UNIVERSITÉ SAVOIE MONT BLANC

# Cours

## Programmation C

## pour les Systèmes Embarqués

Sylvain MONTAGNY / sylvain.montagny@univ-smb.fr

Université Savoie Mont Blanc

**1 LE PROCESSUS DE COMPILATION ..... 3**

1.1 LA COMPILATION ET LA COMPILATION CROISEE ..... 3
1.2 LES ETAPES DE LA COMPILATION ..... 3
1.3 MODIFIER LES PARAMETRES DE LA COMPILATION ..... 5

**2 L'UTILISATION DES VARIABLES ..... 7**

2.1 LES DIFFERENTS TYPES DE VARIABLES ..... 7
2.2 LES CALCULS SUR LES VARIABLES ..... 9
2.3 CALCUL SUR LES POINTEURS ..... 11
2.4 CONVERTIR LES TYPES DES VARIABLES ..... 12
2.5 LE STOCKAGE DES VARIABLES EN MEMOIRE ..... 13

**3 LA PROGRAMMATION MULTI-FICHIERS ..... 20**

3.1 UTILISATION DES DIRECTIVES PREPROCESSEURS ..... 20
3.2 UTILISATION DE VARIABLES DEFINIES DANS UN AUTRE FICHIER ..... 21
3.3 INTERDICTION DE L'EXTERNALISATION ..... 21

**4 LECTURE ET MODIFICATION DES REGISTRES D'UN MICROCONTROLEUR ..... 23**

4.1 OPERATION SUR LES REGISTRES ..... 23
4.2 L'ORGANISATION MEMOIRE D'UN MICROCONTROLEUR STM32 ..... 25
4.3 REPRESENTATION DE LA MEMOIRE EN LANGAGE C ..... 25
4.4 SIMPLIFIER L'ACCES AUX BITS DES REGISTRES ..... 29

**5 LA GESTION DE L'AFFICHAGE DANS LES SYSTEMES EMBARQUES ..... 36**

5.1 PRINTF SUR UART ..... 36
5.2 PRINTF SUR AVEC ITM SUR SWO ..... 37

**6 REGLES DE CODAGE ET LISIBILITE DU CODE ..... 38**

6.1 OFUSCATED CODE ..... 38
6.2 NOS REGLES COMMUNES ..... 38
6.3 COMMENTS AND DOXYGEN ..... 39

# Introduction

Ce cours est à disposition des étudiants du master ESET de l'université Savoie Mont Blanc [ https://scem-eset.univ-smb.fr ]. Il est mis en ligne gratuitement, sans contrepartie et libre d'utilisation. Certaines parties des exercices sont cachées. Vous pouvez me solliciter si en tant qu'enseignant vous souhaitez visualiser la version complète.

Voici la liste non exhaustive d'autres ressources disponibles :

- Low Power on STM32 [English]: https://cutt.ly/lowpowerSTM32
- LoRa-LoRaWAN et IOT : https://cutt.ly/livrelorawan
- Processeurs de traitement du signal : https://cutt.ly/digitalsignalprocessors
- Systèmes d'exploitation temps réel : https://cutt.ly/TempsReel
- Bus de communication : https://cutt.ly/BusCom
- Programmation C pour les systèmes embarqués : https://cutt.ly/ProgC

Bon apprentissage.

# 1 Le processus de compilation

## 1.1 La compilation et la compilation croisée

Lorsque nous compilons un projet, nous générons du code exécutable (*.exe*, *.bin*, *.hex*, *.elf*). Ce code exécutable peut être destiné à plusieurs cibles :

- Le même processeur que celui sur lequel nous avons réalisé la compilation : On parle de compilation native. C'est le cas lorsque nous créons du code que nous exécutons directement sur notre PC.
- Un autre processeur que celui sur lequel nous avons réalisé la compilation. On parle de compilation croisée (Cross-Compilation). C'est le cas lorsque nous compilons une application à destination d'un Arduino, d'une Raspberry PI ou d'un STM32.

## 1.2 Les étapes de la compilation

La compilation permet de générer un fichier qui sera exécutable sur un processeur. La compilation est un terme générique qui est en fait découpé en plusieurs étapes.

- Fichier.c > [**Preprocesseur**] > Fichier.i (Fichier Source)
- Fichier.i > [**Compilation**] > Fichier.s (Fichier Assembleur)
- Fichier.s > [**Assemblage**] > Fichier.o (Fichier Objet)
- Fichier.o > [**Edition de liens**] > Fichier.out (Fichier Exécutable)

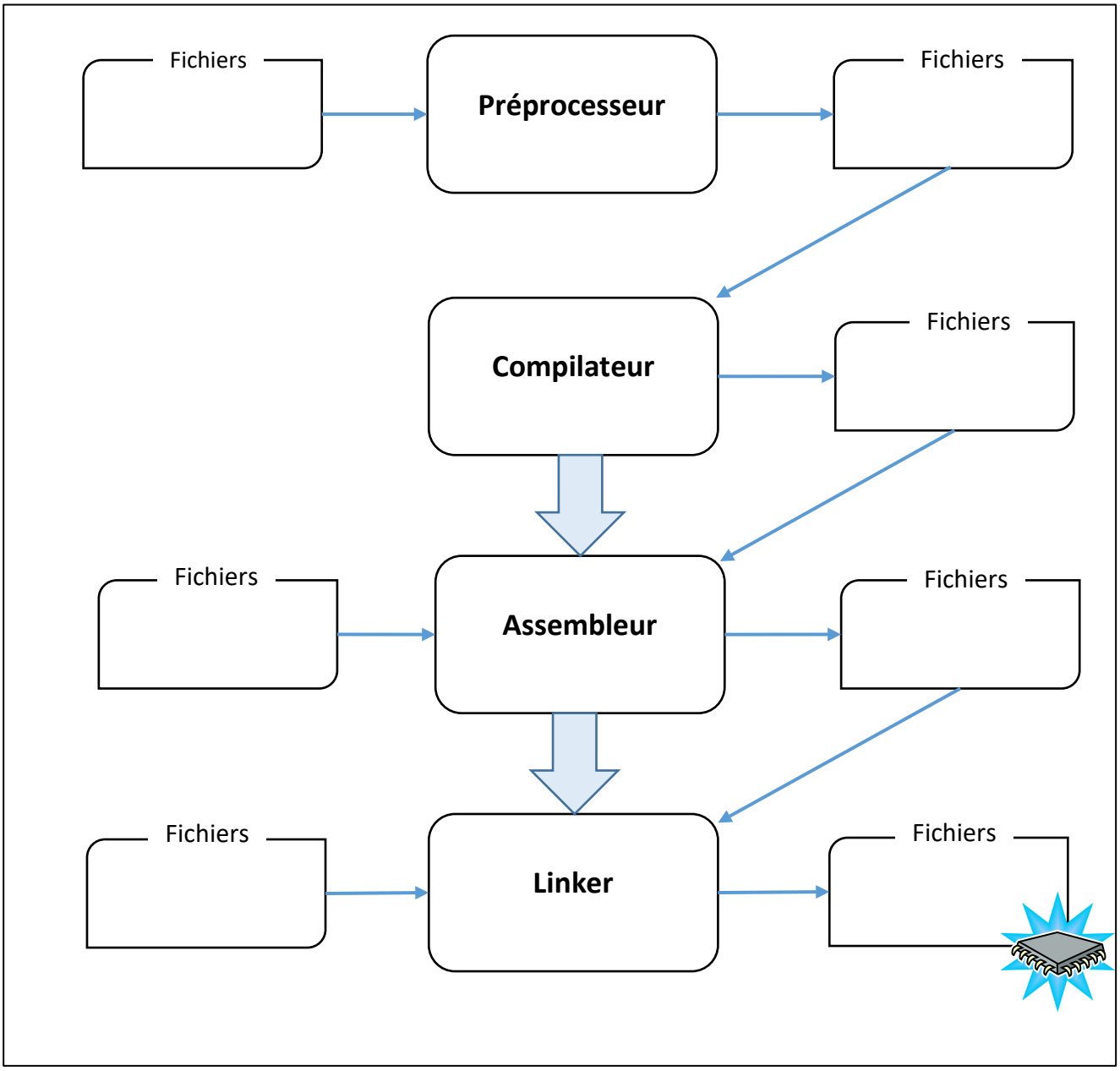


*Figure 1 : Processus de compilation d'une application*

Lors de l'utilisation de gcc (compilateur GNU), nous pouvons maitriser l'ensemble des étapes de compilation grâce au commandes suivantes :

| Commande / option | Compilation | Type de fichier en sortie |
|---|---|---|
| gcc fichier.c | Preprocessor Compiler Assembler Linker | |
| gcc –c fichier.c | Preprocessor Compiler Assembler | |
| gcc –S fichier.c | Preprocessor Compiler | |
| gcc –E fichier.c | Preprocessor | |
| | | |
| gcc fichier.c | Preprocessor Compiler Assembler Linker | |
| gcc fichier.i | Compiler Assembler Linker | |
| gcc fichier.s | Assembler Linker | |
| gcc fichier.o | Linker | |

*Tableau 1 : Fichier de sortie de compilation en fonction des options*

- Remplir la dernière colonne du tableau précédent.
- Démonstration avec un compilateur C.

## 1.3 Modifier les paramètres de la compilation

### 1.3.1 Pendant le preprocesseur avec *#error* et *#warning*

*#error* et *#warning* permettent au préprocesseur de produire des messages d'erreurs et de warning spécifiques lorsque le programmeur compile son projet.

```
#ifndef DEBUG
  #error("DEBUG must be defined");
#endif
```

### 1.3.2 Pendant la compilation

Il existe de très nombreuses options (directive de compilation). Par exemple :

- L'option ***-o*** permet de spécifier le nom du fichier de sortie (o=output).
- L'option ***-Wall*** (All Warning) permet d'avoir un retour plus précis sur les warnings.
- L'option ***-save-temps*** permet de faire apparaitre tous les fichiers intermédiaires (.i, .s, .o). Il faut utiliser le flag " à la compilation : ***gcc fichier.c –save-temps***

Dans STM32CubeIDE, nous pouvons préciser le flag de compilation dans : **Properties > C/C++ Build > Settings > Onglet Tool Settings > MCU GCC Compiler**.

### 1.3.3 Pendant l'édition de lien

L'objectif de l'édition de lien est de générer **un unique** fichier exécutable (*hex, elf ou bin*) qui permettra de programmer le microcontrôleur. Ce fichier définira :

- Pour chaque adresse de la mémoire flash, le mot mémoire qui sera stocké. Ce mot mémoire sera principalement une instruction (la Flash contient principalement du code) comme nous le verrons au paragraphe 2.5.1 (
- Pour chaque adresse de la mémoire RAM, le mot mémoire qui sera stocké. Ce mot mémoire sera principalement une donnée comme nous le verrons au paragraphe 2.5.2.

Ou souhaitez-vous placer exactement votre code ? Où souhaitez-vous placer exactement vos données ? Le linker ne peut pas deviner les emplacements que vous préconisez. Cette information doit lui être fournie sous forme d'un fichier appelé *linker script* (.ld).

La première partie (MEMORY) du *linker script* doit décrire les mémoires présentes, leur adresse et leur taille. Voici un exemple pour le STM32F446

```
/* Memories definition */
MEMORY
{
  RAM        (xrw)  : ORIGIN = 0x20000000,    LENGTH = 128K
  FLASH      (rx)   : ORIGIN = 0x8000000,     LENGTH = 512K
}
```

- Donner l'adresse de fin de SRAM2 sachant qu'elle commence à *0x2001 C000* et que sa taille est de 16KB.
- Donner l'adresse de fin de la Flash sachant qu'elle commence à *0x0800 0000* et que sa taille est de 512KB.
- Donner l'adresse de fin de SRAM1 sachant qu'elle commence à *0x2000 0000* et que sa taille est de 112KB.

La seconde partie (SECTION) du linker script donne la répartition dans les mémoires. Voici une version simplifiée en exemple pour le STM32F446.

```
SECTIONS
{
  .text :                 /* Le code (.text) va en Flash */
  {  } >FLASH

  .rodata :               /* Les constants (.rodata ) vont en Flash */
  {  } >FLASH

  .data :                 /* Les données vont en RAM */
  {  } >RAM
}
```

**STM32CubeIDE nous propose 2 fichiers *Linker Script*: STM32F446RETX_RAM.ld et STM32F446RETX_FLASH.ld. Le premier (RAM.ld) permet de tout mettre en RAM (code et data). Le second est celui de notre exemple ci-dessus. Il permet de mettre le code en FLASH et les données en RAM (ce qui est fait la plupart du temps).**

# 2 L'utilisation des variables

## 2.1 Les différents types de variables

### 2.1.1 Les variables entières et réels

Les deux types standard du langage C sont les entiers et les nombres réels. La déclaration se fait avec les mots clés *char, short, int, long, float* ou *double* en fonction de la taille de la variable souhaitée. (8, 16, 32 bits…). Seul l'intitulé *char* possède une taille définie et normalisée sur 8 bits. Tous les autres intitulés ont des tailles qui dépendent du compilateur et du processeur utilisé. Il convient de toujours se référer à la documentation afin de connaitre les valeurs possibles d'une variable.

Pour les variables entières, on peut différentier les nombres *unsigned* (Toujours positif) ou *signed* (positif ou négatif) :

- Les nombres *unsigned* sont écrit en binaire naturel
- Les nombres *signed* sont écrit en complément à 2

➡ Dans le tableau suivant, donner les plages de valeurs de chacune des variables. Pour les nombres réels, on rappelle que -2<|mantisse|<2

| Taille de la variable | Plage de valeur |
|---|---|
| 8 bits (entier signé) | |
| 32 bits (entier non signé) | |
| 32 bits (entier signé) | |
| 32 bits (réel signé)<br>24 bits de mantisse<br>8 bits d'exposant | |
| 64 bits (réel signé)<br>53 bits de mantisse<br>11 bits d'exposant | |

*Tableau 2 : Les différents types de variables utilisés en langage C*

### 2.1.2 L'uniformisation entre les systèmes

Les variables n'ont pas de taille normalisées et dépendent complétement du processeur et du compilateur utilisé. Afin de connaitre la taille d'une variable, il faut utiliser la fonction *sizeof()*.

```c
int main(void){
	printf("Size of char\t\t: %d octet\n",sizeof(char));
	printf("Size of short\t\t: %d octets\n",sizeof(short));
	printf("Size of int\t\t: %d octets\n",sizeof(int));
	printf("Size of long int\t: %d octets\n",sizeof(long int));
	printf("Size of long long int\t: %d octets\n",sizeof(long long int));
	printf("Size of float\t\t: %d octets\n",sizeof(float));
	printf("Size of double\t\t: %d octets\n",sizeof(double));
	printf("Size of long double\t: %d octets\n",sizeof(long double));
}
```

On exécute ce code sur trois architectures différentes : Un core i7 avec compilateur 32 bits, un ST32F446 (32 bits), et un PIC16F (8 bits).

| Architecture | x86 (64 bits) | STM32F4 (32 bits) | PIC16F877 (8 bits) |
|---|---|---|---|
| **Compilateur** | **MinGW32** | **ARM gcc** | **mikroC** |
| Size of char | 1 | 1 | 1 |
| Size of short | 2 | 2 | 1 |
| Size of int | 4 | 4 | 2 |
| Size of long int | 4 | 4 | 4 |
| Size of long long int | 8 | 8 | 4 |
| Size of float | 4 | 4 | 4 |
| Size of double | 8 | 8 | 4 |
| Size of long double | 12 | 8 | 4 |

*Tableau 3 : Taille des variables en fonction de l'architecture et du compilateur*

**Note : Les valeurs auraient été différentes avec un compilateur 64 bits (MinGW64).**

Cette multitude de représentation est source de beaucoup d'erreur et empêche la portabilité d'un code. C'est pourquoi on utilise de préférence les types normalisés de la bibliothèque *<stdint.h>.*

- *uint8_t, int8_t, uint16_t, int16_t, uint32_t, int32_t, uint64_t, int64_t*

Pour les valeurs réelles, quelques librairies spécifiques permettent aussi de normaliser les représentations. C'est le cas par exemple de *<arm_math.h>* :

- *float32_t, float64_t*

### 2.1.3 Adresse d'une variable

Chaque variable possède une adresse qu'il peut être utile de stocker. Nous utilisons donc des variables de type pointeur pour cela.

- *uint8_t*, int8_t*, uint16_t*, int16_t*, uint32_t*, int32_t*, uint64_t*, int64_t*, float32_t*, float64_t**

La taille d'un pointeur dépend aussi du compilateur et de l'architecture :

| Architecture | x86 (64 bits) | STM32F4 (32 bits) |
|---|---|---|
| Compilateur | MinGW32 | ARM gcc |
| Size of *char** | 4 | 4 |
| Size of *short** | 4 | 4 |
| Size of *int** | 4 | 4 |
| … | … | … |

*Tableau 4 : Taille des pointeurs en fonction de l'architecture et du compilateur*

**Note : Les valeurs auraient été différentes avec un compilateur 64 bits (MinGW64).**

## 2.2 Les calculs sur les variables

### 2.2.1 Les opérateurs et leur priorité

Les opérateurs peuvent être l'un des suivants :

| Opérateur par priorité | Nom ou signification |
|---|---|
| ++ -- | Incrémente , décrémente |
| * & | Déréférencement, Adressage |
| ! ~ | Négation, Complément |
| * / % + - | Multiplication, division, modulo, Addition, soustraction |
| << >> | Décalage logique |
| < > <= >= == != | Inf à, inf ou = à, sup à, sup ou = à, égale à, différent de |
| & ^ \| | ET, OU ex , OU **bit à bit** |
| && \|\| | Opérateur ET, OU **logique** |
| ?: | Opérateur conditionnel ternaire |
| = *= /= %= += -= <<= >>= &= \|= ^= | Opérateur d'affectation |

Le niveau de priorité est donné du plus fort au plus faible. L'opérateur "*++*" en haut à gauche du tableau précédent est l'opérateur réalisé en priorité.

| Expression en langage C | Résultat |
|---|---|
| `uint8_t tab[] = { 0xF0 , 0x0F };`<br>`printf ("\n %d", *tab<0xFF&&*tab>=0xF0?0x01:0x02);` | |

### 2.2.2 Incrémente, décrémentation (++ , --)

Cet opérateur ne s'attache qu'à un seul opérande qui doit être de type entier. Cela incrémente de 1 la variable qui lui est associée. Le positionnement de l'opérateur *++* ou - - est très important car suivant s'il est placé devant ou derrière la variable, il s'agit d'un post-incrémentation (l'incrémentation se fera après le calcul) ou d'une pré-incrémentation (l'incrémentation se fera avant le calcul).

| Expression utilisée en C | Expression équivalente |
|---|---|
| i++; | i = i + 1 |
| i--; | i = i - 1 |
| a = i++; | a = i<br>i = i + 1 |
| a = ++i; | i = i + 1<br>a = i |
| a = i--; | a = i<br>i = i - 1 |
| a = --i; | i = i - 1<br>a = i |

*Tableau 5 : Equivalence des expressions en langage C*

### 2.2.3 Test de relation entre opérandes (<, <=, > , >=, ==, !=)

Le résultat de cette opération peut être uniquement **1** ou **0**. La relation entre l'opérande de droite et celle de gauche sont testé et remplacé par le résultat.

| Expression en langage C | Résultat |
|---|---|
| `printf("\n %d %d", 3<5 , 5<3 );` | |
| `printf("\n %d %d", 5==5 , 5==3 );` | |
| `printf("\n %d %d", 3<=5 , 5<=3 );` | |
| `printf("\n %d %d", 5!=3 , 5!=5 );` | |

*Tableau 6 : Les tests entre opérandes*

Ces tests sont principalement utilisés dans les expressions conditionnelle if ou les boucle *while* / *for*.

### 2.2.4 Les opérateurs logiques (&&, ||, !)

Le résultat de cette opération peut être uniquement **1** ou **0**. Un *OU* (||), *ET* (&&) ou *NON* ( !) logique est opéré entre les deux opérandes.

| Expression en langage C | Résultat |
|---|---|
| `printf("\n %d %d", 3 && 5 , 3 && 0 );` | |
| `printf("\n %d %d", 5 \|\| 0 , 0 \|\| 0 );` | |
| `printf("\n %d %d", !0 , !5 );` | |

*Tableau 7 : Les opérateurs logiques*

### 2.2.5 Les opérateurs bit à bit (&, |, <<, >>, ~, ^)

Le résultat de cette opération est un nombre dont la taille correspond à la taille du plus grand opérande. Il s'agit d'une opération binaire bit à bit classique *ET*, *OU*, décalage à droite, décalage à gauche, complément, ou exclusif.

| Expression en langage C | Résultat |
|---|---|
| `printf("\n %08X %08X", 0xFFFFFFFF & 0x01 , 0xFFFFFFFF & 0xF0);` | 0x00000001<br>0x000000F0 |
| `printf("\n %08X %08X", 0xFFFFFFFF \| 0x01 , 0x00000003 \| 0x0C);` | 0xFFFFFFFF<br>0x0000000F |
| `printf("\n %08x %08x", 0xFFFFFFFF >> 1 , 0x00000003 << 5 );` | 0x7FFFFFFF<br>0x00000060 |
| `printf("\n %08x %08x", ~0x0000000F , ~0x00000003 );` | 0xFFFFFFF0<br>0xFFFFFFFC |
| `printf("\n %08x %08x", 0xFFFFFFFF ^ 0xF0 , 0x00000000 ^ 0xF0);` | 0xFFFFFF0F<br>0x000000F0 |
| `printf("\n %08x %08x",(int32_t)0xFFFFFFFF >> 1 ,(int32_t)0x00000003 << 5);` | 0xFFFFFFFF<br>0x00000060 |

*Tableau 8 : Les opérateurs bit à bit*

### 2.2.6 Les opérateurs d'affectation (=, +=, -=, *=, /=, %=, <<=, …)

Ces opérateurs permettent d'alléger la syntaxe d'une expression lorsque l'opérande est aussi le résultat. Mais attention, car parfois la lecture est moins aisée.

| Expression utilisée en C | Expression équivalente |
|---|---|
| i += 10 | i = i + 10 |
| i -= 10 | i = i - 10 |
| i *= 10 | i = i * 10 |
| i /= 10 | i = i / 10 |
| i %= 10 | i = i % 10 |

*Tableau 9 : Les opérateurs d'affectation*

### 2.2.7 L'opérateur conditionnel ternaire

Cet opérateur est la syntaxe allégée de *if{} else{}*.

```
condition ? vrai : faux;
```

La *condition* est testée. Si elle est vraie, l'instruction *vrai* est exécutée, sinon l'instruction *faux* est exécutée.

## 2.3 Calcul sur les pointeurs

Toutes les opérations vues précédemment sont valables sur les pointeurs. Cependant, le type de pointeur influence sur le résultat de certaines opérations.

```
uint8_t * ptrUint8  = 0x2000 0000;
uint16_t* ptrUint16 = 0x2000 0000;
uint32_t* ptrUint32 = 0x2000 0000;
```

| Incrémentation du pointeur | Valeur du résultat |
|---|---|
| ptrUint8 += 1; | |
| ptrUint16 += 1; | |
| ptrUint32 += 1; | |

*Tableau 10 : L'incrémentation des pointeurs*

- *ptrUint8* est un pointeur sur des entiers de 8 bits. Chaque entier de 8 bits (1 octet) pointé prend une adresse. Donc pour passer d'un entier à un autre on incrémente l'adresse de 1.

- *ptrUint16* est un pointeur sur des entiers de 16 bits. Chaque entier de 16 bits (2 octets) pointé prend 2 adresses. Donc pour passer d'un entier à un autre on incrémente l'adresse de 2.
- *ptrUint32* est un pointeur sur des entiers de 32 bits. Chaque entier de 32 bits (4 octets) pointé prend 4 adresses. Donc pour passer d'un entier à un autre on incrémente l'adresse de 4.

> **C'est le compilateur qui s'occupe de cela. A partir du moment où la déclaration a été faite correctement, le programmeur n'a pas besoin de se soucier de la valeur de l'incrémentation à réaliser.**

## 2.4 Convertir les types des variables

Le type de variable de part et d'autre du signe = doivent obligatoirement être identiques. Si ce n'est pas le cas, une conversion de type est réalisée. Si le programmeur précise cette conversion alors elle est dite **explicite**. Si le programmeur de le précise pas, alors le compilateur s'en chargera de façon **implicite**. Certaines conversions implicites ne sont pas autorisées et génèrent des erreurs de compilation ou des warnings.

### 2.4.1 Implicite

Lors des conversions implicites le compilateur utilise des règles hiérarchiques entre les types de variables afin de déterminer dans quel sens la conversion sera faite. En effet, si on prend le cas d'un calcul entre un *int* et un *float* : Est-ce que le *float* est convertit en *int* ou l'*int* en *float* ? La règle est donc la suivante sachant que les type à gauche sont les plus faibles.

*char < short < int < long < float < double < long double*

*signed < unsigned*

- Préciser les conversions implicites du code suivant :

```
1.  int n, m;
2.  double d;
3.   d = 2;
4.   n = 1;
5.   m = 5;
6.   d = n / m;
```

### 2.4.2 Explicite

Si on souhaite préciser au compilateur les conversions à réaliser, nous devons lui dire exactement ce que nous voulons faire.

- Préciser les conversions explicites du code suivant :

```
1.  int n, m = 5, l;
2.  double d = 2;
3.   n = (int) 1.4;
4.   l = (int) (m / d);
5.   d = n / ((double) m);
```

## 2.5 Le stockage des variables en mémoire

Lorsque nous injectons un programme dans un microcontrôleur (*.elf, .bin, .hex,* …), celui-ci est transféré en **mémoire Flash**. En plus du code, ce programme contient aussi toutes les variables de l'application (locales, globales, statiques, dynamiques, constantes, volatiles, …). Certaines resterons en mémoire Flash, d'autres seront destinées à êtes utilisées depuis la mémoire RAM. Nous allons voir :

- Quelles variables sont conservées en Flash
- Et pour les autres, comment après le chargement du programme en flash, les variables pourront être transférées en mémoire RAM.

### 2.5.1 Chargement du programme en flash

Après le chargement du code, la mémoire RAM est entièrement vide. Toute la programmation du composant est réalisée en mémoire flash comme le montre la Figure 2.

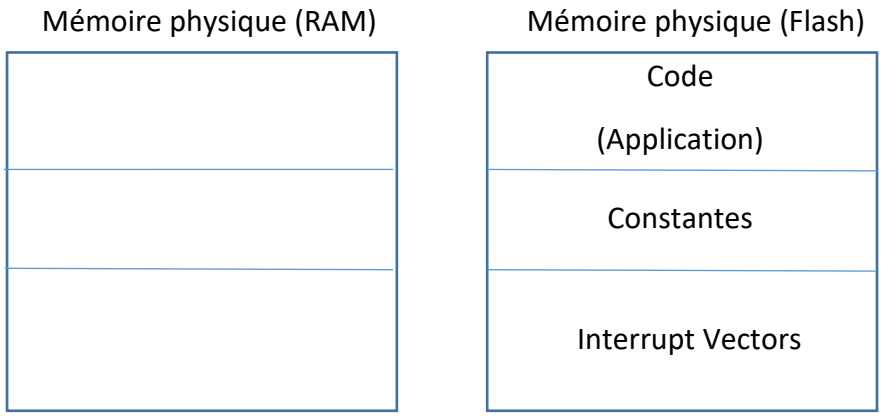


*Figure 2 : Organisation des mémoires Flash et RAM lors du chargement du code*

- **Le code** (application) correspond aux instructions du programme. Elles seront exécutée par le microprocesseur en commençant par la fonction *main().*
- **Les constantes** seront aussi stockées en flash car elles sont non modifiables. Ces variables ont été programmé avec le qualificateur *const* (voir 4.3.2).
- **Les vecteurs d'interruptions** sont les adresses des sous programmes d'interruption vers lequel le programme sera dérouté lorsque l'interruption correspondante interviendra. Cette liste des vecteurs est appelée table des vecteurs d'interruption. Dans le cas des STM32, cette table est fournie dans le fichier assembleur *Startup/startup_stm32xx.s* de votre projet. Voici une ébauche de ce fichier :

```
.word     I2C2_ER_IRQHandler   /* I2C2 Error        */
.word     SPI1_IRQHandler      /* SPI1              */
.word     SPI2_IRQHandler      /* SPI2              */
.word     USART1_IRQHandler    /* USART1            */
.word     USART2_IRQHandler    /* USART2            */
```

*void USART1_IRQHandler(void)* sera par exemple la fonction d'interruption liée au périphérique UART1, etc…

Comment est remplie la mémoire Flash ?

1. On regarde le fichier *linker script*. Dans la partie SECTIONS, on regarde le premier intitulé positionné en mémoire Flash : il s'agit de *.isr_vector* qui correspond aux vecteurs d'interruption.
2. On recherche les parties du codes où a été définie la section *.isr_vector*. Nous la retrouvons dans le fichier *startup.s*.

```
.section  .isr_vector
```

3. Chaque mot *(.word*) définie prend 32 bits en mémoire.
4. On continue alors avec le fichier *linker script*. Il y a ensuite la section .text…. etc …

Vérifions la cohérence des valeurs de la mémoire Flash d'un microcontrôleur STM32.

Pourquoi range t on la table des vecteurs d'interruption à cet endroit en particulier (@ 0x0000 0000) du microcontrôleur ?

Parce que c'est la documentation constructeur qui nous l'impose. Le microcontrôleur est câblé pour lire l'adresse d'une interruption à une adresse mémoire précise. Donc le fichier *startup.s* respecte simplement cette contrainte matérielle.

| Exception number | IRQ number | Offset | Vector |
|---|---|---|---|
| 16+n | n | 0x0040+4n | IRQn |
| . | | . | . |
| . | | . | . |
| . | | . | . |
| | | 0x004C | |
| 18 | 2 | 0x0048 | IRQ2 |
| 17 | 1 | 0x0044 | IRQ1 |
| 16 | 0 | 0x0040 | IRQ0 |
| 15 | -1 | 0x003C | Systick |
| 14 | -2 | 0x0038 | PendSV |
| 13 | | | Reserved |
| 12 | | | Reserved for Debug |
| 11 | -5 | 0x002C | SVCall |
| 10 | | | Reserved |
| 9 | | | |
| 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 |

*Tableau 11 : Table des vecteurs d'interruption ( Cortex M4 Generic User Guide)*

### 2.5.2 Organisation de la mémoire RAM

La mémoire RAM est par convention organisée de la façon suivante :

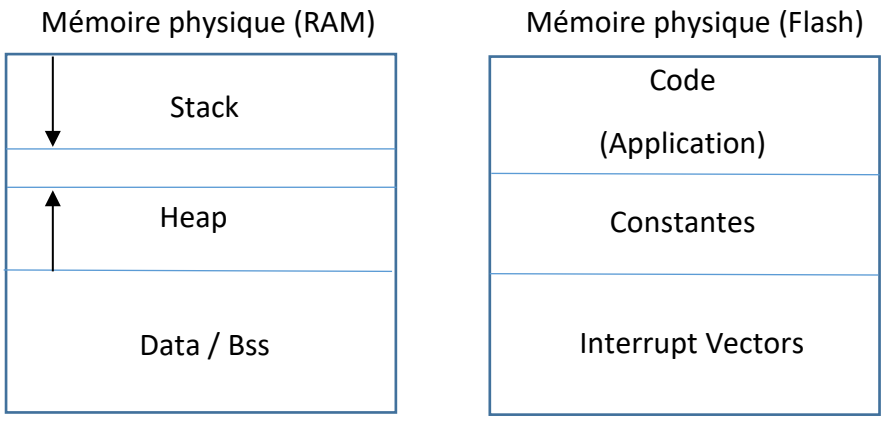


*Figure 3 : Organisation des mémoires après initialisation de la RAM*

- *Stack* : Emplacement pour le stockage des variables locales.
- *Heap* : Emplacement pour le stockage des variables allouées dynamiquement (malloc).
- *Data/ Bss* : Emplacement pour le stockage des variables globales ou des variables locales statics. Initialisées > *data*. Non initialisées > *bss*.

**La plupart du temps, la stack (pile) grandit vers le bas et le tas (heap) grandit vers le haut.**

- Donner l'emplacement en mémoire RAM des variables *var1, var2, var3, var4, ptrVar5* suivantes :

```
uint32_t var1 = 10;
uint32_t var2;

void main(void){
  uint32_t var3;
  static uint32_t var4 = 12 ;
  uint32_t* ptrVar5 = (uint32_t*) malloc( sizeof(uint32_t) );
}
```

Les valeurs des adresses de chacune des sections en mémoire sont les suivantes :

- L'adresse de la *stack* est donnée dans le *linker script* (STM32F446RETX_FLASH.ld) en recherchant l'intitulé *_estack*. On trouve la ligne correspondante ci-dessous :

```
/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM);   /* end of "RAM" Ram type memory */
```

- A l'aide du paragraphe 1.3.3, donner la valeur de ORIGIN(RAM) + LENGTH(RAM) permettant de connaitre l'adresse de départ de la pile (*stack*). On remarque que la pile débute à la dernière adresse de la mémoire RAM.

On rappelle que cette valeur est stockée à l'adresse *0x0000 0000* de la Flash. Nous pouvons donc le vérifier lors d'une session de debuggage.

Quand est-ce que le microcontrôleur va utiliser cette valeur d'adresse de la pile ?

Dès le début du programme, avant même l'exécution du *main()*. C'est donc une nouvelle fois le fichier *startup* qui s'occupe de cela en initialisant le pointeur de pile (*sp*).

Lors du débogage du STM32F446, on peut vérifier l'initialisation du pointeur de pile en mettant un point d'arrêt au moment de son initialisation : juste après le reset du microcontrôleur (*Reset_Handler*).

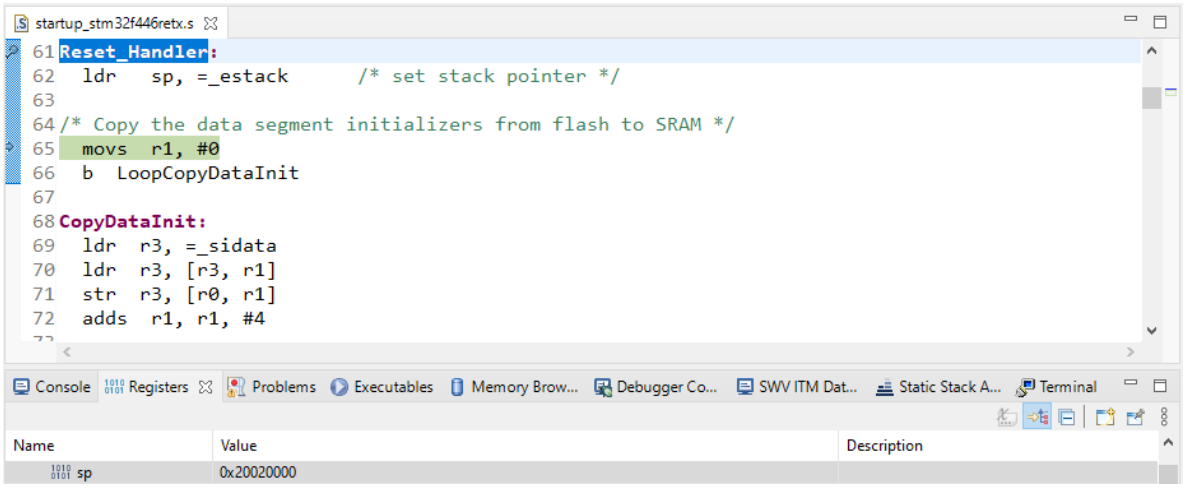


*Figure 4 : Valeur du registre sp (stack pointer) au reset (STM32F446)*

- L'adresse du tas (*heap*) n'est pas explicite dans le *linker script* STM32F446RETX_FLASH.ld, mais nous pouvons la retrouver d'après le schéma de la mémoire RAM (Figure 3) et d'après les informations du linker script suivantes :

```
_Min_Heap_Size = 0x200 ;  /* required amount of heap  */
_Min_Stack_Size = 0x400 ; /* required amount of stack */
```

On peut retrouver l'adresse de début du tas qui correspond à :

@start heap = @end RAM – stack size - heap size // d'après Figure 3

- L'adresse du segment data est donnée dans le *linker script* (STM32F446RETX_FLASH.ld) dans la section *.data*, sous l'intitulé *_sdata* (**s**tart **data**). En étudiant plus dans le détail le *linker script* on peut déterminer que *_sdata* prend la première adresse de la mémoire RAM, soit *0x2000 0000*.

- L'adresse du segment *bss* est donnée dans le *linker script* (STM32F446RETX_FLASH.ld) dans la section *.bss*, sous l'intitulé *_sbss* (**s**tart **bss**). En étudiant plus dans le détail le *linker script* on peut déterminer que le début de la section *_sbss* est située juste après la section *.data*.

### 2.5.3 Le startup code

Nous avons vu dans les paragraphes ci-dessus l'organisation de la Flash et de la RAM. Mais nous n'avons toujours pas expliquée comment les variables chargées en FLASH étaient stockées en RAM **avant** le lancement de la fonction *main()*.

Lorsque le microprocesseur démarre, le *PC* (*Program Counter*), ne s'initialise pas à l'adresse du *main()* mais à une adresse appelée *Reset_Handler*. Cette information est encore une fois donnée dans le *linker script* (STM32F446RETX_FLASH.ld) sous la forme suivante :

```
/* Entry Point */
ENTRY(Reset_Handler)
```

*Reset_Handler* est en fait le vecteur d'interruption du Reset, aussi appelé le *Reset Vector*. La fonction *Reset_Handler* est données dans le fichier *Startup/startup_stm32f446retx.s*. Le code est donné ci-dessous. Les lignes importantes ont été commentées. Ce code a pour rôle :

- De placer chaque variable de la section *.data* dans la mémoire RAM
- D'affecter à chaque variable de la section *.data* sa valeur d'initialisation
- De placer chaque variable de la section *.bss* dans la mémoire RAM. Ces variables n'étant pas initialisée, nous n'avons pas besoin de leur affecter des valeurs particulières. En réalité, la valeur 0 leur est systématiquement affectée.

```
**
 * @brief  This is the code that gets called when the processor first
 *          starts execution following a reset event. Only the absolutely
 *          necessary set is performed, after which the application
 *          supplied main() routine is called.
 * @param  None
 * @retval : None
*/

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

/* Copy the data segment initializers from flash to SRAM */
  movs  r1, #0            // r1 = 0
  b  LoopCopyDataInit     // Jump à l'étiquette LoopCopyDataInit

CopyDataInit:
  ldr  r3, =_sidata       // r3 = _sidata (@ des valeurs en Flash)
  ldr  r3, [r3, r1]       // r3 = *(r3 + r1)
  str  r3, [r0, r1]       // *(r0 + r1) = r3
  adds  r1, r1, #4        // r1 = r1 + 4

LoopCopyDataInit:
  ldr  r0, =_sdata        // _sdata (start @ des variables .data) dans r0
  ldr  r3, =_edata        // _edata (end @ des variables .data) dans r3
  adds  r2, r0, r1        // r2 = r0 + r1
  cmp  r2, r3             // Compare r2 et r3
  bcc  CopyDataInit       // Si r2 < r3 alors on va dans à CopyDataInit
  ldr  r2, =_sbss         // Sinon r2 = _sbss (start @ des variables .bss)
  b  LoopFillZerobss      // Jump à l'étiquette LoopFillZerobss
/* Zero fill the bss segment. */
FillZerobss:
  movs  r3, #0            // r3 = 0
  str  r3, [r2], #4       // *r2 + 4 = r3
```

```
LoopFillZerobss:
  ldr  r3, = _ebss        // r3 = _ebss (end @ des variables .bss)
  cmp  r2, r3             // Compare r2 et r3
  bcc  FillZerobss        // Si r2 < r3 alors on va à FillZerobss

/* Call the clock system intitialization function.*/
  bl  SystemInit
/* Call static constructors */
    bl __libc_init_array
/* Call the application's entry point.*/
  bl  main
  bx  lr
```

A partir de ce fichier, on peut résumer le déroulement du démarrage d'un microcontrôleur STM32 à l'aide de l'organigramme simplifié suivant :

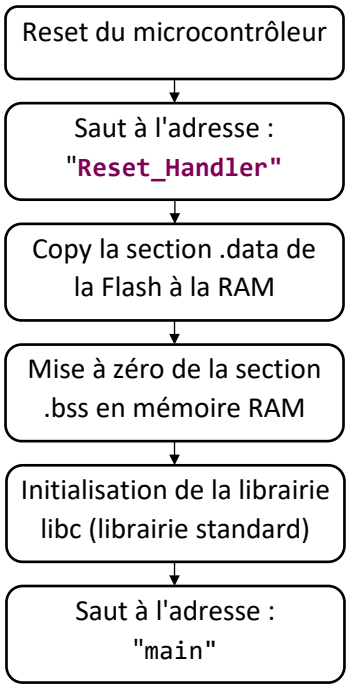


*Figure 5 : Séquence de Reset d'un microcontrôleur STM32*

- Compléter les tableaux pour vérifier les placements des données en RAM et les initialisations des variables *.data*.

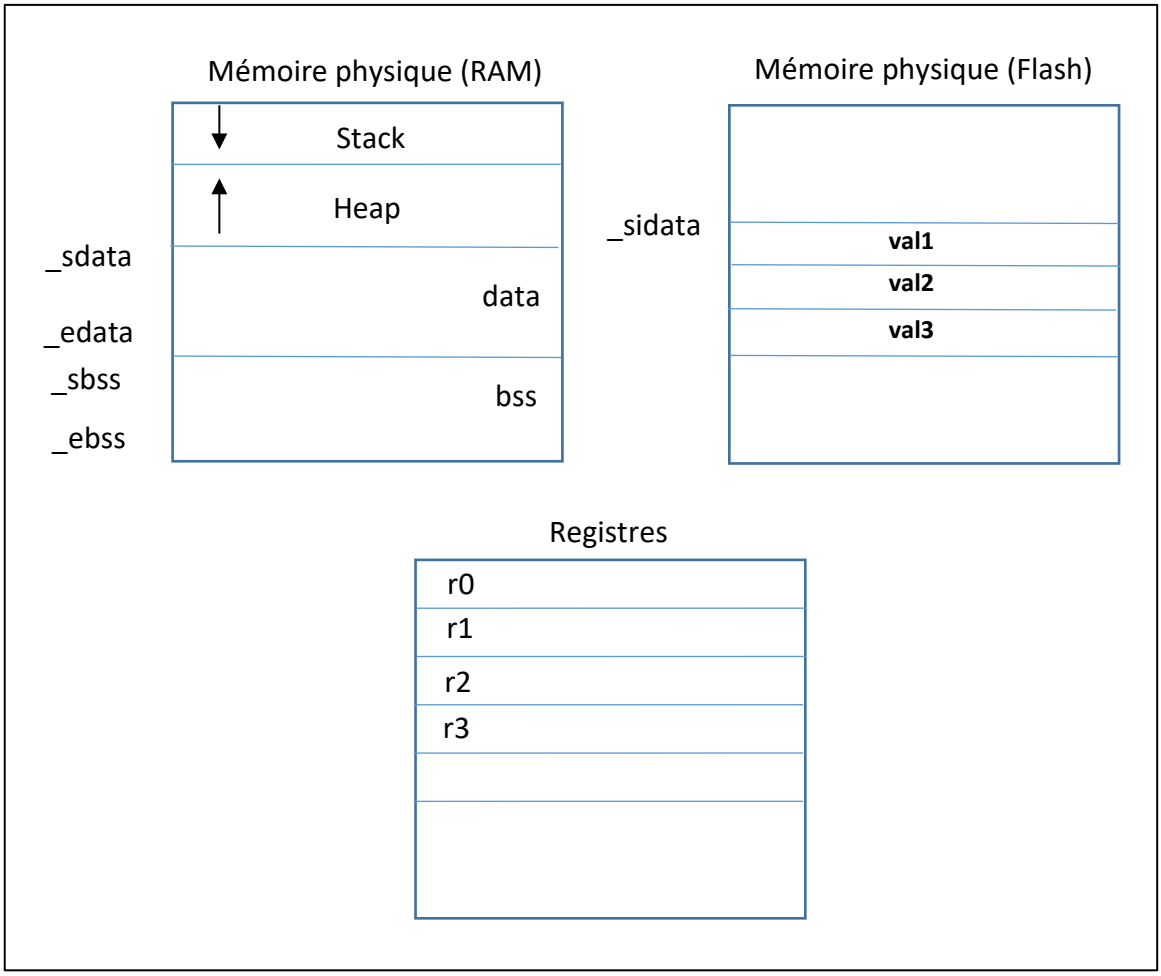


*Figure 6 : Initialisation des segments .data et .bss en RAM*

# 3 La programmation multi-fichiers

Au-delà d'une certaine taille, une application en langage C est décomposée en plusieurs fichiers. Chaque fichier sera représentatif d'une fonctionnalité de l'application. Il est préférable que ces fonctionnalités soient les plus indépendantes les unes des autres.

Les règles de la programmation multi-fichiers sont les suivantes :

1. Tous les *fichier.c* doivent inclure un *fichier.h* du même nom inclus (voir *#include*).
2. Tous les *fichier.h* doit être protégé contre les inclusions multiples (voir *#ifndef* et *#define*).
3. Les variables globales utilisées dans un autre fichier doivent avoir la classe *extern*.
4. Les variables globales ou les fonctions dont on souhaite interdire l'externalisation doivent être *static*.

Nous allons faire un exemple de programmation multi-fichiers à l'aide d'un projet complet.

## 3.1 Utilisation des directives préprocesseurs

L'utilisation de ces directives permettent de répondre aux règles (1)et (2) ci-dessus. Les directives préprocesseurs sont étudiées avant la compilation. Elles sont toutes précédées du symbole #.

### 3.1.1 Inclure des fichiers : *#include*

Permet d'inclure le contenu (**copier / coller**) d'un autre fichier. Afin d'éviter les inclusions multiples, les fichiers qui sont inclus doit être protégés (voir *#ifndef* au paragraphe 3.1.3).

- ***#include <fichier.h>* :** Inclus un fichier dont le chemin est déjà connu par le système (*stdio.h*, *stdint.h*, etc…)
- ***#include "dossier/fichier.h"* :** Inclus un fichier donc vous donner vous-même le chemin relatif.

### 3.1.2 Définir des macros : *#define*

Permet de définir des macros qui seront remplacées afin la compilation. Par convention, on écrit les macros en majuscule. Par exemple : *#define DEBUG 0*

On peut aussi passer des paramètres à nos macros :

```
#define PRINT(x)  printf("%d",x);
```

L'utilisation de *PRINT(10)* dans le code affichera 10.

### 3.1.3 Définir des macros sous condition : *#if*

Les directives conditionnelles sont *#if, #ifdef, #ifndef, #if defined(), #if !defined(), #elif, #else, #endif*. Elles permettent de réaliser des actions préprocesseurs sous condition.

La différence entre *#ifdef* et *#if defined()* est que *#ifdef* ne peut recevoir qu'une condition unique alors que *#if defined()* peut utiliser une combinaison de condition (*ET* / *OU*).

Par exemple, le code *printf("Mode Debug");* sera compilé que si *DEBUG* a déjà été défini :

```
#ifdef DEBUG
  printf("Mode Debug");
#endif
```

Ici, le *printf* sera compilé que si *DEBUG* et *DEBUG_SERIAL* sont définis :

```
#if defined(DEBUG) && defined(DEBUG_SERIAL)
```

```
  printf("Mode DEBUG sur liaison serie");
#endif
```

Ici, le printf est affiché que si *DEBUG = 0* :

```
#if DEBUG == 0
  printf("Mode Debug désactivé");
#endif
```

La macro *DEBUG* sera définie que si *DEBUG* n'était pas déjà définie :

```
#ifndef DEBUG
  #define DEBUG
#endif
```

## 3.2 Utilisation de variables définies dans un autre fichier

Cela permet de répondre à la règle (3) énoncée au début de ce paragraphe.

Lorsque qu'un projet complet est compilé, le compilateur opère fichier après fichier. Il a besoin de connaitre l'existence de toutes les variables. Si un *fichierA.c* utilise une variable qui a été définie dans un *fichierB.c*, alors il faudra le préciser explicitement au compilateur en utilisant la classe "extern".

fichierA.c

```
int i = 10;

void main (void){
i = 5;
fonction();
}
```

fichierB.c

```
extern int i;

void fonction (void){
i ++;
}
```

- Le *fichierA.c* définit la variable et un espace mémoire lui est réservé.
- Le *ficherB.c* déclare simplement l'existence de la variable i. Il n'y aura pas de nouvel emplacement mémoire qui lui sera attribué car cela a déjà été fait dans le *fichierA.c*.
- L'initialisation ( *i = 10* ) ne doit pas être présente lors de la déclaration en *extern*, car encore une fois, on ne définit pas une nouvelle variable, mais on utilise bien la même que celle déjà définie dans le *fichierA.c*.

## 3.3 Interdiction de l'externalisation

Cela permet de répondre à la règle (4) énoncée au début de ce paragraphe.

Si les fonctions ou les variables globales définies dans votre fichier ne peuvent pas être sujettes à l'externalisation, c'est-à-dire ne peut pas être utilisé dans un autre fichier, alors celle-ci peuvent prendre la classe *static*. C'est par exemple le cas des fonctions d'initialisation des périphériques lors de la génération du code par CubeMX :

```
static void MX_GPIO_Init(void);
static void MX_I2C1_Init(void);
static void MX_SPI2_Init(void);
```

Dans le cas des fonctions d'initialisation des périphériques STM32, la classe *static* peut être supprimée lors de la configuration du projet. Le programmeur est donc le seul à décider si la classe *static* est pertinente dans son cas.

| Function Name | IP Instance Name | ☐ Visibility (Static) |
|---|---|---|
| MX_GPIO_Init | GPIO | ☑ |
| MX_DMA_Init | DMA | ☑ |
| SystemClock_Config | RCC | ☐ |
| MX_USART2_UART_Init | USART2 | ☑ |
| MX_I2C1_Init | I2C1 | ☑ |
| MX_SPI2_Init | SPI2 | ☑ |

*Figure 7 : Configuration de CubeMX pour la visibilité (static ou non) des fonctions d'initialisation*

# 4 Lecture et modification des registres d'un microcontrôleur

Un registre est une case mémoire RAM du microcontrôleur. Pour y accéder, il suffit de connaitre son adresse. L'environnement de développement sur lequel on développe nous fournit très souvent des fichiers (*#include*) répertoriant toutes les adresses de tous les registres. Il suffit donc d'y faire référence en utilisant simplement le nom du registre.

Dans le paragraphe suivant, nous considérons que REG est la valeur du registre sur lequel nous souhaitons réaliser des opérations. Ces opérations ne sont pas exclusives aux registres mais peuvent être, bien sûr réalisée sur de simples variables.

## 4.1 Opération sur les registres

Les opérations sur les bits d'un registre doivent être réalisées sans modifier les autres bits du registre. On utilise pour cela les opérateurs *OU*, *ET* et *OU* exclusif.

### 4.1.1 Mettre un bit à 1

Le positionnement d'un bit à 1 se fait à l'aide de l'opérateur logique *OR*.

Exemple :

```
REG = REG | 0x01 // Set bit0 to 1.
REG = REG | 0x81 // Set bit0 and bit7 to 1.
```

Ecrire une macro *MACRO_SET_BIT( REG , BIT)* qui met à 1 le bit numéro *BIT* du registre *REG* :

### 4.1.2 Mettre un bit à 0

Le positionnement d'un bit à 0 se fait à l'aide de l'opérateur logique *ET*.

Exemple :

```
REG = REG & 0xFE // Set bit0 to 0.
REG = REG & 0x7E // Set bit0 and bit7 to 0
```

Ecrire une macro *MACRO_CLEAR_BIT( REG , BIT°)* qui met à 0 le bit numéro *BIT* du registre *REG*.

### 4.1.3 Toggle un bit

La modification d'un bit par son contraire (toggle) d'un bit se fait à l'aide de l'opérateur logique *XOR*.

Exemple :

```
REG = REG ^ 0x01 // Negate bit0
REG = REG ^ 0x81 // Negate bit0 and bit7
.
```

Ecrire une macro *MACRO_TOGGLE_BIT( REG , BIT°)* qui inverse le bit numéro *BIT* du registre *REG*.

### 4.1.4 Lire une valeur dans un champ binaire d'un registre

Cette opération consiste à lire une valeur sur plusieurs bits d'une partie d'un registre. On considère par exemple que nous souhaitions lire le champ *ST[2:0]* du registre suivant :

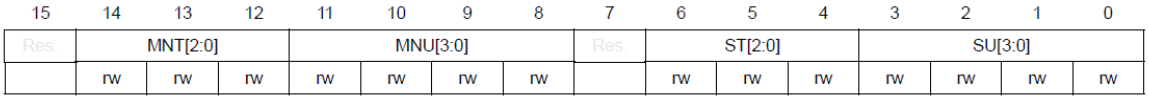


*Figure 8 : 16 bits de poids faible du registre de la RTC d'un STM32F446*

Il y a plusieurs façons possibles de procéder, nous allons en étudier une.

Nous aurons besoin :

- Du registre à lire
- Du masque représentant la taille du champ à lire (ici 0x0007).
- De la position du 1$^{er}$ bit du champ (ici 4).

Ecrire une macro *MACRO_READ_FIELD(REG , MASK , POS)* permettant de lire la valeur d'un champ binaire.

### 4.1.5 Ecrire une valeur dans un champ binaire d'un registre

Cette opération consiste à écrire une valeur sur plusieurs bits d'une partie d'un registre. On considère par exemple que nous souhaitions écrire le champ ST[2:0] du registre suivant :

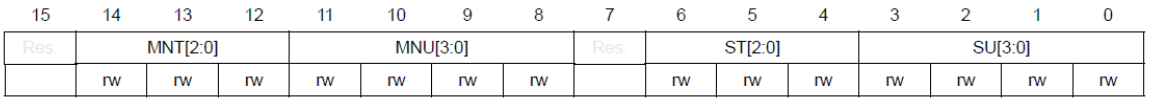


*Figure 9 : 16 bits de poids faible du registre de la RTC d'un STM32F446*

Il y a plusieurs façons possibles de procéder, nous allons en étudier une.

Nous aurons besoin :

- Du registre à écrire
- De la valeur du champ binaire à écrire
- Du masque représentant l'ensemble du champ (ici 0x0007)
- De la position du 1$^{er}$ bit (ici 4)

➡ Ecrire une macro MACRO_WRITE_FIELD(REG , VAL , MASK , POS) permettant d'écrire la valeur d'un champ binaire.

```
#define MACRO_WRITE_FIELD( REG , VAL, MASK , POS)
REG=(REG & ((~MASK)<<POS)) | (VAL<<POS)
```

### 4.1.6 Macros fournies par ST

ST fourni un jeu de macro légèrement différent mais permettant de réaliser l'ensemble des opérations que nous venons de voir (sauf TOGGLE). Celles-ci sont données dans les fichiers *stm32f4xx.h* et sont réécrites ci-dessous :

Mettre un bit à 1, mettre un bit à 0, lire un bit :

```
#define SET_BIT(REG, BIT)     ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT)   ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT)    ((REG) & (BIT))
```

Dans les *MACRO* de ST, *BIT* prends les valeurs 1, 2, 4, … , 64, 128 … Il représente donc l'emplacement du bit déjà correctement positionné.

Effacer registre, écrire registre, lire registre, modifier une partie d'un registre :

```
#define CLEAR_REG(REG)        ((REG) = (0x0))
#define WRITE_REG(REG, VAL)   ((REG) = (VAL))
#define READ_REG(REG)         ((REG))
#define MODIFY_REG(REG, CLEARMASK, SETMASK)  WRITE_REG((REG),
(((READ_REG(REG)) & (~(CLEARMASK))) | (SETMASK)))
```

ST donne la valeur des masques et des positions du 1er bits du champ pour tous les champs de tous les registres du microcontrôleur dans le fichier *stm32f446.h*. Par exemple, pour le champs ST de du registre de la RTC, ces informations sont données de la façon suivante :

```
#define RTC_TR_ST_Pos    (4U)
#define RTC_TR_ST_Msk    (0x7UL << RTC_TR_ST_Pos)
```

Dans la dernière macro MODIFY_REG :

- *CLEARMASK* est la valeur du masque déjà positionnée convenablement dans le registre.
- *SETMASK* est la valeur à écrire déjà positionnée convenablement dans le registre.

**Nos macros sont donc sensiblement identiques à celles fournies par ST.**

## 4.2 L'organisation mémoire d'un microcontrôleur STM32

Dans les architecture ARM Cortex M, la mémoire programme, la mémoire donnée, les registres et les périphériques d'entrée/sortie sont accessibles par un unique plan d'adressage de 4 Go divisé en 8 blocs de 512 Mo chacun. Pour adresser une place de 4 Go, cela demande 32 bits d'adresse ($2^{32}$ = 4 G). Les adresses seront donc toujours sur 32 bits lorsque nous utiliserons des variables de type *pointeur*.

Lorsque nous souhaitons écrire une donnée en mémoire, il faut connaitre l'adresse exacte que nous souhaitons cibler. Cela ne peut se faire qu'en consultant le *Reference Manual* de votre microcontrôleur.

## 4.3 Représentation de la mémoire en langage C

### 4.3.1 Les pointeurs

En langage C, les adresses sont représentées par des pointeurs. A l'aide des adresses, on peut réaliser toutes les opérations sur une zone mémoire. Si on connait l'adresse d'un registre on peut donc lire ou écrire une valeur.

```
uint32_t* ptrReg = 0xXXXX;   // ptrReg est l'@ d'un registre
*ptrReg = valeur;            // Le registre prend une valeur
```

Comment connaitre l'adresse d'un registre ? Pour cela nous prendrons l'exemple de la manipulation des registres gérant les *ports I/O*. On se réfère donc au *Reference Manual* dans la section *Memory Map description*.

| | | | |
|---|---|---|---|
| 0x4002 1C00 - 0x4002 1FFF | GPIOH | AHB1 | Section 7.4.11: GPIO register map on page 193 |
| 0x4002 1800 - 0x4002 1BFF | GPIOG | | |
| 0x4002 1400 - 0x4002 17FF | GPIOF | | |
| 0x4002 1000 - 0x4002 13FF | GPIOE | | |
| 0x4002 0C00 - 0x4002 0FFF | GPIOD | | |
| 0x4002 0800 - 0x4002 0BFF | GPIOC | | |
| 0x4002 0400 - 0x4002 07FF | GPIOB | | |
| 0x4002 0000 - 0x4002 03FF | GPIOA | | |

*Figure 10 : Adresse des périphériques GPIOX dans un STM32F446*

On note alors que chaque *GPIOA* à une zone mémoire qui lui est affectée. Pour les *GPIO* du port A on retrouvera tous les registres permettant de le configurer aux adresses *0x4002 0000* à *0x4002 03FF*. On remarque par ailleurs, que tous les *GPIOx* ont une zone similaire pour leurs registres respectifs (*GPIOB*, *GPIOC*, …, *GPIOH*). Si on regarde la documentation on s'aperçoit que toutes ces zones sont organisées avec la même succession de registres dont une partie est répertoriée dans le tableau suivant :

| Offset | Registre | Rôle |
|---|---|---|
| 0x00 | GPIO**X**_MODER | Choix Input / Output / Alternate / Analog |
| 0x04 | GPIO**X**-OTYPER | Output Push-pull / Ouptut Open Drain |
| 0x08 | GPIO**X**_OSPEEDER | Low / Medium / Fast / High Speed |
| 0x0C | GPIO**X**_PUPDR | Pull-up / Pull-Down / Nothing |
| 0x10 | GPIO**X**_IDR | Input values |
| 0x14 | GPIO**X**_ODR | Output values |
| 0x18 | GPIO**X**_BSRR | Bit Set / Bit Reset |
| 0x1C | GPIO**X**_LCKR | Lock value on GPIO |
| 0x20 | GPIO**X**_AFRL | Alternate Function Low |
| 0x24 | GPIO**X**_AFRH | Alternate Function High |

*Tableau 12 : Les offsets à appliquer sur l'adresse de base des registres*

Chaque *GPIO* possède donc l'ensemble de ces registres, où X correspond à la lettre du *GPIOX*. La colonne offset à très importante car elle nous donne l'adresse exacte du registre dans le plan mémoire. Un registre a donc pour adresse finale :

```
@ d'un registre GPIOX_XXX = @ de base du GPIOX + offset
```

Par exemple, dans le cas du *GPIOA*, le registre *GPIOA_PUPDR* a pour adresse : *0x4002 000C*. Donc si nous souhaitons configurer le registre *GPIOA_PUPDR* (modifier les résistances de pull-up, pull-down du *GPIOA*), on pourra écrire à cette adresse à l'aide d'une instruction d'affectation :

```
*(0x4002000C) = valeur;
```

Mais cette méthode à de forte chance de donner une erreur (ou un warning au minimum) car on ne précise pas que *0x4002000C* est une adresse. On peut donc avantageusement utiliser une conversion explicite pour signaler le type *(uint32_t*)* de *0x4002000C* au compilateur :

```
*(uint32_t*)0x4002000C = valeur;
```

Bien que cette méthode fonctionne parfaitement, on se rend compte très rapidement que la maintenance de ce code devient difficile car il faut se souvenir des adresses de tous les registres que

nous souhaitons manipuler. On peut donc prendre soin de stocker la valeur de l'adresse du registre dans une variable plus explicite avant de l'utiliser :

```
uint32_t* ptr_GPIO_PUPDR = (uint32_t*) 0x4002000C;
*ptr_GPIO_PUPDR = valeur;
```

Exercice : Ecrire le code pour réaliser un clignotement de la LED PA5 (registre GPIOA_ODR) sur la carte Nucleo toutes les 1000ms en considérant que toutes les initialisations (choix input/output, choix pull-up / pull-down, …) ont déjà été faites par CubeMX.

### 4.3.2 Constante

Une constante peut être déclaré comme telle en utilisant le qualificateur *const* juste après le type de la variable déclarée. Une constante sera stockée en mémoire FLASH et ne pourra pas être modifiée. Le compilateur vérifiera que l'élément déclaré comme *const* ne soit jamais affecté SAUF lors de l'initialisation.

```
uint32_t const var = 10;     // var est un entier constant
var = 11;                    // Erreur de compilation
```

La déclaration de *var* peut aussi être faite de la façon suivante, mais on préférera toujours la méthode ci-dessus pour des soucis de lisibilité :

```
const uint32_t var = 10;     // var est un entier constant
```

Le qualificateur *const* peut aussi être appliqué aux pointeurs. La position du mot *const* est alors très importante :

*ptrVar1* est un pointeur constant qui pointe sur une donnée variable (pouvant être modifiée). C'est le cas par exemple des registres d'un microcontrôleur accessibles en lecture / écriture dont les adresses sont fixes :

```
uint32_t* const ptrVar1;
ptrVar1 = valeur;            // Erreur de compilation
*ptrVar1 = valeur;           // OK
```

*ptrVar2* est un pointeur variable (pouvant être modifié), pointant sur une constante (ne pouvant pas être modifiée). C'est le cas par exemple de l'argument de la fonction printf [ *int printf(const char *format, ...)* ] qui considère que la chaine de caractère qui lui est passée en paramètre ne doit pas être modifiable dans la fonction. Voici un autre exemple :

```
uint32_t const * ptrVar2;
ptrVar2 = valeur;            // OK
*ptrVar2 = valeur;           // Erreur de compilation
```

**ptrVar3** est un pointeur constant (ne pouvant pas être modifié) pointant sur une constante (ne pouvant pas être modifiée). Nous verrons un exemple en fin de chapitre.

```
uint32_t const * const ptrVar3;
ptrVar3 = valeur;           // Erreur de compilation
*ptrVar3 = valeur;          // Erreur de compilation
```

On rappelle alors l'importance de l'emplacement du mot *const*. Le qualificateur *const* est toujours placé à droite de l'élément constant. Il est donc plus aisé de lire de droite à gauche afin de savoir ce qui est constant ou pas.

Dans le cadre de ce chapitre, nous nous intéressons à la lecture / écriture des registres. Nous avions défini le code suivant permettant de modifier le registre *GPIO_PUPDR* :

```
uint32_t* ptr_GPIO_PUPDR = (uint32_t*) 0x4002000C;
*ptr_GPIO_PUPDR = valeur;
```

ptr_GPIO_PUPDR est l'adresse d'un registre. Celui-ci est fixe et ne peut pas être modifié. Il a été fixé par les concepteurs du microcontrôleur. Nous pouvons donc améliorer le code précédent en ajoutant le qualificateur *const* afin de prévenir toute modification de son adresse.

```
uint32_t* const ptr_GPIO_PUPDR = (uint32_t*) 0x4002000C;
*ptr_GPIO_PUPDR = valeur;    // OK
ptr_GPIO_PUPDR = valeur;     // Erreur de compilation
```

Si le registre avec lequel nous travaillons est un registre accessible uniquement en lecture (écriture interdite ou sans effet) alors on peut utiliser la notation suivante pour prévenir toute modification de son adresse et de son contenu :

```
uint32_t const * const ptrReg_XXX = (uint32_t*) 0xXXXXXXXX;
*ptrReg_XXX = valeur;        // Erreur de compilation
 ptrReg_XXX = valeur;        // Erreur de compilation
```

Exercice : Donner le code d'une application qui allume la *LED PA5* (bit 5 registre *ODR* pour le *PORTA* dont l'adresse est *0x40020014*) en fonction de la valeur du bouton poussoir *PC13* (bit 13 registre *IDR* pour le *PORTC* dont l'adresse est *0x40020810*).

### 4.3.3 Volatile

Un programmeur souhaite réaliser une application simple : Une variable est mise à 1 si le bit 0 du port *GPIOA* est à 1. Cette variable est mise à 0 si le bit 0 du *GPIOA* est à 0. Le programmeur réalise donc le code suivant :

```
uint32_t var;
uint32_t const * const ptr_GPIOA_IDR = (uint32_t*) 0x40020010;

int main(void){
  while(1){
  var = (*ptr_GPIOA_IDR) & 0x0001;
  }
}
```

Ce code fonctionne parfaitement. Le programmeur décide donc d'augmenter le degré d'optimisation du code alors que jusqu'à maintenant nous étions au niveau *None (-O0)*. En effet, nous souhaitons maintenant améliorer la taille prise en mémoire ou la rapidité d'exécution du code.

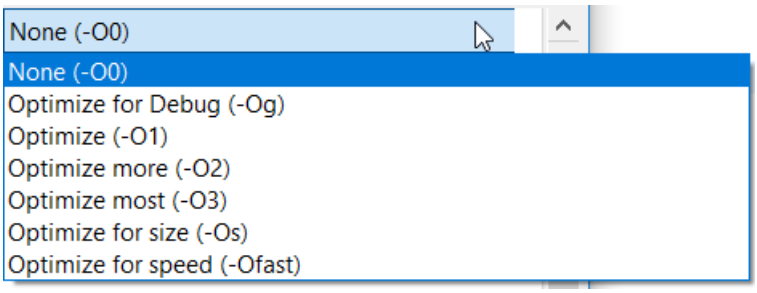


*Figure 11 : Les différents degrés d'optimisation du code*

Le code ne fonctionne plus. D'où peut venir le problème ?

On peut se douter qu'il s'agit d'un problème d'optimisation du code puisque c'est son activation qui déclenche le défaut. En effet, comme nous ne modifions jamais le contenu du registre *ptr_GPIOA_IDR*, le compilateur estime qu'une unique lecture de ce registre lui permet de connaitre sa valeur. Il considère la relecture du registre inutile et va donc remplacer le code initial par l'optimisation suivante :

```
uint32_t var;
uint32_t const * const ptr_GPIOA_IDR = (uint32_t*) 0x40020010;

int main(void){
  var = (*ptr_GPIOA_IDR) & 0x0001;
}
```

On s'aperçoit que la boucle *while(1)* à disparue car le compilateur ne voit pas l'intérêt de tester une expression qui (selon lui) sera toujours équivalente. En réalité, le registre *ptr_GPIOA_IDR* sera bien modifié par le périphérique lui-même, mais le compilateur ne peut pas le savoir. Il faut donc trouver un moyen de dire au compilateur : "*Cette variable ne doit pas être optimisée car d'autres éléments que mon code l'utilise*". Cela peut être fait avec le qualificateur *volatile*. Le code initial sera donc le suivant :

```
uint32_t var;
uint32_t volatile const * const ptr_GPIOA_IDR=(uint32_t*)0x40020010;

int main(void){
  while(1){
  var = (*ptr_GPIOA_IDR) & 0x0001;
  }
}
```

*ptr_GPIOA_IDR* est un pointeur constant, qui pointe vers une données constante (qu'on ne peut pas modifier), mais *volatile* (que le processeur lui-même peut modifier). La lecture de droite à gauche est une nouvelle fois préconiser pour comprendre les affectations des qualificateur *const* et *volatile*.

Enfin, le qualificateur *volatile* pourrait aussi être appliquer au pointeur lui-même. Ci-dessous, ptrVar est un pointeur volatile qui pointe var une donnée non volatile. Mais cette représentation n'est que très rarement utilisée et je n'ai pas d'exemple de cas concret. Néanmoins le code serait le suivant :

```
uint32_t  * volatile ptrVar;
```

## 4.4 Simplifier l'accès aux bits des registres

Les opérations effectuées sur les registres vues au chapitre 4.3 nous permettent de modifier ou de lire certains bits. Cette méthode est la plus couramment utilisée, mais elle souffre d'un manque de lisibilité car on ne nomme pas explicitement les bits représentés. Une solution est d'utiliser des

*MACRO* avec des *#define* avec le nom du bit. Nous avons (en partie) utilisé cette méthode au chapitre 4.1.

Dans ce chapitre, verrons tout d'abord l'utilisation générale des structures en C, puis nous verrons au chapitre 4.4.2 comment améliorer l'accès aux bits des registres en utilisant les *bits field*.

### 4.4.1 Utilisation générale des structures

Une structure peut être considérée comme un tableau, à la seule différence, c'est que un type de variable différent peut être utilisé pour chaque case du tableau.

```
struct exemple {                    // Déclaration
       uint8_t  champ0;
       uint16_t champ1;
       uint32_t champ2;
};

struct exemple uneStructure;        // Définition
```

L'accès aux champs de la structure se faire de la façon suivante :

```
uneStructure.champ1 = 0x11;
uneStructure.champ2 = 0x2122;
uneStructure.champ3 = 0x31323334;
```

Calculons la taille de la structure *exemple* : **uint8_t (1) + uint16_t (2) + uint32 (4) = 7 octets**. En réalité, lorsque nous travaillons sur STM32F446, la fonction *sizeof(struct exemple)* nous retourne 8 octets. D'où vient la différence ?

Un microcontrôleur ne stockera pas systématiquement les données de façon contiguës, car pour conserver ses performances, il a besoin qu'elles soient alignées :

- Sur des adresses multiples de 4 pour des mots de 32 bits : 0x00, 0x04, 0x08, …
- Sur des adresses multiples de 2 pour des mots de 16 bits : 0x00, 0x02, 0x04, …
- Sur n'importe quelle adresse pour des mots de 8 bits : 0x00, 0x01, 0x02, …

Dans notre cas, le remplissage se fait donc de la façon suivante :

| @ de la structure | Champ | valeur | Alignement |
|---|---|---|---|
| 0x00 | champ1 | 0x11 | **OK** |
| 0x01 | vide | | |
| 0x02 | champ2 | 0x22 | **OK** |
| 0x03 | champ2 | 0x21 | **OK** |
| 0x04 | champ3 | 0x34 | **OK** |
| 0x05 | champ3 | 0x33 | **OK** |
| 0x06 | champ3 | 0x32 | **OK** |
| 0x07 | champ3 | 0x31 | **OK** |

*Tableau 13 : Remplissage de la structure exemple avec alignement.*

Nous remarquons donc qu'il y a un espace de perdu à l'adresse 0x01. Si nous souhaitons optimiser la place malgré le risque de non alignement des données et donc de perdre en performance, il est possible de le faire en le spécifiant avec l'attribut : `__attribute__((packed)).`

```
struct exemple {                    // Déclaration
       uint8_t  champ0;
       uint16_t champ1;
       uint32_t champ2;
}__attribute__((packed));
```

Le remplissage se fait donc de la façon suivante pour un total de 7 octets :

| @ de la structure | Champ | valeur | Alignement |
|---|---|---|---|
| 0x00 | champ1 | 0x11 | **OK** |
| 0x01 | champ2 | 0x22 | **NON** |
| 0x02 | champ2 | 0x21 | **NON** |
| 0x03 | champ3 | 0x34 | **NON** |
| 0x04 | champ3 | 0x33 | **NON** |
| 0x05 | champ3 | 0x32 | **NON** |
| 0x06 | champ3 | 0x31 | **NON** |

*Tableau 14 : Remplissage de la structure exemple sans alignement.*

### 4.4.2 Accéder aux bits grâce aux "bits field" d'une structure

L'utilisation d'une structure peut aussi faciliter l'accès aux bits en spécifiant le nombre de bits utilisés par chaque champ. On souhaite par exemple représenter le registre *RTC_TR* à l'aide d'une structure. Dans la documentation, on relève le format du registre *RTC_TR* qui est représenté ci-dessous. On relève aussi l'adresse de base du périphérique *RTC* : ***0x4000 2800***.

### 22.6.1 RTC time register (RTC_TR)

The RTC_TR is the calendar time shadow register. Address offset: 0x00

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | PM | HT[1:0] | | HU[3:0] | | | |
| | | | | | | | | | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Res. | MNT[2:0] | | | MNU[3:0] | | | | Res. | ST[2:0] | | | SU[3:0] | | | |
| | rw | rw | rw | rw | rw | rw | rw | | rw | rw | rw | rw | rw | rw | rw |

Bit 22 **PM**: AM/PM notation
0: AM or 24-hour format
1: PM

Bits 21:20 **HT[1:0]**: Hour tens in BCD format

Bits 19:16 **HU[3:0]**: Hour units in BCD format

Bit 15 Reserved, must be kept at reset value.

Bits 14:12 **MNT[2:0]**: Minute tens in BCD format

Bits 11:8 **MNU[3:0]**: Minute units in BCD format

Bit 7 Reserved, must be kept at reset value.

Bits 6:4 **ST[2:0]**: Second tens in BCD format

Bits 3:0 **SU[3:0]**: Second units in BCD format

*Figure 12 : Registre RTC_TR (RTC Time Register) d'un STM32F446*

Ce registre est un mot unique de 32 bits dont l'adresse est *0x4000 2800 + offset = 0x4000 2800*. On aimerait concevoir des champs individualisés pour chacun des éléments (*SU[3:0]*, *ST[2:0]*, *MNU[3:0]*,…). La réalisation avec des structures organisées en *bits field* est la suivante :

```
struct struct_RTC_TR {              // Declaration
  uint32_t volatile SU:4;           // Second Units (BCD)
  uint32_t volatile ST:3;           // Second Tens  (BCD)
  uint32_t volatile reserved0:1;    // Reserved
  uint32_t volatile MNU:4;          // Minute Units (BCD)
  uint32_t volatile MNT:3;          // Minute Tens (BCD)
  uint32_t volatile reserved1:1;    // Reserved
  uint32_t volatile HU:4;           // Hour Units (BCD)
  uint32_t volatile HT:2;           // Hour Tens (BCD)
  uint32_t volatile PM:1;           // AM/PM notation
  uint32_t volatile reserved2:9;    // Reserved
  };

struct struct_RTC_TR volatile * const regBitField_RTC_TR = (struct
struct_RTC_TR volatile * const) 0x40002800;     // Definition
```

L'affectation se fait donc directement via les champs de la structure. Par exemple, si on souhaite initialiser l'horloge temps réel à 10h11, nous rentrerons les informations suivantes :

```
regBitField_RTC_TR->HT = 1;
regBitField_RTC_TR->HU = 0;
regBitField_RTC_TR->MNT = 1;
regBitField_RTC_TR->MNU = 1;
```

### 4.4.3 Accéder aux bits grâce aux *bits field* d'une union

Une limitation de la représentation des registres par des structures sous formes de *bits field* est qu'il n'est pas possible simplement d'écrire (ou lire) la valeur complète sur 32 bits du registre. On est obligé de réaliser l'écriture (ou la lecture) individuelle de chaque champ. Cela est contraignant.

Les unions sont similaires à des structures, sauf que chaque champ fait référence à un unique emplacement mémoire. La taille d'une union est donc celle de son champ le plus grand. On reprend l'exemple vu précédemment au chapitre 4.4.1 sur les structures :

```
struct exemple {                    // Déclaration
       uint8_t  champ0;
       uint16_t champ1;
       uint32_t champ2;
};

struct exemple uneStructure;        // Définition
```

Cette structure *exemple* possède 3 champs disposés les uns à la suite des autres : champ0, champ1, champ2. Si on utilise maintenant une union :

```
union exemple {                     // Déclaration
       uint8_t  champ0;
       uint16_t champ1;
       uint32_t champ2;
};

union exemple uneUnion;             // Définition
```

Cette union *exemple* ne possède qu'un seul champ dont la taille est celle de son champ le plus grand (*uint32_t = 4 octets*). On peut donc modifier :

- les 4 octets d'un seul coup si on utilise le *champ2*,
- seulement les deux premiers octets si on utilise le *champ1*,
- seulement le premier octet si on utilise le *champ0*.

```
uneUnion.champ2 = 0x31323334;       // Ecriture des 4 octets
uneUnion.champ1 = 0x2122;           // Ecriture de 2 octets
uneUnion.champ0 = 0x11;             // Ecriture d'un octet
```

A l'issu de ce code, la valeur stockée dans cette *union* est *0x31 32 21 11*

On reprend l'exemple du paragraphe 4.4.2 où nous décrivions le registre RTC_TR à l'aide d'un structure *bit field*. Nous allons intégrer cela sous la forme d'une union. Cette union sera composée d'une structure *bit field* pour l'accès aux différents bits, **et** d'un champ sur 32 bits pour l'accès direct au registre complet. Le code en C déclare donc la structure et l'union, puis définie un pointeur sur cette union. Ce pointeur devra être initialisé à l'adresse du registre correspondant : *0x40002800* dans notre cas.

```
    /* 1. Declaration de la structure et de l'union */
struct struct_RTC_TR {                  // Declaration
      uint32_t volatile SU:4;           // Second Units (BCD)
      uint32_t volatile ST:3;           // Second Tens  (BCD)
      uint32_t volatile reserved0:1;
      uint32_t volatile MNU:4;          // Minute Units (BCD)
      uint32_t volatile MNT:3;          // Minute Tens (BCD)
      uint32_t volatile reserved1:1;
      uint32_t volatile HU:4;           // Hour Units (BCD)
      uint32_t volatile HT:2;           // Hour Tens (BCD)
      uint32_t volatile PM:1;           // AM/PM notation
      uint32_t volatile reserved2:9;
      };

union union_RTC_TR {              // Declaration de l'union
      uint32_t volatile RTC_TR;  // RTC_TR accessible sur 32 bits
      struct struct_RTC_TR regBitField_RTC_TR; // Accessible par bit
};


    /* 2.Definition et initialisation du pointeur sur le registre */
union union_RTC_TR volatile * const reg_RTC_TR = (union union_RTC_TR
volatile * const) 0x40002800;
```

L'affectation se fait donc directement via les champs de la structure **ou** par le mot de 32 bits directement. Par exemple, si on souhaite initialiser l'horloge temps réel à 10h11, nous rentrerons les informations suivantes :

```
    reg_RTC_TR->regBitField_RTC_TR.HT = 1;       // Dizaine heure
    reg_RTC_TR->regBitField_RTC_TR.HU = 0;       // Unité heure
    reg_RTC_TR->regBitField_RTC_TR.MNT = 1;      // Dizaine minute
    reg_RTC_TR->regBitField_RTC_TR.MNU = 1;      // Unité minute
```

Si on souhaite configurer directement l'ensemble du registre directement, on écrira :

```
    reg_RTC_TR->RTC_TR = 0x00101100;
```

Mais attention, dans ce cas on touche aussi aux autres champs du registre.

### 4.4.4 Simplifier l'écriture des structures et des unions : Typedef

Si on regarde l'écriture des pointeurs sur nos structures (paragraphe 4.4.2) et sur nos unions (paragraphe 4.4.3), on peut à juste titre remarquer la lourdeur de l'écriture.

```
struct struct_RTC_TR volatile * const regBitField_RTC_TR = (struct
struct_RTC_TR volatile * const) 0x40002800;     // Definition


union union_RTC_TR volatile * const reg_RTC_TR = (union union_RTC_TR
volatile * const) 0x40002800;
```

- *struct struct_RTC_TR* est un type de variable.
- *union union_RTC_TR* est un autre type de variable.

A l'aide du mot clé *typedef*, on peut améliorer la lisibilité en redéfinissant *struct_RTC_TR* et *union_RTC_TR* comme étant un nouveau type. On n'écrira donc plus *struct struct_RTC_TR* mais seulement *struct_RTC_TR*. On écrira donc plus *union union_RTC_TR* mais seulement *union_RTC_TR*. La déclaration de la structure, de l'union et la définition des pointeurs se fait donc comme suit :

```
    /* 1. Declaration de la structure et de l'union */
typdef struct {                         // Declaration
      uint32_t volatile SU:4;           // Second Units (BCD)
      uint32_t volatile ST:3;           // Second Tens  (BCD)
      uint32_t volatile reserved0:1;
      uint32_t volatile MNU:4;          // Minute Units (BCD)
      uint32_t volatile MNT:3;          // Minute Tens (BCD)
      uint32_t volatile reserved1:1;
      uint32_t volatile HU:4;           // Hour Units (BCD)
      uint32_t volatile HT:2;           // Hour Tens (BCD)
      uint32_t volatile PM:1;           // AM/PM notation
      uint32_t volatile reserved2:9;
      } struct_RTC_TR;

typedef union {                   // Declaration de l'union
      uint32_t volatile RTC_TR;  // RTC_TR accessible sur 32 bits
      struct_RTC_TR regBitField_RTC_TR; // Accessible par bit
} union_RTC_TR;


    /* 2.Definition et initialisation du pointeur sur le registre */
union_RTC_TR volatile * const reg_RTC_TR = (union_RTC_TR volatile *
const) 0x40002800;
```

On remarque que l'écriture est un peu allégée, mais cela peut parfois avoir des effets négatifs dans la compréhension globale du code. Cela est donc à utiliser avec parcimonie.

# 5 La gestion de l'affichage dans les systèmes embarqués

Un microcontrôleur ne possède pas d'unité d'affichage. La visualisation des valeurs des variables est donc une tâche délicate que nous pouvons réaliser de plusieurs façons.

1. En mode debug, nous pouvons faire des lectures de la mémoire aux adresses souhaitées. Cette première méthode est la plus simple mais demande de mettre des points d'arrêt dans le code. C'est une procédure fastidieuse.
2. A l'exécution, nous pouvons retransmettre les valeurs des variables sur une liaison série (souvent l'UART). Cette méthode est simple, mais la transmission sur liaison série peut perturber le déroulement temps réel du programme.
3. Nous pouvons utiliser des périphériques spécialisés du cœur ARM pour afficher des variables (par exemple le module ITM). Cette troisième méthode est la plus efficace mais aussi la plus complexe à mettre en place.

## 5.1 Printf sur UART

### 5.1.1 Transmettre sur la liaison série UART

Nous pouvons simplement transmettre les chaines de caractères sur la liaison série. Sur un STM32, on peut par exemple utiliser les fonctions HAL pour envoyer sur l'UART2. Le code est le suivant :

```
uint8_t myText[50];

void myPrintf(uint8_t * text){
  sprintf(myText,text);
  HAL_UART_Transmit(&huart2, myText, strlen(myText), 1000);
}
```

### 5.1.2 Rediriger le flux du printf

L'inconvénient de la fonction *myPrintf* que nous avons créé ci-dessus est que nous ne pouvons pas facilement profiter des arguments de la fonction *printf* (%d, %s, %cp) pour faire un affichage efficace. Une autre idée est donc de rediriger directement le flux de la fonction *printf* de la librairie *stdio.h*.

La fonction *printf()* fait appelle à la fonction *write()* qui est disponible dans le fichier *syscall.c* :

```
__attribute__((weak)) int _write(int file, char *ptr, int len)
{
  int DataIdx;
  for (DataIdx = 0; DataIdx < len; DataIdx++)
  {
        __io_putchar(*ptr++);
  }
  return len;
}
```

On s'aperçoit que cette fonction *write()* appelle elle-même une fonction *__io_putchar()* déclarée avec l'attribut *weak* :

```
extern int __io_putchar(int ch) __attribute__((weak));
```

Cela signifie que nous pouvons réécrire cette fonction *__io_putchar* pour les besoins de notre application, en redirigeant directement le caractère vers l'UART. Le code est le suivant :

```
int __io_putchar(int ch){
    HAL_UART_Transmit(&huart2,&ch , 1, 1000);
    }
```

> **La fonction printf() fonctionne uniquement si la chaine se termine par un '\n'**

## 5.2 Printf sur avec ITM sur SWO

ITM et console : TODO

# 6 Règles de codage et lisibilité du code

## 6.1 Ofuscated code

Il est très maladroit d'écrire du code illisible. Le respect d'une charte commune est indispensable pour que tous les programmeurs arrivent à relire du code de manière aisée. A titre d'exemple, seriez-vous capable de comprendre l'application simple suivante :

```
_(__,___){(__>=___)?printf("%d\t",__):printf("%d\t",___);}main(){_(010,0x9);}
```

Solution : Ce code affiche le maximum entre les deux valeurs passées en paramètre d'une fonction.

- Cette fonction s'appelle _.
- Les paramètres de cette fonction sont (__ , ___)
- Dans le main, la fonction est appelée avec les paramètre 010 (écrit en octal) et 9 (écrit en hexadécimal)

## 6.2 Nos règles communes

Afin d'uniformiser nos codes et d'avoir une lisibilité optimale, on se propose les règles suivantes. Elles sont fortement conseillées, mais ne sont ni exhaustives, ni obligatoires.

### 6.2.1 Organisation d'un projet

Fichiers :

- Tous les fichiers sont écrits en minuscule : fichier.c
- Les mots du fichier sont séparés par des _ (underscore) : fonction_tri.c
- Tous les fichier.c doivent inclure un fichier.h du même nom
- Tous les fichier.h doit être protégés contre les inclusions multiples

Indentation / Espace :

- Chaque bloc est décalé d'une tabulation
- Les accolades ouvrantes sont mises à la fin de la ligne

```
void function(void){
  while(1){
  }
}
```

- Coller le symbole * au nom du pointeur plutôt que au type pendant la déclaration
- Utiliser un espace entre les mots-clés suivants et la parenthèse : if, switch, for, while,…
- Utiliser un espace autour des opérateurs pour des calculs ou comparaisons
- Ne pas utiliser d'espace pour les pointeurs * et &
- Ne pas utiliser d'espaces pour les opérations de post et pré incrémentation ++, --

```
int8_t   *ptr_MaVarialble;
if (a == 5)
&a = 2;
a = i++;
(*a << 2) == 1 && (a >> 2) <= 10;
```

Les variables :

- Les variables commencent par une minuscule et chaque mot suivant commence par une majuscule :

```
maVariable
```

- Les pointeurs commencent par p ou ptr

```
pMaVariable, ptrMaVarialbe
```

- Donner des noms de variable explicite

```
uint32_t compteurFinBoucle, testDebutTransmission
```

- Les type de variables doivent utiliser la bibliothèque stdint : unint8_t, uint16_t, unint32_t…
- Les variables sont déclarées uniquement au début de la fonction, sauf pour le cas du for.

```
void fonction (void){
  uint8_t x, y, z;
  for(uint32_t i ; i < 0 ; i++){
  }
}
```

> **Attention : Certains compilateurs interdisent la déclaration en dehors du début des fonctions, l'exemple du if précèdent ne fonctionnerait pas.**

Constantes et Macro :

Les constantes et macros sont écrites en majuscule et les mots sont séparés par des _ (underscore):

```
#define TEST
#define CLEAR_BIT( REG , BIT )
```

## 6.3 Comments and Doxygen

Pour établir une documentation précise du code lié aux commentaires ajouter dans le code source, nous utiliserons doxygen [ www.doxygen.org ].