

ELE3312
Microcontrôleurs et applications
Laboratoire 5

Auteur : Jean Pierre David

Introduction

De manière générale, quand c'est possible, on a toujours intérêt à utiliser des langages de programmation et des bibliothèques qui utilisent des concepts de haut niveau. Le code est plus court, plus lisible, plus facilement vérifiable et modifiable. Par exemple, si on écrit deux boucles imbriquées en C pour effacer tous les pixels de l'écran, le code pourrait se lire ainsi :

```
for (int y=0; y<320; y++)
    for (int x=0; x<240; x++)
        ili9341_draw_pixel (_screen,0,x,y);
```

Par contre, si on l'écrit en assembleur, on aurait le code suivant :

0x080030CC	2500	MOVS	r5,#0x00
0x080030CE	E00A	B	0x080030E6
0x080030D0	2600	MOVS	r6,#0x00
0x080030D2	E005	B	0x080030E0
0x080030D4	2200	MOVS	r2,#0x00
0x080030D6	B229	SXTH	r1,r5
0x080030D8	B230	SXTH	r0,r6
0x080030DA	F7FEFB51	BL.W	ili9341_draw_pixel (0x08001780)
0x080030DE	1C76	ADDS	r6,r6,#1
0x080030E0	2EF0	CMP	r6,#0xF0
0x080030E2	DBF7	BLT	0x080030D4
0x080030E4	1C6D	ADDS	r5,r5,#1
0x080030E6	F5B57FA0	CMP	r5,#0x140
0x080030EA	DBF1	BLT	0x080030D0

Le second code est beaucoup plus difficile à comprendre et il le serait davantage encore si la méthode DrawPixel était recopiée en assembleur directement dans le code. Cependant, pour des questions d'efficacité (temps d'exécution, taille de la mémoire occupée) ou de compatibilité avec d'autres codes sources, il est parfois nécessaire d'utiliser des langages de plus bas niveau (comme l'assembleur) pour certaines parties du code. On doit alors bien comprendre non seulement comment chaque langage fonctionne mais également comment on peut communiquer d'un langage à l'autre et comment s'assurer de ne pas interférer avec le bon fonctionnement de l'autre langage. C'est le but de ce laboratoire.

Objectifs

1. Comprendre comment les données sont stockées en C
2. Apprendre comment déclarer des données dans un langage et les utiliser dans l'autre
3. Apprendra comment déclarer des fonctions dans un langage et les utiliser dans l'autre

1^{ère} partie : préparation à la maison

Ce laboratoire est organisé autour de la structure de donnée `ball_s` suivante, qui représente une balle dessinée à l'écran :

```

struct ball_s {
    int x;
    int y;
    short radius;
    short color;
};

```

Les champs x et y (des entiers de 32 bits), représentent la position du centre en x et y respectivement. Le champ radius (un entier 16 bits) représente le rayon de la balle. Un rayon négatif indique que la balle ne doit pas être affichée. Le champ color (un entier 16 bits) représente la couleur encodée en RGB (5 bits - 6 bits - 5 bits).

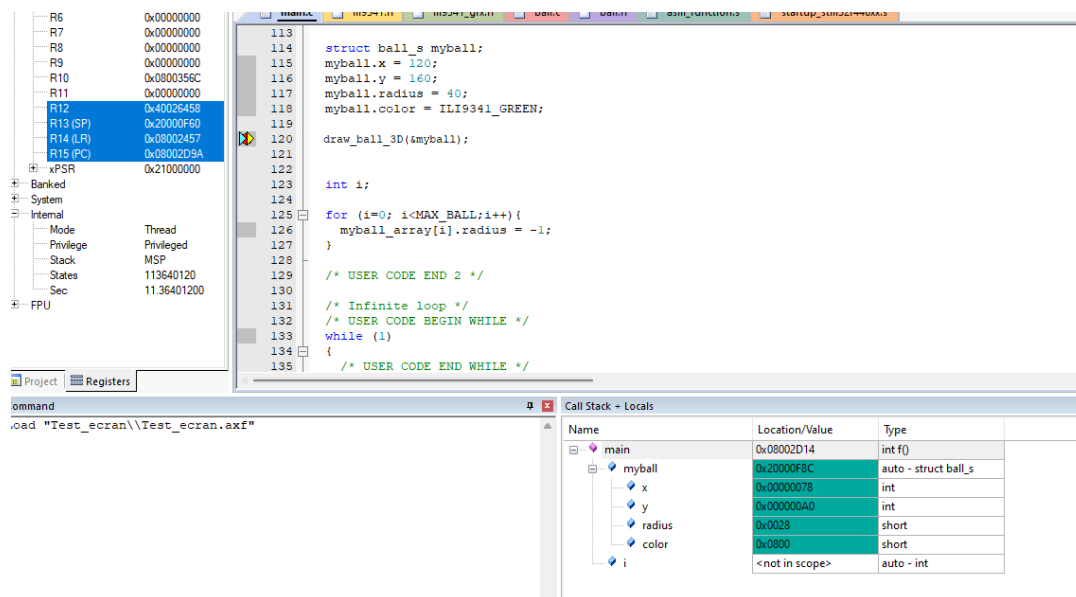
Cette structure nécessite donc 4+4+2+2=12 octets en mémoire, organisés en mode Little Endian (le poids faible est stocké à l'adresse la plus faible pour chaque champ).

L'application utilise un tableau myball_array de 3 éléments, soit 36 octets.

Expérience 1

Cette expérience vise à vous faire comprendre la correspondance entre chaque octet du tableau, leurs adresses et le contenu des champs des objets.

Compilez le code et mettez un point d'arrêt juste après l'initialisation de la structure myball. Ensuite, lancez une exécution pas à pas jusqu'à ce que le processeur s'arrête tel qu'indiqué à la figure suivante :



Le code source est disponible ci-dessous (ou [simplement télécharger l'archive sur Moodle](#)) :

```
#include "ball.h"
```

...

```

* USER CODE BEGIN PV */
#define MAX_BALL 3

```

```

...

struct ball_s myball_array[MAX BALL];      // Add extern for assembly
declaration
/* USER CODE END PV */
...

/* Private function prototypes -----*/
/* USER CODE BEGIN PFP */
void update_myball_array(void);
void asm_init_myball_array(void);
void asm_draw_all_ball_3D(struct ball_s *p_ball, int nb_ball);
/* USER CODE END PFP */
...

/* USER CODE BEGIN 2 */
_screen = ili9341_new(
    &hspi1,
    Void_Display_Reset_GPIO_Port, Void_Display_Reset_Pin,
    TFT_CS_GPIO_Port,      TFT_CS_Pin,
    TFT_DC_GPIO_Port,      TFT_DC_Pin,
    isoLandscape,
    NULL, NULL,
    NULL, NULL,
    itsNotSupported,
    itnNormalized);

ili9341_fill_screen(_screen, ILI9341_BLACK);

struct ball_s myball;
myball.x = 120;
myball.y = 160;
myball.radius = 40;
myball.color = ILI9341_GREEN;

draw_ball_3D(&myball);

int i;

for (i=0; i<MAX BALL; i++) {
    myball_array[i].radius = -1;
}

//Experience 2
//Uncomment here after to call the assembly function
//asm_init_myball_array();

//Experience 3
//Comment here after to call the assembly function
draw_all_ball_3D(myball_array, MAX BALL);

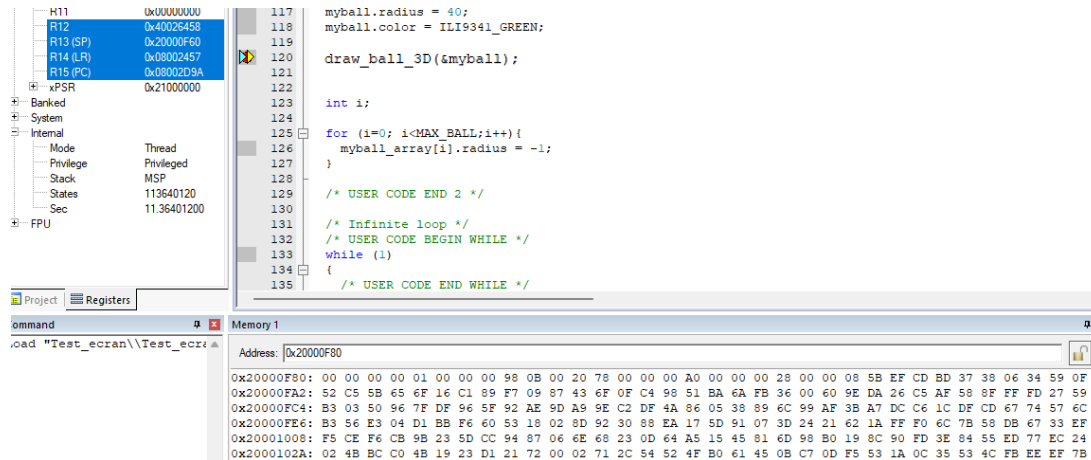
//Uncomment here after to call the assembly function
//asm_draw_all_ball_3D(myball_array, MAX BALL);

```

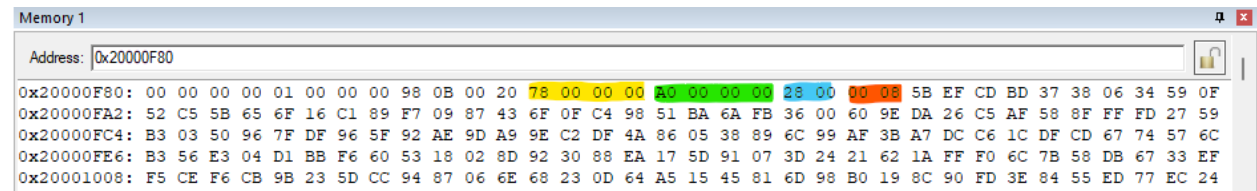
En bas à droite, vous voyez l'adresse du début de la structure (0x20000F8C dans notre cas) et la valeur de chaque champ (0x78, 0xA0, 0x28, 0x800). En toute logique, on devrait donc avoir :

- 1) La valeur 0x78 (x) aux adresses 0x20000F8C à 0x20000F8F
- 2) La valeur 0xA0 (y) aux adresses 0x20000F90 à 0x20000F93
- 3) La valeur 0x28 (radius) aux adresses 0x20000F94 à 0x20000F95
- 4) La valeur 0x800 (color) aux adresses 0x20000F96 à 0x20000F97

Pour s'en convaincre, cliquer sur l'onglet Memory pour en afficher le contenu :



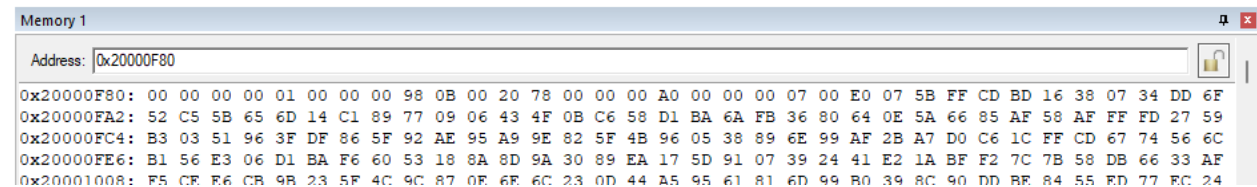
Le zoom ci-dessous identifie bien les 4 champs à partir de l'adresse 0x20000F80 :



On observe également la représentation en Little Endian : 0xA0 (32 bits) est en fait 0xA0-0x00-0x00-0x00 tandis que 0x800 (16 bits) est en fait 0x00-0x08

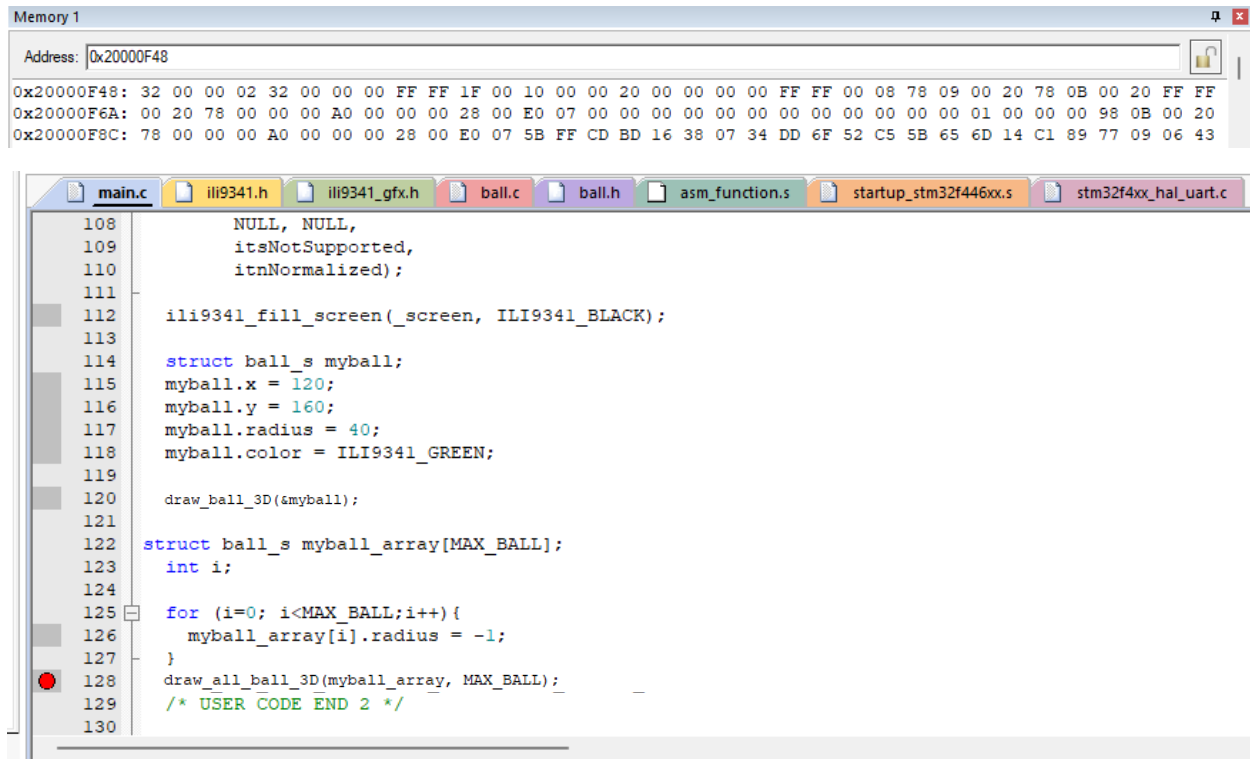
Si vous relancez l'exécution du programme, une balle verte de 40 pixels de rayon va s'afficher. À la place, modifiez un octet en mémoire (lequel ?) pour que la balle affichée ait un rayon de seulement 7 pixels.

La mémoire devient donc :



Relancez l'exécution (F5) et vérifiez que la balle verte est maintenant toute petite.

Présentement, le tableau de 3 balles n'est pas encore affiché car les rayons des balles ont été initialisés à -1 :



The screenshot shows a debugger interface with two main windows. The top window, titled 'Memory 1', displays a memory dump starting at address 0x20000F48. The data is organized in rows of 16 bytes each, showing hexadecimal values and their corresponding ASCII representations. The bottom window shows the source code of a C program. The code is in the file 'main.c' and shows the initialization of a ball array. The array is named 'myball_array' and has a size of 'MAX_BALL'. The code sets the radius of each ball to -1. The code is as follows:

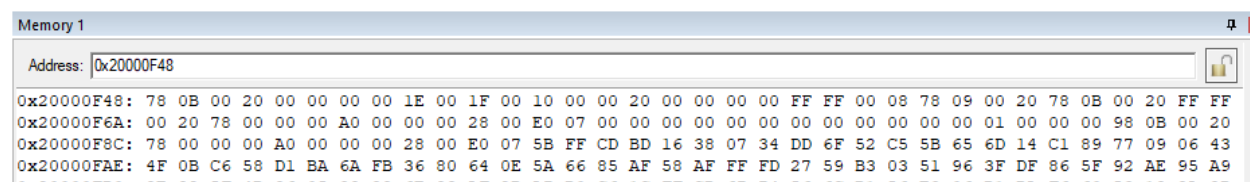
```

108     NULL, NULL,
109     itsNotSupported,
110     itnNormalized);
111
112     ili9341_fill_screen(_screen, ILI9341_BLACK);
113
114     struct ball_s myball;
115     myball.x = 120;
116     myball.y = 160;
117     myball.radius = 40;
118     myball.color = ILI9341_GREEN;
119
120     draw_ball_3D(&myball);
121
122     struct ball_s myball_array[MAX_BALL];
123     int i;
124
125     for (i=0; i<MAX_BALL;i++){
126         myball_array[i].radius = -1;
127     }
128     draw_all_ball_3D(myball_array, MAX_BALL);
129     /* USER CODE END 2 */
130

```

Ajoutez un point d'arrêt avant l'affichage du tableau. Faites un reset et relancez l'exécution pas à pas jusqu'à ce que vous soyez rendu au point d'arrêt en question. Ensuite, trouvez l'emplacement du tableau en mémoire et modifiez les valeurs de la première balle pour afficher une balle bleue (code couleur 0x1F) dont le centre est en (50,50) et dont le rayon est 30.

Dans notre exemple, cela donne ceci :



The screenshot shows the same debugger interface as before, but the memory dump now shows the first ball in the array with a radius of 30 and a color of 0x1F. The memory dump is as follows:

```

0x20000F48: 78 0B 00 20 00 00 00 00 1E 00 1F 00 10 00 00 20 00 00 00 00 FF FF 00 08 78 09 00 20 78 0B 00 20 FF FF
0x20000F6A: 00 20 78 00 00 00 A0 00 00 00 28 00 E0 07 00 00 00 00 00 00 00 00 00 01 00 00 00 98 0B 00 20
0x20000F8C: 78 00 00 00 A0 00 00 00 28 00 E0 07 5B FF CD BD 16 38 07 34 DD 6F 52 C5 5B 65 6D 14 C1 89 77 09 06 43
0x20000FAE: 4F 0B C6 58 D1 BA 6A FB 36 80 64 0E 5A 66 85 AF 58 AF FF FD 27 59 B3 03 51 96 3F DF 86 5F 92 AE 95 A9

```

Continuez l'exécution pour voir si votre balle bleue s'affiche bien.

Pouvez-vous définir la deuxième et la troisième balle à votre convenance dans le tableau également ?

Expérience 2

Nous allons maintenant écrire une fonction assembleur `asm_init_myball_array` qui va initialiser le tableau `myball_array`.

Pour que la fonction assembleur connaisse l'adresse du tableau, il faut donner au compilateur une directive particulière :

```
IMPORT      myball_array
```

Ensuite, pour que la fonction assembleur soit accessible par le code C, nous devons ajouter la directive suivante :

```
EXPORT      asm_init_myball_array
```

Nous devons aussi la déclarer dans le code C (déjà fait dans le code source) :

```
void asm_init_myball_array(void);
```

Le code source assembleur (dans le fichier `fonction.asm`) est le suivant :

```
AREA        |.text|, CODE, READONLY
EXPORT      asm_init_myball_array
IMPORT      myball_array

asm_init_myball_array    PROC
    LDR      r0,=myball_array
    MOV      r1,#60
    STR      r1,[r0],#4
    STR      r1,[r0],#4
    MOV      r1,#30
    STRH     r1,[r0],#2
    MOV      r1,#0x1F    ;blue
    STRH     r1,[r0],#2

    MOV      r1,#80
    STR      r1,[r0],#4
    STR      r1,[r0],#4
    MOV      r1,#30
    STRH     r1,[r0],#2
    MOV      r1,#0x07E0 ;green
    STRH     r1,[r0],#2

    MOV      r1,#100
    STR      r1,[r0],#4
    STR      r1,[r0],#4
    MOV      r1,#30
    STRH     r1,[r0],#2
    MOV      r1,#0xF800 ;red
    STRH     r1,[r0],#2
asm_init_myball_array    ENDP
```

```
    BX        lr
    ENDP
    END
```

Et finalement il faut ajouter l'appel de la fonction dans le code C :

```
asm_init_myball_array();    //Uncomment to call the assembly ...
```

Vérifiez que tout se passe comme prévu : trois balles de couleur bleue, verte et rouge doivent s'afficher dans le coin supérieur gauche de l'écran.

Expérience 3

Nous allons maintenant remplacer une fonction C par son équivalent assembleur. Dans un premier temps, dans le fichier ball.c, mettez un point d'arrêt sur la fonction :

```
void draw_all_ball_3D(struct ball_s *p_ball, int nb_ball);
```

Lancez l'exécution pas à pas et regardez le code assembleur qui a été généré pour cette fonction :

```
60: void draw_all_ball_3D(struct ball_s *p_ball, int nb_ball) {
```

```
0x08003148 B580 PUSH {r7,lr}
0x0800314A B084 SUB sp,sp,#0x10
0x0800314C 9003 STR r0,[sp,#0x0C]
0x0800314E 9102 STR r1,[sp,#0x08]
0x08003150 2000 MOVS r0,#0x00
```

```
61: for (int i=0; i<nb_ball; i++) {
```

```
0x08003152 9001 STR r0,[sp,#0x04]
0x08003154 E7FF B 0x08003156
0x08003156 9801 LDR r0,[sp,#0x04]
0x08003158 9902 LDR r1,[sp,#0x08]
0x0800315A 4288 CMP r0,r1
0x0800315C DA19 BGE 0x08003192
0x0800315E E7FF B 0x08003160
```

```
62: if (p_ball[i].radius>0) draw_ball_3D(&(p_ball[i]));
```

```
0x08003160 9803 LDR r0,[sp,#0x0C]
0x08003162 9901 LDR r1,[sp,#0x04]
0x08003164 EB010141 ADD r1,r1,r1,LSL #1
0x08003168 EB000081 ADD r0,r0,r1,LSL #2
0x0800316C F9B00008 LDRSH r0,[r0,#0x08]
0x08003170 2801 CMP r0,#0x01
0x08003172 DB09 BLT 0x08003188
0x08003174 E7FF B 0x08003176
0x08003176 9803 LDR r0,[sp,#0x0C]
0x08003178 9901 LDR r1,[sp,#0x04]
0x0800317A EB010141 ADD r1,r1,r1,LSL #1
0x0800317E EB000081 ADD r0,r0,r1,LSL #2
0x08003182 F000F809 BL.W 0x08003198 draw_ball_3D
0x08003186 E7FF B 0x08003188
```

```
63: }
```

```
0x08003188 E7FF B 0x0800318A
```

```
61: for (int i=0; i<nb_ball; i++) {
```

```
62: if (p_ball[i].radius>0) draw_ball_3D(&(p_ball[i]));
```

```
63: }
```

```
0x0800318A 9801 LDR r0,[sp,#0x04]
0x0800318C 3001 ADDS r0,r0,#0x01
```

```

0x0800318E 9001 STR r0,[sp,#0x04]
0x08003190 E7E1 B 0x08003156

```

64: }

```

0x08003192 B004 ADD sp,sp,#0x10
0x08003194 BD80 POP {r7,pc}
0x08003196 0000 MOVS r0,r0

```

Ce code utilise le Stack Pointer `sp` pour déposer les opérandes de la fonction sur la pile, et les lire quand nécessaire. Nous allons modifier ce code pour obtenir une version plus efficace en assembleur :

```

EXPORT      asm_draw_all_ball_3D
IMPORT      draw_ball_3D

asm_draw_all_ball_3D PROC
    PUSH     {r4-r6,lr}
    MOV      r5,r0
    MOV      r6,r1
    MOVS     r4,#0x00
    B        LABEL1
LABEL3      ADD      r0,r4,r4,LSL #1
            ADD      r0,r5,r0,LSL #2
            LDRSH    r0,[r0,#0x08]
            CMP      r0,#0x00
            BLE      LABEL2
            ADD      r1,r4,r4,LSL #1
            ADD      r0,r5,r1,LSL #2
            BL.W     draw_ball_3D
LABEL2      ADDS     r4,r4,#1
LABEL1      CMP      r4,r6
            BLT      LABEL3
            POP      {r4-r6,pc}
ENDP

```

Vous noterez les `IMPORT/EXPORT`.

Vérifiez que le code assembleur fait bien la même chose que le code C initial en appelant la fonction assembleur plutôt que la fonction C dans le main.

Assurez-vous de bien comprendre comment le code fonctionne. Pour vous aider, vous noterez que :

- 1) R0 contient initialement l'adresse du tableau (copiée au début dans R5)
- 2) R1 contient initialement la longueur du tableau (copiée au début dans R6)
- 3) R4 contient la valeur de `i`

Essentiellement, on a une boucle sur R4, qui est incrémenté en LABEL2 et comparé à sa valeur finale en LABEL3.

Dans le corps de la boucle, on calcule $R0 = (R4 + 2 * R4) * 4 + R5$, soit $12 * i + \text{l'adresse du tableau}$, ce qui va mettre l'adresse de la structure courante `p_ball[i]` dans `r0`.

Etc. ... mais assurez-vous de bien comprendre ☺

Pour terminer, il faudrait que nous soyons capables de définir des variables en assembleur tout en y ayant accès en C également. Nous allons donc déplacer la déclaration du tableau `myball_array` en assembleur.

D'abord, dans le fichier `main`, nous allons ajouter le mot clef `extern` pour dire au compilateur que le tableau existe mais qu'il est déclaré ailleurs :

```
48
49 /* Private variables -----
50
51 /* USER CODE BEGIN PV */
52 ili9341_t *_screen;
53 #define MAX_BALL 3
54
55 extern struct ball_s myball_array[MAX_BALL];
56 //struct ball_s myball_array[MAX_BALL];
57 /* USER CODE END PV */
58
59 /* Private function prototypes -----
60 void SystemClock_Config(void);
61 void asm_init_myball_array(void);
62 void update_myball_array(void);
63 void asm draw all ball 3D(struct ball s *p ball, int nb ball);
```

Ensuite, nous allons le déclarer comme un espace de 36 bytes (3 éléments de 12 octets chacun) dans le fichier `function.asm` avec la commande `SPACE` :

```
1 PRESERVE8
2 AREA |.data|, DATA
3 EXPORT myball_array
4
5 myball_array
6 SPACE 36
7
8
9 AREA |.text|, CODE, READONLY
10 EXPORT asm_init_myball_array
11
12 ;IMPORT myball_array
13
14 asm_init_myball_array PROC
15 LDR r0, =myball_array
16 MOV r1, #60
17 STR r1, [r0], #4
18 STR r1, [r0], #4
19 MOV r1, #30
20 STRH r1, [r0], #2
21 MOV r1, #0x1F ;blue
22 STRH r1, [r0], #2
23
24 MOV r1, #80
25 STR r1, [r0], #4
26 STP r1, r0, #4
```

Et il ne faut pas oublier de commenter la ligne avec le `IMPORT` (en bleu ci-dessus) puisque la variable est maintenant déclarée dans le fichier assembleur.

Recompilez et vérifiez que tout fonctionne toujours convenablement.

2^{ème} partie : Laboratoire en salle à Polytechnique

Soit la boucle principale suivante :

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

/* USER CODE BEGIN 3 */
    draw_all_ball_3D(myball_array, MAX_BALL);
    update_myball_array();
    HAL_Delay(1000);
}
/* USER CODE END 3 */
```

Avec la méthode `update_myball_array()` définie comme suit :

```
void update_myball_array() {
    for (int i=0; i<MAX_BALL; i++) {
        if (myball_array[i].radius == -1) {
            myball_array[i].x=rand()%320;
            myball_array[i].y=rand()%240;
            myball_array[i].radius=rand()%10;
            switch (rand()%7+1) {
                case 0: myball_array[i].color=ILI9341_BLACK;
break;
                case 1: myball_array[i].color=ILI9341_BLUE;
break;
                case 2: myball_array[i].color=ILI9341_RED;
break;
                case 3: myball_array[i].color=ILI9341_GREEN;
break;
                case 4: myball_array[i].color=ILI9341_CYAN;
break;
                case 5: myball_array[i].color= ILI9341_MAGENTA;
break;
                case 6: myball_array[i].color= ILI9341_YELLOW;
break;
                case 7: myball_array[i].color= ILI9341_WHITE;
break;
            }
        } else {
            myball_array[i].radius+=3;
            if ((myball_array[i].radius >= myball_array[i].x) ||
                (myball_array[i].radius >=
myball_array[i].y) ||
                (myball_array[i].radius >= 320-
myball_array[i].x) ||
```

```

                                (myball_array[i].radius >= 240-
myball_array[i].y) ||
                                (myball_array[i].radius >= 40) )
                                myball_array[i].radius=-1;
                                }
                                }
                                }
}

```

Commencez par visualiser l'effet graphique de ce programme.

On vous demande ensuite d'implanter cette méthode `update_myball_array()` en assembleur. Toutefois, pour implanter l'appel à la méthode `rand()`, vous définirez la fonction suivante en C, que vous appellerez depuis l'assembleur :

```

int update_rand(int my_modulo) {
    return rand()%my_modulo;
}

```

Remarque importante : en début de laboratoire, le chargé de laboratoire vous assignera une nouvelle structure `struct ball_s` qui sera relativement compatible avec la précédente ... par exemple :

```

struct ball_s {
    char radius;
    int color;
    short x;
    int y;
};

```

À vous d'adapter votre code à la structure qui vous sera donnée. Pour vous aider, une version assembleur préliminaire a été réalisée pour vous ci-dessous. Cette version fonctionne avec la structure initiale de `ball_s` mais vous devrez l'adapter à celle que vous allez recevoir. **Très peu de commentaires ont été mis pour vous forcer à chercher à quoi servent toutes les instructions. Il vous appartient de commenter le code complètement une fois celui-ci transformé par vos soins.**

```

PRESERVE8
AREA                |.text|, CODE, READONLY

EXPORT              asm_update_myball_array
IMPORT              update_rand
IMPORT              myball_array

```

```

asm_update_myball_array      PROC
    PUSH                      {r4-r5,lr}
    ; r4 = i
    ; r5 = adresse de myball_array[i]

    MOV                       r4,#0
    LDR                       r5,=myball_array
    B                         TEST_LOOP

LAB1

    LDRSH                     r1,[r5,#8]
    ADDS                      r1,#1
    BNE                       LAB2

    MOV                       r0,#320
    BL                        update_rand
    STR                       r0,[r5]

    MOV                       r0,#240
    BL                        update_rand
    STR                       r0,[r5,#4]

    MOV                       r0,#10
    BL                        update_rand
    STRH                      r0,[r5,#8]

    MOVS                      r0,#7
    BL                        update_rand
    ADDS                      r0,#1

    CMP                       r0,#0
    MOVEQ                     r1,#0x0000           ;black

    CMP                       r0,#1
    MOVEQ                     r1,#0x001F           ;blue

    CMP                       r0,#2
    MOVEQ                     r1,#0xF800           ;red

    CMP                       r0,#3
    MOVEQ                     r1,#0x07E0           ;green

    CMP                       r0,#4
    MOVEQ                     r1,#0x07FF           ;cyan

    CMP                       r0,#5
    MOVEQ                     r1,#0xF81F           ;magenta

```

	CMP	r0,#6	
	MOVEQ	r1,#0xFFE0	;yellow
	CMP	r0,#7	
	MOVEQ	r1,#0xFFFF	;whilte
	STRH	r1,[r5,#10]	
	B	END_LOOP	
LAB2	LDRSH	r1,[r5,#8]	
	ADDS	r1,#3	
	STRH	r1,[r5,#8]	
	LDR	r2,[r5,#0]	
	CMP	r1,r2	
	BGE	CLEAR BALL	
	LDR	r3,[r5,#4]	
	CMP	r1,r3	
	BGE	CLEAR BALL	
	RSB	r2,r2,#320	
	CMP	r1,r2	
	BGE	CLEAR BALL	
	RSB	r3,r3,#240	
	CMP	r1,r3	
	BLT	END_LOOP	
CLEAR BALL	MOV	r1,#0xFFFF	
	STRH	r1,[r5,#8]	
END_LOOP	ADDS	r4,#1	
	ADDS	r5,#12	
TEST_LOOP	CMP	r4,#3	
	BLT	LAB1	
	POP	{r4-r5,pc}	
	ENDP		
	END		