

Compte-rendu des TP de microprocesseur : oscilloscope

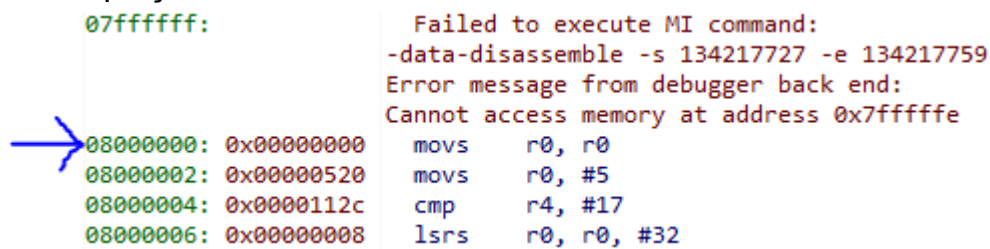
Séances 1 & 2

Reset vector :

Task 4 :

Le vecteur SCB_VTOR se situe à l'adresse **0xE000ED08** et vaut **0x20000000**. Cette adresse pointe le premier élément du tableau des vecteurs. Un vecteur est une adresse attribuée à une interruption indiquant l'adresse du début d'une fonction à exécuter en cas d'interruption.

En se plaçant en assembleur, on trouve :



The screenshot shows a debugger window with assembly code on the left and an error message on the right. A blue arrow points to the first instruction at address 08000000.

Address	Hex Value	Assembly
08000000	0x00000000	movs r0, r0
08000002	0x00000520	movs r0, #5
08000004	0x0000112c	cmp r4, #17
08000006	0x00000008	lsrs r0, r0, #32

Failed to execute MI command:
-data-disassemble -s 134217727 -e 134217759
Error message from debugger back end:
Cannot access memory at address 0x7fffffe

La première instruction exécutée se situe à l'adresse 0x08000000 et est :
Mov R0, R0.

Clocks before and after initialization :

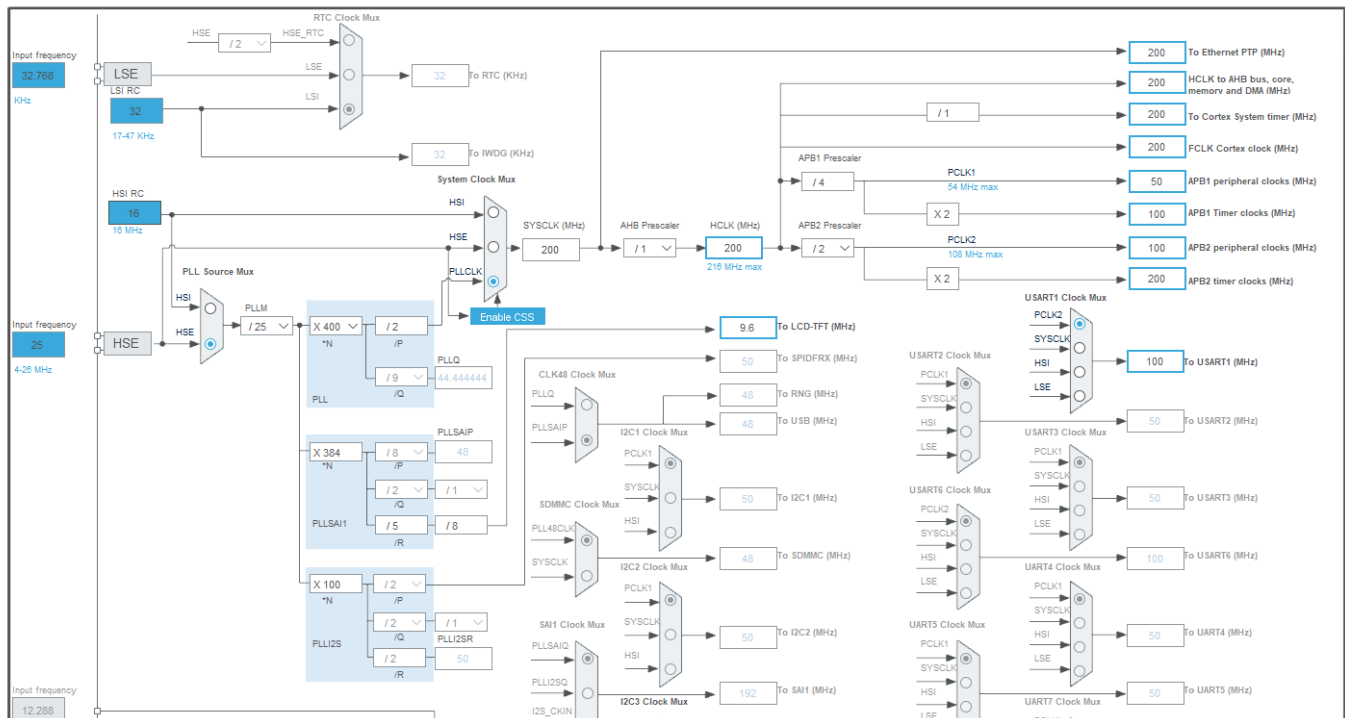
Required work 6 :

Avant l'initialisation, tous les modules de RCC_CFGR sont à 0, ainsi l'horloge choisie est HSI (high speed internal clock) qui vaut **16MHz**. Les bus APB1 et APB2 sont donc à cette fréquence.

> 1010 0101 CFGR	0x40023808	0x0
---------------------	------------	-----

Required work 7 :

Après initialisation, on obtient le schéma suivant :



La PLL est sélectionnée cette fois-ci avec des divisions de fréquences à la suite. On trouve donc que le bus **APB1** est calibré sur une horloge de **50MHz** et **APB2** sur une horloge de **100MHz**.

Register	Address	Value
1010 0101 PLLCFGR	0x40023804	0x29406419
1010 0101 CFGR	0x40023808	0x940a
1010 0101 MCO2	[30:2]	0x0
1010 0101 MCO2PRE	[27:3]	0x0
1010 0101 MCO1PRE	[24:3]	0x0
1010 0101 I2SSRC	[23:1]	0x0
1010 0101 MCO1	[21:2]	0x0
1010 0101 RTCPRE	[16:5]	0x0
1010 0101 PPRE2	[13:3]	0x4
1010 0101 PPRE1	[10:3]	0x5
1010 0101 HPRE	[4:4]	0x0
1010 0101 SWS1	[3:1]	0x1
1010 0101 SWS0	[2:1]	0x0
1010 0101 SW1	[1:1]	0x1
1010 0101 SW0	[0:1]	0x0

On voit ici que la valeur de CFGR a changé permettant de changer l'horloge interne.

The green LED :

Preparation 8 :

RCC_AHB1ENR est le registre permettant d'activer le registre GPIOI. Pour ce faire, il faut mettre le bit 1 à la 9e place (bit numéro 8).

Task 9 :

Ainsi dans RCC_AHB1ENR, on a bien :

 GPIOIEN	[8:1]	0x1
---	-------	-----

Preparation 10 :

Pour configurer les pins du registre GPIOI, il faut modifier les registres GPIOI_OTYPER, GPIOI_MODER, GPIOI_OSPEEDR et GPIOI_PUPDR.

16 pins sont sur le port GPIOI mais seulement 4 sont disponibles sur le dos de la carte.

Le pin I1 doit être configuré comme GP Output Push Pull sans résistances de Pull-up ou Pull-down pour contrôler la LED.

Required work 11 :

Le code de la configuration du pin est généré automatiquement dans la fonction MX_GPIO_Init() :

```
/*Configure GPIO pins : ARDUINO_D7_Pin ARDUINO_D8_Pin PI1 LCD_DISP_Pin */
GPIO_InitStruct.Pin = ARDUINO_D7_Pin|ARDUINO_D8_Pin|GPIO_PIN_1|LCD_DISP_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOI, &GPIO_InitStruct);
```

Pour allumer la LED on peut modifier la valeur du registre ODR à la main

La fonction GPIO_InitStruct(), est présente dans la mémoire des périphériques. Le code a un effet sur la configuration donc elle est dans la mémoire des périphériques.

En visualisant avec le mode debugger le passage avant et après l'appel de HAL_GPIO_Init :

```

14
15@ /**
16 * @brief The application entry point.
17 * @retval int
18 */
19 int main(void)
20 {
21     /* USER CODE BEGIN 1 */
22
23     /* USER CODE END 1 */
24
25     /* MCU Configuration-----*/
26
27     /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
28     HAL_Init();
29
30     /* USER CODE BEGIN Init */
31
32     /* USER CODE END Init */
33
34     /* Configure the system clock */
35     SystemClock_Config();
36
37     /* USER CODE BEGIN SysInit */
38
39     // BEGIN 3: extra Initialization
40     // to set DBG_TIM6_STOP bit in DBG peripheral
41     // in DBGMCU_APB1_FZ register
42     // It stops the TIM6 to count as you enter in debug mode
43     // So it avoids to generate too many inter.

```

Register	Address	Value
> GPIOC		
> GPIOK		
> GPIOJ		
> GPIOI		
> GPIOI	0x40020000	0x0
> GPIOI	0x40020004	0x0
> GPIOI	0x40020008	0x0
> GPIOI	0x4002000c	0x0
> GPIOI	0x40020010	0x0
> GPIOI	0x40020014	0x0
> GPIOI	0x40020018	0x0
> GPIOI	0x4002001c	0x0
> GPIOI	0x40020020	0x0
> GPIOI	0x40020024	0x0
> GPIOI	0x40020028	0x0
> GPIOH		
> GPIOG		
> GPIOF		
> GPIOE		
> GPIOB		

Puis :

```

/*Configure GPIO pin : DCMI_VSYNC_Pin */
GPIO_InitStruct.Pin = DCMI_VSYNC_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
GPIO_InitStruct.Alternate = GPIO_AF13_DCMI;
HAL_GPIO_Init(DCMI_VSYNC_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : OTG_FS_OverCurrent_Pin */
GPIO_InitStruct.Pin = OTG_FS_OverCurrent_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(OTG_FS_OverCurrent_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : SDMMC_CKD_Pin */
GPIO_InitStruct.Pin = SDMMC_CKD_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF12_SDMMC1;
HAL_GPIO_Init(SDMMC_CKD_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pins : TP3 Pin NC2 Pin */

```

> AFRH	0x40022424	0x0
> BRR	0x40022428	0x0
> GPIOI		
> GPIOI	0x40020000	0x100aa54
> GPIOI	0x40020004	0x0
> GPIOI	0x40020008	0x0
> GPIOI	0x4002000c	0x0
> GPIOI	0x40020010	0x3101
> GPIOI	0x40020014	0x1000
> GPIOI	0x40020018	0x0
> GPIOI	0x4002001c	0x0
> GPIOI	0x40020020	0xaaaa0000
> GPIOI	0x40020024	0x0
> GPIOI	0x40020028	0x0
> GPIOH		
> GPIOG		
> GPIOF		
> GPIOE		
> GPIOB		

On voit que l'appel de cette fonction modifie les registres de GPIOI.

Task 12 :

La LED s'allume en mettant 1 sur le registre ODR de GPIOI.

On écrit la fonction LED_DispGreen :

```

#include "main.h"
void LED_DispGreen(int val){
    if (val==0){
        GPIOI->ODR=GPIOI->ODR & ~0x2;
    }else{
        GPIOI->ODR=GPIOI->ODR | 0x2;
    }
}

```

Que l'on appelle dans le while(1) du main.

Elle fonctionne et on voit la LED s'allumer et s'éteindre avec des appels LED_DispGreen(1) et LED_DispGreen(0) dans le debug mode.

Let's take a step back on the first part :

Assignment 13 :

Dans cette partie nous avons initialisé les horloges en choisissant les valeurs des exécutant des fonctions permettant d'utiliser la PLL et de donner des fréquences d'utilisation aux périphériques reliés aux bus APB1 et APB2. Nous avons ensuite configuré ces périphériques en utilisant le registre permettant d'activer le registre GPIOI et ensuite utiliser le sous-registre ODR pour allumer la LED. Nous avons également configuré différents pins de la carte en utilisant les fonctions MX et HAL qui permettent ensuite l'allumage de la LED. À la fin, dans la fonction principale on pouvait, grâce à la fonction LED_DisGreen() que l'on a codée en utilisant les registres précédents, faire allumer la LED ou l'éteindre. Cependant un problème s'est posé car la fonction principale n'était consacrée qu'à l'utilisation de l'allumage et l'éteinte de la LED. Il faut donc utiliser les Timers (partie suivante) pour pouvoir allumer la LED sans monopoliser le corps principal du microcontrôleur.

Light animation with interrupt handler :

Required work 14 :

Les timers 2, 3, 4, 5, 6, 7, 12, 13 et 14 sont à la fréquence de synchronisation du double de celle du bus APB1, c'est-à-dire 100MHz. Les timers 1, 8, 9, 10 et 11 sont à la fréquence de synchronisation du double de celle du bus APB2, c'est-à-dire 200MHz.

Work requested 15 :

Un timer dispose d'une entrée qui est une horloge du bus APB1 ou APB2. Cette fréquence est ensuite divisée par un nombre entier PSC+1. Ensuite un signal sera généré à cette fréquence obtenue qui va compter de 0 à une valeur ARR. Le comptage va être stocké dans le registre CNT. Les timers peuvent être utilisés pour compter à une fréquence fixe, générer un arrêt de comptage pour interrompre un code ou encore générer un signal PWM.

Une combinaison qui fonctionne pour obtenir une interruption quatre fois par seconde dans le code, est de choisir **PSC = 9999** et **ARR = 2499**. Ainsi, la fréquence obtenue est $100\text{MHz}/(10000*2500) = 4\text{Hz}$.

RCC_APB1 est le registre permettant d'activer TIM2 en mettant un "1" dans le bit 0.

Les registres permettant la configuration du Timer 2 sont à l'adresse 0x40000000.

Le registre **PSC** est localisé à l'adresse **0x40000028** et le registre **ARR** est localisé à l'adresse **0x4000002C**.

Job requirement 16 :

Après génération du code, on obtient les valeurs voulues pour PSC et ARR.

▼ 1010 0101 PSC	0x40000028	0x270f
1010 0101 PSC	[0:16]	0x270f
▼ 1010 0101 ARR	0x4000002c	0x9c3
1010 0101 ARR_H	[16:16]	0x0
1010 0101 ARR_L	[0:16]	0x9c3

Et après dépassement de l'appel TIM2_Init() en debugger, le registre TIM2 passe à 1.

La fonction configurant le timer 2 est la suivante :

```

411  */
412 static void MX_TIM2_Init(void)
413 {
414
415     /* USER CODE BEGIN TIM2_Init 0 */
416
417     /* USER CODE END TIM2_Init 0 */
418
419     TIM_ClockConfigTypeDef sClockSourceConfig = {0};
420     TIM_MasterConfigTypeDef sMasterConfig = {0};
421
422     /* USER CODE BEGIN TIM2_Init 1 */
423
424     /* USER CODE END TIM2_Init 1 */
425     htim2.Instance = TIM2;
426     htim2.Init.Prescaler = 9999;
427     htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
428     htim2.Init.Period = 2499;
429     htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
430     htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_DISABLE;
431     if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
432     {
433         Error_Handler();
434     }
435     sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
436     if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
437     {
438         Error_Handler();
439     }
440     sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
441     sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
442     if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
443     {
444         Error_Handler();
445     }
446     /* USER CODE BEGIN TIM2_Init 2 */
447
448     /* USER CODE END TIM2_Init 2 */
449 }
450
451

```

Le problème ici est que le registre CNT n'évolue pas donc il n'y a pas de comptage, pour ce faire il faut l'activer. Pour activer le comptage on ajoute les lignes suivantes :

```

/* USER CODE BEGIN TIM2_Init 2 */
TIM2->DIER=TIM2->DIER|0x1;
TIM2->CR1=TIM2->CR1|0x1;

/* USER CODE END TIM2_Init 2 */





```

Ainsi le timer 2 fonctionne désormais comme voulu.

Required work 17 :

On définit une variable globale dans le fichier main.c et en extern dans le fichier qui contient la fonction TIM2IRQ_Handler(), appelée lors de l'interruption initiée par le timer 2. Après initialisation de la LED via l'appel de

```
ledState=(ledState+1)%2;
```

▼  NVIC		
▶  ICTR	0xe000e004	0x3
▼  NVIC_ISER0	0xe000e100	0x10000000
 SETENA	[0:32]	0x10000000

1010 0101	NVIC_ICER0	0xe000e180	0x10000000
--------------	------------	------------	------------

MSB 0 1 1 1 1 0 0 0 0 LSB

Register: IPR14
Address: 0xe000e438

Lorsque l'on lance le programme on voit que la LED s'allume deux fois par seconde donc la variable globale évolue quatre fois par seconde ce qui est cohérent avec les valeurs de PSC et ARR implémentées.

On réalise l’affichage du temps depuis la réinitialisation du board grâce au code suivant :


```

149  /* Infinite loop */
150  /* USER CODE BEGIN WHILE */
151  while (1)
152  {
153      g_n++;
154      BSP_TS_GetState(&g_ts);
155      sprintf(bufStr, "X:%3d", g_ts.touchX[0]);
156      BSP_LCD_DisplayStringAt(20,20,(uint8_t *) bufStr,LEFT_MODE);
157      sprintf(bufStr, "Y:%3d", g_ts.touchY[0]);
158      BSP_LCD_DisplayStringAt(20,50,(uint8_t *) bufStr,LEFT_MODE);
159      int quotient = globvar2/10;
160      int reste = globvar2%10;
161      /** LED_DispGreen(0);
162      LED_DispGreen(1);
163      LED_DispGreen(0); */
164      sprintf(bufStr, "time:%d.%d s", quotient, reste);
165      BSP_LCD_DisplayStringAt(20,80,(uint8_t *) bufStr,LEFT_MODE);

```

On initialise une variable globale dans le main (extern dans le fichier de la fonction TIM3IRQ_Handler). Cette variable ne peut pas être un float car la fonction sprintf n'affiche pas de float. C'est donc un entier. À partir de cette variable, on crée une variable quotient qui est cette variable globale divisée par 10. Cette variable quotient désigne donc les secondes et on crée une seconde variable reste qui vaut le reste de la division euclidienne de la variable globale par 10. Cette variable reste désigne en réalité les millisecondes. Grâce à la fonction sprintf, on met ces variables sous forme d'un String que l'on affiche avec BSP_LCD_DisplayStringAt sur l'écran de la STM32.

Required work 19 :

Les registres NVIC qui permettent de gérer les autorisations des interruptions sont les NVIC_ISERx. On doit lire un **1 aux bits 28 et 29** (pour activer les timers 2 et 3) dans le registre **NVIC_ISER0** (adresse 0xE000E100). Le **numéro d'identification de l'interruption du timer 2 est 28**, ainsi le **décalage associé est $(28+16)*4 = 176$** . On ajoute ce décalage à l'adresse théorique de **départ de la table de vecteur qui est 0x00000040**, on trouve l'adresse de départ de la fonction associée à l'interruption créée par le **timer 2** à **0x000000B0**. Par le même raisonnement on trouve un **numéro d'identification d'interruption du timer 3 de 29** donc un **décalage de 180** et une adresse de départ de la fonction à **0x000000B4**. Pour le timer 4, le numéro d'identification est **28**, le décalage est **184** et l'adresse de départ est **0x0000_00B8**.

Les priorités d'interruption des timer 2 et 4 sont stockées dans les registres **NVIC_IPR7** car $28/4 = 7$ ainsi le code de la priorité du timer 2 occupe les huit premiers bits de NVIC_IPR7 tandis que celui du timer 4 occupe les bits 16 au 23 de NVIC_IPR7 (adresse 0xE000E41C).

Required work 20 :

On regarde le registre NVIC_I SER0 et on observe :

MSB [0] [0] [1] [1] [0] **LSB**

Register: ISER0
Address: 0xe00e100

On voit des 1 à la 28e place (autorisation de l'interruption de TIM2) à la 29e place (autorisation de l'interruption de TIM3) mais pas à la 30e place ce qui signifie qu'il n'y a pas d'autorisation de l'interruption de TIM4.

Lors du lancement du programme le registre NVIC IPR18 évolue :

MSB 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 LSB

Register: IPR18
Address: 0xe000e448

Ce registre se situe à l'adresse 0xE000E448. Il y a quatre bits en plus mis à 1 que dans le registre NVIC_IPR14 car le niveau de priorité est le même donc il faut donner des priorités subsidiaires différentes pour un possible appel des interruptions en même temps.

On sait depuis la première séance que SCB_VTOR contient l'adresse 0x20000000, ce qui signifie que la table des vecteurs commence à cette adresse. Comme on a déduit les adresses des vecteurs des timers 2, 3 et 4 lors du travail 19, on déduit que :

l'adresse du vecteur de TIM2 est 0x200000B0, celle du vecteur de TIM3 est 0x200000B4, et celle du vecteur de TIM4 est 0x200000B8.

Lorsque l'on observe le code des fonctions appelées lors des interruptions initiées par TIM2 et TIM3, on a :

```
TIM2_IRQHandler:
08001b84: push    {r7, lr}
08001b86: add     r7, sp, #0
216      HAL_TIM_IRQHandler(&htim2);
08001b88: ldr     r0, [pc, #28] ; (0x8001ba8 <TIM2_IRQHandler+36>)
08001b8a: bl      0x80072fc <HAL_TIM_IRQHandler>
218      globvar = (globvar+1)%2;
08001b8e: ldr     r3, [pc, #28] ; (0x8001bac <TIM2_IRQHandler+40>)
08001b90: ldr     r3, [r3, #0]
08001b92: adds    r3, #1
08001b94: cmp     r3, #0
08001b96: and.w   r3, r3, #1
08001b9a: it      lt
08001b9c: neglt   r3, r3
08001b9e: ldr     r2, [pc, #12] ; (0x8001bac <TIM2_IRQHandler+40>)
08001ba0: str     r3, [r2, #0]
220      }
08001ba2: nop
08001ba4: pop     {r7, pc}
08001ba6: nop
08001ba8: lsls    r0, r2, #21
08001baa: movs    r0, #0
08001bac: lsls    r4, r1, #14
08001bae: movs    r0, #0
226      {
TIM3_IRQHandler:
08001bb0: push    {r7, lr}
08001bb2: add     r7, sp, #0
228      globvar2 += 1;
08001bb4: ldr     r3, [pc, #16] ; (0x8001bc8 <TIM3_IRQHandler+24>)
08001bb6: ldr     r3, [r3, #0]
08001bb8: adds    r3, #1
08001bba: ldr     r2, [pc, #12] ; (0x8001bc8 <TIM3_IRQHandler+24>)
08001bbc: str     r3, [r2, #0]
230      HAL_TIM_IRQHandler(&htim3);
08001bbe: ldr     r0, [pc, #12] ; (0x8001bcc <TIM3_IRQHandler+28>)
08001bc0: bl      0x80072fc <HAL_TIM_IRQHandler>
234      }
08001bc4: nop
08001bc6: pop     {r7, pc}
08001bc8: lsls    r4, r0, #18
08001bca: movs    r0, #0
08001bcc: lsls    r0, r2, #14
08001bce: movs    r0, #0
```

On voit que le code TIM2_IRQHandler commence à l'adresse 0x08001B84 et celui de TIM3_IRQHandler commence à l'adresse 0x08001BB0. Cela est normal car lorsque l'on regarde l'adresse 0x200000B0 (celle du vecteur de TIM2), on observe l'adresse 0x08001B84 (celle du début du code de la fonction appelée par l'interruption initiée par le timer 2, à laquelle on a mis le premier bit à 0). De même pour le timer 3, à l'adresse 0x200000B4, on observe l'adresse 0x08001BB0.

Let's take a step back from this second part :

Assignment 21 :

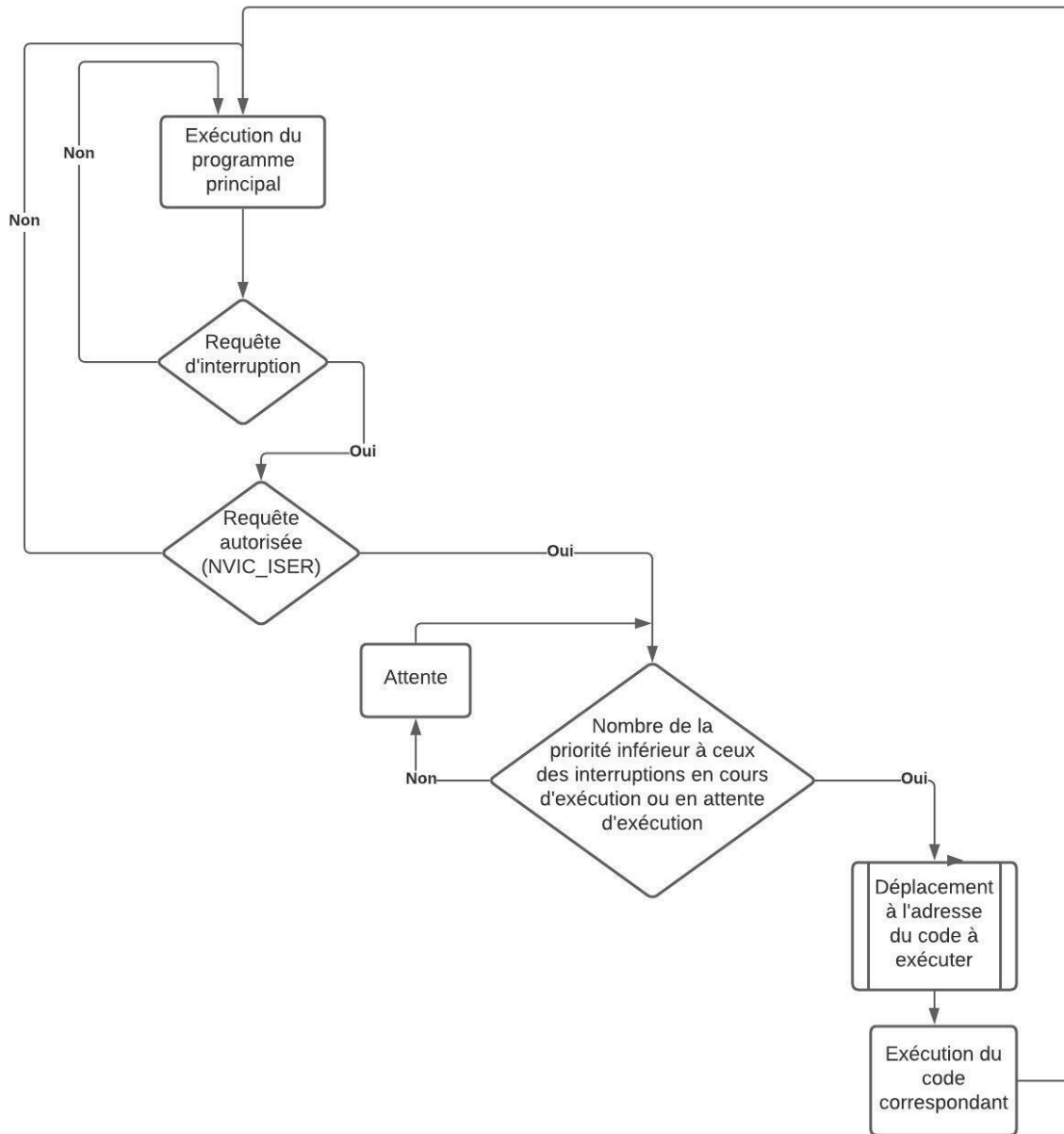


Diagramme résumant le principe d'une interruption