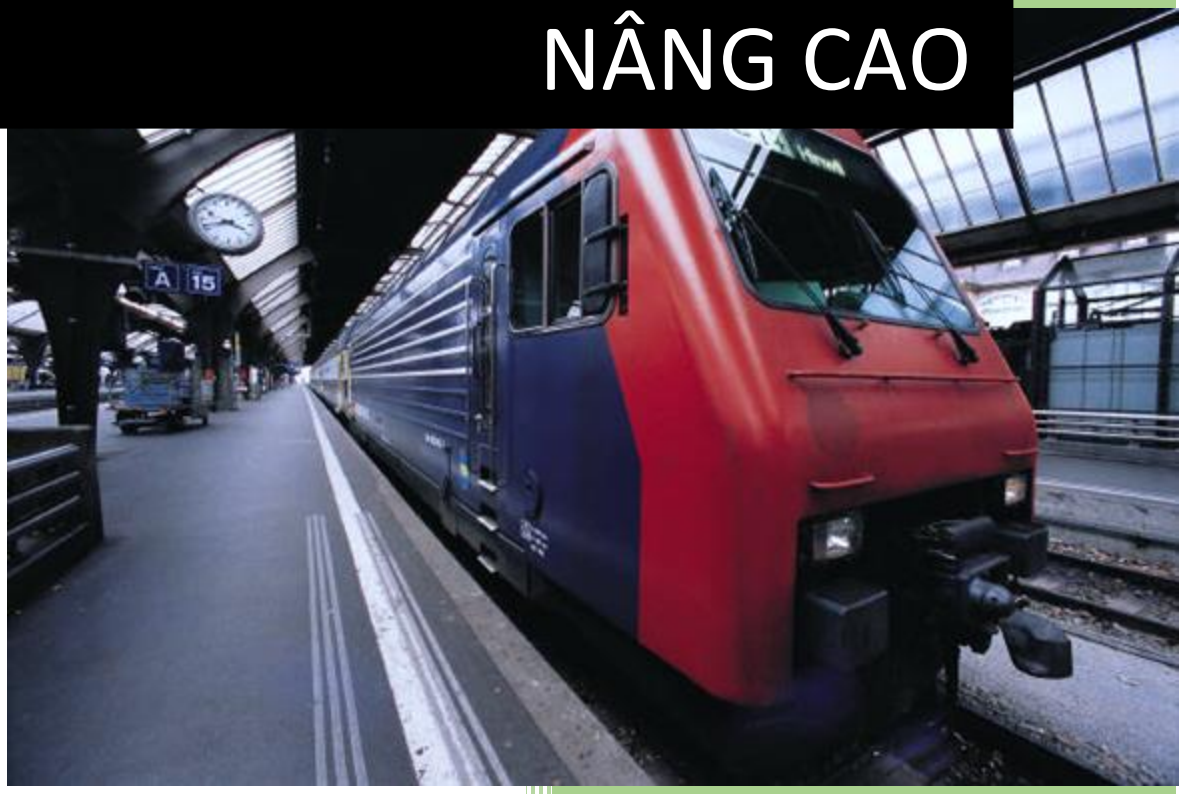
2020

THỰC HÀNH LẬP TRÌNH NHÚNG NÂNG CAO



Biện tập: Mai Cường Thọ

1/1/2020

BÀI 01:

LẬP TRÌNH GIAO TIẾP GPIO CƠ BẢN

Khi mới bắt đầu tìm hiểu, nghiên cứu bất kỳ dòng vi điều khiển nào, GPIO luôn là phần kiến thức đầu tiên mà lập trình viên sử dụng, nghiên cứu.

I. Lý thuyết

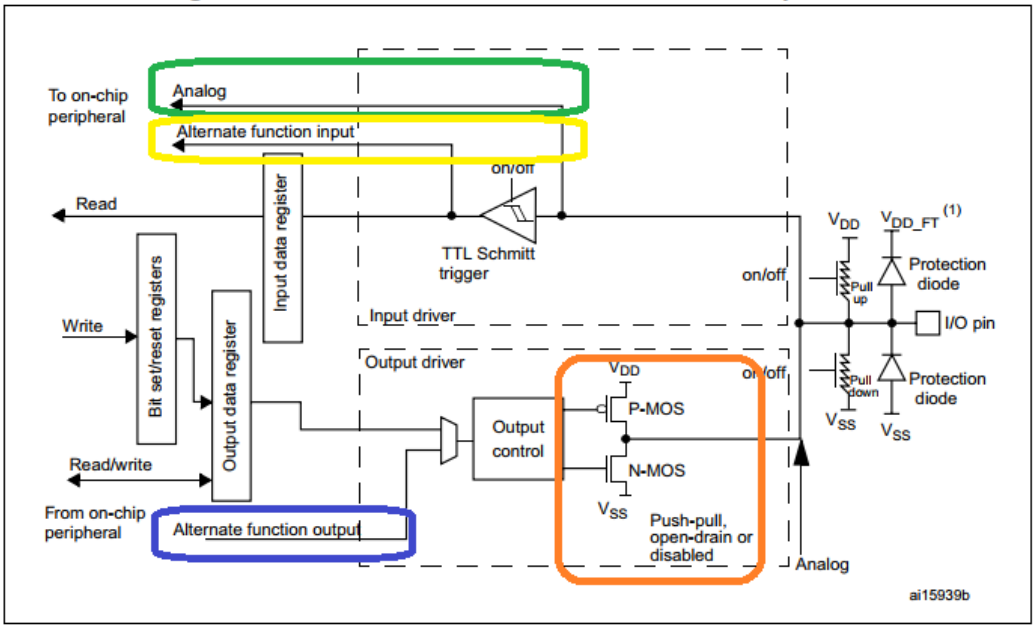
General-purpose Input/Output (GPIO) rất phổ biến, là một chức năng ngoại vi cơ bản của mỗi loại vi điều khiển, bao gồm các chân đầu vào và chân đầu ra, có thể được điều khiển bởi người dùng. Nó tương tự với các dòng vi điều khiển 8bit như AVR, 8051, PIC.

Không như các dòng vi điều khiển 8bit, chỉ có 8 chân IO trên 1 port, ở các vi điều khiển 32bit có đến 16 chân IO trên 1 port.

Cụ thể đối với kit STM32F407VG, có 5 port chính là GPIOA, GPIOB, GPIOC, GPIOD, GPIOE, trên mỗi port có các chân I/O được ký hiệu 0 đến 15.

Sơ đồ cấu trúc mỗi chân GPIO của chip:

Figure 25. Basic structure of a five-volt tolerant I/O port bit



1. V_{DD_FT} is a potential specific to five-volt tolerant I/Os and different from V_{DD}.

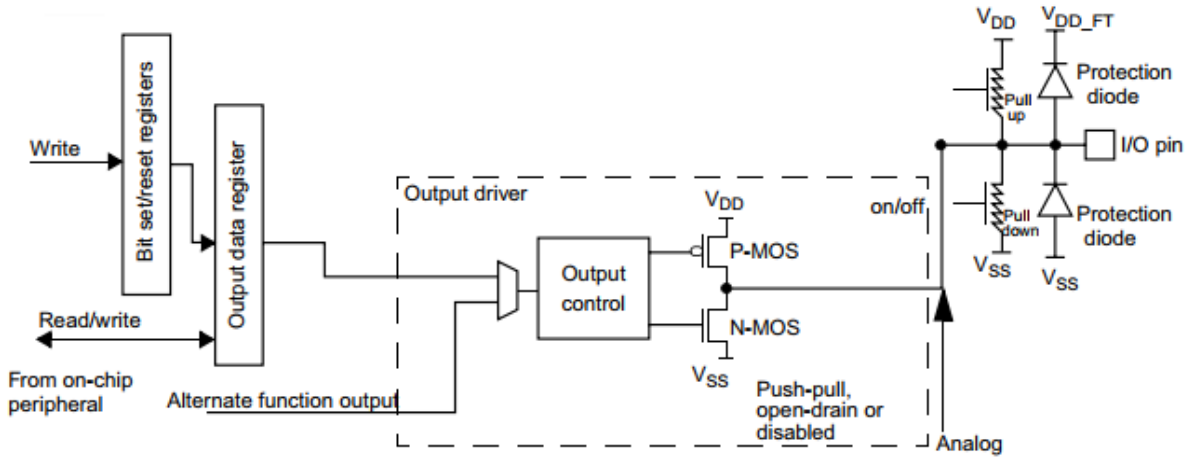
Có 2 khối điều khiển chính của mỗi GPIO (2 khối được vẽ đứt trong hình), đó chính là :

- Input driver
- Output driver

GPIO bao gồm 8 chức năng chính sau đây :

- Input floating
- Input pull-up
- Input-pull-down
- Analog
- Output open-drain with pull-up or pull-down capability
- Output push-pull with pull-up or pull-down capability
- Alternate function push-pull with pull-up or pull-down capability
- Alternate function open-drain with pull-up or pull-down capability

Mặc định khi lập trình viên không cấu hình gì, trạng thái của các chân I/O sẽ là Input Floating. Trong bài viết này, chúng ta sẽ sử dụng chức năng Output của GPIO, dưới đây là sơ lược về cấu trúc phần cứng của khối Output.



1. Các thanh ghi quan trọng của GPIO

Mỗi chân GPIO đều có 2 thanh ghi cơ bản cấu hình 32 bit là (GPIOx_CRL – Control Register Low, GPIO_CRH – Control Register High)

Chúng ta quan tâm đến 2 thanh ghi sau:

- **GPIO port bit set/reset register (GPIOx_BSRR)**

Thanh ghi này dùng để cấu hình các chân ở mức **set** (mức cao) hoặc **reset** (mức thấp)

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

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Resets the corresponding ODRx bit

Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Sets the corresponding ODRx bit

GPIO port output data register (GPIOx_ODR)

Dữ liệu sau khi các bit đã được set/reset ở thanh ghi trên sẽ được truyền sang thanh ghi dữ liệu đầu ra 32bit (GPIOx_ODR: Output Data Register) và truyền đến khối điều khiển để xuất mức tín hiệu cho chân I/O. Ngoài ra đối với thanh ghi này, chúng ta có thể đọc dữ liệu để xem trạng thái hiện tại của các chân IO đang ở mức "1" hoặc mức "0".

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
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

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

These bits can be read and written by software.

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

2. Xuất tín hiệu output thông qua khối CMOS

Khi một I/O pin được cấu hình hoạt động với chức năng Output thì khối điều khiển Output driver được sử dụng với các chế độ **: Open drain mode** hoặc **Push-Pull mode**:

- **Open drain mode**: Ở chế độ này, mạch sẽ không sử dụng P-MOS (luôn khóa) và chỉ sử dụng N-MOS. Khi một giá trị bit của thanh ghi ODR bằng 0 sẽ làm N-MOS dẫn, lúc này chân vi điều khiển được kéo xuống GND và có mức logic thấp (mức 0). Một giá trị bit của thanh ghi ODR bằng 1 sẽ làm N-MOS đóng, chân tương ứng sẽ ở trạng tháng Hi-Z (trở kháng cao).

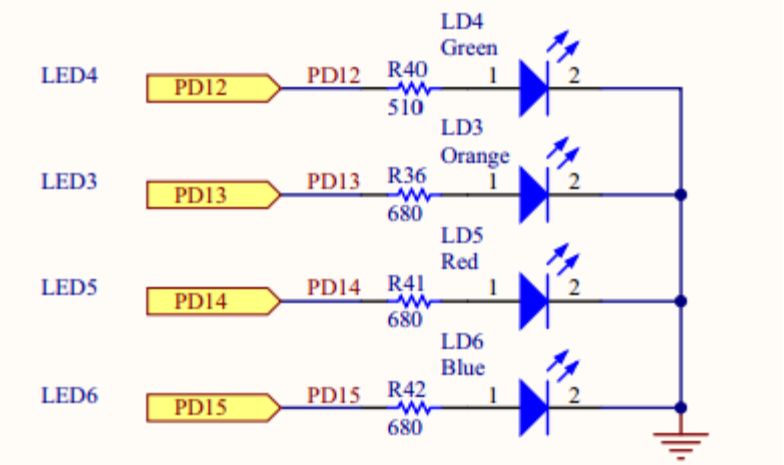
- **Push-pull mode** : Ở chế độ này mạch sẽ sử dụng cả P-MOS và N-MOS. Một giá trị bit của thanh ghi ODR bằng 0 sẽ làm N-MOS dẫn, P-MOS ngưng dẫn, lúc này chân vi điều khiển có mức thấp (được nối với GND). Một giá trị bit của thanh ghi ODR bằng 1 sẽ làm N-MOS ngưng dẫn và P-MOS dẫn. Lúc này chân vi điều khiển có mức cao (mức logic 1 – được nối với VDD).

Như vậy, để điều khiển giá trị logic của 1 I/O pin được cấu hình hoạt động với chức năng Output, chúng ta cần ghi giá trị logic vào Output Data Register (GPIOx_ODR). Bit tương ứng của thanh ghi sẽ điều khiển pin ở vị trí tương ứng.

Ví dụ: bit thứ *0* của thanh ghi GPIOB-ODR sẽ điều khiển chân PB0.

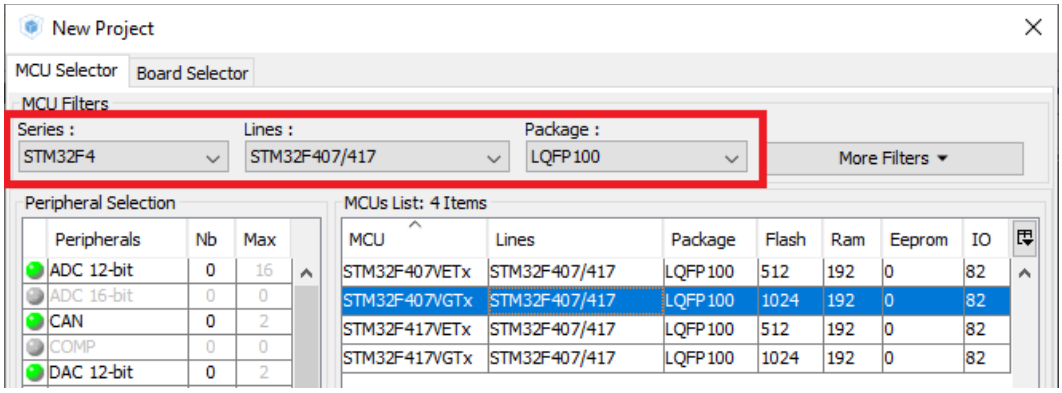
II. Lập trình

Như đã giới thiệu ở trên, kit **STM32F407VG** có 5 Port chính **A, B, C, D, E**, mỗi port này có 16 chân và được ký hiệu từ 0 đến 15. Để quan sát được sự thay đổi tín hiệu trên các chân, cách đơn giản nhất chúng ta kết nối chân với l đèn led. Trên kit này, nhà sản xuất đã kết nối sẵn cho chúng ta 4 chân với 4 đèn Led khác nhau. Đó là các chân **PD12, PD13, PD14** và **PD15**.



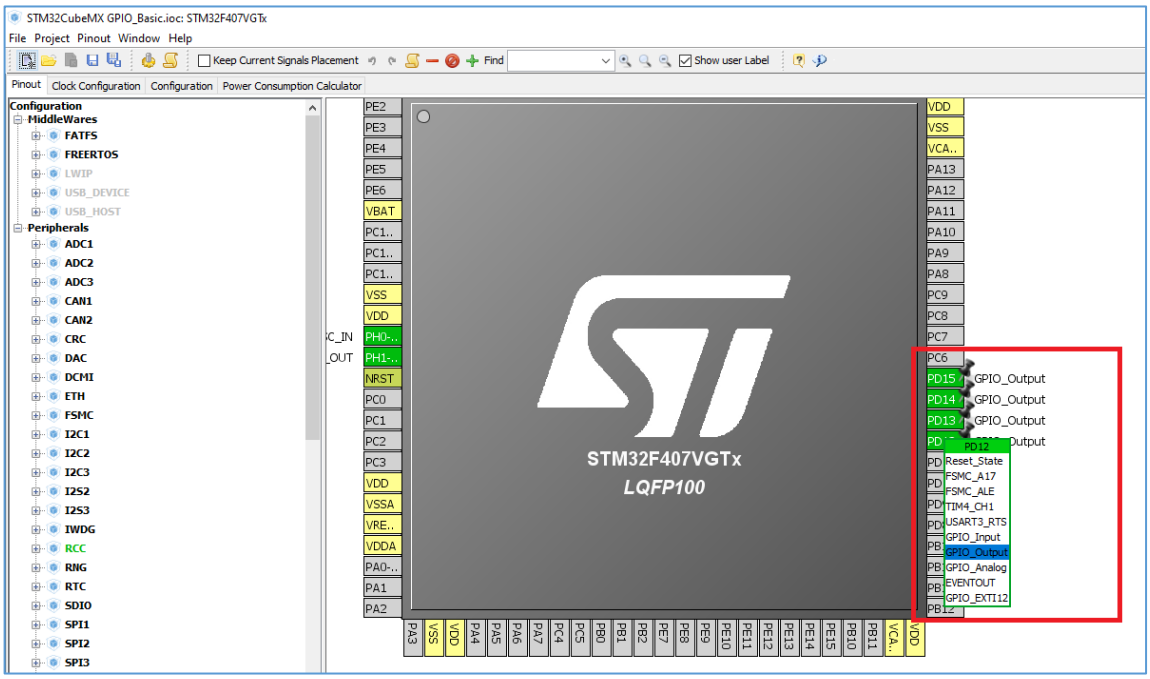
1, Cấu hình với CubeMX:

❖ *Bước 1:* Khởi động CubeMX và chọn dòng vi điều khiển muốn sử dụng



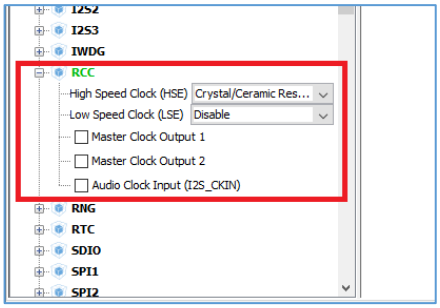
❖ *Bước 2:* **Chọn các cổng/chân sẽ xuất dữ liệu**

Tab **PINOUT**, chọn các chân **PD12, PD13, PD14, PD15** có chức năng "**GPIO_Output**".



❖ *Bước 3.* **Chọn nguồn xung cho Chip**

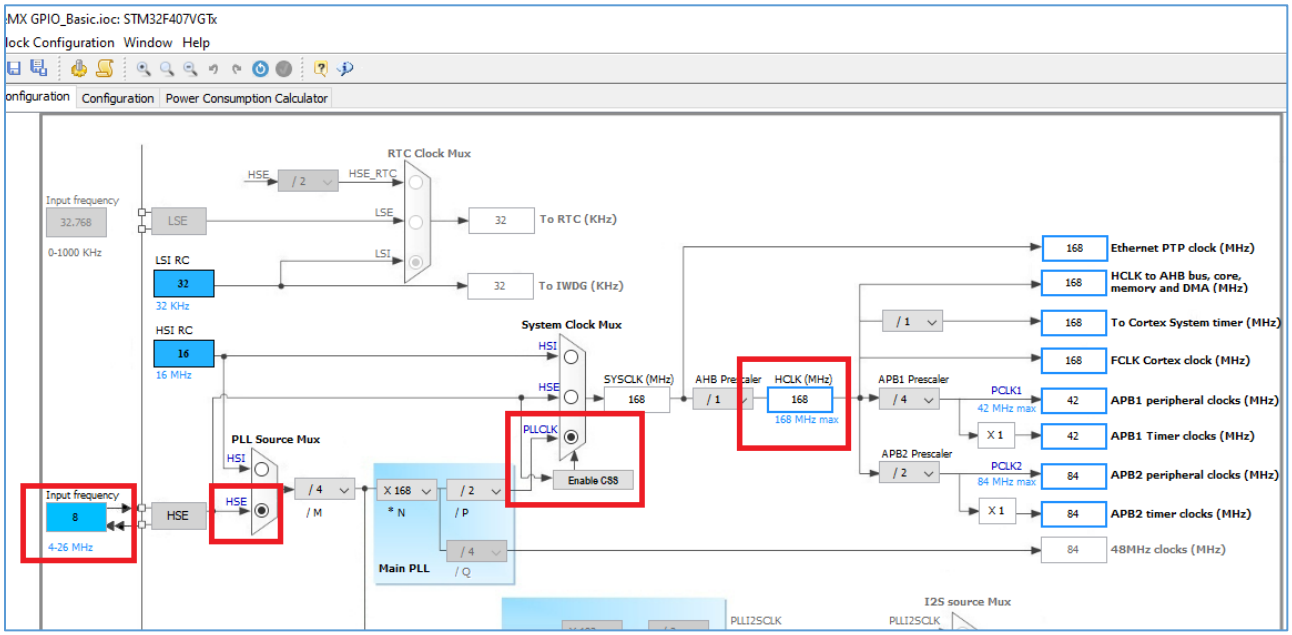
- *3.1.* Cấu hình chip hoạt động với thạch anh ngoại được gắn sẵn trên board mạch **RCC → High Speed Clock (HSE)** và chọn "**Crystal/Ceramic Resonator**"



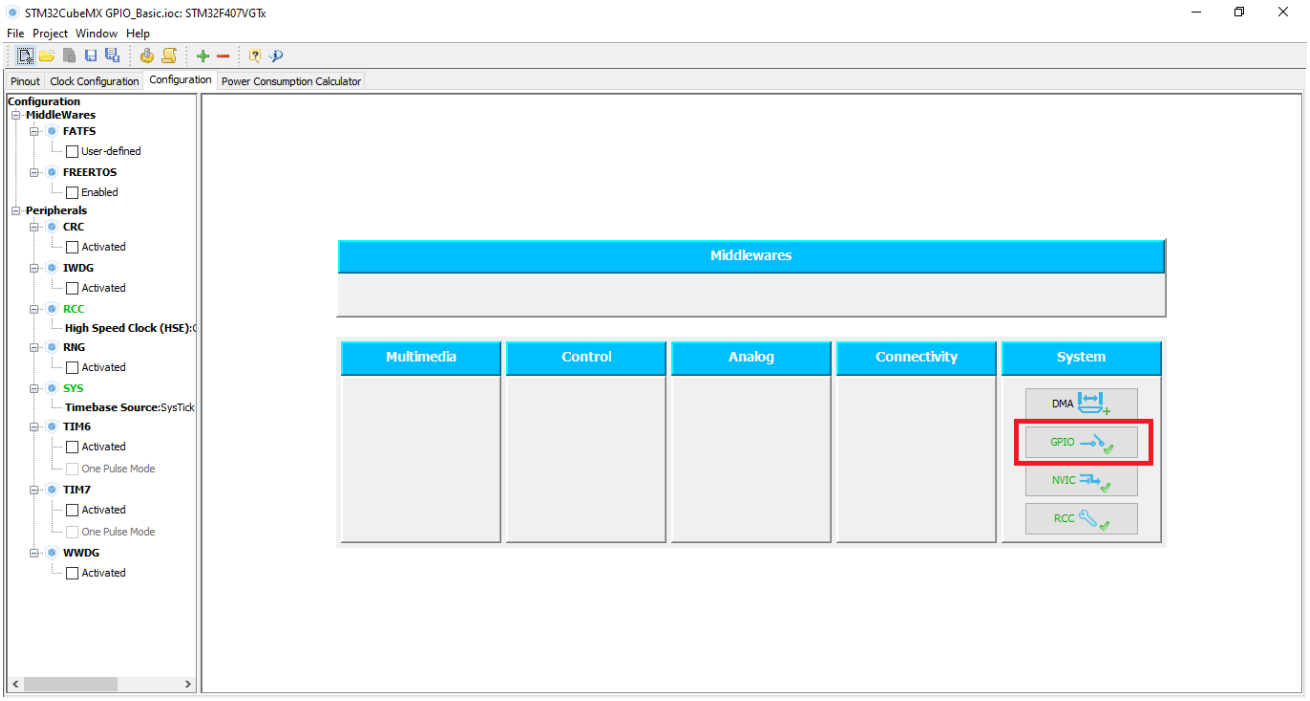
❖ *Bước 4.* ***Cấu hình tần số cho chíp và ngoại vi –*** *Tab* **Clock Configuration**

Tiếp theo, chúng ta tìm đến mục "Clock Configuration" và tích chọn mục HSE (nguồn thạch anh ngoài), tín hiệu clock sẽ đi qua bộ nhân tần PLLCLK giúp chip đạt được ở tần số hoạt động tối đa.

Đặt **Input frequency** = 8 (thạch anh hàn sẵn trên board là loại 8Mhz). Sau đó chúng ta điền "168" tại mục "HCLK" (đây là tần số hoạt động tối đa của chip) và ấn Enter, đợi cho CubeMX tự tính toán các thông số còn lại.

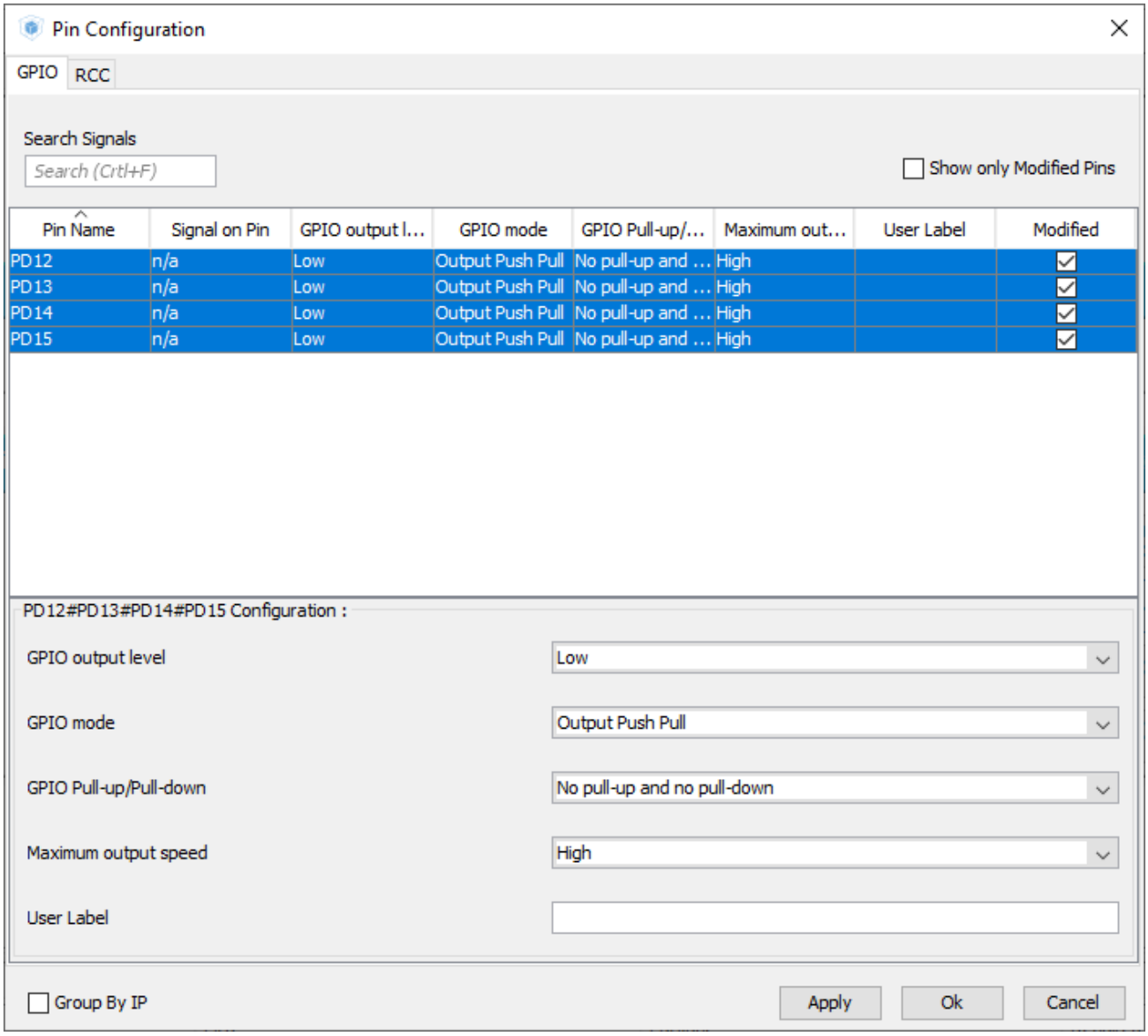


❖ *Bước 5.* ***Cấu hình cho các chân GPIO*** **-** Tab "**Configuration**"



Chúng ta chọn các thông số cho các GPIO như dưới đây :

- **GPIO output level**: Low (cấu hình ban đầu cho các chân đang ở mức thấp)
- **GPIO mode**: Output Push Pull
- **GPIO Pull-up/Pull-down**: No pull-up and no pull-down (không cần điện trở kéo lên và kéo xuống)
- **Maximum output speed**: High



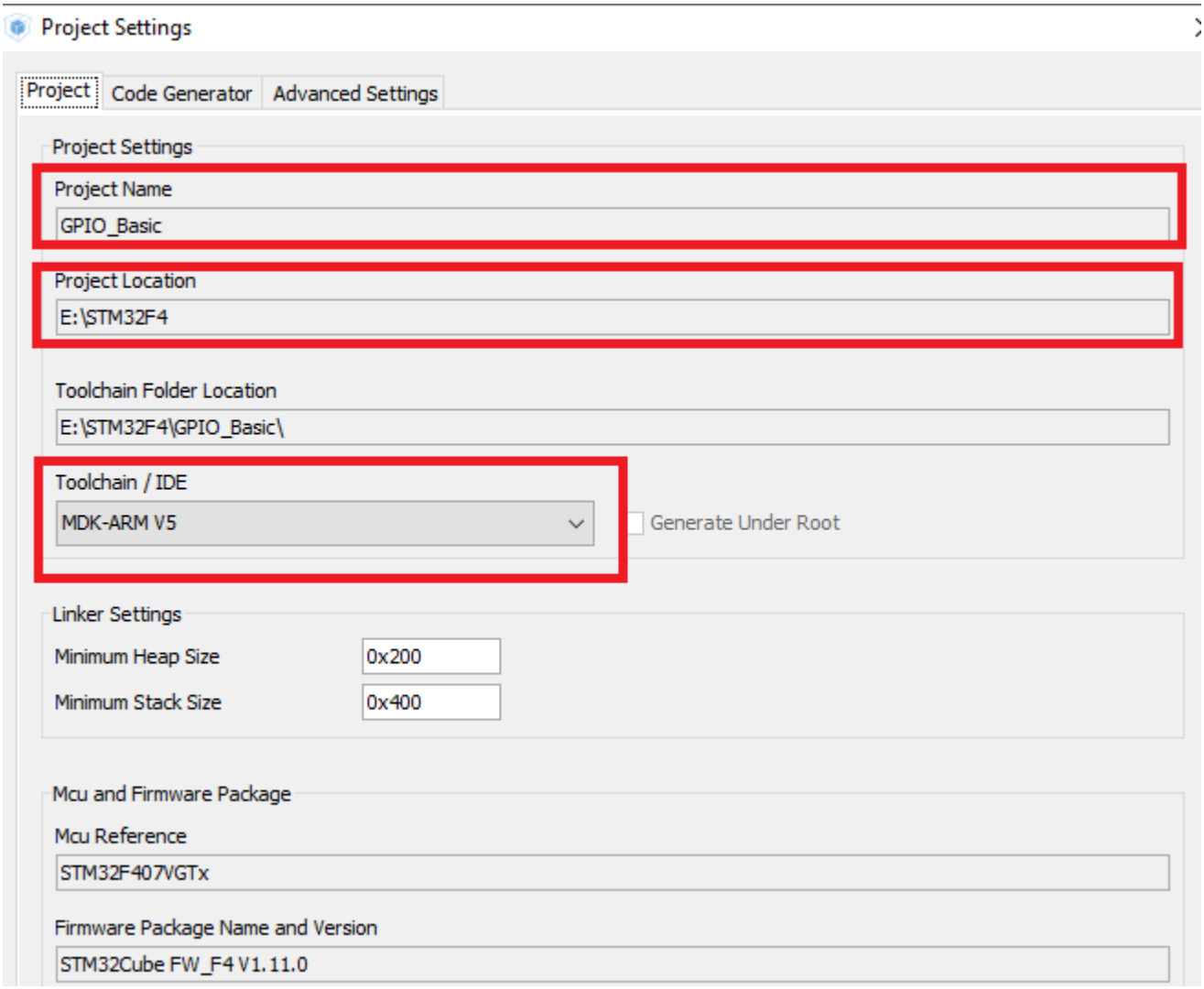
❖ *Bước 6.* ***Cuối cùng là Setting cho Project và sinh code:***

6.1*. Điền các thông tin*

Project Name: tên muốn đặt cho project.

Project Location: Vị trí lưu project

Toolchain/IDE: Bộ công cụ lập trình, ví dụ MDK-ARM V5



6.2. Tùy chọn sinh code

Tab "**Code Generator**", chọn "**Copy only the necessary library files**" để trong project của chúng ta chỉ có những thư viện cần thiết, điều này sẽ giúp tiết kiệm đáng kể dung lượng.

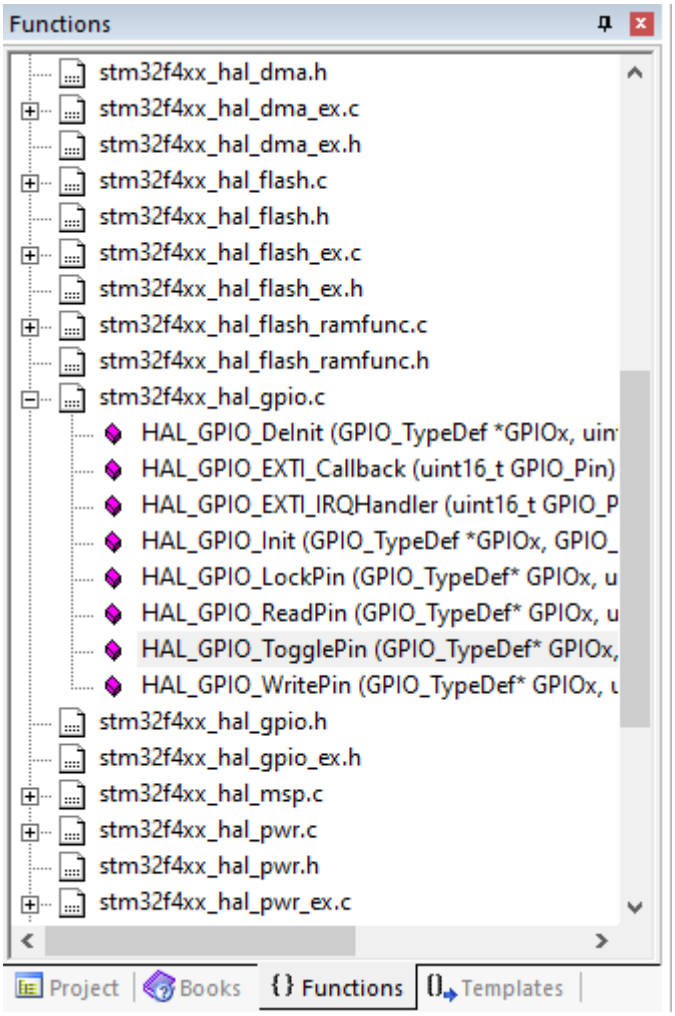
6.3. Sinh code: Generate Code và Open Project sau khi CubeMX sinh code xong

2. Lập trình với KeilC (MDK- ARM5)

Tại mục Functions, trong file "**stm32f4xx_hal_gpio.c**" sẽ chứa các hàm cơ bản để điều khiển GPIO.

HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin) cho phép đảo trạng thái của l chân bất kỳ. Ở đây mình sẽ truyền vào 2 tham số, thứ nhất là Port cần sử dụng (GPIOx) và tham số thứ 2 là chân IO cần sử dụng (GPIO_Pin) cụ thể là:

HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15);



Ngoài ra có thể xuất mức "1" hoặc mức "0" tại chân IO thông qua hàm:

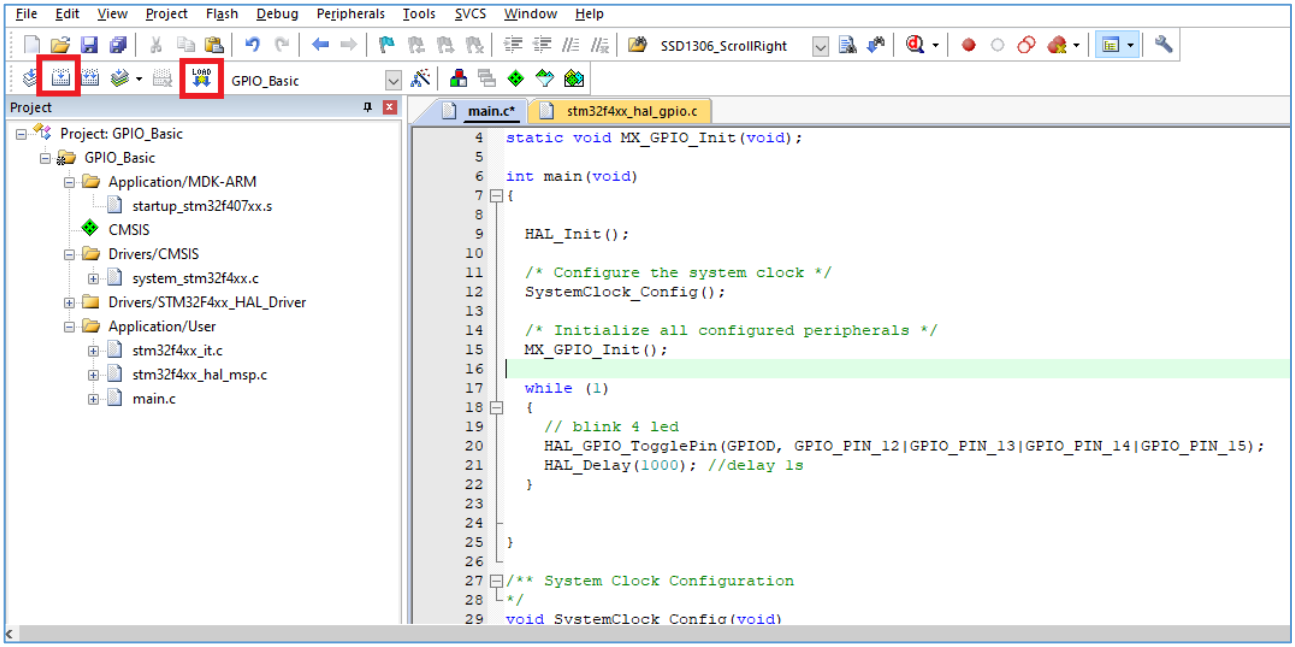
```
HAL_GPIO_WritePin(GPIOD, GPIO_PIN_13, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOD, GPIO_PIN_13, GPIO_PIN_RESET);
```

Hai hàm này sẽ đặt trạng thái cho 1 chân bất kỳ, "**GPIO_PIN_SET**" là mức "1", "**GPIO_PIN_RESET**" là mức "0".

Hàm while(1) sẽ là 1 vòng lặp vô hạn, trong hàm này mình sẽ viết code để đảo trạng thái chớp tắt led liên tục chu kỳ 1s.

```
while (1) {
// đảo trạng thái 4 led
  HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15);
  HAL_Delay(1000); //delay 1s
}
```

Chúng ta build chương trình (**F7**) và nạp code xuống kit (**F8**)



Nhấn Reset button và cùng quan sát led trên kit hoạt động.

BÀI 02

NGẮT NGOÀI VÀ ƯU TIÊN NGẮT TRÊN STM32F4

***NVIC** – **N**ested **V**ectored **I**nterrupt **C**ontroller là bộ điều khiển xử lý ngắt có trong MCU STM32F407VG. Việc lập trình sử dụng ngắt là một kĩ năng rất quan trọng đối với lập trình vi điều khiển. Nếu không có ngắt, chương trình của chúng ta sẽ thực hiện tuần tự từ trên xuống dưới, ngắt sẽ giúp chương trình xử lý theo sự kiện, đáp ứng được các sự kiện như thay đổi mức logic từ 1 chân I/O (ngắt ngoài), nhận 1 ký tự (ngắt nhận UART),…*

*Trong phần này chúng ta cùng tìm hiểu về ngắt ngoài (**EXTI – External interrupt**) cùng với vi điều khiển STM32F407VG.*

I. Lý thuyết

Ngắt (Interrupt) **là gì** - như tên của nó, là một số sự kiện khẩn cấp bên trong hoặc bên ngoài bộ vi điều khiển xảy ra, buộc vi điều khiển tạm dừng thực hiện chương trình hiện tại, phục vụ ngay lập tức nhiệm vụ mà ngắt yêu cầu – nhiệm vụ này gọi là trình phục vụ ngắt (ISR: Interrupt Service Routine).

Một số ngắt phổ biến trên vi điều khiển:

– *Ngắt ngoài*: Sự kiện là khi sự thay đổi sườn tín hiệu sườn lên, sườn xuống, hoặc cả 2.
– *Ngắt UART*: Sự kiện là khi buffer nhận đủ 1 byte dữ liệu
– *Ngắt ADC*: Sự kiện là khi hoàn thành việc chuyển đổi ADC
– *Ngắt Timer*: Sự kiện là khi khi tràn thanh ghi đếm, hoặc khi giá trị đếm bằng với thanh ghi so sánh

Một số tính năng của NVIC với STM32F407 :

- 82 kênh ngắt
- 16 mức ưu tiên ngắt (có thể lập trình được)
- Quản lý, điều khiển năng lượng cho vector ngắt
- Thực hiện trên các thanh ghi điều khiển hệ thống
- Đỗ trễ thấp, xử lý ngắt cực kỳ nhanh

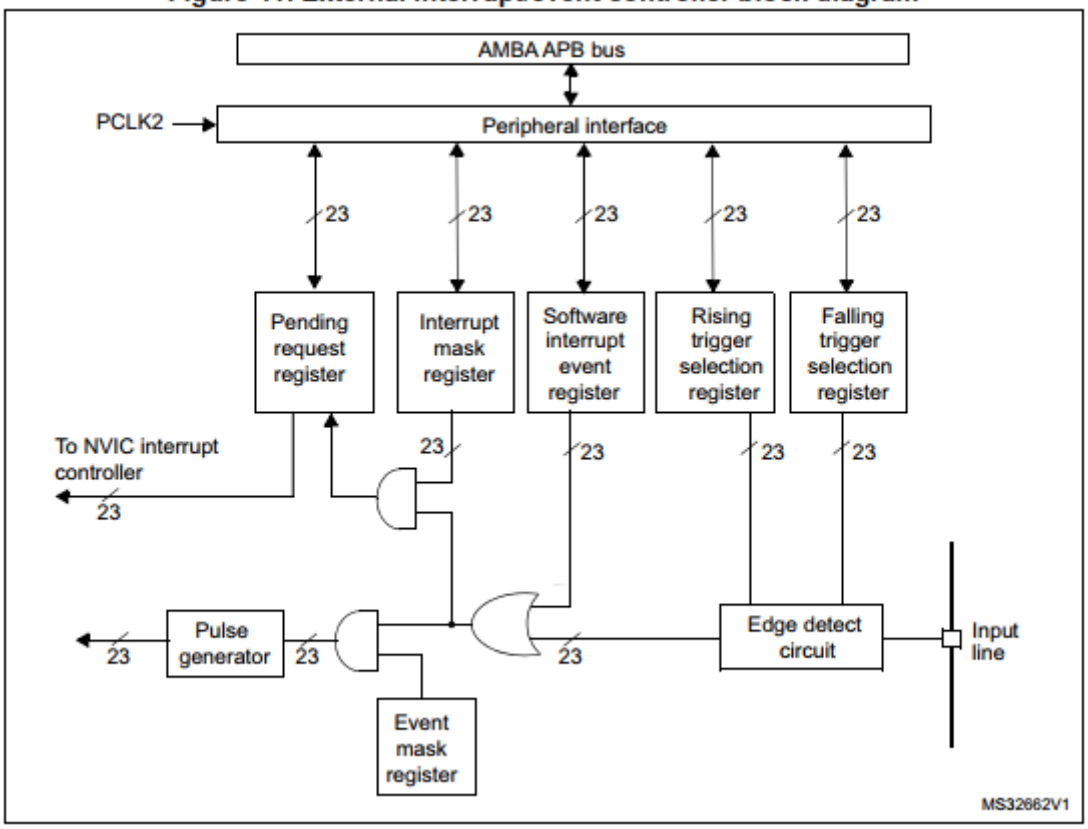
Một số tính năng của ngắt ngoài trên STM32F407:

Kích hoạt độc lập và mask cho mỗi line sự kiện/ngắt.

- Có bit trạng thái (status) riêng cho mỗi line ngắt
- Có thể có tối đa 23 sự kiện/ ngắt
- Kiểm tra tín hiệu ngoài có độ rộng xung nhỏ hơn clock trên APB2

Sơ đồ khối của các khối điều khiển EXTI

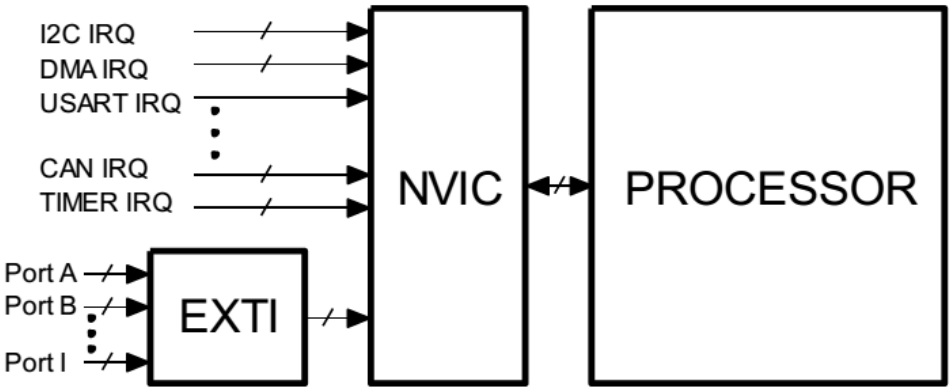
Figure 41. External interrupt/event controller block diagram



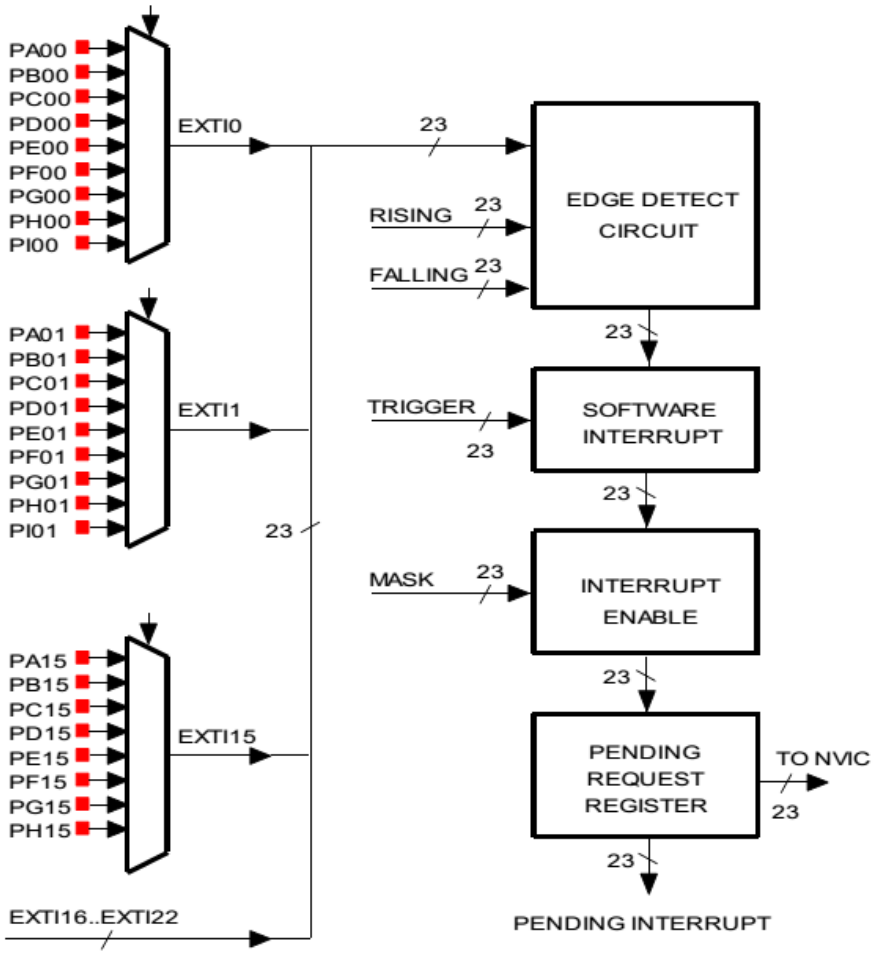
Cấu hình với thư viện chuẩn của ST. Có 2 loại ngắt ngoài chính đó là **ngắt ngoài trên các chân điều khiển ở dạng thông thường** và **ngắt ngoài trên các ứng dụng** như : PVD, RTC, USB, Ethernet.

EXTI nằm trong 1 phần của NVIC, bao gồm 23 bộ phát hiện sự kiện, từ đó khởi tạo nên các yêu cầu ngắt. Mỗi đường đầu vào có thể được cấu hình độc lập để lựa chọn kiểu là *interrupt* hay *event* và *trigger event* tương ứng (rising, falling hoặc cả 2). Mỗi đường ngắt cũng có thể được che một cách độc lập.

EXTI được kết nối với bộ xử lý ngắt lồng nhau NVIC như sau:



Bộ điều khiển ngắt ngoại EXTI xử lý tất cả các tín hiệu yêu cầu ngắt đến từ tất cả các chân của vi điều khiển. Ngoài ra nó còn xử lý các yêu cầu ngắt đến từ các nguồn khác. Các yêu cầu ngắt được phân thành 23 đường ngắt khác nhau, trong đó các yêu cầu đến từ chân 0 của tất cả các port được xử lý trên line 0, các yêu cầu đến từ chân 1 của tất cả các port được xử lý trên line 1…



7 đường ngắt EXTI còn lại được nối như sau:

- EXTI line 16 được nối vào PVD output
- EXTI line 17 được nối vào RTC Alarm event
- EXTI line 18 được nối vào USB OTG FS Wakeup event
- EXTI line 19 được nối vào Ethernet Wakeup event
- EXTI line 20 được nối vào USB OTG HS (configured in FS) Wakeup event
- EXTI line 21 được nối vào RTC Tamper and TimeStamp events
- EXTI line 22 được nối vào RTC Wakeup event

Một số thanh ghi quan trọng với EXTI:

+ EXTI_IMR – Interrupt mask register:

Thanh ghi này cài đặt cho phép có yêu cầu ngắt trên Line tương ứng. (cho phép ngắt)

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

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

Bits 22:0 **MRx:** Interrupt mask on line x
0: Interrupt request from line x is masked
1: Interrupt request from line x is not masked

+ EXTI_RTSR – Rising trigger selection register:

Thanh ghi này được sử dụng để cấu hình chọn sườn lên làm tín hiệu kích hoạt ngắt.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

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

+ EXTI_FTSR – Falling trigger selection register:

Thanh ghi này được sử dụng để cấu hình chọn sườn xuống làm tín hiệu kích hoạt ngắt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

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

Note: The external wakeup lines are edge triggered, no glitch must be generated on these lines. If a falling edge occurs on the external interrupt line while writing to the EXTI_FTSR register, the pending bit is not set.

Rising and falling edge triggers can be set for the same interrupt line. In this configuration, both generate a trigger condition.

+ EXTI_SWIER – Software interrupt even register:

Thanh ghi này cho phép kích hoạt Line ngắt tương tứng bằng phần mềm.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									SWIER 22	SWIER 21	SWIER 20	SWIER 19	SWIER 18	SWIER 17	SWIER 16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWIER 15	SWIER 14	SWIER 13	SWIER 12	SWIER 11	SWIER 10	SWIER 9	SWIER 8	SWIER 7	SWIER 6	SWIER 5	SWIER 4	SWIER 3	SWIER 2	SWIER 1	SWIER 0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

Bits 22:0 **SWIERx:** Software Interrupt on line x

If interrupt are enabled on line x in the EXTI_IMR register, writing '1' to SWIERx bit when it is set at '0' sets the corresponding pending bit in the EXTI_PR register, thus resulting in an interrupt request generation.

This bit is cleared by clearing the corresponding bit in EXTI_PR (by writing a 1 to the bit).

+ EXTI_PR – Pending register:

Đây là thanh ghi chờ xử lý ngắt, khi có yêu cầu ngắt được tạo ra trên một Line ngắt thì bit tương tứng của thanh ghi này được bật lên cho đến khi ngắt này được xử lý. Nhiều trước hợp có sự thay đổi sườn tín hiệu tạo ra yêu cầu ngắt nhưng ngắt không được thực thi như: độ ưu tiên thấp, chưa cho phép ngắt toàn cục.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PR22	PR21	PR20	PR19	PR18	PR17	PR16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

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

Bits 22:0 **PRx:** Pending bit

0: No trigger request occurred

1: selected trigger request occurred

This bit is set when the selected edge event arrives on the external interrupt line.

This bit is cleared by programming it to '1'.

Mức độ ưu tiên ngắt NVIC :

Có 2 loại ưu tiên ngắt khác nhau, đó là **Preemption Priorities** và **Sub Priorities**:

- Mặc định thì ngắt nào có **Preemtion Priority** cao hơn thì sẽ được thực hiện trước.
- Khi nào 2 ngắt có cùng một **mức Preemption Priority** thì ngắt nào có **Sub Priority** cao hơn thì ngắt đó được thực hiện trước.

- Còn trường hợp 2 ngắt có cùng mức **Preemption** và **Sub Priority** luôn thì ngắt nào đến trước được thực hiện trước.

Lưu ý: Ngắt có giá trị càng bé thì mức ưu tiên càng cao.

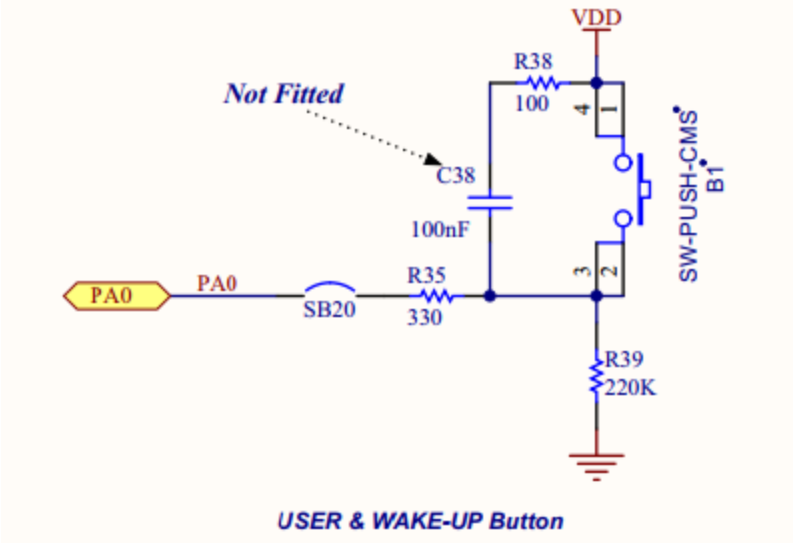
NVIC_PriorityGroup	NVIC_IRQChannelPreemptionPriority	NVIC_IRQChannelSubPriority	Description
NVIC_PriorityGroup_0	0	0-15	0 bits for pre-emption priority, 4 bits for subpriority
NVIC_PriorityGroup_1	0-1	0-7	1 bits for pre-emption priority, 3 bits for subpriority
NVIC_PriorityGroup_2	0-3	0-3	2 bits for pre-emption priority, 2 bits for subpriority
NVIC_PriorityGroup_3	0-7	0-1	3 bits for pre-emption priority, 1 bits for subpriority
NVIC_PriorityGroup_4	0-15	0	4 bits for pre-emption priority, 0 bits for subpriority

II. Lập trình thực hành ngắt ngoài

Bài toán đặt ra là VĐK thực hiện chương trình bình thường, khi ta nhấn một nút nhấn thì phát sinh sự kiện ngắt ngoài gửi vào vi điều khiển, khi đó vi điều khiện triệu gọi một chương trình con phục vụ ngắt để thực hiện bật/tắt led ở PA12

Sau khi tìm hiểu lý thuyết về EXTI, chúng ta cùng thực hành 1 project sử dụng ngắt ngoài trên KIT STM32F407VG. Trên MCU này, nhà sản xuất đã kết nối sẵn cho chúng ta chân PA0 với User button (nút nhấn màu xanh trên KIT). Vì vậy chúng ta sẽ tận dụng nút nhấn này để thực hành.

Sơ đồ nối mạch của **User button** như hình dưới đây:



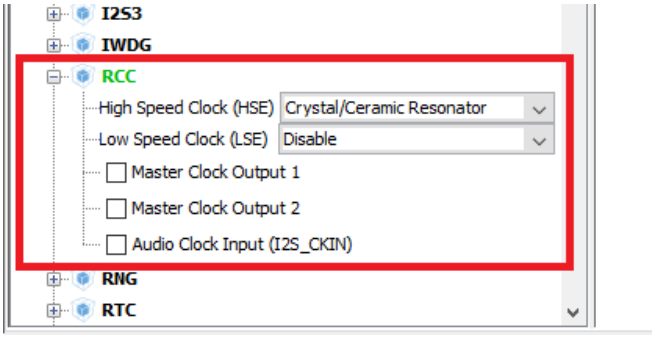
1. Khởi tạo Project với CubeMX

Khởi tạo **New Project** với CubeMX, chọn dòng chip chúng ta sử dụng.

Tiếp theo, chúng ta cấu hình thạch anh và xung Clock cho Chip:

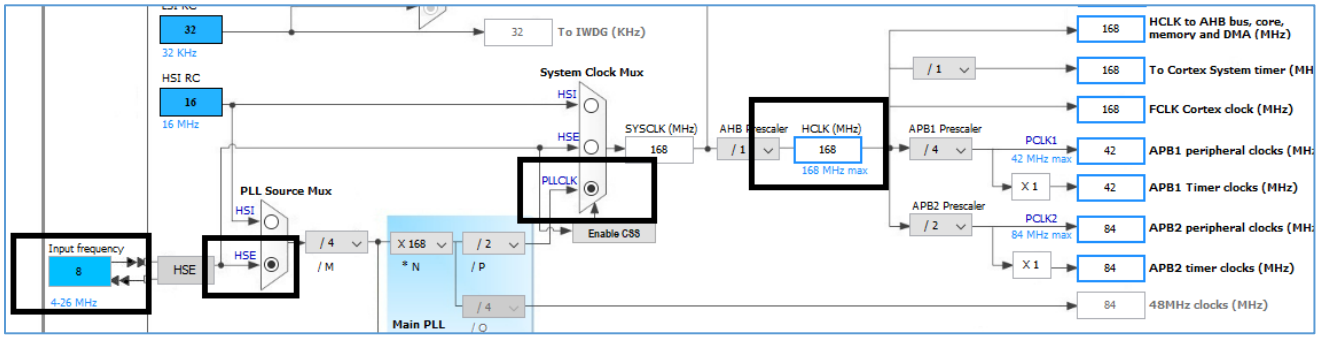
Tại mục **RCC** → **High Speed Clock(HSE)** và chọn "**Crystal/Ceramic Resonator**"

Chức năng này sẽ giúp chip hoạt động với thạch anh ngoại được gắn sẵn trên board mạch.

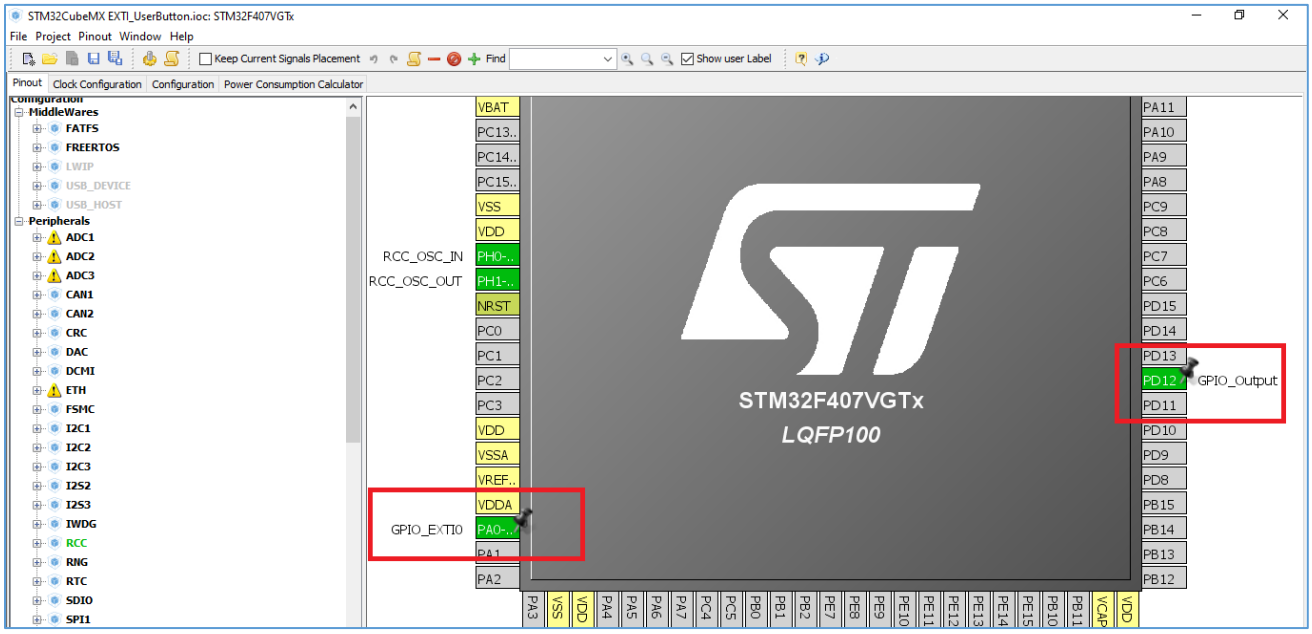


Tiếp theo, chúng ta chuyển sang Tab "**Clock Configuration**" và tích chọn mục **HSE**, tín hiệu Clock đi qua bộ nhân tần PLLCLK giúp chip đạt được tần số hoạt động tối đa.

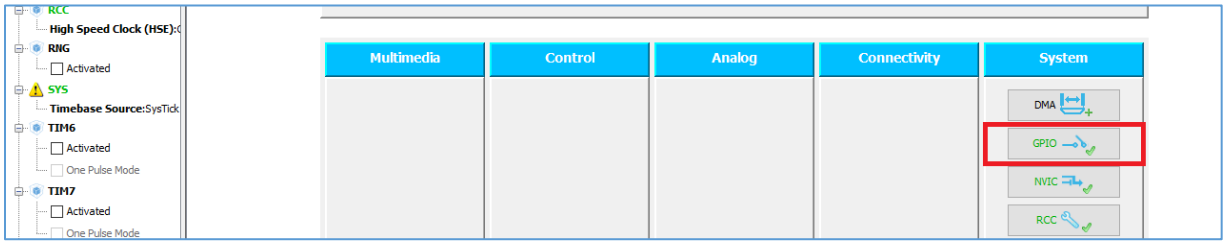
Tại mục **Input frequency**, các bạn điền "8" (thạch anh hàn sẵn trên board là loại 8Mhz). Sau đó chúng ta điền "168" tại mục HCLK (đây là tần số hoạt động tối đa của chip) và ấn Enter, đợi cho CubeMX tự tính toán các thông số.



Trong pinout, mình sẽ cấu hình cho chân **PA0** hoạt động với chức năng **GPIO_EXTI0** và cấu hình cho chân **PD12** ở chế độ **GPIO_Output** để quan sát sự hoạt động của ngắt

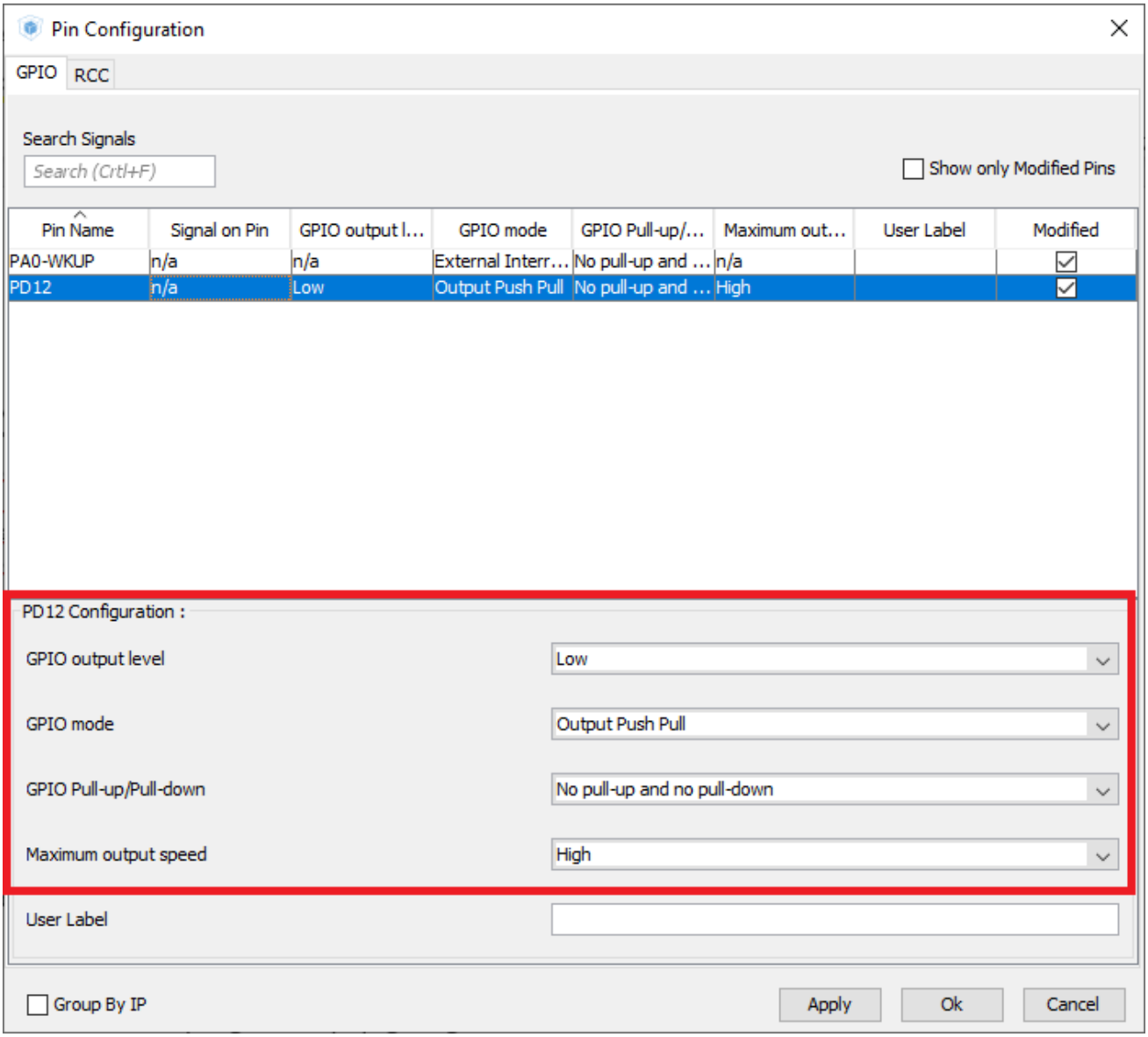


Cấu hình **GPIO** : chuyển sang Tab "**Configuration**", chúng ta chọn mục GPIO.



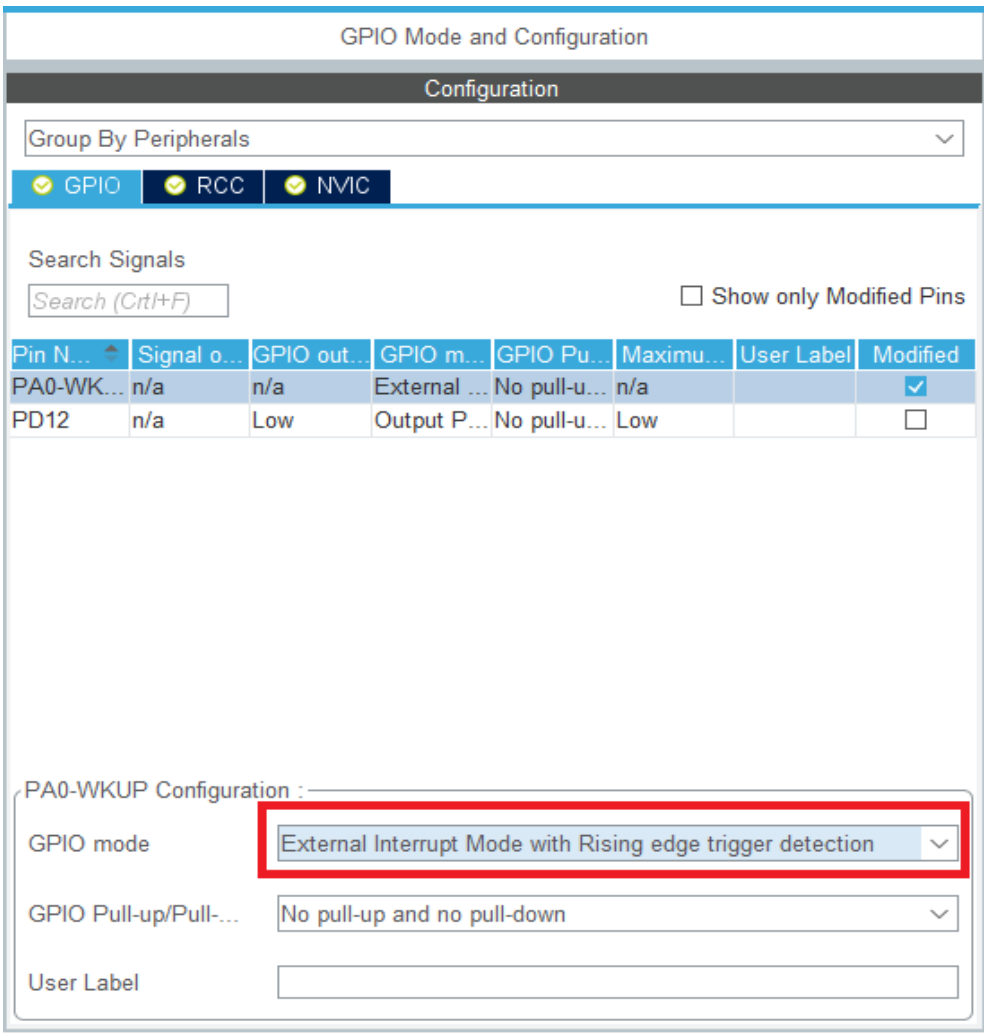
Tại đây, mình sẽ thiết lập các thông số như sau:

Đối với **PD12** :

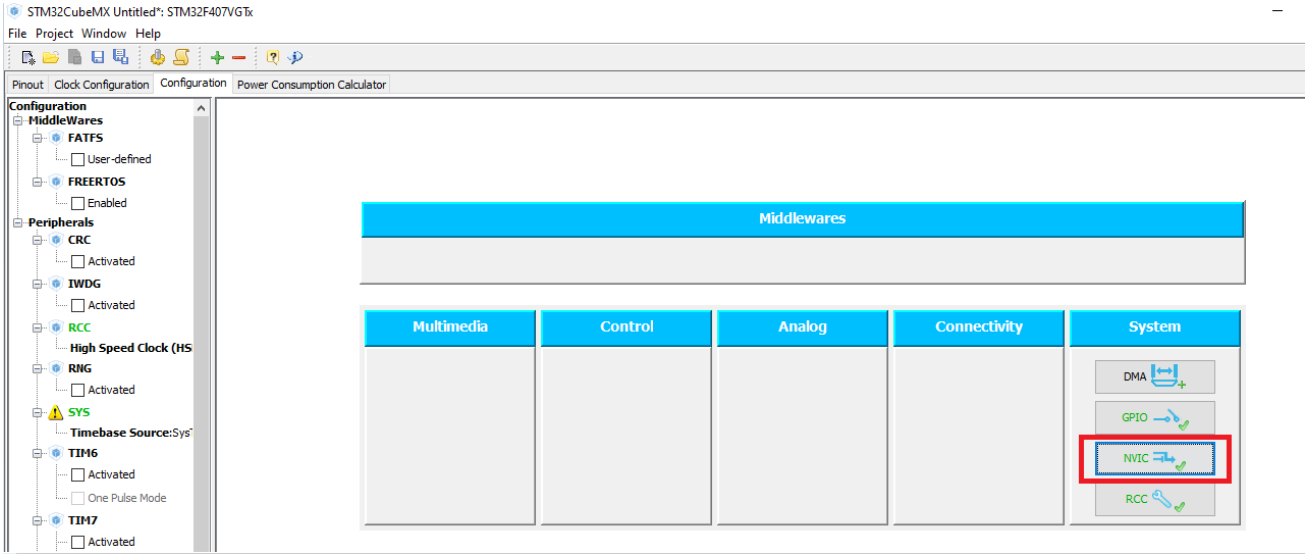


Đối với **PA0** :

Ta chọn bắt ngắt theo sườn lên vì nhìn vào schematic của khối nút bấm, các bạn có thể dễ dàng nhận ra ở thời điểm nút nhả thì chân PA0 có mức ưu logic là 0, khi ta ấn nút thì chân PA0 lên mức logic 1.

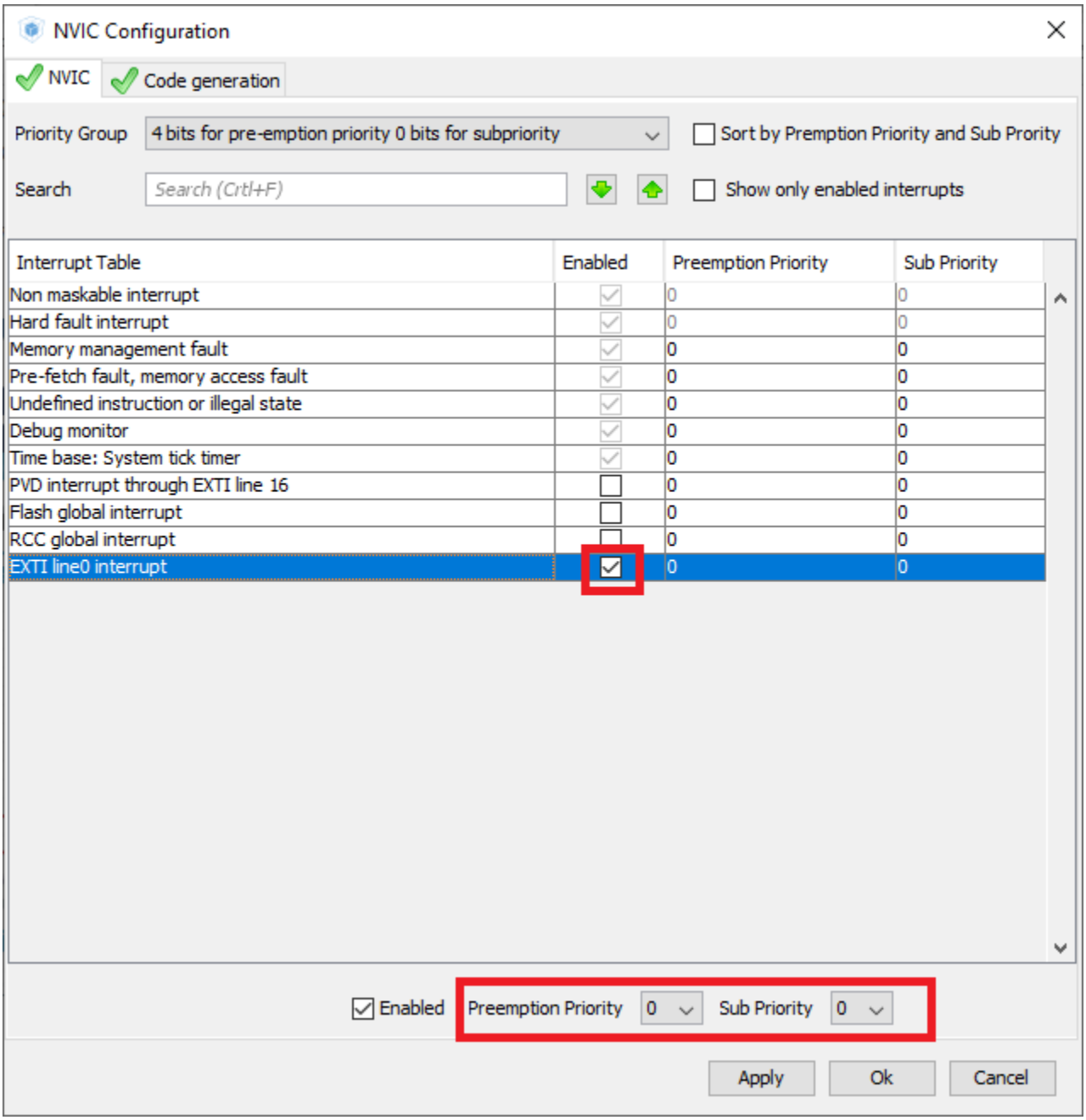


Tiếp đến, các ta chọn mục NVIC để cấu hình ngắt.

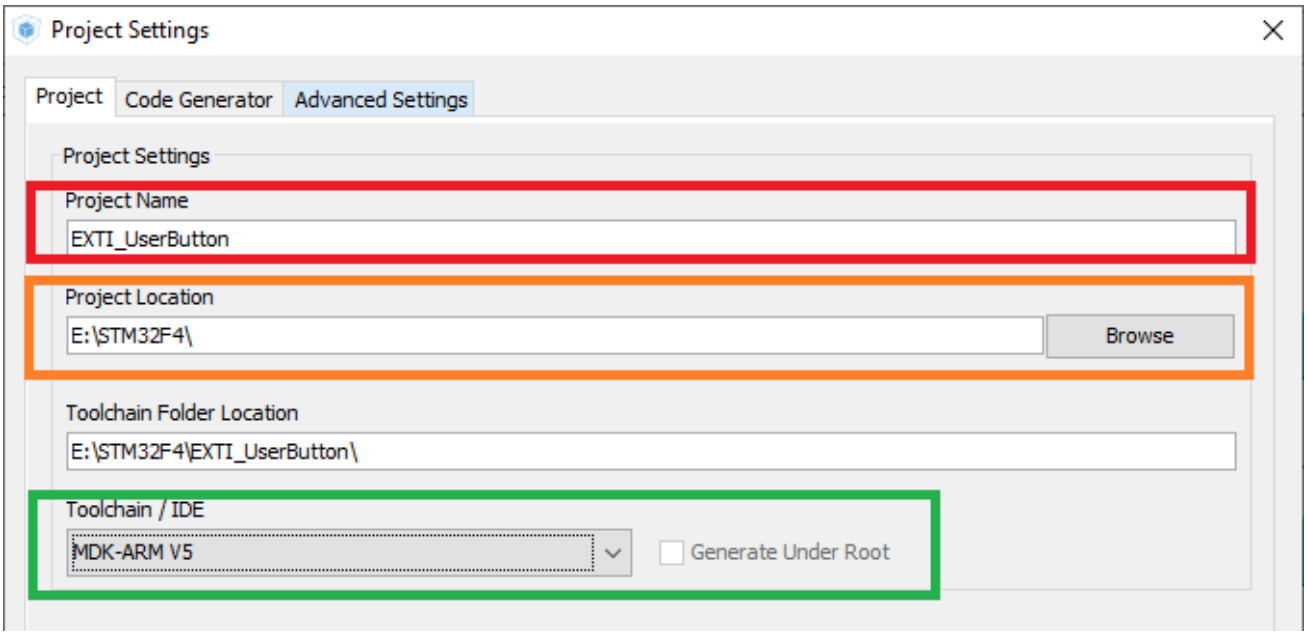


Khi cửa sổ NVIC Configuration hiện lên, chúng ta sẽ **Enable** cho EXTI **line0** interrupt.

Tại mục Preemption Priority và Sub Priority, chúng ta có thể thay đổi mức 2 thông số cho mức ưu tiên ngắt. Nhưng trong ví dụ này, các bạn hãy để mặc định là "0" và "0", và cùng theo dõi tiếp để hiểu rõ hơn **mức ưu tiên ngắt là gì, tác dụng như thế nào** ở cuối bài viết nhé !



Cuối cùng là **Setting Project** và tạo code.

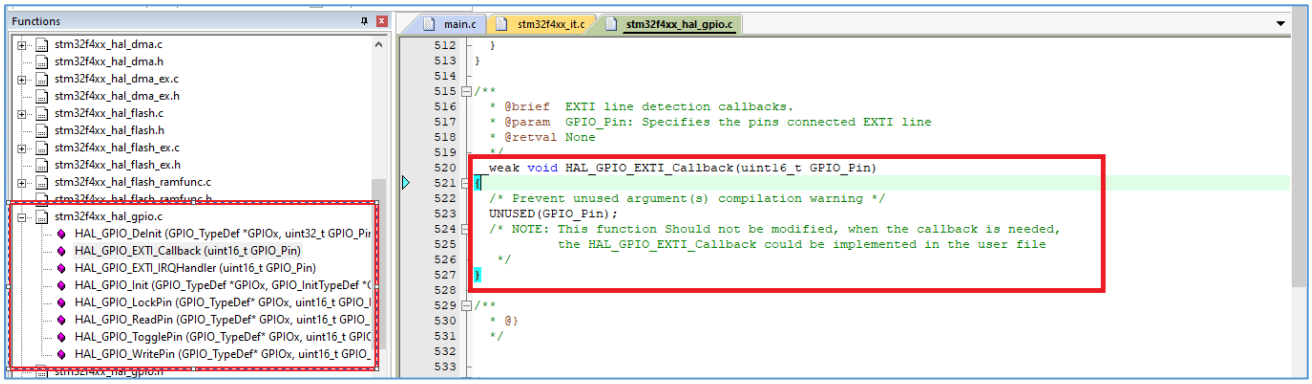


Ở Tab "**Code Generator**", hãy chọn "**Copy only necessary library files**" để chương trình sinh ra chỉ với các thư viện cần thiết, tiết kiệm dung lượng và thời gian build code

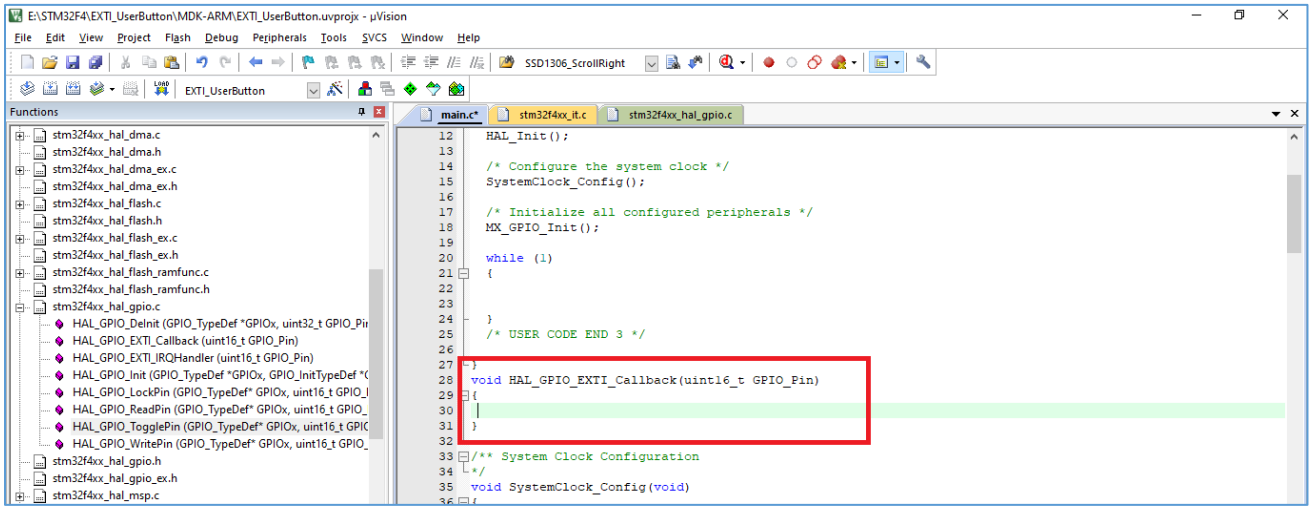
2. Lập trình với KeilC

Sau khi CubeMX sinh code xong, chúng ta chọn Open Project để mở chương trình trên KeilC.

Tại mục Functions, chúng ta mở file "**stm32f4xx_hal_gpio.c**", tìm đến hàm **HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin).** Hàm này là chương trình con phục vụ ngắt (sẽ phát hiện có sự kiện ngắt và xử lý yêu cầu ngắt đó).



Lưu ý : hàm này không nên chỉnh sửa vì được khai báo với **__weak** , nếu muốn sử dụng đến nó, chúng ta phải khai báo ở 1 file khác, ở đây ta sẽ khai báo trong file "main.c".



Trong hàm void **HAL_GPIO_EXTI_Callback**(**uint16_t GPIO_Pin**) ta sẽ viết chương trình như sau :

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if(GPIO_Pin == GPIO_PIN_0)
    {
        HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
    }
}
```

Câu lệnh **if(**GPIO_Pin **==** GPIO_PIN_0**)** sẽ giúp kiểm tra, phân luồng, phát hiện ngắt có đúng đang sinh ra có phải ở chân 0 hay không.

Build chương trình (**F7**) và nạp code xuống kit (**F8**)

Nhấn Reset button trên kit để reset lại KIT, thực hiện thao tác nhấn User button để quan sát led xanh lá cây trên KIT.

III. Mức ưu tiên ngắt trên vi điều khiển STM32

1. Đặt vấn đề

Như đã tìm hiểu ở trên, vi điều khiển STM32 hỗ trợ rất nhiều ngắt khác nhau và chúng được quản lý bởi NVIC (Nested Vector Interrupt Controller).

Vậy điều gì sẽ xảy ra nếu có 2 yêu cầu ngắt đang chờ để xử lý hoặc nếu 1 ngắt đang được xử lý thì 1 ngắt khác xuất hiện và 2 ngắt này có cùng mức ưu tiên cả Preemption Priority,Sub Priority?

Để làm rõ vấn đề này, chúng ta hãy làm 1 phép thử sau : thêm 1 dòng code HAL_Delay(1000);

vào trong hàm void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) trong file "main.c"

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if(GPIO_Pin == GPIO_PIN_0)
  {
    HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
    HAL_Delay(1000);
  }
}
```

Build lại chương trình và nạp code xuống kit, sau đó reset lại KIT.

Nào, giờ hay nhấn User button như lúc nãy. Chắc chắn đèn led xanh lá sẽ sáng hẳn hoặc tối hẳn.

Điều này có nghĩa là gì?

Chương trình của bạn đã bị "treo", vì 2 yêu cầu ngắt có cùng mức ưu tiên đồng thời xuất hiện, điều này khiến chương trình bị đứng tại đây.

Hàm HAL_Delay() chúng ta hay sử dụng cũng là 1 kiểu ngắt, và nó có mức ưu tiên là Preemption Priority : 0, Sub Priority : 0

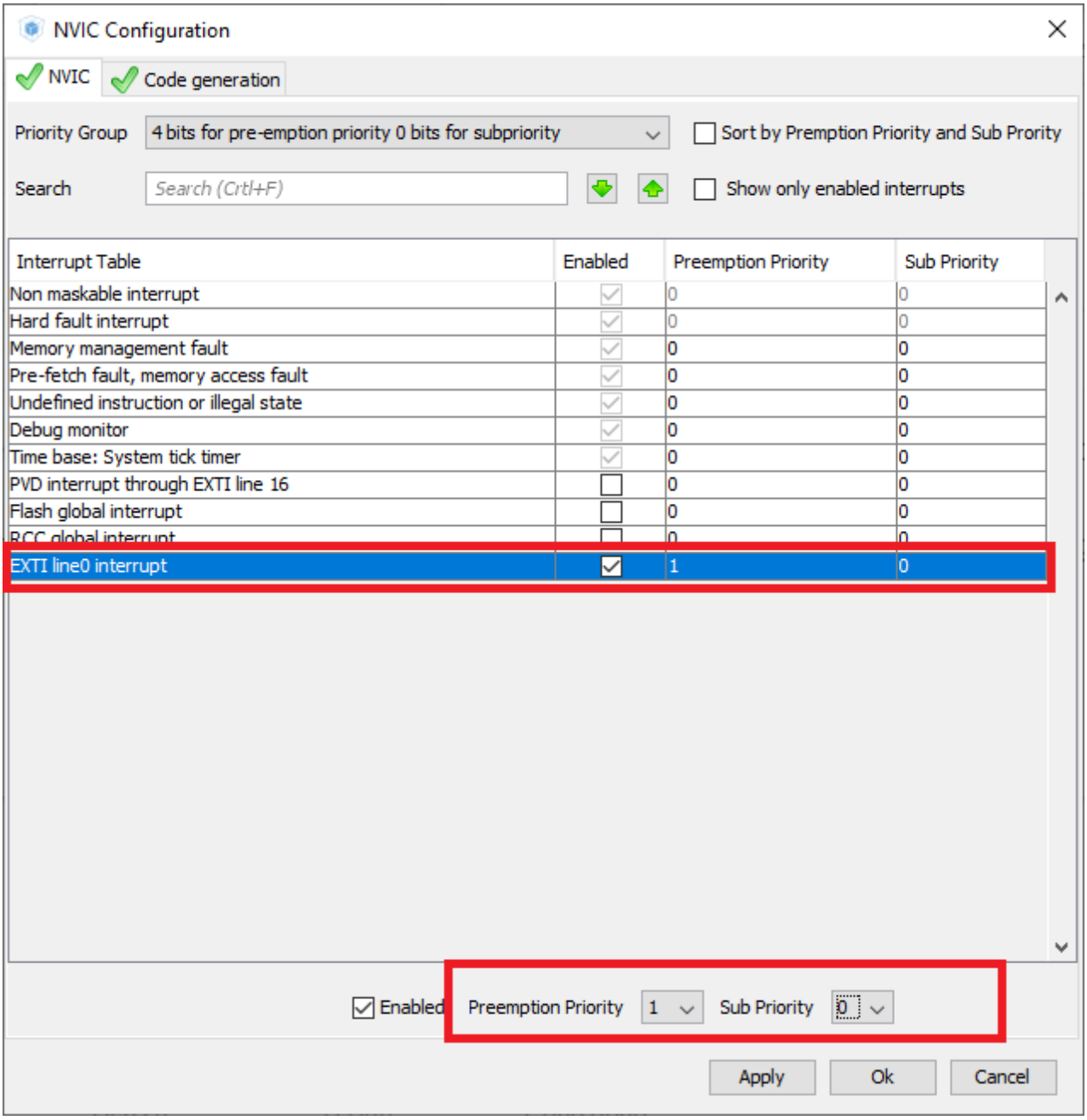
Vì vậy, mức ưu tiên của hàm HAL_Delay() ngang bằng với mức ưu tiên của EXTI line0 mà chúng ta đang sử dụng.

2. Cách khắc phục

Để khắc phục điều này, chúng ta cần xử lý như thế nào?

Cách 1:

Mở lại CubeMX và thiết lập lại thông số cho Preemption Priority, Sub Priority của EXTI line0, tạo lại code mới.

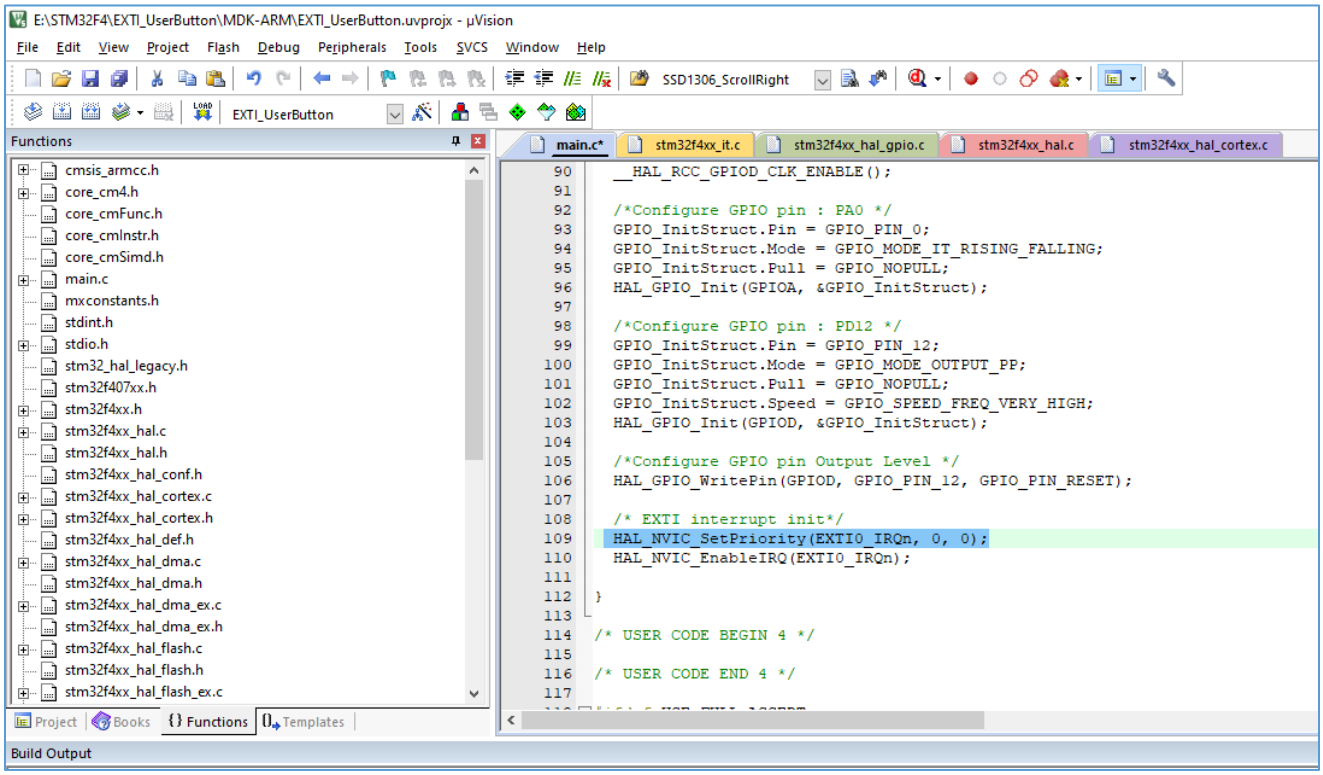


Lưu ý : Cách này chỉ nên thực hiện trước khi sinh code ra KeilC, vì nếu bạn khởi tạo lại 2 thông số này, sau đó remake project, toàn bộ code trong chương trình cũ sẽ bị reset lại.

Cách 2 : Chỉnh sửa mức ưu tiên ngắt ngay trong chương trình của mình

Trong file "main.c", chúng ta kéo xuống và tìm đến hàm.

```
HAL_NVIC_SetPriority(EXTI0_IRQn, 0, 0);
```



Hàm này cho phép chúng ta thiết lập mức ưu tiên cho các line ngắt. Chúng ta sẽ sửa hàm này thành như sau :

```
HAL_NVIC_SetPriority(EXTI0_IRQn, 1, 0);
```

Điều này có nghĩa chúng ta đã thay đổi Preemption Priority cho EXTI line0 từ "0" xuống "1", và Sub Priority vẫn giữ là "0" . Như vậy mức ưu tiên của EXTI line0 sẽ không ngang bằng với ngắt của hàm HAL_Delay() nữa.

Sau đó build lại chương trình và nạp code xuống kit. Reset lại board và nhấn User button, quan sát sự thay đổi của led nhé!

Trên đây là bài giới thiệu về ngắt ngoài (EXTI) cho vi điều khiển STM32F4 và cách cấu hình mức ưu tiên ngắt.

BÀI 03

CHỨC NĂNG ADC TRÊN VI ĐIỀU KHIỂN STM32F4

I. ADC là gì? Tác dụng của nó như thế nào?

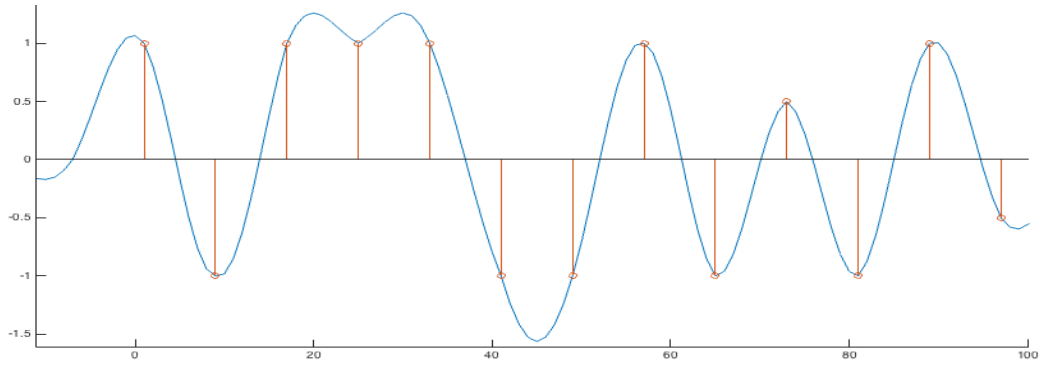
Các tín hiệu chúng ta thường gặp trong tự nhiên như điện áp, ánh sáng, âm thanh, nhiệt độ… đều tồn tại dưới dạng tương tự (Analog), có nghĩa là tín hiệu liên tục và mức độ chia nhỏ vô hạn. Ví dụ: trong khoảng điện áp từ 0 -> 5V có vô số khoảng giá trị điện áp, ánh sáng sẽ tồn tại từ mờ cho tới sáng tỏ, âm thanh từ nhỏ cho đến lớn dưới dạng liên tục.

Ngược lại trong vi điều khiển chỉ có khái niệm số (Digital), cấu trúc từ nhân cho đến bộ nhớ hoạt động dựa trên các Transistor chỉ gồm mức 0-1 nên nếu muốn giao tiếp với chip thì tín hiệu phải được số hóa trước khi đưa vào chip. Quá trình số hóa có thể thực hiện bằng nhiều cách và nhiều công đoạn nhưng mục đích cuối cùng là để vi điều khiển hiểu được tín hiệu tương tự đó.

ADC (Analog-to-Digital Converter) bộ chuyển đổi tín hiệu tương tự - số là thuật ngữ nói đến sự chuyển đổi một tín hiệu tương tự thành tín hiệu số để dùng trong các hệ thống số (Digital Systems) hay vi điều khiển.

Trong bộ chuyển đổi ADC, có 2 thuật ngữ mà chúng ta cần chú ý đến, đó là độ phân giải (**resolution**) và thời gian lấy mẫu (**sampling time**).

- **Độ phân giải (resolution):** dùng để chỉ số bit cần thiết để chứa hết các mức giá trị số (digital) sau quá trình chuyển đổi ở ngõ ra. Bộ chuyển đổi ADC của STM32F407VG có độ phân giải mặc định là 12 bit, tức là có thể chuyển đổi ra 2^{12}= 4096 giá trị ở ngõ ra số.
- **Thời gian lấy mẫu (sampling time):** là khái niệm được dùng để chỉ thời gian giữa 2 lần số hóa của bộ chuyển đổi. Như ở đồ thị dưới đây, sau khi thực hiện lấy mẫu, các điểm tròn chính là giá trị đưa ra tại ngõ ra số. Dễ nhận thấy nếu thời gian lấy mẫu quá lớn thì sẽ làm cho quá trình chuyển đổi càng bị mất tín hiệu ở những khoảng thời gian không nằm tại thời điểm lấy mẫu. Thời gian lấy mẫu càng nhỏ sẽ làm làm cho việc tái thiết tín hiệu trở nên tin cậy hơn.



Để hiểu quá trình số hóa trong STM32 diễn ra như thế nào ta theo dõi ví dụ sau. Giả sử ta cần đo điện áp tối thiểu là 0V và tối đa là 3.3V, trong STM32 sẽ chia 0 → 3.3V thành 4096 khoảng giá trị (từ 0 → 4095, do $2^{12} = 4096$), giá trị đo được từ chân IO tương ứng với 0V sẽ là 0, tương ứng với 1.65V là 2047 và tương ứng 3.3V sẽ là 4095.

Trong STM32F407VG có 3 bộ ADC chuyển đổi tín hiệu tương tự thành tín hiệu số với độ phân giải 12-bit. Mỗi ADC có khả năng tiếp nhận tín hiệu từ 16 kênh ngoài.

Ở bài này, chúng ta sẽ sử dụng *ADC1 channel 0* để đọc điện áp từ một biến trở đưa vào.

II. Thực hành

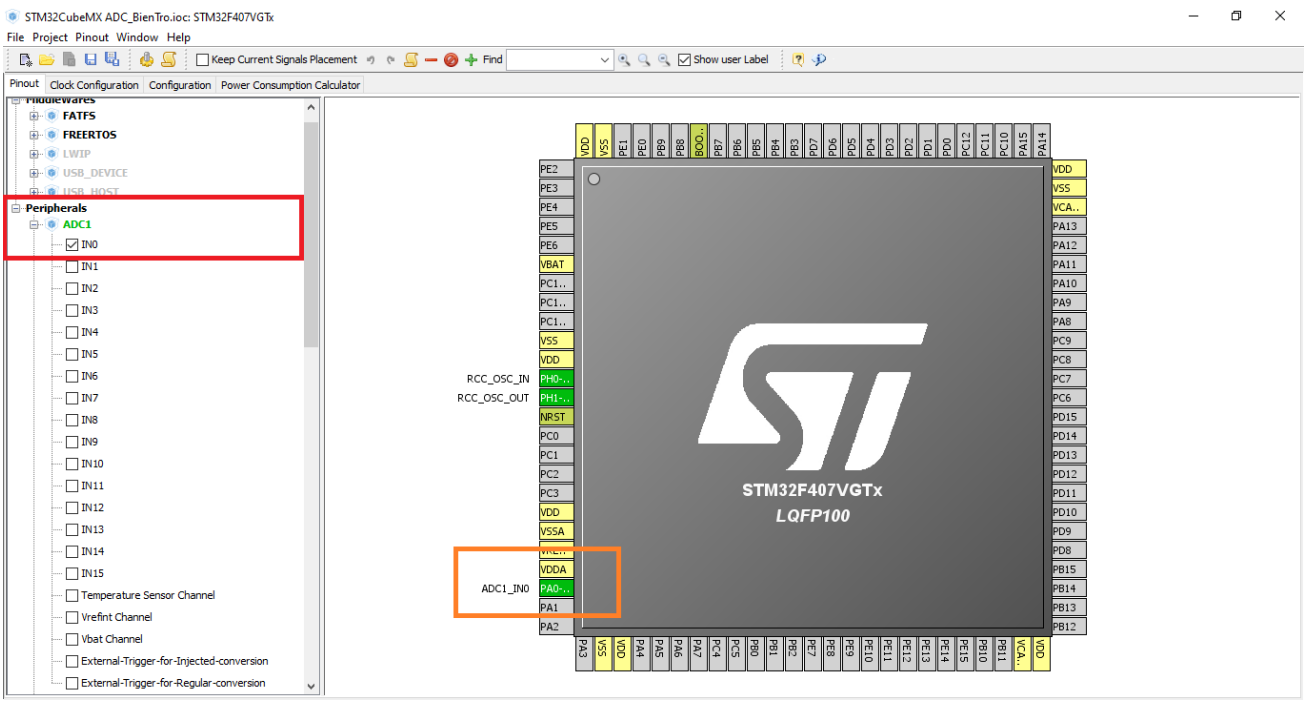
1. Cấu hình với CubeMX

Khởi động CubeMX, chọn chip mà chúng ta sử dụng. Ở đây mình sẽ chọn STM32F407VG.

Tại mục "**Peripherals**", các bạn chọn ADC1 và click chọn IN0, thao tác này đã kích hoạt ADC1 channel 0 trên chip.

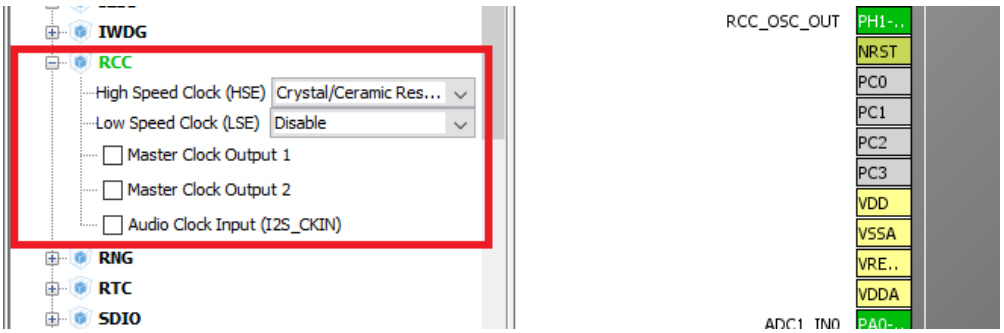
Hoặc chúng ta có thể tìm đến chân PA0 và chọn chế độ ADC1_IN0 .

(Chân PA0 là chân được liên kết sẵn với channel 0 của bộ ADC1)



Cấu hình thạch anh và xung clock cho chip:

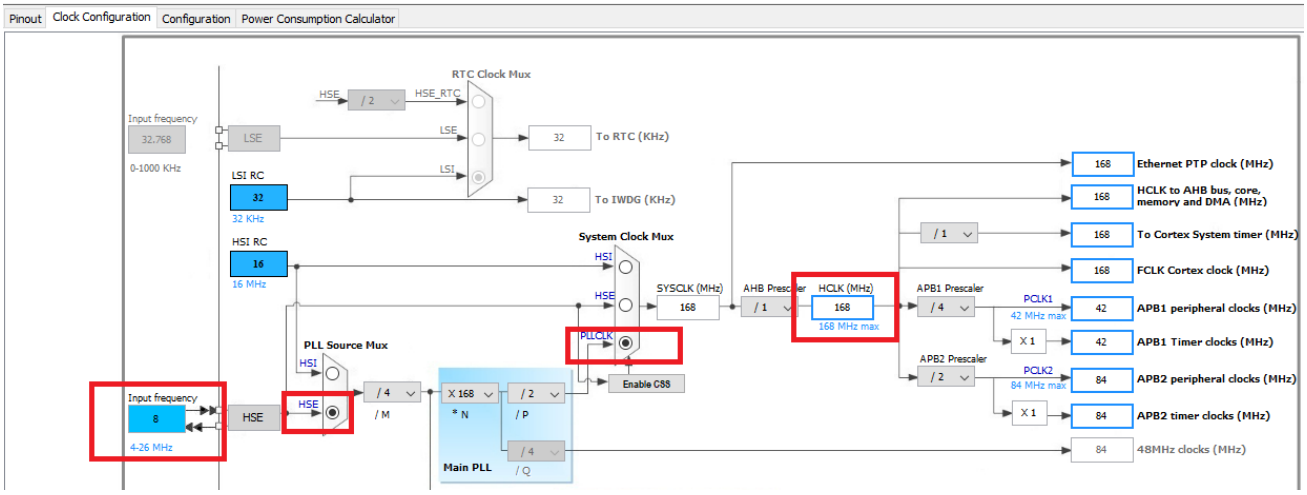
Tại mục RCC, trong phần "**High Speed Clock(HSE)**", chúng ta chọn "**Crystal/Ceramic Resonator**" để cấu hình cho chip hoạt động với thạch anh ngoại.



Tại Tab "**Clock Configuration**", chúng ta thiết lập các thông số Clock cho Chip.

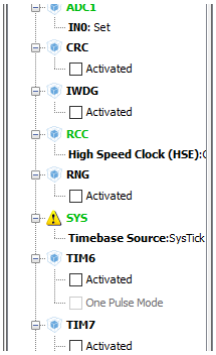
Chọn HSE, xung Clock sẽ đi qua bộ nhân tần PLLCLK, giúp Chip hoạt động với tần số tốt nhất.

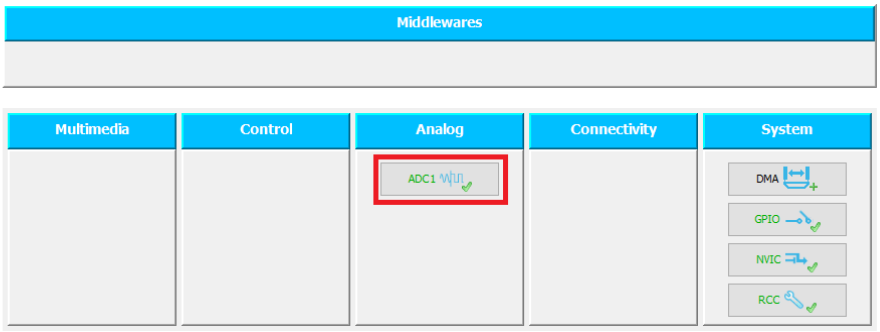
Điền "8" ở mục "Input frequency" (thông số của thạch anh ngoại gắn trên kit) và điền "168" ở mục "HCLK" (168MHz là tần số tối đa của chip).



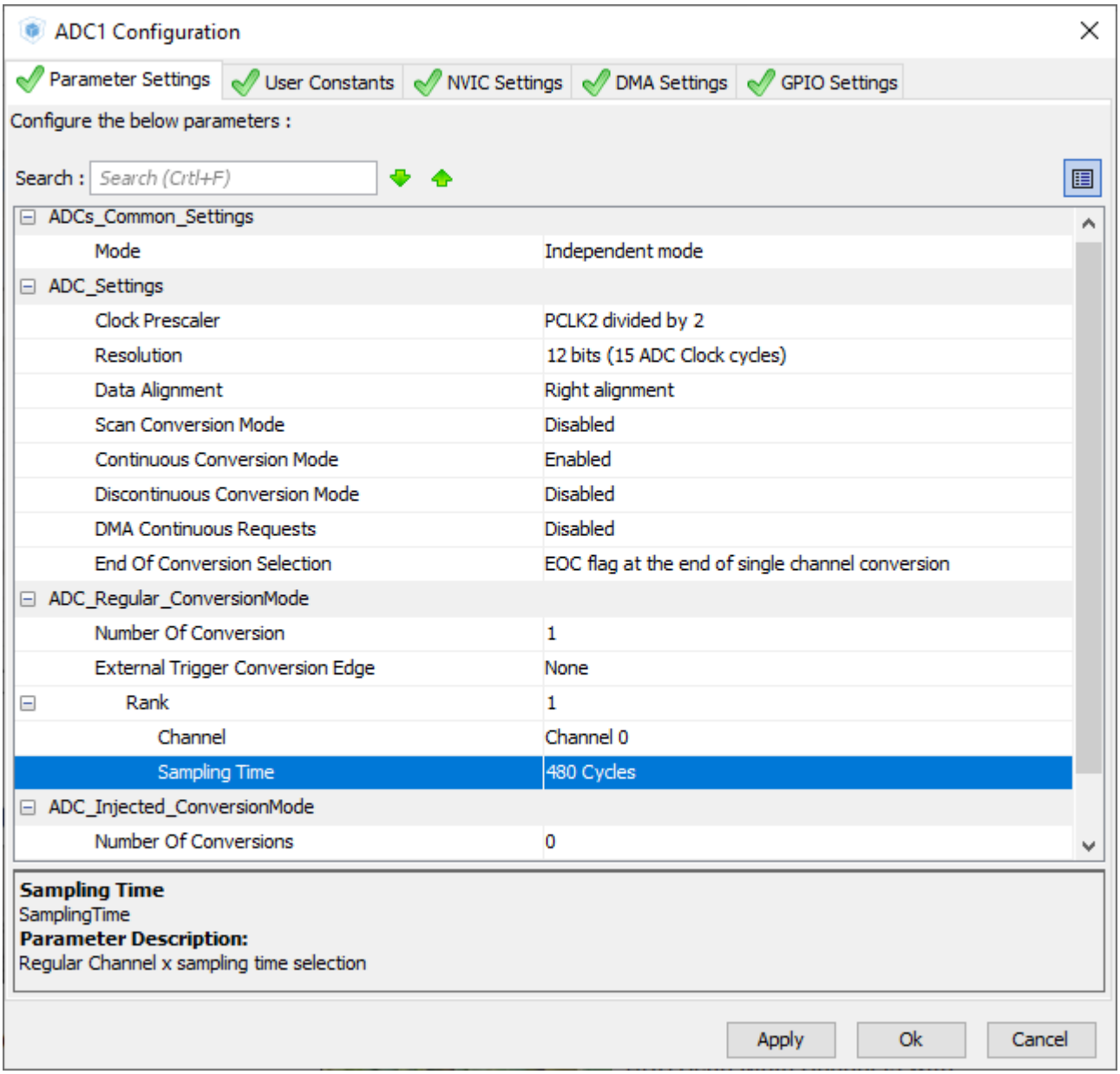
Cấu hình cho bộ ADC

Trong Tab "Configuration", chọn mục ADC1.



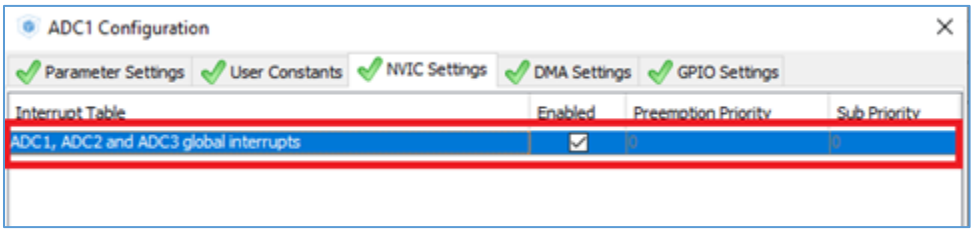


Khi cửa sổ "ADC1 Configuration" hiện lên, chúng ta thiết lập các thông số như sau:

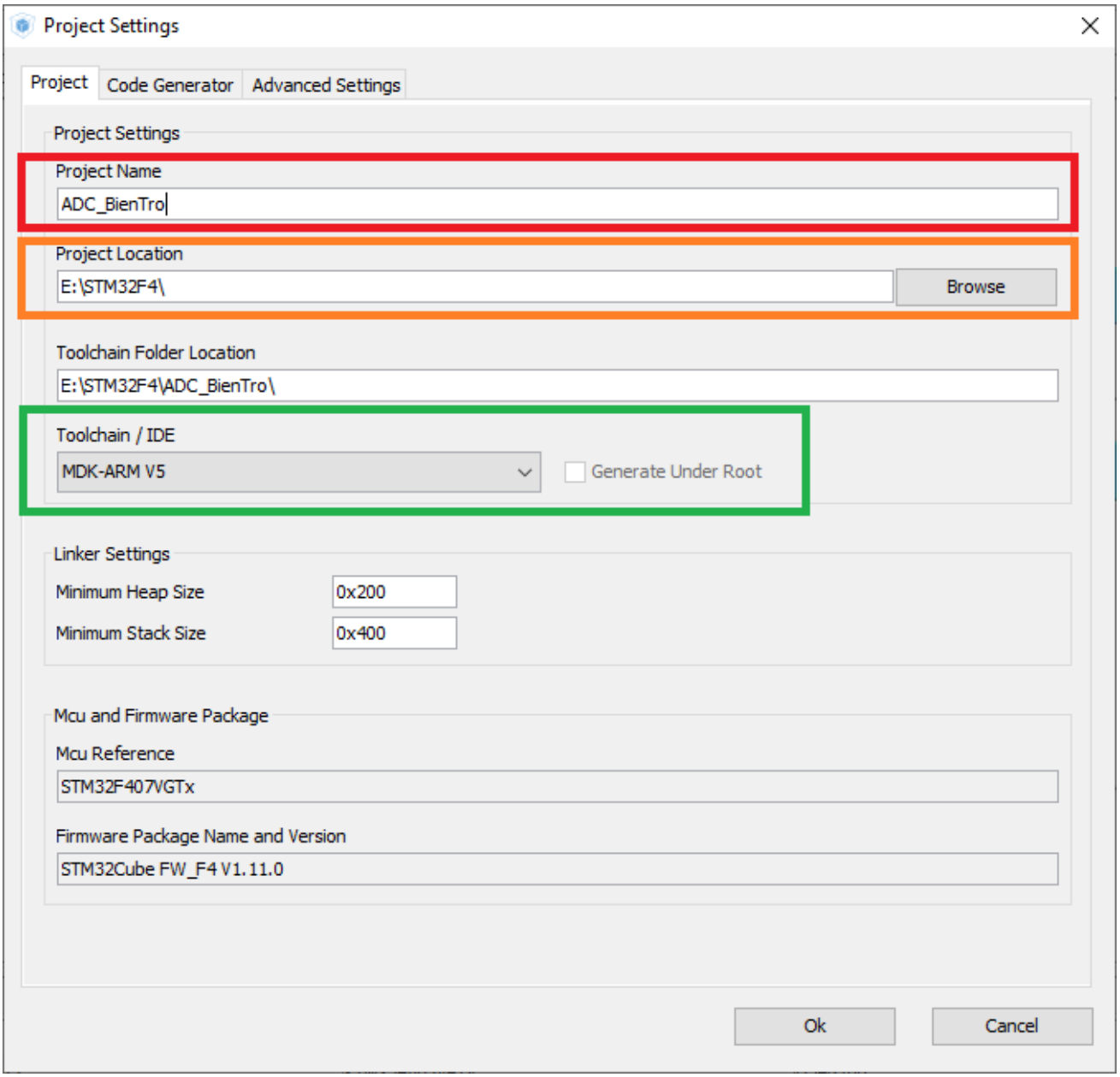


- **Resolution**: như đã giới thiệu ở trên, độ phân giải càng cao, quá trình chuyển đổi sẽ càng chính xác, vì vậy chúng ta chọn "12 bits (15 ADC Clock cycles".
- **Data Alignment**: vì độ phân giải là 12 bit, nên chúng ta cần 1 thanh ghi 16 bit để lưu dữ liệu, như vậy sẽ còn dư 4bit. Chúng ta chọn "Right alignment" tức là lưu 12bit dữ liệu dịch về phía bên phải của thanh ghi 16bit này.
- **Continuous Conversion Mode**: cho phép các quá trình chuyển đổi diễn ra liên tục, chúng ta sẽ "enable" chức năng này.
- **Sampling Time**: thời gian lấy mẫu, nếu thông số này càng lớn, độ chính xác càng lớn nhưng bù lại quá trình chuyển đổi sẽ diễn ra lâu hơn.

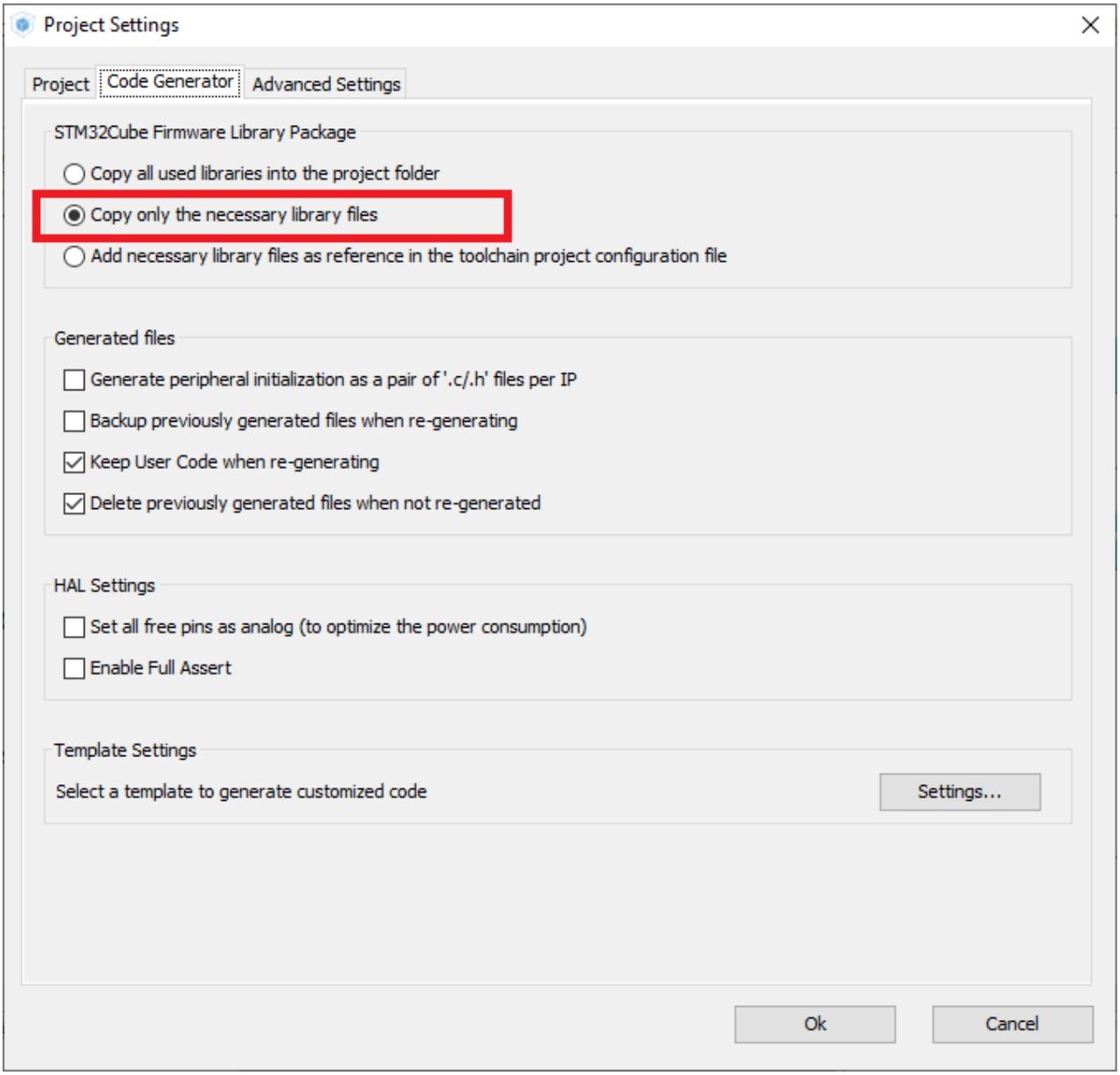
Trong Tab "**NVIC Settings**", chọn "**Enable**" để cho phép ngắt xảy ra trên bộ ADC.



Cuối cùng là Setting Project và sinh code.



Ở Tab "**Code Generator**", các bạn chọn "**Copy only the necessary library files**" để project tạo ra chỉ với những thư viện cần thiết, tiết kiệm dung lượng và thời gian build code nhé!



2. Lập trình với KeilC

Sau khi CubeMX sinh code xong, các bạn "Open Project" để mở chương trình trên KeilC

Trong file "main.c", mình sẽ khai báo 1 biến toàn cục `volatile uint16_t adc_value = 0;`

Biến "adc_value" này mình sẽ dùng để nhận giá trị chuyển đổi của bộ ADC.

Ở đây, "volatile" có tác dụng tránh những lỗi sai khó phát hiện do tính năng Optimization của Compiler gây ra vì giá trị của biến "adc_value" sẽ bị thay đổi đột ngột do tác động của phần cứng. Khi Compiler nhận định biến này không hề có sự thay đổi giá trị do phần mềm nó sẽ coi như đây là 1 biến không được sử dụng và loại bỏ nó trong quá trình Optimize.

Trong hàm main, chúng ta thêm câu lệnh sau HAL_ADC_Start_IT(&hadc1);

Câu lệnh này đã cho phép ngắt và bắt đầu chuyển đổi với các kênh của bộ ADC1.

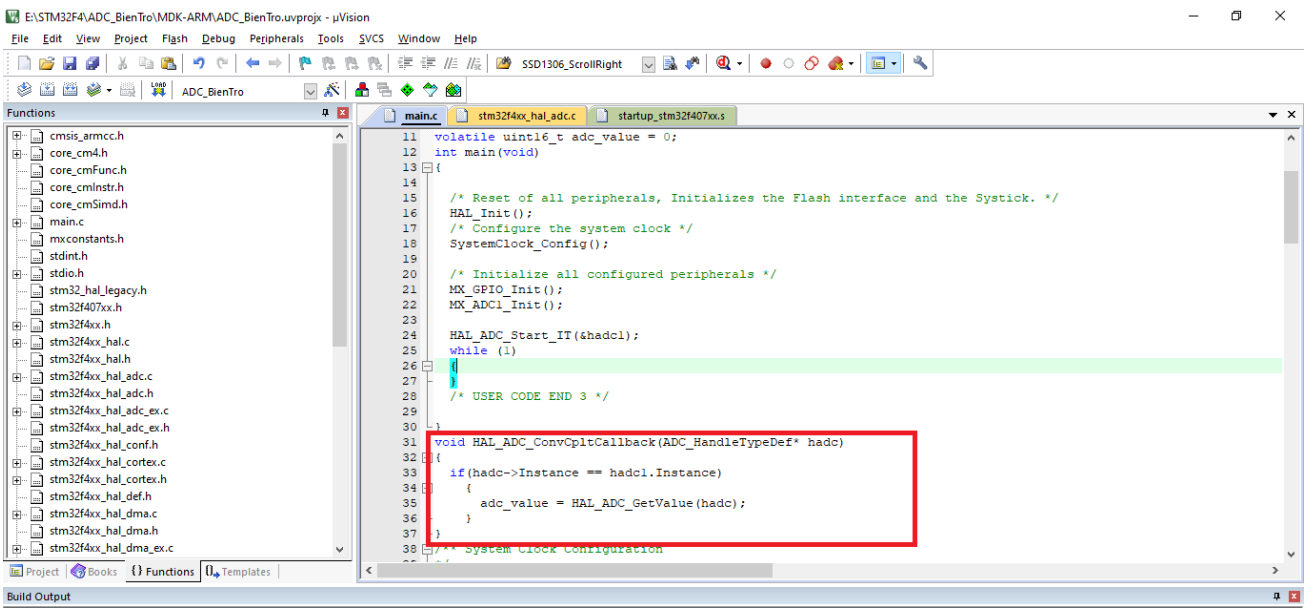
```
volatile uint16_t adc_value = 0;
int main(void)
{

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  HAL_ADC_Start_IT(&hadc1);

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC1_Init();

  while (1)
  {

  }
```

Tại mục "**Functions**", các bạn tìm đến file "**stm32f4xx_hal_adc.c**" chọn hàm "**HAL_ADC_ConvCpltCallback**". Hàm này là hàm phục vụ ngắt ADC, được gọi đến khi 1 quá trình chuyển đổi ADC hoàn thành.

Vì hàm này được khai báo với **__weak**, nếu muốn sử dụng nó, chúng ta phải khai báo sang 1 file khác. Ở đây mình sẽ khai báo và sử dụng nó trong file "main.c"

Chúng ta sẽ lập trình như sau với hàm này :

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
  if(hadc->Instance == hadc1.Instance)
        {
            adc_value = HAL_ADC_GetValue(hadc);
        }
}
```



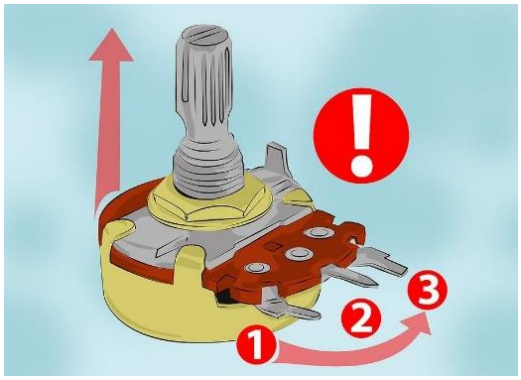
Câu lệnh **if(**hadc**->**Instance **==** hadc1**.**Instance**)** sẽ kiểm tra xem bộ ADC đang yêu cầu xử lý ngắt có phải là bộ ADC1 không.

Sau đó chúng ta thực hiện gán giá trị chuyển đổi từ bộ ADC1 về biến adc_value thông qua hàm HAL_ADC_GetValue**(**hadc**);**

Build chương trình (**F7**) và nạp code xuống kit (**F8**), nhấn nút reset KIT.

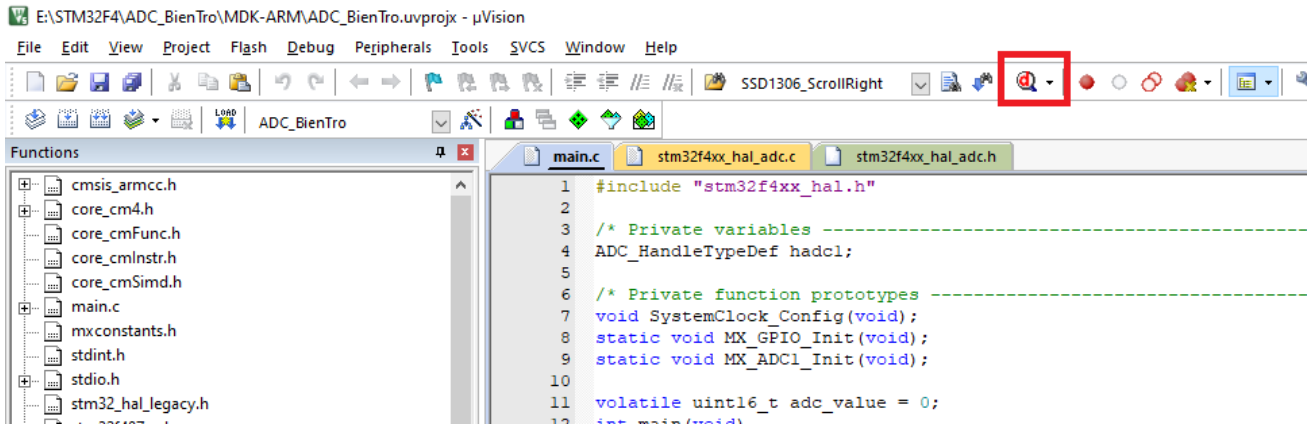
Kết nối biến trở với kit STM32F407 :

Chân – GND
Chân 2 – PA0
Chân 3 – 3V

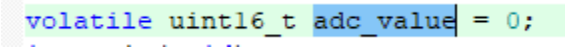


Debug và quan sát sự thay đổi giá trị của biến trở :

Trên thanh công cụ, chúng ta click vào icon Debug.

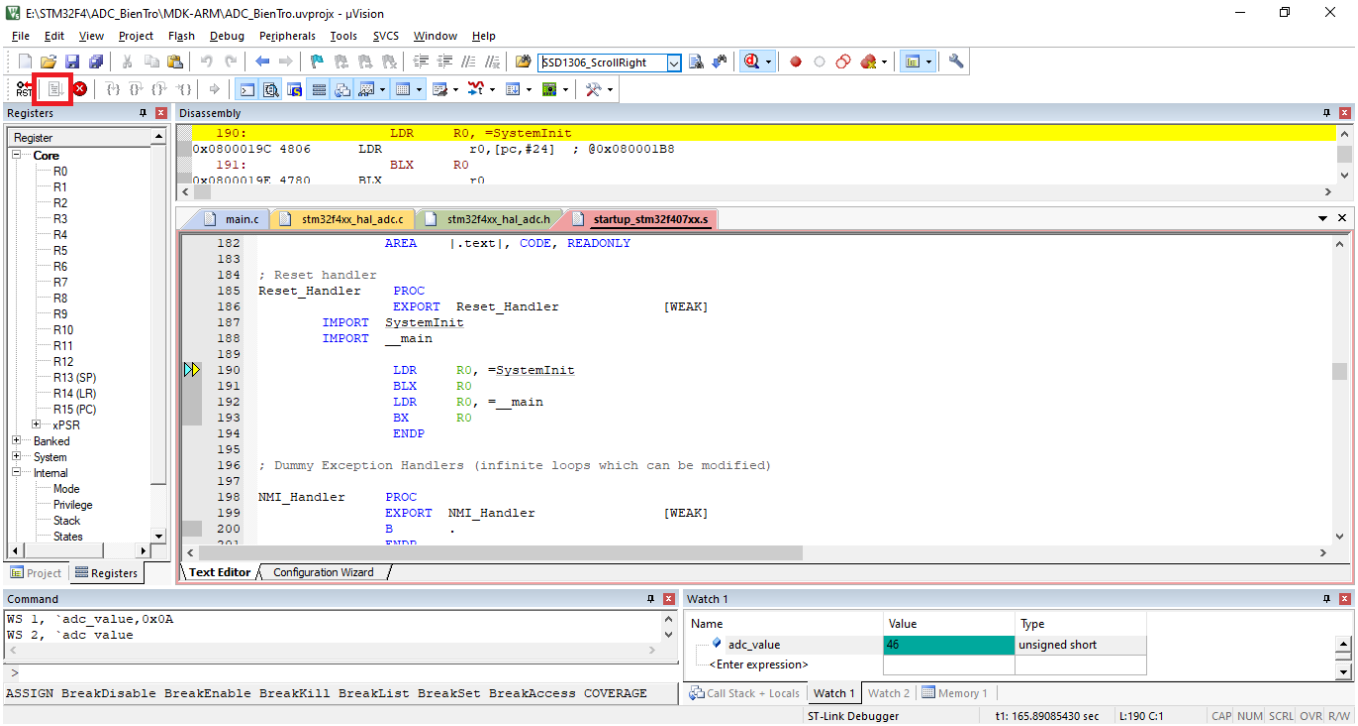


Khi cửa sổ Debug hiện lên, chúng ta tìm đến biến adc_value đã khai báo ở đầu chương trình

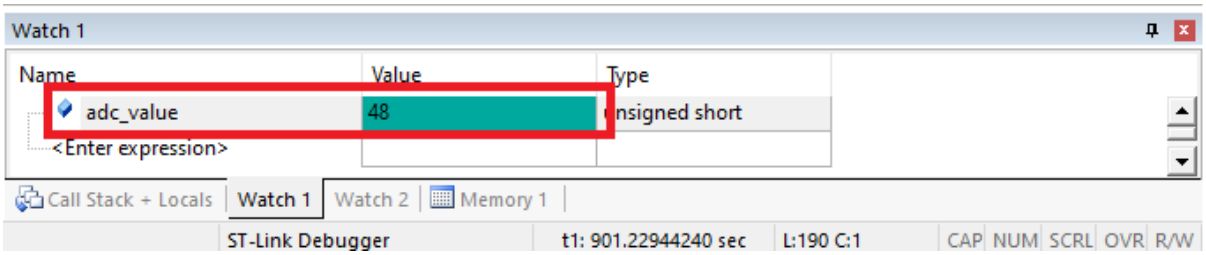


Click chuột phải tại biến đó, chọn Add '**adc_value'** tới **Watch 1.**

Biến 'adc_value' đã được thêm vào cửa sổ **Watch 1**, các bạn bắt đầu chạy Debug bằng cách click vào icon **Run** trên thanh công cụ.



Tại cửa sổ **Watch 1**, để theo dõi giá trị của biến ở dạng số thập phân, chúng ta click chuột phải và "**adc_value**", bỏ tích tại "**Hexadecimal Display**".



Điều chỉnh biến trở và quan sát sự thay đổi giá trị của biến "**adc_value**" nhé!

BÀI 04

DMA TRÊN STM32F4

I. Giới thiệu về DMA

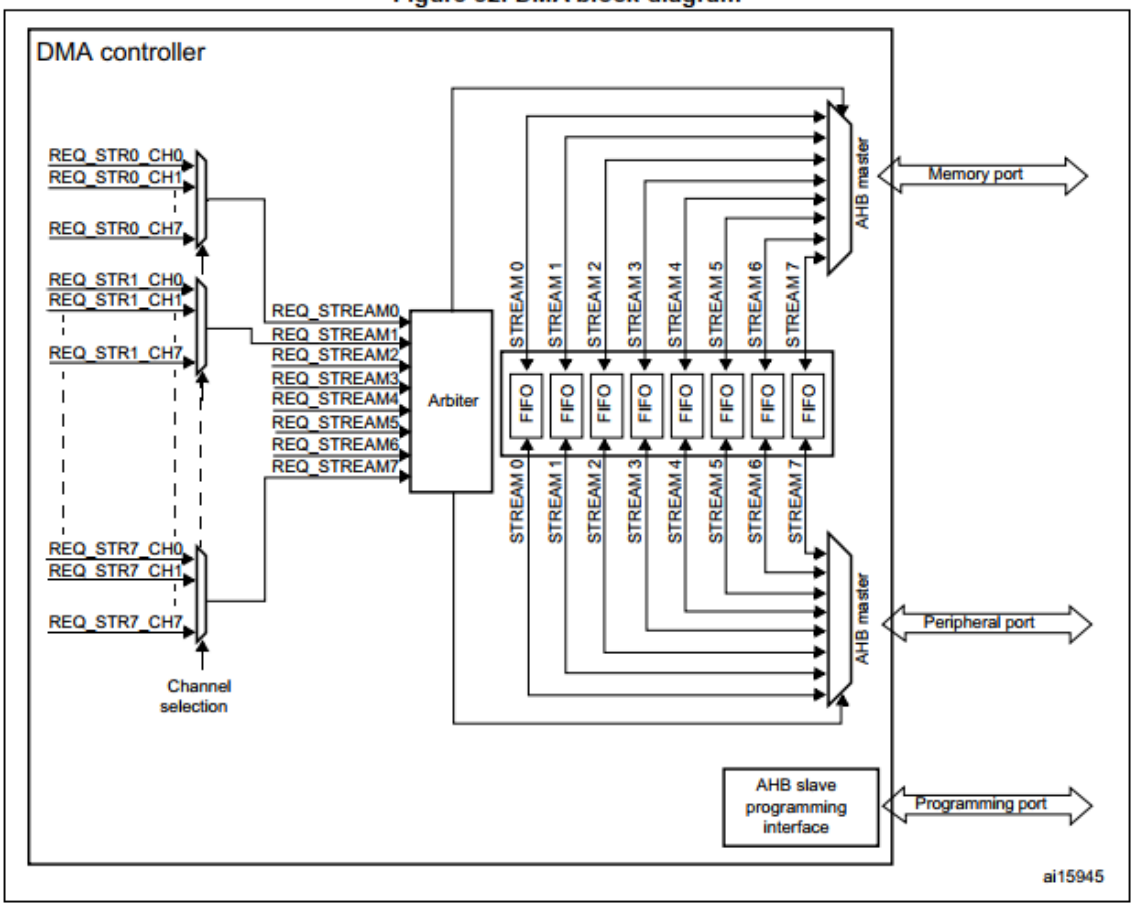
DMA – Direct memory access được sử dụng với mục đích truyền data với tốc độ cao từ thiết bị ngoại vi đến bộ nhớ cũng như từ bộ nhớ đến bộ nhớ.

Với DMA, dữ liệu sẽ được truyền đi nhanh chóng mà không cần đến bất kỳ sự tác động nào của CPU. Điều này sẽ giữ cho tài nguyên của CPU được rảnh rỗi cho các thao tác khác.

Bộ điều khiển DMA được kết nối với 2 bus AHB chính, cùng với cấu trúc FIFO độc lập để tối ưu hóa băng thông của hệ thống, hoạt động dưa trên kiến trúc ma trận bus khá phức tạp.

Đối với STM32F4, có 2 bộ điều khiển DMA, mỗi bộ DMA có 8 luồng, mỗi luồng đều có vai trò riêng để quản lý các yêu cầu truy cập bộ nhớ từ 1 hoặc nhiều ngoại vi. Mỗi luồng có thể có tối đa 8 kênh.

Figure 32. DMA block diagram



Các chức năng chính

- Các Channel đều có thể được cấu hình riêng biệt.
- Mỗi Channel được kết nối để dành riêng cho tín hiệu DMA từ các thiết bị ngoại vi hoặc tín hiệu từ bên trong MCU.
- Có 4 mức ưu tiên có thể lập trình cho mỗi Channel.
- Kích thước data được sử dụng là 1 Byte, 2 Byte (Half Word) hoặc 4byte (Word)
- Hỗ trợ việc lặp lại liên tục Data.
- 5 cờ báo ngắt (DMA Half Transfer, DMA Transfer complete, DMA Transfer Error, DMA FIFO Error, Direct Mode Error).
- Quyền truy cập tới Flash, SRAM, APB1, APB2, APB.
- Số lượng data có thể lập trình được lên tới 65535.
- Đối với DMA2, mỗi luồng đều hỗ trợ để chuyển dữ liệu từ bộ nhớ đến bộ nhớ.

Các ngoại vi có thể sử dụng DMA trên kit STM32F407

- SPI và I2S
- I2C
- USART
- Timer
- DAC
- SDIO
- Camera interface (DCMI)
- ADC

II. Cách thức hoạt động của DMA

Mỗi giao thức với DMA đều bao gồm 1 chuỗi dữ liệu cần chuyển nhất định. Số lượng của dữ liệu được truyền và độ lớn của chúng (8bit, 16bit hoặc 32bit) đều có thể lập trình được.

Mỗi lần chuyển dữ liệu với DMA bao gồm 3 bước :

- Tải dữ liệu từ thanh ghi ngoại vi hoặc trong bộ nhớ, được xử lý qua thanh ghi DMA_SxPAR hoặc DMA_SxM0AR.
- Lưu trữ dữ liệu vừa được tải vào thanh ghi dữ liệu hoặc 1 địa chỉ trong bộ nhớ, được xử lý thông qua thanh ghi DMA_SxPAR hoặc DMA_SxM0AR.
- Giảm dần số lần chuyển dữ liệu vẫn cần thực hiện trong thanh ghi DMA_SxNDTR.

Sau mỗi sự kiện, thiết bị ngoại vi sẽ gửi tín hiệu yêu cầu đến bộ điều khiển DMA. Bộ điều khiển DMA sẽ xử lý yêu cầu tùy thuộc vào mức độ ưu tiên của Channel. Ngay khi bộ điều khiển DMA truy cập vào thiết bị ngoại vi, DMA sẽ gửi 1 tín hiệu ACK đến ngoại vi. Lúc này,

thiết bị ngoại vi được giải phóng vì yêu cầu của nó đã được thực hiện. Nếu có thêm các sự kiện, thiết bị ngoại về có thể bắt đầu thực hiện các yêu cầu kế tiếp.

III. Một số thanh ghi quan trọng (x mang giá trị từ 0-7)

1. DMA stream x configuration register (DMA_SxCR)

Thanh ghi này được sử dụng để cấu hình các luồng DMA

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **PL**[1:0] : mức ưu tiên của channel tương ứng 00: thấp nhất, 11: cao nhất.
- **MSIZE**[1:0] : kích thước của bộ nhớ 00,01,10 tương ứng 8 bit, 16 bit, 32 bit.
- **PSIZE**[1:0] : kích thước ngoại vi 00,01,10 tương ứng 8 bit, 16 bit, 32 bit.
- **MINC**: có cho phép mode tăng địa chỉ bộ nhớ hay không.
- **PINC**: có cho phép mode tăng địa chỉ ngoại vi hay không.
- **CIRC**: có cho phép việc chuyển đổi được diễn ra liên tục hay không.
- **TEIE**: cho phép ngắt khi có lỗi trong quá trình truyền hay không.
- **HTIE**: cho phép ngắt khi truyền xong data ở chế độ half word.
- **TCIE**: cho phép ngắt khi truyền xong data ở chế độ word.
- **EN** : cho phép bộ DMA hoạt động hay không.

2. DMA stream x number of data register (DMA_SxNDTR)

Thanh ghi này có giá trị là 16bit tương ứng với 65535, chứa số lượng data truyền.

Address offset: 0x14 + 0x18 × *stream number*

Reset value: 0x0000 0000

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
NDT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

3. DMA stream x peripheral address register (DMA_SxPAR)

Thanh ghi này là thanh ghi 32bit chứa địa chỉ của ngoại vi.

Address offset: 0x18 + 0x18 × *stream number*

Reset value: 0x0000 0000

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

4. DMA stream x memory 0 address register (DMA_SxM0AR)

Address offset: 0x1C + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
M0A[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M0A[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Thanh ghi 32bit này chứa địa chỉ của vùng nhớ thứ 0, nơi dữ liệu được đọc/ghi.

Những bit trong thanh ghi này chỉ được ghi lên nếu:

- Luồng đang bị Disable.
- Hoặc luồng đang được enable nhưng bit CT = '1' (trong thanh ghi DMA_SxCR ở chế độ Double buffer mode).

IV. LẬP TRÌNH NHIỀU KÊNH ADC TRÊN STM32F4 SỬ DỤNG DMA

Trong phần trên về ADC, chúng ta đã sử dụng chức năng ADC trên vi điều khiển STM32F407VG và sử dụng ngắt trên 1 kênh. Phần này, mình sẽ hướng dẫn các bạn đọc giá trị ADC trên nhiều kênh khác nhau, kết hợp với DMA để lưu kết quả thu được vào bộ nhớ.

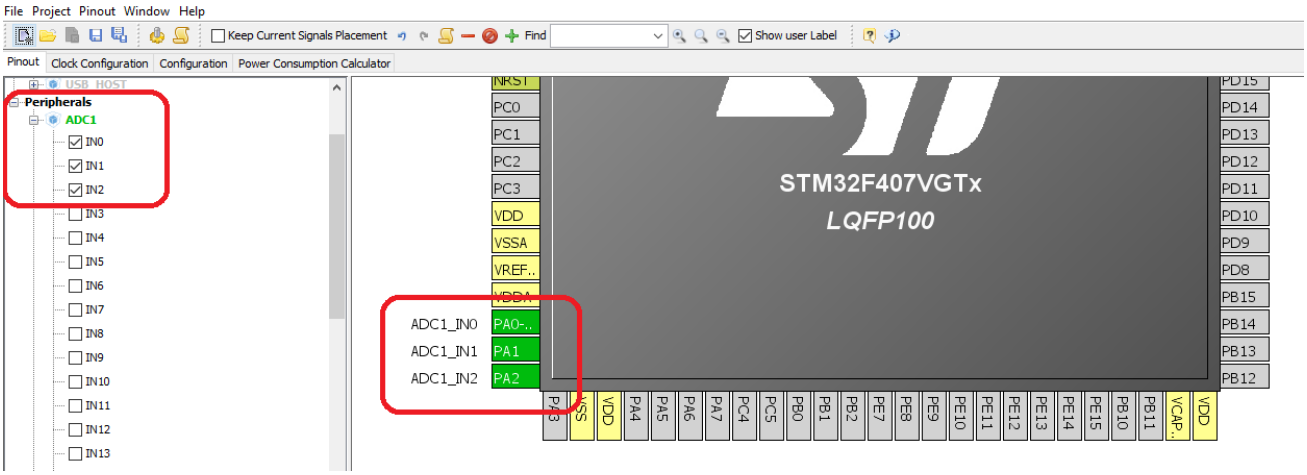
Mình sẽ sử dụng 3 Channel 0, 1, 2 của bộ ADC1 trên kit STM32F407VG để đọc giá trị từ 3 điện trở khác nhau. Các kênh này được kết nối lần lượt với các chân PA0, PA1, PA2 của KIT.

1. Cấu hình với CubeMX

Khởi động phần mềm CubeMX, chọn vi điều khiển STM32F407VG.

Tại phần "**Peripherals**", trong mục ADC1, chúng ta click chọn "**IN0**", "**IN1**", "**IN2**" để kích hoạt các channel 0, 1, 2 của bộ ADC1.

Hoặc các bạn có thể tìm đến các chân PA0, PA1, PA2 và cấu hình cho từng chân.



+ Cấu hình thạch anh và clock cho chip

- Tại mục RCC, trong phần "**High Speed Clock(HSE)**", chúng ta chọn "**Crystal/Ceramic Resonator**" để cấu hình cho chip hoạt động với thạch anh ngoại.

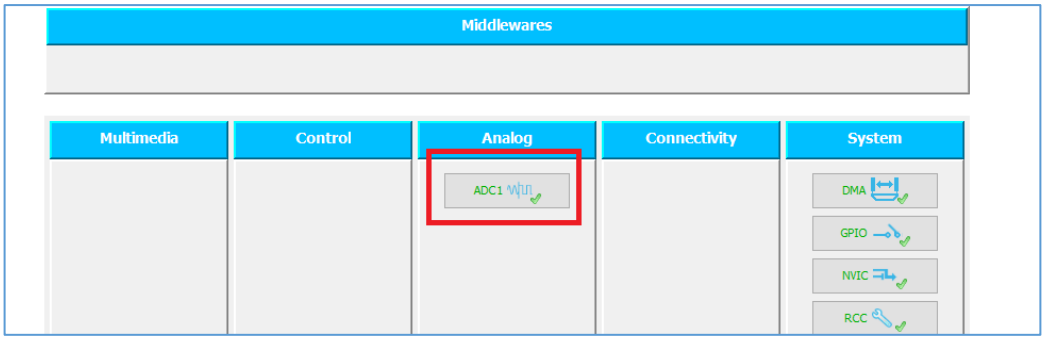
- Trong Tab "**Clock Configuration**", các bạn thiết lập các thông số cho Clock của Chip.

Chọn HSE, xung clock sẽ đi qua bộ nhân tần PLLCLK, giúp Chip hoạt động với tần số cao nhất.

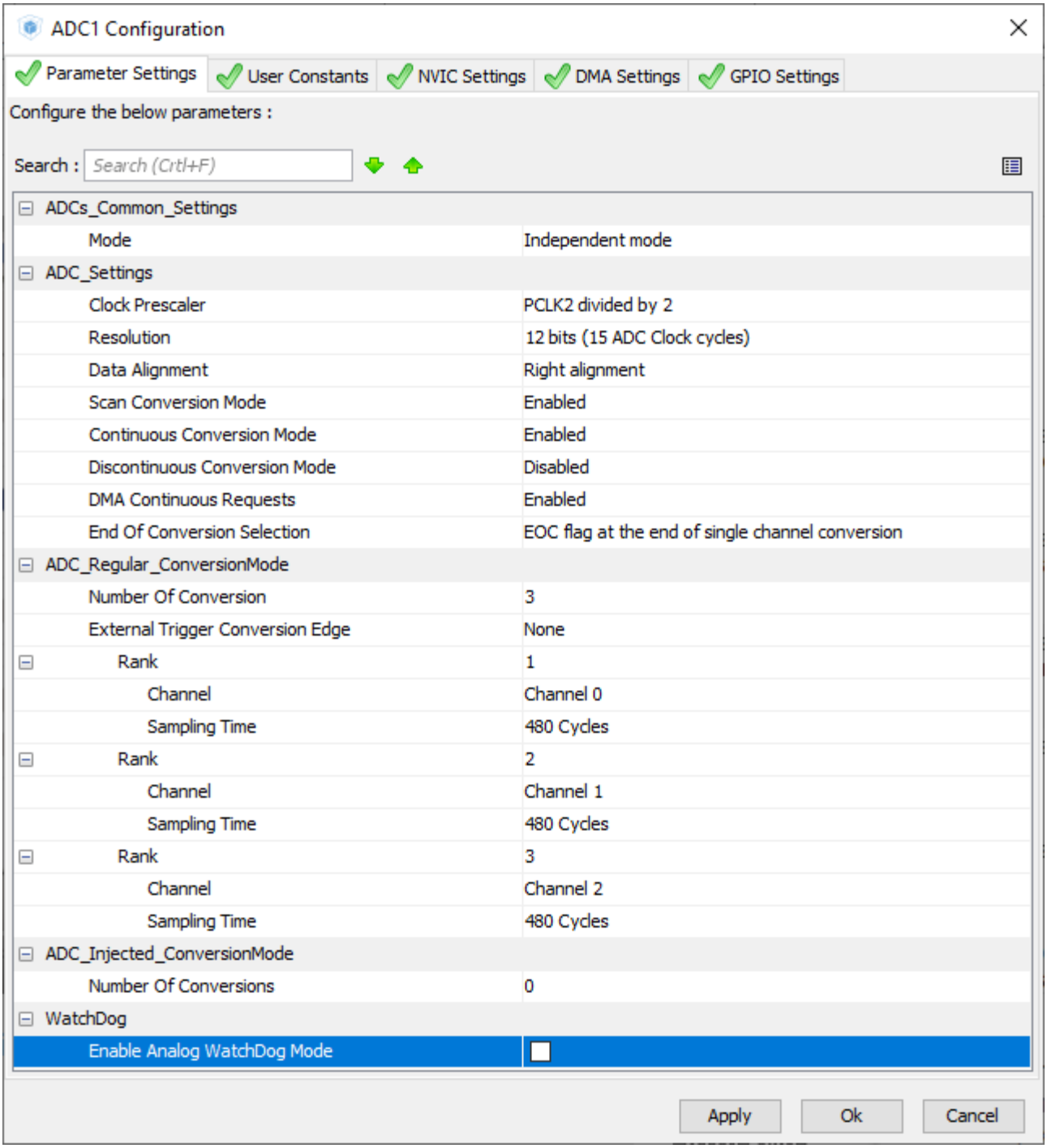
Điền "8"ở mục "**Input frequency**" (thông số của thạch anh ngoại gắn trên KIT của mình) và điền "168"ở mục "HCLK" (168MHz là tần số tối đa của chip).

+ Cấu hình cho bộ ADC:

Chuyển qua Tab "**Configuration**", chúng ta chọn mục ADC1:



Khi cửa sổ "**ADC1 Configuration**" hiện lên, các bạn thiết lập các thông số như hình dưới đây

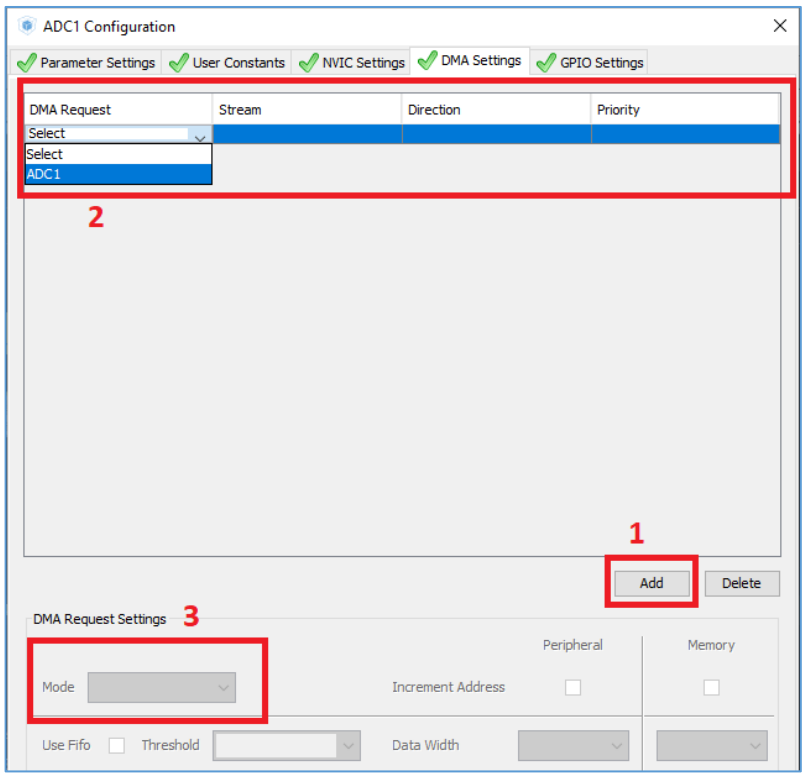


Mình sẽ giải thích qua một chút về các thông số đáng chú ý:

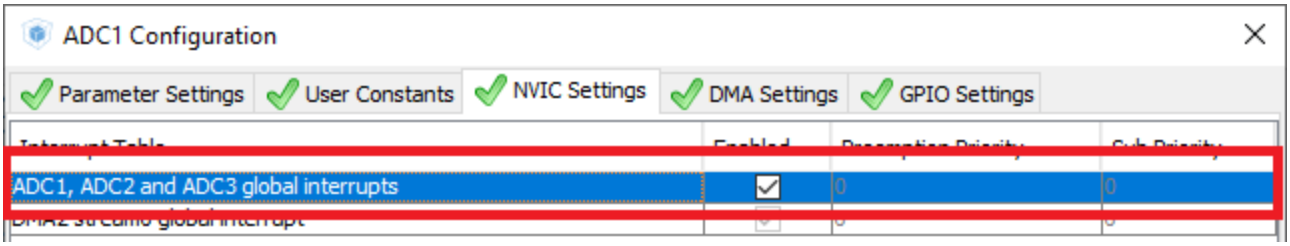
- **Scan Conversion Mode**: khi sử dụng 1 bộ ADC nhưng cần lấy mẫu trên nhiều kênh, chúng ta cần Enable cho chế độ này để quét liên tục các kênh
- **Continuous Conversion Mode**: cho phép lấy mẫu liên tục
- **DMA Continuous Requests**: ở bài này chúng ta sử dụng DMA nên sẽ phải Enable chức năng này.
- **Number Of Conversion**: số chuyển đổi được thực hiện, ở đây mình sẽ điền "3" vì sử dụng 3 kênh ADC.
- **Rank**: thiết lập mức ưu tiên cho các kênh ADC, kênh có rank nhỏ hơn thì độ ưu tiên cao hơn.

Tại Tab "**DMA Settings**", chúng ta sẽ chọn "1- **Add**", sau đó chọn "2- **ADC1**" ở mục "DMA Request". Tiếp theo, tại mục "3 –**Mode**", các bạn chọn "Circular". Ở mục Mode này có 2 lựa chọn là "**Normal**" và "**Circular**":

- **Normal**: tại tất cả các lần lấy mẫu, DMA đều lưu trữ kết quả vào cùng 1 vị trí trong bộ nhớ, bị ghi đè lên nhau.
- **Circulal**: các lần lấy mẫu sẽ được lưu vào các vị trí kế tiếp nhau trong bộ nhớ, không bị ghi đè lên nhau.

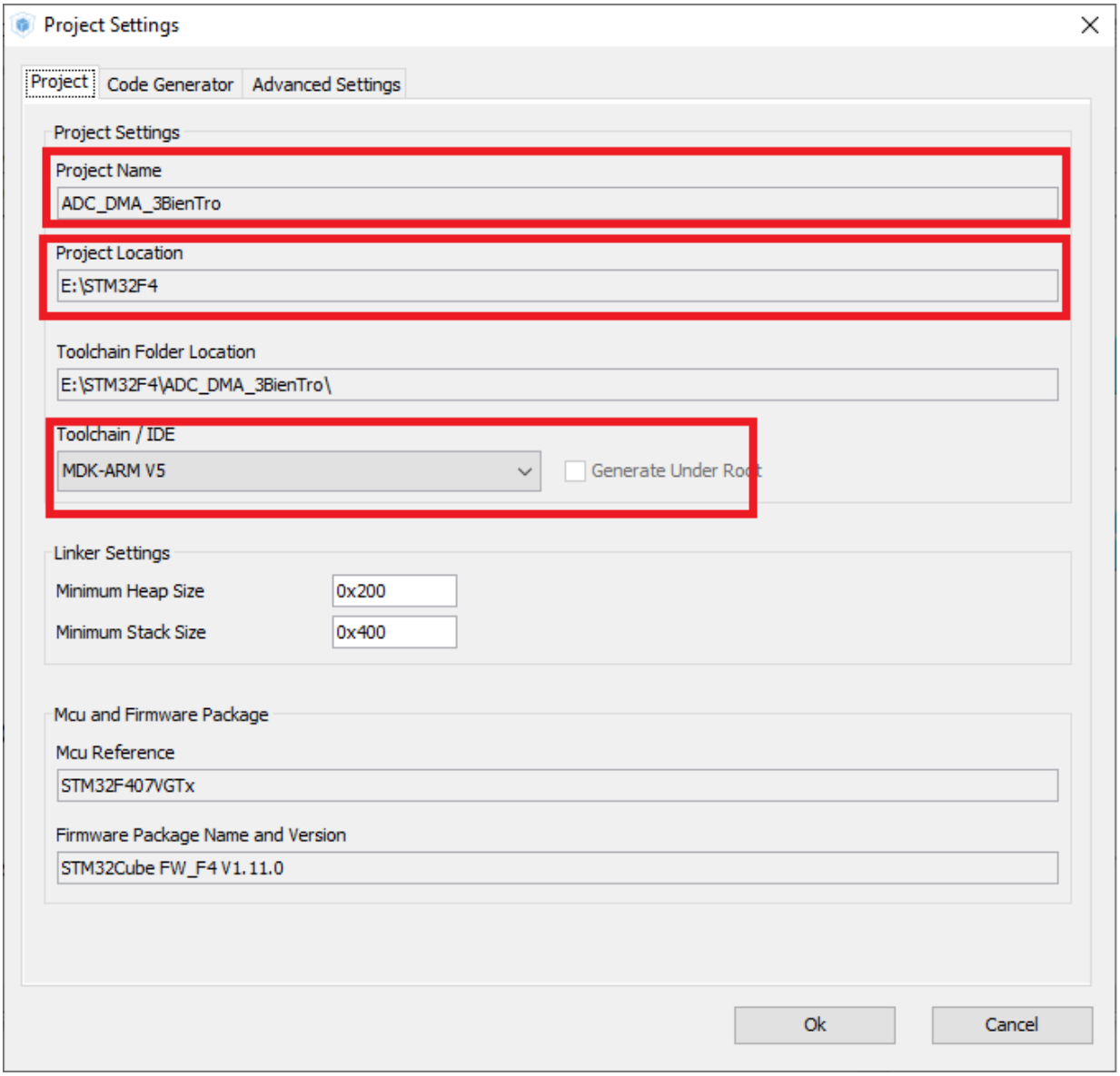


Chuyển qua Tab "**NVIC Settings**", chúng ta sẽ Enable cho phép ngắt đối với các bộ ADC.

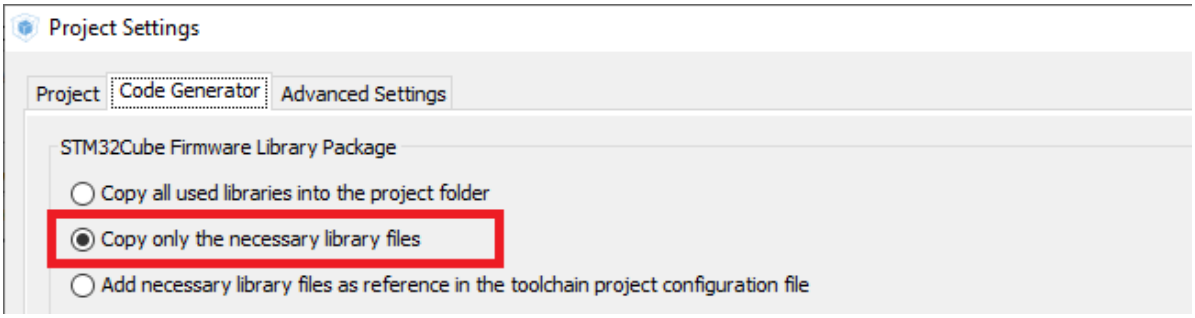


Setting project và sinh code

Các bạn thiết lập các thông tin cơ bản cho Project



Chọn mục "**Copy only the necessary library files**" trong Tab "**Code Generator**" để Project sinh ra chỉ có những thư viện cần thiết, giảm dung lượng Project và tiết kiệm thời gian Build code nhé!



2. Lập trình với KeilC

Sau khi CubeMX sinh code xong, chúng ta hãy chọn "**Open Project**" để mở chương trình và lập trình với KeilC.

Trong file "main.c", chúng ta khai báo 1 mảng để lưu giá trị các giá trị nhận về sau khi quét ADC : volatile uint16_t res_value**[**3**];**

- **res_value**[0] : lưu giá trị nhận về của kênh 0
- **res_value**[1] : lưu giá trị nhận về của kênh 1
- **res_value**[2] : lưu giá trị nhận về của kênh 2

Ở đây mình sử dụng "volatile" tránh những lỗi sai khó phát hiện do tính năng Optimization của Compiler vì giá trị của biến "adc_value" sẽ bị thay đổi giá trị do tác động của phần cứng.

Trong hàm "main", mình sẽ viết câu lệnh sau :

```
HAL_ADC_Start_DMA(&hadc1, (uint32_t *) res_value, 3);
```

Câu lệnh này cho phép kích hoạt bộ ADC1 và sử dụng DMA, lưu giá trị nhận về vào mảng "res_value", "3" ở đây là độ dài của dữ liệu chuyển đổi ADC nhận về từ ngoại vi đến bộ nhớ.

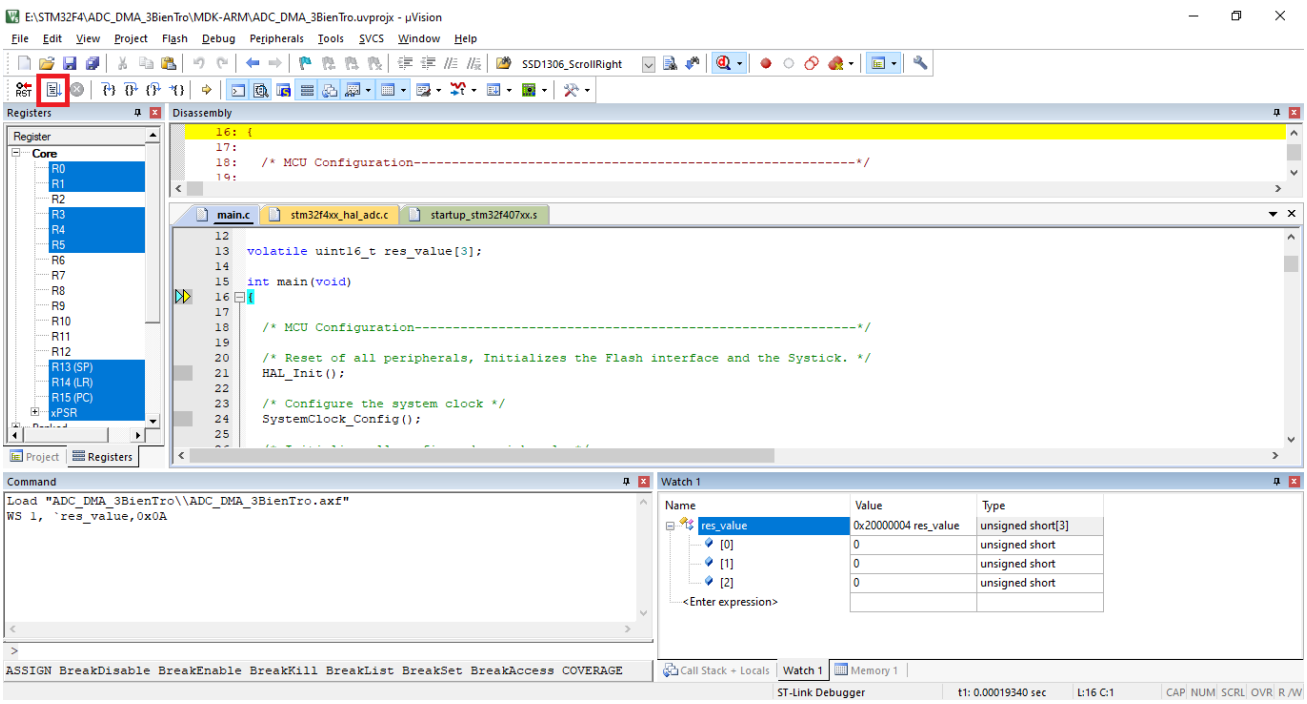
Lưu ý ép kiểu **(**uint32_t ***)** cho biến "res_value" để cùng định dạng dữ liệu với biến cục bộ được khai báo khi khởi tạo hàm HAL_ADC_Start_DMA()

```
volatile uint16_t res_value[3];

int main(void)
{

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_ADC1_Init();

  HAL_ADC_Start_DMA(&hadc1, (uint32_t *) res_value, 3);

  while (1)
  {
  }
  /* USER CODE END 3 */
}
```

Chúng ta build chương trình (**F7**) và load code xuống kit (**F8**), sau đó nhấn nút reset kit

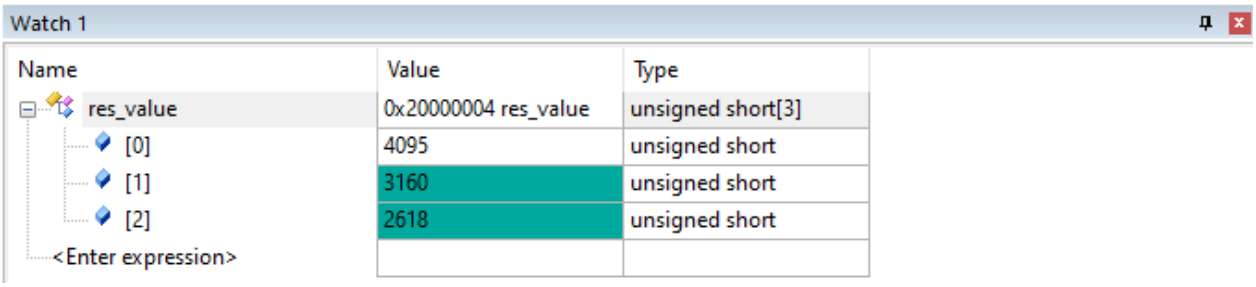
Kết nối các chân 2 của biến trở với các chân PA0, PA1, PA2, các chân số 1 với GND và các chân số 3 với 3V.

Tiếp theo, chúng ta bật chế độ Debug để quan sát các giá trị nhận về.



Điều chỉnh biến trở.

Tại cửa sổ Watch1, chúng ta sẽ quan sát sự thay đổi giá trị của các biến. Muốn xem giá trị các biến ở hệ thập phân, các bạn click chuột phải vào "res_value" tại đây và bỏ chọn "**Hexdecimal Display**"



Trên đây là 1 project nhỏ để các bạn cùng làm quen với quét nhiều kênh ADC trên kit STM32F407VG, sử dụng DMA.

BÀI 5

LẬP TRÌNH TIMER CƠ BẢN VỚI STM32

Timer (Bộ định thời) là ngoại vi không thể thiếu đối với các dòng vi điều khiển. Đây là khối thực hiện nhiều chức năng quan trọng như làm bộ đếm, phát hiện, đo tín hiệu đầu vào, tạo xung PWM, điều khiển và cấp xung cho các thiết bị bên ngoài, định thời các sự kiện đặc biệt.

I. Lý thuyết

Dòng vi điều khiển STM32 có ba loại Timer:

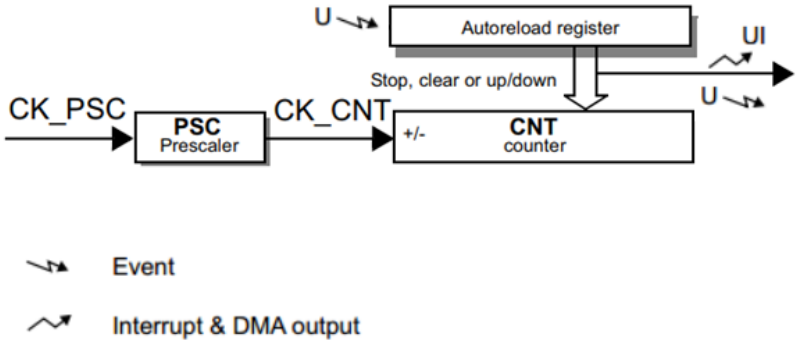
- **Basic Timer**: là loại Timer đơn giản và dễ sử dụng nhất, chỉ có chức năng đếm và thường được dùng để tạo cơ sở thời gian.
- **General Purpose Timer**: là loại Timer nhiều tính năng hơn Basic Timer, có đầy đủ các tính năng của một bộ định thời như đếm thời gian, tạo xung PWM, xử lí tín hiệu vào, so sánh đầu ra, …
- **Advanced Timer**: đây là loại Timer nâng cao, mang đầy đủ đặc điểm của General Purpose Timer, ngoài ra còn có nhiều tính năng khác và độ chính xác cao hơn. Thường được sử dụng để làm bộ đếm thời gian cho hệ thống.

STM32F411 có 8 bộ Timer, trong đó có 1 bộ Advanced – control timer (TIM1) thường được các bộ thư viện sử dụng để tạo bộ đếm thời gian chuẩn của hệ thống (như ngắt System Tick, các hàm tạo Delay, TimeOut…), và 7 bộ General – purpose timer (TIM2 đến TIM5 và TIM9 đến TIM11).

Timer	Type	Resolution	Prescaler	Channels	MAX INTERFACE CLOCK	MAX TIMER CLOCK*	APB
TIM1	Advanced	16bit	16bit	4	SysClk/2	SysClk	2
TIM2, TIM5	General purpose	32bit	16bit	4	SysClk/4	SysClk, SysClk/2	1
TIM3, TIM4	General purpose	16bit	16bit	4	SysClk/4	SysClk, SysClk/2	1
TIM9	General purpose	16bit	16bit	2	SysClk/2	SysClk	2
TIM10, TIM11	General purpose	16bit	16bit	1	SysClk/2	SysClk	2

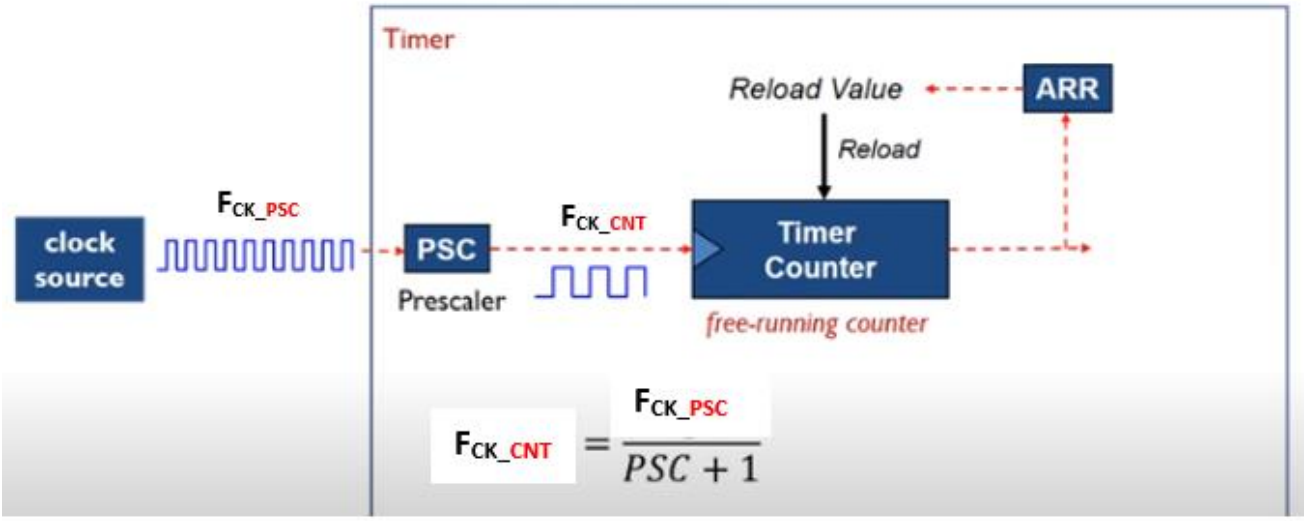
Cấu trúc cơ bản của một bộ Timer

- Bộ đếm- Counter (Giá trị được lưu ở thanh ghi Counter Register)
- Giá trị Auto Reload (Giá trị được lưu ở thanh ghi Auto Reload)
- Bộ chia tần - Prescaler (Giá trị được lưu ở thanh ghi Prescale)



Các thanh ghi quan trọng:

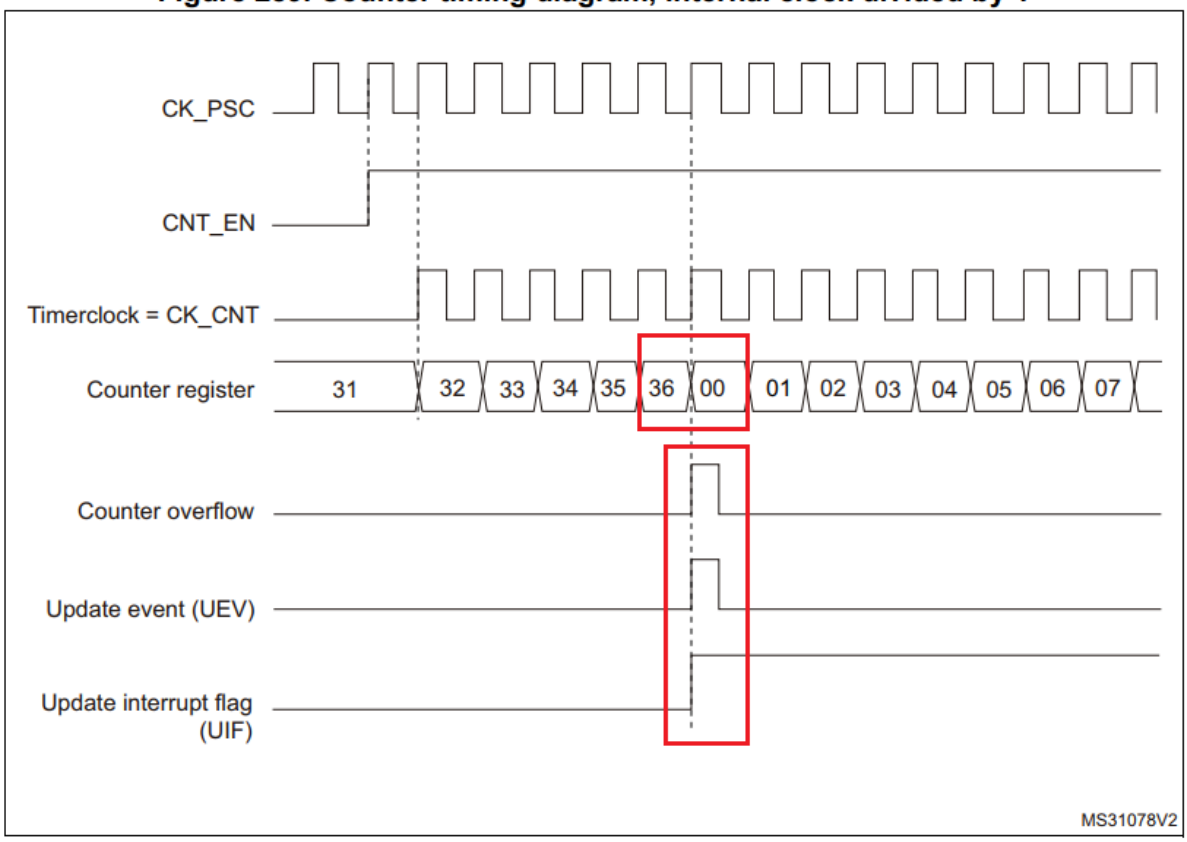
- **Auto Reload**(TIMx_ARR): Lưu giá trị Auto Reload (ARR), là ngưỡng trên của giá trị đếm Counter (ví dụ: ta muốn đếm từ 0 đến 9, rồi lặp lại → ngưỡng trên là 9).
- **Counter Register**(TIMx_CNT): Lưu giá trị đếm Counter (CNT), tăng hoặc giảm mỗi nhịp xung clock của Timer. Giá trị của Counter luôn nằm trong khoảng [0; ARR]. Nếu ngoài khoảng đó, Timer sẽ thực hiện nạp lại giá trị CNT như ban đầu và tiếp tục hoạt động. Tùy vào mỗi Timer mà CNT và ARR có cỡ 16 hoặc 32 bit.
- **Prescaler** (TIMx_PSC): Lưu giá trị chia tần PSC (16 bit), thuộc khoảng [1;65536]. Kết hợp việc sử dụng hai giá trị PSC và ARR, chúng ta có thể tính toán được tần số, chu kì đếm của Timer.



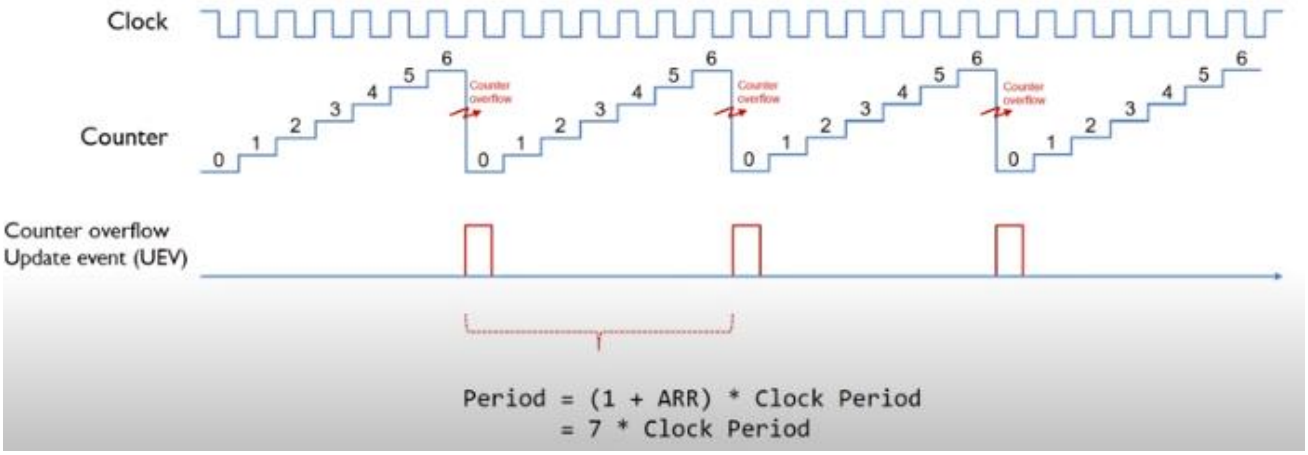
Các chế độ đếm của Timer:

Upcounting Mode: CNT bắt đầu tăng từ giá trị 0 đến giá trị ARR. Sau đó, CNT quay trở lại giá trị 0, đồng thời sinh ra sự kiện tràn trên(counter overflow event).

Figure 289. Counter timing diagram, internal clock divided by 1



Upcounting mode với ARR=36



Upcounting mode với ARR=6,

$F_{CK\_CNT} = \frac{F_{CK\_PSC}}{PSC+1}$ *, Tần số đếm, Trong đó* F_{CK_PSC} *là tần số Clock cấp cho Timer*

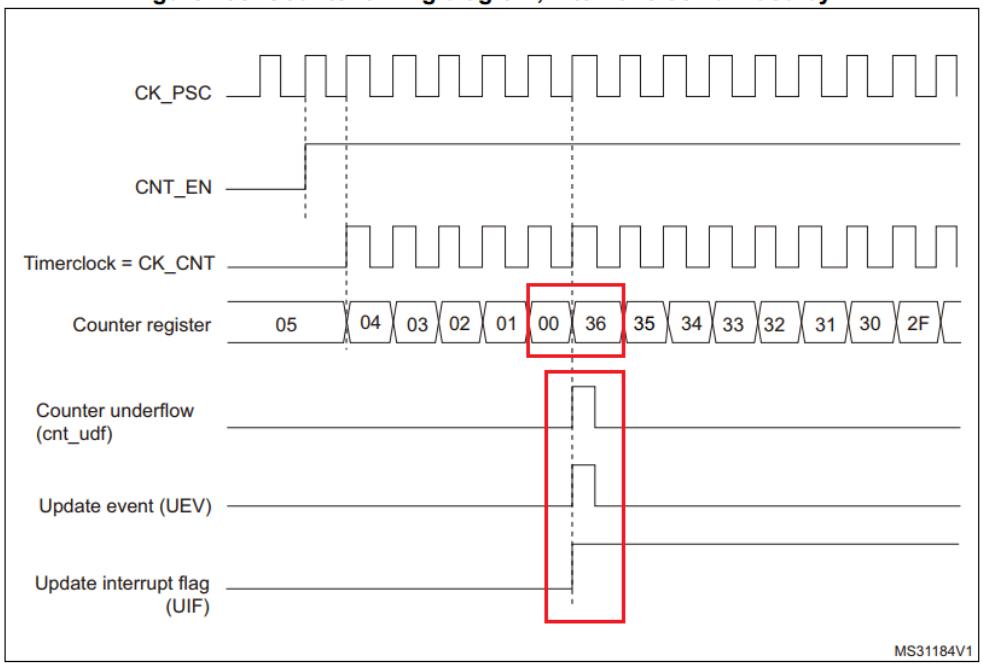
$\rightarrow T_{CK\_CNT} = Clock\ period = TimerClock = \frac{1}{F_{CK\_CNT}} = \frac{PSC+1}{F_{CK\_PSC}}$ *(Thời gian đếm)*

➔ Khoảng Thời gian mỗi lần phát sinh sự kiện là

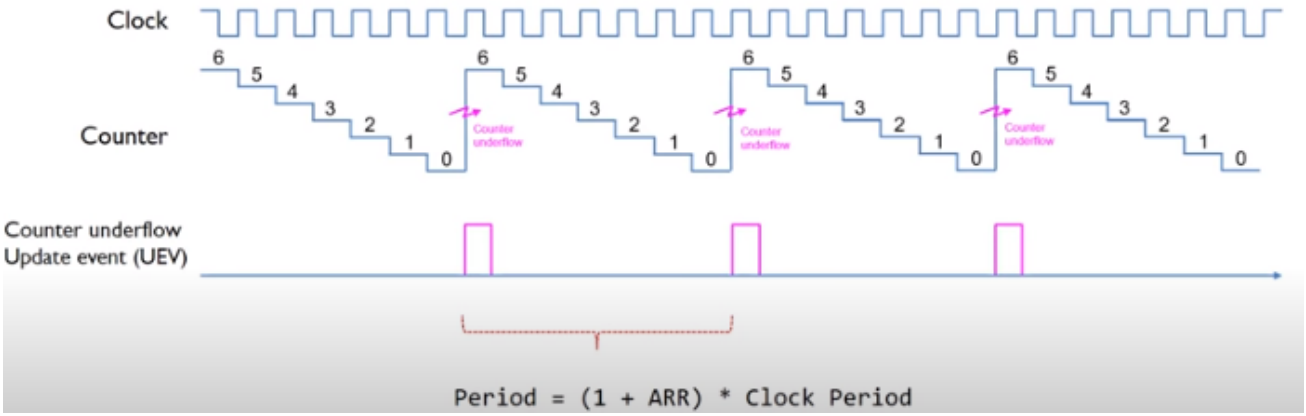
$$T_{UEV} = Period = (1 + ARR) * T_{CK_{CNT}} = \frac{(1 + ARR) * (1 + PSC)}{F_{CK\_PSC}}$$

Downcounting mode: CNT bắt đầu giảm từ giá trị ARR về 0. Sau khi về 0, CNT quay trở lại giá trị ARR, đồng thời sinh ra sự kiện tràn dưới(counter underflow event).

Figure 295. Counter timing diagram, internal clock divided by 1



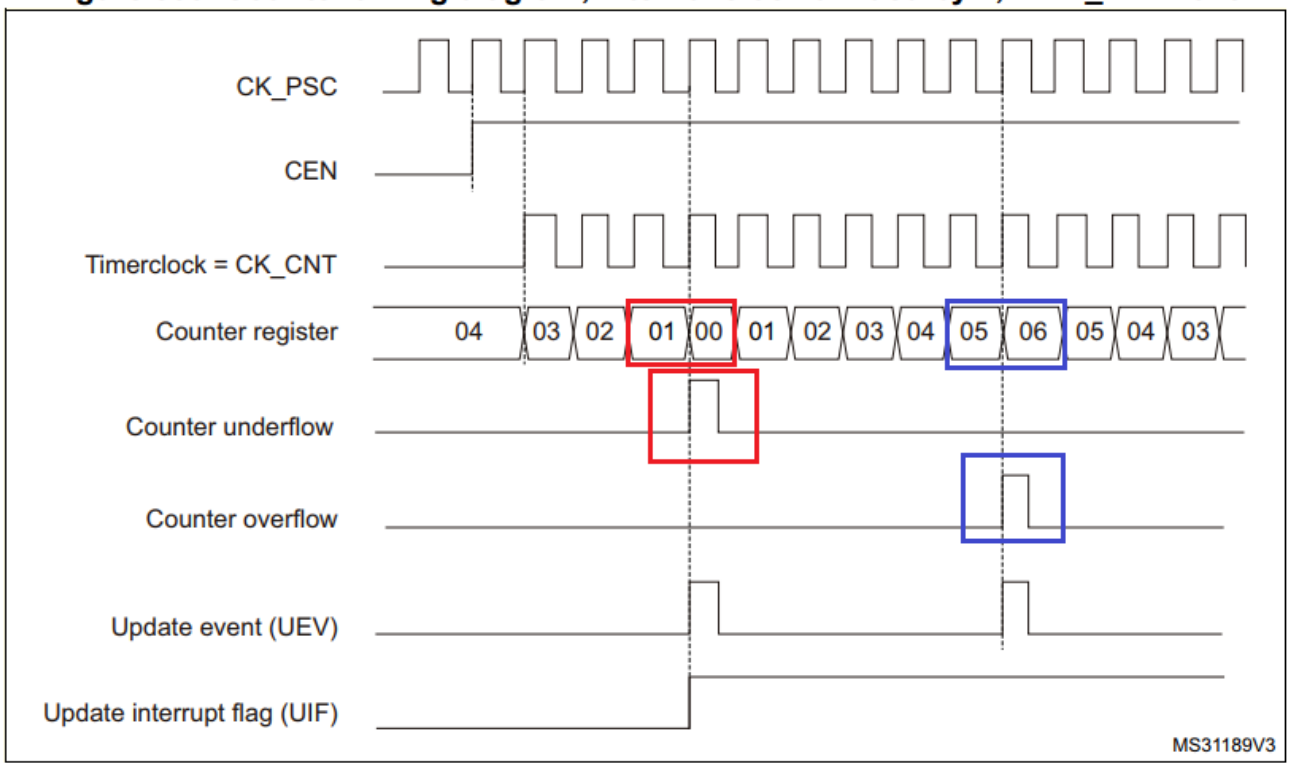
Downcounting mode với ARR=36



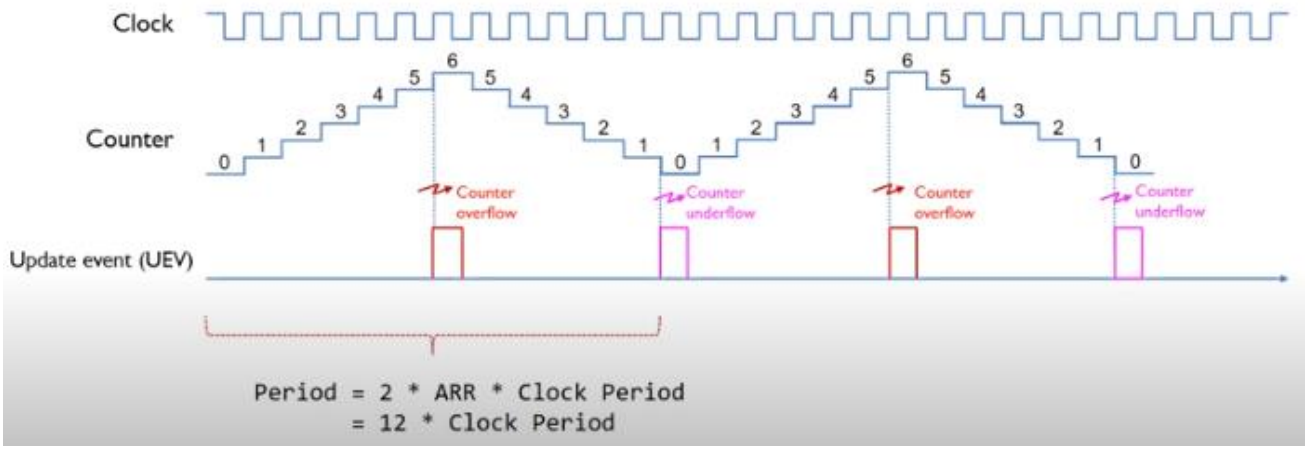
Upcounting mode với ARR=6

Center-aligned mode (up/down counting): ở chế độ đếm này, CNT bắt đầu tăng từ 0, đến giá trị ARR-1, khi đó sinh ra sự kiện tràn trên(counter overflow event). Sau đó CNT đạt giá trị ARR và bắt đầu giảm về giá trị 1, khi đó sinh ra sự kiện tràn dưới(counter underflow event). Cuối cùng, CNT về giá trị 0 và quá trình trên lại tiếp tục.

Figure 300. Counter timing diagram, internal clock divided by 1, TIMx_ARR=0x6



Center-aligned mode với ARR=6



Một vài chức năng thường xuyên được sử dụng của Timer:

Ngoại trừ các Basic Timer chỉ có hoạt động cơ bản là đếm, các Timers còn lại của vi điều khiển còn có nhiều chức năng khác, điển hình như:

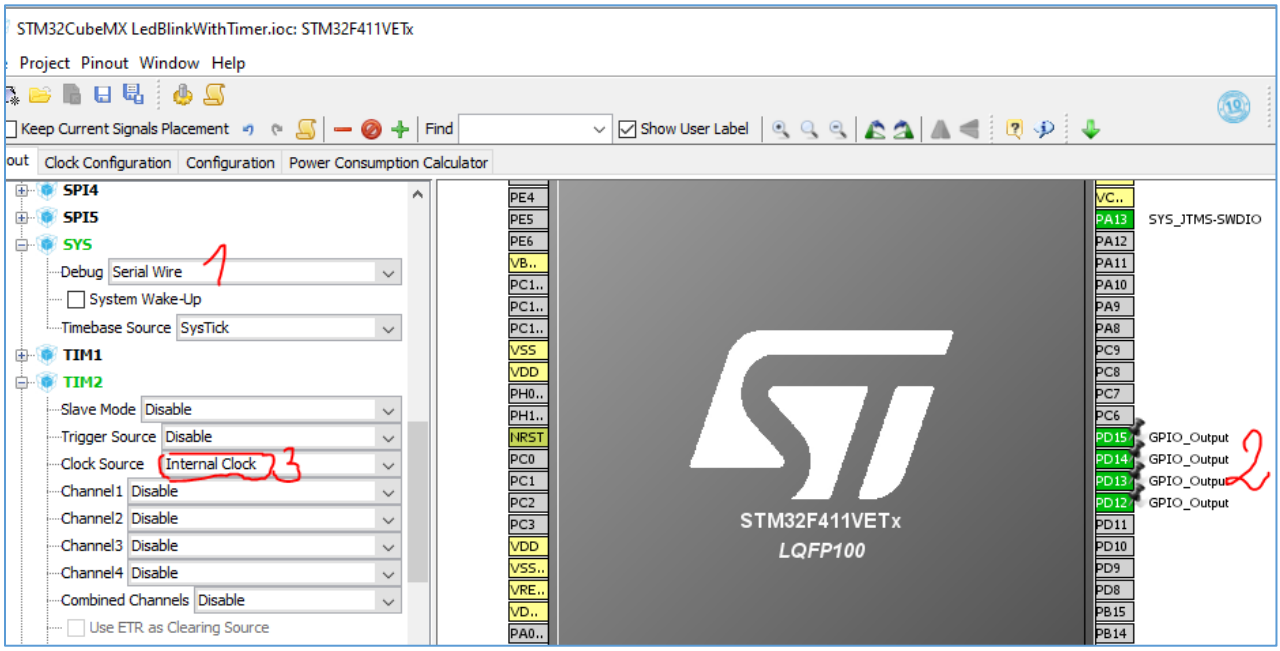
- **PWM Generation**: Tính năng điều chế độ rộng xung (băm xung).
- **One Pulse Mode**: Tạo ra một xung duy nhất với độ rộng có thể cấu hình được, CNT sẽ tự động dừng khi có sự kiện tràn.
- **Input Capture**: Chế độ này phát hiện và lưu lại sự xuất hiện sự thay đổi mức logic (sườn lên/ sườn xuống) của tín hiệu. Từ đó, ta có thể biết được khoảng thời gian giữa hai lần có sườn lên/ sườn xuống.
- **Output Compare**: Đây là chế độ giúp tạo ra các sự kiện(ví dụ như ngắt) khi CNT đạt đến giá trị được lưu trong các thanh ghi TIMx_CCMRx (capture/compare mode register). Ứng dụng phổ biến nhất của Output Compare là tạo ra nhiều xung PWM với các tần số khác nhau trên cùng một Timer.

II. Lập trình cơ bản với Timer (Basic Timer)

Ở bài này, chúng ta sẽ làm ví dụ đơn giản nhất với Timer, đó là dùng Timer để điều khiển một LED sáng 1s, sau đó tắt 1s, rồi lại sáng, tắt, … Lập trình các tính năng nâng cao hơn như PWM, Input Capture, Output Compare, … sẽ được nói đến ở những bài sau.

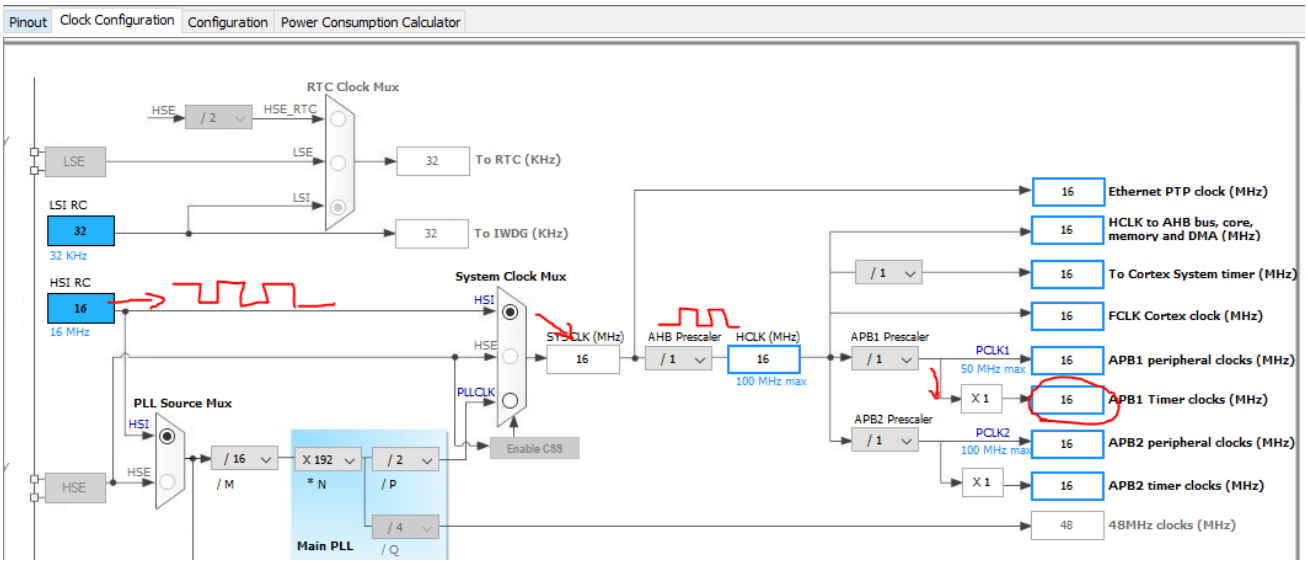
Cấu hình Chip với Cube Mx:

Mở phần mềm Cube Mx, chọn dòng chip các bạn sử dụng. Ở đây, ta chọn chip STM32F411VE



Cấu hình Timer 2

Clock Source cho Timer ở đây ta chọn là **Internal Clock,** sang Tab cấu hình Clock, ta thấy thông qua bộ chia tần, cấu hình tần số cho các Timer là 16MHz

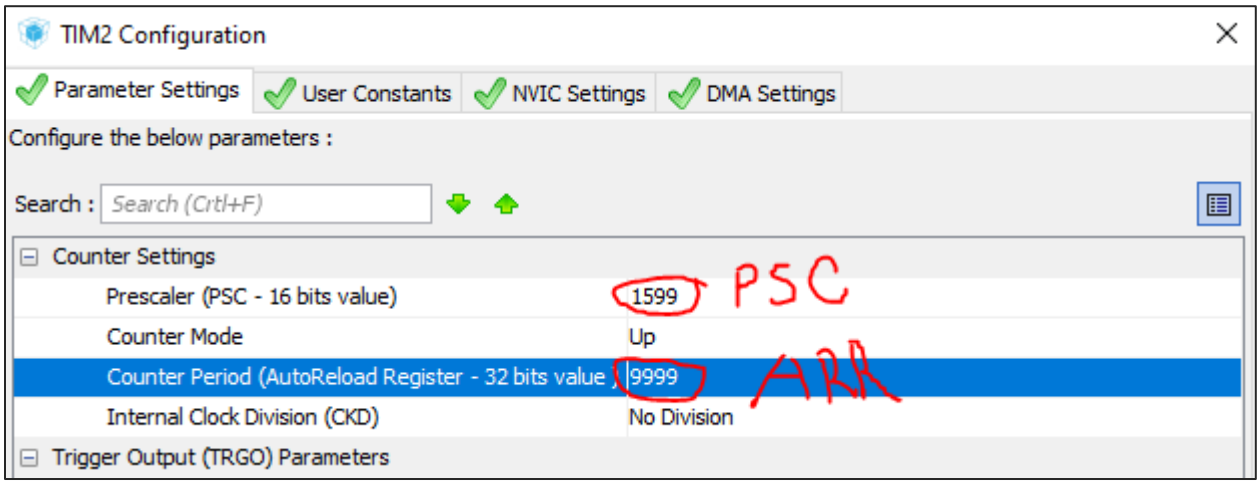


Áp dụng công thức

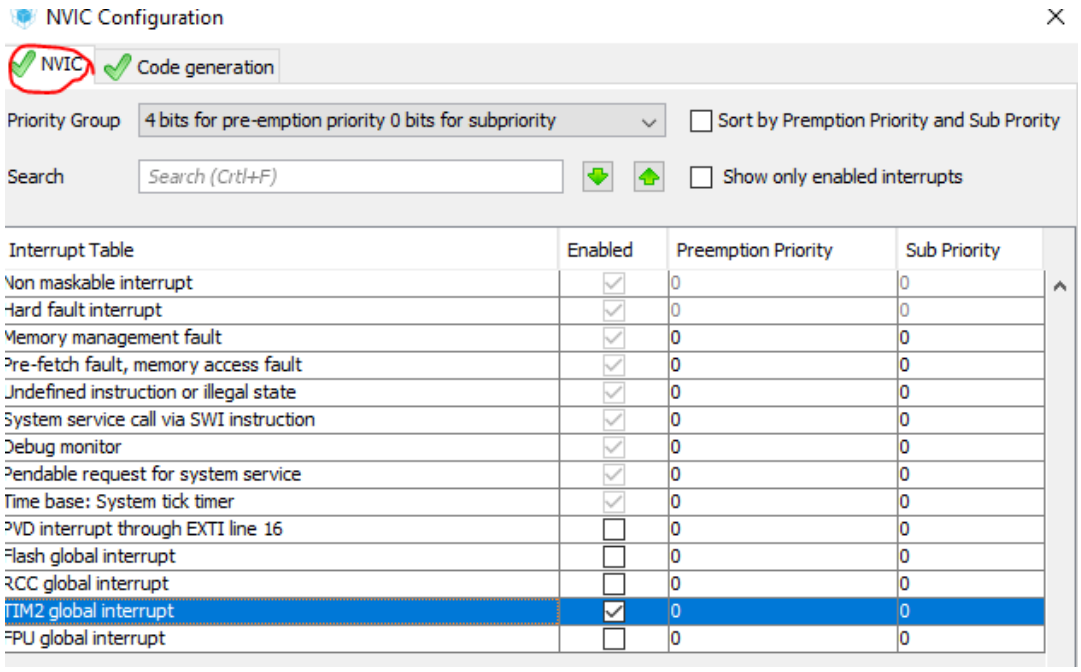
$$T_{UEV} = Period = (1 + ARR) * T_{CK_{CNT}} = \frac{(1 + ARR) * (1 + PSC)}{F_{CK\_PSC}}$$

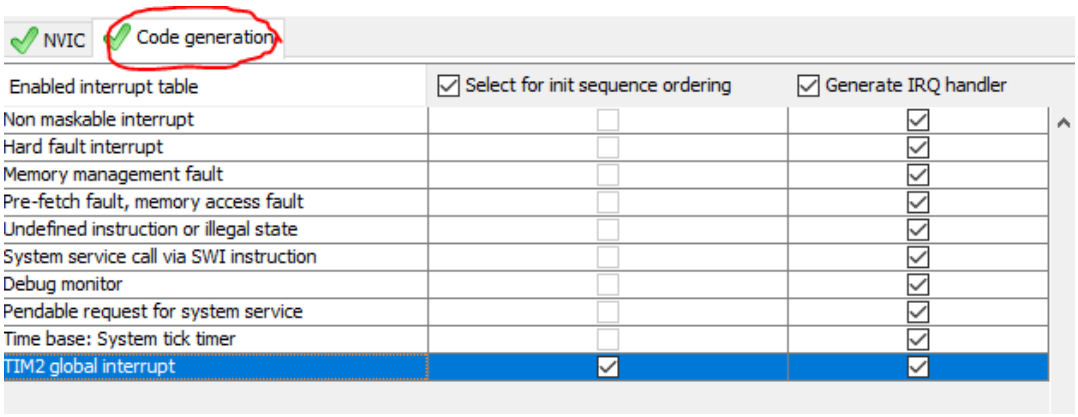
Như vậy, để thực hiện bài toán, sáng/tắt LED sau mỗi 1s, thì T_{UEV}=1

Với F_{CK_PSC}=16MHz = 16.000.000 Hz; ta có thể chọn PSC=1599, và ARR = 9999

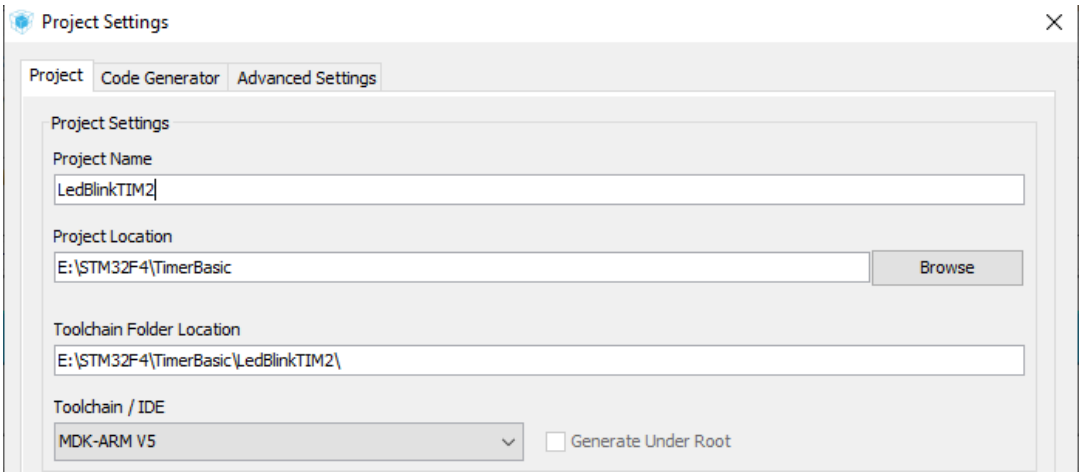


Bật ngắt cho Timer 2





Cấu hình Project sinh mã, (Copy only the necessary library file**)**



Một số hàm làm việc cơ bản

```
HAL_TIM_Base_Start(TIM_HandleTypeDef *htim)
```

Hàm HAL_TIM_Base_Start_IT là hàm cho phép bắt đầu chạy TIM2 đồng thời Enable ngắt tràn cho TIM2, bất cứ khi nào CNT của TIM2 tăng quá giá trị được đặt trong **ARR** (ví dụ này là 9999) nó sẽ được reset về 0 đồng thời tạo ra 1 ngắt tràn, và triệu gọi thực hiện chương trình con phục vụ ngắt .

```
HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
```

Hàm HAL_TIM_PeriodElapsedCallback trong khai báo ở thư viện HAL là hàm weak, có nghĩa là nó cần được code lại cụ thể ở chương trình.

```
4   TIM_HandleTypeDef htim2;
5   void SystemClock_Config(void);
6   static void MX_GPIO_Init(void);
7   static void MX_TIM2_Init(void);
8   int main(void)
9   {
10    HAL_Init();
11    SystemClock_Config();
12    MX_GPIO_Init();
13    MX_TIM2_Init();
14    HAL_TIM_Base_Start(&htim2);
15    while (1)
16    {
17    }
18  }
```

```
66  /* USER CODE BEGIN 0 */
67  void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
68  {
69    if(htim->Instance == TIM2)
70    {
71      HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
72    }
73  }
74  /* USER CODE END 0 */
```

Bài 6

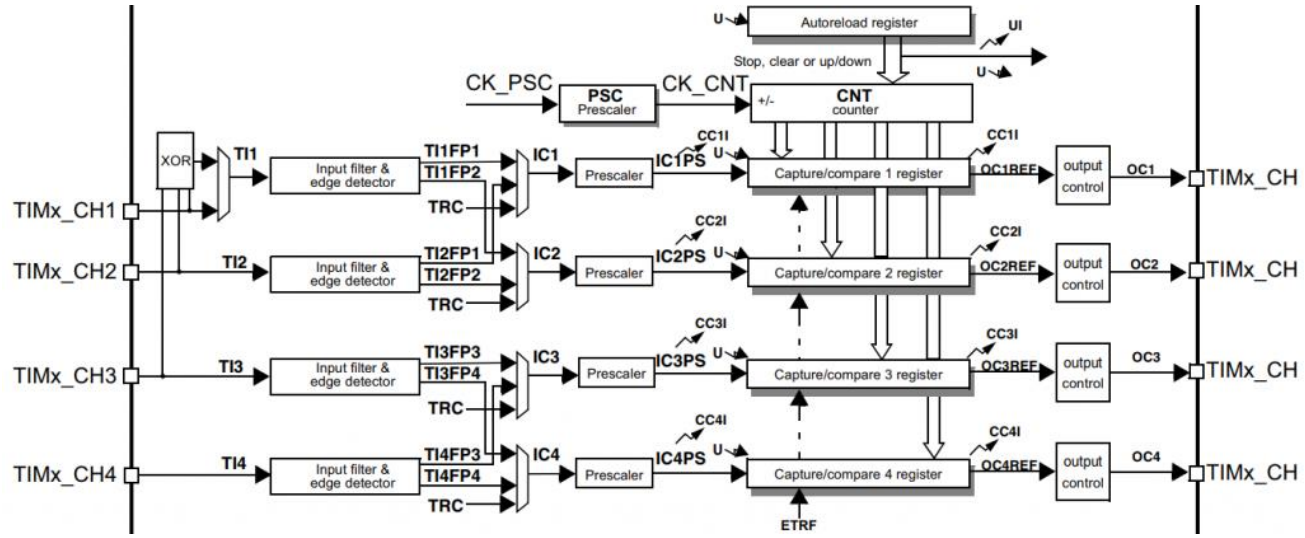
PWM VỚI STM32

PWM (Pulse Width Modulation) – Điều chế độ rộng xung/Băm xung: là phương pháp điều chỉnh giá trị điện áp trung bình ra tải như các thiết bị như động cơ, đèn LED,… từ đó có thể làm thay đổi công suất thiết bị (tốc độ động cơ, độ sáng của đèn,…). Điều này được thực hiện bằng cách thay đổi Duty Cycle của xung tín hiệu điện áp ra, là phương pháp dễ dàng và ít tốn kém hơn việc điều chỉnh các thông số của dòng điện.

I. Giới thiệu General Purpose Timer

General Purpose Timer: là loại Timer nhiều tính năng hơn Basic Timer, có đầy đủ các tính năng của một bộ định thời như đếm thời gian, tạo xung PWM, xử lí tín hiệu vào, so sánh đầu ra, …

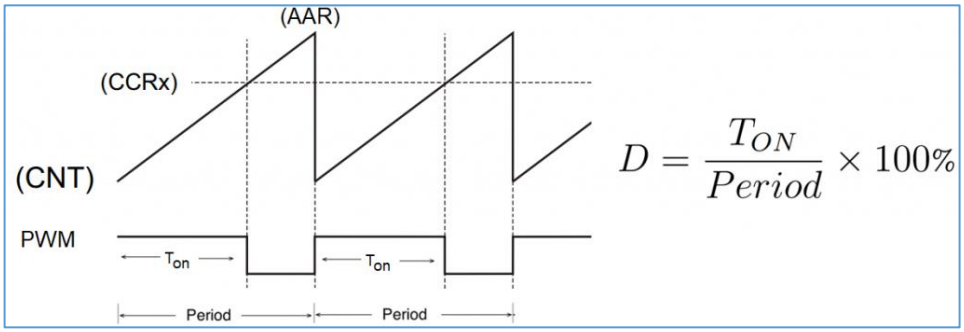
Các kênh Caputer/Campare: Mỗi bộ timer có 4 kênh Capture/Compare độc lập, mỗi kênh này phối hợp Time-base unit có thể tạo ra các tính năng khác nhau:



– ***Input caputer:*** Ở chế độ Input capture, thanh ghi CRR của kênh đầu vào tương ứng sẽ được sử dụng để lưu giá trị của CNT khi phát hiện sự thay đổi mức logic (sườn lên/ sườn xuống) như được cấu hình trước đó. Từ đó có thể biết được khoảng thời gian giữa 2 lần có sườn lên hoặc sườn xuống.

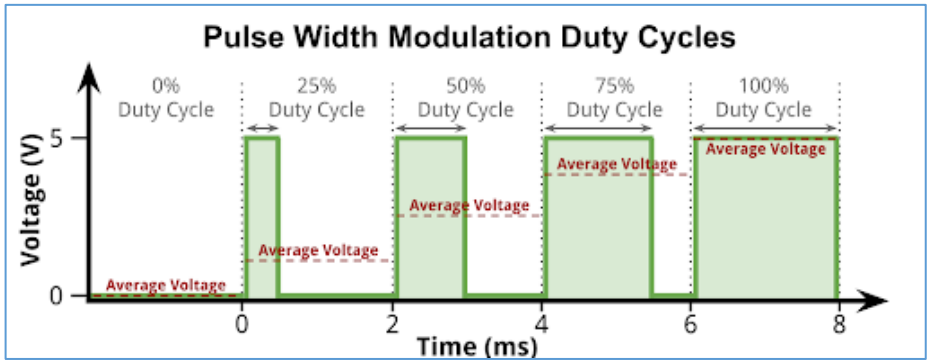
– Output compare: Chế độ này thường được sử dụng để điều khiển đầu ra của 1 I/O PIN khi timer đạt được một chu kỳ thời gian, cũng chính là khi giá trị CNT đếm tới giá trị bằng với giá trị thanh ghi capture/campare (đã được nạp sẵn). Người dùng có thể cài đặt I/O Pin tương ứng với các giá trị logic: mức 1, mức 0 hoặc đảo giá trị logic hiện tại. Đồng thời, cờ ngắt được bật lên và yêu cầu ngắt cũng được tạo ra nếu người dùng cấu hình cho phép ngắt.

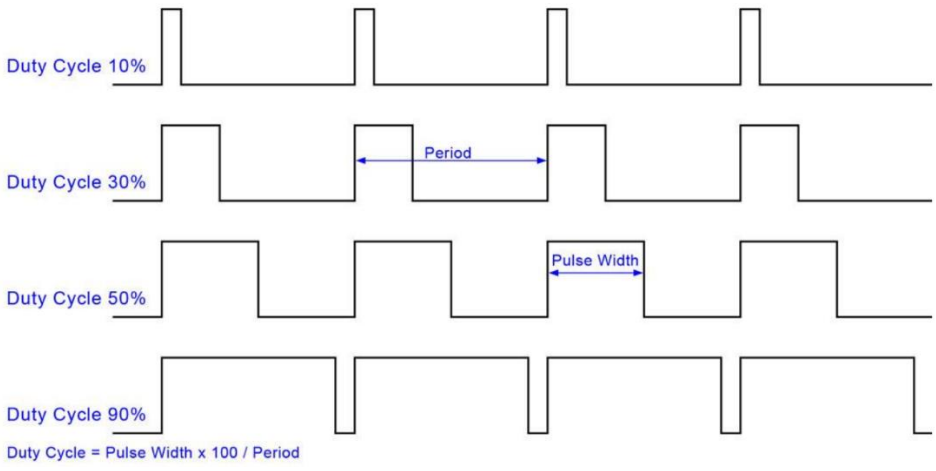
- PWM generation: Tính năng điều chế độ rộng xung cho phép tạo ra xung với tần số được xác định bởi giá trị của thanh ghi ARR, và chu kỳ nhiệm vụ (Duty cycle) được xác định bởi giá trị thanh ghi CCR (**C**apture **C**ompare **R**egister).



II. Lý thuyết PWM

Phương pháp điều xung PWM (Pulse Width Modulation) là phương pháp điều chỉnh điện áp ra tải, hay nói cách khác, là phương pháp điều chế dựa trên sự thay đổi độ rộng của chuỗi xung vuông, dẫn đến sự thay đổi điện áp ra.





- **Duty Cycle** là tỷ lệ phần trăm mức cao.
- **Period** là chu kỳ xung.

- **Pulse Width (T_{ON})** là thời gian ở mức cao trong một chu kì.

Khi làm việc với STM32, tính năng PWM nằm trong khối Timer của vi điều khiển.

Một số thanh ghi quan trọng trong chế độ PWM Generation:

TIMx prescaler (TIMx_PSC): Thanh ghi chứa giá trị chia xung clock được cấp vào từ bus APB.

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

+ TIMx counter register(TIMx_CNT): Thanh ghi lưu giá trị của Counter (CNT).

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

CNT[15:0] (Bit 15:0): Dãy bit lưu giá trị CNT.

+ **TIMx auto-reload register (TIMx_ARR):** Trong chế độ PWM, thanh ghi này lưu giá trị độ phân giải của xung PWM (ví dụ độ phân giải là 100 thì LED có thể có 100 mức độ sáng khác nhau)

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

ARR[15:0] (Bit 15:0): Dãy bit lưu giá trị ARR

+ TIMx capture/compare register x (TIMx_CCRx): Giá trị của thanh ghi này được dùng để làm mốc so sánh với Counter

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR2[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

CCR[15:0] (Bit 15:0): Dãy bit lưu giá trị CCR

STM32F411 hỗ trợ 2 chế độ PWM như sau

+ **Mode1:** Nếu sử dụng chế độ đếm lên thì ngõ ra sẽ ở mức logic 1 khi CNT <CRR và ngược lại, ở mức 0 nếu CNT>CRR. Nếu sử dụng chế độ đếm xuống, đầu ra sẽ ở mức 0 khi CNT > CRR và ngược lại, ở mức 1 khi CNT < CRR.

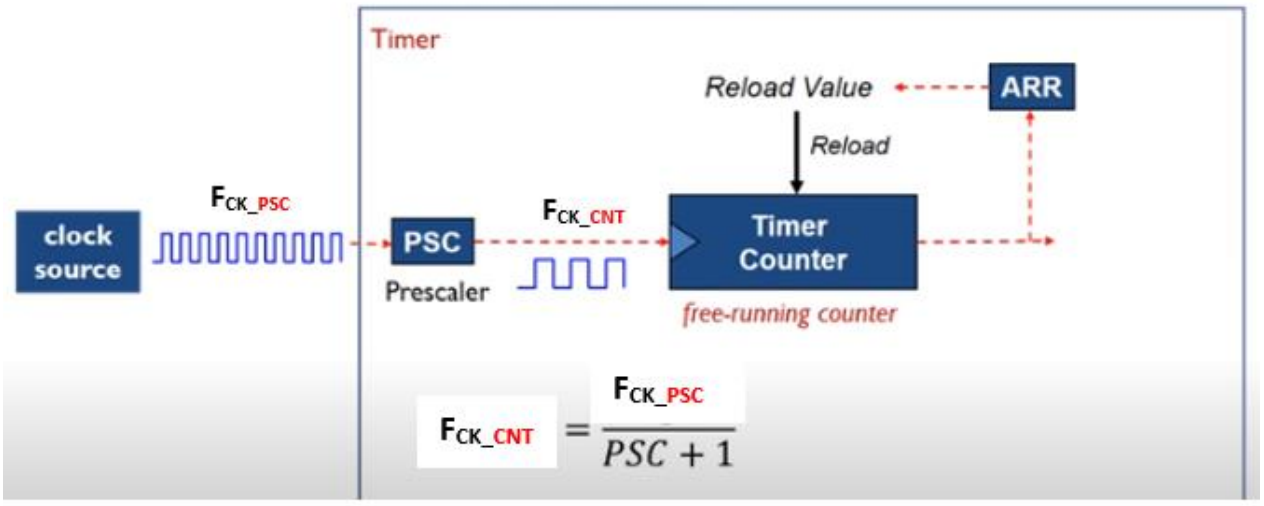
+ **Mode2**: Nếu sử dụng chế độ đếm lên thì ngõ ra sẽ ở mức logic 0 khi CNT <CRR và ngược lại, ở mức 1 nếu CNT>CRR. Nếu sử dụng chế độ đếm xuống, đầu ra sẽ ở mức 1 khi CNT > CRR và ngược lại, ở mức 0 khi CNT<CRR.

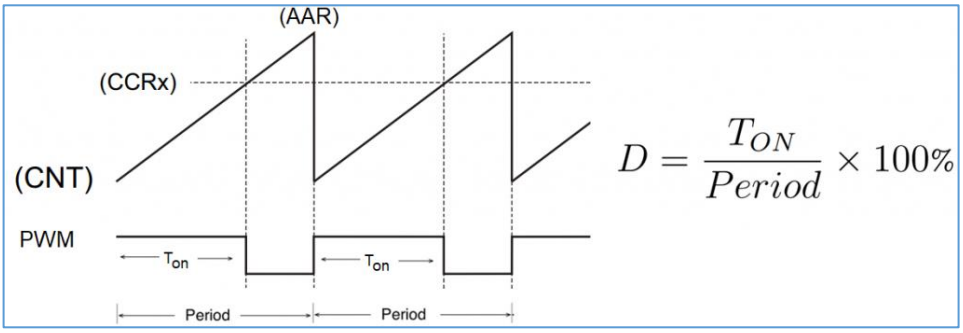
PWM được sử dụng trong rất nhiều ứng dụng như điều chỉnh điện áp đầu ra để thay đổi độ sáng đèn LED, điều khiển động cơ; điều chế bản tin bởi sóng mang, tạo âm thanh…

III. Lập trình PWM, thay đổi độ sáng Led

Ví dụ: Lập trình sáng dẫn 4 LED, gắn trên các Pin PA0, PA1, PA2, PA3, sử dụng 4 kênh PWM của TIM2 (gắn trên các Pin PA0, PA1, PA2, PA3). Thực hiện bằng cách xuất 4 xung PWM có duty cycle tăng dần.

Nhớ lại,





Như vậy, giả sử với bài toán sáng Led, chúng ta có 100 mức sáng ([0..99]), tức là thời gian ở mức cao lần lượt sẽ là 0%, 1%, 2%, .., 98%, 99%

- Thanh ghi **TIMx_ARR** có giá trị là 99
- Tiếp theo ta cần xác định Chu kỳ của xung đếm (Period).

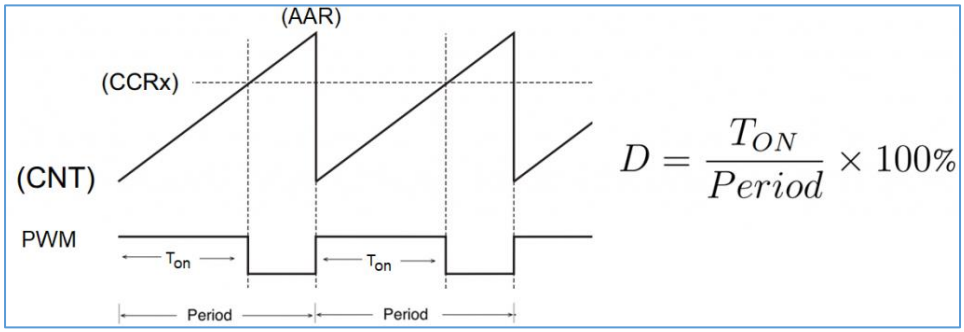
$$Period = (1 + ARR) * T_{CK_{CNT}} = \frac{(1 + ARR) * (1 + PSC)}{F_{CK\_PSC}}$$

F_{CK_PSC} là tần số cung cấp cho timer, cố định
→ cần xác định giá trị **PSC** để đặt vào thanh ghi **TIMx_PSC**

- Thanh ghi **TIMx_CCRx (output)** lưu nội dung thay đổi từ 0 đến 99 tạo ra các mốc so sánh, có thể gọi đây chính là thay đổi **T_{ON}**

Ví dụ: Giả sử

- **ARR** = 99 (chọn);
- **F_{CK_PSC}** (Ftimer, Timer_tick_frequency) = 16MHz =16.000.000 Hz
- **PSC** = 1599 (chọn)
- TIM_Period = (PSC+1)*(ARR+1)/16.000.000 = 16000*100/16.000.000 = 0.01s

Áp dụng công thức 

Ta chỉ cần thay đổi giá trị thanh chi CRR để đạt được T_{ON} khác nhau.

Như đề bài, ta muốn có 100 mức sáng

→ cần có 100 giá trị khác nhau của T_{ON}

➔ Để tạo ra 100 giá trị khác nhau của **CRR**

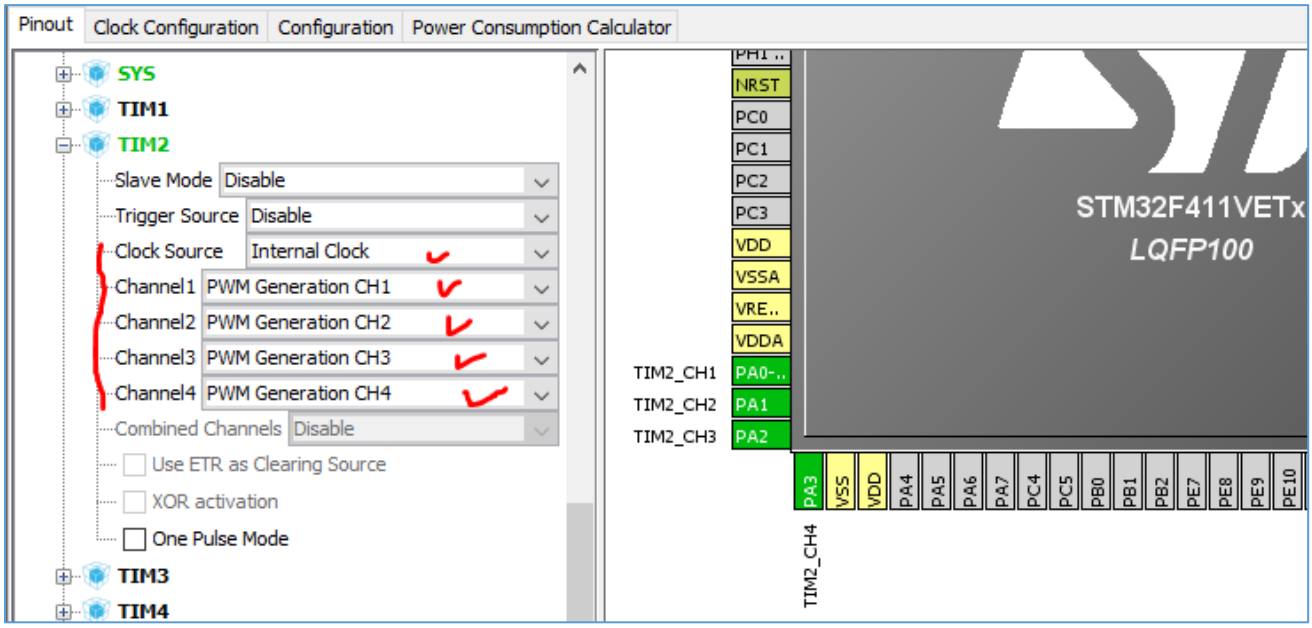
⇨ Lập trình tạo ra CRR khác nhau thông qua thay đổi T_{ON}

Sử dụng hàm để thực hiện băm xung

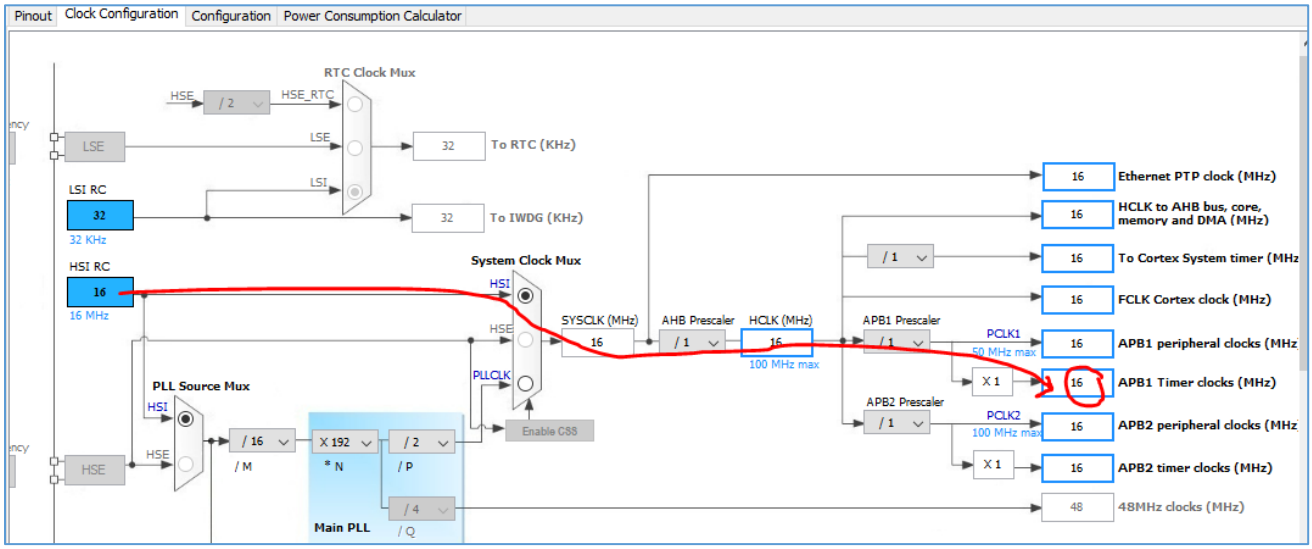
```
__HAL_TIM_SET_COMPARE( timer_handle,  timer_channel,  Ton );
```

THỰC HIỆN

Chọn nguồn xung cho Timer và kích hoạt sử dụng 4 kênh ở chế độ băm xung

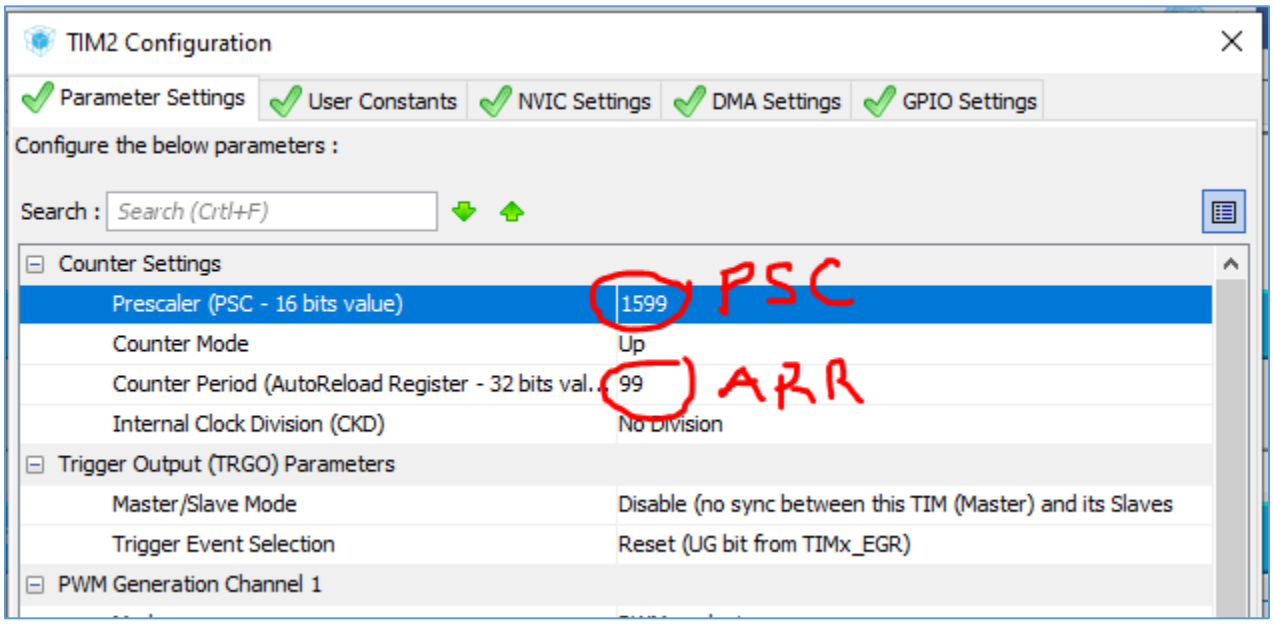


Cấu hình và xác định tần số cấp cho Timer2, ở đây là 16MHz

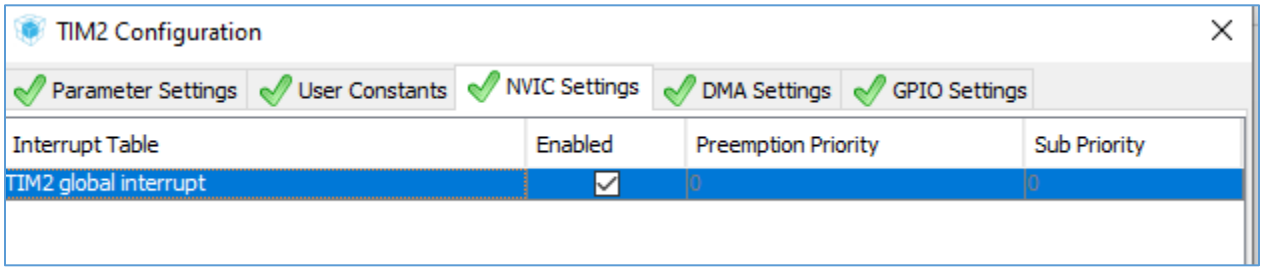


Cấu hình Timer 2

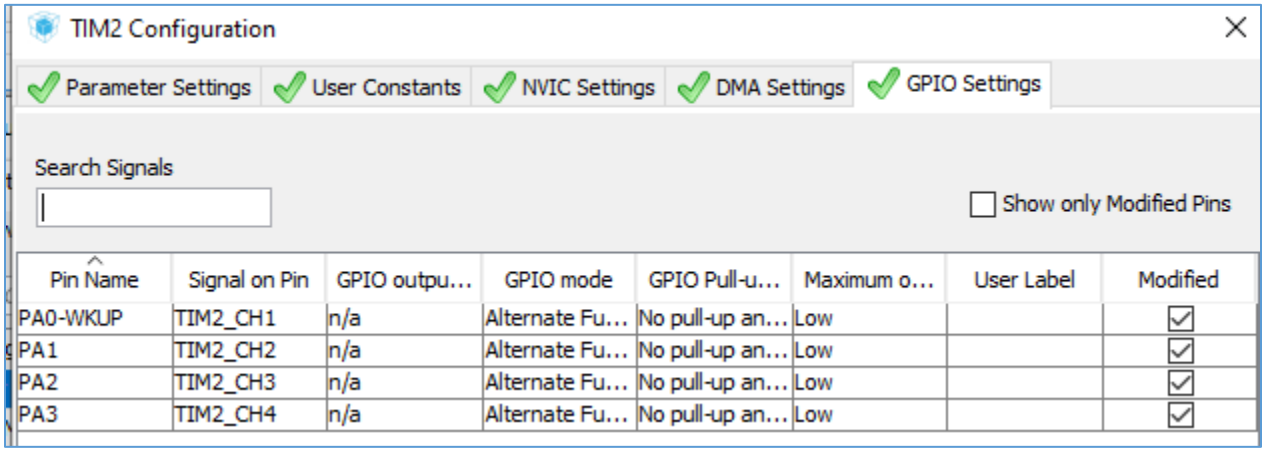
- Các tham số ARR, PSC



- Cho phép ngắt



- Kiểm tra cấu hình GPIO



Sinh code, kiểm tra hàm Main, ta thấy

- Có sẵn một handle của Timer2 là **htim2**
- Ta khai báo thêm 1 biến **pwm** để lập trình thay đổi độ rộng xung

```
/* Private variables ---------------------------------------------------------*/
TIM_HandleTypeDef htim2;

/* USER CODE BEGIN PV */
/* Private variables ---------------------------------------------------------*/
uint8_t pwm=0;
```

- Khởi động Timer trên các kênh

```
  /* USER CODE BEGIN 2 */
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_3);
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_4);
```

- Lập trình thay đổi độ sáng của các Led

```
  while (1)
  {
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
    for(pwm =0;pwm<100;pwm=pwm+10)
    {
      __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_1,pwm);
      __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_2,pwm);
      __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_3,pwm);
      __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_4,pwm);
      HAL_Delay(100);
      }
      HAL_Delay(2000);
    for(pwm =100;pwm>0;pwm=pwm-10)
      {
        __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_1,pwm);
        __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_2,pwm);
        __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_3,pwm);
        __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_4,pwm);
        HAL_Delay(100);
      }
      HAL_Delay(2000);
  }
  /* USER CODE END 3 */
```