Résumé: Cours 4

Tschudin Moritz, Tschammer-Osten Aiden, Luis Rodriguez, Aude-Line Fleury

March 19, 2024

1 Les nombres à virgule

1.1 Gestion des variables

Une bonne habitude à prendre en codant est de se demander si on a vraiment besoin d'utiliser des nombres à virgule car cela demande en effet plus de ressources. Pourquoi par exemple mettre les valeurs en "mètre" avec une précision au millième, si on peut tout calculer en "millimètre", évitant ainsi la virgule? De plus, il faut aussi prendre en compte qu'avoir des valeurs plus précises que la résolution de nos capteurs/moteurs ne nous bénéficie en rien.

Si nous voulons convertir un entier en flottant, ou vis-versa, nous pouvons utiliser un *cast* (changement forcé du type d'une variable). Méfions nous en: en effet, le compilateur le fait parfois implicitement. Ceci arrive par exemple lorsqu'on multiplie un *int* et un *float*: la multiplication sera faite en *float* à moins qu'on ne précise au compilateur de convertir la variable de type *float* en *int*, auquel cas on a une perte de résolution, comme dans la Figure 1.

```
int main() {

  float PI = 3.1415;
  int mul = 2;

  int result =  mul * (int)PI;

  return 0;
}
```

Figure 1: Exemple de problématique de cast où le type d'une variable peut être modifié pendant une opération. Source: [1].

Certains calculs, comme les calculs trigonométriques, ne peuvent pas être faits grâce à l'hardware et doivent être fait par software: ils requièrent donc plus de temps. Il peut alors être avantageux et intéressant de générer des Lookup tables. Il s'agit de tableaux contenant les valeurs pré-calculés dont on a besoin et qui nous éviterons de devoir refaire ces calculs, par exemple lors des calculs trigonométriques tels que *sin* ou *cos* (voir Fig. 2).

```
int i, sinus[360];
…
for(i=0;i<360;i++)
{
    sinus[i] = 10*sin(i/180*3.1415);
    printf("sin %d =
%d\n",i,sinus[i]);
}
```

Figure 2: Déclaration pour le pré-calcul d'une lookup table pour la fonction sinus. Source: [2].

1.2 Représentation des nombres à virgule

Les nombres à virgule peuvent aussi être stockés dans la mémoire, mais avec une représentation différente par rapport aux entiers. Il existe deux options, les virgules fixes qui rendent les opérations de multiplication plus simples car compatibles avec les entiers, et les nombres à virgule flottante qui eux sont plus utilisés car permettent une plus grande flexibilité.
Pour notre microcontrôleur STM32F4 avec un processeur ARM Cortex M4 les nombres à virgule flottante sont représentés avec la norme IEEE 754. Un *float* sera stocké sur 32 bits tandis qu'un *double* prendra 64 bits (voir Fig. 3).

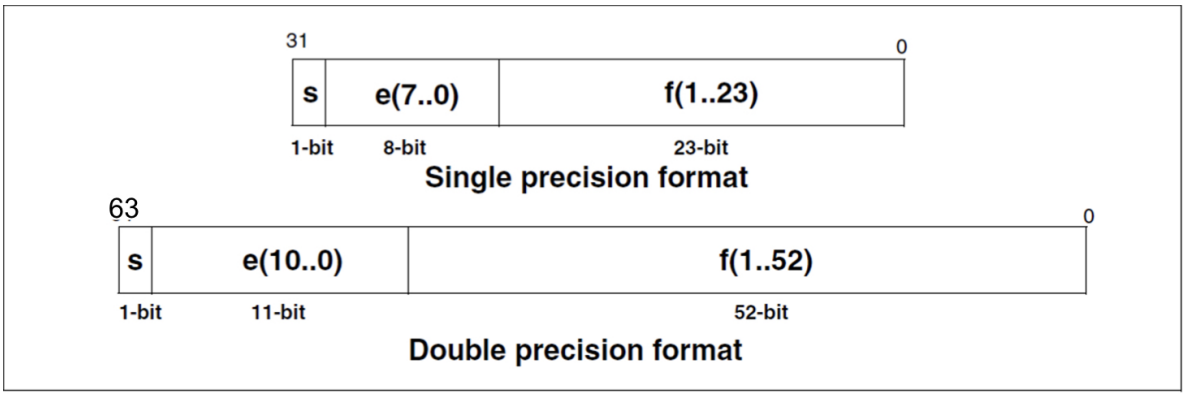

Figure 3: Structure pour un chiffre 32-bits et 64-bits. Source: [3].

Le chiffre dans le code source en assembleur sera donné en décimal. Pour l'interpréter il est pratique de le passer en hexadécimal puis en binaire. Ensuite, il faut transformer le chiffre binaire en représentation flottante, ce qui nous donnera les trois composants de la représentation à virgule flottante: signe, exposant et mantisse (voir Fig. 4). Il ne faut pas oublier que l'exposant est stocké avec un offset de 127 et que le zéro a sa représentation propre.

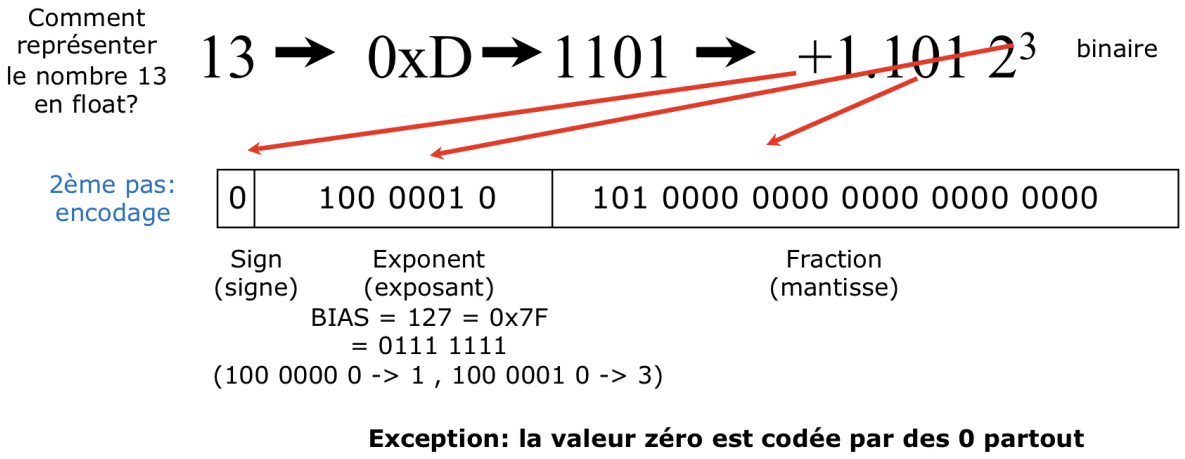

Figure 4: Etapes montrant comment convertir un chiffre (ici 13) en *float*. Source: [4].

1.3 Gestion des nombres à virgule

Notre processeur (ARM Cortex-M4) est équipé d'une FPU (floating point unit) qui permet de faire des opérations sur les nombres à virgule flottante grâce à l'hardware. Cette unité doit être activée en utilisant, lors de la compilation, les options *-mfloat-abi=hard -mfpu=vfpv2*. Si on ne l'active pas, les calculs en *float* et en *double* vont être fait de manière software et prendre beaucoup de cycles, on les reconnaîtra grâce à la fonction décrite en Fig. 5.

```
BEGIN_ARM_FUNCTION __aeabi_fmul
```

Figure 5: Exemple d'appel à une fonction software pour le calcul de la multiplication en float. Source: [5].

La FPU contient ses propres registres pour stocker des variables de type *float* ou *double* et pour effectuer des opérations avec en utilisant des mécanismes hardware. Il est essentiel de se rappeler que ces registres ont leurs propres fonctions en assembleur et doivent être sauvegardés si on utilise des *float* dans les interrupts. Il faut aussi garder en tête que les calculs en double précision (double) ne sont pas supportés par le compilateur, dans le cas écheant il vaut mieux opter pour l'utilisation de *floats*.

2 RTOS Real-Time Operating System

Les RTOS (Real-Time Operating System) sont des logiciels dont le but est d'efficacement gérer les ressources d'un système embarqué par exemple. Contrairement à un OS générique, le RTOS gère les tâches selon leurs priorités dans le temps tout en assurant la stabilité du système. Pour comprendre son utilité il est important de comprendre les notions de scheduling de tâches, le multi-tâches/multi-threading et plus généralement la programmation concurrente.

2.1 Programmation concurrente et temps réel

Contrairement à la programmation séquentielle qui exécute les tâches "l'une après l'autre", la programmation concurrente est caractérisée par l'exécution en (pseudo-)parallèle de tâches qui peuvent interagir, partager des ressources, etc. Il est évident que de tels processus nécessitent une excellente maîtrise des timings et exécution des tâches: il faut rester en "temps réel". C'est pourquoi le RTOS met la priorité sur la fiabilité du timing. Un tel OS doit être:

- Déterministe: Pour un input et des conditions données l'output sera toujours le même. Le comportement d'un tel système, son exécution des tâches, son timing et son allocation de ressources est consistant et répétable.
- Prédictible: Plus que déterministe, la prédictibilité implique que le programmeur a une idée/peut prévoir le comportement du système sous différentes conditions et différentes charges de travail.

Dans notre cas de systèmes embarqués, les RTOS se basent surtout sur le système Timer/Interrupt qui nous permettent d'allouer des temps précis à certaines tâches et les mettre à "dormir" pendant que nous n'en avons pas besoin. De plus, les interrupts ne disposent pas tous de la même priorité (Fig. 6). C'est ce qui nous permet d'assurer que les tâches les plus critiques sont exécutées à temps.

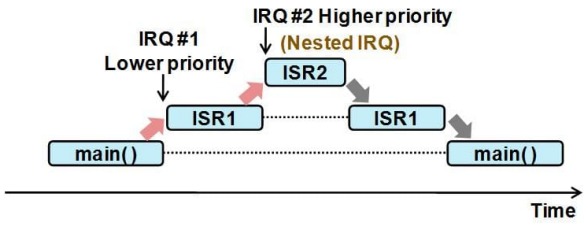

Figure 6: Illustration de la priorité des tâches. Source: [6].

2.2 Multi-tâches et multi-threading

Un thread est une séquence indépendante d'exécutions au sein d'un processus. Par exemple, une séquence dédiée à faire tourner le moteur d'un robot. En plus d'une séquence d'exécutions, chaque thread est caractérisé par une certaine priorité et d'un temps alloué. Il est possible de faire "dormir" un thread pendant que d'autres threads sont actifs et le réveiller quand il est nécessaire (Fig. 7).

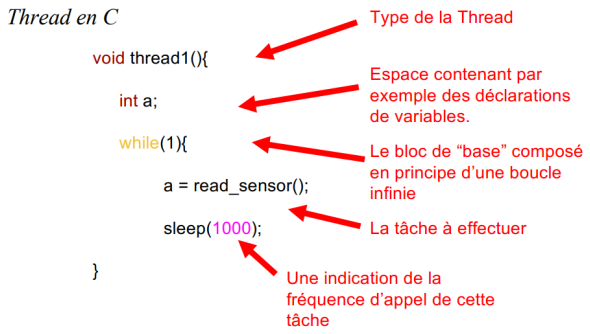

Figure 7: Forme générale d'un thread. Source: [7].

Ces caractéristiques nous permettent de faire du multi-tâche, de l'exécution en parallèle de nos différentes tâches. Pour faire du multi-tâches, on inclut le fichier d'initialisation "$crt0_v7m.s$" à l'initialisation. C'est lui qui initialise le process stack (et stack pointer) spécifique au contrôle des threads. Le process stack est un stack spécifiques au thread actuel. C'est aussi ce fichier qui lance le main et qui définit ce qui se passe une fois le main terminé.

2.3 ChibiOS

ChibiOS est un RTOS qui est Open Source qui permet la gestion des multi-tâches en temps réel avec une implémentation disponible pour le STM32F4. C'est une librairie qui facilite la gestion de tâches multiples simultanément tout en respectant le bon timing, grâce à son kernel (seulement un noyau et pas plusieurs comme un ordinateur classique) et son scheduler efficace (Voir Fig. 8). Un système robotique embarqué comme le e-puck2 avec ses périphériques nécessite une gestion précise du temps et des ressources d'une manière concurrente qui peut être bien implementée en utilisant ChibiOS.

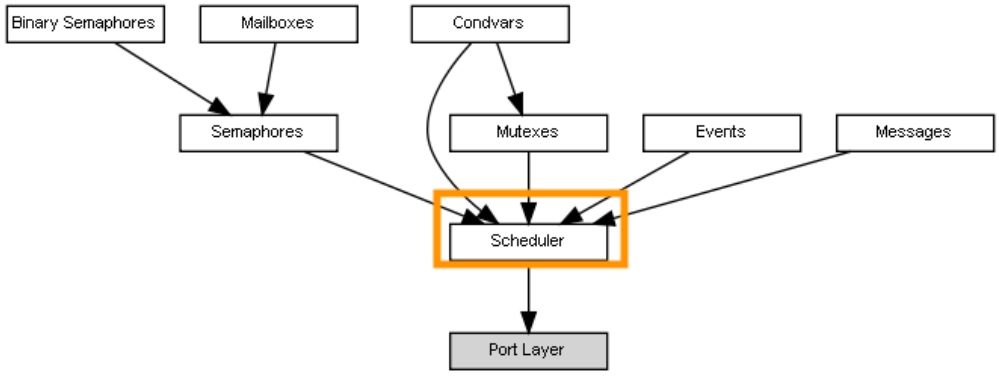

Figure 8: L'architecture globale du ChibiOS avec l'élément clé: le Scheduler qui permet des multi-tâches en pseudo parallèle. Source: [8].

2.4 Scheduling

Le scheduling décrit le processus de décision sur l'ordre et du timing lors d'une exécution des multi-tâches dans un RTOS. Toutes les tâches ont une priorité par rapport aux autres et possèdent des timings exact dans leur tâches. Le Scheduler est responsable pour l'ordre de l'exécution des tâches en respectant leur propriétés de priorités et timings. Le but est d'exécuter une couche d'une priorité plus bas pendant un sleep d'une priorité plus haute qui permet de laisser tourner les tâches en pseudo parallel. Cela permet d'avoir une efficacité et une réactivité sur le système où les ressources sont limitées et les spécifications en temps réel sont strictes.

Il est important de respecter et libérer des ressources car il existe le risque du phénomène de l'inversion de priorité illustrée dans la Fig. 9. Ce phénomène de l'inversion de priorité peut se produire lorsque une tâche de haute priorité, comme le main, doit attendre une ressource qui est bloquée

par une tâche de priorité inférieure par exemple par Thread03. Mais si Thread03 devient bloquée et interrompu par des tâches de haute priorités (i.e. Thread01 et Thread02), le main doit d'abord attendre que Thread03 se termine et libère la ressource. Ce cas oblige main à attendre Thread03 pendant la durée du blocage de la ressource. Cette inversion de priorité est à éviter car il y a un risque d'avoir des retards significatifs pour les tâches de haute priorité.

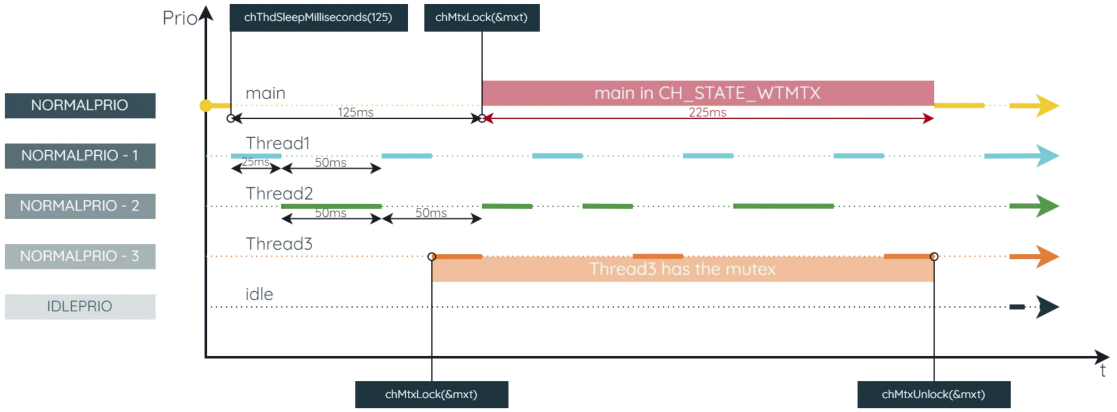

Figure 9: Diagramm du timing d'un "mauvais" planning avec problème du inversion de priorité. Source: [9].

References

[1] Prof. Francesco Mondada. Micro-315, systèmes embarqués et robotique, 2024. Cours 4, slides p. 28.

[2] Prof. Francesco Mondada. Micro-315, systèmes embarqués et robotique, 2024. Cours 4, slides p. 22.

[3] Prof. Francesco Mondada. Micro-315, systèmes embarqués et robotique, 2024. Cours 4, slides p. 6.

[4] Prof. Francesco Mondada. Micro-315, systèmes embarqués et robotique, 2024. Cours 4, slides p. 7.

[5] Prof. Francesco Mondada. Micro-315, systèmes embarqués et robotique, 2024. Cours 4, slides p. 12.

[6] Posted by: LTP. Stm32f0 tutorial 3: External interrupts. https://letanphuc.net/2015/03/stm32f0-tutorial-3-external-interrupts/, 2015/03/21. Accessed: 2024-03-15.

[7] Prof. Francesco Mondada. Micro-315, systèmes embarqués et robotique, 2024. Cours 4, slides p. 51.

[8] chibios.org. chibios.org/dokuwiki. http://www.chibios.org/dokuwiki/lib/exe/detail.php?id=chibios%3Adocumentation%3Abooks%3Art%3Akernel&media=chibios:documentation:books:rt:kernel:rt_arch.png, 2020/05/12. Accessed: 2024-03-12.

[9] Posted by: Rocco Marco Guglielmi. The complete reference for multithreading in chibios/rt. https://www.playembedded.org/blog/the-complete-reference-for-multithreading-in-chibios-rt/, 2024/01/03. Accessed: 2024-03-12.