliza un umbral alto de 3105 cuentas (*línea 81*), equivalente a unos 2,5 V: $3105/4095 \times 3{,}3\,\text{V} \approx 2{,}5\,\text{V}$. La **IRQ** del **NVIC** se habilita una sola vez para ambos (*línea 90*), ya que comparten el vector `adc1_2_isr()`. Finalmente, ambos se disparan con `adc_start_conversion_regular()` (*líneas 93–94*) y desde ese momento corren de forma autónoma.

Dentro de la **ISR** se consulta el flag `ADC_SR_AWD` de cada **ADC** por separado para determinar cuál superó su umbral. A diferencia del flag `EOC` —que se baja automáticamente al leer el registro de datos— el flag `AWD` debe bajarse explícitamente con `adc_clear_flag()` antes de retornar, de lo contrario la interrupción se volvería a disparar de inmediato.

Sensor de temperatura interno

El *STM32F103* integra un sensor de temperatura conectado internamente al canal 16 del **ADC1**, accesible mediante la macro `ADC_CHANNEL_TEMP`. A diferencia de los canales externos, este canal no está disponible por defecto: debe habilitarse explícitamente antes de la calibración con la función:

```
void adc_enable_temperature_sensor(void); // habilita ch16 (temp) y ch17 (Vrefint)
```

Esta función también activa el canal 17 (V_{refint}), por lo que ambos quedan disponibles simultáneamente. Además, como se advirtió en la Subsubsección 5.2.4, el fabricante exige un tiempo de muestreo mínimo de 17,1 µs para estos canales internos, lo que obliga a seleccionar `ADC_SMPR_SMP_239DOT5CYC` con un `ADC_CLK` de 12 MHz: $239{,}5 + 12{,}5 = 252$ ciclos $\Rightarrow 252/12\,\text{MHz} = 21\,\mu\text{s}$.

Una vez obtenido el valor digital del **ADC**, la temperatura en grados Celsius se calcula con la fórmula provista en la hoja de datos del *STM32F103*:

$$T\,(°\text{C}) = \frac{V_{25} - V_{sense}}{Avg\_Slope} + 25$$

donde $V_{25} = 1{,}43\,\text{V}$ es la tensión de salida del sensor a 25 °C y $Avg\_Slope = 4{,}3\,\text{mV/°C}$ es la pendiente de la curva tensión-temperatura. Ambos son valores típicos: la hoja de datos indica que la variación de chip a chip puede alcanzar los 45 °C, por lo que este sensor es adecuado para detectar *variaciones* de temperatura pero no como referencia absoluta.

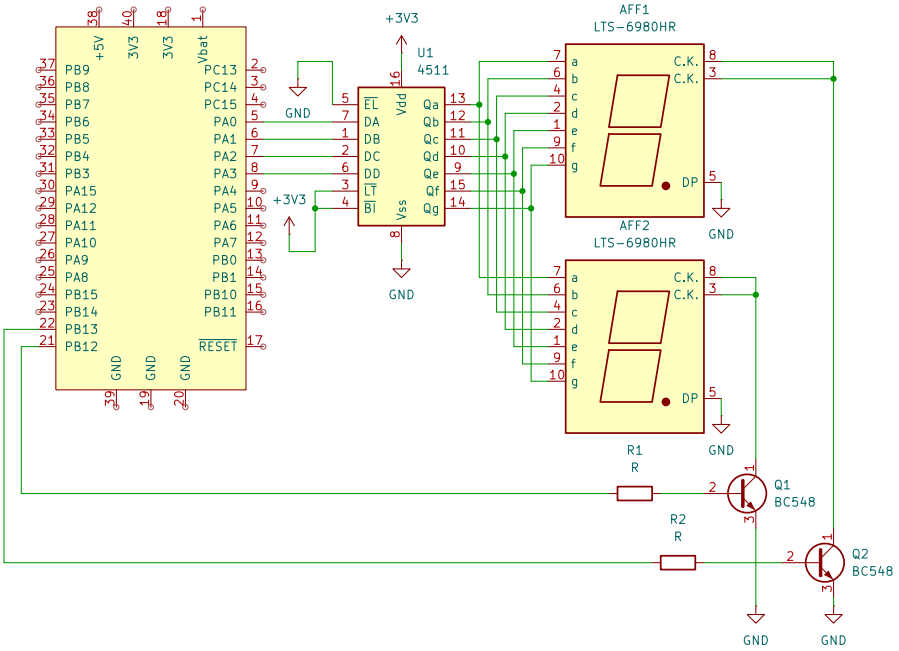
En el código se trabaja en aritmética entera con *milivolts* para evitar el uso de punto flotante[4]. La conversión del valor digital a tensión es: $V_{sense}\,(\text{mV}) = \texttt{adcdata} \times 3300/4095$, y sustituyendo en la fórmula anterior con los valores en mV ($V_{25} = 1430\,\text{mV}$, $Avg\_Slope = 4{,}3\,\text{mV/°C}$):

[4] Este microcontrolador no tine **FPU**, a diferencia de otros de la familia como los STM32F4xx.

$$T = \frac{(1430 - V_{sense}) \times 10}{43} + 25$$

El factor $\times 10/43$ equivale a dividir por $4{,}3$ usando solo enteros, lo que evita perder la parte decimal en la división intermedia.

En cuanto al circuito, este ejemplo utiliza dos displays de 7 segmentos multiplexados con un decodificador **CD4511** y dos transistores de conmutación para mostrar la temperatura. La lógica de multiplexado se maneja en la **ISR** del **TIM2**, independientemente del **ADC**.



temp_sensor.c

```
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/adc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/timer.h>

#define D_UNIT GPIO12 // PB12 unidades del 7 seg
#define D_TENS GPIO13 // PB13 decenas del 7 seg
// Pines de datos del 4511, PA0~PA3:
#define CD4511_DATA (GPIO0 | GPIO1 | GPIO2 | GPIO3)

volatile int32_t temp = 0;

void config_adc(void);
void config_tim_trig(void);
void config_7segmux(void);

int main(void)
{ /* Configuración del clock principal */
  rcc_clock_setup_in_hse_8mhz_out_72mhz();
  /* Configuración de los GPIOs y TIM2 para los 7 segmentos */
  config_7segmux();
  /* Configuración del ADC */
  config_adc();
  while (true);
}

```

```
28  void config_adc(void)
29  { // Se configura el prescaler del ADC:
30    rcc_periph_clock_enable(RCC_ADC1);
31    adc_power_off(ADC1); // apagar antes de resetear
32    rcc_periph_reset_pulse(RST_ADC1);
33    // 72Mhz/6 = 12Mhz (14Mhz es el máximo permitido)
34    rcc_set_adcpre(RCC_CFGR_ADCPRE_PCLK2_DIV6);
35    // configurar el trigger por el TRGO del TIM3 (cada 50ms):
36    adc_enable_external_trigger_regular(ADC1, ADC_CR2_EXTSEL_TIM3_TRGO);
37    // modo de conversión única
38    adc_set_single_conversion_mode(ADC1);
39    // modo discontinuo con subgrupos de 1 canal:
40    uint8_t ch[] = {ADC_CHANNEL_TEMP};
41    adc_set_regular_sequence(ADC1, 1, ch);
42    // Sampleo -> 12.5 + 239.5 -> 252 ciclos => 21us (al menos se
43    // necesitan 17.1us)
44    adc_set_sample_time(ADC1, ADC_CHANNEL_TEMP, ADC_SMPR_SMP_239DOT5CYC);
45    // Se habilita el sensor de temperatura
46    adc_enable_temperature_sensor();
47    // Calibración:
48    adc_power_on(ADC1);               // encender
49    adc_reset_calibration(ADC1);      // resetear registros de calibración
50    adc_calibrate(ADC1);              // calibrar (bloqueante)
51    adc_enable_eoc_interrupt(ADC1);   // habilitación del End-of-Conversion
52    nvic_enable_irq(NVIC_ADC1_2_IRQ); // habilita la ISR
53    config_tim_trig(); // habilitación del TIM3 para disparar el ADC1
54  }
55
56  void config_tim_trig(void)
57  {
58    rcc_periph_clock_enable(RCC_TIM3);
59    timer_set_mode(TIM3, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
60    timer_set_prescaler(TIM3, 3599); // 72MHz/3600 = 20KHz
61    timer_set_period(TIM3, 999);     // 20KHz / 1000 = 20Hz => T = 50ms
62    timer_set_master_mode(TIM3, TIM_CR2_MMS_UPDATE); // TRGO en cada update
63    timer_enable_counter(TIM3);                      // Arranca el timer
64  }
65
66  void config_7segmux(void)
67  { /* Configuración de pines para usar el 4511 y transistores de multiplexado. */
68    rcc_periph_clock_enable(RCC_GPIOA);
69    gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, CD4511_DATA);
70    gpio_clear(GPIOA, CD4511_DATA); // inicialmente en 0
71    rcc_periph_clock_enable(RCC_GPIOB);
72    gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, D_UNIT | D_TENS);
73    gpio_clear(GPIOB, D_UNIT | D_TENS); // ambos dígitos apagados
74
75    // Refresco por interrupción del TIM2 cada 5ms
76    rcc_periph_clock_enable(RCC_TIM2);
77    timer_set_mode(TIM2, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
78    timer_set_prescaler(TIM2, 719); // 72MHz/720 = 100KHz
79    timer_set_period(TIM2, 499);    // 100KHz / 500 = 200Hz => T = 5ms
80    timer_enable_irq(TIM2, TIM_DIER_UIE);
81    timer_enable_counter(TIM2); // Arranca el timer
82    nvic_enable_irq(NVIC_TIM2_IRQ);
83  }
84
85  void adc1_2_isr(void)
86  { // -------- Lectura del ADC --------
87    static uint16_t lastadc = 0;
88    uint32_t adcdata = adc_read_regular(ADC1);
89    adcdata = (lastadc * 7 + adcdata) / 8;
90    lastadc = adcdata;
91    // --- Calculo de la temperatura ---
92    // T = ((V25 - Vsense)/Avg_Slope) + 25
93    // en mV:
94    int32_t vsense_mv = (lastadc * 3300L) / 4095L;
95    // Datos de la hoja de datos del STM32F103xx:
96    // V25 = 1.43V -> 1430mV
97    // Avg_Slope = 4.3 mV/°C
98    temp = ((1430L - vsense_mv) * 10L / 43L) + 25L;
99    temp %= 99; // valores entre 0 y 99
100 }
```

```
101
102 void tim2_isr(void)
103 {
104   static bool mux = true; // Variable que cambia el dígito.
105   // Cada 5 ms:
106   timer_clear_flag(TIM2, TIM_SR_UIF); // se baja el flag.
107   gpio_clear(GPIOB, D_UNIT | D_TENS); // se apagan los dígitos.
108   // Se toma el nuevo valor del dígito correspondiente:
109   uint32_t digit = mux ? temp % 10 : temp / 10;
110   // Valor actual del GPIOA con los 4 primeros bits borrados.
111   uint32_t gpio_status = GPIO_ODR(GPIOA) & ~(0xFUL);
112   // Se escribe el nuevo valor en el puerto con los bits actualizados.
113   gpio_port_write(GPIOA, gpio_status | (digit & 0xFUL));
114   // Se enciende el dígito correspondiente:
115   gpio_set(GPIOB, mux ? D_UNIT : D_TENS);
116   mux = !mux; // Se cambia de dígito para la siguiente interrupción.
117 }
```

El programa tiene tres funciones de configuración. La primera, `config_7segmux()` (*líneas 66–83*), configura los pines de salida para el **CD4511** (`PA0`–`PA3`) y los transistores de selección de dígito (`PB12` y `PB13`), e inicializa el **TIM2** para generar una interrupción cada 5 ms que alternará el dígito visible. La segunda, `config_tim_trig()` (*líneas 56–64*), configura el **TIM3** en modo maestro para disparar el **ADC** cada 50 ms a través de su **TRGO**. La tercera, `config_adc()` (*líneas 28–54*), sigue la secuencia habitual con dos diferencias respecto a los ejemplos anteriores: el canal a convertir es `ADC_CHANNEL_TEMP` (*línea 40*) y se llama a `adc_enable_temperature_sensor()` antes de la calibración (*línea 46*), que es el requisito del fabricante para activar el canal interno.

En la **ISR** del **ADC** (*líneas 85–100*) se aplica el filtro pasa bajos habitual y luego se calcula la temperatura con la fórmula anterior (*línea 98*). El resultado se acota al rango 0–99 con el operador módulo (*línea 99*) para que quepa en dos dígitos decimales. Nótese que en este ejemplo no se verifica el flag `EOC` dentro de la **ISR**: como el único canal habilitado es el de temperatura y la única interrupción configurada es el **EOC**, llegar a la rutina ya garantiza que el dato está disponible.

En la **ISR** del **TIM2** (*líneas 102–116*) se implementa el multiplexado: primero se apagan ambos dígitos (*línea 107*), luego se calcula qué dígito mostrar en este ciclo (*línea 109*) y se actualiza el bus de datos del **CD4511** en `PA0`–`PA3`. Como el **GPIOA** tiene otros pines en uso, no se puede escribir el puerto completo directamente: hay que preservar el estado de los demás bits. Para ello se lee el registro de salida `GPIO_ODR(GPIOA)` —que refleja el valor que el programa escribió en las salidas, a diferencia del **IDR** que refleja el estado físico del pin—, se borran los cuatro bits inferiores con una máscara y se insertan los nuevos datos (*líneas 111–113*). Finalmente se enciende el dígito correspondiente (*línea 115*) y la variable estática `mux` alterna para que en la próxima interrupción se muestre el otro dígito.

Limitaciones del sensor interno

La precisión absoluta del sensor de temperatura del *STM32F103* es limitada: la variación de chip a chip puede superar los 45 °C según la hoja de datos. Para aplicaciones que requieran lecturas absolutas confiables conviene utilizar un sensor externo dedicado (por ejemplo, el **DS18B20** o el **LM35**). El sensor interno es más útil para detectar cambios relativos de temperatura, como proteger el sistema ante sobrecalentamiento.

5.4. Consideraciones finales

En esta unidad se cubrió el uso del **ADC** en sus configuraciones más directas: conversión única con disparo por software y por hardware, lectura de múltiples canales sin **DMA** mediante el modo discontinuo, vigilancia automática de umbrales con el **Analog Watchdog** y la lectura del sensor de temperatura interno. Sin embargo, quedan pendientes dos partes importantes del periférico.

Por un lado, los **canales inyectados** no fueron abordados en esta unidad. Como se describió al inicio, permiten interrumpir una secuencia regular para convertir hasta 4 canales con alta prioridad, de manera análoga a como una **ISR** interrumpe el programa principal. Su utilidad aparecerá naturalmente a medida que se avance en otras unidades, por ejemplo al necesitar medir una señal crítica sincronizada con un evento de otro periférico sin detener el resto del sistema.

Por otro lado, el modo de **escaneo** (*scan*), que permite convertir una secuencia de canales, solo cobra sentido práctico cuando se combina con el **DMA**: al existir un único registro de resultado (`ADC_DR`), cada nueva conversión sobreescribe la anterior antes de que el programa pueda leerla. El **DMA** resuelve esto transfiriendo cada resultado automáticamente a un arreglo en memoria, sin intervención del **CPU**. Esta combinación representa el caso de uso más potente del **ADC** y será retomada en la unidad dedicada al **DMA**, junto con el modo dual de los dos conversores.

5.5. Ejercicios

EJERCICIO N°9

Utilizando un potenciómetro conectado a un canal del **ADC**, implemente un sistema que encienda un LED distinto según el rango de tensión en el que se encuentre la señal. Divida el rango de 0 a 3,3 V en tres franjas iguales y asigne un LED a cada una. El cambio de LED debe ocurrir por interrupción, sin polling en el bucle principal.

EJERCICIO N°10

Extienda el ejercicio anterior reemplazando los LEDs por una barra de progreso de 8 LEDs. El encendido debe ser proporcional a la tensión leída: con 0 V ningún LED encendido, con 3,3 V los 8 encendidos, y con valores intermedios la cantidad correspondiente. Utilice el **Analog Watchdog** para detectar cuando la tensión supera el 90 % del rango máximo y parpadee todos los LEDs como señal de alerta.

EJERCICIO N°11

Agregue un segundo potenciómetro al circuito del **EJERCICIO N°9**. Utilizando el modo discontinuo, lea ambos canales y muestre cada valor en un par de displays de 7 segmentos multiplexados: el primero mostrará el valor del canal 1 escalado al rango 0–99 y el segundo el del canal 2 al mismo rango. Utilice

el **TIM3** como disparador periódico y el **TIM2** para el refresco del display.

EJERCICIO N°12

Diseñe un termostato simple utilizando el sensor de temperatura interno y el **Analog Watchdog**. Defina por software un umbral superior y uno inferior. Cuando la temperatura supere el umbral superior, un LED rojo debe encenderse y un LED verde debe apagarse; cuando caiga por debajo del umbral inferior, la lógica se invierte. Muestre la temperatura actual en dos displays de 7 segmentos multiplexados. Para probar el sistema puede calentar el integrado con los dedos o con aire caliente.

5.6. Links y materiales de la unidad

- **Documentación de libOpenCM3**
  http://libopencm3.org/docs/latest/stm32f1/html/modules.html
- **Datasheet STM32F103C8/B**
  https://www.st.com/resource/en/datasheet/stm32f103c8.pdf
- **Nota de aplicación sobre el ADC en los STM32**
  https://tinyurl.com/yr5z4278
- **Manual de referencia STM32F1xx**
  https://tinyurl.com/yxv9m5b2
- **STM32 Analogue-to-Digital Converter**
  https://embedded-lab.com/blog/stm32-adc-2/