



Tutoriels STM32F0

Interruptions

Interruptions externes

Arouna DARGA – enseignant chercheur Sorbonne Université
arouna.darga@sorbonne-universite.fr

Table des matières

1.	Introduction.....	3
2.	Scrutation d'évènement.....	3
2.1	Mise place du projet pour le tutoriel.....	3
2.2	Pourquoi ça ne fonctionne pas ?	4
2.3	Exemple de scrutation bloquante	5
2.4	Exemple de scrutation Non bloquante.....	7
2.5	Discussion	9
3.	Les interruptions	10
3.1	Configuration du périphérique pour qu'il demande ou produise une interruption	11
3.2	Configuration du contrôleur d'interruption NVIC	13
3.3	Implémentation du gestionnaire d'interruption ou interrupt handler (ISR)	16
3.4	Gestion du délai d'exécution des ISR.....	19
4.	Résumé.....	24

1. Introduction

Ce chapitre traite de la gestion des événements. Un événement est un signal logique que vous attendez d'une source. La source peut être externe à la MCU ou interne (périphérique). Il peut être lié à une action de l'utilisateur sur un bouton, il peut être un UART signalant un octet entrant, il peut être un ADC indiquant la fin d'une conversion, il peut être un événement de mise à jour de Timer, ou bien d'autres encore.

Il existe plusieurs façons de traiter ces signaux. La "scrutation" d'un événement consiste à écrire une instruction conditionnelle (si, tant que, jusqu'à...) pour vérifier de manière répétée si l'évènement s'est produit ou non. Comme l'interrogation maintient l'unité centrale occupée en attendant l'évènement, vous devez compter sur d'autres mécanismes pour traiter les signaux externes ou internes : interruptions, signaux de déclenchement (trigger) ou demandes DMA (Direct Memory Access).

2. Scrutation d'évènement

En guise d'introduction, rappelons en quoi consistent les scrutations d'évènements à travers une application pédagogique. Dans ce scénario, la boucle principale est supposée effectuer une tâche importante de manière répétée tout en étant réactif à l'action de l'utilisateur sur le bouton bleu de la carte Nucleo.

Nous allons utiliser la console de débogage pour illustrer le comportement de notre application :

- La tâche importante est illustrée par l'envoi au PC (ordinateur) d'un caractère "." suivi d'un délai qui représente le temps nécessaire à l'exécution de cette tâche
- L'action de l'utilisateur sur le bouton (appuis sur le bouton) est illustrée l'envoi au PC (ordinateur) ou impression d'un caractère "#".
- Le relâchement du bouton est illustré par l'envoi au PC (ordinateur) ou impression d'un caractère "|".

2.1 Mise place du projet pour le tutoriel

Pour ce tutoriel, nous allons apporter des modifications majeures au code développé dans les tutoriels précédents. Il faut donc un projet dédié à ce tutoriel.

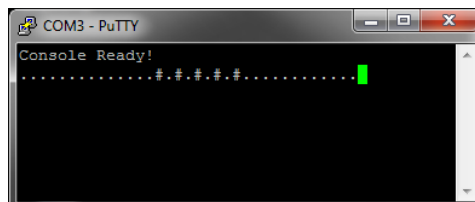
Cloner le projet précédent « **my_project** » et nommer le projet cloné par « **Tutoriels_Interruptions** ». Pour la procédure de clonage suivre la procédure décrite dans le tutoriel « **Tutoriels STM32F0-Environnement de développement v05022021.pdf** » (à partir de la page 30), disponible sur moodle.

2.2 Pourquoi ça ne fonctionne pas ?

```
#include <math.h>
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"
static void SystemClock_Config(void);
int main(void)
{
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();
    //Initialisation de pin PC13 pour bouton Bleu
    BSP_PB_Init();
    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");
    //Initialisation de Timer TIM6 pour delai
    BSP_DELAY_TIM_init();

    // boucle principale des applications
    while(1)
    {
        // Verifier que le bouton est appuyer
        if (BSP_PB_GetState() == 1)
        {
            // si bouton appuyer envoyer "#" au PC par USART
            mon_printf("#");
        }
        // Exemple de tache importante a faire
        // envoyer "." au PC par USART et attendre 200 ms
        mon_printf(".");
        BSP_DELAY_TIM_ms(200);
    }
}
```

Avec le code ci-dessus, la tâche associée à l'action du bouton sera exécutée de manière répétée tant que l'utilisateur maintiendra le bouton enfoncé :



Il faut configurer votre logiciel d'émulation de terminal (Ici, PuTTY). Revoir le tutoriel sur l'introduction de la liaison en mode transmission si besoin.

Nous souhaitons que la tâche associée au bouton ne soit exécutée qu'une seule fois, à chaque fois que le bouton est pressé. Pour cela, nous devons détecter un changement d'état du bouton. Nous utiliserons deux

méthodes de scrutation pour détecter le changement d'état du bouton : scrutation bloquante et scrutation non bloquante.


2.3 Exemple de scrutation bloquante

```
#include <math.h>
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"
static void SystemClock_Config(void);
int main(void)
{
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();
    //Initialisation de pin PC13 pour bouton Bleu
    BSP_PB_Init();
    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");
    //Initialisation de Timer TIM6 pour delai
    BSP_DELAY_TIM_init();

    // boucle principale des applications
    while(1)
    {
        // attendre ici tant que le bouton est appuyer ou non relacher
        while (BSP_PB_GetState() != 0)
        {
            // on attend que le bouton soit relacher
        }
        // arriver ici : signifie que le bouton a ete relacher
        mon_printf("|");

        // attendre ici tant que le bouton est non appuyer ou relacher
        while (BSP_PB_GetState() != 1)
        {
            // on attend que le bouton soit appuyer
        }
        // arriver ici : signifie que le bouton a ete appuyer
        mon_printf("#");
        // Exemple de tache importante a faire
        // envoyer "." au PC par USART et attendre 200 ms
        mon_printf(".");
        BSP_DELAY_TIM_ms(200);
    }
}
```

Tester le code ci-dessus. Utiliser le terminal PuTTY pour analyser son fonctionnement avec appuis et relâchement du bouton bleu.

	<p>Sauvegarder votre code main et les captures d'écrans montrant vos essais</p> <p>Expliquez pourquoi la tâche importante « <i>La tâche importante est illustrée par l'envoi au PC (ordinateur) d'un caractère "." suivi d'un délai qui représente le temps nécessaire à l'exécution de cette tâche</i> » ne s'exécute qu'une seule fois.</p>
---	---

2.4 Exemple de scrutation Non bloquante

La méthode précédente, « scrutation bloquante » ne permet pas de répondre à notre cahier des charges. En outre, cette méthode est très dangereuse : si pour une raison quelconque l'événement ne se produit jamais, l'application y est suspendue. D'après l'expérience sur les interfaces homme-machine, cela est interpréter par l'utilisateur comme un plantage qui va chercher à réaliser une réinitialisation. Le moins que vous puissiez faire est de mettre en place une condition de temporisation (TIMEOUT) afin qu'une sortie soit possible si l'événement ne se produit pas dans un délai acceptable. Nous verrons un exemple de mise en œuvre de TIMEOUT dans le tutoriel sur le bus série I2C.

Pour ne peut utiliser la scrutation bloquante, nous pouvons utiliser la structure conditionnelle " **if** " au lieu de la boucle " **while** ". Dans cas, vous avez besoin d'un moyen de mémoriser l'état actuel du bouton afin d'éviter de répéter la même action encore et encore alors que l'état du bouton ne change pas.

Il existe plusieurs possibilités d'implémentation. Dans ce tutoriel nous utilisons une machine à états pour des raisons pédagogiques. Nous utilisons une variable d'état, **button_state**, et la structure " **switch case** ".

La partie **while(1)** du code **main.c**, fonction **main()** est donnée ci-dessous.

```
// initialisation de la variable etat
button_state = 0;
// boucle principale des applications
while(1)
{
    // machine a etat
    switch (button_state)
    {
        case 0: // le bouton est actuellement Non appuyer
        {
            // le bouton est appuyer
            if (BSP_PB_GetState() == 1)
            {
                mon_printf("#");
                button_state = 1;
            }
            break;
        }

        case 1: // le bouton est actuellement appuyer
        {
            // le bouton est relacher
            if (BSP_PB_GetState() == 0)
            {
                mon_printf("|");
                button_state = 0;
            }
            break;
        }
    }

    // Exemple de tache importante a faire
    // envoyer "." au PC par USART et attendre 200 ms
    mon_printf(".");
    BSP_DELAY_TIM_ms(200);
}
```



Implémenter cette nouvelle méthode et réaliser les essais avec différentes valeurs de **BSP_DELAY_TIM_ms(x)**; : **x=500 et 1000** ; que Constatez-vous ? Expliquez
Sauvegarder votre code main et les captures d'écrans montrant vos essais

2.5 Discussion

Les méthodes de scrutation sont parfaitement valables, mais vous devez être conscient de trois inconvénients principaux :

- Vous passez du temps CPU à vérifier si l'événement s'est produit (c'est-à-dire à lire un registre périphérique), que l'événement se soit réellement produit ou non.
- Vous pourriez réagir à l'apparition de l'événement longtemps après qu'il se soit réellement produit (perte d'informations) parce que le CPU n'analyse pas toujours l'entrée et peut être occupée par d'autres tâches.
- Vous risquez de manquer des événements. Vous pouvez facilement illustrer ce point en appuyant rapidement sur le bouton avec notre code actuel en cours d'exécution. Vous verrez que la console ne signale pas les tapes rapides, sauf si vous êtes chanceux...

*Le mécanisme matériel qui capte un événement dès qu'il se produit, tout en laissant l'unité centrale libre d'effectuer d'autres tâches, est appelé "**interruption**".*

3. Les interruptions

Une interruption est un **mécanisme matériel** qui **suspend automatiquement l'exécution** de la fonction en cours afin d'exécuter **une tâche spécifique (une autre fonction)** dès qu'un événement se produit. Lorsque cette tâche dédiée est réalisée, le programme reprend exactement là où il a été suspendu.

La plupart du temps, ce que nous appelons "**événement**" provient d'un **périphérique matériel**. Il peut s'agir d'un changement sur la broche d'entrée, d'un octet entrant de l'USART, d'une fin de conversion de ADC, d'une mise à jour ou débordement du compteur d'un Timer, et bien d'autres choses encore... Cet événement est également appelé **source d'interruption**.

En outre, le MCU est dotée d'un **contrôleur d'interruption** avancé permettant de gérer plusieurs sources d'interruption. Par exemple, une interruption "B" peut se produire pendant que le MCU exécute la tâche dédiée à l'interruption "A", ce contrôleur doit mettre en œuvre un mécanisme de priorité et traiter les appels imbriqués. Dans la gamme de produits STM32, ce contrôleur est **appelé NVIC (Nested Vector Interrupt Controller)**. Vous pouvez le voir comme une sorte de commutateur d'interruption. Notez que le NVIC fait partie du noyau ARM Cortex-M.

Pour travailler avec une interruption, il faut procéder à la configuration suivante :

1. Configurer le périphérique source afin de permettre la signalisation d'interruption au contrôleur NVIC lorsqu'un événement spécifique se produit
2. Configurer le contrôleur NVIC pour qu'il accepte ce signal comme une source d'interruption valable avec une priorité donnée
3. Ecrire la fonction qui doit être exécutée en cas d'interruption

Si les étapes 1 ou 2 sont manquantes, aucun signal d'interruption n'atteint l'unité centrale. Si l'étape 3 est manquante, l'unité centrale est suspendue dans une boucle infinie qui est mise en œuvre en tant que gestionnaire d'interruption par défaut.

3.1 Configuration du périphérique pour qu'il demande ou produise une interruption

Dans cet exemple, nous voudrions qu'une action sur le bouton-poussoir (broche PC13) produise un signal d'interruption.

Pour y parvenir, nous allons rajouter une nouvelle fonction, **BSP_PB_IT_Init()**, à notre bibliothèque de fonction, **bsp.c**. Il ne faut pas la confondre avec la fonction **BSP_PB_Init()**. Le code de la fonction **BSP_PB_IT_Init()**, est donné ci-dessous. Rajouter cette fonction au code source **bsp.c** et **rajouter la déclaration de son prototype** dans le fichier en-tête **bsp.h**.

```
/*
 * BSP_PB_IT_Init()
 * Initialise le pin PC13 (Bouton bleu) en input
 * Pas de resistances de tirage Pull-up/Pull-down
 * Activation de la source interruption EXTI13
 * Mode de declenchement : Front descendant
 */
void BSP_PB_IT_Init()
{
    // Activation de horloge de GPIOC
    RCC->AHBENR |= RCC_AHBENR_GPIOCEN;

    // Configuration de pin PC13 en INPUT
    GPIOC->MODER &= ~GPIO_MODER_MODER13_Msk;
    GPIOC->MODER |= (0x00 <<GPIO_MODER_MODER13_Pos);

    // Deactivation des resistances de tirage Pull-up/Pull-down pour pin PC13
    GPIOC->PUPDR &= ~GPIO_PUPDR_PUPDR13_Msk;

    // Activation hologe du peripherique du System configuration controller : SYSCFG
    // SYSCFG est responsable de la configuration des interruptions Externes
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;

    // Choisir le Port C comme source interruption pour la ligne interruption externe
    // EXTI line 13
    // voir page 178 & 179
    SYSCFG->EXTICR[3] &= ~SYSCFG_EXTICR4_EXTI13_Msk;
    SYSCFG->EXTICR[3] |= SYSCFG_EXTICR4_EXTI13_PC;

    // Activation de la ligne de source interruption externe EXTI line 13
    EXTI->IMR |= EXTI_IMR_IM13;

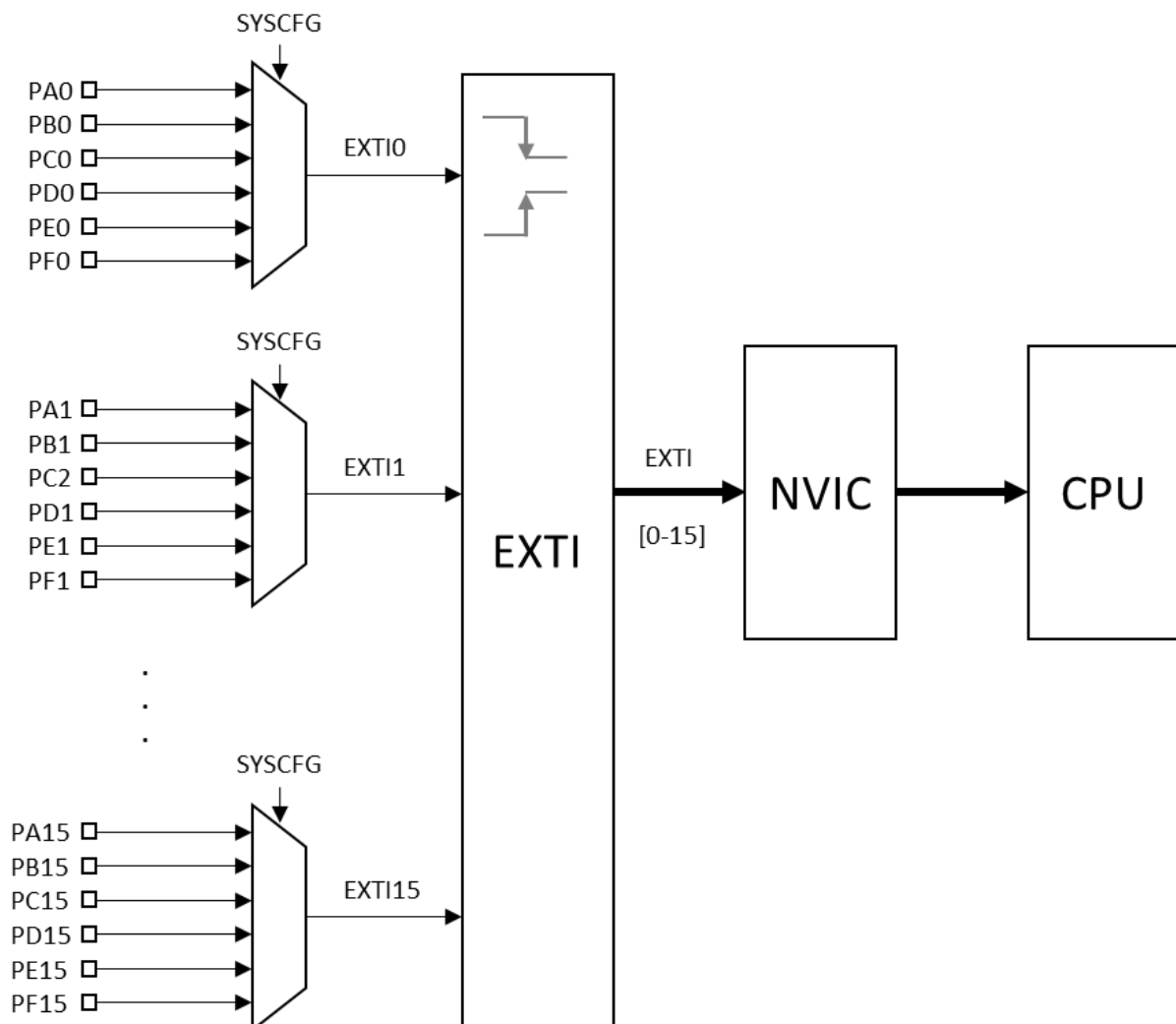
    // Choisir le mode de declenchement
    // Deactiver front Montant : Rising
    // Activer front Descendant Falling trigger
    EXTI->RTSR &= ~EXTI_RTISR_RT13;
    EXTI->FTSR |= EXTI_FTSR_FT13;
}
```

Quelques explications du code :

Nous voulons générer une interruption lors de l'action sur le bouton utilisateur. Lorsque le bouton est enfoncé, nous avons un front descendant sur la broche PC13.

- Le périphérique qui gère les interruptions externes est EXTIF. EXTI a 16 entrées (lignes) [0-15], ce qui est inférieur au nombre d'E/S disponibles sur le MCU. Par conséquent, nous devons choisir quelle broche est utilisée pour quelle ligne EXTI.
- En principe, la ligne **EXTI n** peut être connectée à n'importe quel numéro n de **GPIOx**. Par exemple, la **ligne EXTI 13** peut être connectée à l'une des broches **PA13, PB13, PC13, PD13, PE13, PF13**. Cela se fait à l'aide des **registres SYSCFG**.
- Une fois que la broche choisie est connectée au **contrôleur EXTI**, il faut activer la **ligne EXTI** correspondante (dans le périphérique EXTI), puis choisir si c'est le front **montant** et/ou **descendant** qui est retenu pour générer un signal d'interruption.

Lorsque tout ce qui précède est correctement réglé, le signal d'interruption est généré et envoyé au contrôleur NVIC. À ce stade, le signal n'est pas encore autorisé à interrompre l'unité centrale.



3.2 Configuration du contrôleur d'interruption NVIC

Le contrôleur NVIC est le circuit qui reçoit tous les signaux d'interruption et qui décide si l'unité centrale doit être interrompue ou non. Il gère les priorités des interruptions.

Une bonne pratique consiste à configurer le **NVIC** indépendamment de la configuration périphérique pour au moins deux raisons :

- Le regroupement des paramètres **NVIC** en une seule fonction pour toutes vos sources d'interruption vous donne un meilleur aperçu des interruptions autorisées et de la hiérarchie des priorités.
- Dès que le **NVIC** est activé, des interruptions peuvent se produire. Vous devez donc terminer toutes les initialisations des périphériques avant d'activer le **NVIC**. Par conséquent, le **NVIC** ne doit pas être défini dans l'initialisation périphérique mais plutôt appelé depuis la fonction **main()** lorsque tout est prêt pour gérer correctement les interruptions.

Ajouter la fonction ci-dessous au fichier de codes source **bsp.c**.

```
/*
 * BSP_NVIC_Init()
 * Configuration du controleur NVIC pour autoriser et accepter les sources interruptions
 * activer
 */

void BSP_NVIC_Init()
{
    // Mettre en priorite maximuim les lignes interruptions externes EXTI 4 a 15
    NVIC_SetPriority(EXTI4_15_IRQn, 0);

    // Autoriser les lignes interruptions externes EXTI 4 a 15
    // le bouton Bleu est sur PC13 (ligne 13)
    NVIC_EnableIRQ(EXTI4_15_IRQn);
}
```

Rajouter également la déclaration de son prototype dans le fichier des en-têtes **bsp.h**

```
/*
 * Fonctions Initialisation du NVIC
 *
 */

void BSP_NVIC_Init(void);
```

Nous utiliserons le code ci-dessous pour tester les configurations précédentes :



```

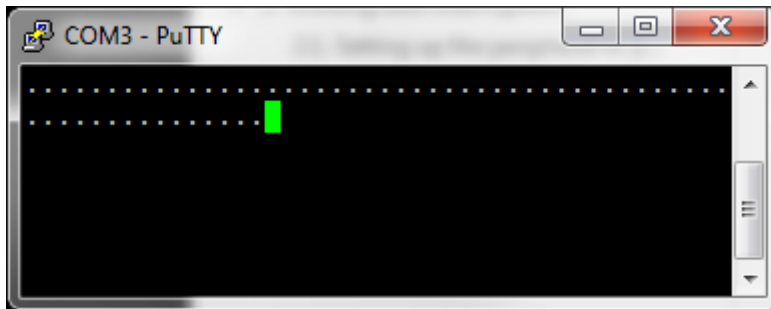
#include <math.h>
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"


static void SystemClock_Config(void);

int main(void)
{
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();
    //Initialisation de pin PC13 pour bouton Bleu avec interruption
    BSP_PB_IT_Init();
    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");
    //Initialisation de Timer TIM6 pour delai
    BSP_DELAY_TIM_init();
    // initialisation du controleur interruption NVIC
    BSP_NVIC_Init();
    // boucle principale des applications
    while(1)
    {
        // Exemple de tache importante a faire
        // envoyer "." au PC par USART et attendre 200 ms
        mon_printf(".");
        BSP_DELAY_TIM_ms(200);
    }
}

```

- Compiler le projet
- Lancer le mode debugger  et lancer l'exécution du programme  Vous devriez voir des points s'afficher sur la console PuTTY.



- Appuyez ensuite sur le bouton bleu. L'affichage des points s'arrête. L'exécution du programme est maintenant interrompue. Le MCU ne répond plus à rien...
- Suspendez l'exécution du programme  pour voir ce que fait réellement le MCU. Vous découvrirez que l'exécution a été piégée dans une boucle infinie ne faisant rien appelée "**Default Handler**".

```

97 .size Reset_Handler, .-Reset_Handler
98
99 /**
100  * @brief This is the code that gets called when the processor receives an
101  *         unexpected interrupt. This simply enters an infinite loop, preserving
102  *         the system state for examination by a debugger.
103  *
104  * @param None
105  * @retval : None
106  */
107 .section .text.Default_Handler,"ax",%progbits
108 Default_Handler:
109 Infinite_Loop:
110 b Infinite_Loop
111 .size Default_Handler, .-Default_Handler
112 /*****
113  *
114  * The minimal vector table for a Cortex M0. Note that the proper constructs
115  * must be placed on this to ensure that it ends up at physical address
116  * 0x0000.0000.
117  */
118 *****/
119 .section .isr_vector,"a",%progbits
120 .type g_pfnVectors, %object
121 .size g_pfnVectors, .-g_pfnVectors

```

C'est ce qui apparaît lorsque vous avez configuré une interruption mais que vous n'avez pas écrit le gestionnaire d'interruption, **interrupt handler**, correspondant. L'interruption a bien fonctionné, mais la tâche spécifique à effectuer n'a pas été trouvée.

3.3 Implémentation du gestionnaire d'interruption ou interrupt handler (ISR)

Comme nous l'avons dit précédemment, le signal d'interruption est délivré par le matériel. Lorsque le signal d'interruption est transmis par le **contrôleur NVIC**, l'exécution du programme passe automatiquement à une adresse mémoire de code spécifique.

Cette adresse mémoire de code particulière est située dans une zone réservée à la mémoire appelée "**table de vecteurs d'interruption**". Ouvrez le fichier assembleur *startup_stm32f0xx.S* dans l'éditeur principal et faites défiler les lignes suivantes :

```
131 g_pfnVectors:
132 .word _eram
133 .word Reset_Handler
134 .word NMI_Handler
135 .word HardFault_Handler
136 .word 0
137 .word 0
138 .word 0
139 .word 0
140 .word 0
141 .word 0
142 .word 0
143 .word SVC_Handler
144 .word 0
145 .word 0
146 .word PendSV_Handler
147 .word SysTick_Handler
148 .word WWDG_IRQHandler
149 .word PVD_VDDUSB_LP_IRQHandler
150 .word RTC_IRQHandler
151 .word FLASH_IRQHandler
152 .word RCC_CRIS_IRQHandler
153 .word EXTI0_1_IRQHandler
154 .word EXTI2_3_IRQHandler
155 .word EXTI4_15_IRQHandler ←
156 .word TS_IRQHandler
157 .word DMA1_Channel1_IRQHandler
158 .word DMA1_Channel2_3_IRQHandler
159 .word DMA1_Channel4_5_6_7_IRQHandler
160 .word ADC1_COMP_IRQHandler
161 .word TIM1_BRK_UP_TRG_COM_IRQHandler
162 .word TIM1_CC_IRQHandler
163 .word TIM2_IRQHandler
164 .word TIM3_IRQHandler
165 .word TIM6_DAC_IRQHandler
166 .word TIM7_IRQHandler
167 .word TIM14_IRQHandler
168 .word TIM15_IRQHandler
169 .word TIM16_IRQHandler
170 .word TIM17_IRQHandler
171 .word I2C1_IRQHandler
172 .word I2C2_IRQHandler
173 .word SPI1_IRQHandler
174 .word SPI2_IRQHandler
175 .word USART1_IRQHandler
176 .word USART2_IRQHandler
177 .word USART3_4_IRQHandler
178 .word CEC_CAN_IRQHandler
179 .word USB_LP_IRQHandler
180 .word BootRAM /* @0x108. This
181                STM32F0xx dev
```

C'est la définition de la table de vecteurs d'interruption STM32F0. Il s'agit en fait d'appels de fonction, en fonction du signal d'interruption reçu.

Sur la base de cette liste, on peut supposer que le nom probable de la fonction appelée lors de l'événement **EXTI13** est **EXTI4_15_IRQHandler()**

Une fonction telle que **EXTI4_15_IRQHandler()** est connue sous le nom de "**Interrupt Handler**", ou "**Interrupt Service Routine (ISR)**". Il s'agit de la fonction qui "gère" ou "sert" l'événement d'interruption.

L'implémentation du gestionnaire d'interruption peut être écrite n'importe où dans la structure de votre fichier de projet. Il peut être associé à **main.c**, ou à **bsp.c**, ou à n'importe quel autre endroit. Bien qu'il soit courant de regrouper les gestionnaires d'interruptions dans un fichier source sous **app/src** appelé **stm32f0xx_it.c**.

Ajoutons notre implémentation de l'ISR pour l'interruption de la ligne 13 EXTI au fichier **stm32f0xx_it.c**.

```
/******  
*****/  
/*          STM32F0xx Peripherals Interrupt Handlers          */  
/* Add here the Interrupt Handler for the used peripheral(s) (PPP), for the */  
/* available peripheral interrupt handler's name please refer to the startup */  
/* file (startup_stm32f0xx.s). */  
/******  
*****/  
  
/**  
 * This function handles EXTI line 13 interrupt request.  
 */  
  
void EXTI4_15_IRQHandler()  
{  
    // Verifier si cest PC13 (Ligne 13) qui a declencher interruption  
    if ((EXTI->PR & EXTI_PR_PR13_Msk) != 0)  
    {  
        // Acquitter ou valider la demande interruption  
        // Reset du drapeau en écrivant '1' sur le bit b13 de EXTI_PR  
        EXTI->PR = EXTI_PR_PR13;  
  
        // Tache a realiser  
        my_printf("#");  
    }  
}
```

Vous devez inclure **stm32f0xx.h** dans ce fichier afin d'obtenir les noms de registre. Il est préférable de l'ajouter au fichier **main.h** qui est déjà inclus dans **stm32f0xx_it.c**.

```
#include "stm32f0xx.h"  
  
#ifndef APP_INC_MAIN_H_  
#define APP_INC_MAIN_H_  
.....
```

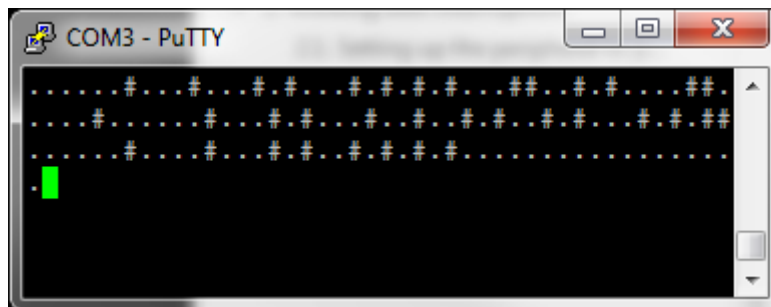
Vous avez peut-être déjà remarqué que le gestionnaire d'interruptions **EXTI4_15_IRQHandler()** est partagé entre des lignes d'interruption **externes allant de 4 à 15**. Jusqu'à présent dans notre application, il n'y a aucune chance qu'une ligne autre que 13 produise une interruption (seule la ligne 13 a été activée dans le périphérique EXTI). Quoi qu'il en soit, il est fortement recommandé de toujours vérifier l'origine de l'interruption au début de l'ISR.


Par conséquent, la première tâche consiste à s'assurer que le déclenchement de fonction est le résultat d'un événement sur la broche **PC13**. Pour cela, nous pouvons lire un drapeau dans le **registre du contrôleur EXTI dédié** pour voir quelle ligne produit réellement le signal d'interruption.

Ensuite, vous devez effacer le drapeau d'interruption dans le contrôleur EXTI. Si vous omettez cette opération, l'interruption sera considérée comme non desservie, et aucun autre signal d'interruption ne sera déclenché à partir de cette source.

Enfin, la tâche dédiée est une simple impression ou affichage **"#"** sur la console (envoi vers le PC par liaison USART). Si l'ISR effectue toutes les tâches que vous souhaitez en réponse à un événement, le programme principal peut simplement l'oublier.

Laissez la fonction **main()** telle quelle, Compiler et tester l'exécution du programme



	Implémenter cette nouvelle méthode et réaliser les essais avec différentes valeurs de <code>BSP_DELAY_TIM_ms(x)</code> ; : x=500 et 1000 ; que constatez-vous ?
	Sauvegarder votre code main, le fichier stm32f0xx_it.c et les captures d'écrans montrant vos essais

3.4 Gestion du délai d'exécution des ISR

Étant donné qu'une interruption suspend l'exécution du code principal pendant la durée d'exécution de l'**ISR**, il est fortement recommandé de maintenir le temps d'exécution de l'**ISR** aussi court que possible. Cela permet également d'éviter un grand nombre d'appels d'interruption imbriqués (c'est-à-dire l'interruption d'un gestionnaire d'interruption par une interruption de priorité plus élevée).

Dans l'exemple précédent, nous avons utilisé l'**ISR** pour envoyer le caractère "#" à la console par liaison série **USART**. Lorsque vous jouez avec le code, vous verrez le caractère "#" apparaître dans la console presque instantanément lorsque vous appuyez sur le bouton.

Cependant, l'envoi d'octets via un port de communication série (l'**UART** utilisé derrière la fonction **mon_printf()**) est intrinsèquement un processus lent et devrait être évité dans tout **ISR** bien écrit.

Voulons-nous vraiment traiter immédiatement le signal d'interruption ?

Si ce n'est pas le cas, utilisons simplement une variable pour garder une trace de l'événement, puis traitons cette information dans la boucle principale, lorsque nous avons le temps.

Modifiez le fichier **main.c** comme suit :

```
#include <math.h>
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"
static void SystemClock_Config(void);
// variable globale
uint8_t button_irq = 0;
int main(void)
{
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();
    //Initialisation de pin PC13 pour bouton Bleu avec interruption
    BSP_PB_IT_Init();
    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");
    //Initialisation de Timer TIM6 pour delai
    BSP_DELAY_TIM_init();
    // initialisation du controleur interruption NVIC
    BSP_NVIC_Init();
    // boucle principale des applications
    while(1)
    {
        // Traitement de la tache liee a la detection du front descendant du bouton
        if (button_irq == 1)
        {
            mon_printf("#");
            button_irq = 0;
        }
        // Exemple de tache importante a faire
        // envoyer "." au PC par USART et attendre 200 ms
        mon_printf(".");
        BSP_DELAY_TIM_ms(200);
    }
}
```

Ici, nous déclarons une variable globale **button_irq** qui sera utilisée pour enregistrer l'événement du bouton-poussoir (c'est-à-dire le signal d'interruption du **PC13**). La tâche d'impression '#' n'est alors effectuée que lorsque **button_irq** est défini, et ne suspend plus les "choses importantes" que le CPU fait dans la boucle principale.

L'ISR peut maintenant être rendu beaucoup plus rapide en mettant simplement la variable **button_irq** à '1':

Nous pouvons améliorer les choses.

Modifier l'ISR comme ci-dessous :

```
/**
 * This function handles EXTI line 13 interrupt request.
 */

extern uint8_t button_irq;

void EXTI4_15_IRQHandler()
{
    // Vérifier si c'est PC13 (Ligne 13) qui a déclenché l'interruption
    if ((EXTI->PR & EXTI_PR_PR13_Msk) != 0)
    {
        // Acquitter ou valider la demande d'interruption
        // Reset du drapeau en écrivant '1' sur le bit b13 de EXTI_PR
        EXTI->PR = EXTI_PR_PR13;

        // Tâche à réaliser
        button_irq++;
    }
}
```

Modifier également la fonction main comme ci-dessous :

```
#include <math.h>
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"

static void SystemClock_Config(void);

// variable globale
uint8_t button_irq = 0;

int main(void)
{
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();
    //Initialisation de pin PC13 pour bouton Bleu avec interruption
    BSP_PB_IT_Init();
    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");
    //Initialisation de Timer TIM6 pour delai
    BSP_DELAY_TIM_init();
    // initialisation du controleur interruption NVIC
    BSP_NVIC_Init();
    // boucle principale des applications
    while(1)
    {
        // Traitement de la tache lier a la detection du front descendant du bouton
        if (button_irq > 0)
        {
            mon_printf("#");
            button_irq--;
        }
        // Exemple de tache importante a faire
        // envoyer "." au PC par USART et attendre 200 ms
        mon_printf(".");
        BSP_DELAY_TIM_ms(200);
    }
}
```

Compiler le projet et programmer le MCU. Ouvrir une console PuTTY et vérifier le fonctionnement



Implémenter cette nouvelle fonctionnalité et réaliser les essais

Sauvegarder votre code main et le fichier **stm32f0xx_it.c** et les captures d'écrans montrant vos essais

4. Résumé

Dans ce tutoriel essentiel, vous avez appris comment configurer une interruption externe et comment écrire la fonction de traitement correspondante (ISR).