uc3m

Departamento Tecnología Electrónica
dte UC3M

Tema 9: Uso de la Abstracción Hardware (CubeMX y HAL)

Sistemas Digitales Basados en Microprocesadores

Universidad Carlos III de Madrid

Dpto. Tecnología Electrónica

Nota: Las figuras utilizadas para ilustrar las características y funcionalidades del microcontrolador del curso se han obtenido de la documentación técnica disponible en https://www.st.com/en/microcontrollers-microprocessors/stm32l151-152.html

Índice

La Abstracción Hardware

- Hay proyectos donde su complejidad provoca interacción entre los recursos del microcontrolador
  - Cuando la complejidad de la solución sube, la probabilidad que las interacciones no contemplen todos los casos es alta
  - Es factible que aparezcan incompatibilidades funcionales entre los recursos
  - Muchas de esas interacciones, así como la configuración de los recursos, se puede hacer de forma automatizada
- Hay recursos del microcontrolador (por ejemplo, periféricos) que pueden ser extremadamente complejos
  - No sólo es necesario conocer los registros internos, sino que es posible que haya que crear estructuras de datos complejas o inclusos programar una gestión compleja
  - Dicha gestión es en gran medida común a todos los proyectos que usan ese recurso
- Es necesario proporcionar ayudas al desarrollador de tal modo que éste se pueda abstraer del hardware a utilizar:
  - Sistema automático de configuración de los recursos: CubeMX
  - Bibliotecas de programación que faciliten la vida al desarrollador: HAL

Abstracción Visual

Abstracción Visual

- Debido al gran número de recursos que ofrecen los microcontroladores actuales, es habitual que los fabricantes proporcionen herramientas que ayuden al desarrollador a elegir los recursos a utilizar
- Dichas herramientas permiten:
  - Seleccionar los periféricos a utilizar
  - Configurar la funcionalidad de cada uno de esos periféricos
    - Incluido los pines asociados
  - Comprobar la coherencia de las selecciones realizadas, detectando conflictos
  - Generar el código de inicialización de todos los recursos
    - Asegurando que la configuración de un recurso, no deteriora la configuración de otros
  - Modificar las selecciones realizadas, sin tener que iniciar el proyecto nuevamente

STM32 CubeMX

- ST Microelectronics proporciona como herramienta de Abstracción Visual para sus microcontroladores el CubeMX
- Esta herramienta permite:
  - Seleccionar el microcontrolador y/o la tarjeta de desarrollo utilizada
  - Indicar la funcionalidad de cada uno de los pines
    - Automáticamente activa los periféricos asociados, dejando los que no se utilizan en modo de bajo consumo
  - Seleccionar los periféricos a utilizar
    - Y para cada uno de ellos, seleccionar la funcionalidad y la configuración inicial
      - Muchas de dichas selecciones se hacen a través de menús desplegables
  - Determinar qué periféricos van a usar interrupciones
  - Incluir middleware, tales como periféricos avanzados o incluso sistemas operativos en tiempo real
  - Configurar el reloj del sistema y de cada uno de los buses de periféricos
  - Seleccionar, entre varios, el entorno de desarrollo a utilizar
  - Generar, no sólo el código, sino el proyecto completo en el entorno de desarrollo a utilizar
  - Incluir las bibliotecas de abstracción relacionadas con los recursos seleccionados

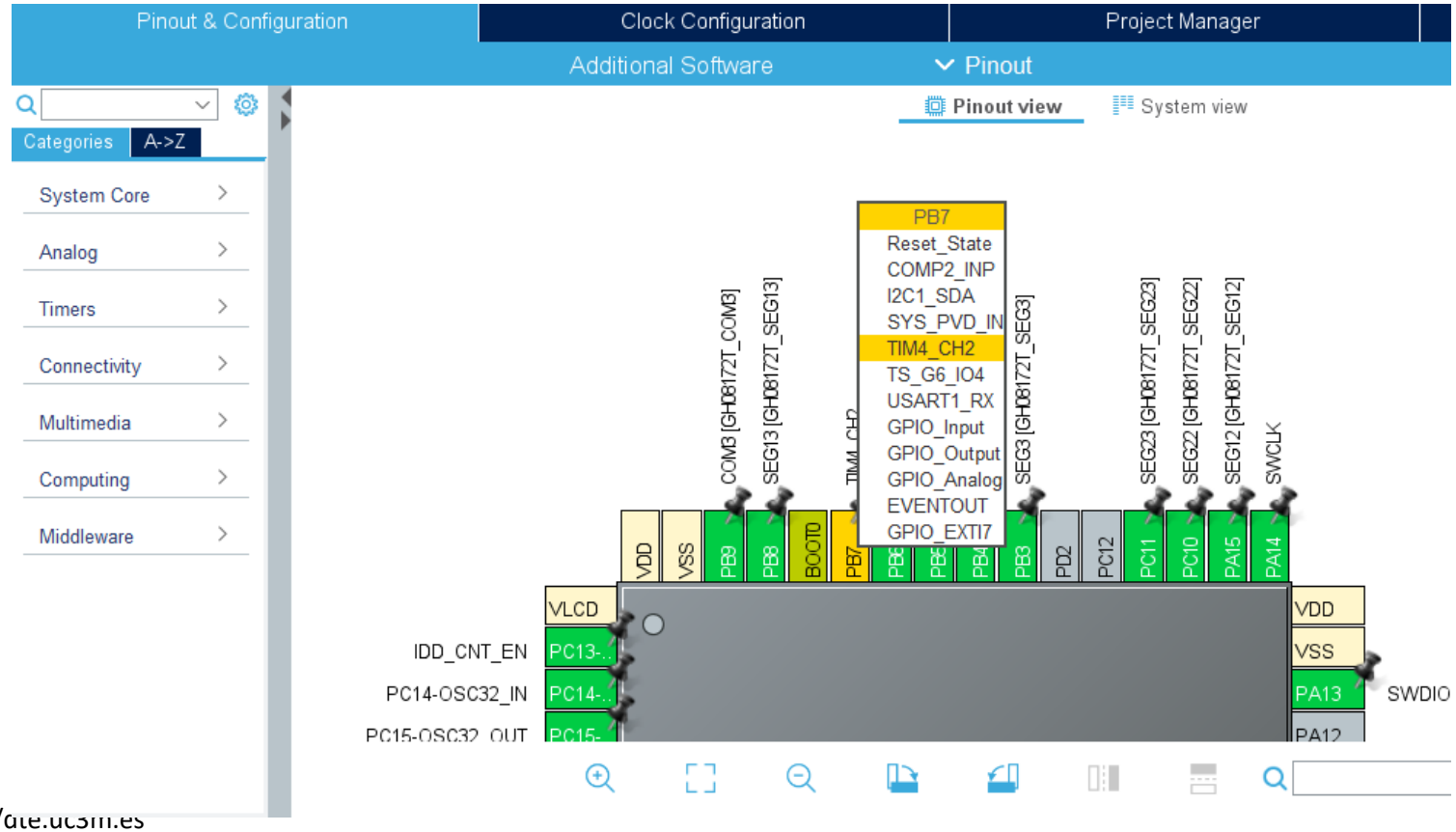
Ejemplos utilizando Abstracción Visual

Generación de una señal PWM

- Si esta es la única funcionalidad de nuestro programa, y además el DC va a ser fijo, todo el proyecto puede ser hacerlo con un mínimo número de líneas de código
- Como ejemplo, vamos a considerar el encendido del LED verde (PB7), el cual está asociado con en canal 2 del TIM4, y se le va a excitar con una señal PWM de periodo 2s y DC del 40%
  - Diseño de la solución:
    - Pin PB7 en AF para TIM4
    - PWM con PSC = 31999, ARR=1999 (2s), y CCR2=800 (40% de ARR+1)
  - Crea un nuevo proyecto, como siempre y modifica los valores en la perspectiva Cube para ajustarse a los cálculos anteriores (vea las 2 transparencia siguientes)
  - Save (para general el código)
  - Añade el código necesario a la sección 2 de USER CODE:
    - TIM4->CCER|= (0x01 << 4); // Activate the output
    - TIM4->CR1|=0x0001; // Start Timer
  - Build
  - Run/Debug

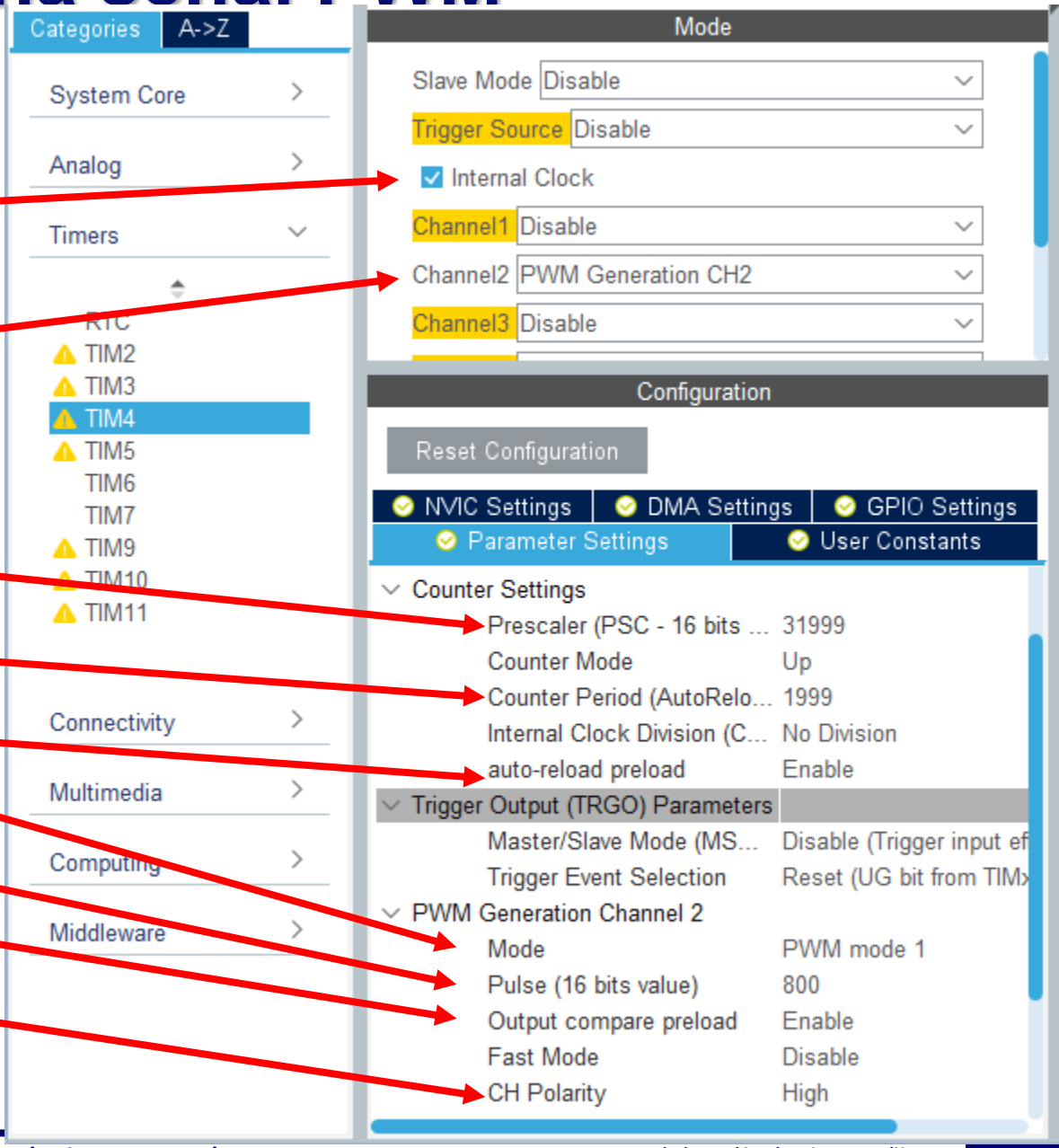
Generación de una señal PWM

- En la perspectiva de Cube, vaya al pinout y selecciona TIM_CH2 como funcionalidad del PB7
  - Lo verá en Naranja, indicando algún tipo de warning o error. Esto es porque el TIM4 no ha sido activado todavía.



Generación de una señal PWM

- Configura el TIM4:
  1. Activa el Internal Clock
  2. Selecciona PWM Generation CH2 en el Channel 2
- En los parámetros del TIM4, selecciona:
  1. Prescaler = 31999
  2. Counter Period = 1999
  3. Auto-reload = Enable
  4. Mode = mode 1
  5. Pulse = 800
  6. Output compare preload = Enable
  7. CH Polarity = High



Generación de una señal PWM

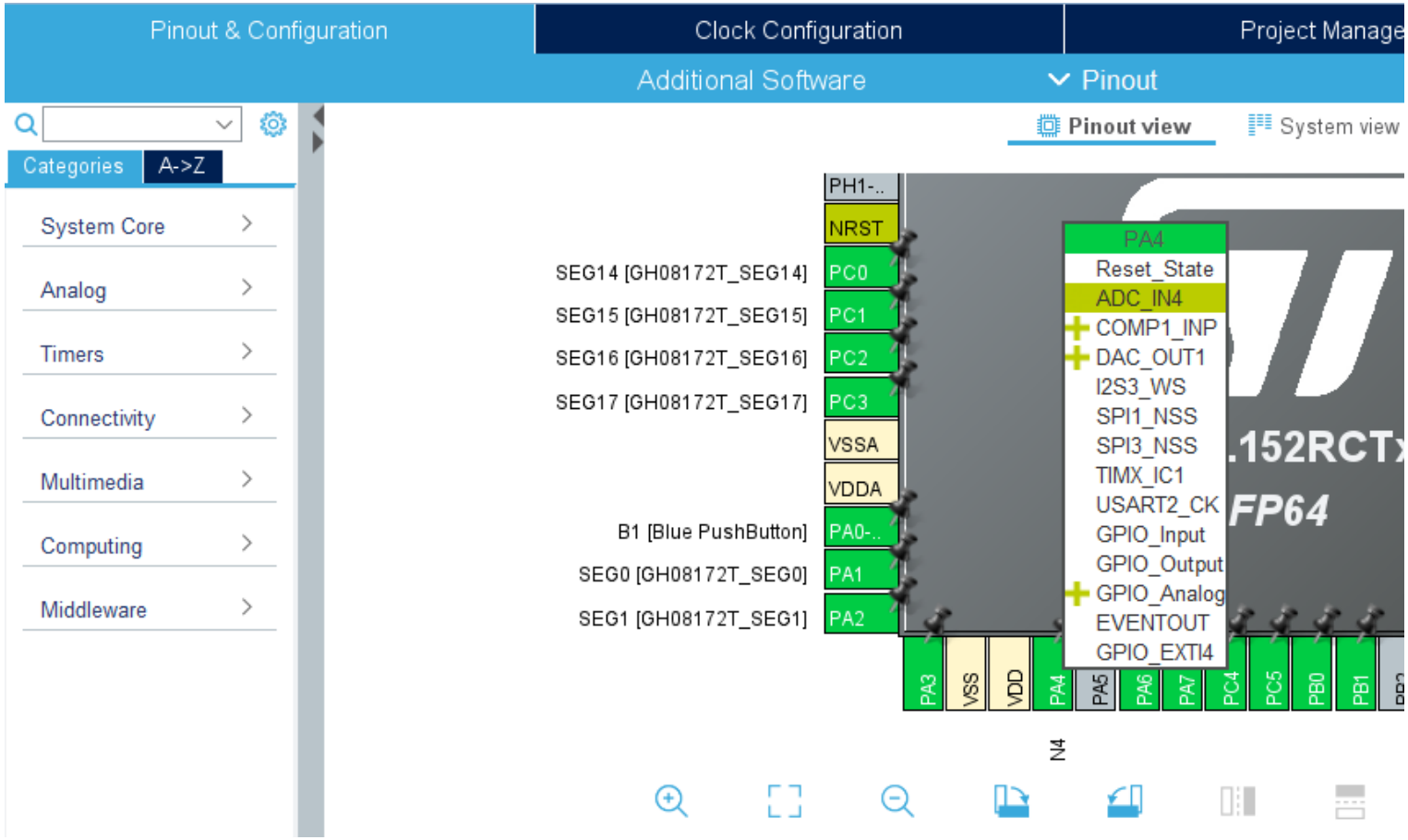
- Por seguridad, la perspectiva de Cube NO ACTIVA las salidas automáticamente, ni tampoco enciende las funcionalidades de los periféricos.
- Por tanto, hay que añadir algo de código al final de la configuración del proyecto (sección USER CODE BEGIN/END 2)
- En este caso:
  - Activa la salida del canal 2 del TIM4
    - TIM4->CCER |= (0x01 << 4);
  - Arranca el TIM4
    - TIM4->CR1 |= 0x01;

Muestreo de una Señal Analógica

- Consideremos ahora que se necesita muestrear una señal analógica al ritmo de 1Hz, mostrando el valor de conversión en el LCD:
  - Utilizando interrupciones, tanto para el temporizador como el ADC
- El diseño de la solución:
  - TIM2 para contar el tiempo de muestreo:
    - TOC sin salida
    - Utilicemos el canal 3 con IRQs
    - PSC = 31999; ARR = 0xFFFF; CCR3 = SAMPLING_TIME
    - #define SAMPLING_TIME 1000
    - Funcionalidad de la RAI:
      - Actualiza CCR3 sumándole SAMPLING_TIME
      - Arranca la conversión simple del ADC
      - Limpia los flags
  - ADC para muestrear PA4
    - PA4 como pin analógico
    - ADC con 12 bits y conversión simple
    - IRQ activa por el final de conversión (EOC)
    - La RAI actualizará una variable global, que será utilizada por el programa principal

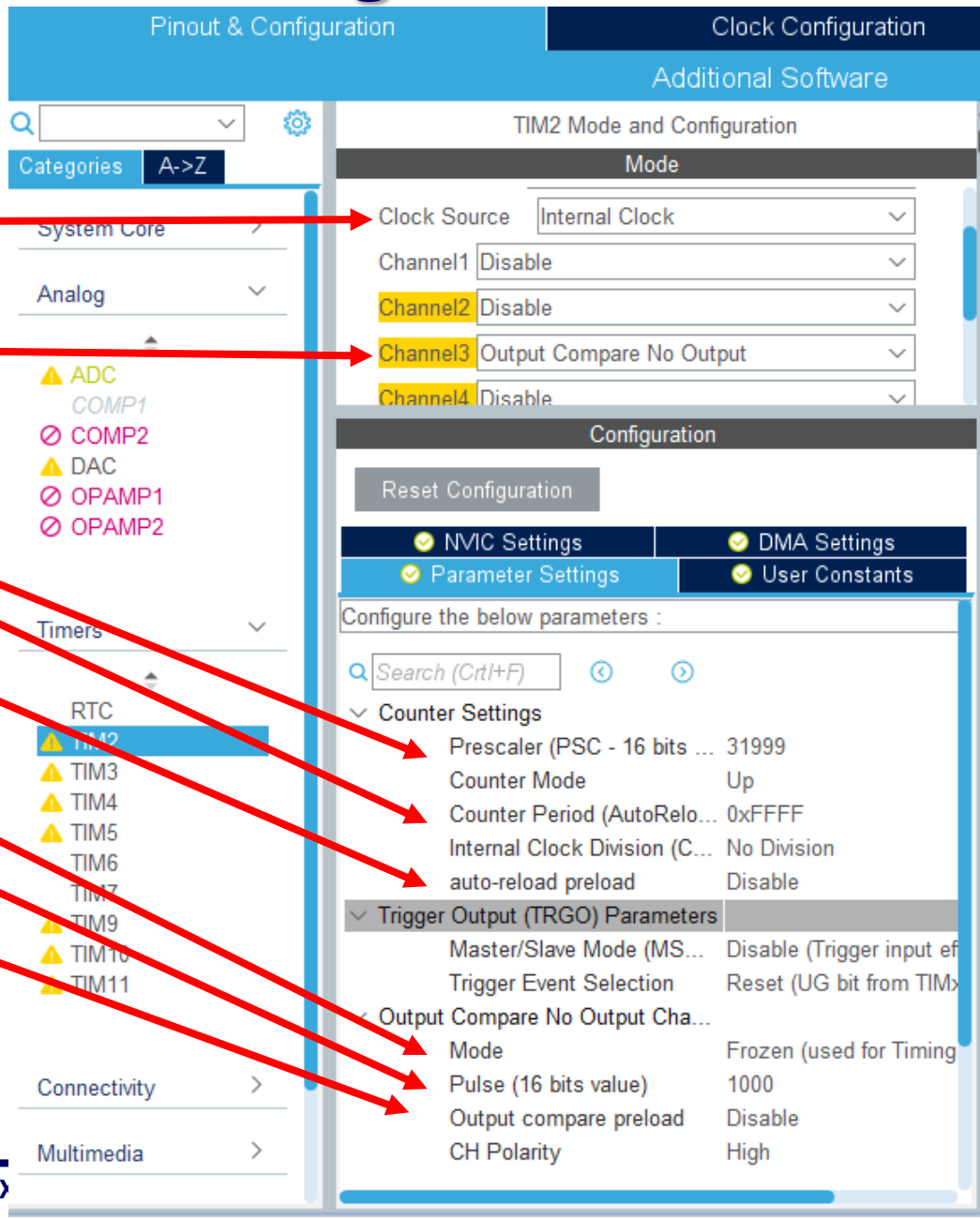
Muestreo de una Señal Analógica

- En la perspectiva Cube, elige la funcionalidad ADC_IN4 para el PA4



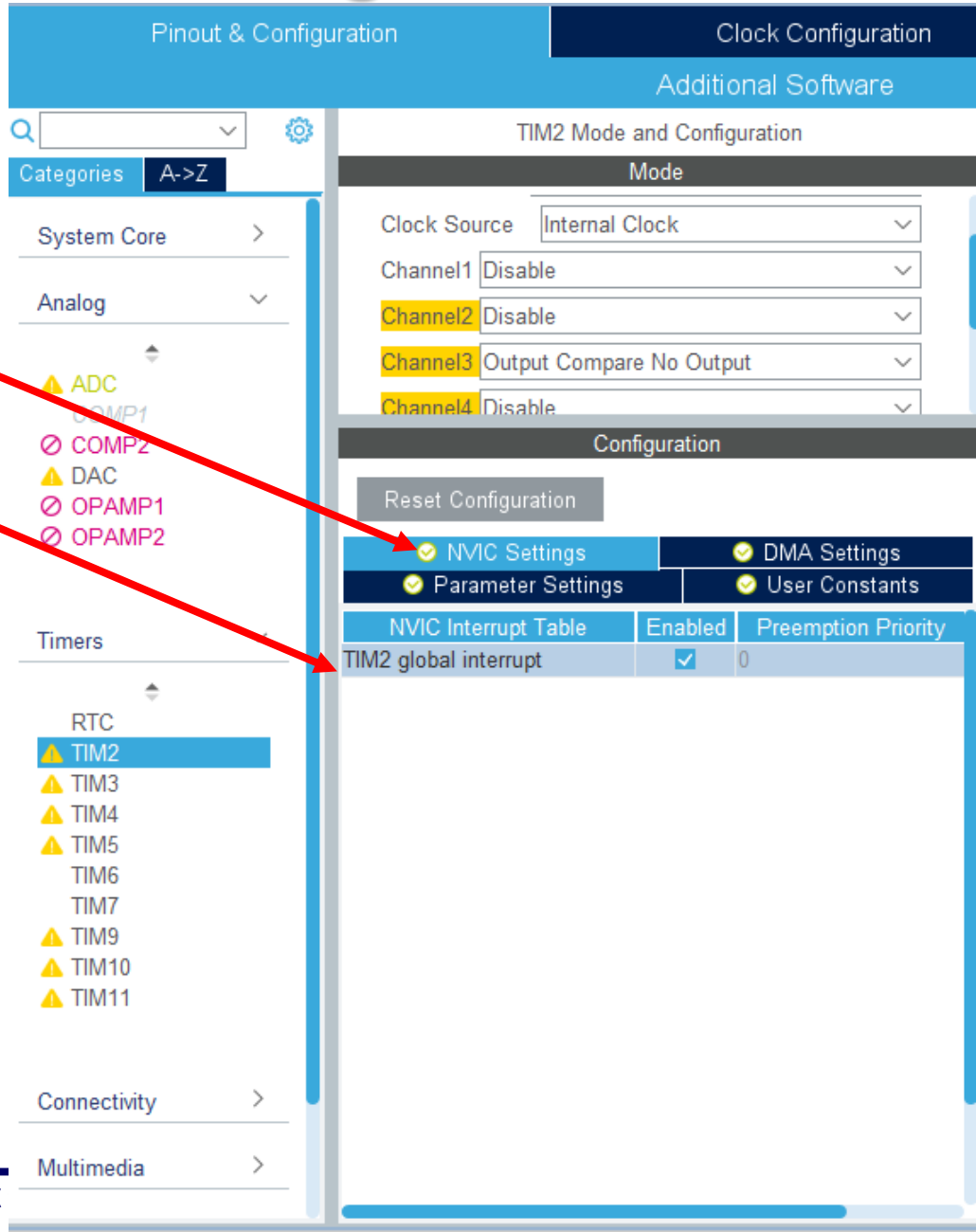
Muestreo de una Señal Analógica

- Configura el TIM2:
  1. Selecciona “Internal Clock” como Clock Source
  2. Pon “Output Compare No Output” en el Channel 3
- En los parámetros del TIM2:
  1. Prescaler = 31999
  2. Counter Period = 0xFFFF
  3. Auto-reload = Disable
  4. Mode = Frozen
  5. Pulse = 1000
  6. Output compare preload = Disable



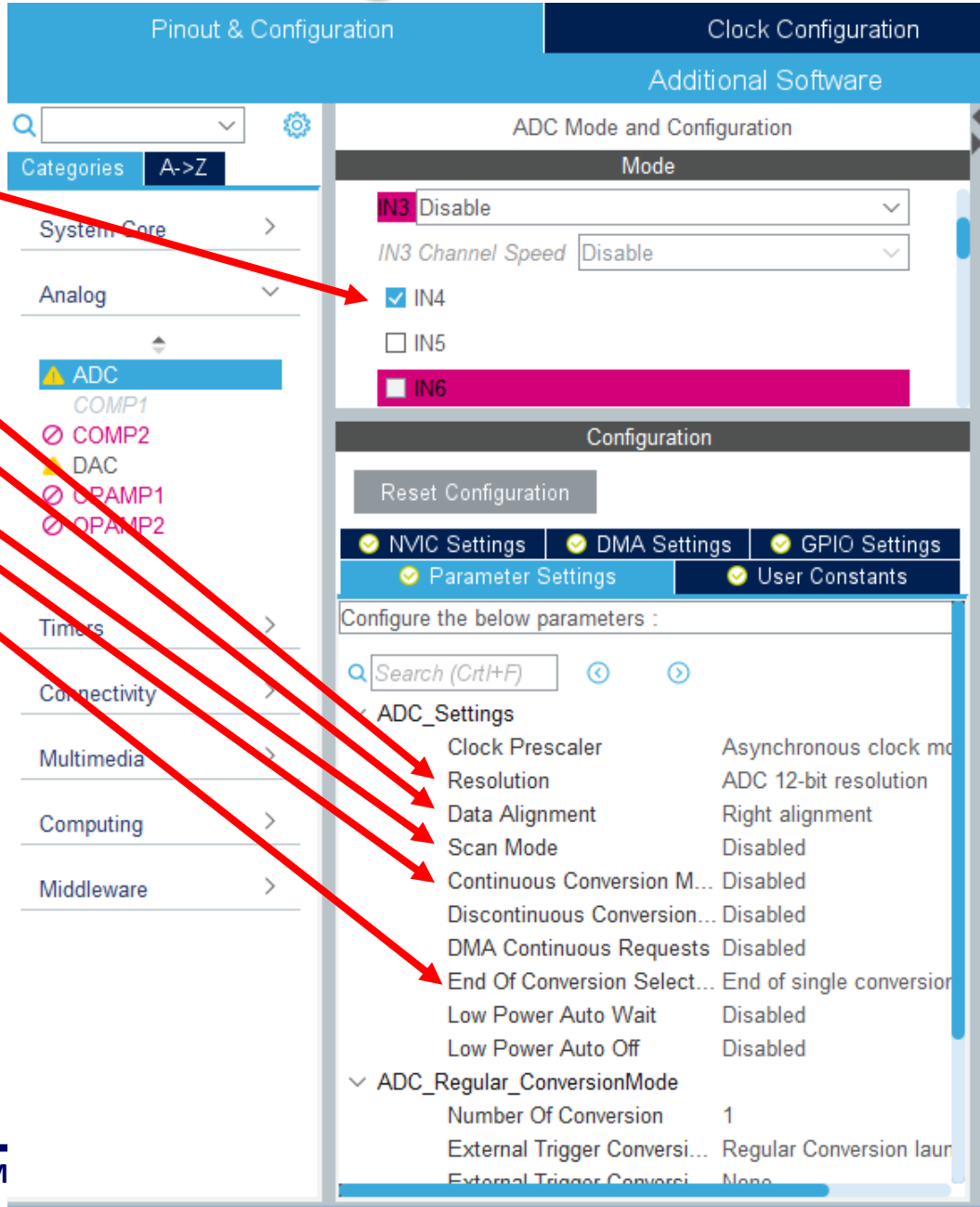
Muestreo de una Señal Analógica

- Como se van a utilizar interrupciones, hay que activar el NVIC para el TIM2
  1. Selecciona la pestaña NVIC Settings
  2. Habilita el TIM2 global interrupt



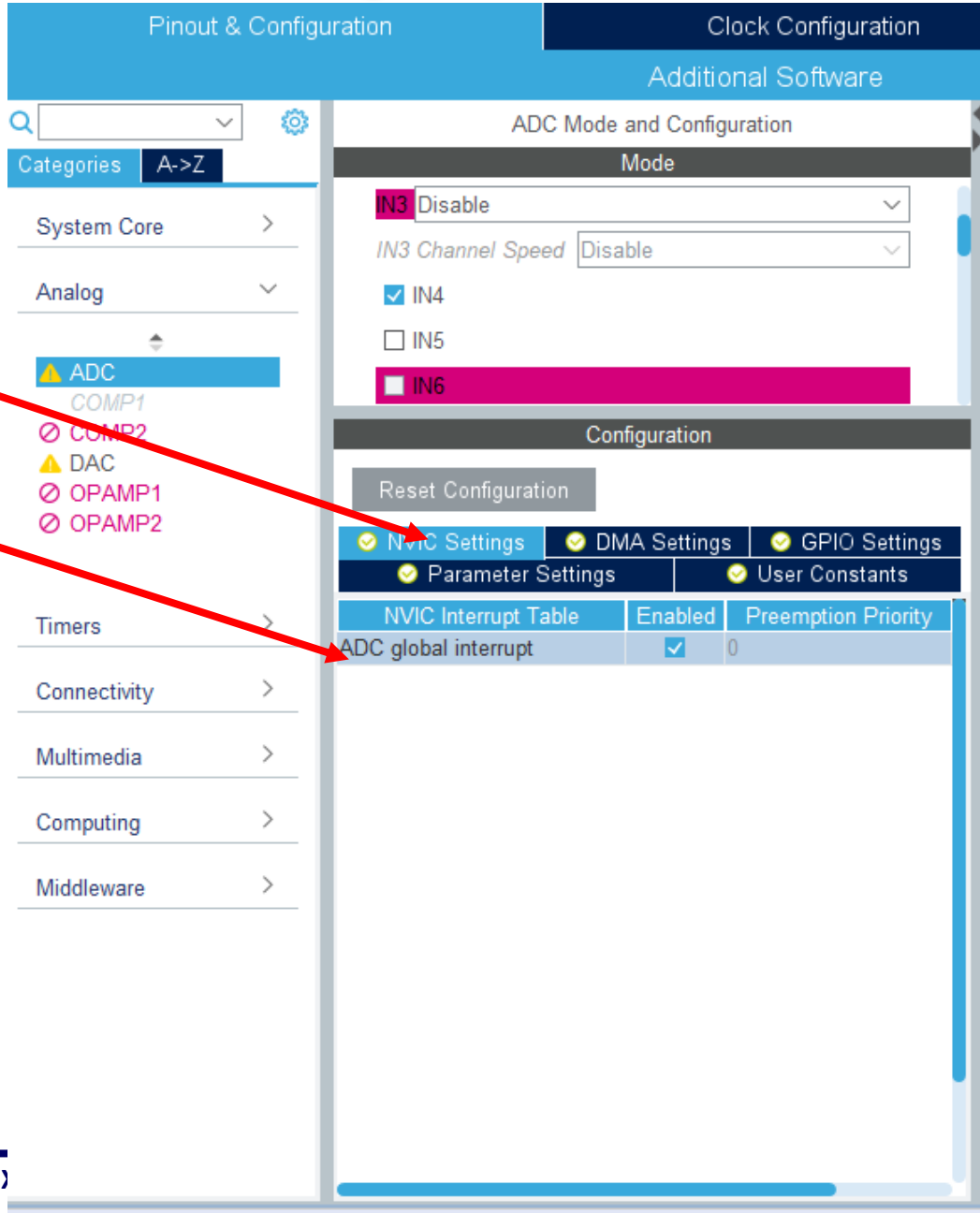
Muestreo de una Señal Analógica

- Configura el ADC:
  1. Activa la entrada IN4
- En los parámetros del ADC:
  1. Resolution = 12-bit
  2. Data alignment = Right
  3. Scan Mode = Disable
  4. Continuous Conv. = Disable
  5. End of Conversion = Single



Muestreo de una Señal Analógica

- Como se van a utilizar interrupciones, se debe activar el NVIC para el ADC
  1. Selecciona la pestaña NVIC Settings
  2. Habilita la ADC global interrupt



Muestreo de una Señal Analógica

- Cube IDE también generará el código para las RAIs, pero en lugar de hacerlo en el main.c, lo hace en stm32l1xx_it.c
- Añade tu código en la sección USER CODE BEGIN ADC1_IRQn 0

Ch9_Cube_ADC... | main.c | stm32l1xx_it.c | system_stm3... | stm32l1xx_ha...

```
194 /******************************************************************************
195  * STM32L1xx Peripheral Interrupt Handlers
196  * Add here the Interrupt Handlers for the used peripherals.
197  * For the available peripheral interrupt handler names,
198  * please refer to the startup file (startup_stm32l1xx.s).
199  ******************************************************************************/
200
201 /**
202   * @brief This function handles ADC global interrupt.
203   */
204 void ADC1_IRQHandler(void)
205 {
206   /* USER CODE BEGIN ADC1_IRQn 0 */
207   if ((ADC1->SR & (1 << 1))!=0){  // If EoC, catch the value
208     conversion_value = ADC1->DR;
209     }
210   ADC1->SR = 0;
211
212   /* USER CODE END ADC1_IRQn 0 */
213   HAL_ADC_IRQHandler(&hadc);
214   /* USER CODE BEGIN ADC1_IRQn 1 */
215
216   /* USER CODE END ADC1_IRQn 1 */
217 }
```

Muestreo de una Señal Analógica

- También añade el código de la RAI del TIM2 en el mismo fichero

Ch9_Cube_ADC... | main.c | stm32l1xx_it.c | system_stm3... | stm32l1xx_ha... | Reset_Handle... | main.h

```
218
219 /**
220   * @brief This function handles TIM2 global interrupt.
221   */
222 void TIM2_IRQHandler(void)
223 {
224   /* USER CODE BEGIN TIM2_IRQn 0 */
225   if ((TIM2->SR & (1 << 3))!=0)  // If the comparison is successful, then the IRQ is
226                                  // launched and this ISR is executed. This line check
227                                  // which event launched the ISR
228       {
229       TIM2->CCR3 += 1000; // Update the comparison value
230       ADC1->CR2 |= 0x40000000;     // Start conversion (SWSTART = 1)
231       }
232   TIM2->SR = 0; // Clear all flags
233
234   /* USER CODE END TIM2_IRQn 0 */
235   HAL_TIM_IRQHandler(&htim2);
236   /* USER CODE BEGIN TIM2_IRQn 1 */
237
238   /* USER CODE END TIM2_IRQn 1 */
239 }
240
```

Muestreo de una Señal Analógica

- Como las RAIs están en otro fichero, debe permitir la visibilidad de las variables globales y constantes a través del main.h
- En main.c:
  - /* USER CODE BEGIN PV */
  - unsigned short conversion_value=0;
  - /* USER CODE END PV */

- En main.h:
  - /* USER CODE BEGIN EFP */
  - extern unsigned short conversion_value;
  - /* USER CODE END EFP */

  - …

  - /* USER CODE BEGIN Private defines */
  - #define SAMPLING_TIME 1000
  - /* USER CODE END Private defines */

Muestreo de una Señal Analógica

```
/* USER CODE BEGIN 1 */
unsigned short pre_value=0;
unsigned char text[7];
/* USER CODE END 1 */
```

```
/* USER CODE BEGIN 2 */
BSP_LCD_GLASS_Init();
BSP_LCD_GLASS_BarLevelConfig(0);
BSP_LCD_GLASS_Clear();
TIM2->CR1|=0x01;
TIM2->SR = 0;
TIM2->DIER |= 1 << 3; // Activate TIM2 CH3 IRQ
ADC1->CR1 |= 1 << 5;  // Activate EOC IRQ
ADC1->CR2 |= 1 << 0;  // Power ADC on
/* USER CODE END 2 */
```

```
/* USER CODE BEGIN WHILE */
while (1)
{
  if (conversion_value != pre_value){
    pre_value = conversion_value;
    // Convert result to a string
    Bin2Ascii(conversion_value,&text[0]);
    // Show result in the LCD
    BSP_LCD_GLASS_DisplayString((uint8_t *)text);
  }
  /* USER CODE END WHILE */
```

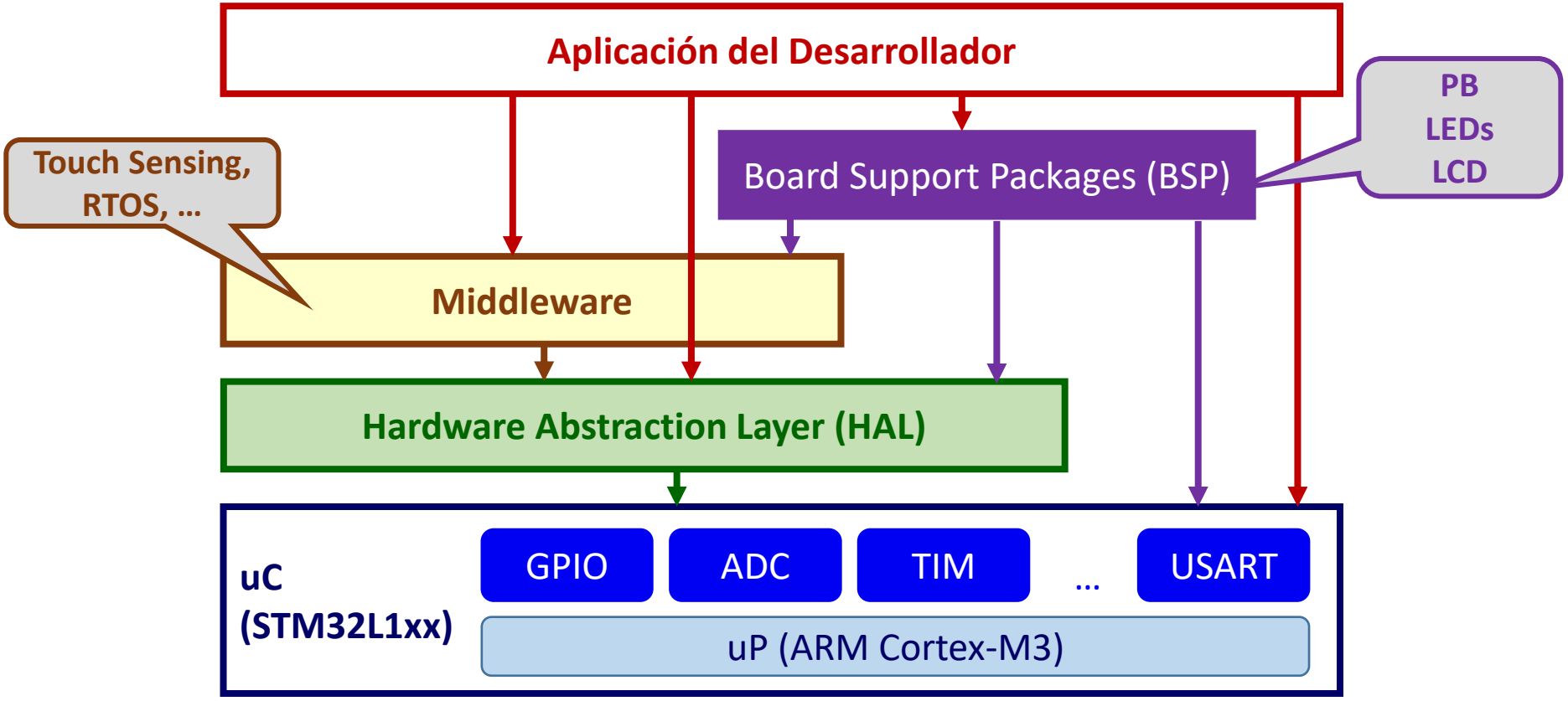
Abstracción en Programación

Hardware Abstraction Libraries (HAL)

- A la hora de programar, es habitual que parte del trabajo sea común con otros proyectos y desarrolladores, por lo que se intenta reutilizar código que se considera ya probado
- Esta misma filosofía es compartida por los fabricantes de microcontroladores, sistemas de desarrollo y placas de desarrollo
  - Proporcionan bibliotecas que faciliten el uso de los distintos recursos
- En concreto, a la hora de desarrollar, es habitual el planteamiento de una solución en capas, que vayan desde el uso directo del hardware del microcontrolador (es decir, los recursos), a funciones de alto nivel que abstraigan al desarrollador de la complicación de gestionar dichos recursos
  - A partir de esas librerías de alto nivel, el desarrollador implementa su aplicación final
  - El desarrollador, si lo necesita, puede acceder directamente a los recursos del microcontrolador, saltándose las funciones de alto nivel
    - Según se complica el desarrollo esto es cada vez menos interesante

Hardware Abstraction Libraries (HAL)

- ST Microelectronics proporciona una serie de bibliotecas que facilitan ese desarrollo software:
  - Otros fabricantes proporcionan soluciones equivalentes



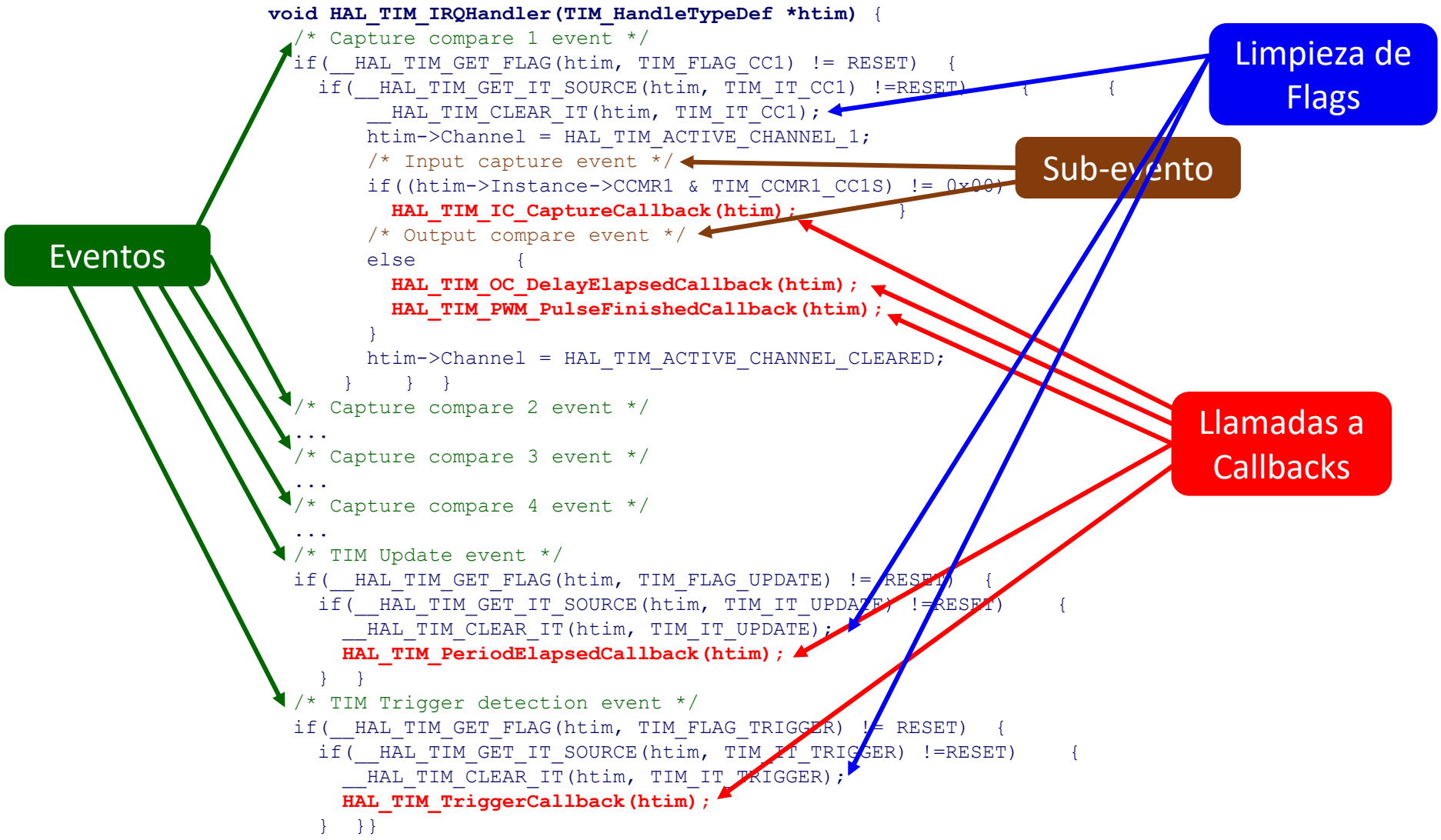
Hardware Abstraction Libraries (HAL)

- Las HAL aumentan el nivel de abstracción de trabajo, simplificando llamadas y creando nuevos usos. Por ejemplo, en las de STM32L1:
  - ADC:
    - Existe ya una función para hacer polling sobre la conversión
  - TIM:
    - De los 3 modos de funcionamiento básico (TIC, TOC y PWM), se pasa a 6 modos: Time Base, TIC, TOC, PWM, One-pulse mode output, Encoder mode output
- Además, STM32L1 proporciona unos BSPs que gestionan:
  - El botón USER: `BSP_PB_Init()`, `BSP_PB_GetState()`, etc.
  - Los LEDs: `BSP_LED_Init()`, `BSP_LED_On()`, `BSP_LED_Off()`, `BSP_LED_Toggle()`
  - El LCD: tal y como se ha utilizado a lo largo del curso (por ejemplo, `BSP_LCD_GLASS_DisplayString()`)

Hardware Abstraction Libraries (HAL)

- Las HAL suelen estar orientadas a un tipo de programación más de alto nivel, por lo que determinados conceptos de bajo nivel se diluyen
- Un ejemplo son las RAIs:
  - El uso de abstracción contempla que las RAIs se generan de forma automática
  - Esa generación ya contempla todos los tipos de eventos que pueden dar lugar a esa RAI y realiza la limpieza de los flags correspondientes
  - Para cada uno de esos eventos, la RAI plantea la llamada a una función que puede programar el desarrollador
    - A estas funciones se las denomina **Callback**
  - Si el desarrollador no ha implementado una determinada Callback, el compilador la sustituye por un código inocuo
- La siguiente transparencia muestra la RAI implementada en las HAL para el TIM
  - El desarrollador debería implementar dentro del mail aquella/s Callback necesaria

Hardware Abstraction Libraries (HAL)




```
void HAL_TIM_IRQHandler(TIM_HandleTypeDef *htim) {
  /* Capture compare 1 event */
  if(__HAL_TIM_GET_FLAG(htim, TIM_FLAG_CC1) != RESET)   {
    if(__HAL_TIM_GET_IT_SOURCE(htim, TIM_IT_CC1) !=RESET)      {      {
        __HAL_TIM_CLEAR_IT(htim, TIM_IT_CC1);
        htim->Channel = HAL_TIM_ACTIVE_CHANNEL_1;
        /* Input capture event */
        if((htim->Instance->CCMR1 & TIM_CCMR1_CC1S) != 0x00)
          HAL_TIM_IC_CaptureCallback(htim);          }
        /* Output compare event */
        else        {
          HAL_TIM_OC_DelayElapsedCallback(htim);
          HAL_TIM_PWM_PulseFinishedCallback(htim);
        }
        htim->Channel = HAL_TIM_ACTIVE_CHANNEL_CLEARED;
     }    }  }
  /* Capture compare 2 event */
  ...
  /* Capture compare 3 event */
  ...
  /* Capture compare 4 event */
  ...
  /* TIM Update event */
  if(__HAL_TIM_GET_FLAG(htim, TIM_FLAG_UPDATE) != RESET)   {
    if(__HAL_TIM_GET_IT_SOURCE(htim, TIM_IT_UPDATE) !=RESET)     {
      __HAL_TIM_CLEAR_IT(htim, TIM_IT_UPDATE);
      HAL_TIM_PeriodElapsedCallback(htim);
    }  }
  /* TIM Trigger detection event */
  if(__HAL_TIM_GET_FLAG(htim, TIM_FLAG_TRIGGER) != RESET)   {
    if(__HAL_TIM_GET_IT_SOURCE(htim, TIM_IT_TRIGGER) !=RESET)     {
      __HAL_TIM_CLEAR_IT(htim, TIM_IT_TRIGGER);
      HAL_TIM_TriggerCallback(htim);
    }  }}
```

Hardware Abstraction Libraries (HAL)

- El uso de HAL conlleva un coste:
  - Se pierde algo de control sobre los procesos internos
    - Por ejemplo, las RAIs pasan a ser gestionadas internamente, y el desarrollador se encarga de programar funciones Callback
    - En Time Base, los timers se basan en utilizar la pre-carga
  - Aunque se simplifica el código a escribir, el código generado:
    - Ocupa más, puesto que supone muchas más funciones e incluso código relativo a funcionalidades no utilizadas
    - Tarda más en ejecutarse, puesto que se realizan múltiples llamadas a funciones en situaciones donde podría no hacerse ninguna llamada
      - Por ejemplo, una RAI va a llamar a las distintas funciones Callback , en lugar de hacer la gestión la propia RAI
    - Se va a hacer un uso intensivo de la pila
- La documentación de la capa de abstracción de la familia L1 de los microcontroladores STM32 se encuentra en:
  - https://www.st.com/en/embedded-software/stm32cubel1.html
- En concreto, la documentación de las HAL es un documento de 1300 páginas que se encuentra en:
  - https://www.st.com/resource/en/user_manual/dm00132099.pdf

en.DM00132099_HAL_for_L1xxx.pdf - Adobe Acrobat Reader DC

Archivo Edición Ver Ventana Ayuda

Inicio Herramientas Documento 1 / 1300 Compartir Iniciar sesión

Marcadores

life.augmented

UM1816
User manual

Description of STM32L1 HAL and low-layer drivers

Introduction

STMCube™ is an STMicroelectronics original initiative to make developers' lives easier by reducing development effort, time and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube Version 1.x includes:

- The STM32CubeMX, a graphical software configuration tool that allows the generation of C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per Series (such as STM32CubeL1 for STM32L1 Series)
  - The STM32Cube HAL, STM32 abstraction layer embedded software ensuring maximized portability across the STM32 portfolio.
  - The low-layer APIs (LL) offering a fast light-weight expert-oriented layer which is closer to the hardware than the HAL. LL APIs are available only for a set of peripherals.
  - A consistent set of middleware components such as RTOS, USB, TCP/IP and Graphics.
  - All embedded software delivered with a full set of examples.

The HAL driver layer provides a generic multi-instance simple set of APIs (application programming interfaces) to interact with the upper layer (application, libraries and stacks).

The HAL driver APIs are split into two categories: generic APIs which provide common and generic functions for all the STM32 series and extension APIs which include specific and customized functions for a given line or part number. The HAL drivers include a complete set of ready-to-use APIs which simplify the user application implementation. As an example, the communication peripherals contain APIs to initialize and configure the peripheral, manage data transfers in polling mode, handle interrupts or DMA, and manage communication errors.

The HAL drivers are feature-oriented instead of IP-oriented. As an example, the timer APIs are split into several categories following the IP functions: basic timer, capture, pulse width modulation (PWM), and so on. The HAL driver layer implements run-time failure detection by checking the input values of all functions. Such dynamic checking contributes to enhance the firmware robustness. Run-time detection is also suitable for user application development and debugging.

The LL drivers offer hardware services based on the available features of the STM32 peripherals. These services reflect exactly the hardware capabilities and provide atomic operations that must be called following the programming model described in the product line reference manual. As a result, the LL services are not based on standalone processes and do not require any additional memory resources to save their states, counter or data pointers: all operations are performed by changing the associated peripheral registers content. Contrary to the HAL, the LL APIs are not provided for peripherals for which optimized access is not a key feature, or for those requiring heavy software configuration and/or complex upper level stack (such as USB).

The HAL and LL are complementary and cover a wide range of applications requirements:

- The HAL offers high-level and feature-oriented APIs, with a high-portability level. They hide the MCU and peripheral complexity to end-user.
- The LL offers low-level APIs at registers level, with better optimization but less portability. They require deep knowledge of the MCU and peripherals specifications.

The source code of HAL and LL drivers is developed in Strict ANSI-C which makes it independent from the development tools. It is checked with CodeSonar™ static analysis tool. It is fully documented and is MISRA-C 2004 compliant.

en.DM00132099_HAL_for_L1xxx.pdf - Adobe Acrobat Reader DC

Archivo Edición Ver Ventana Ayuda

Inicio Herramientas Documento 74 / 1300 Compartir Iniciar sesión

Marcadores

6 HAL ADC Generic Driver

6.1 ADC Firmware driver registers structures

6.1.1 ADC_InitTypeDef

Data Fields

- *uint32_t ClockPrescaler*
- *uint32_t Resolution*
- *uint32_t DataAlign*
- *uint32_t ScanConvMode*
- *uint32_t EOCSelection*
- *uint32_t LowPowerAutoWait*
- *uint32_t LowPowerAutoPowerOff*
- *uint32_t ChannelsBank*
- *uint32_t ContinuousConvMode*
- *uint32_t NbrOfConversion*
- *uint32_t DiscontinuousConvMode*
- *uint32_t NbrOfDiscConversion*
- *uint32_t ExternalTrigConv*
- *uint32_t ExternalTrigConvEdge*
- *uint32_t DMAContinuousRequests*

Field Documentation

- *uint32_t ADC_InitTypeDef::ClockPrescaler*
  Select ADC clock source (asynchronous clock derived from HSI RC oscillator) and clock prescaler. This parameter can be a value of *ADC_ClockPrescaler* Note: In case of usage of channels on injected group, ADC frequency should be lower than AHB clock frequency /4 for resolution 12 or 10 bits, AHB clock frequency /3 for resolution 8 bits, AHB clock frequency /2 for resolution 6 bits. Note: HSI RC oscillator must be preliminarily enabled at RCC top level.
- *uint32_t ADC_InitTypeDef::Resolution*
  Configures the ADC resolution. This parameter can be a value of *ADC_Resolution*
- *uint32_t ADC_InitTypeDef::DataAlign*
  Specifies ADC data alignment to right (MSB on register bit 11 and LSB on register bit 0) (default setting) or to left (if regular group: MSB on register bit 15 and LSB on register bit 4, if injected group (MSB kept as signed value due to potential negative value after offset application): MSB on register bit 14 and LSB on register bit 3). This parameter can be a value of *ADC_Data_align*
- *uint32_t ADC_InitTypeDef::ScanConvMode*
  Configures the sequencer of regular and injected groups. This parameter can be associated to parameter 'DiscontinuousConvMode' to have main sequence subdivided in successive parts. If disabled: Conversion is performed in single mode (one channel converted, the one defined in rank 1). Parameters 'NbrOfConversion' and 'InjectedNbrOfConversion' are discarded (equivalent to set to 1). If enabled: Conversions are performed in sequence mode (multiple ranks defined by 'NbrOfConversion'/'InjectedNbrOfConversion' and each channel rank). Scan direction is upward: from rank1 to rank 'n'. This parameter can be a value of *ADC_Scan_mode*

en.DM00132099_HAL_for_L1xxx.pdf - Adobe Acrobat Reader DC

Archivo Edición Ver Ventana Ayuda

Inicio Herramientas Documento 79 / 1300 Compartir Iniciar sesión

Marcadores

6.2.2 How to use this driver

Configuration of top level parameters related to ADC

1. Enable the ADC interface
   - As prerequisite, ADC clock must be configured at RCC top level. Caution: On STM32L1, ADC clock frequency max is 16MHz (refer to device datasheet). Therefore, ADC clock prescaler must be configured in function of ADC clock source frequency to remain below this maximum frequency.

   - Two clock settings are mandatory:
     - ADC clock (core clock).
     - ADC clock (conversions clock). Only one possible clock source: derived from HSI RC 16MHz oscillator (HSI). ADC is connected directly to HSI RC 16MHz oscillator. Therefore, RCC PLL setting has no impact on ADC. PLL can be disabled (".PLL.PLLState = RCC_PLL_NONE") or enabled with HSI16 as clock source (".PLL.PLLSource = RCC_PLLSOURCE_HSI") to be used as device main clock source SYSCLK. The only mandatory setting is ".HSIState = RCC_HSI_ON"
     - Example: Into HAL_ADC_MspInit() (recommended code location) or with other device clock parameters configuration:
     - __HAL_RCC_ADC1_CLK_ENABLE();
     - HAL_RCC_GetOscConfig(&RCC_OscInitStructure);
     - RCC_OscInitStructure.OscillatorType = (... | RCC_OSCILLATORTYPE_HSI);
     - RCC_OscInitStructure.HSIState = RCC_HSI_ON;
     - RCC_OscInitStructure.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
     - RCC_OscInitStructure.PLL.PLLState = RCC_PLL_NONE;
     - RCC_OscInitStructure.PLL.PLLSource = ...
     - RCC_OscInitStructure.PLL...
     - HAL_RCC_OscConfig(&RCC_OscInitStructure);
   - ADC clock prescaler is configured at ADC level with parameter "ClockPrescaler" using function HAL_ADC_Init().
2. ADC pins configuration
   - Enable the clock for the ADC GPIOs using macro __HAL_RCC_GPIOx_CLK_ENABLE()
   - Configure these ADC pins in analog mode using function HAL_GPIO_Init()
3. Optionally, in case of usage of ADC with interruptions:
   - Configure the NVIC for ADC using function HAL_NVIC_EnableIRQ(ADCx_IRQn)

Marcadores

6.2.4 IO operation functions

This section provides functions allowing to:

- Start conversion of regular group.
- Stop conversion of regular group.
- Poll for conversion complete on regular group.
- Poll for conversion event.
- Get result of regular channel conversion.
- Start conversion of regular group and enable interruptions.
- Stop conversion of regular group and disable interruptions.
- Handle ADC interrupt request
- Start conversion of regular group and enable DMA transfer.

- Stop conversion of regular group and disable ADC DMA transfer.

This section contains the following APIs:

- *HAL_ADC_Start()*
- *HAL_ADC_Stop()*
- *HAL_ADC_PollForConversion()*
- *HAL_ADC_PollForEvent()*
- *HAL_ADC_Start_IT()*
- *HAL_ADC_Stop_IT()*
- *HAL_ADC_Start_DMA()*
- *HAL_ADC_Stop_DMA()*
- *HAL_ADC_GetValue()*
- *HAL_ADC_IRQHandler()*
- *HAL_ADC_ConvCpltCallback()*
- *HAL_ADC_ConvHalfCpltCallback()*
- *HAL_ADC_LevelOutOfWindowCallback()*
- *HAL_ADC_ErrorCallback()*

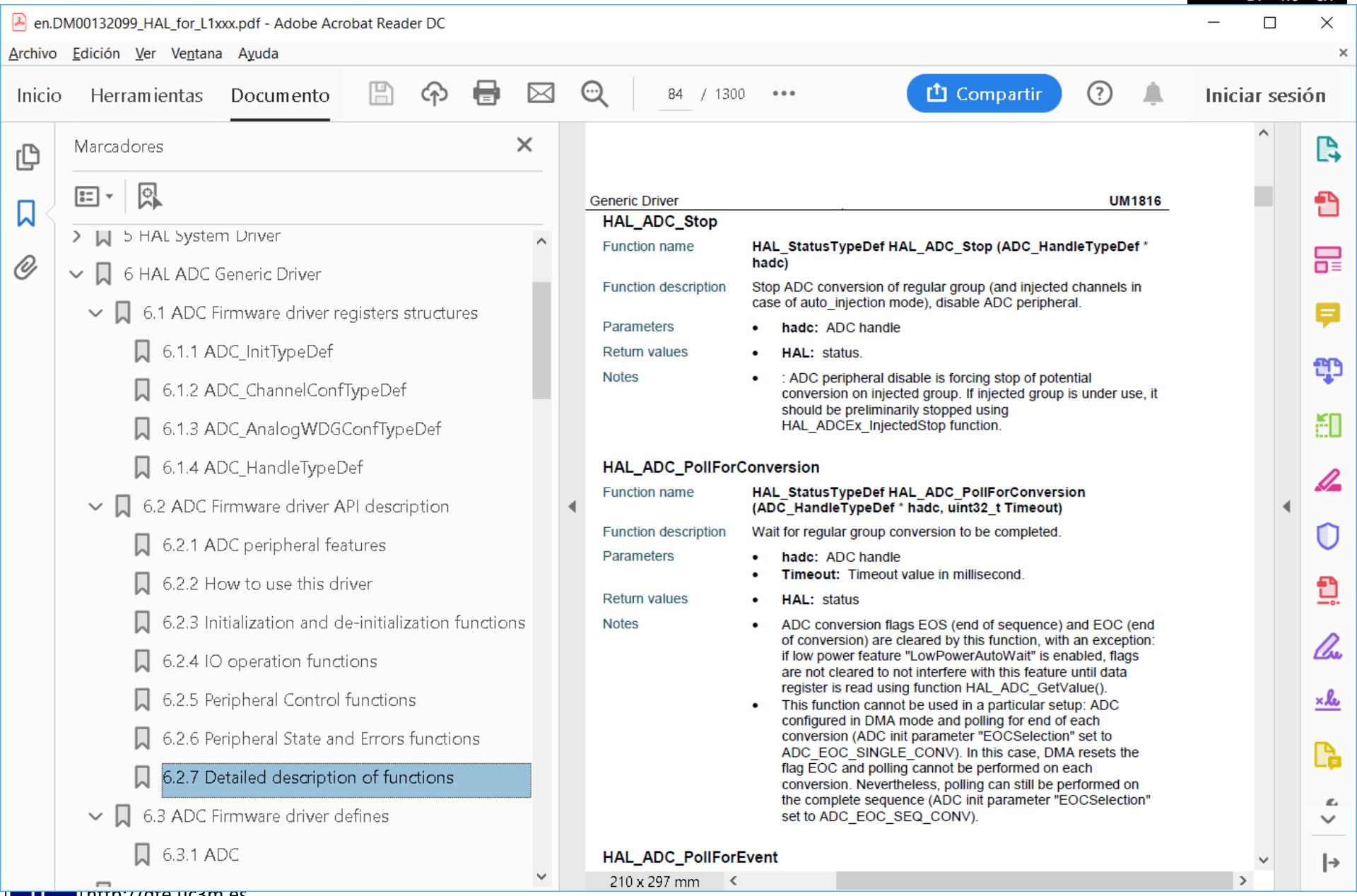
6.2.5 Peripheral Control functions

This section provides functions allowing to:

- Configure channels on regular group
- Configure the analog watchdog

This section contains the following APIs:

- *HAL_ADC_ConfigChannel()*
- *HAL_ADC_AnalogWDGConfig()*


en.DM00132099_HAL_for_L1xxx.pdf - Adobe Acrobat Reader DC
Archivo Edición Ver Ventana Ayuda
Inicio Herramientas Documento
84 / 1300
Compartir
Iniciar sesión
Marcadores
5 HAL System Driver
6 HAL ADC Generic Driver
6.1 ADC Firmware driver registers structures
6.1.1 ADC_InitTypeDef
6.1.2 ADC_ChannelConfTypeDef
6.1.3 ADC_AnalogWDGConfTypeDef
6.1.4 ADC_HandleTypeDef
6.2 ADC Firmware driver API description
6.2.1 ADC peripheral features
6.2.2 How to use this driver
6.2.3 Initialization and de-initialization functions
6.2.4 IO operation functions
6.2.5 Peripheral Control functions
6.2.6 Peripheral State and Errors functions
6.2.7 Detailed description of functions
6.3 ADC Firmware driver defines
6.3.1 ADC
Generic Driver UM1816
HAL_ADC_Stop
Function name HAL_StatusTypeDef HAL_ADC_Stop (ADC_HandleTypeDef * hadc)
Function description Stop ADC conversion of regular group (and injected channels in case of auto_injection mode), disable ADC peripheral.
Parameters • hadc: ADC handle
Return values • HAL: status.
Notes • : ADC peripheral disable is forcing stop of potential conversion on injected group. If injected group is under use, it should be preliminarily stopped using HAL_ADCEx_InjectedStop function.
HAL_ADC_PollForConversion
Function name HAL_StatusTypeDef HAL_ADC_PollForConversion (ADC_HandleTypeDef * hadc, uint32_t Timeout)
Function description Wait for regular group conversion to be completed.
Parameters • hadc: ADC handle
• Timeout: Timeout value in millisecond.
Return values • HAL: status
Notes • ADC conversion flags EOS (end of sequence) and EOC (end of conversion) are cleared by this function, with an exception: if low power feature "LowPowerAutoWait" is enabled, flags are not cleared to not interfere with this feature until data register is read using function HAL_ADC_GetValue().
• This function cannot be used in a particular setup: ADC configured in DMA mode and polling for end of each conversion (ADC init parameter "EOCSelection" set to ADC_EOC_SINGLE_CONV). In this case, DMA resets the flag EOC and polling cannot be performed on each conversion. Nevertheless, polling can still be performed on the complete sequence (ADC init parameter "EOCSelection" set to ADC_EOC_SEQ_CONV).
HAL_ADC_PollForEvent
210 x 297 mm

Ejemplo sin Interrupciones

Descripción y Análisis

- El ejercicio trata de hacer conversiones de un canal analógico (IN4) a ritmo de 44KHz, sacando por el LCD la media de las 10 últimas medidas. El botón USER encenderá y apagará la funcionalidad.
- Recursos a utilizar:
  - GPIOA: Botón USER (PA0)
    - Se va a utilizar el BSP del PB (Push Button) para el BOTON_USER
  - LCD
  - TIM3 en modo TOC sin salida hardware, con pre-escalado de 31 (para saltos de 1us, e incremento de CCR = 23 (para cumplir con 1/44KHz)
  - ADC en 12 bits y conversión simple

Diagrama de Bloques

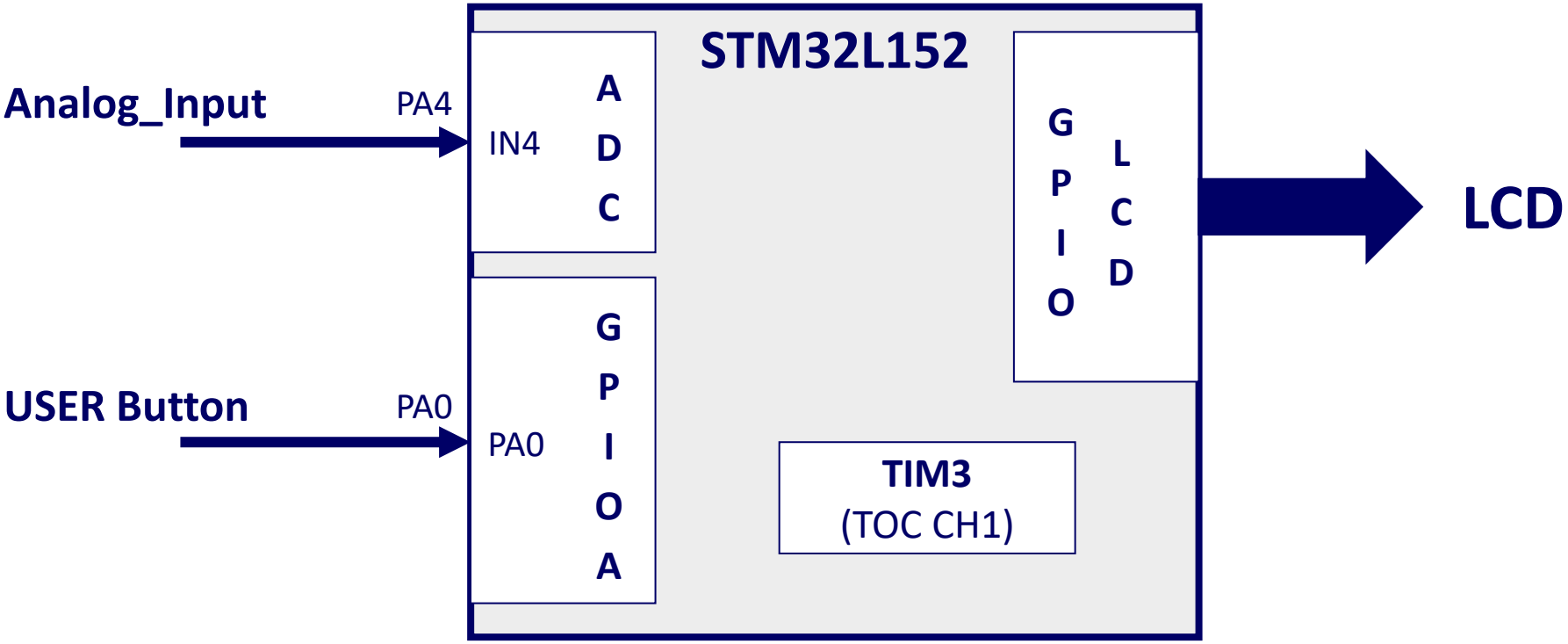
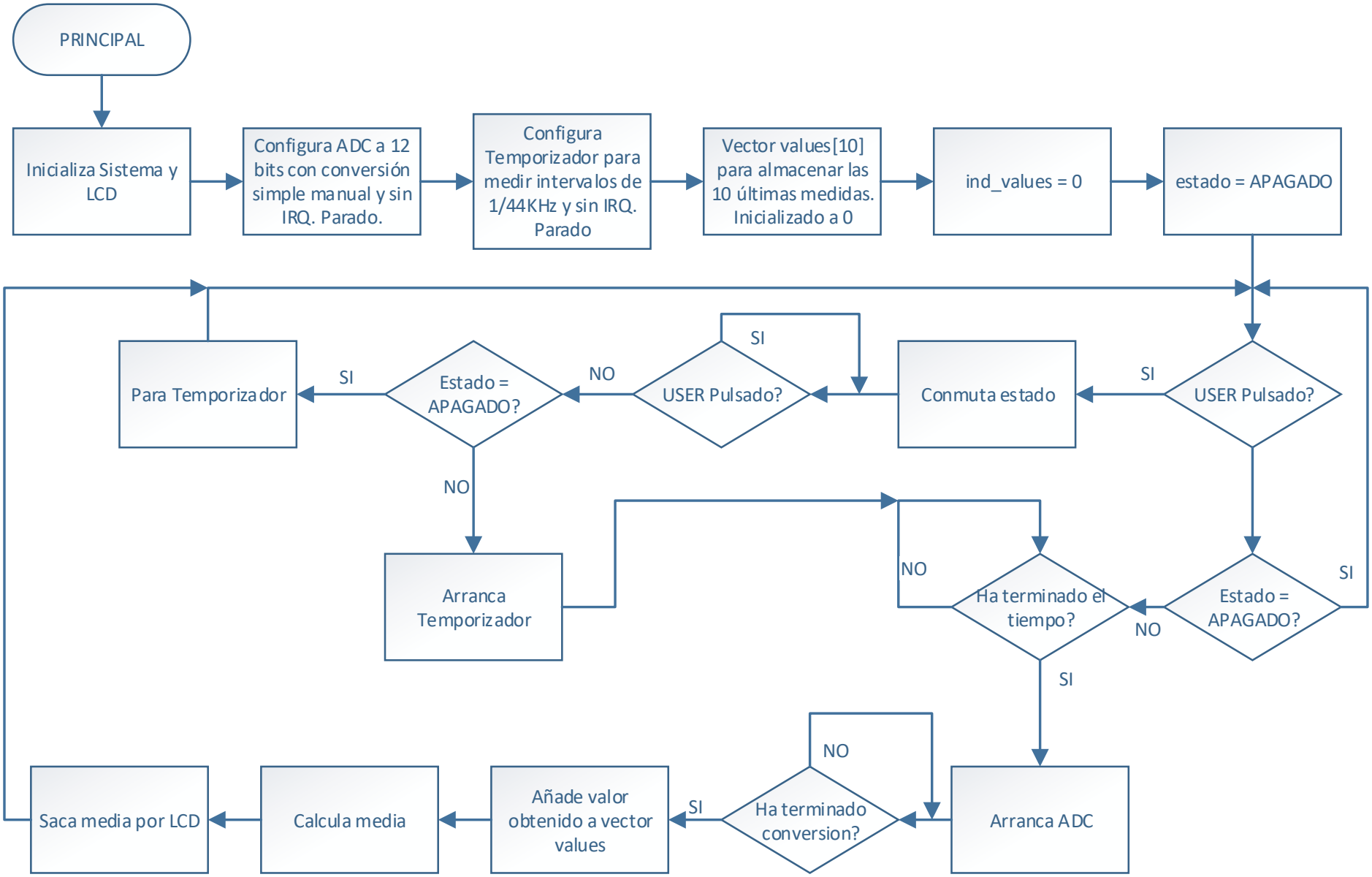


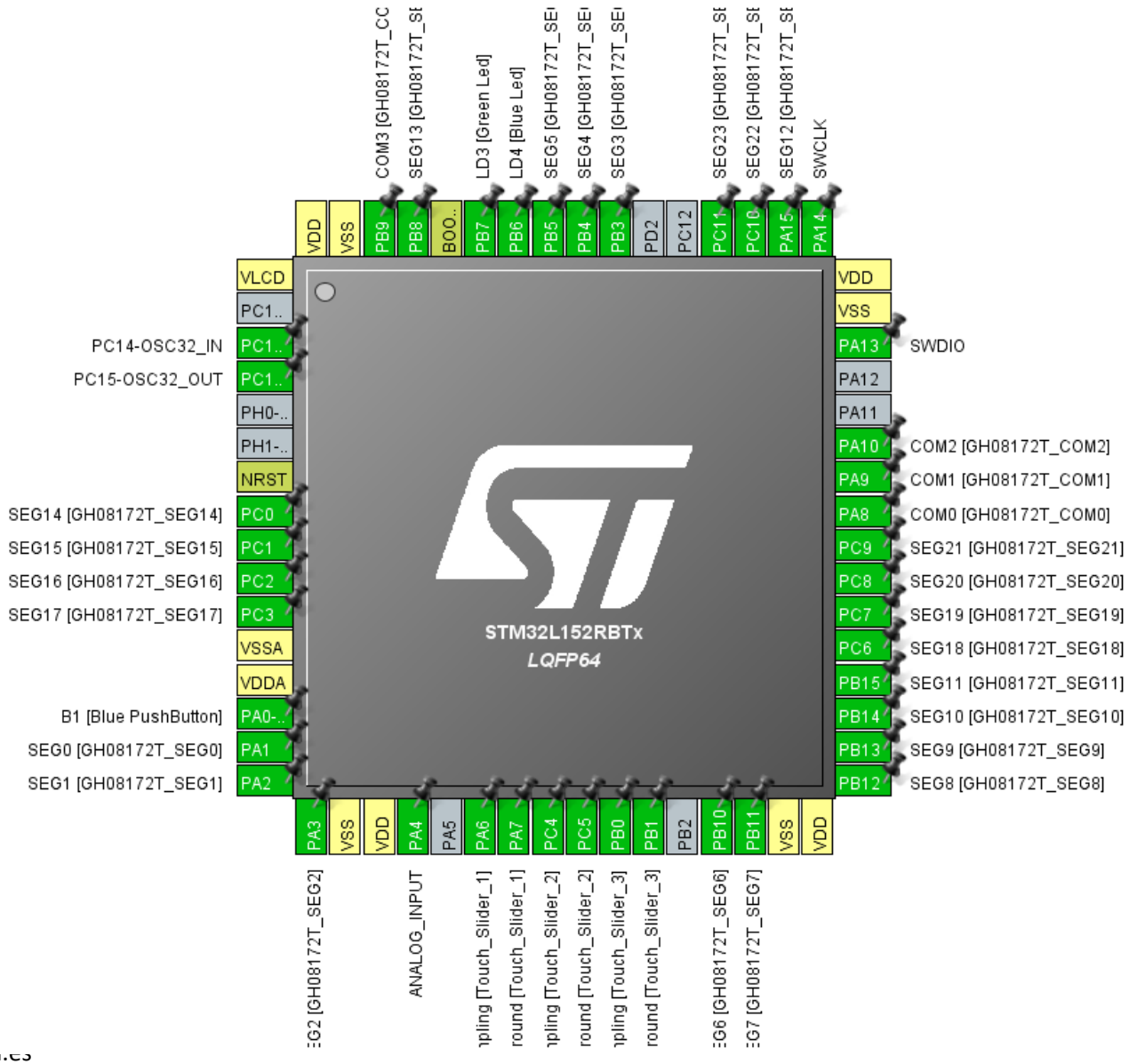
Diagrama de Flujo



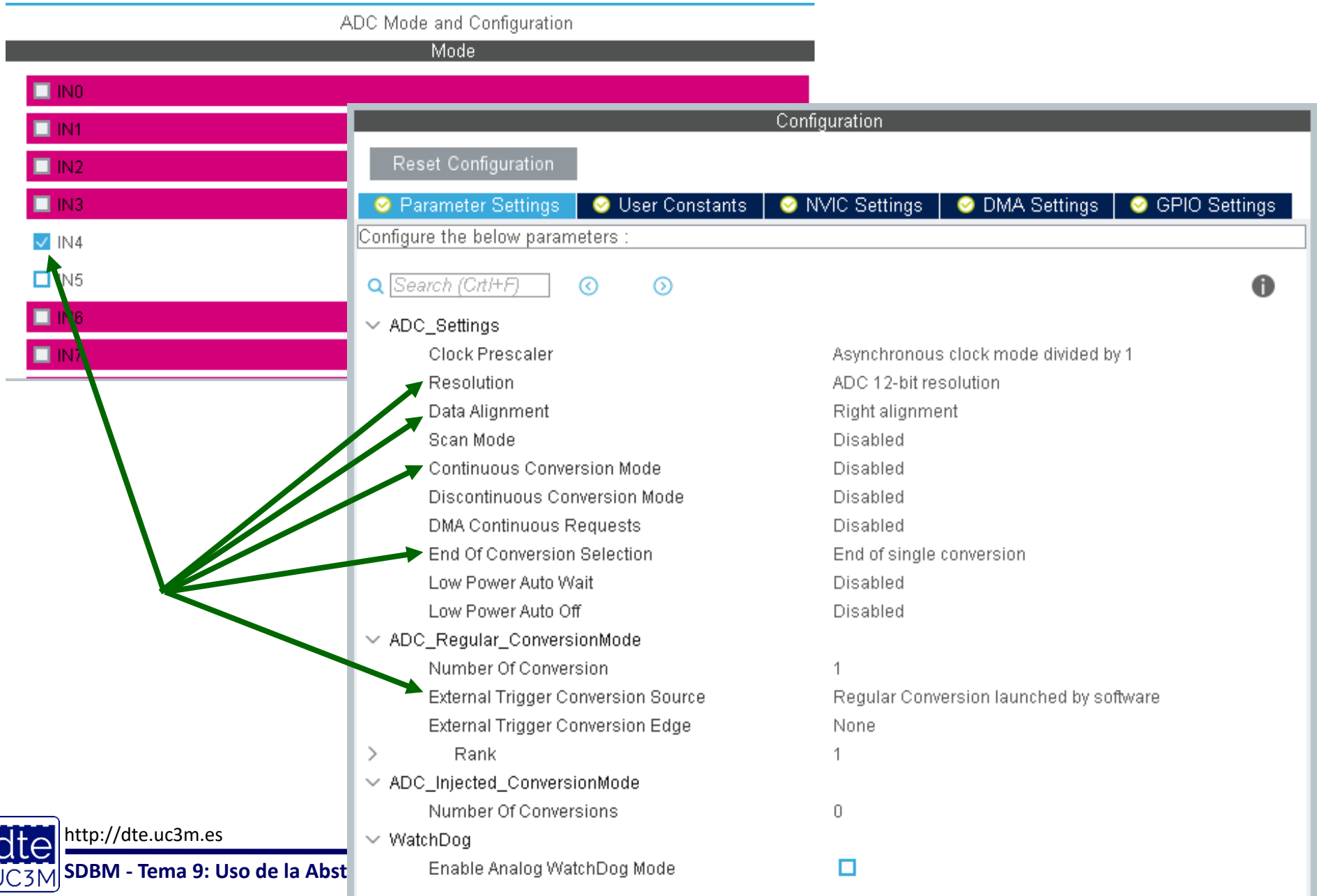
STM32 CubeMX

- Se selecciona la placa Discovery
- Se indica que sí que quiere la configuración por defecto
- Se selecciona el LCD y se configura como se indicó en el Tema 5
- Configuración inicial de pines:
  - PA0 – por defecto para botón
  - PC13 – Reset State para evitar problemas con PA0
  - PA4 – Analog Input (y le podemos cambiar la etiqueta por ANALOG_INPUT)
  - PH0, PH1 – Reset State
- En el Project Manager, dentro de Code Generator, seleccionar:
  - Copy only the necessary library files
  - Set all free pins as analog (to optimize the power consumption)

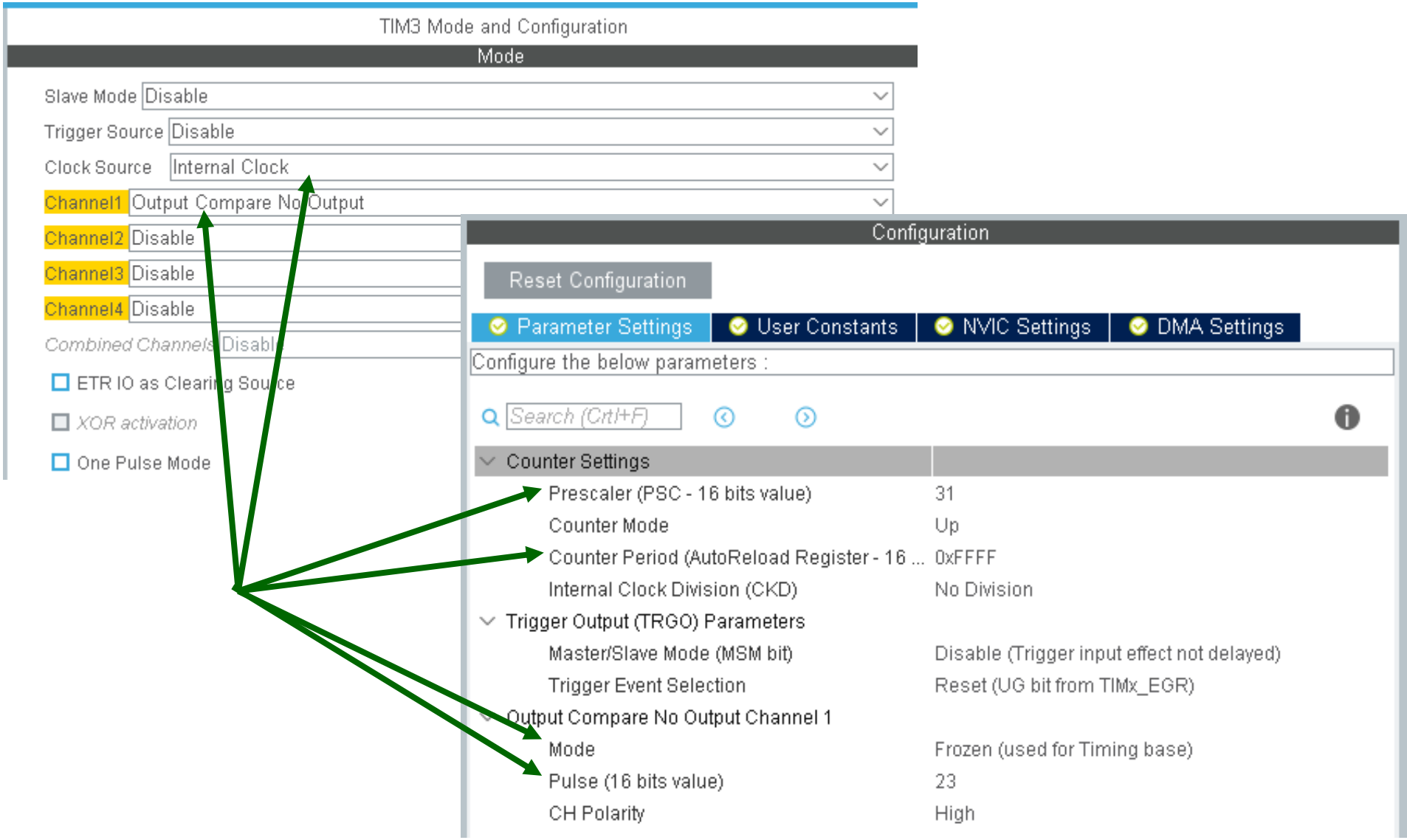
STM32 CubeMX



STM32 CubeMX: ADC



STM32 CubeMX: TIM3



Hardware Abstraction Libraries: ADC

- **Inicialización:**
  - `GPIOA->MODER |= 0x00000300;`
  - `ADC1->CR2 &= ~(0x00000001);`
  - `ADC1->CR1 = 0x00000000;`
  - `ADC1->CR2 = 0x00000400;`
  - `ADC1->SQR1 = 0x00000000;`
  - `ADC1->SQR5 = 0x00000004;`
  - `ADC1->CR2 |= 0x00000001;`
- **Arranque del ADC:**
  - `while ((ADC1->SR&0x0040)==0);`
  - `ADC1->CR2 |= 0x40000000;`
- **Polling de Conversión:**
  - `while ((ADC1->SR&0x0002)==0);`
- **Obtener el Valor:**
  - `valor = ADC1->DR;`

- **Inicialización:**
  - `ADC_HandleTypeDef hadc;`
  - `MX_ADC_Init();`
  - Junto con la configuración dada por CubeMX
- **Arranque del ADC:**
  - `HAL_ADC_Start(&hadc);`
- **Polling de Conversión:**
  - `HAL_ADC_PollForConversion(&hadc, T0000);`
- **Obtener el Valor:**
  - `HAL_ADC_GetValue(&hadc);`

Hardware Abstraction Libraries: TIM

- **Inicialización:**
  - `TIM3->CR1 = 0x0000;`
  - `TIM3->CR2 = 0x0000;`
  - `TIM3->SMCR = 0x0000;`
  - `TIM3->PSC = 31;`
  - `TIM3->CNT = 0;`
  - `TIM3->ARR = 0xFFFF;`
  - `TIM3->CCR1 = 23;`
  - `TIM3->DIER = 0x0000;`
  - `TIM3->CCMR1 = 0x0000;`
  - `TIM3->CCER = 0x0000;`

- **Inicialización:**
  - `TIM_HandleTypeDef htim3;`
  - `MX_TIM3_Init(void);`
  - El Init, ya inicializa también el TOC, por lo que no hay que llamar a su Init
  - Configuración dada por CubeMX
  - Se va a tener que crear una estructura auxiliar para cambiar el comportamiento del TOC (por ejemplo, valor del CCR1)
    - `TIM_OC_InitTypeDef my_sConfigOC = {0};`
    - `my_sConfigOC.OCMode = TIM_OCMODE_TIMING;`
    - `my_sConfigOC.Pulse = TIM_CNT_CNT + 23;`
    - `my_sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;`
    - `my_sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;`
    - `HAL_TIM_OC_ConfigChannel(&htim3, &my_sConfigOC, TIM_CHANNEL_1)`

Hardware Abstraction Libraries: TIM

- **Arranque:**
  - `TIM3->CR1 |= 0x0001;`
  - `TIM3->EGR |= 0x0001;`
  - `TIM3->SR = 0;`
- **Polling y Actualización:**
  - `while ((TIM3->SR & 0x0002)==0);`
  - `TIM3->SR &= ~(0x0002);`
  - `TIM3->CCR1 += 23;`

- **Arranque:**
  - `HAL_TIM_OC_Start(&htim3,TIM_CHANNEL_1);`
- **Polling y Actualización:**
  - `while (!__HAL_TIM_GET_FLAG(&htim3, TIM_FLAG_CC1));`
  - `__HAL_TIM_CLEAR_FLAG(&htim3, TIM_FLAG_CC1);`
  - `my_sConfigOC.Pulse += 23;`
  - `HAL_TIM_OC_ConfigChannel(&htim3, &my_sConfigOC, TIM_CHANNEL_1)`

Código

```
/* USER CODE BEGIN 1 */
uint8_t status_sys = 0;
uint8_t old_status_sys = 0;
uint16_t values[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int ind_values = 0;
int average = 0;
uint8_t text[6];
TIM_OC_InitTypeDef my_sConfigOC = {0};
/* USER CODE END 1 */
```

```
/* USER CODE BEGIN 2 */
BSP_PB_Init(BUTTON_USER, BUTTON_MODE_GPIO);
BSP_LCD_GLASS_Init();
BSP_LCD_GLASS_BarLevelConfig(0);
BSP_LCD_GLASS_Clear();

my_sConfigOC.OCMode = TIM_OCMODE_TIMING;
my_sConfigOC.Pulse = TIM_CNT_CNT + 23;  // Be careful tu update this if pulse changes
my_sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
my_sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_OC_ConfigChannel(&htim3, &my_sConfigOC, TIM_CHANNEL_1) != HAL_OK) {
  Error_Handler();
}
/* USER CODE END 2 */
```

```
 /* USER CODE BEGIN WHILE */
 while (1) {
   if (BSP_PB_GetState(BUTTON_USER) == 1) {  // Button pressed
     status_sys++;
     if (status_sys >= 2) status_sys = 0; // 0 = OFF, 1 = ON
     espera(10000);
     while (BSP_PB_GetState(BUTTON_USER)==1);
   }
 switch (status_sys) {
     case 0:   if (status_sys!=old_status_sys) {
                 old_status_sys = status_sys;
                 HAL_TIM_OC_Stop(&htim3,TIM_CHANNEL_1);
                 BSP_LCD_GLASS_Clear();
               }
               break;
     default:  if (status_sys!=old_status_sys) {
                 old_status_sys = status_sys;
                 my_sConfigOC.Pulse = TIM_CNT_CNT + 23;
                 if (HAL_TIM_OC_ConfigChannel(&htim3, &my_sConfigOC, TIM_CHANNEL_1) != HAL_OK) {
                   Error_Handler(); }
                 HAL_TIM_OC_Start(&htim3,TIM_CHANNEL_1);
               }
               while (!__HAL_TIM_GET_FLAG(&htim3, TIM_FLAG_CC1));
               __HAL_TIM_CLEAR_FLAG(&htim3, TIM_FLAG_CC1);
               my_sConfigOC.Pulse += 23;
               if (HAL_TIM_OC_ConfigChannel(&htim3, &my_sConfigOC, TIM_CHANNEL_1) != HAL_OK) {
                   Error_Handler(); }
               HAL_ADC_Start(&hadc);
               HAL_ADC_PollForConversion(&hadc, 10000);
               values[ind_values] = HAL_ADC_GetValue(&hadc);
               ind_values++;
               if (ind_values >= 10) ind_values = 0;
               average = 0;
               for (int i=0; i<10; i++) average += values[i];
               average = average / 10;
               Bin2Ascii((unsigned short)average, text);
               BSP_LCD_GLASS_DisplayString((uint8_t *)text);
               break;
   }
/* USER CODE END WHILE */
```

Ejemplo con Interrupciones

Descripción y Análisis

- El ejercicio es el mismo que el anterior, pero utilizando interrupciones.
- Recursos a utilizar:
  - GPIOA: Botón USER (PA0)
  - LCD
  - TIM3 en modo TOC sin salida hardware, con pre-escalado de 31 (para saltos de 1us, e incremento de CCR = 23 (para cumplir con 1/44KHz)
    - Interrupción por comparación exitosa en el canal 1
  - ADC en 12 bits y conversión simple
    - Interrupción por finalización de la conversión

Diagrama de Bloques

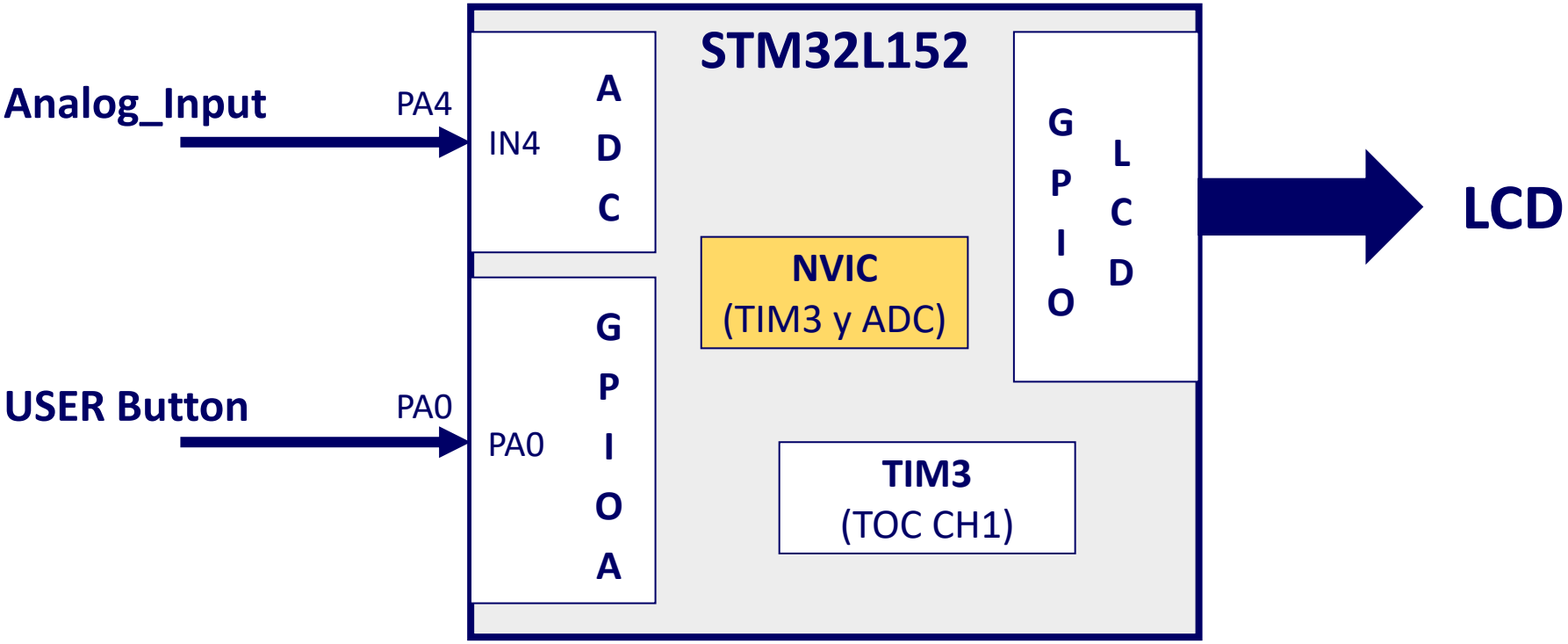


Diagrama de Flujo

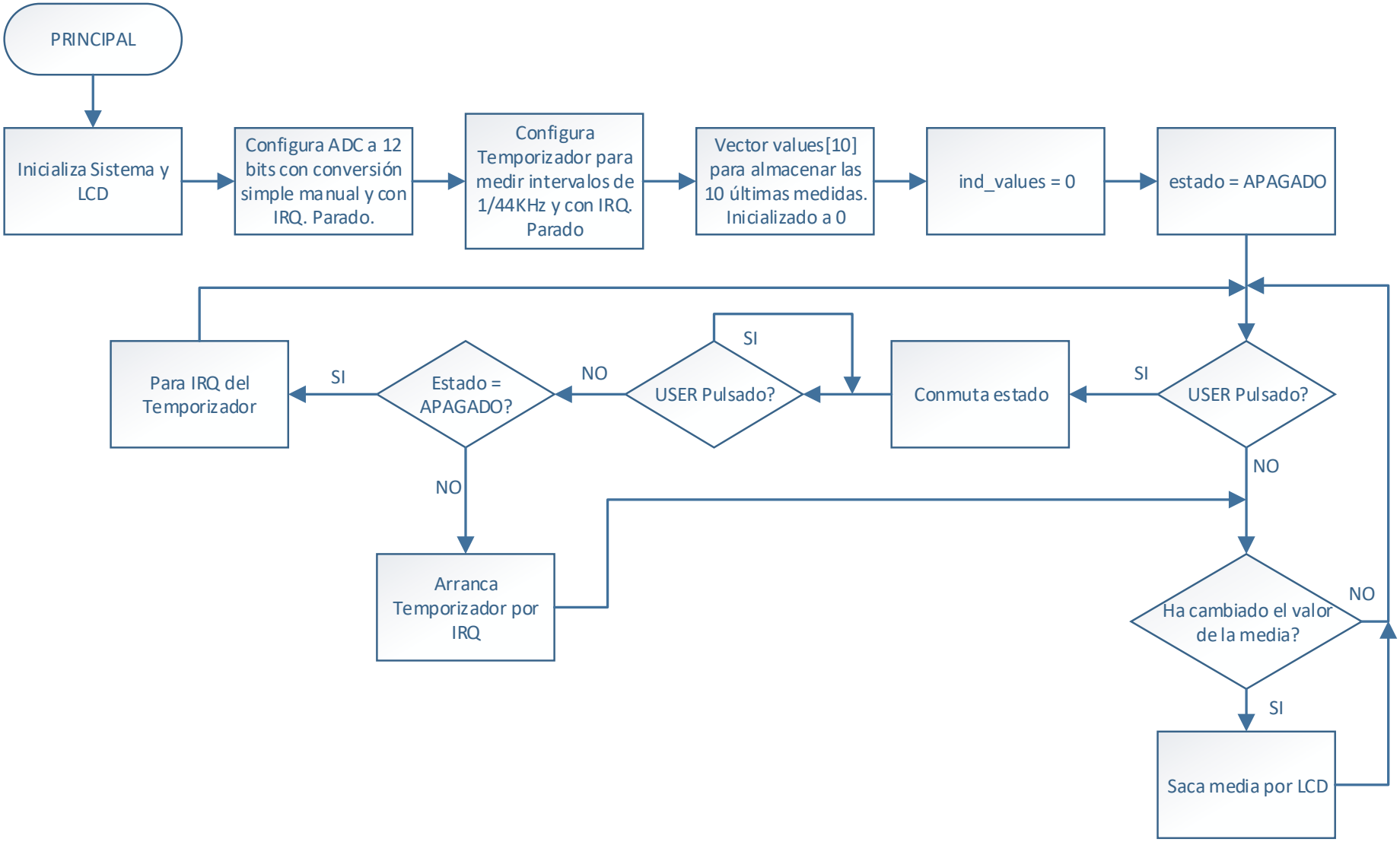
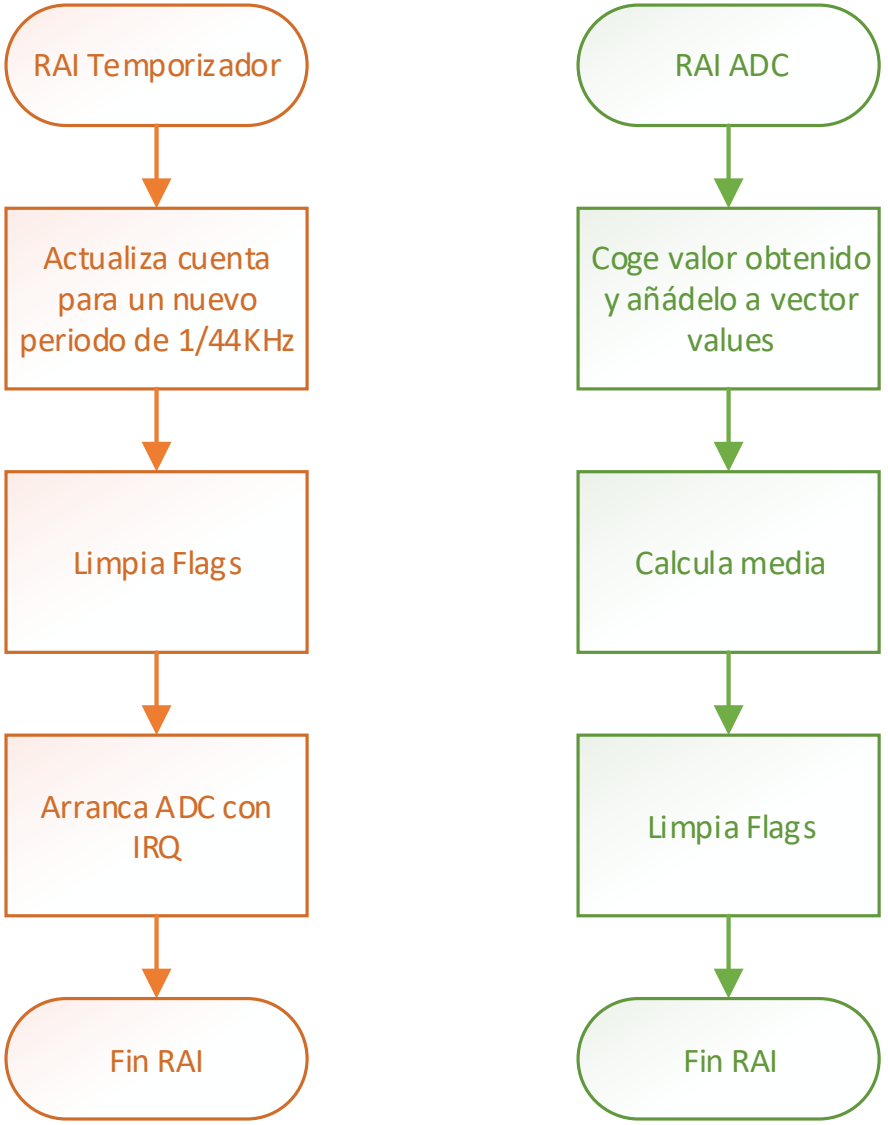


Diagrama de Flujo

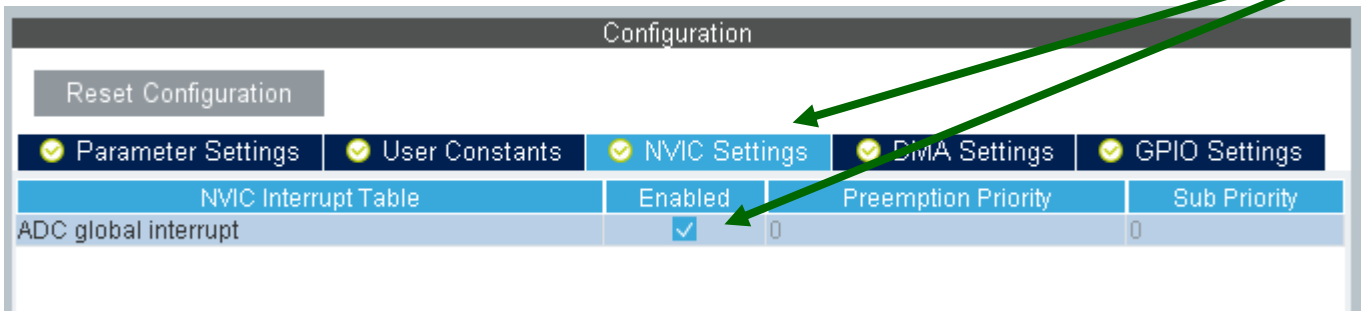


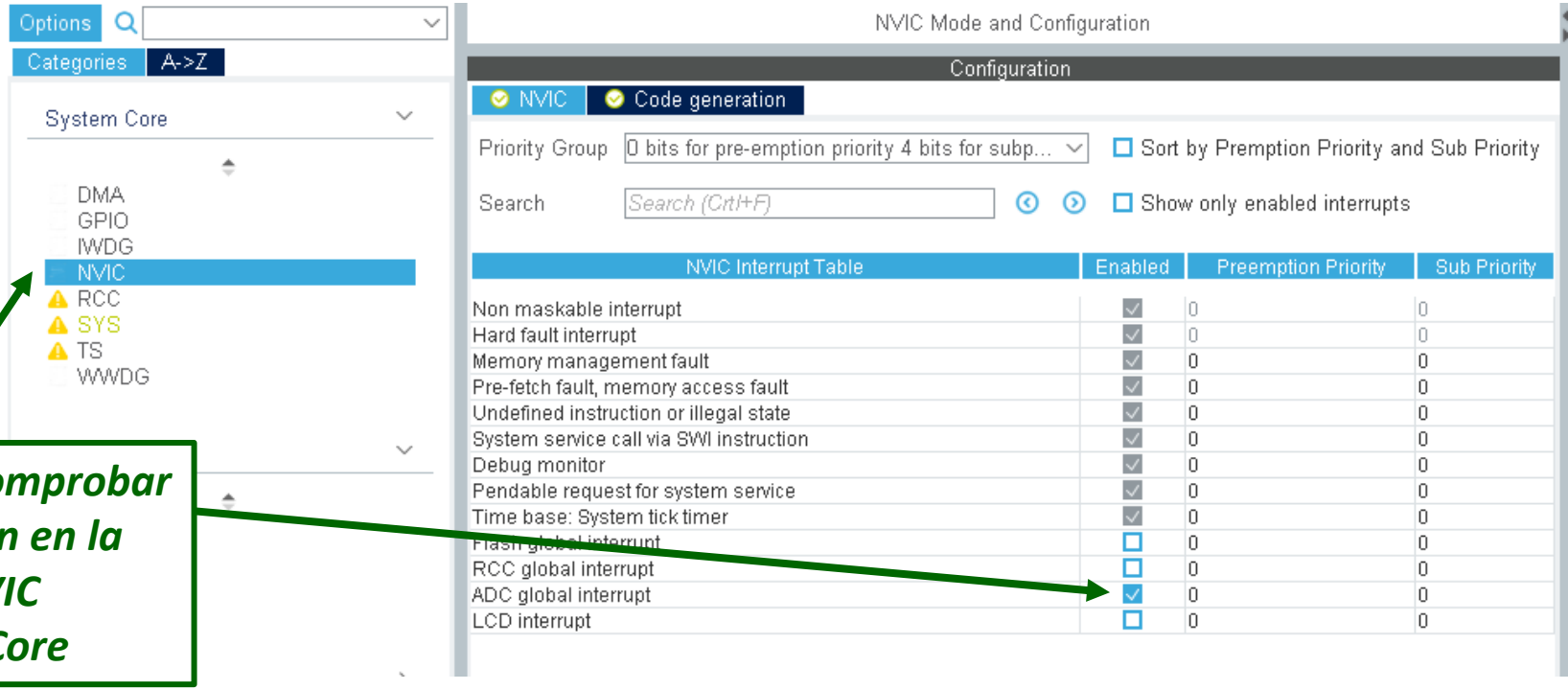
STM32 CubeMX

- Todo es idéntico a lo que se ha realizado SIN Interrupciones
- Adicionalmente, en aquellos periféricos en los que se va a activar la interrupción, hay que activar el NVIC en cada periférico
  - TIM3
  - ADC

STM32 CubeMX: IRQ en ADC

Activa la IRQ en la pestaña NVIC Settings

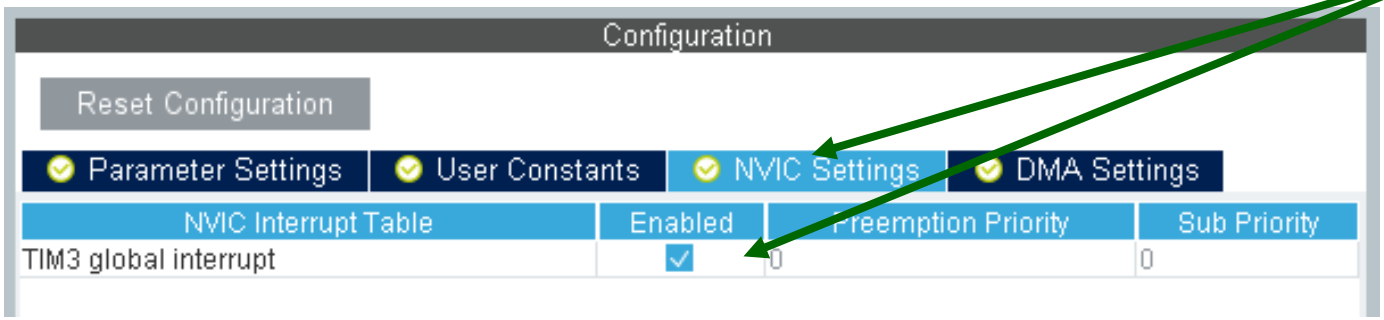




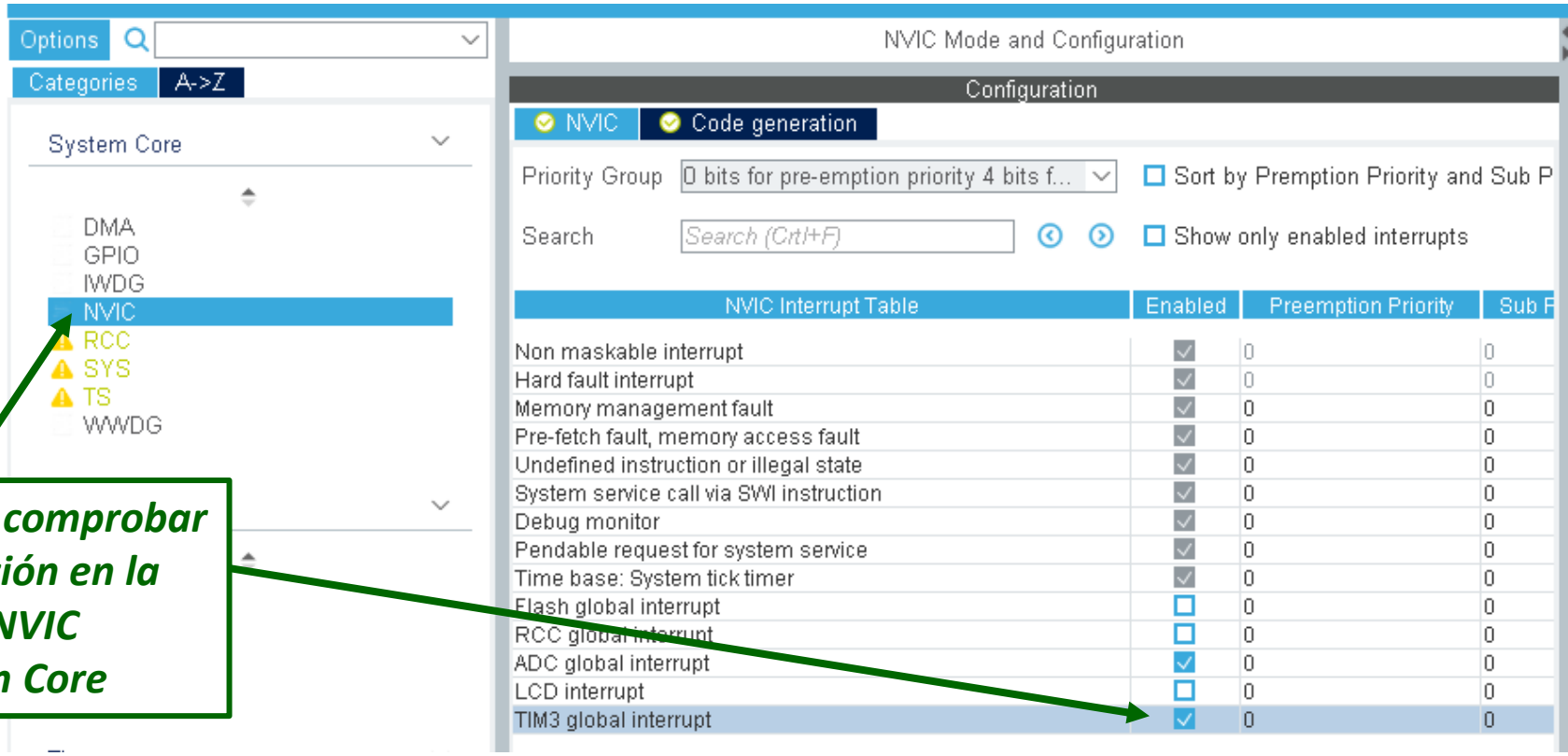
Se puede comprobar la activación en la tabla de NVIC en System Core

STM32 CubeMX: IRQ en TIM3

Activa la IRQ en la pestaña NVIC Settings



Se puede comprobar la activación en la tabla de NVIC en System Core



Hardware Abstraction Libraries: IRQs

- Hay que crear las funciones Callback necesarias
  - Se recomienda que se haga en la sección entre `USER CODE BEGIN 4` y `USER CODE END 4`
- En este caso hay que crear:
  - `void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim) { … }`
  - `void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) { … }`
- Los flags ya se limpian en la RAI creada por el CubeMX
  - Esa RAI es la que llamará a la Callback dependiendo del evento correspondiente
- En los lugares correspondientes, hay que activar la IRQ
  - Por ejemplo, las llamadas `XXXXX_Start()` y `XXXXX_Stop()` hay que cambiarlos por `XXXXX_Start_IT()` y `XXXXX_Stop_IT()`
- Y por supuesto, se han de crear las variables globales necesarias
  - Se recomienda que se haga entre `USER CODE BEGIN PV` y `USER CODE END PV`

Código

```
/* USER CODE BEGIN PV */
uint16_t values[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int ind_values = 0;
int average = 0;
TIM_OC_InitTypeDef my_sConfigOC = {0};
/* USER CODE END PV */
```

```
  /* USER CODE BEGIN 1 */
  uint8_t status_sys = 0;
  uint8_t old_status_sys = 0;
  uint8_t text[6];
  int old_average = 0;
  /* USER CODE END 1 */
```

```
  /* USER CODE BEGIN 2 */
  BSP_PB_Init(BUTTON_USER, BUTTON_MODE_GPIO);
  BSP_LCD_GLASS_Init();
  BSP_LCD_GLASS_BarLevelConfig(0);
  BSP_LCD_GLASS_Clear();
  //Copy from HAL_TIM_Init() as to get sConfigOC values
  my_sConfigOC.OCMode = TIM_OCMODE_TIMING;
  my_sConfigOC.Pulse = TIM_CNT_CNT + 23;  // Be careful tu update this if pulse changes
  my_sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  my_sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim3, &my_sConfigOC, TIM_CHANNEL_1) != HAL_OK) {
    Error_Handler();
  }
  /* USER CODE END 2 */
```

Código

```
/* USER CODE BEGIN WHILE */
while (1)
{
  if (BSP_PB_GetState(BUTTON_USER) == 1) {  // Button pressed
    status_sys++;
    if (status_sys >= 2) status_sys = 0; // 0 = OFF, 1 = ON
    espera(10000);
    while (BSP_PB_GetState(BUTTON_USER)==1);
  }
  switch (status_sys) {
    case 0:   if (status_sys!=old_status_sys) {
                old_status_sys = status_sys;
                HAL_TIM_OC_Stop_IT(&htim3,TIM_CHANNEL_1);
                BSP_LCD_GLASS_Clear();
              }
              break;
    default:  if (status_sys!=old_status_sys) {
                old_status_sys = status_sys;
                my_sConfigOC.Pulse = TIM_CNT_CNT + 23;
                if (HAL_TIM_OC_ConfigChannel(&htim3, &my_sConfigOC, TIM_CHANNEL_1) != HAL_OK) {
                  Error_Handler();
                }
                HAL_TIM_OC_Start_IT(&htim3,TIM_CHANNEL_1);
              }
              if (average != old_average) {
                old_average = average;
                // Show average in LCD
                Bin2Ascii((unsigned short)average, text);
                BSP_LCD_GLASS_DisplayString((uint8_t *)text);
              }
              break;
  }
  /* USER CODE END WHILE */
```

Código

```
/* USER CODE BEGIN 4 */

// Callback for TIM3 IRQ
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim) {
  my_sConfigOC.Pulse += 23;
  if (HAL_TIM_OC_ConfigChannel(&htim3, &my_sConfigOC, TIM_CHANNEL_1) != HAL_OK)
    {
      Error_Handler();
    }
  // Start ADC
  HAL_ADC_Start_IT(&hadc);
}

// Callback for ADC IRQ
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc) {
  values[ind_values] = HAL_ADC_GetValue(hadc);
  ind_values++;
  if (ind_values >= 10) ind_values = 0;
  // Compute average
  average = 0;
  for (int i=0; i<10; i++) average += values[i];
  average = average / 10;
}

/* USER CODE END 4 */
```