

TP électronique
2020/2021

Microcontrôleur STM32/Cortex M3 et langage assembleur

Jean-Jacques Meneu

Chapitre 1. Introduction.....	3
1.1. Préparation	3
1.2. Compte-rendu.....	3
Chapitre 2. Découverte du microcontrôleur et de l'IDE	4
2.1. Présentation du microcontrôleur	4
2.2. Programmation en langage assembleur.....	9
2.3. Instructions du STM32.....	155
2.4. Modes d'adressage du STM32	15
2.5. Ensemble de développement.....	17
Chapitre 3. Premiers pas	18
3.1. Introduction	188
3.2. Réalisation d'un compteur modulo 10	27
3.3. Comment faire une boucle "for" ?	29
3.4. Créer et appeler des sous-programmes.....	29
3.5. A quoi sert la pile ?	332
3.6. Utilisation des tableaux	34
Chapitre 4. La carte de développement	36
4.1. Périphériques disponibles sur la carte	36
4.2. Gestion des horloges	39
Chapitre 5. Les entrées-sorties	41
5.1. Principe	422
5.2. Initialisation des ports entrées/sorties	43
5.3. Caractéristiques détaillées	443
5.4. Commande d'une LED par un bouton poussoir	44
5.5. Clignotement d'une LED	46
Chapitre 6. Les interruptions	47
6.1. Principe de gestion des interruptions externes	47
6.2. Clignotant avec marche/arrêt sous interruption	48
Annexe 1. Les instructions de saut conditionnel.....	51
Annexe 2. Crer un projet STM32	52
Annexe 3. Bases pour dboguer avec STM32CubeIDE.....	56

Chapitre 1. Introduction

Les objectifs de cette série de TP sont les suivants :

- étudier le fonctionnement d'un microcontrôleur 32 bits, le STM32L152 (référence complète STM32L152) de la famille STM32 de STMicroelectronics. Plus généralement, ce microcontrôleur intègre un cœur Cortex M3 qui est la référence mondiale et vous permettra de programmer les microcontrôleurs de nombreux fabricants.
- étudier un environnement de développement de programmes écrits en assembleur ou en C. Cet environnement est un IDE (Integrated Development Environment) et intègre tous les outils de développement d'un logiciel embarqué (gestionnaire de projet, éditeur avec coloriage syntaxique, assembleur, éditeur de lien, débogueur symbolique, carte de développement du STM32 appelé Nucleo-L152),

Vous développerez tout d'abord des programmes simples mettant en œuvre les différents constituants de la carte. Cela vous familiarisera avec le STM32, sa programmation et son environnement de développement.

Ensuite, vous réaliserez des programmes plus complexes en vous appuyant sur les différents programmes que vous aurez déjà développés.

1.1. Préparation

- lecture du sujet
- réalisation des organigrammes (exemples page 19 à 20)
- révision des programmes précédents (pour ne pas oublier les bases du langage)

1.2. Compte-rendu

- organigramme
- programme source commenté
- résultat des tests

Chapitre 2. Découverte du microcontrôleur et de l'IDE

2.1. Présentation du microcontrôleur

Un microcontrôleur est un microprocesseur ou CPU autour duquel ont été rajoutés des mémoires et des périphériques.

La figure ci-dessous présente le microcontrôleur STM8L152RE.

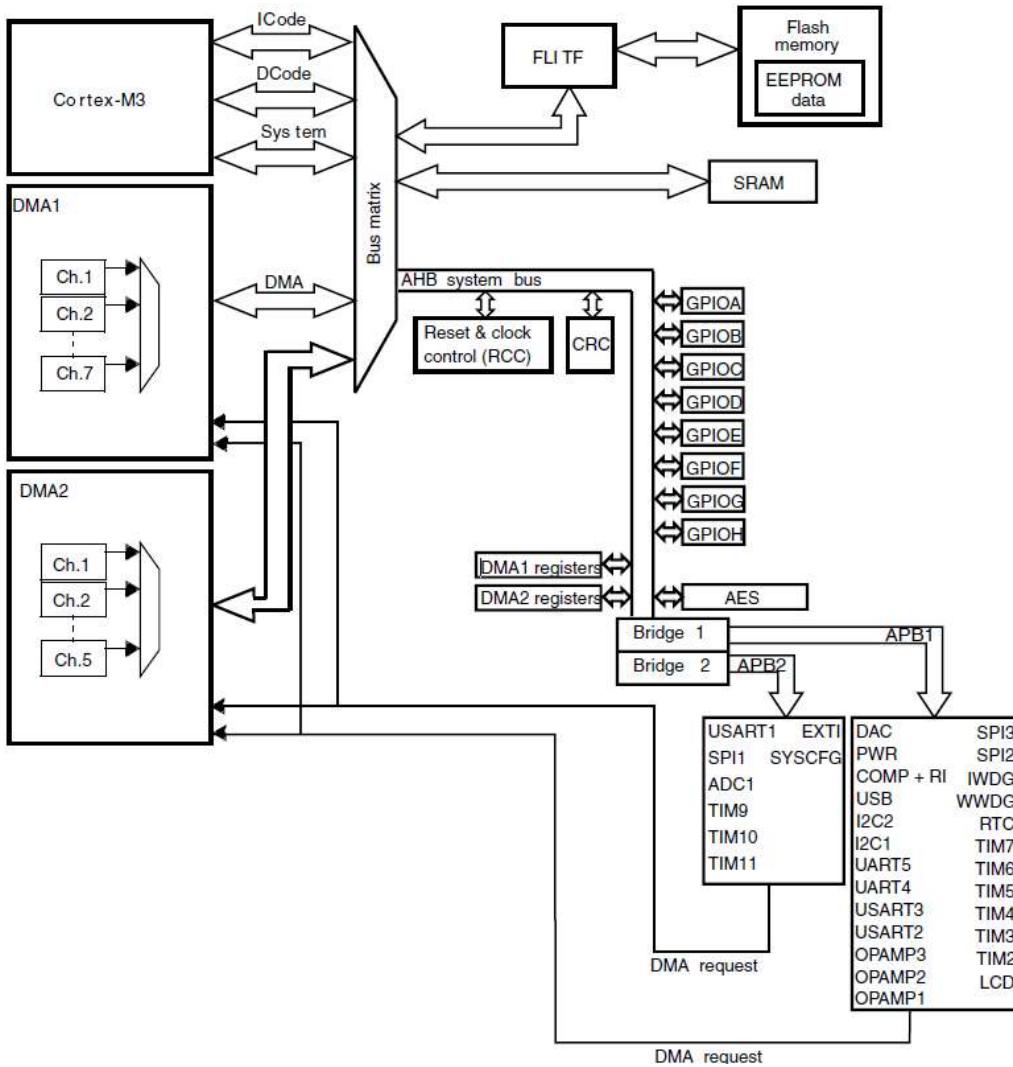


Figure 1. Schéma block du STM32L152

L'architecture du **CPU** est de type HARVARD avec un bus d'adresse de 32 bits vers la mémoire programme et un bus d'adresse de 32 bits entre la mémoire et les données. Le bus de données vers la mémoire programme est de 32 bits et vers la mémoire de données de 32 bits.

Le **CPU** contient 18 registres tous de taille 32 bits :

- douze registres généraux (R0 à R12)
- un compteur ordinal PC (Program Counter),
- Un pointeur de pile SP (Stack Pointer)
- Une registre de lien LR (Linked Register)

- un registre d'état PSR (Program Status Register),
- un pointeur de pile SP (16 bits).

La **mémoire** présente est de type :

- Flash (256 Ko) pour la mémoire programme,
- RAM (80Ko),
- EEPROM (16Ko).

Les **périphériques** sont :

- ports d'entrées/sorties (PORT A, PORT B, PORT C, PORT D) avec un nombre de broches qui varient
- Watchdog : IWDG (Independant Watchdog) et WWDG (Window Watchdog)
- Timers : timers basiques et avancés, 32 ou 16-bit
- module de communication I2C
- module de communication SPI
- module de communication UART
- convertisseur analogique numérique
- convertisseur numérique analogique
- ...

L'accès aux périphériques se fait à travers différents bus : AHB, APB1 et APB2. Vous ferez attention de gérer l'horloge pour chaque périphérique.

L'utilisation des registres des périphériques est expliquée dans la documentation technique du µcontrôleur (reference Manual) et bien sûr pendant les cours.

Dans les systèmes à µcontrôleur, contrairement aux ordinateurs qui utilisent uniquement de la RAM, différents types de mémoires sont utilisés selon leur fonction : la mémoire non volatile à lecture (ROM ou Flash) seule pour le programme et les constantes, la mémoire volatile à lecture et écriture (RAM) pour les données temporaires. Cependant vue du programmeur, la mémoire forme un unique espace adressable continu, comme le montre la figure suivante :

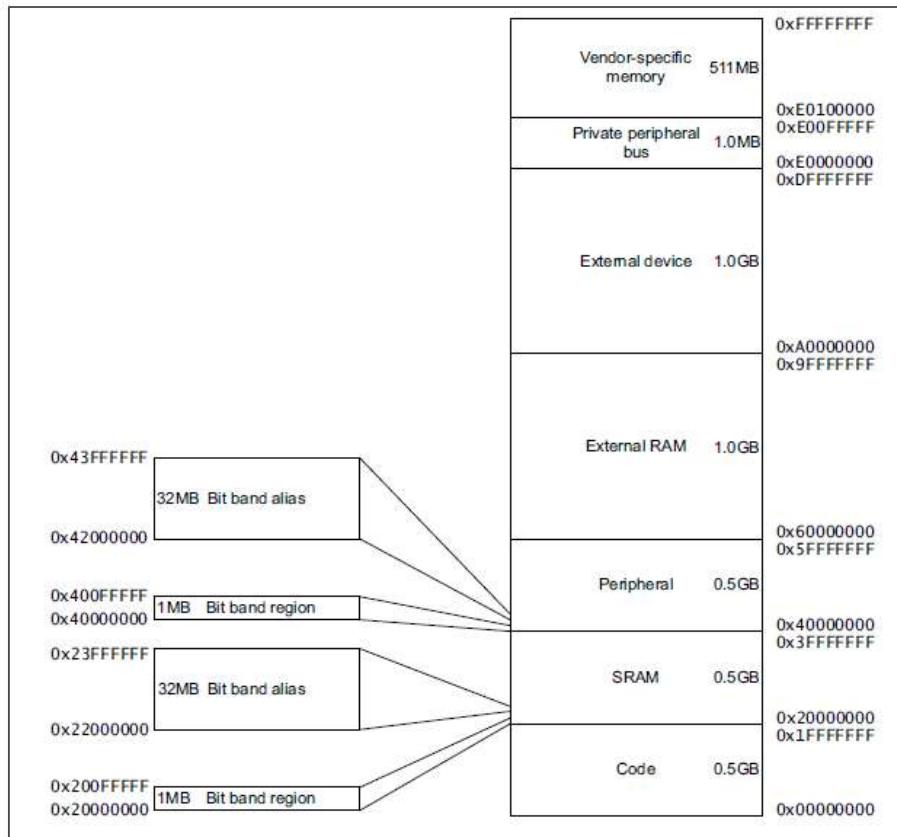


Figure 1 : organisation de l'espace adressable

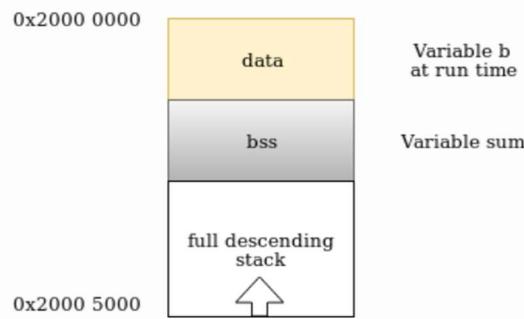


Figure 3 : organisation de l'espace RAM

On peut voir dans cette structure trois espaces **RAM** différents La RAM est composée de trois espaces différents:

1. le premier espace correspond aux variables qui sont initialisés dans le programme lors de l'initialisation. Cet espace, appelé data, commence toujours à 0x2000 0000 et est ascendant
2. le second espace bss correspond à des variables qui ne sont pas initialisés mais dont on a réservé un espace mémoire. Son adresse de départ dépend de la taille de la section data.
3. le troisième espace correspond à la **pile** (Stack). L'adresse de départ de la pile peut être programmée et est descendante. La valeur de départ maximum dépend de la taille SRAM. Avec 80Ko, cette valeur est égale à 0x2001 4000

Les périphériques sont désignés par un nom et correspondent en réalité à l'accès à des registres particuliers se trouvant dans les adresses mémoire à partir de 0x4000 0000.

Les vecteurs d'interruption démarrent à l'adresse 0x0000 0004 de la mémoire programme. Ils définissent les adresses des sous-programmes à exécuter lorsqu'une interruption arrive. Par exemple l'interruption due au Reset du µcontrôleur est liée à l'adresse du début du programme principal (main). Ainsi, dès que le µcontrôleur est mis sous tension, l'interruption du reset est générée, et cela force le CPU à sauter à l'adresse du début du programme pour l'exécuter. Un tableau donne la correspondance entre les vecteurs d'interruptions (le nombre d'interruptions dépend des microcontrôleurs. ARM permet un maximum de 256 interruptions, mais la vaste majorité des microcontrôleurs possèdent beaucoup de vecteurs d'interruption. Le STM32I152RE possède 56 interruptions..

Exception number	IRQ number	Offset	Vector
83	67	0x014C	IRQ67
.	.	.	.
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			
9			Reserved
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Figure 4 : Table des vecteurs d'interruption

Table 51. Vector table (Cat.4, Cat.5 and Cat.6 devices)

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000_0000
-	-3	fixed	Reset	Reset	0x0000_0004
-	-2	fixed	NMI_Handler	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000_0008
-	-1	fixed	HardFault_Handler	All class of fault	0x0000_000C
-	0	settable	MemManage_Handler	Memory management	0x0000_0010
-	1	settable	BusFault_Handler	Pre-fetch fault, memory access fault	0x0000_0014
-	2	settable	UsageFault_Handler	Undefined instruction or illegal state	0x0000_0018
-	-	-	-	Reserved	0x0000_001C - 0x0000_002B
-	3	settable	SVC_Handler	System service call via SWI instruction	0x0000_002C
-	4	settable	DebugMon_Handler	Debug Monitor	0x0000_0030
-	-	-	-	Reserved	0x0000_0034
-	5	settable	PendSV_Handler	Pendable request for system service	0x0000_0038
-	6	settable	SysTick_Handler	System tick timer	0x0000_003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000_0040
1	8	settable	PVD	PVD through EXTI Line16 detection interrupt	0x0000_0044
2	9	settable	TAMPER_STAMP	Tamper, LSECSS andTimeStamp through EXTI line19 interrupts	0x0000_0048
3	10	settable	RTC_WKUP	RTC Wakeup through EXTI line20 interrupt	0x0000_004C
4	11	settable	FLASH	Flash global interrupt	0x0000_0050
5	12	settable	RCC	RCC global interrupt	0x0000_0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000_0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000_005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000_0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000_0064

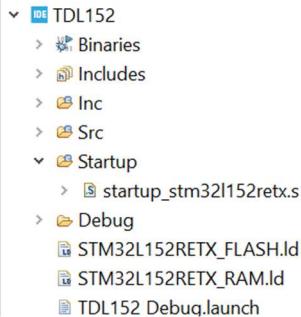
Figure 4 : Table des vecteurs d'interruption avec adresse

2.2. Programmation en langage assembleur

2.2.1. Squelette de programme

La programmation du microcontrôleur se fait en langage assembleur suivant les instruction Thumb-2, ARMV7. L'assembleur comporte un jeu d'instruction complet et de nombreux modes d'adressage.

Le programme de démarrage est toujours écrit en langage assembleur. Lorsque STM32CubeIDE initialise un projet, un fichier startup_stm32l152retx.s est créé. Un fichier .s signifie que le fichier contient du code assembleur.



Les directives dans ce fichier suivent les normes gcc. Il n'est pas nécessaire de connaître ces directives pour le TP. Un autre fichier vous est fourni avec des indications où écrire le code et déclarer les variables afin de vous concentrer uniquement sur le code assembleur. Après la création d'un projet, remplacer le fichier de startup par celui fourni.

Le squelette du programme " stm32l152retx.s " fourni a pour but de bien définir la structure d'un programme et les différentes zones mémoires nécessaires pour la construction du code exécutable. Il servira de base pour l'écriture de vos programmes et comporte plusieurs zones.

2.2.1.1. Zone de description

Cette zone est constituée de commentaires et permet de décrire ce que fait le programme. Il est très important de commenter les programmes, quel que soit le langage de programmation, mais c'est particulièrement vrai en langage assembleur, car c'est un langage peu lisible.

La syntaxe des commentaires est la suivante :

```
/* ceci est un commentaire sur une ligne */
INST           // commentaire d'une instruction
```

2.2.1.2. Zone de déclaration des symboles

Cette zone permet de déclarer des symboles pour simplifier la lecture du code source. Ils sont désignés par un nom (on dit également label) et ne prennent pas de place en mémoire (contrairement aux constantes). Lors de la compilation, le label est automatiquement remplacé par la valeur qu'il désigne. C'est l'équivalent d'un "#define" en C.

La syntaxe est la suivante :

```
.equ nom, valeur
```

Par exemple :

```
47 /*********************************************************************
48 *
49 *          AREA TO DECLARE SYMBOLS
50 *
51 /*********************************************************************/
52 .equ myvar, 0x87654321
53
54 /*********************************************************************
55 *
56 *          END  AREA TO DECLARE STMBOLS
57 *
58 /********************************************************************/
```

Remarques :

- Pensez à donner des noms clairs afin de simplifier la lecture du code. Cette remarque est valable pour tous les types de noms que vous définirez : symboles, variables, constantes, sous-programmes.
- Pour la lisibilité de votre code, utilisez la convention suivante où [TAB] désigne la touche de tabulation :

Label :

[TAB]INST [TAB]opérande[TAB] // commentaire

2.2.1.3. Zone de déclaration des variables

Cette zone se trouve dans la RAM de la mémoire.

Chaque variable est déclarée par un nom (ou label) et une taille en nombre d'octets ou de mots(word) (1 mot = 4 octets). Pour simplifier le programme, nous ne déclarerons que des mots. On peut ainsi déclarer également des tableaux et des chaînes de caractère.

La syntaxe est la suivante :

Nom de variable :

.word valeur

Et un pour un tableau

Nom de variable :

.word valeur1, valeur2, valeur3

Quelques exemples de déclaration :

```
*****  
*  
*          AREA TO DECLARE INITIALIZED VARIABLES  
*  
*****  
.section .data  
  
CPT:  
    .word 10  
  
varidata:  
    .word 0x12345678, 0x0A, 0xDEADBEEF  
    .string "hello word"  
  
decalage:  
    .word 0x3  
  
*****  
*  
*          END      AREA TO DECLARE INITIALIZED VARIABLES  
*  
*****
```

De manière similaire, il est également possible de déclarer des variables non initialisées dans la section bss.

2.2.1.4. Zone de déclaration des constantes

Cette zone se trouve dans la zone mémoire programme (Flash). Chaque constante est déclarée par un nom (ou label) et une valeur. On peut aussi déclarer des tableaux de constantes.

```

/*
*
*          AREA TO DECLARE CONSTANTS
*
*/
.section .text
varconstant:
    .word 0xDEADBEEF

END      AREA TO DECLARE CONSTANTS
*/

```

Afin de simplifier le programme, déclarer seulement des .word qui correspondnt à des variables de longueur 32 bits.

La syntaxe est la suivante :

Nom de variable :

.word valeur

Et un pour un tableau

Nom de variable :

.word valeur1, valeur2, valeur3

2.2.1.5. Zone de déclaration des sous-programmes

Cette zone se trouve également dans la mémoire programme. Elle contiendra tous les sous-programmes que vous définirez.

La syntaxe d'un sous-programme est la suivante :

nom : ; nom du sous-programme

INST1

INST2

...

bx lr ; instruction de fin de sous-programme

```

7 ****
3 *
3*          AREA TO DECLARE SUBPROGRAMS
3*
4 ****
2 dosomething:
3     movs r0,#0xAB
4     bx lr
5 ****
7*
3*          END      AREA TO DECLARE SUBPROGRAMS
3*
4 ****

```

2.2.1.6. Programme principal

Ce programme principal se trouve dans la mémoire programme. **C'est le point de départ du programme.**

Le programme démarre au reset et initialise le pointeur de pile. Ceci est généré lors de la création du projet et aucun changement n'est nécessaire.

```
.section .text.Reset_Handler
.weak Reset_Handler
.type Reset_Handler, %function
Reset_Handler:
    ldr r0, =_estack
    mov sp, r0          /* set stack pointer */

/* Copy the data segment initializers from flash to SRAM */
    ldr r0, =_sdata
    ldr r1, =_edata
    ldr r2, =_sidata
    movs r3, #0
    b LoopCopyDataInit
```

Après l'initialisation

Sa syntaxe est la suivante :

Main :

```
    Instructions // initialisations diverses des périphériques (horloge, timer, ports E/S...)
boucl ...
...
b      boucl ; boucle sur une partie du programme
```

```
*****
*
*           Main Program
*
*****
```



```
main:
LoopForever:
    b LoopForever
```

L'instruction "b boucl" effectue un saut dans le programme principal. Celui-ci boucle donc à l'infini. Ceci doit toujours être le cas. En effet un µcontrôleur est en général embarqué dans un système qui fonctionne en permanence tant qu'il est sous tension. Le programme principal doit donc faire en sorte qu'il ne s'arrête jamais grâce à une boucle infinie.

En général, la boucle "infinie" s'effectue par un saut vers une position qui se trouve après la phase d'initialisation.

2.2.1.7. Zone de déclaration des sous-programmes d'interruption

Cette zone se trouve dans la mémoire programme. Elle contiendra tous les sous-programmes d'interruption que vous définirez.

Ces sous-programmes d'interruption sont les sous-programmes à exécuter lorsqu'une interruption (IT) provenant d'un périphérique ou d'un port du µcontrôleur survient.

La syntaxe d'un sous-programme d'interruption est la suivante :

```
/*
 *          Area to Declare interrupt sub-programs
 *
 */
testfunc:
// Instructions
    bx lr

/*
 *          END Area to Declare interrupt sub-programs
 *
 */

```

Le nom des programmes d'interruption est imposé par ARM à des fins de portabilité entre tous les fabricants de microcontrôleurs basés sur le cœur Cortex M. Lors de la création d'un projet, des fonctions sont déjà créées dans la section weak aliases

```
' ****
;*
;* Provide weak aliases for each Exception handler to the Default_Handler.
;* As they are weak aliases, any function with the same name will override
;* this definition.
;*
;****

;     .weak    NMI_Handler
;     .thumb_set NMI_Handler,Default_Handler
```

Toutes les appellent un default_Handler.

Par simplification, pour appeler un programme d'interruption, changer le default_handler par le nom de la fonction qui doit être appelée par l'interruption

```
.weak    EXTI15_10_IRQHandler
.thumb_set EXTI15_10_IRQHandler,testfunc
```

2.2.1.8. Zone de déclaration des vecteurs d'interruption

Cette zone se trouve dans la mémoire programme. Les vecteurs d'interruption sont en fait les liens entre l'interruption elle-même et le sous-programme d'interruption correspondant.

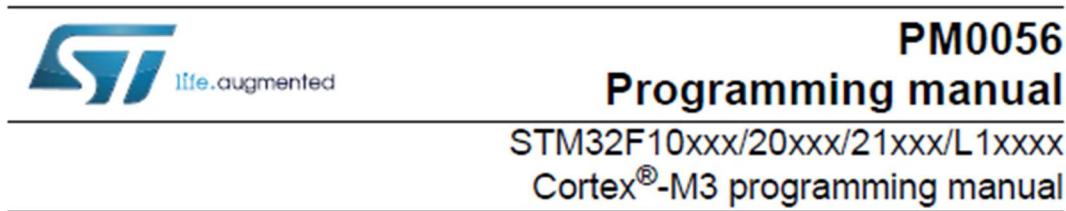
```
/*
 * The STM32L152RETx vector table. Note that the proper constructs
 * must be placed on this to ensure that it ends up at physical address
 * 0x0000.0000.
 *
 */
.section .isr_vector,"a",%progbits
.type g_pfnVectors, %object
.size g_pfnVectors, .-g_pfnVectors

g_pfnVectors:
.word _estack
.word Reset_Handler
.word NMI_Handler
.word HardFault_Handler
```

Ces noms sont imposés par ARM. Il ne faut rien changer à cette zone.

2.3. Instructions du STM32

L'ensemble des instructions est présentée dans le programming Manual du STM32 L152.



L'explication de l'encodage des instructions est expliquée dans le ARMv7-M Architecture Reference Manual.

ARM®v7-M Architecture Reference Manual

2.4. Modes d'adressage du STM32

La famille STM32 possède divers modes d'adressage. Toutes les instructions n'acceptent pas forcément tous les modes d'adressage, il faut donc vérifier dans l'aide quel mode convient.

Lors de vos TP, vous aurez souvent à manipuler des valeurs en décimal, en binaire mais aussi en hexadécimal. En assembleur, cela se traduit ainsi :

- pour coder en décimal la valeur 10, on écrira : **10**
- pour coder en binaire la valeur 10, on écrira : **0b00001010**
- pour coder en hexadécimal la valeur 10, on écrira : **0x0A**

2.4.1. Mode immédiat

Pour charger un registre avec une valeur fixe, cette valeur peut être incorporée à l'instruction. Dans ce cas, la valeur est stockée dans le programme et il n'y a pas besoin de donner l'adresse de la donnée à lire, l'adresse à pointer sera calculée directement par le compilateur.

Par exemple :

Ndoigts	EQU	10
Hexa	EQU	0x10
Mask	EQU	0b10100011
...		
LDR	r0,=Ndoigts	; r0 est chargé avec la valeur 10
LDR	r0,#Hexa	; r0 est chargé avec la valeur hexadécimale 10h
LDR	r0,#Mask	; r0 est chargé avec la valeur binaire 1010 0011 soit A3h

2.4.2. Mode Register

Dans ce mode, l'adresse où est stockée la valeur qui nous intéresse est enregistrée dans le registre rn. La valeur est lire en mémoire et stockée dans le registre rd.

Syntaxe : LDR rd,[rn]

Par exemple :

LDR r1,[r0] ; r0 est chargé avec la valeur du mot situé à l'adresse enregistrée dans r0

2.4.3. Mode Register avec offset immédiat

Dans ce mode, l'adresse où est stockée la valeur qui nous intéresse est enregistrée dans le registre rn incrémenté de l'offset. La valeur est lire en mémoire et stockée dans le registre rd.

Syntaxe : LDR rd,[rn,#offset]

Exemple pour lecture dans un tableau

```
varidata2:  
    .word 0xFFFFFFFF, 0xDEADBEEF, 0xA  
  
    movs r0, #0b1010  
    ldr r0, =varidata2  
    ldr r1, [r0,#4]
```

A la fin de code, la valeur stockée dans r1 vaut 0xDEADBEEF

2.4.4. Mode Register avec registre d'offset

Ce mode est similaire au mode précédent.

Syntaxe : LDR rd,[rn,rm]

```
varidata2:  
    .word 0xFFFFFFFF, 0xDEADBEEF, 0xA  
  
main:  
    movs r0, #0b1010  
    ldr r0, =varidata2  
    ldr r2, =0x4  
    ldr r1, [r0,r2]
```

A la fin de code, la valeur stockée dans r1 vaut 0xDEADBEEF

2.4.12 Instructions

Afin de chercher les instructions, référez-vous au cours et surtout au programming manual qui explique toutes les instructions acceptées par le STM32L152.

2.5. Ensemble de développement

La carte de développement est la NUCLEO-L152RE. Toute la documentation utile est disponible sur le site web de STMicroelectronics

https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards/nucleo-l152re.html

Le logiciel de développement utilisé est le STM32CubeIDE qui a l'avantage de fonctionner sur Windows, MAC et Linux. Il est gratuit et sans limite de taille mémoire.

2.5.1. Crédit d'un projet

Un document d'aide à la création d'un projet STM32 dans l'environnement de développement (IDE) STM32CubeIDE a été rédigé. Référez-vous à ce document pour chaque nouveau programme. Vous le trouverez en annexe.

L'écriture du code en assembleur et en C se fait dans l'IDE avec un éditeur à coloriage syntaxique. La compilation et l'édition de lien se font dans le même environnement ainsi que le débogage. Noter que Le STM32CubeIDE ne possède pas de simulateur et la carte est donc nécessaire pour tester un programme.

2.5.2. Utilisation du débogueur

Le STM32CubeIDE possède de nombreuses fonctionnalités qui ne peuvent pas être toutes enseignées. Néanmoins, l'annexe 3 et le chapitre 3 vous montre les fenêtres indispensables à utiliser pour bien développer un programme. Lors de la compilation, lisez les messages d'erreur. Lors du débogage, utilisez tous ces outils à votre disposition pour bien comprendre l'exécution de votre code.

Les enseignants présents avec vous sont également là pour vous aider à prendre en main cet environnement, donc n'hésitez pas à poser des questions.

Chapitre 3. Premiers pas

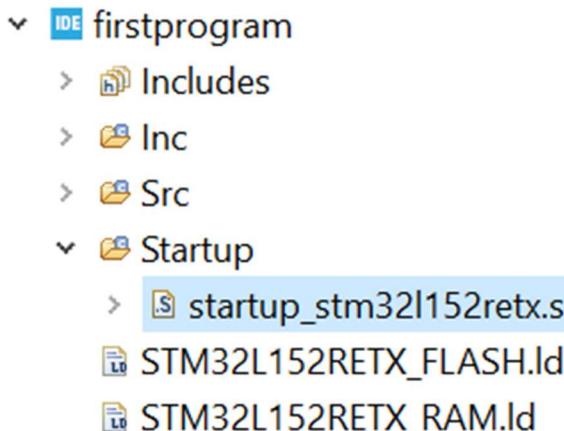
3.1. Introduction

Les premières étapes consistent à organiser l'espace de travail puis à mettre en place les éléments vous permettant de créer votre premier programme "assembleur".

En premier lieu, suivre les étapes décrites dans l'annexe 2 "Créer un projet STM32 en assembleur".

Après avoir créé un projet nous allons vous montrer comment écrire un premier programme et comment utiliser les fenêtres de débogage.

Dans votre projet, double-cliquer pour ouvrir le fichier startup_stm32l152retx.s



Repérer, comme expliqué pendant le cours, les différentes zones pour écrire le programme. Par exemple, la zone pour déclarer des variables initialisées

```
1<
76 /*****  
77 *  
78 *          AREA TO DECLARE INITIALIZED VARIABLES  
79 *  
80 /*****  
81 .section .data  
82  
83  
84  
85 /*****  
86 *  
87 *          END    AREA TO DECLARE INITIALIZED VARIABLES  
88 *  
89 /*****  
--
```

Pour déclarer une variable sur 32 bits (par souci de simplification, nous ne déclarerons que des variables 32-bit pour les TP en assembleur). La syntaxe est la suivante :

Nom de variable :

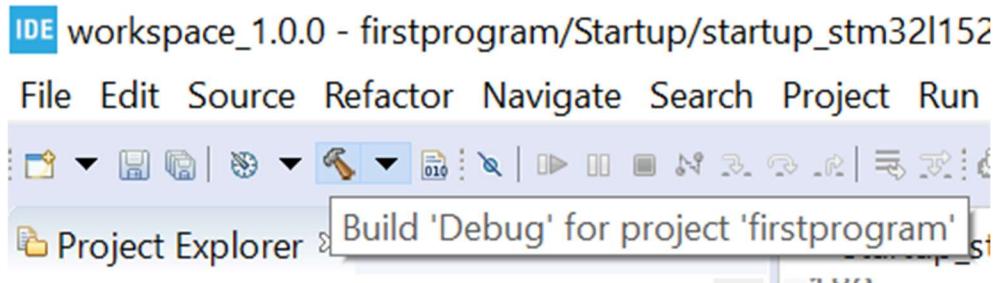
.word valeur de la variable

Donc pour déclarer la variable de nom ‘maVariable’ initialisée à 2, le code est écrit comme ci-dessous

```
76 /******  
77 *  
78 *          AREA TO DECLARE INITIALIZED VARIABLES  
79 *  
80 */*****  
81 .section .data  
82  
83 maVariable:  
84     .word    2  
85  
86  
87 /******  
88 *          END      AREA TO DECLARE INITIALIZED VARIABLES  
89 *  
90 */*****  
91 /******
```

Nous allons faire un programme qui récupère l’adresse de la variable maVariable au début du programme puis soustrait 1 dans la boucle infinie.

Compiler le programme



La compilation donne des informations sur la taille mémoire du programme et

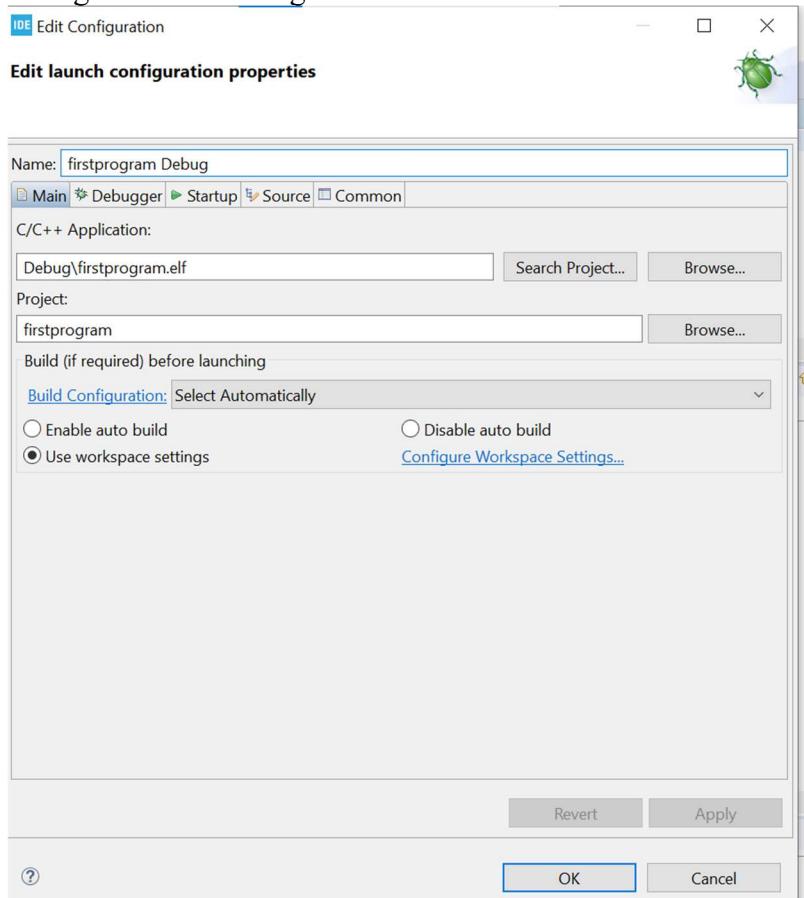
```

arm-none-eabi-objdump -h -S firstprogram.elf > "firstprogram.list"
arm-none-eabi-objcopy -O binary firstprogram.elf "firstprogram.bin"
arm-none-eabi-size firstprogram.elf
    text      data      bss      dec      hex filename
      548        12     1564     2124     84c firstprogram.elf
Finished building: default.size.stdout
Finished building: firstprogram.list
Finished building: firstprogram.bin

```

10:28:37 Build Finished. 0 errors, 0 warnings. (took 1s.81ms)

Lancer le programme en mode debug. La première fois, il est demandé de confirmer la configuration du debug.



Cliquer sur ok

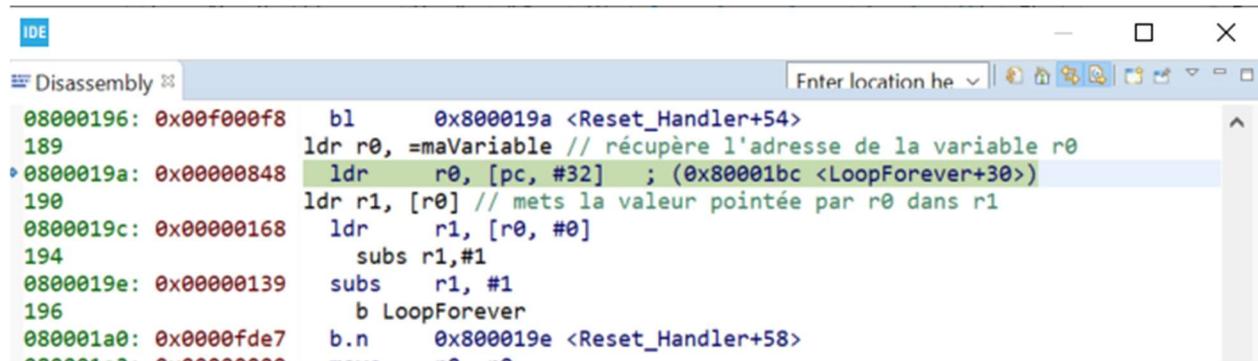
Le programme s'arrête à la première ligne de code du main.

```

7 main:
8
9 ldr r0, =maVariable // récupère l'adresse de la variable r0
10 ldr r1, [r0] // mets la valeur pointée par r0 dans r1
11
12 LoopForever:
13 // soustrait 1 à r1 en mettant à jour le registre xPSR
14     subs r1,#1
15
16     b LoopForever
17
18 .size Reset_Handler, .-Reset_Handler
19

```

Ouvrir la fenêtre de désassemblage, qui permet de voir le codé écrit, le code compilé, l'encodage et l'adresse des instructions.



The screenshot shows the 'Disassembly' tab of a debugger interface. The assembly code is as follows:

```

08000196: 0x00f000f8    bl      0x800019a <Reset_Handler+54>
189          ldr r0, =maVariable // récupère l'adresse de la variable r0
0800019a: 0x00000848    ldr r0, [pc, #32] ; (0x80001bc <LoopForever+30>)
190          ldr r1, [r0] // mets la valeur pointée par r0 dans r1
0800019c: 0x00000168    ldr r1, [r0, #0]
194          subs r1,#1
0800019e: 0x00000139    subs r1, #1
196          b LoopForever
080001a0: 0x0000fde7    b.n   0x800019e <Reset_Handler+58>

```

Grâce à la fenêtre de désassemblage, on voit que le programme est stoppé à l'adresse 0x0800019A. Bien noté que l'instruction à l'adresse 0x0800019A n'est pas encore exécuté. Le Program Counter pointe sur la prochaine instruction à exécuter au prochain coup d'horloge et non sur l'instruction qui vient d'être exécutée !

Ouvrir la fenêtre des registres.

Name	Value
General Registers	
r0	0x20000000 (Hex)
r1	0x20000004 (Hex)
r2	0x20000020 (Hex)
r3	0x8000149 (Hex)
r4	0x20000020 (Hex)
r5	0x0 (Hex)
r6	0x0 (Hex)
r7	0x0 (Hex)
r8	0x0 (Hex)
r9	0
r10	0
r11	0
r12	0
sp	0x20014000
lr	0x800019b (Hex)
pc	0x800019a <Reset_Handle...
xpsr	0x61000000 (Hex)

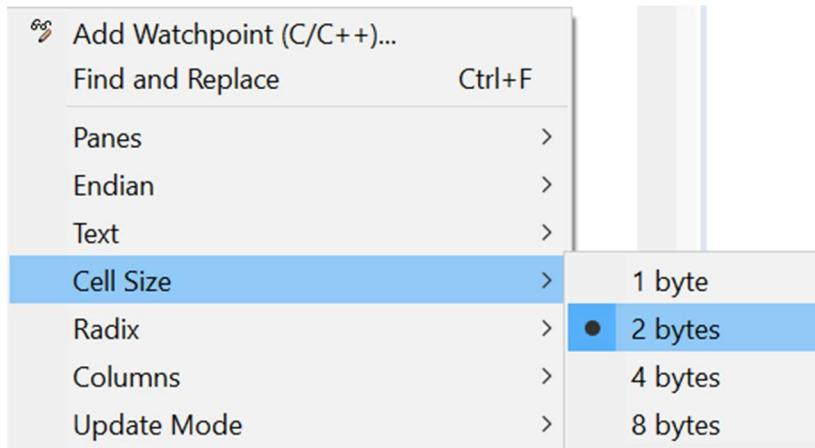
La valeur du Program Counter est bien 0x0800019A. La prochaine instruction à exécuter sera :
 ldr r0, =mavariable

Ouvrir la fenêtre Memory Browser et faire apparaître la mémoire à l'adresse 0x0800019A

Memory Browser	Memory
0x0800019A	
0x800019a - 0x800019A <Traditional>	
0x0800019A 68014808 E7FD3901 40000000 00002001 00042000	
0x080001AE 022C2000 00040800 00202000 00002000 E7FE2000	
0x080001C2 B5700000 4E0C2500 1BA44C0D 42A510A4 F000D109	
0x080001D6 2500F81A 4C0A4E0A 10A41BA4 D10542A5 F856BD70	
0x080001EA 47983025 E7EE3501 3025F856 35014798 0224E7F2	

Par défaut, la présentation est faîte par blocs de 32 bits. Comme les instructions sont codées sur 16 bit, il peut être plus lisible, pour étudier l'encodage, de faire un affichage sur 2 octets.

Placer le curseur sur la fenêtre Memory browser et cliquer sur le bouton droit de la souris.
 Sélectionner la taille de la cellule sur 2 octets.



La fenêtre est alors organisée comme suit

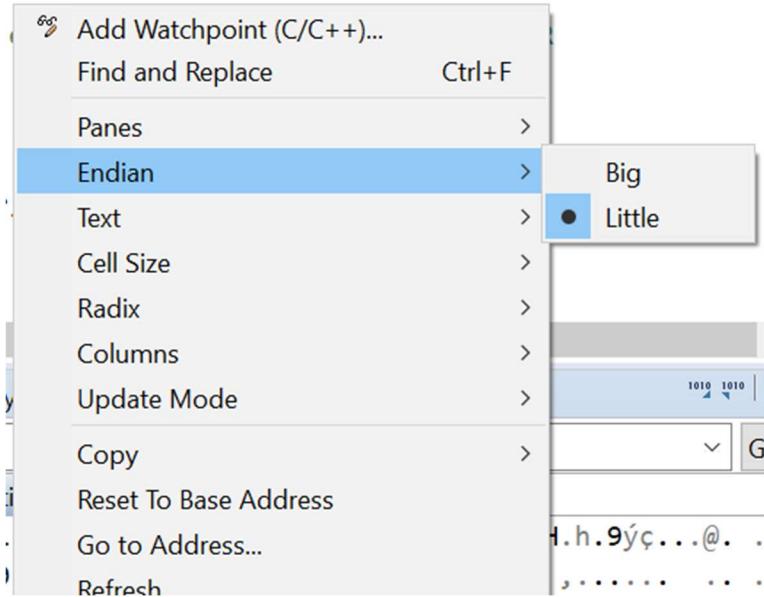
The screenshot shows the 'Memory Browser' window with the title bar 'Memory Browser' and 'Memory'. The address bar shows '0x0800019A'. The main pane displays a memory dump from address 0x0800019A to 0x0800019A. The dump consists of several lines of hex values:

```
0x0800019A 4808 6801 3901 E7FD 0000 4000 2001 0000 2000 0004
0x080001AE 2000 022C 0800 0004 2000 0020 2000 0000 2000 E7FE
0x080001C2 0000 B570 2500 4E0C 4C0D 1BA4 10A4 42A5 D109 F000
0x080001D6 F81A 2500 4E0A 4C0A 1BA4 10A4 42A5 D105 BD70 F856
0x080001EA 3025 4798 3501 E7EE F856 3025 4798 3501 E7F2 0224
```

On voit qu'à l'adresse 0x0800019A, en mémoire, nous avons l'encodage 0x4808 alors que la fenêtre de désassemblage donne 0x0848.

Le STM32L152 est configuré en little endian.

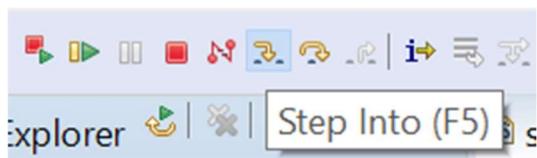
Mettre le curseur sur la fenêtre Memory Browser, cliquer sur le bouton droit et sélectionner little endian. L'affichage est bien en little endian



Si on passe à une présentation par octet, on voit bien qu'on a 0x08 puis 0x48 en mémoire

```
0x800019a - 0x800019A <Traditional> x
0x0800019A 08 48 01 68 01 39 FD E7 00 00 00 40 01 20 00 00 00 20 04 00
0x080001AE 00 20 2C 02 00 08 04 00 00 20 20 00 00 20 00 00 00 20 FE E7
0x080001C2 00 00 70 B5 00 25 0C 4E 0D 4C A4 1B A4 10 A5 42 09 D1 00 F0
0x080001D6 1A F8 00 25 0A 4E 0A 4C A4 1B A4 10 A5 42 05 D1 70 BD 56 F8
0x080001EA 25 30 98 47 01 35 EE E7 56 F8 25 30 98 47 01 35 F2 E7 24 02
0x080001FE 00 08 24 02 00 08 24 02 00 08 28 02 00 08 F8 B5 00 BF F8 BC
```

Pour exécuter une instruction, faire un step into en cliquant sur l'icône flèche



L'adresse de maVariable est copiée dans r0. La valeur de r0 vaut 0x20000000.

Grâce au memory browser aller à l'adresse 0x20000000. La valeur 2 est bien stockée à cet emplacement mémoire

```
0x20000000 <Traditional> x
0x20000000 00000002 00000000 00000000 00000000 00000000 00000000
0x20000018 00000000 00000000 02C06800 F245D40D F8DF5055 600815E8
0x20000030 F8DF2006 600815E4 70FFF640 15DCF8DF 20016008 B5F8BD02
```

Pour récupérer la valeur 2, exécuter l'instruction grâce au step into
La valeur du registre r1 vaut alors 2 ; remarque que le PC change également.

Name	Value
General Registers	
r0	0x20000000 (Hex)
r1	0x2 (Hex)
r2	0x20000020 (Hex)
r3	0x8000149 (Hex)
r4	0x20000020 (Hex)
r5	0x0 (Hex)
r6	0x0 (Hex)
r7	0x0 (Hex)
r8	0x0 (Hex)
r9	0
r10	0
r11	0
r12	0
sp	0x20014000
lr	0x800019b (Hex)
pc	0x800019e <Reset_Handle.
xpsr	0x61000000 (Hex)
PRIMASK	0
BASEPRI	0

Exécuter le programme pas à pas. Lorsque r0 vaut 0, xpsr vaut 0x6100 0020
0x6 = 0b0110 et correspond aux drapeaux N, Z, C et V

Name	Value
General Registers	
r0	0x20000000 (Hex)
r1	0x0 (Hex)
r2	0x20000020 (Hex)
r3	0x8000149 (Hex)
r4	0x20000020 (Hex)
r5	0x0 (Hex)
r6	0x0 (Hex)
r7	0x0 (Hex)
r8	0x0 (Hex)
r9	0
r10	0
r11	0
r12	0
sp	0x20014000
lr	0x800019b (Hex)
pc	0x80001a0 <Reset_Handle.
xpsr	0x61000020 (Hex)

Continuer le programme pour obtenir 0xFFFF FFFF dans r1. Vérifier comment xPSR a été modifié. Reprendre le cours pour en comprendre la signification.

Name	Value
General Registers	
r0	0x20000000 (Hex)
r1	0xffffffff (Hex)
r2	0x20000020 (Hex)
r3	0x8000149 (Hex)
r4	0x20000020 (Hex)
r5	0x0 (Hex)
r6	0x0 (Hex)
r7	0x0 (Hex)
r8	0x0 (Hex)
r9	0
r10	0
r11	0
r12	0
sp	0x20014000
lr	0x800019b (Hex)
pc	0x80001a0 <Reset_Handle.
xpsr	0x81000020 (Hex)
PRIMASK	0
BASEPRI	0

Name : xpsr
 Hex:0x81000020
 Decimal:-2130706400
 Octal:020100000040
 Binary:10000001000000000000000000001000
 Default:-2130706400

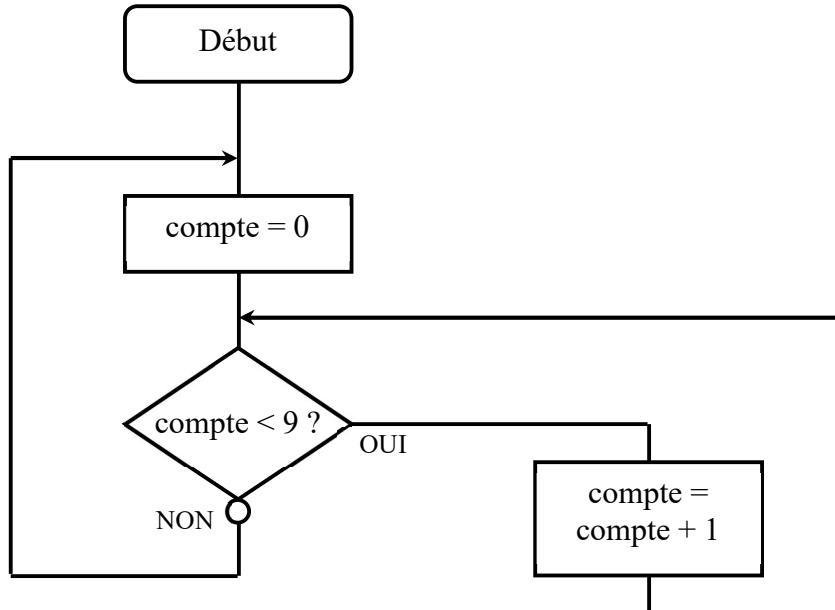
Pour bien déboguer les programmes, il faudra suivre une démarche similaire pour trouver vos erreurs. A vous de jouer maintenant.

3.2. Réalisation d'un compteur modulo 10

Cet exercice consiste à réaliser un programme qui compte de 0 à 9 puis recommence à 0. Ce programme boucle indéfiniment.

Le compteur sera mémorisé dans une variable nommée "compte" que vous devrez définir dans la zone des variables. Vous utiliserez une séquence répétitive du type "tant que ... faire".

L'organigramme suivant illustre l'architecture de ce programme :



Vous devez :

- Ecrire le programme en respectant l'organigramme ci-dessus.
- Compiler le programme.
- Tester le programme avec le débogueur en mode pas à pas afin de vérifier son bon fonctionnement.
- Visualiser et relever l'évolution des registres PC, PSR, R0, R1.
- Visualiser et relever l'évolution de la variable "compte". A quelle adresse mémoire se trouve t'elle ?.

Déroulement pas à pas du programme "compteur modulo 10" :

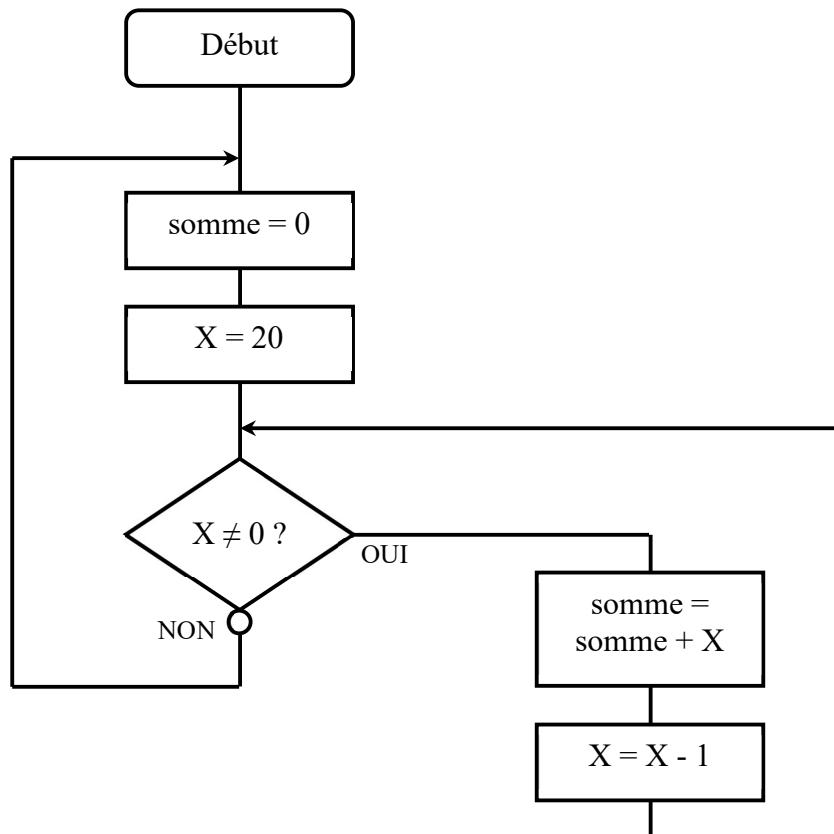
Les valeurs des registres et de la variable correspondent à l'état **avant** l'exécution de l'instruction.

3.3. Comment faire une boucle "for" ?

Cet exercice consiste à réaliser, en assembleur, une boucle "for", connue dans les langages évolués. Pour cela, on vous demande de calculer la somme des 20 premiers entiers (1 à 20).

La somme sera mémorisée dans une variable d'un octet nommée "somme" que vous devrez définir dans la zone des variables. Par contre, l'indice de la boucle "X" ne sera pas déclaré comme une variable.

L'organigramme suivant illustre l'architecture de ce programme :



Vous devez :

- Calculer la somme finale attendue pour juger le fonctionnement du programme.
- Écrire le programme en respectant l'organigramme ci-dessus.
- Tester le programme avec le débogueur en mode pas à pas afin de vérifier son bon fonctionnement.
- Visualiser la fenêtre "désassembleur".
- Visualiser la fenêtre "mémoire".
- Situer dans la mémoire, la variable "somme" et le programme.

3.4. Créer et appeler des sous-programmes

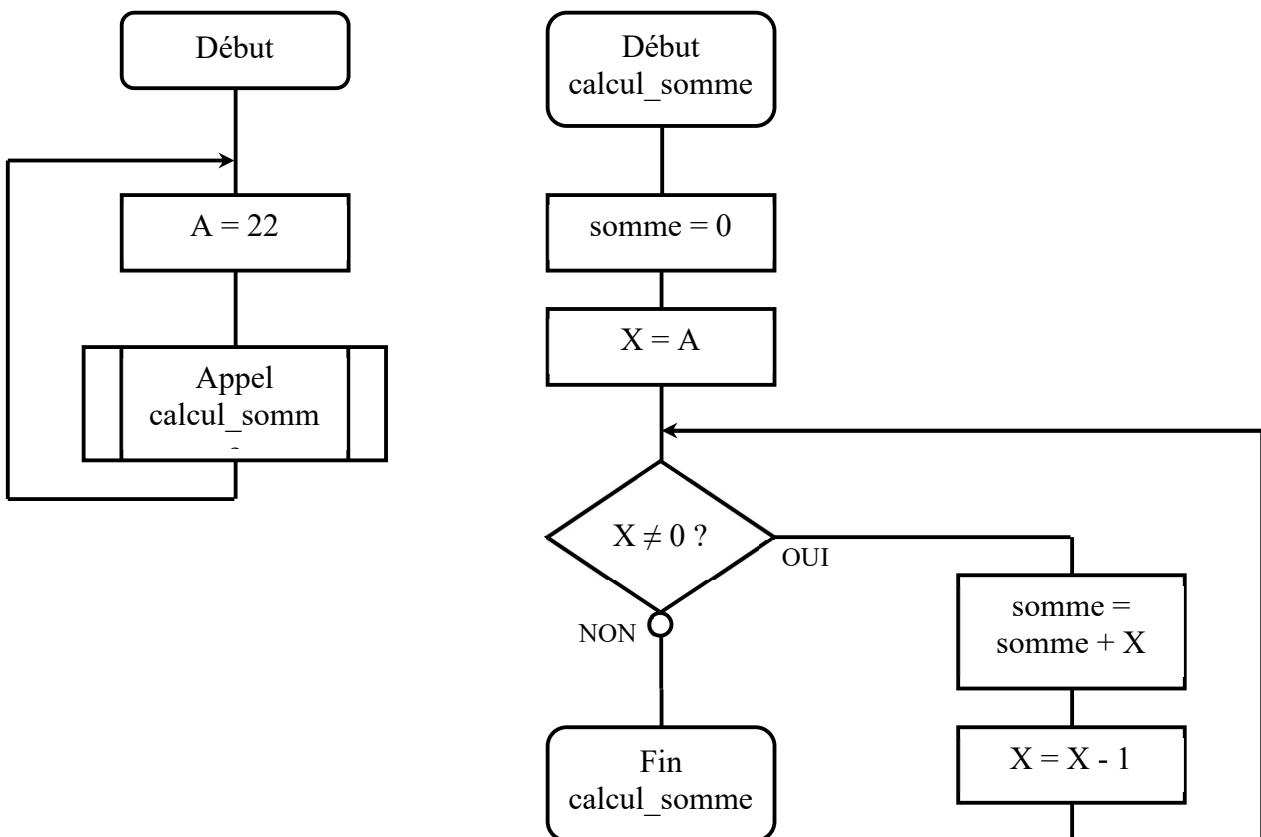
Comme dans les langages évolués, l'assembleur permet de faire des appels à des sous-programmes (procédures ou fonctions), permettant de rendre plus lisible et plus efficace un programme.

Cet exercice consiste à reprendre l'exercice précédent de calcul de la somme des 20 premiers entiers (dans un nouveau répertoire), et de créer le sous-programme "calcul_somme", qui prendra en entrée, dans le registre r0, le nombre d'entiers à ajouter (on ne se limite plus à 20) et calculera la somme qu'il retournera dans le registre r0. Le programme principal doit alors stocker le résultat dans la variable "somme".

Le programme principal effectuera les opérations suivantes :

- mettre un nombre dans r0 (nombre d'entiers que l'on veut ajouter),
- appeler "calcul_somme" avec l'instruction **bl**.

L'organigramme suivant illustre l'architecture de ce programme :



Vous devez :

- Ecrire le programme en respectant l'organigramme ci-dessus.
- Tester le programme avec le débogueur en mode pas à pas afin de vérifier son bon fonctionnement.
- Relever l'évolution des registres PC et LR.

Appel et retour de sous-programme "calcul_somme" :

Les valeurs des registres et de la variable ainsi que le contenu de la pile correspondent à l'état avant l'exécution de l'instruction.
Seules les données dans la pile aux adresses supérieures au Stack Pointer concernent votre programme.

Program Counter PC	Link Register LR	Reg R0	Reg R1	Condition Flag N Z C	VARIABLE somme	INSTRUCTION	OBSERVATIONS

3.5. A quoi sert la pile ?

Cet exercice consiste à reprendre l'exercice précédent pour illustrer l'utilisation de la pile (stack) avec les contraintes suivantes :

Calcul_somme est appelé dans la boucle infini. Dans le registre r0, vous stockerez le nombre de fois que calcul_somme est appelé.

R0 sert à passer le paramètre à la fonction et à retourner la somme.

Vous devez :

- relever grâce au débogueur, la valeur de r0 avant et après l'appel de "calcul_somme".
- préciser la méthode pour conserver le nombre de fois que calcul_somme est appelé?
- tester le programme modifié avec le débogueur en mode pas à pas afin de vérifier son bon fonctionnement.
- relever l'évolution des contenus de PC, RL et SP et de la mémoire à l'adresse pointée par SP.

Remarque : pour la suite, dans chaque sous-programme, vous veillerez à sauvegarder tous les registres utilisés si nécessaire. ARM donne des normes CMSIS (Cortex Microcontroller Software Interface Standard) pour passer les paramètres à des fonctions et la sauvegarde de registres. Nous n'étudierons pas ces normes dans ce TP, mais pour un programme dans le milieu professionnel, vous devrez les respecter.

Utilisation de la pile :

Les valeurs des registres ainsi que le contenu de la pile correspondent à l'état **avant** l'exécution de l'instruction.
Seules les données dans la pile aux adresses supérieures au Stack Pointer concernent votre programme.

Program Counter PC	Stack Pointer SP	PILE Métre adresse pile								Reg R0	Reg R1	Condition Flag N Z C	INSTRUCTION	OBSERVATIONS

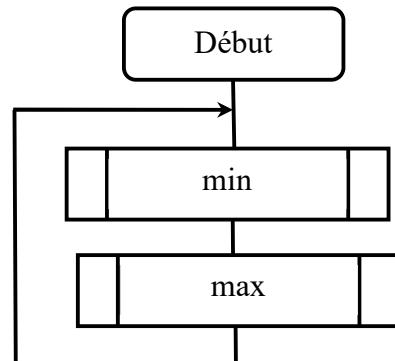
3.6. Utilisation des tableaux

Dans les langages évolués, les tableaux sont très utilisés pour résoudre de nombreux problèmes. Nous allons donc voir comment utiliser les tableaux en assembleur avec la famille STM32.

Cet exercice consiste à réaliser un programme d'extraction de minimum et maximum sur un tableau d'octets non signés. Pour cela, vous allez déclarer un tableau de 8 mots, comme ci-dessous.

tableau[0]	25
tableau[1]	4
tableau[2]	2
tableau[3]	15
tableau[4]	16
tableau[5]	101
tableau[6]	33
tableau[7]	3

Vous créerez alors deux sous-programmes "min" et "max" qui recherchent respectivement la valeur minimale et la valeur maximale dans le tableau. Les résultats seront stockés dans les variables "minimum" et "maximum" à déclarer.



Vous devez :

- Détalier l'organigramme des sous-programmes "min" et "max"

Lorsque l'on crée un programme dans un nouveau langage, il est intéressant d'intégrer au fur et à mesure les sous-programmes.

- Ecrire le sous-programme "min".
- Tester le programme partiel.

- Ecrire le sous-programme "max" en s'inspirant du sous-programme "min".
- Tester le programme complet.
- Option : Créer un seul sous-programme qui retourne le min et le max

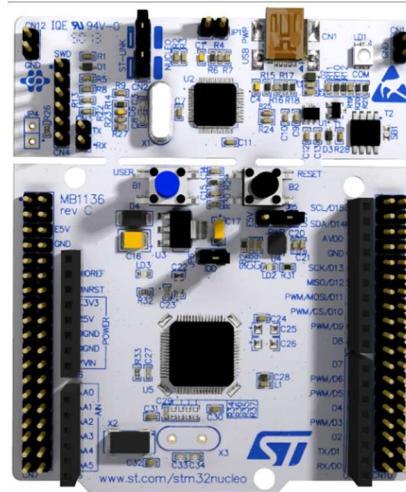
Remarques :

- *Tester un programme implique de tester toutes les branches du programme et des sous-programmes.*
- *Penser à utiliser toutes les fenêtres de débogage offertes par le STM32CubeIDE*

Chapitre La4. carte de développement

Maintenant que vous avez réalisé les exercices précédents, vous avez acquis les bases de la programmation en assembleur sur la famille STM32, et vous êtes capable de créer un programme et de le déboguer avec l'environnement de développement de chez STMicroelectronics STM32CubeIDE.

Nous allons à présent vous présenter brièvement la carte de développement NUCLEO-L152 développée par STMicroelectronics afin de tester les fonctionnalités d'un microcontrôleur.

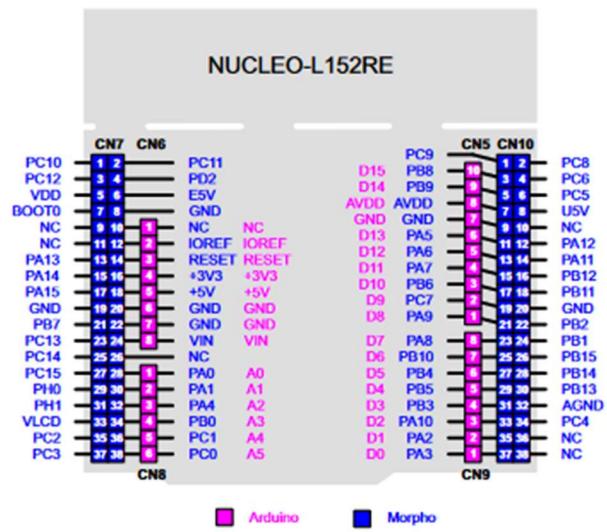
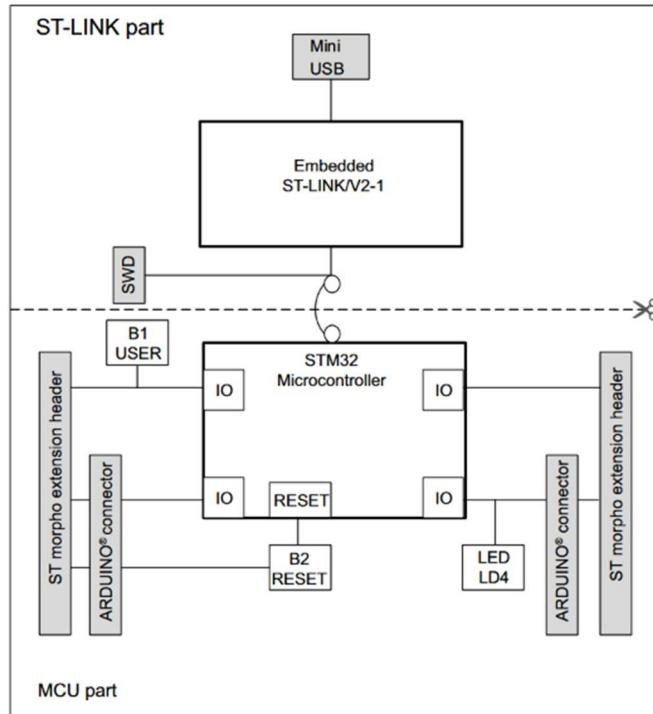


Cette carte vous servira aussi pour le projet et pour le cours de FreeRTOS. Elle permet d'utiliser les différents composants et fonctionnalités du µcontrôleur.

4.1. Périphériques disponibles sur la carte

Les périphériques disponibles sur la carte sont :

- 1 LED qui s'allume avec un état "1", connectée à la broche PA5
 - 1 bouton poussoir qui donne un état '0' quand il est appuyé, connecté à la broche PC13.
 - 1 bouton poussoir de Reset,
 - Un connecteur Arduino et un connecteur Morpho qui donnent accès aux broches du microcontrôleur et permet de connecter facilement des composants externes.
 - Une connexion USB pour débogage et la programmation avec connecteur SWD



STMicroelectronics fournit le schéma de la carte ce qui permet de voir comment les broches du microcontrôleur sont connectées à la carte

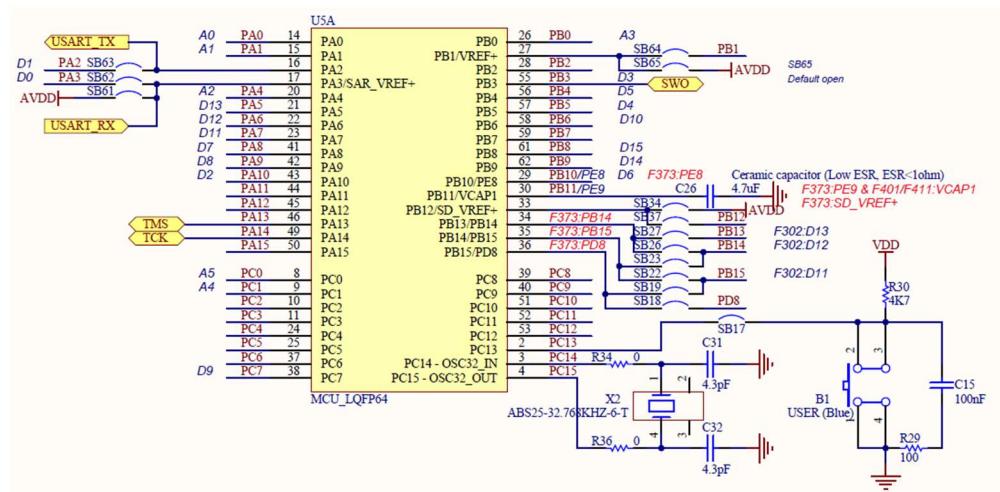


Figure : Connection STM32L152RE Nucleo board

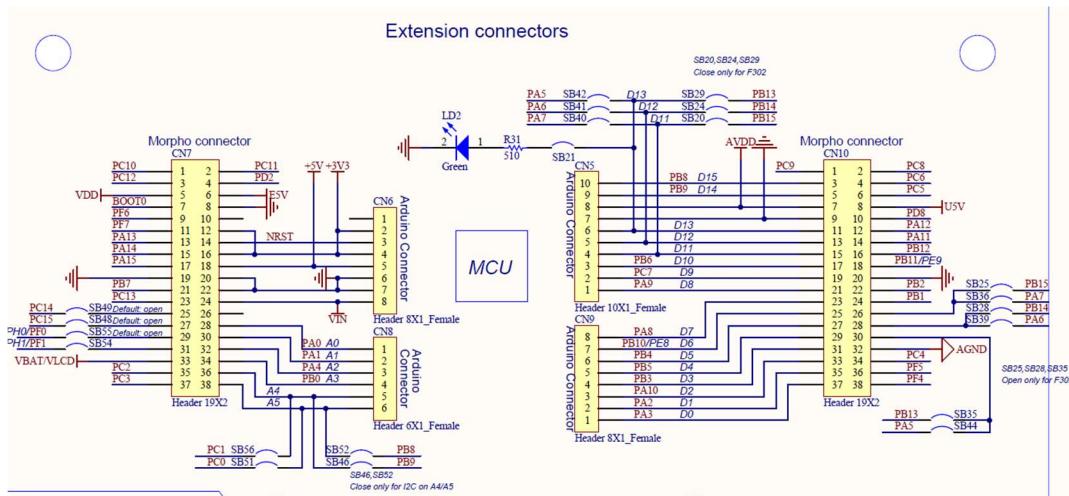
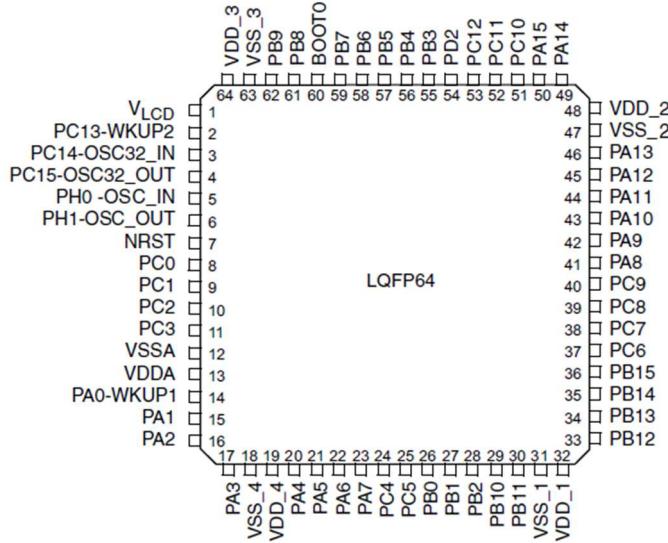


Figure : Connection STM32L152RE Nucleo board Connecteurs

La datasheet fournit également la vue du boîtier du composant avec les broches



Un tableau avec toutes les fonctionnalités possibles sur une broche est également disponible avec un exemple ci-dessous

LQFP144	Pins					Pin name	Pin Type ⁽¹⁾	I/O structure	Main function ⁽²⁾ (after reset)	Pin functions	
	UFBGA132	LQFP100	LQFP64	WLCSPI104						Alternate functions	Additional functions
41	K4	30	21	M8		PA5	I/O	TC	PA5	TIM2_CH1_ETR/ SPI1_SCK	ADC_IN5/ DAC_OUT2/ COMP1_INP
42	L4	31	22	H6		PA6	I/O	FT	PA6	TIM3_CH1/TIM10_CH1/S P1_MISO/ LCD_SEG3	ADC_IN6/ COMP1_INP/ OPAMP2_VINP
43	-	32	23	K7		PA7	I/O	FT	PA7	TIM3_CH2/TIM11_CH1/ SPI1_MOSI/ LCD_SEG4	ADC_IN7/ COMP1_INP/ OPAMP2_VINM
-	J5	-	-	-		PA7	I/O	FT	PA7	TIM3_CH2/TIM11_CH1/ SPI1_MOSI/ LCD_SEG4	ADC_IN7/ COMP1_INP

De plus amples informations sur la carte sont disponibles dans le document user manual UM1724.

4.2. Gestion des horloges

Le STM32L152 est un microcontrôleur 32-bit « low power », c'est-à-dire destiné à des applications peu gourmandes en énergie. Pour optimiser la consommation d'énergie d'un microcontrôleur, plusieurs moyens ont été mis en place et notamment la gestion de l'horloge selon le principe que plus l'horloge est rapide (High), plus le microcontrôleur consomme.

- Le premier moyen est un système de gestion d'horloge permettant de choisir la source de l'horloge et la fréquence à laquelle va travailler le microcontrôleur. La figure ci-dessous montre

que les sources de l'horloge sont internes HSI (High Speed Internal), MSI (Multi Speed Internal), LSI (Low Speed Internal) ou externes HSE (High Speed External) ou LSE (Low Speed External) selon que l'on utilise un quartz externe ou pas.

- Le deuxième moyen consiste à activer ou pas l'horloge des périphériques. En effet, si l'horloge d'un périphérique est désactivée alors il est inactif et consomme un minimum d'énergie. Lorsque tous les périphériques restent "connectés" à l'horloge, le µcontrôleur consomme beaucoup plus que si on connecte juste les périphériques réellement utilisés.

Il existe donc un périphérique "(RCC) Reset and Clock Control" entièrement dédié à la gestion de l'horloge vers les périphériques.

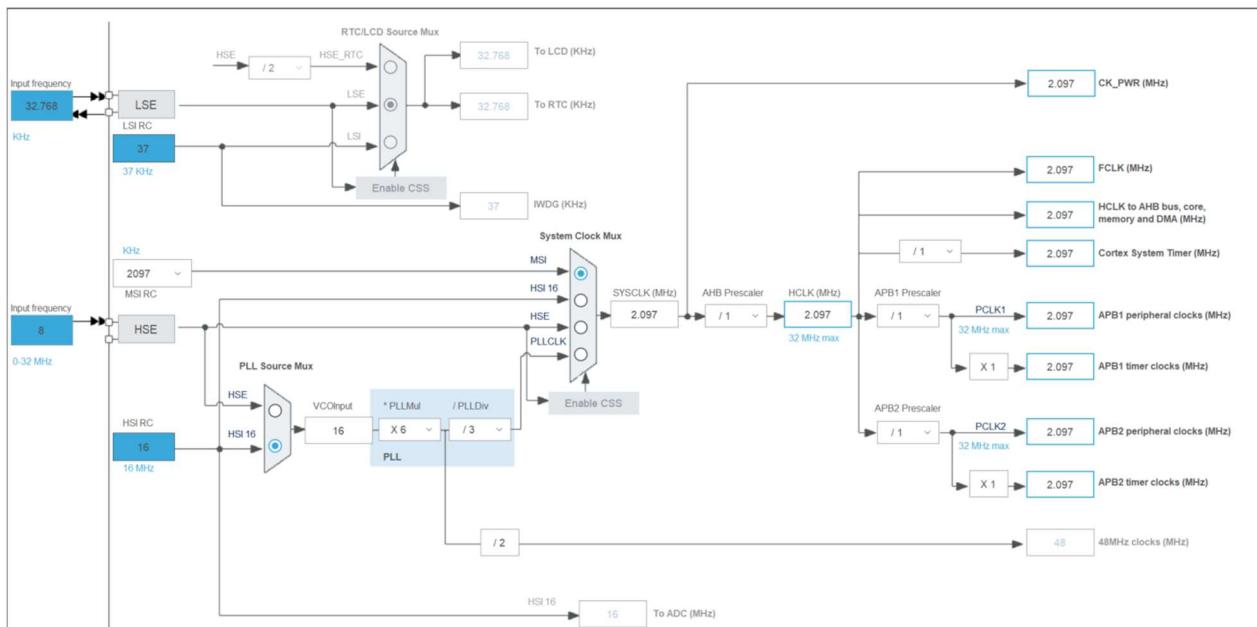
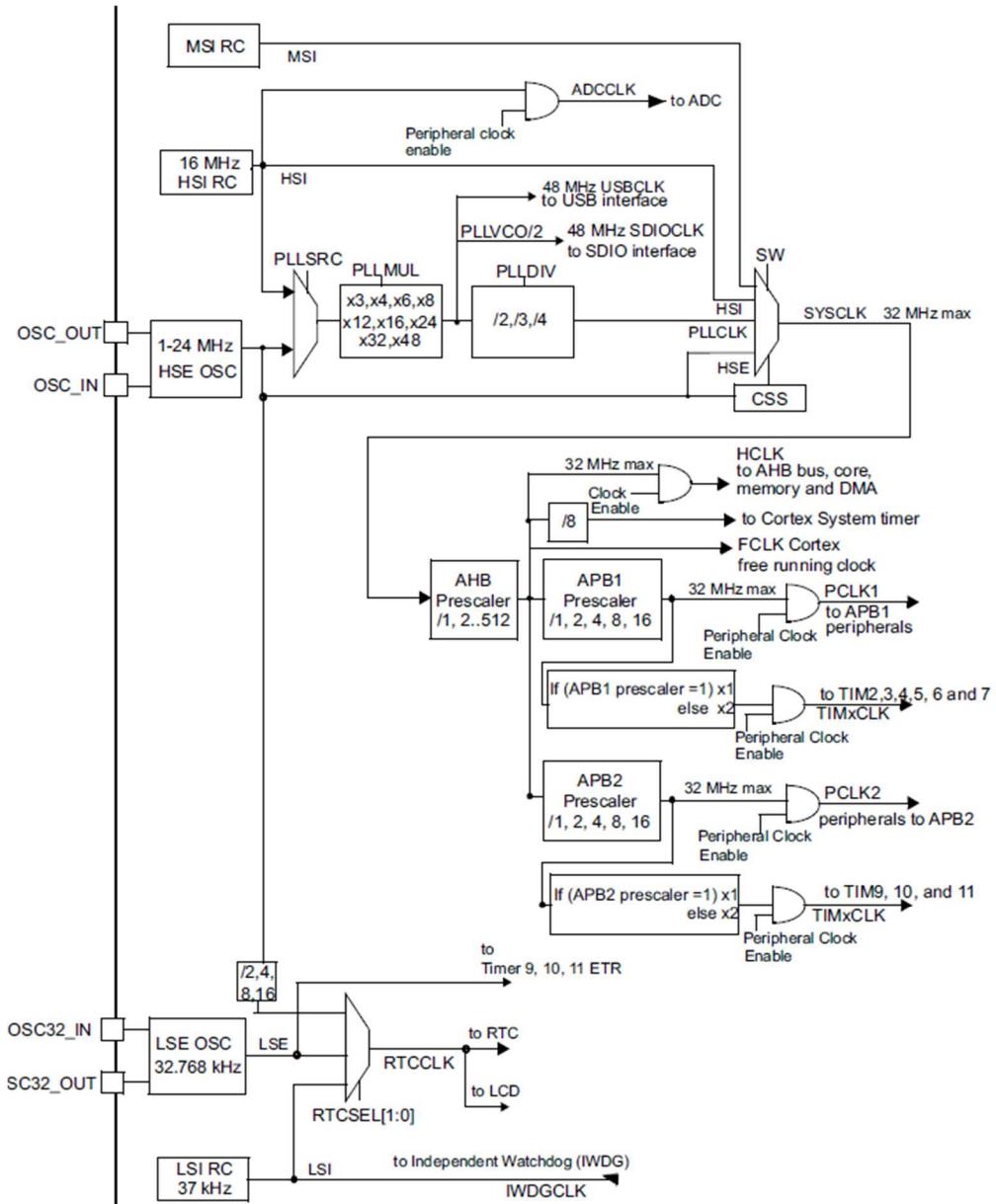


Figure 2 : schéma bloc de l'arbre des horloges

La configuration utilisée lors des TP sera la configuration par défaut au reset : **MSI on**. La fréquence fCPU est alors de 2,097 MHz. Pour limiter la consommation, toutes les horloges des périphériques ont été désactivées. **Il vous sera demandé d'activer l'horloge au niveau des périphériques lorsque cela sera nécessaire.**

Les périphériques sont connectés via les bus internes AHB, APB1 ou APB2. Vous devrez donc allumer et fournir une horloge aux périphériques avant d'y avoir accès.



Pour cela, vous devrez lire et comprendre les fonctions des registres du périphérique RCC.

Chapitre Les5. entrées-sorties

Un µcontrôleur possède des ports d'entrées/sorties (E/S ou GPIO) afin de dialoguer avec le monde extérieur.

Chaque E/S (GPIO) d'un port d'entrées/sorties peut être configurée en entrée pour tester l'état d'un signal logique ou configurée en sortie pour générer un signal logique.

Dans ce chapitre, nous allons utiliser les ports d'entrées/sorties du µcontrôleur STM32 pour commander l'allumage ou l'extinction de LEDs.

5.1. Principe

Les broches sont regroupées en port et un port E/S peut comporter jusqu'à 16 E/S. Chaque E/S peut être configurée de manière indépendante en allant simplement agir sur le bit des registres où elle est affectée.

Le STM32 dispose de 10 registres différents pour chaque port afin d'interagir avec les E/S. Le nom et la fonction des registres sont donnés dans le tableau suivant.

MODER: Mode Register (input; output, alternate function, analog)
OTYPER: Output Type Register (Output Speed Register)
PUPDR: Pullup / Pull-down register (if pin configured as open drain)
IDR: Input Data Register
ODR: Output Data Register
BSSR: Bit Set / Reset Register
LCKR: Lock Register
AFRL/H: Alternate Function Register (connect pin to timers, bus, event)
BRR: Bit Reset Register (reset ODR registers)

Le registre de direction **GPIOx_ODR** contrôle la direction de chaque bit n du port x.

GPIOx_ODR Etat bit n	0	1
	Port x bit n	Broche à 0V

Comme un port peut contrôler jusqu'à 16 broches, le registre 32-bit ODR a 16 bits pour contrôler le niveau de tension des broches et 16 bits qui sont sans fonction (reserved)

7.4.6 GPIO port output data register (**GPIOx_ODR**) (x = A..H)

Address offset: 0x14

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
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..H).

5.2. Initialisation des ports entrées/sorties

Tous les programmes qui utilisent les ports d'entrées/sorties doivent comporter un sous-programme d'initialisation des registres contrôlant les ports du contrôleur. Ce sous-programme d'initialisation placé au début du programme configure les ports E/S dont la valeur par défaut ne correspond aux besoins.

Vous devrez donc écrire un sous-programme "init_ports" qui réalise l'initialisation des ports dans le mode demandé.

Lorsque vous utilisez les registres associés aux ports E/S, vous devez conserver l'état des bits que vous ne modifiez pas, d'où le besoin de masques.

5.3. Caractéristiques détaillées

Les détails sur les modes de fonctionnement des entrées/sorties sont donnés dans le manuel de référence de la famille STM32L152 Reference Manual RM0038 dans la section General Purpose I/Os.

Le schéma bloc suivant détaille le contrôle et la structure d'une entrée/sortie :

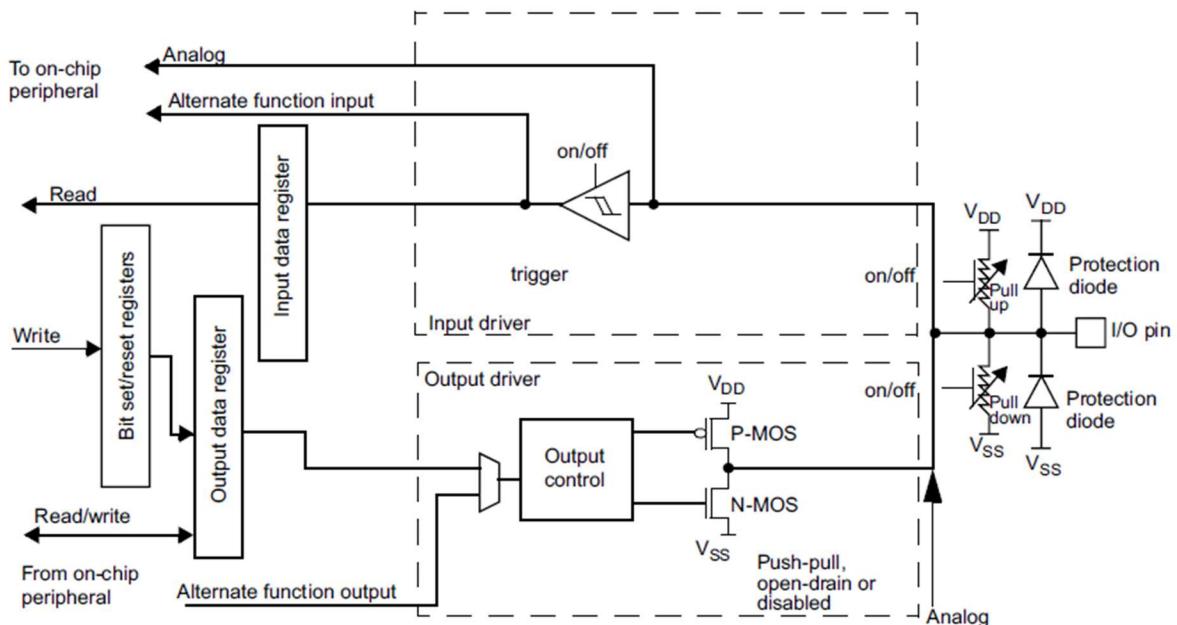


Figure 3 : Schéma bloc d'une E/S

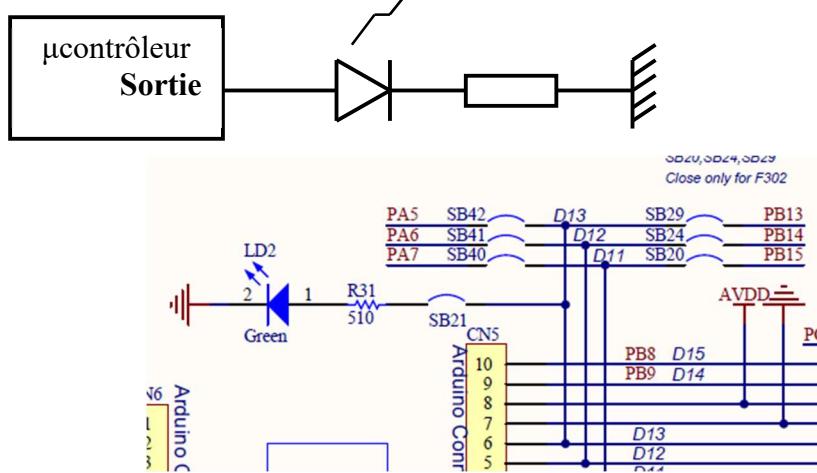
La broche peut être configurée en entrée/sortie, en push-pull ou Open Drain, utilisée comme en analogue ou numérique, connectée à un autre périphérique (timer, CAN,...)

Ces nombreuses possibilités expliquent le nombre élevé de registres pour configurer une broche.

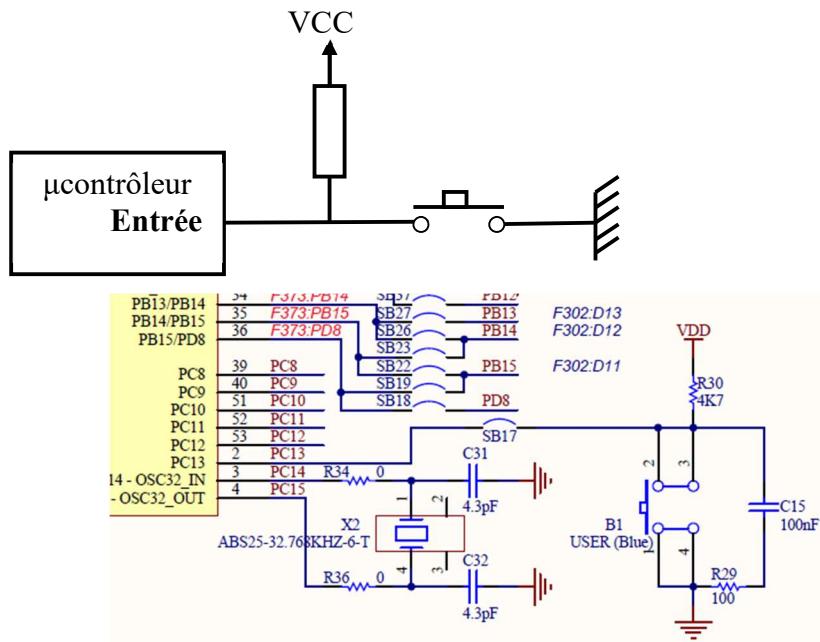
5.4. Commande d'une LED par un bouton poussoir

Cet exercice consiste à allumer la LED lorsque le bouton poussoir est appuyé.

La LED est commandée par le bit 5 du port A en sortie *push-pull* comme l'indique la figure suivante :



L'état du bouton poussoir 1 est vu par le bit 13 du port C (PC13) en entrée comme l'indique la figure suivante :



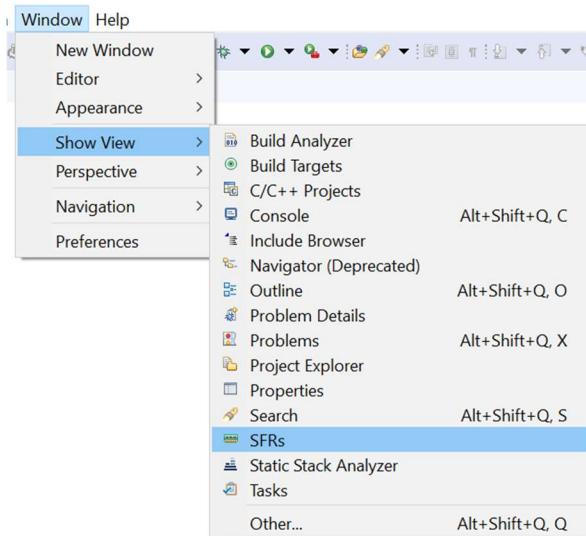
Vous devez :

- Écrire un sous-programme "init_ports_LED" afin d'initialiser le port A pour commander la LED.
- Écrire un sous-programme "init_ports_BP" afin d'initialiser le port C pour acquérir l'état bouton poussoir.

- Écrire un sous-programme "allume_LED" qui allume la LED voulue sans allumer les autres.
- Écrire un sous-programme "eteint_LED" qui éteint la LED voulue.
- Écrire un sous-programme "etat_poussoir" qui lit le port C et indique l'état du poussoir dans la variable "poussoir".
- Écrire votre programme principal qui appelle "init_ports_LED" et "init_ports_BP" puis boucle à l'infini sur :
 1. Appel "etat_poussoir"
 2. Appel de "allume_LED" ou "eteint_LED" selon la variable "poussoir"
- Tester votre programme afin de vérifier son bon fonctionnement.

Remarque : Si vous utilisez une fonction du style bl<c> avec un message d'erreur ‘‘Error: thumb conditional instruction should be in IT block’’, codez la fonction IT<c> avant. Demandez de l'aide à votre enseignant si nécessaire.

Remarque : la configuration et l'état d'un port peut être visualiser avec le débogueur en mode pas à pas en affichant la fenêtre SFR grâce au menu "Show View".



Il est également possible de modifier la valeur d'un registre directement dans cette fenêtre en cliquant directement sur les bits

Register	Address	Value
> OTYPER	0x40020...	0xa8000000
> OSPEEDER	0x40020...	0xa8000000
> PUPDR	0x40020...	0xa8000000
> IDR	0x40020...	0xa8000000
> ODR	0x40020...	0xa8000000
ODR15	[15:1]	0x0
ODR14	[14:1]	0x0
ODR13	[13:1]	0x0
ODR12	[12:1]	0x0
ODR11	[11:1]	0x0
ODR10	[10:1]	0x0
ODR9	[9:1]	0x0
ODR8	[8:1]	0x0
ODR7	[7:1]	0x0
ODR6	[6:1]	0x0
ODR5	[5:1]	0x0
ODR4	[4:1]	0x0
ODR3	[3:1]	0x0
ODR2	[2:1]	0x0
ODR1	[1:1]	0x0
ODR0	[0:1]	0x0
> BSRR	0x40020...	
> LCKR	0x40020...	0xa8000000
> AFRL	0x40020...	0xa8000000

5.5. Clignotement d'une LED

Cet exercice consiste à allumer la LED n°1, à attendre 0,5 seconde, puis à l'éteindre, à attendre à nouveau 0,5 seconde et à recommencer.

L'attente entre les différentes phases est réalisée par un sous-programme de temporisation.

Un sous-programme de temporisation est un sous-programme qui exécute des opérations "inutiles" pendant le temps désiré, soit par des boucles (parfois imbriquées) soit par d'autres méthodes. La durée de temporisation se trouve en calculant le nombre de cycles du sous-programme (la documentation des instructions donne le nombre de cycles de chaque instruction).

Vous devez :

- Calculer le nombre de cycles de votre sous-programme de temporisation de 0,5 seconde sachant que la fréquence fCPU est de 2,097 MHz
 - Écrire un sous-programme "tempo_500ms" qui effectue la temporisation voulue.
 - Écrire votre programme principal qui doit appeler "init_ports_LED", puis successivement "allume_LED", "tempo_500ms", "eteint_LED", "tempo_500ms" et boucler sur "allume_LED" pour recommencer.
 - Tester votre programme
 - **Vérifier la durée de votre temporisation en chronométrant la durée de 30 clignotements.**

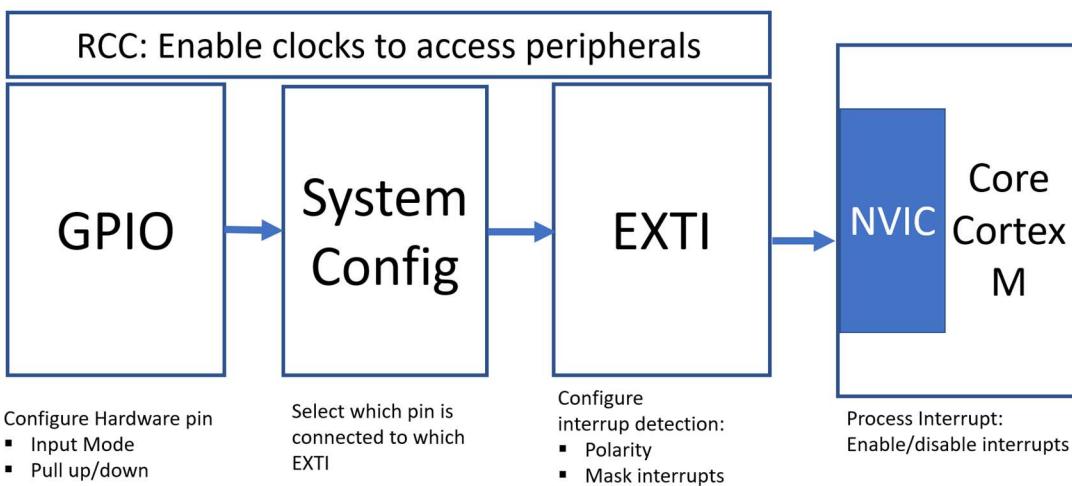
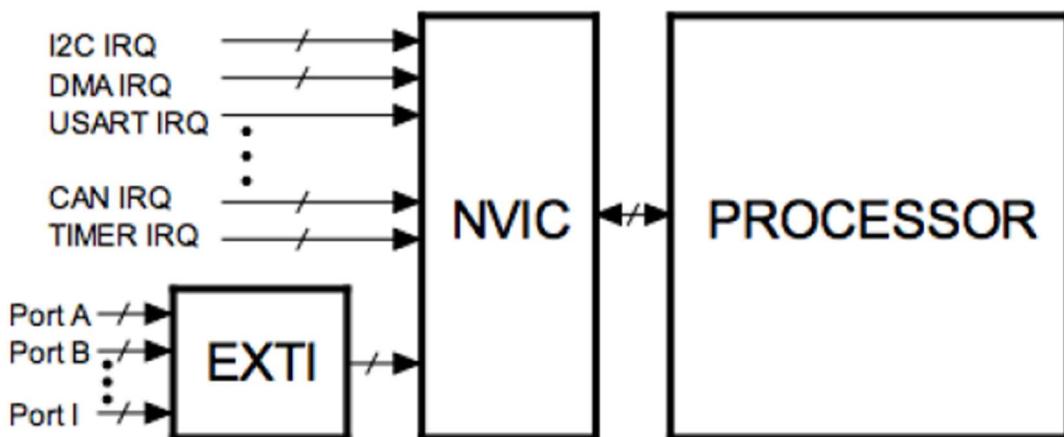
Chapitre Les6. interruptions

6.1. Principe de gestion des interruptions externes

Afin de configurer une interruption sur une broche avec le STM32, 5 blocks sont impliqués

- ✓ RCC : allumer les horloges pour pouvoir écrire sur les périphériques
- ✓ GPIO pour mettre la broche en entrée
- ✓ System Config pour savoir quelle broche est configurée à quelle "external interrupt"
- ✓ EXTI pour gérer la polarité : front montant, descendant ou les deux
- ✓ NVIC (Nested Vectored Interrupt Controller) appartient au Cœur du Cortex M3 et gère les interruptions (allumée ou éteinte, priorité...)

Vous devrez donc configurer chaque block correctement. Si un block n'est pas configuré convenablement, l'interruption ne fonctionne pas bien que programme compile



Les vecteurs d'interruption sont déclarés à la fin du fichier de start-up. La première interruption (Reset) est à l'adresse 0x0000 0004 (voir Figure 4).

Chaque interruption est identifiée par une adresse, un numéro et un nom. Le nom est fixé par ARM afin de garantir la portabilité des programmes entre les différents vendeurs de silicium.

Position	Priority	Type of priority	Acronym	Description	Address
-	3	settable	SVC_Handler	System service call via SWI instruction	0x0000_002C
-	4	settable	DebugMon_Handler	Debug Monitor	0x0000_0030
-	-	-	-	Reserved	0x0000_0034
-	5	settable	PendSV_Handler	Pendable request for system service	0x0000_0038
-	6	settable	SysTick_Handler	System tick timer	0x0000_003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000_0040
1	8	settable	PVD	PVD through EXTI Line detection interrupt	0x0000_0044
2	9	settable	TAMPER_STAMP	Tamper andTimeStamp through EXTI line interrupts	0x0000_0048
3	10	settable	RTC_WKUP	RTC Wakeup through EXTI line interrupt	0x0000_004C
4	11	settable	FLASH	Flash global interrupt	0x0000_0050
5	12	settable	RCC	RCC global interrupt	0x0000_0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000_0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000_005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000_0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000_0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000_0068

En dessous des vecteurs d'interruption, les fonctions appelée en cas d'interruption sont déjà définies. Remplacer Default_Handler par le nom de la fonction que vous appellerez.

```

594
595     .weak    EXTI0_IRQHandler
596     .thumb_set EXTI0_IRQHandler,Default_Handler
597
598     .weak    EXTI1_IRQHandler
599     .thumb_set EXTI1_IRQHandler,Default_Handler
600
601     .weak    EXTI2_IRQHandler
602     .thumb_set EXTI2_IRQHandler,Default_Handler
603
604     .weak    EXTI3_IRQHandler
605     .thumb_set EXTI3_IRQHandler,Default_Handler
606
607     .weak    EXTI4_IRQHandler
608     .thumb_set EXTI4_IRQHandler,Default_Handler
609

```

6.2. Clignotant avec marche/arrêt sous interruption

Cet exercice consiste à reprendre l'exercice "clignotement d'une LED" du chapitre 5 et à y ajouter la gestion de l'appui sur le bouton poussoir bleu connecté à la broche PC13 grâce aux interruptions :

- L'appui sur le bouton poussoir démarre le clignotement de la LED en commençant par la LED allumée durant 0,5s.
- Un nouvel appui sur le bouton poussoir arrête le clignotement de la LED (LED éteinte après 0,5s au maximum).

Le changement d'état sera dès l'appui sur le bouton et non quand le bouton est relâché. Vous devez donc étudier le schéma et vérifier comment est conçu ce bouton pour décider de la polarité à configurer.

Vous devez :

- Créer une variable "clignotant" qui va indiquer l'état du clignotant : marche ou arrêt.
- Initialiser le STM32 pour que PC13 soit configurée pour l'interruption et PA5 pour gérer la LED. Vous créerez 2 sous-programmes init_led et init_interrupt_ext .
- Créer dans la zone de déclaration des sous-programmes d'interruption, "marche_arret_interrupt" qui sera appelé à chaque appui sur PC13..
- Tester le programme afin de vérifier son bon fonctionnement.
- Relever l'évolution des registres PC, SP, LR et xPSR à l'entrée et la sortie de l'interruption
- Identifier les informations sauvegardées dans la pile.

Entrée et sortie d'interruption :

Vérifier la valeur des registres R0, R1, R2, R3, SP, PC, LR et xPSR, avant pendant et en sortie d'interruption. Que se passe t'il automatiquement à chaque interruption ?

Annexe 1. Les instructions saut conditionnel

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) ^{ab}	Condition flags
0000	EQ	Equal	Equal	$Z == 1$
0001	NE	Not equal	Not equal, or unordered	$Z == 0$
0010	CS ^c	Carry set	Greater than, equal, or unordered	$C == 1$
0011	CC ^d	Carry clear	Less than	$C == 0$
0100	MI	Minus, negative	Less than	$N == 1$
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	$N == 0$
0110	VS	Overflow	Unordered	$V == 1$
0111	VC	No overflow	Not unordered	$V == 0$
1000	HI	Unsigned higher	Greater than, or unordered	$C == 1$ and $Z == 0$
1001	LS	Unsigned lower or same	Less than or equal	$C == 0$ or $Z == 1$
1010	GE	Signed greater than or equal	Greater than or equal	$N == V$
1011	LT	Signed less than	Less than, or unordered	$N != V$
1100	GT	Signed greater than	Greater than	$Z == 0$ and $N == V$
1101	LE	Signed less than or equal	Less than, equal, or unordered	$Z == 1$ or $N != V$
1110	None (AL) ^e	Always (unconditional)	Always (unconditional)	Any

Annexe 2. Créer un projet STM32

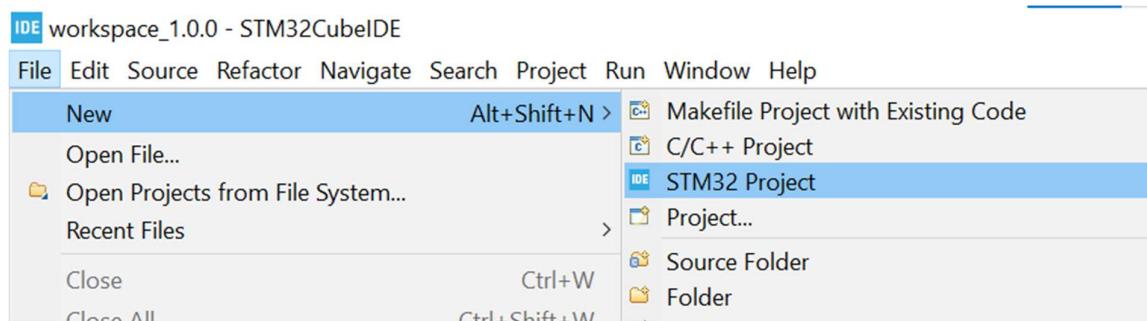
Le kit de développement de ST Microelectronics dédié aux microcontrôleurs des familles STM32 permet de créer très simplement des projets avec un environnement graphique.

Ce livret vous guidera pas à pas dans la création d'un tel projet.

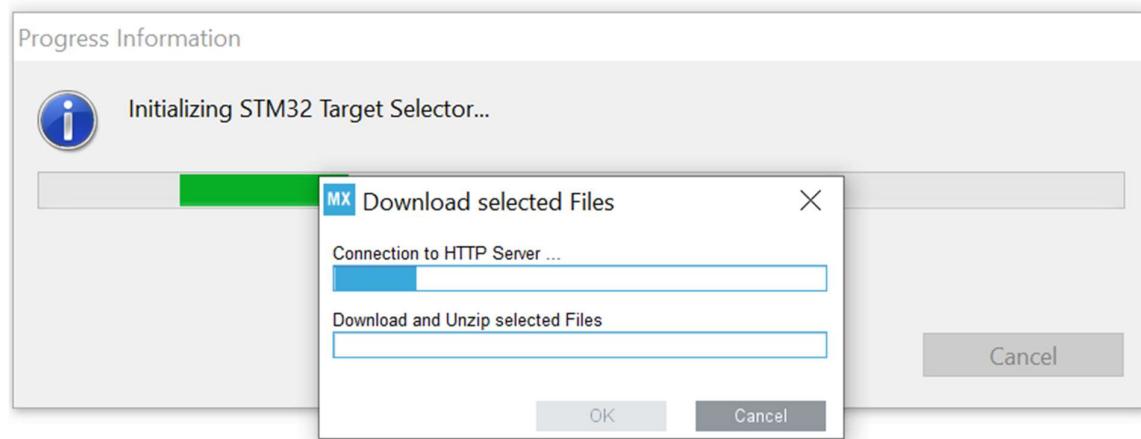
Chaque étape sera détaillée et illustrée de captures d'écran afin de faciliter votre lecture.

Après avoir lu ce petit guide, vous serez capable de créer votre premier projet et de le tester

Pour créer un nouveau projet, cliquer sur File/New/STM32 Project



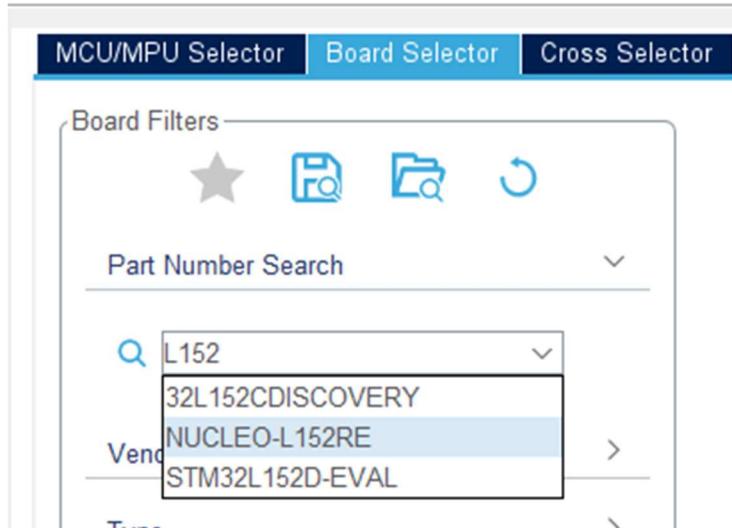
Si une connection à internet est disponible, STM32CubeIDE peut mettre à jour l'offre des produits disponible. Cette mise à jour peut prendre quelques secondes.



Sélectionner la tabulation Board Selector et dans la fenêtre de recherche Parr Number Research, sélectionner Nucleo-L152RE

Target Selection

Select STM32 target



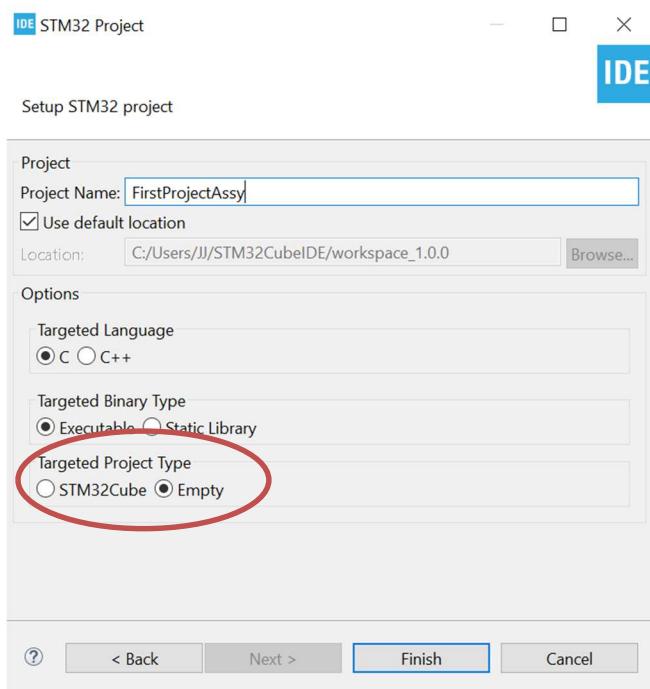
A droite de la fenêtre, la carte utilisée pour le TP apparaît.

Boards List: 1 item							Export
#	Overview	Part No	Type	Marketing Status	Unit Price (US\$)	M	M
1		NUCLEO-L152RE	Nucleo64	Active	13.0		STM32L152RETx

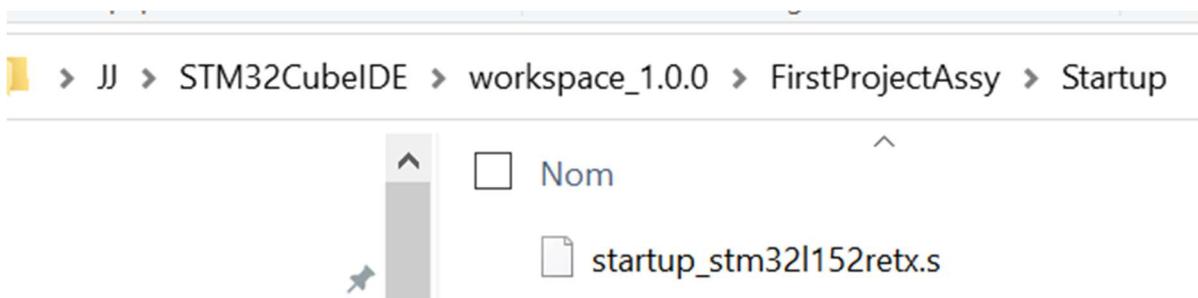
Entrer le nom du projet.

!!! Dans Targeted Project Type, sélectionner Empty !!!

Dans le cas contraire, des librairies inutiles aux TP en assembleur seront ajoutées au projet

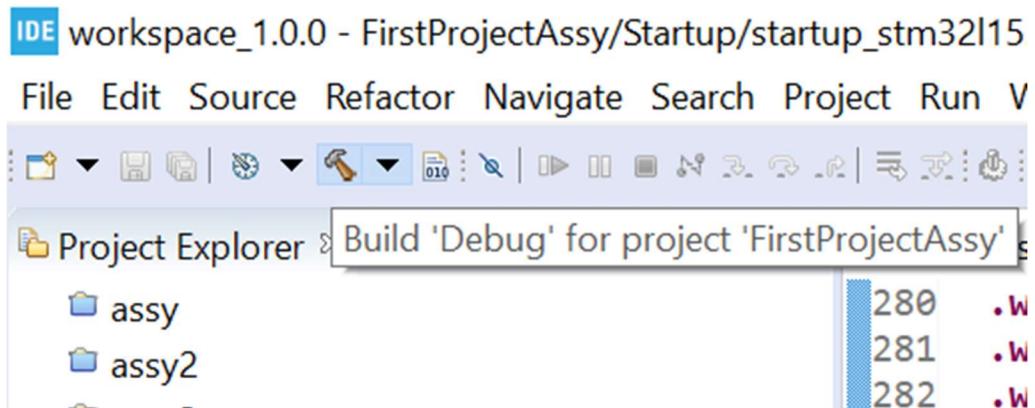


Avec Windows Explorer dans le workspace que vous avez créé et dans le dossier start up.



Attention : Remplacer le fichier startup_stm32l152retx.s par le fichier du même nom publié sur l'ENT.

Puis compiler votre programme



```
arm-none-eabi-size FirstProjectAssy.elf
arm-none-eabi-objdump -h -S FirstProjectAssy.elf > "FirstProjectAssy.list"
  text    data     bss   dec   hex filename
 536      8   1568   2112    840 FirstProjectAssy.elf
arm-none-eabi-objcopy -O binary FirstProjectAssy.elf "FirstProjectAssy.bin"
Finished building: default.size.stdout

Finished building: FirstProjectAssy.list

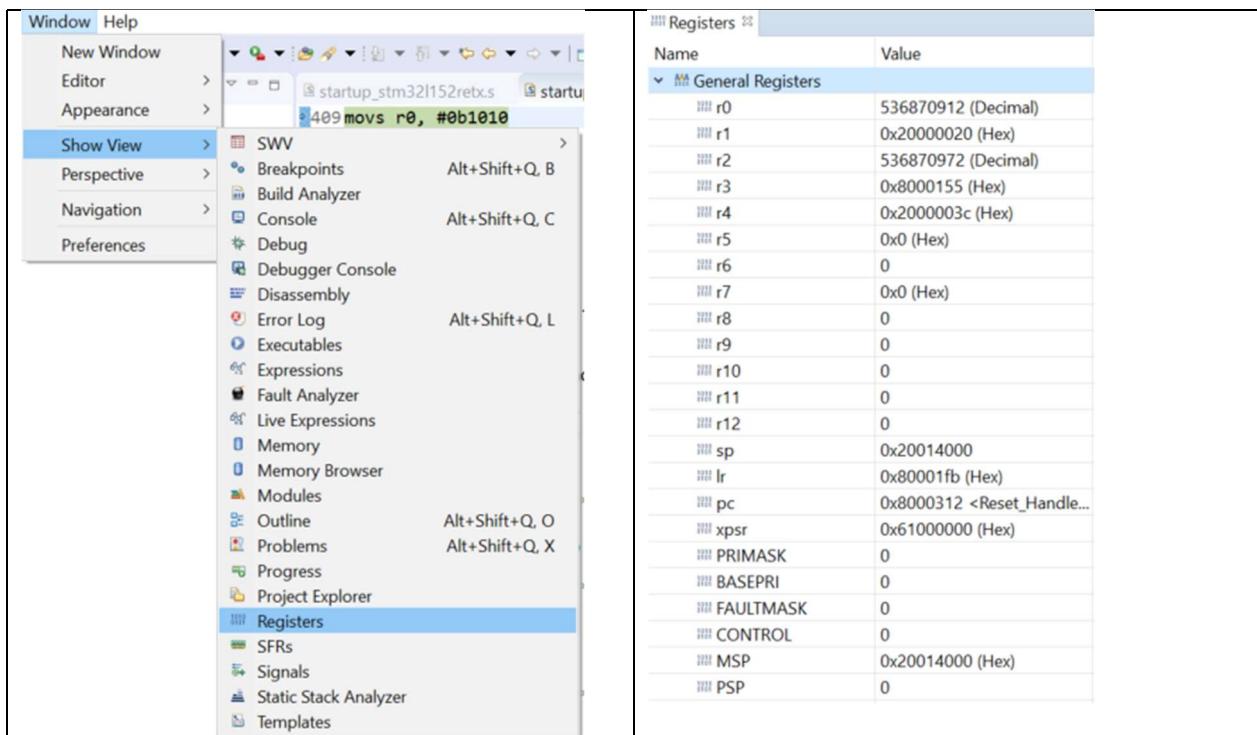
Finished building: FirstProjectAssy.bin

10:29:05 Build Finished. 0 errors, 0 warnings. (took 1s.491ms)
```

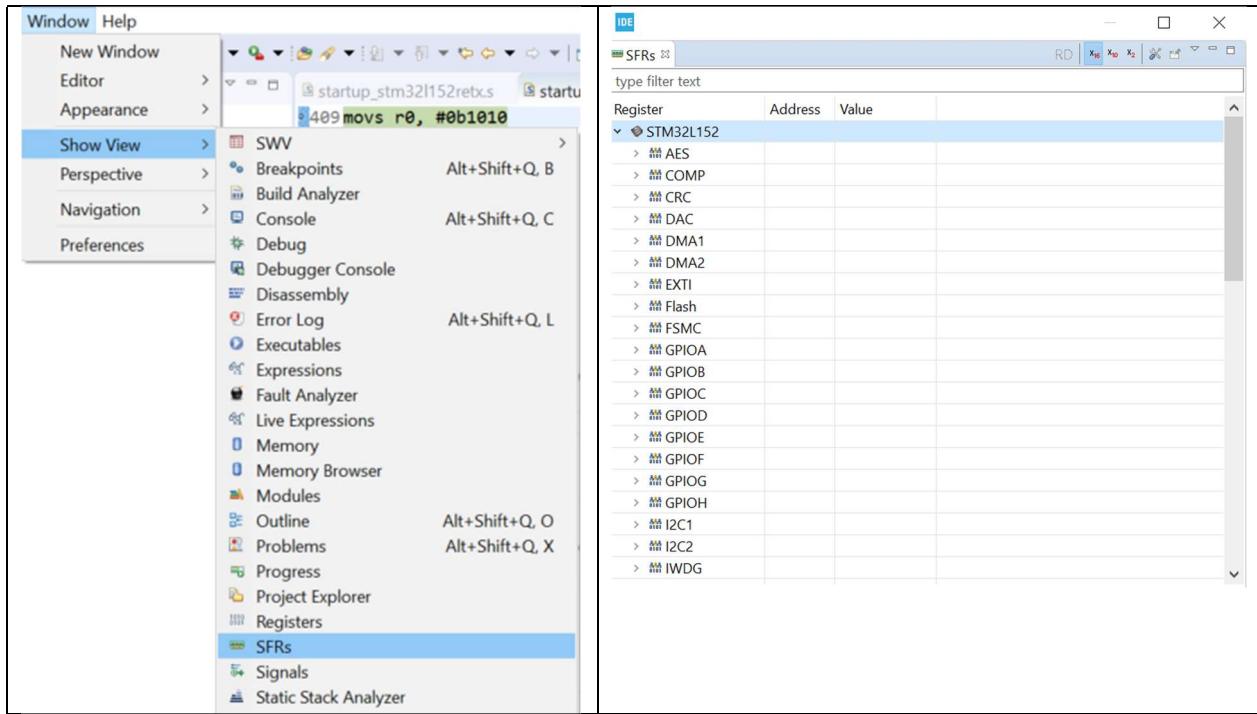
Vous êtes maintenant prêts à écrire vos programmes en assembleur. Les fenêtres présentées en annexe 3 vous seront nécessaires pour répondre aux questions du TP et déboguer votre programme.

Annexe 3. Fenêtre de débogage

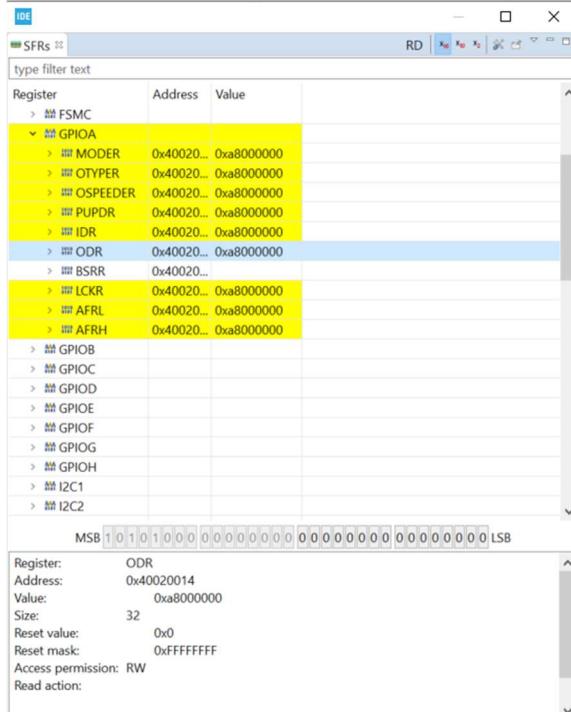
Fenêtre pour accéder aux registres internes



Fenêtre pour accéder aux périphériques



Et bien sûr, chaque détail d'un registre est accessible



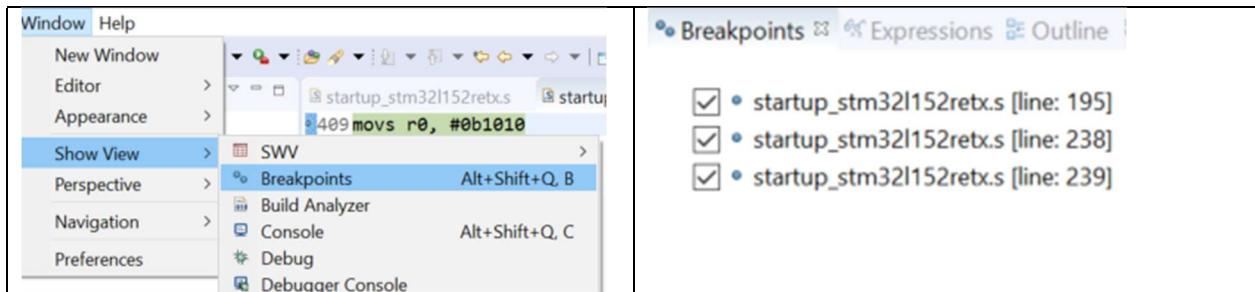
Fenêtre de désassemblage

```

08000304: 0x81ea0201 eor.w r1, r1, r2
392           str r1, [r0]
08000308: 0x000000160 str r1, [r0, #0]
395           ldr r0,=CPT
0800030a: 0x00001a48 ldr r0, [pc, #104]
396           ldr r0,[r0]
0800030c: 0x00000068 ldr r0, [r0, #0]
398           subs r0,r0,#1
0800030e: 0x000000138 subs r0. #1

```

Fenêtre pour accéder aux points d'arrêt



Pour créer un point d'arrêt, double cliquer en face de la ligne. Pour annuler un point d'arrêt, dans la fenêtre breakpoint, cliquer sur un point d'arrêt et appuyer sur la touche ‘suppr’.

Fenêtre pour accéder à une adresse mémoire

