Systèmes embarqués et robotique : résumé cours 3

1. **La Mémoire dans les Systèmes Embarqués** :
   - Les systèmes embarqués utilisent des mémoires de l'ordre des KBytes/MBytes, ce qui nécessite une optimisation du code pour une efficacité maximale dans l'interaction avec la mémoire.
   - Le STM32F4 opère sur une structure de 32 bits, signifiant que les registres et le stockage de données s'effectuent sur 4 bytes.
2. **Memory Map** :
   - Chaque microcontrôleur possède une organisation mémoire unique. Lors de la compilation, le compilateur utilise la memory map pour savoir où placer le code et les données dans la mémoire. La memory map détermine comment les différentes zones de mémoire sont allouées et utilisées par le microcontrôleur, ce qui inclut la mémoire programme (FLASH), la mémoire des données (RAM), et les registres (aussi en RAM).
     - **Mémoire Flash (non-volatile)** :
       - **.text** : Contient le code exécutable.
       - **.rodata** : Pour les données constantes invariables.
       - **.isr_vector** : Héberge la table des vecteurs d'interruption.
     - **Mémoire RAM (volatile)** :
       - **.data** : Destiné aux variables statiques et globales initialisées non-nulles
       - **.bss** : Pour les variables globales et statiques non-initialisées ou initialisées nulles.
       - **.stack** : Utilisée pour les opérations telles que les pop et push, stockant les variables locales.
       - **.heap** : Réservé à l'allocation dynamique de mémoire.

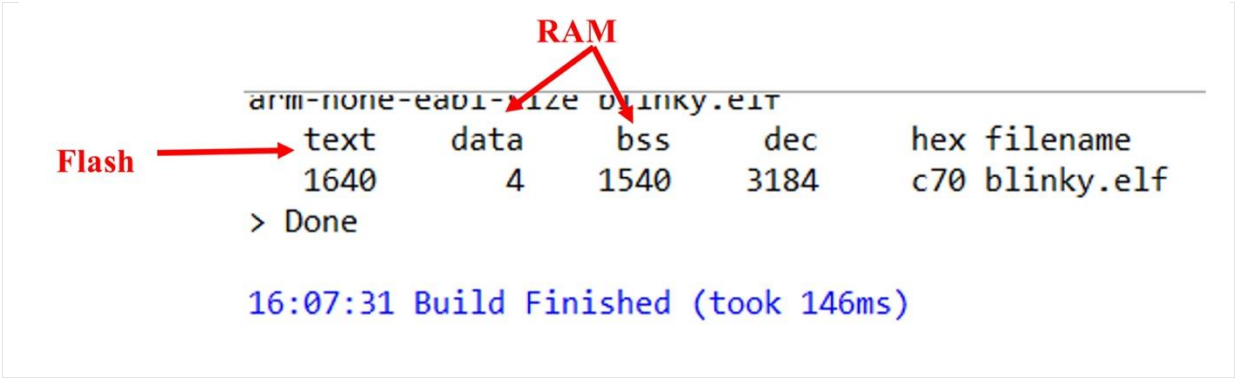

Figure 1- source : slide 12

3. **Gestion de la Memory Map et Stockage** :
    - La Memory Map est fixe, mais il est possible de réserver des zones spécifiques de la mémoire (Flash ou RAM) à des fins déterminées via un fichier spécial (comme STM32F407VGTx_FLASH.ld pour le STM32F4) utilisé par le linker lors de la compilation. Ce fichier définit les adresses de début et les tailles des segments de mémoire, permettant de réserver des parties spécifiques de la mémoire FLASH ou RAM pour différents usages.
    - La distinction est faite entre la SRAM (plus grande mais avec un accès plus lent dû au passage par le bus de communication générique) et la CCRAM (plus petite mais avec un accès plus rapide, car elle communique directement avec le processeur).

4. La mémoire programme

Le code écrit par le programmeur, une fois transformé en binaire (après avoir été assemble) est conservé dans la mémoire Flash (non volatile).

Disposition de ces sections dans la mémoire Flash :

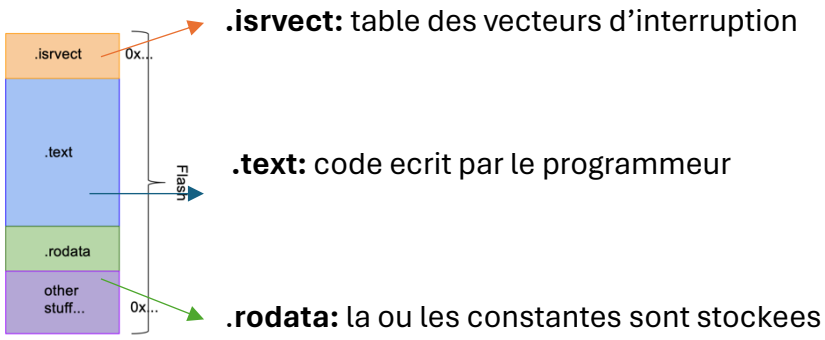

Figure 2 – Slide 22 montrant la structure de la mémoire programme.

Au début, on trouve la table des vecteurs d'interruption (isrvect). On y trouve des adresses qui pointent vers des parties de codes, qui sont exécutées lorsque l'appareil reçoit des signaux d'interruption spécifiques. Lorsqu'une interruption survient, le processeur interrompt le déroulement actuel du programme, sauvegarde le contexte actuel, notamment le compteur de programme (Program Counter, PC), puis accède à la table des vecteurs d'interruption pour savoir quelle partie de code exécuter pour cette interruption spécifique.

La section .text contient le code du programme, le .rodata contient les variable constantes (qu'il ne sera pas possible de modifier) utilisées par le programme.

Démarrage : RESET

Le RESET est un vecteur d'interruption spécial qui se trouve au début de la Flash (cf schéma : on trouve l'adresse de démarrage du code a la position 4 (0x0004) et à la position 0 (0x000) il y a l'adresse ou se trouve le stack) servant de point d'entrée pour le microcontrôleur lors de son initialisation. Lorsque le microcontrôleur est alimenté le RESET est invoqué pour commencer l'exécution du code. Cette interruption a une haute priorité (essentiel pour redémarrer le système convenablement)

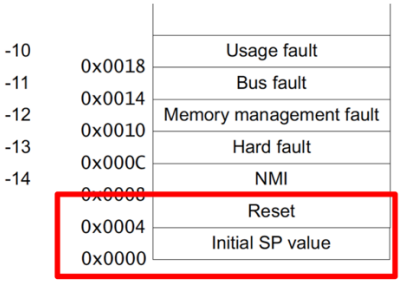

Figure 3 – Source : Vector table slide 30 du cours.

5. Les registres :

Les registres permettent la configuration des GPIOs (entrées et sorties des ports) et ils sont conservés dans une zone dédiée, accessible pour la modification, où sont regroupés l'ensemble des registres associés aux périphériques du microcontrôleur. On peut se référer au document *STM32F407 Reference Manual* pour trouver leurs adresses. Plusieurs exemples ont été réalisés lors du TP1 afin de modifier précisément des registres GPIOx précis.

Le fichier stm32f407xx.h permet à l'utilisateur de trouver les différents *#define* liés aux ports des périphériques mais aussi l'adresse de base des périphériques. Il suffit par la suite de faire des shifts afin de retrouver l'adresse exacte du registre dans la mémoire. Par exemple, les registres propres au GPIOD sont tous localisés en mémoire entre 0x40000000U et 0x40020C00U.

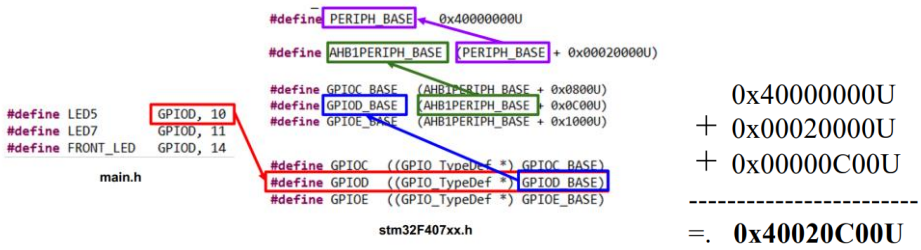

Figure 4 – source : slide 36-37 du cours, adresse exacte d'un registre

Par exemple, si on veut obtenir l'adresse exacte du registre MODER du GPIOD, on va d'abord chercher l'adresse de base de GPIOD puis on va rajouter l'offset du registre correspondant (ici MODER) qui est indiqué dans le fichier stm34f407xx.h comme indiqué ci-dessous.

Figure 5 – source : slide 41,

Les adresses des registres sont simplement définies comme :

Adresse du registre = 0x40020C00U + offset

6. La mémoire des données

6.1 Types de Mémoire et Variables

Les systèmes embarqués disposent généralement de mémoire de l'ordre des KBytes à MBytes, nécessitant une optimisation rigoureuse du code pour une interaction efficace avec la mémoire. La structure 32 bits du STM32F4 implique que les données sont souvent stockées sur 4 bytes, relevant de deux types de mémoire principaux : Flash (non-volatile) et RAM (volatile), divisés en plusieurs segments pour différents usages tels que le code du programme, les données constantes, les variables globales et statiques, ainsi que pour l'exécution de tâches spécifiques comme l'allocation dynamique.

6.2 Gestion des Variables Locales

Les variables locales sont stockées temporairement sur la pile (stack) pendant l'exécution de la fonction qui les contient. Le compilateur s'occupe d'allouer suffisamment d'espace sur la pile en ajustant le Stack Pointer (SP) en fonction du type et de la taille de chaque variable locale. Cette gestion dépend de la taille des données (par exemple, 4 octets pour un int sur le STM32F4) et du nombre de variables utilisées dans la fonction. Une fois la fonction terminée, l'espace alloué sur la pile est nettoyé, et le SP revient à sa position initiale.

6.4 Variables Statiques et Globales

Les variables statiques, contrairement aux locales, sont stockées hors de la pile, dans les segments de mémoire bss ou data selon qu'elles sont initialisées ou non. Elles restent persistantes tout au long de l'exécution du programme mais sont uniquement accessibles dans la fonction où elles sont déclarées. Les variables globales suivent une gestion similaire à celle des statiques mais sont accessibles de manière globale dans tout le programme.

6.5 Overflow de la Pile

Un défi majeur dans la gestion de la mémoire est d'éviter l'overflow de la pile, qui peut se produire lors de l'utilisation intensive de variables locales ou de fonctions récursives profondes. Cela peut entraîner un dépassement de la capacité limitée de la pile, avec des conséquences potentiellement désastreuses, telles que la corruption de la mémoire. Pour mitiger ce risque, il est conseillé de limiter l'utilisation de grandes structures de données comme les tableaux et d'éviter les récursions profondes. L'allocation dynamique sur le tas (heap) via malloc, calloc, etc., bien que plus complexe, offre une alternative pour gérer de grandes quantités de données sans risquer d'overflow de la pile.

En conclusion, la gestion efficace des variables et de la mémoire dans les systèmes embarqués repose sur une compréhension approfondie de la structure de la mémoire du microcontrôleur ciblé, ainsi que sur des choix judicieux concernant le type, la portée, et la durée de vie des variables pour optimiser l'espace de mémoire limité disponible.