

ELE3312  
Microcontrôleurs et applications  
Laboratoire 3

Auteur : Jean Pierre David

## Introduction

Le processeur au cœur du microcontrôleur peut seulement exécuter des instructions appartenant à son jeu d'instruction (*Instruction Set* en anglais) qu'on appelle parfois *instruction machine* ou encore *instruction assembleur*. Ces instructions sont en général très élémentaires : des opérations arithmétiques ou logiques sur des registres locaux, des branchements conditionnels et des échanges (lecture/écriture) avec la mémoire et les entrées/sorties. Le langage assembleur permet d'écrire directement des programmes utilisant ces instructions. Toutefois, quand c'est possible, on préfère utiliser des langages dits de plus haut niveau tels que C/C++. Les compilateurs s'étant considérablement améliorés ces dernières années, un code écrit en C/C++ et compilé automatiquement vers l'assembleur produit souvent un meilleur code que s'il est directement écrit en assembleur par le concepteur. Néanmoins, dans certains cas, on a besoin de comprendre très précisément ce que fait le processeur, instruction assembleur après instruction assembleur. Le but de ce laboratoire est de vous apprendre à écrire des méthodes en assembleur et à les appeler à partir d'un code C. Vous apprendrez aussi à suivre l'exécution d'un programme pas à pas et vous observerez comment les compilateurs produisent du code assembleur.

## Objectifs

1. Installer et configurer l'application Putty, un terminal qui permet (notamment) d'échanger des données par l'interface série RS232
2. Écrire des fonctions en assembleur et les appeler à partir d'un code C
3. Faire l'exécution pas à pas de codes C/assembleur
4. Comprendre la traduction du C vers l'assembleur réalisée par l'IDE Keil

## 1<sup>ère</sup> partie : préparation à la maison

### Mode opératoire

#### Installation de l'application Putty

1. Allez à l'adresse suivante : <https://www.putty.org/>
2. Téléchargez l'installateur (typiquement la version 64 bits sur les ordinateurs récents)
3. Installez le logiciel et acceptez toutes les options par défaut

#### Réaliser une copie du projet réalisé au laboratoire 2

1. Copier intégralement le répertoire créé par STM32Cube (*MyTest1* si vous avez pris le nom suggéré) dans un autre emplacement du disque
2. Effacer le répertoire MyTest1\Src
3. Avec STM32Cube, ouvrir le projet MyTest1.
4. Générez le code avec le menu Projet/Generate Code
5. Ouvrir le projet avec Keil

## Rediriger la fonction printf() vers l'UART

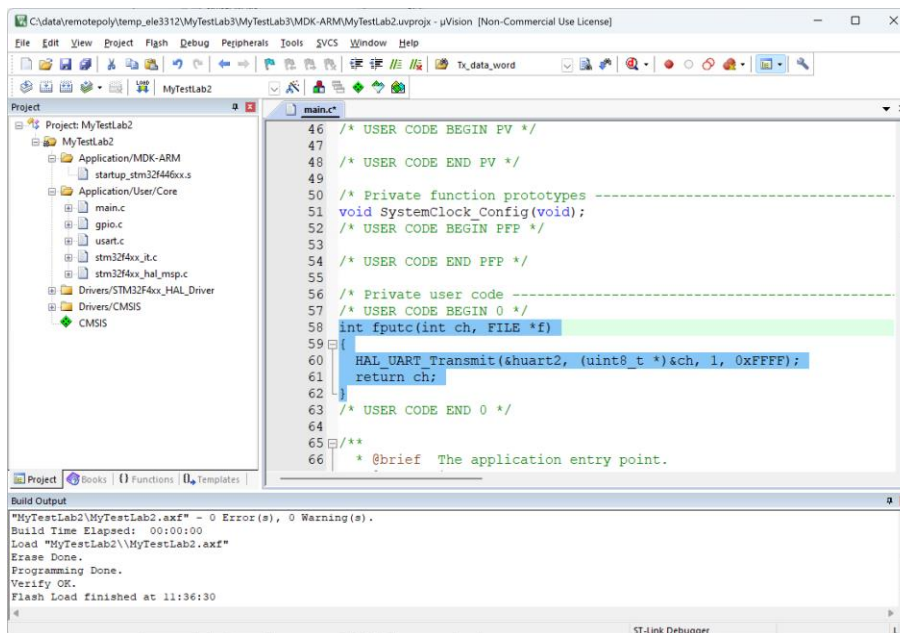
NOTE : Ces instructions proviennent du lien suivant :

<http://www.emcu.eu/how-to-implement-printf-for-send-message-via-usb-on-stm32-nucleo-boards/>

1. Dans le fichier main.c, localiser la mention « /\* USER CODE BEGIN Includes \*/ » et ajouter le code suivant :

```
18  /*
19  /* USER CODE END Header */
20
21  /* Includes -----*/
22  #include "main.h"
23  #include "usart.h"
24  #include "gpio.h"
25
26  /* Private includes -----*/
27  /* USER CODE BEGIN Includes */
28  #include "stdio.h"
29  /* USER CODE END Includes */
30
31  /* Private typedef -----*/
32  /* USER CODE BEGIN PTD */
33
34  /* USER CODE END PTD */
35
36  /* Private define -----*/
37  /* USER CODE BEGIN PD */
38  /* USER CODE END PD */
39
40  /* Private macro -----*/
41  /* USER CODE BEGIN PM */
42
```

2. Dans le fichier main.c, localiser la mention « /\* USER CODE BEGIN 0 \*/ » et ajouter le code suivant :



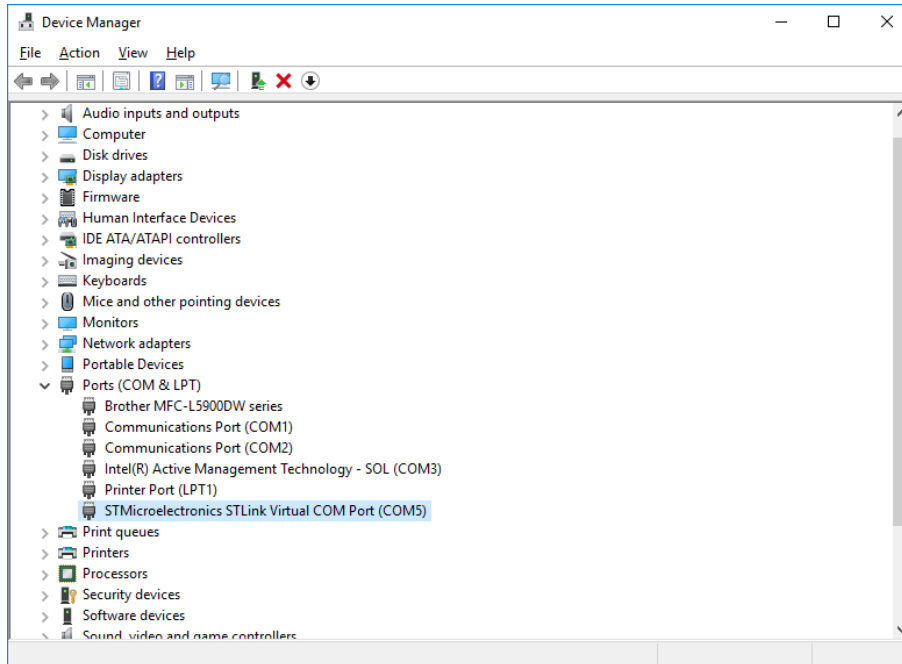
(Vous pouvez copier-coller le code ci-dessous)

```
int fputc(int ch, FILE *f)
{
    HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, 0xFFFF);
    return ch;
}
```

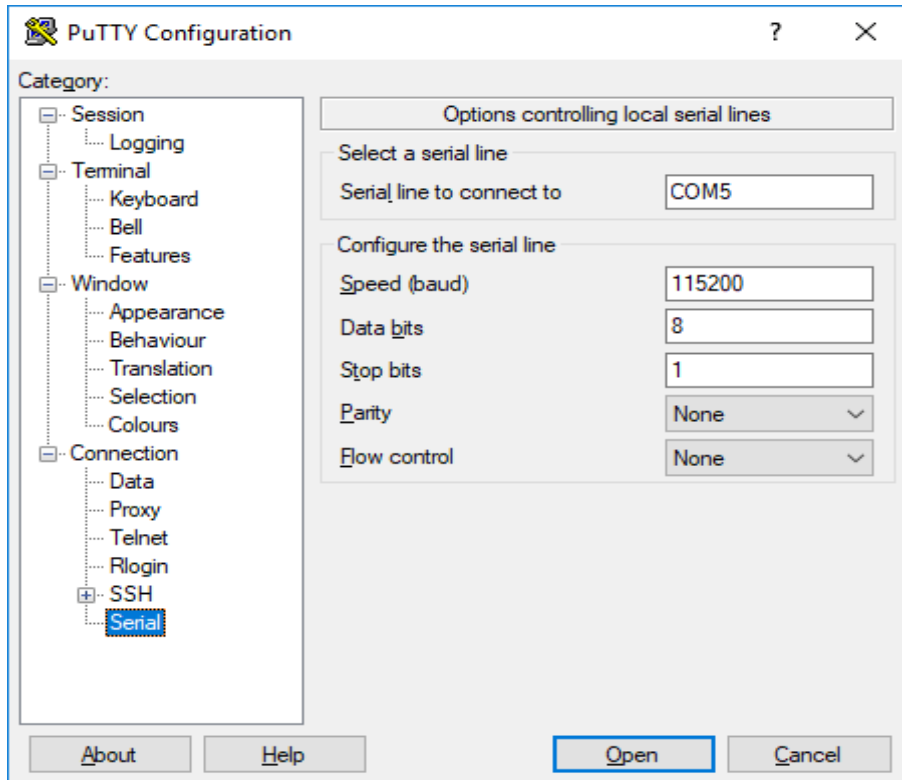
3. Dans Keil->Flash->Configure Flash tools...->Target, cocher « Use MicroLIB » ensuite recompiler tout le projet.

### Exécuter une application de test :

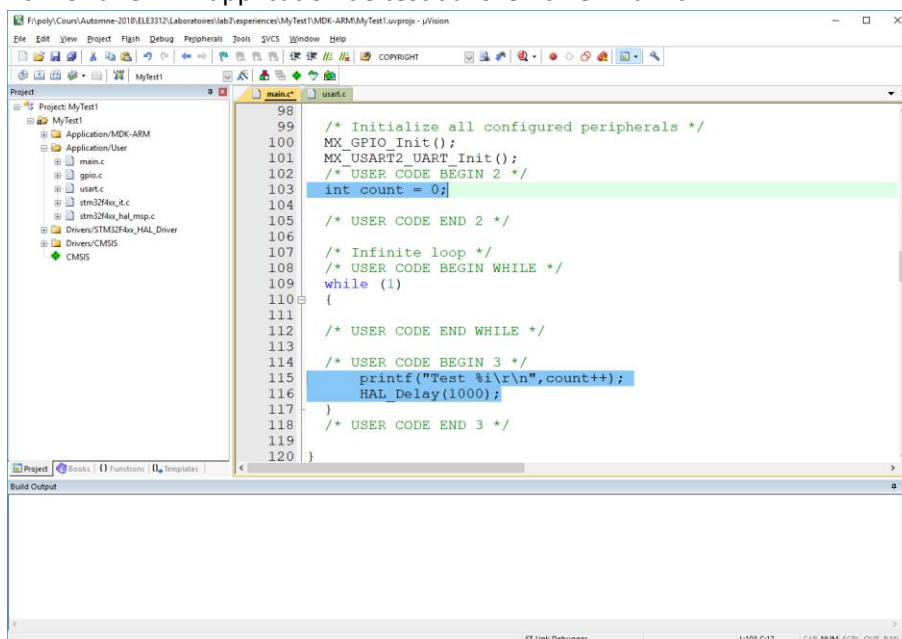
1. Connectez votre carte à l'ordinateur. Votre ordinateur devrait reconnaître un nouveau port RS232. Vous devez identifier son numéro dans le Gestionnaire de périphériques (Device manager). Dans l'exemple ci-dessous, il s'agit du port COM5 :



2. Ouvrir PUTTY et configurer le port série de cette manière (en utilisant votre numéro de port série évidemment) :

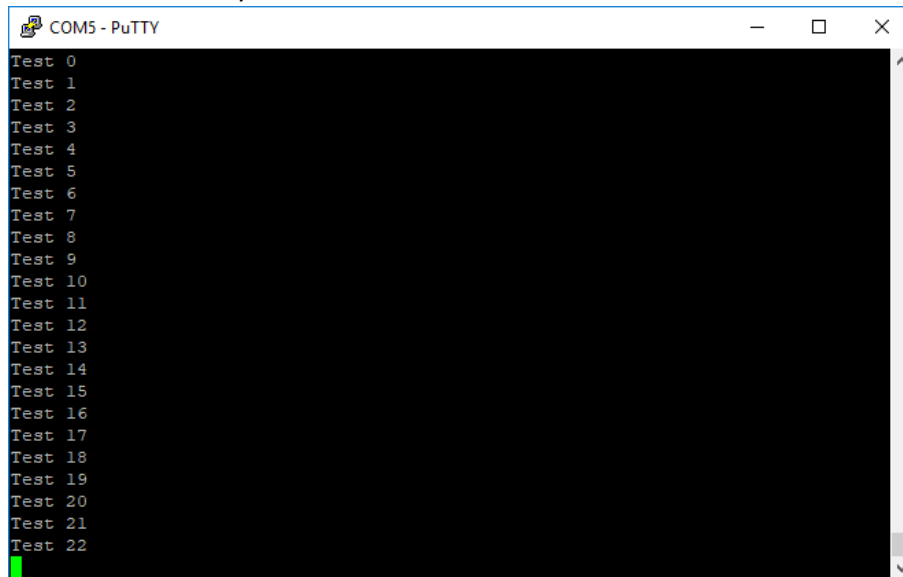


3. Revenez au menu Session, sélectionnez le mode « Serial » et ouvrez la session avec le bouton Open. À partir d'ici, tout ce que la carte enverra vers son port de sortie sera affiché sur ce terminal et tout ce que vous écrirez dans le terminal sera envoyé vers le port d'entrée de la carte.
4. Écrivez une mini application de test dans le fichier main.c :



5. Notez le « \r\n » nécessaire pour créer un saut de ligne en Windows.

6. Compilez (F7), téléversez (F8) et exécutez (bouton RESET noir de la carte).
7. À chaque seconde, vous devriez maintenant voir apparaître une nouvelle ligne « Test ... » dans votre terminal Putty :



```
COM5 - PuTTY
Test 0
Test 1
Test 2
Test 3
Test 4
Test 5
Test 6
Test 7
Test 8
Test 9
Test 10
Test 11
Test 12
Test 13
Test 14
Test 15
Test 16
Test 17
Test 18
Test 19
Test 20
Test 21
Test 22
```

## Écrire une méthode en assembleur

Lorsqu'une application mixe du code C/C++ avec du code assembleur, un certain nombre de conventions doivent être respectées. Il ne faudrait pas que le code assembleur vienne changer l'état du programme C/C++ de manière inopinée. Vous trouverez la description complète des conventions au lien suivant (Procedure Call Standard for the ARM Architecture) :

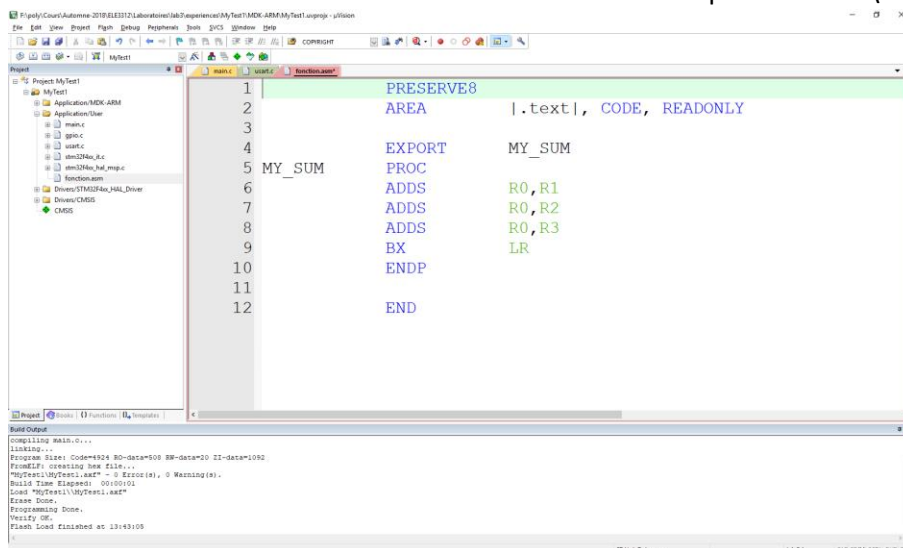
<https://github.com/ARM-software/abi-aa/releases>

Toutefois le document est difficile à comprendre quand on est débutant. Voici quelques règles à respecter qui vous éviteront bien des soucis :

1. Les arguments (32 bits) d'une méthode sont passés dans les registres r0, r1, r2 et r3 dans l'ordre. Si vous avez plus de 4 arguments, c'est plus compliqué 😊 mais cela se fait.
2. Le résultat (32 bits) d'une fonction est retourné dans le registre r0.
3. Une méthode peut modifier les registres suivants : r0-r8, r10-r11 mais elle doit restaurer l'état des registres r4-r8, r10-r11 avant de revenir au programme qui l'a appelée.
4. Une méthode MethA écrite en assembleur qui veut faire appel à une méthode MethC (en C) doit la déclarer dans le code assembleur au moyen de la directive IMPORT MethC. Il en est de même pour toutes les variables C qui devraient être utilisées par MethA.
5. Une méthode MethA écrite en assembleur qui doit être appelée par une méthode MethC (en C) doit se déclarer elle-même dans le code assembleur au moyen de la directive EXPORT MethA. Il en est de même pour toutes les variables assembleur qui devraient être utilisées par MethC.
6. Une méthode C MethC qui veut accéder à une méthode MethA (en assembleur) doit la déclarer (juste son prototype).
7. Une méthode C MethC qui veut accéder à une variable en assembleur doit la déclarer comme externe au moyen du mot clef « extern ».

Pour commencer, nous allons écrire une méthode qui calcule la somme de ses quatre arguments entiers signés :

1. Écrire le fichier assembleur fonction.asm suivant dans le répertoire Core\Src :



```
1 PRESERVE8
2 AREA |.text|, CODE, READONLY
3
4 EXPORT MY_SUM
5 MY_SUM
6 PROC
7 ADDS R0, R1
8 ADDS R0, R2
9 ADDS R0, R3
10 BX LR
11 ENDP
12 END
```

Build Output  
Compiling main.o...  
Linking...  
Program Size: Code=4924 RO-data=508 RW-data=20 ZI-data=1082  
FinalLink: creating new elf...  
MyTest:MyTest1.elf - 0 Error(s), 0 Warning(s).  
Build Time: Elapsed: 00:00:00  
Load MyTest1\MyTest1.elf  
Error Done.  
Programming Done.  
Verify OK.  
Flash Load Finished at 13:43:05

(Vous pouvez copier-coller le code ci-dessous)

```

PRESERVE8
AREA      |.text|, CODE, READONLY

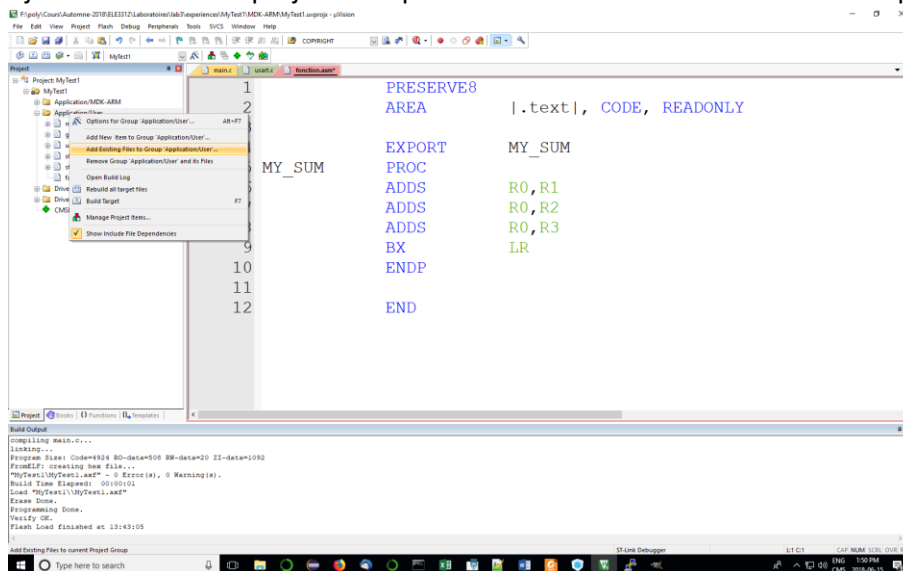
EXPORT      MY_SUM

MY_SUM
PROC
ADDS      R0,R1
ADDS      R0,R2
ADDS      R0,R3
BX          LR
ENDP

END

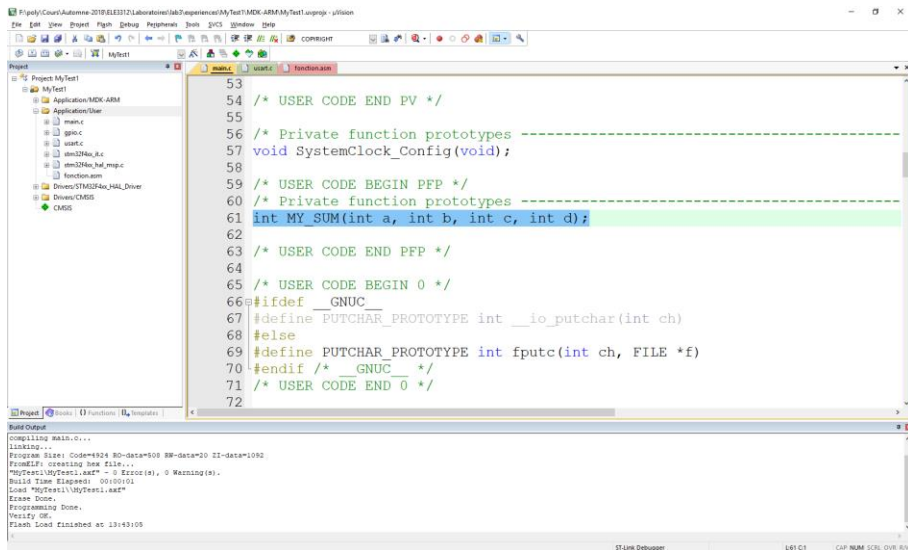
```

2. Ajoutez le fichier au projet en cliquant avec le bouton droit sur la section Application/User :

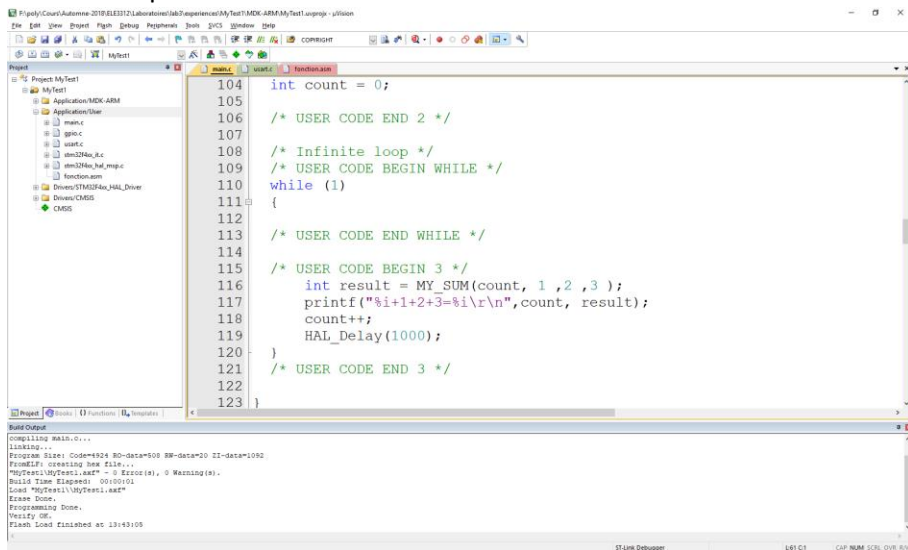


3. Vous remarquerez la commande EXPORT qui rend la fonction visible en C.
4. Ajoutez la déclaration de la fonction dans votre code C :





## 5. Écrivez une petite routine de test :



(Vous pouvez copier-coller le code ci-dessous)

```

/* USER CODE BEGIN 2 */
int count = 0;
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    int result = MY_SUM(1, 2, 3, count);

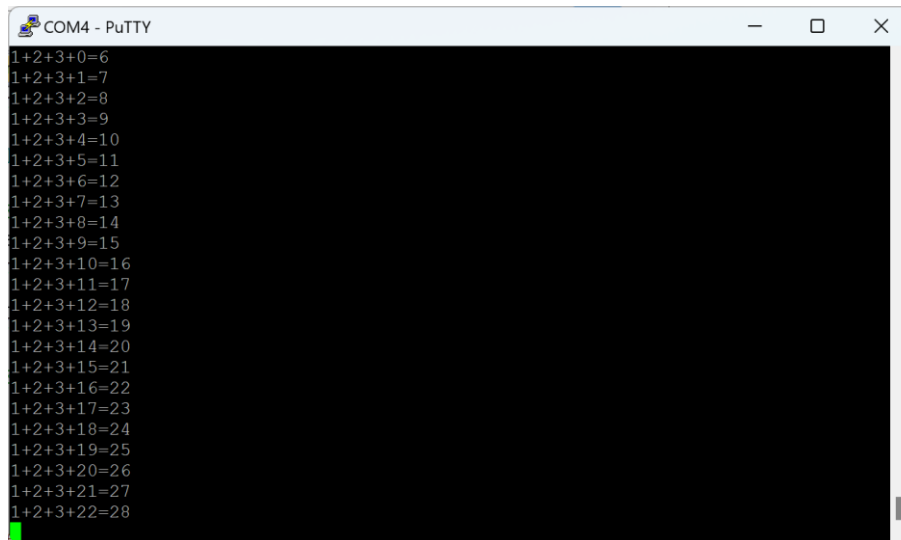
```

```

        printf("1+2+3+%i=%i\r\n",count, result);
        HAL_Delay(1000);
        count++;
    }
/* USER CODE END 3 */

```

6. Compilez (F7), téléversez (F8) et exécutez (bouton RESET noir sur la carte) et contemplez votre résultat :



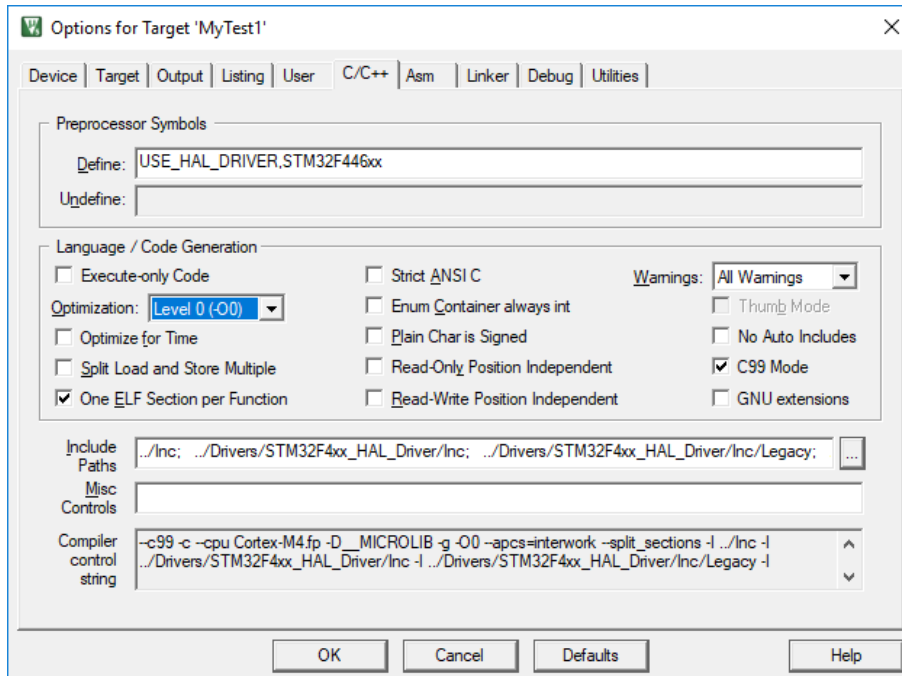
```

COM4 - PuTTY
1+2+3+0=6
1+2+3+1=7
1+2+3+2=8
1+2+3+3=9
1+2+3+4=10
1+2+3+5=11
1+2+3+6=12
1+2+3+7=13
1+2+3+8=14
1+2+3+9=15
1+2+3+10=16
1+2+3+11=17
1+2+3+12=18
1+2+3+13=19
1+2+3+14=20
1+2+3+15=21
1+2+3+16=22
1+2+3+17=23
1+2+3+18=24
1+2+3+19=25
1+2+3+20=26
1+2+3+21=27
1+2+3+22=28

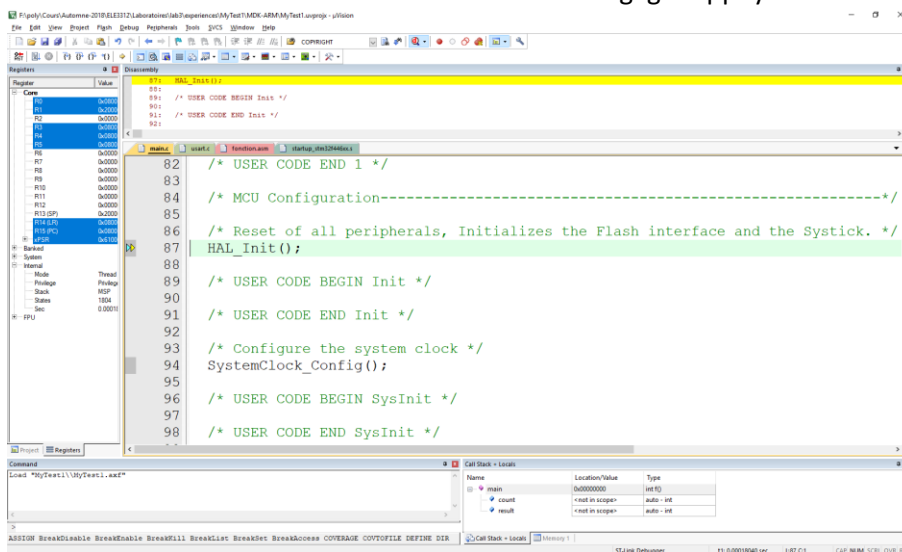
```

## Exécuter un programme pas à pas

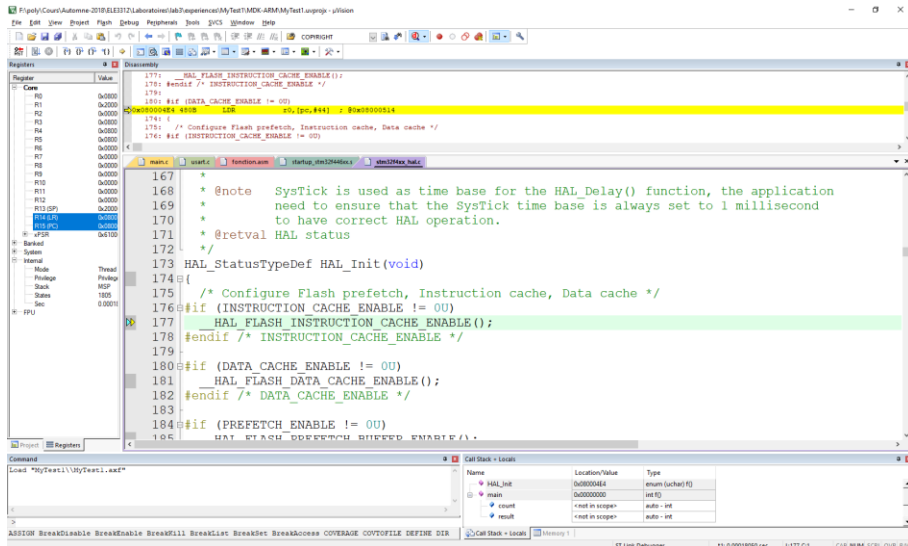
1. Dans un premier temps, nous allons forcer le compilateur à ne pas faire d'optimisation pour mieux suivre l'exécution des instructions pas à pas. Allez dans le menu Project/Options for target ... et choisissez le niveau d'optimisation 0 dans l'onglet C/C++:



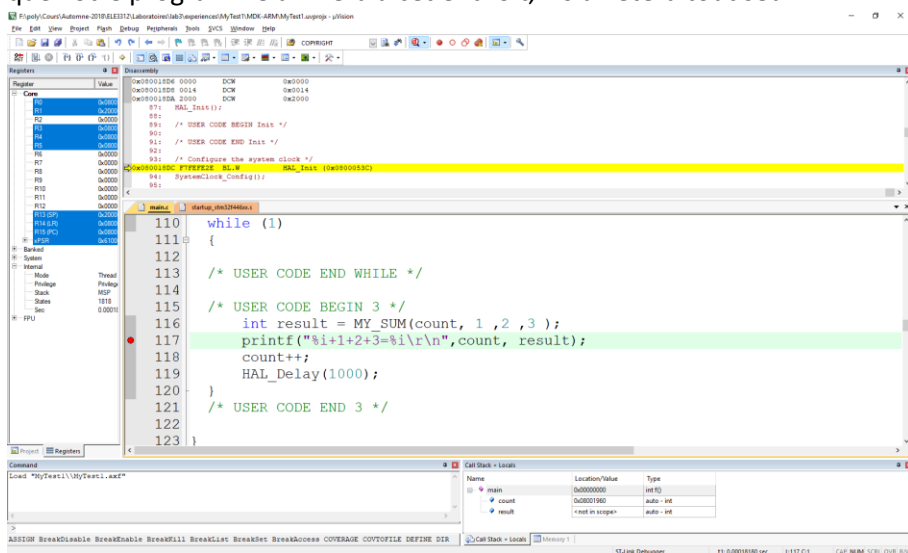
2. Appuyez sur OK et recompilez tout le projet dans le menu Project/Rebuild all target files.
3. Nous allons maintenant travailler en mode de débogage. Appuyez sur Ctrl-F5 :



4. Votre microcontrôleur est maintenant sous le contrôle permanent de l'IDE Keil. Ce dernier peut démarrer ou arrêter l'exécution à tout moment, lire et modifier les registres, les périphériques, la mémoire etc. La ligne précédée par une petite flèche jaune est la prochaine ligne qu'il va exécuter lorsque vous lui demanderez. Automatiquement, l'IDE Keil vous ouvre le fichier stm32f4xx\_hal.c contenant la fonction suivante à exécuter :



5. Appuyez sur F11 (Step in) de manière répétée (au moins une trentaine de fois) pour observer le déroulement du programme, pas à pas
6. Appuyez sur F5 (Run), qui va sortir du mode pas à pas et lancer l'exécution du programme. Vous devriez voir apparaître les messages sur la console Putty
7. Appuyez sur l'icône « Stop » (une croix blanche sur un rond rouge). On est revenu au mode pas à pas.
8. Retournez au fichier main.c et cliquez à gauche de la ligne qui contient le printf(). Il faut cliquer à gauche du numéro de ligne en fait. Vous venez de mettre un point d'arrêt (un point rouge). Dès que votre programme arrivera à cet endroit, il s'arrêtera tout seul :



9. Appuyez sur F5 pour relancer votre programme. Vous remarquerez qu'il arrive assez rapidement sur le point d'arrêt que vous avez introduit.
10. Appuyez sur F10 (Step over) de manière répétée. Vous observez que l'exécution se relance à chaque fois jusqu'à ce que la ligne en cours soit terminée. Si la ligne en cours fait appel à d'autres méthodes, elle seront toutes appelées avec que l'IDE Keil vous rende la main.

11. Vous pouvez aussi utiliser le bouton Ctrl-F11 pour sortir de la fonction en cours.
12. À tout moment, lorsque vous passez votre souris au dessus d'une variable accessible, sa valeur s'affiche. Vous pouvez aussi visualiser la valeur de chaque registre dans la section gauche de la fenêtre.
13. Pour sortir du mode Debug, appuyez (à nouveau) sur Ctrl F5.

### Expérience 1

*Réalisez un compteur binaire 4 bits semblable à celui du laboratoire 2 en assembleur. Plus précisément, la boucle infinie en C appellera une fonction mycount() qui incrémente la valeur du compteur et attend ensuite 1000ms. On suppose toujours que les DELs du compteur sont reliées aux ports GPIOC[3..0] à travers une résistance de 330R pour chaque DEL.*

**Pour rappel, le code C du laboratoire 2 :**

```
int count = 0;

while (1)
{
    GPIOC->ODR = count;
    count = count+1;
    if (count == 16) count = 0;
    HAL_Delay(1000);
}
```

**... doit devenir le suivant :**

```
while (1)
{
    fct_count();
}
```

Pour écrire mycount en assembleur, nous avons besoin de connaître l'adresse du registre pointé par GPIOC->ODR

Vous obtiendrez cette adresse en trouvant d'une part l'adresse de base du port GPIOC et d'autre part la position du champ ODR dans la structure. Pour trouver l'adresse de base du port GPIOC, allez voir dans le **manuel de référence** de votre processeur (typiquement le STM32F446xx).

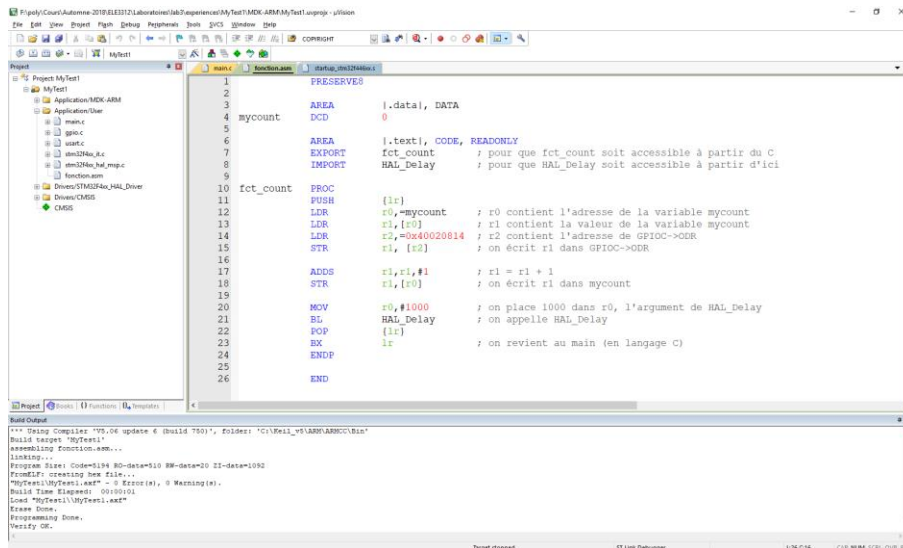
[http://www.st.com/resource/en/reference\\_manual/dm00135183.pdf](http://www.st.com/resource/en/reference_manual/dm00135183.pdf)

1. Faites une recherche sur GPIOC et notez sa valeur ici : \_\_\_\_\_

Dans le même document, faites ensuite une recherche sur « GPIO register map » et vous trouverez le décalage de chaque champ par rapport à l'adresse de base. Par exemple, le champ « OTYPEPER » a un décalage de 4 bytes par rapport à l'adresse de base.

Indiquez ici le décalage pour le champ ODR du port GPIOC : \_\_\_\_\_

2. Vous pouvez maintenant essayer le programme assembleur suivant :



(Vous pouvez copier-coller le code ci-dessous)

```

PRESERVE8

mycount      AREA      |.data|, DATA
              DCD       0

fct_count     AREA      |.text|, CODE, READONLY
              EXPORT    fct_count
              IMPORT    HAL_Delay

              PROC
              PUSH      {lr}
              LDR        r0,=mycount
              LDR        r1,[r0]
              LDR        r2,=0x40020814
              STR        r1, [r2]

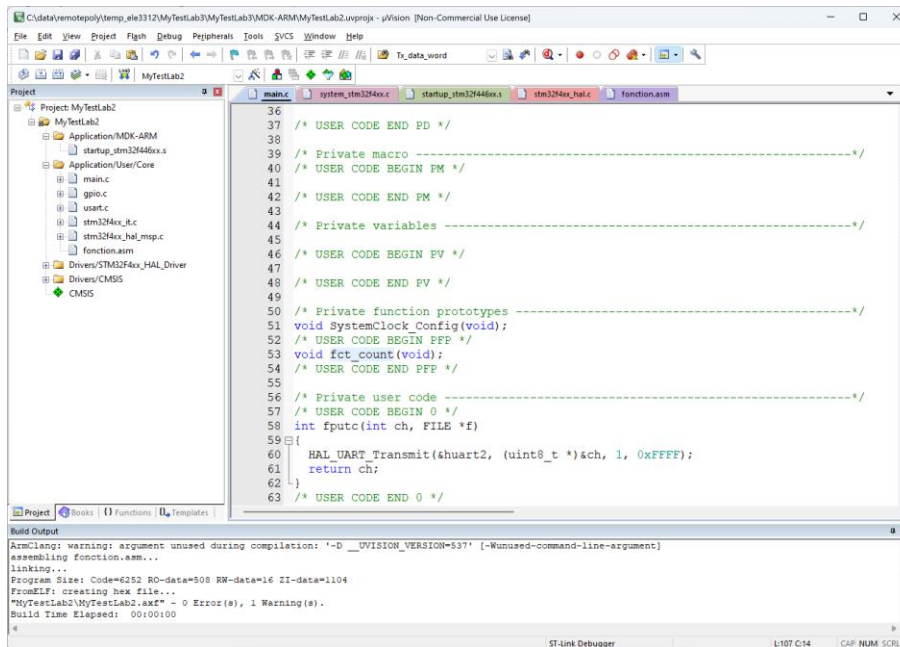
              ADDS       r1,r1,#1
              STR        r1,[r0]

              MOV        r0,#1000
              BL         HAL_Delay
              POP        {lr}
              BX         lr
              ENDP

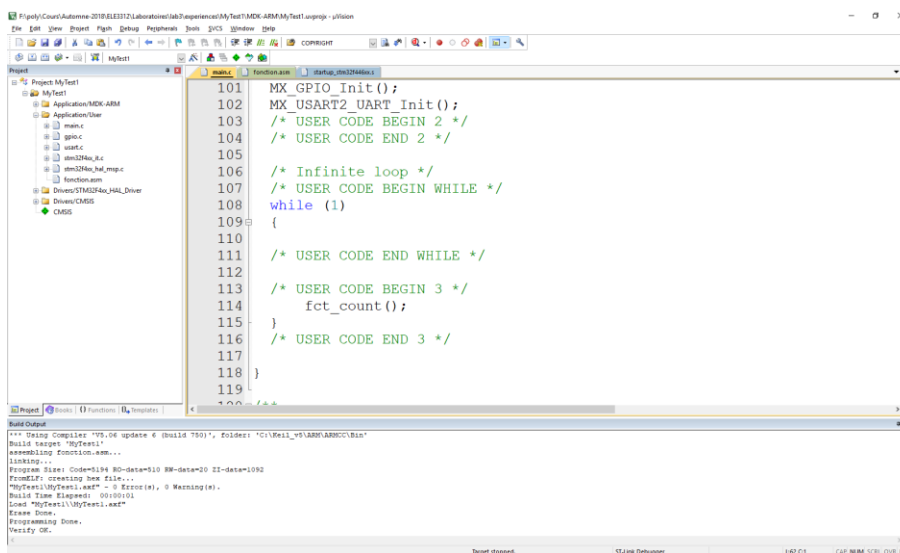
              END

```

3. Déclarez ensuite votre fonction dans le code C :



4. Éditez le code de la boucle principale comme suit :



5. Compilez (F7)
6. Passez en mode Debug (Ctrl F5)
7. Lancez l'application (F5)
8. Allez dans le menu Peripherals/System Viewer/GPIO/GPIOC
9. Vous pouvez maintenant observer en temps réel l'évolution du contenu de tous les registres reliés au port C
10. Essayez les diverses options de débogage pour vous assurer que vous comprenez bien le fonctionnement de chacune des instructions. N'oubliez-pas que vous pouvez voir le contenu des registres dans la partie gauche de l'IDE Keil.

## Expérience 2

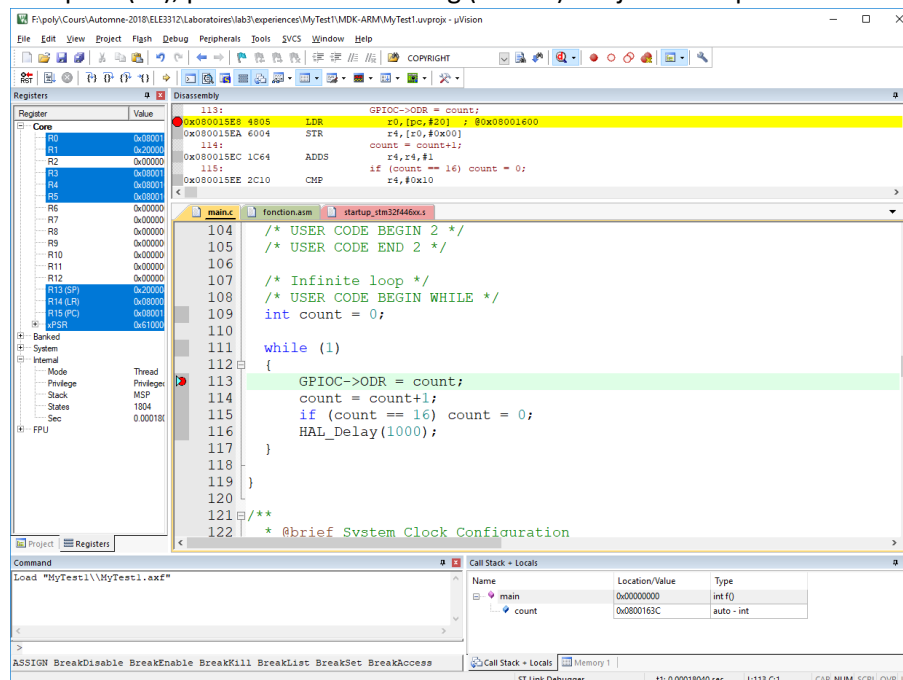
Observez comment l'IDE Keil compile une fonction C en code assembleur

1. Repartez du code C implantant le compteur :

```
int count = 0;

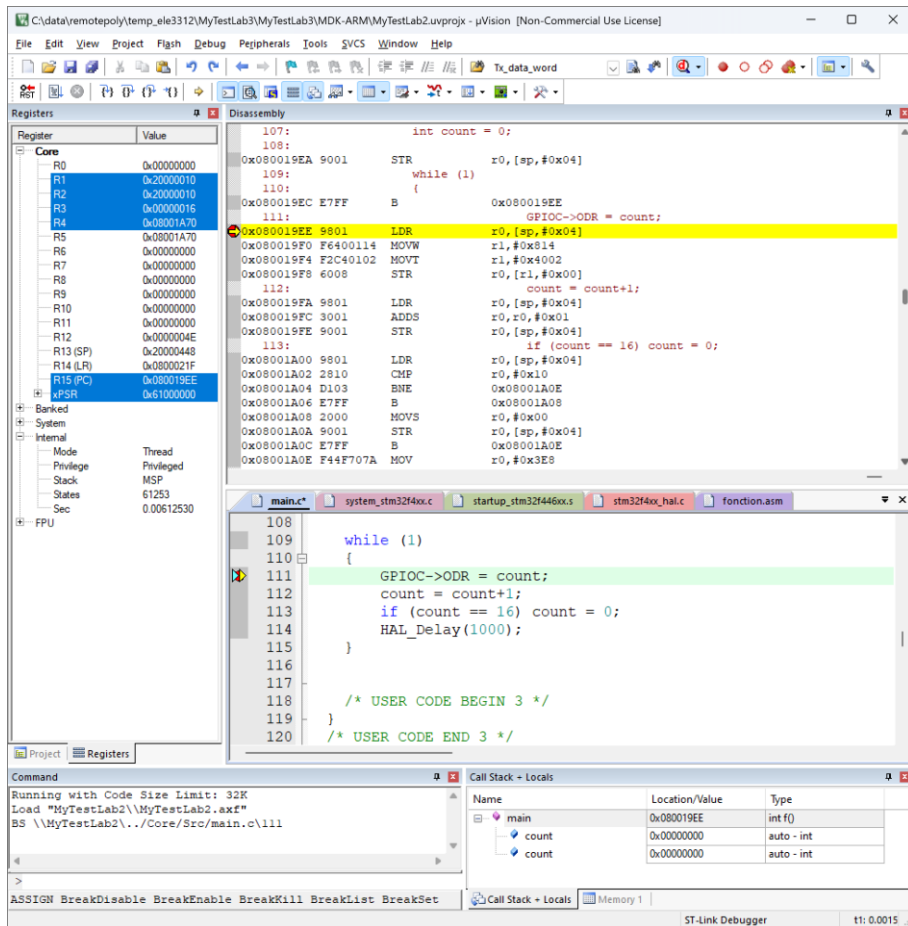
while (1)
{
    GPIOC->ODR = count;
    count = count+1;
    if (count == 16) count = 0;
    HAL_Delay(1000);
}
```

2. Compilez (F7), passez en mode Debug (Ctrl-F5) et ajoutez un point d'arrêt sur la première ligne



3. Lancez l'exécution (F5) et attendez que le programme s'arrête sur le point d'arrêt.
4. Élargissez la zone d'affichage correspondant au code assembleur en haut à droite :





5. Essayons de comprendre le code. Notez que les adresses peuvent différer dans votre version du code :

```

0x080019E8 2000      MOVS          r0,#0x00
107:                int count = 0;
108:
0x080019EA 9001      STR           r0,[sp,#0x04]
109:                while (1)
110:                {
0x080019EC E7FF      B             0x080019EE
111:                GPIOC->ODR = count;
0x080019EE 9801      LDR           r0,[sp,#0x04]
0x080019F0 F6400114  MOVW          r1,#0x814
0x080019F4 F2C40102  MOVT          r1,#0x4002
0x080019F8 6008      STR           r0,[r1,#0x00]
112:                count = count+1;
0x080019FA 9801      LDR           r0,[sp,#0x04]
0x080019FC 3001      ADDS          r0,r0,#0x01
0x080019FE 9001      STR           r0,[sp,#0x04]
113:                if (count == 16) count = 0;
0x08001A00 9801      LDR           r0,[sp,#0x04]
0x08001A02 2810      CMP           r0,#0x10
0x08001A04 D103      BNE          0x08001A0E
0x08001A06 E7FF      B             0x08001A08
0x08001A08 2000      MOVS          r0,#0x00
0x08001A0A 9001      STR           r0,[sp,#0x04]
0x08001A0C E7FF      B             0x08001A0E
0x08001A0E F44F707A  MOV          r0,#0x3E8
114:                HAL_Delay(1000);
0x08001A12 F7FEFC6F  BL.W          0x080002F4 HAL_Delay
109:                while (1)
0x08001A16 E7EA      B             0x080019EE

```

Remarquez d’abord que les lignes qui commencent par un nombre entier décimal correspondent à une ligne de code source en C. Elles sont là pour aider la compréhension du code assembleur qui suit. Les instructions en assembleur commencent par un nombre en hexadécimal qui est l’adresse où elles se trouvent en mémoire. Le deuxième nombre hexadécimal qui suit est le code de l’instruction. Ensuite vient l’instruction elle-même. Par exemple, la première instruction :

```

107:                int count = 0;
108:
0x080019EA 9001      STR           r0,[sp,#0x04]

```

On comprend que le code source C de la ligne 107 a été traduit par l’instruction : *STR r0,[sp,#0x04]*. Cette instruction est une instruction 16 bits qui se code 0x9001. Elle a été déposée en mémoire à l’adresse 0x080019EA / 0x080019EB (2 octets). On en conclut que, à ce stade, la variable count est implantée sur la pile à l’adresse sp+4.

Regardons maintenant le corps de la boucle :

```

111:                GPIOC->ODR = count;
0x080019EE 9801      LDR           r0,[sp,#0x04]
0x080019F0 F6400114  MOVW          r1,#0x814
0x080019F4 F2C40102  MOVT          r1,#0x4002
0x080019F8 6008      STR           r0,[r1,#0x00]

```

La première instruction va lire le contenu de la mémoire à l'adresse `sp+4` (donc la variable *count*) et copie son contenu dans le registre `r0`. Il place ensuite la constante `0x0814` dans `r1` et puis la constante `0x4002` dans les bits de poids fort de `r1`, ce qui revient au final à mettre la constante `0x40020814` dans `r1`. Or cette valeur correspond exactement à l'adresse du registre *GPIOC->ODR*. Cette adresse est donc copiée dans le registre `r1`. Pour compléter l'instruction *GPIOC->ODR = count*, il suffit maintenant d'écrire `r0` (qui contient *count*) dans la mémoire à l'adresse `r1`

Assurez-vous de bien comprendre comment chaque instruction C est traduite en assembleur. Par exemple, l'appel de la fonction `HAL_DELAY(1000)` se fait au moyen de deux instructions :

```
0x08001A0E F44F707A MOV          r0,#0x3E8
114:                                HAL_Delay(1000);
0x08001A12 F7FEFC6F BL.W        0x080002F4 HAL_Delay
```

La première instruction place la valeur 1000 dans le registre `R0` (qui doit toujours contenir la valeur du premier paramètre lors de l'appel d'une fonction). La seconde est l'appel à la fonction du HAL, définie dans les fichiers générés pour le HAL.

Petit commentaire par rapport au code suivant :

```
0x08001A02 2810      CMP          r0,#0x10
0x08001A04 D103      BNE          0x08001A0E
```

Il s'agit d'un branchement conditionnel. La première ligne compare `r0` à la valeur `0x10` (16). La deuxième ligne fait un branchement à l'adresse `0x08001A0E` si, lors de la dernière opération de comparaison, `r4` n'était pas égal à la valeur comparée.

Finalement, faites une exécution pas à pas et assurez-vous que vous comprenez parfaitement tout ce qui se passe à chaque instruction assembleur rencontrée. Vous pouvez suivre l'évolution du contenu des registres dans la section correspondante sur la gauche de la fenêtre.

## 2<sup>ème</sup> partie : Laboratoire en salle à Polytechnique

Un énoncé parmi les suivants sera assigné aléatoirement à chaque équipe.

### Énoncé 1

Écrire une fonction en assembleur dont le prototype C est :

```
int sq_norm(int *p_x, int *p_y, int *p_z);
```

Cette fonction calcule la somme des carrés des entiers x, y, et z passés par leurs adresses p\_x, p\_y et p\_z. Par exemple, on aurait le code suivant :

```
int x, int y, int z;  
int *p_x = &x;  
int *p_y = &y;  
int *p_z = &z;
```

```
... sq_norm(p_x, p_y, p_z);
```

Validez son fonctionnement au moyen d'un programme en C qui affiche (sur le terminal PUTTY) toutes les combinaisons pour x, y et z variant de -2 à +2 (125 combinaisons en tout).

Pour chaque combinaison, vous afficherez également le résultat de sq\_norm(x,y,z).

Finalement, écrivez le code C équivalent de cette fonction, compilez-le avec l'IDE Keil et comparez le code assembleur généré automatiquement avec celui que vous aviez écrit manuellement. Assurez-vous de bien comprendre tout le code assembleur généré (celui correspondant à sq\_norm et celui correspondant à votre programme d'affichage des combinaisons).

## Énoncé 2

Écrire une fonction en assembleur dont le prototype C est :

```
void swap(int *p_x, int *p_y);
```

La fonction doit échanger les deux entiers en mémoire pointés par les arguments `p_x` et `p_y`.

Validez son fonctionnement au moyen du programme de tri de type *bubbleSort* appliqué sur un tableau de 10 éléments {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}.

```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
```

Source : <https://www.geeksforgeeks.org/bubble-sort/>

Affichez le contenu du tableau avant et après l'appel de `bubbleSort` sur le terminal PUTTY au moyen de la méthode `printArray`.

```
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

Finalement, écrivez le code C équivalent de cette fonction, compilez-le avec l'IDE Keil et comparez le code assembleur généré automatiquement avec celui que vous aviez écrit manuellement. Assurez-vous de bien comprendre tout le code assembleur généré (celui correspondant à `swap` et celui correspondant à `bubbleSort`).

### Énoncé 3

Écrire une fonction en assembleur dont le prototype C est :

```
int positive(int *value);
```

Cette fonction remplace l'entier `*value` passé par adresse par sa valeur absolue et retourne cette valeur.

Note : l'algorithme suivant permet de calculer la valeur absolue d'un nombre entier sans faire de test :

```
int y = value >> 31;
return (value ^ y) - y;
```

Écrivez une fonction qui applique `positive` à un tableau de 10 entiers que vous aurez initialisé préalablement. Aussi, affichez le contenu du tableau avant et après l'appel de `positive` sur le terminal PUTTY au moyen de la méthode `printArray`.

```
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

Finalement, écrivez le code C équivalent de cette fonction, compilez-le avec l'IDE Keil et comparez le code assembleur généré automatiquement avec celui que vous aviez écrit manuellement. Assurez-vous de bien comprendre tout le code assembleur généré (celui correspondant à `positive` et celui correspondant aux boucles).