Université de Sherbrooke - S5 - Génie informatique

App2 - Système en temps réel

Sainrat Paul - Ferreira Brian - Caron William

19 septembre 2017

Table des matières

1 Gestion des ressources

Un des objectif de ce problème était de faire un code le plus optimisé en espace possible (<32Ko).
Pour reduire la taille de notre code nous avons évité tout déclaration de variable globale non necessaire et avons essayer d'utiliser des type de varable qui prenait le moins de place possible, des booléans à la place d'entiers, des entiers à la place de flotants ...
Malgrès nos efforts pour minimiser la taille de notre fichier, nous avons constater que nous restions au dessus de 32Ko, nous en avons conclu que la plus grande partie de la place est occupé par les bibliothèque que nous utilisons à savoir **mbed.h,rtos.h** et **rtc.h**.
Nous avons donc par la suite enlevé les partie de ces bibliothèque que ne nous est pas utile.

2 Algorithmes de création et méthodes stockages des évenements

Structures et fonctions pemetant de gérer les évenements Dans un premier temps il nous avons créer une structure **event** dans laquelle nous pourrions stocker la date de l'évenement sous la forme d'une chaine de charactères ainsi que la nature de sa création (Numérique ou Analogique).
Nous avons ensuite utilisé la **Real Time Clock** du **LCP1768** ainsi que la bibliothèque qui facilite son utilisation pour obtenir la date précise à laquelle l'évenement est apparue.
Nous avons donc du au préalable régler l'horloge à la bonne Date.
Puis nous avons coder une fonction **date** qui prend en argument un pointeur vers un **event** et qui remplis la chaine de charactère par la date de l'occurance de l'évenement dans le format attendu (AA :MM :JJ :HH :MM :SS).

Stockage des evenements Pour le stockage des évenements nous avons décidé d'utiliser une **MemoryPool** avec des cases de la taille d'une structure **event** aisin qu'une **Queue** contenant des addresses pointant sur les cases de la **MemoryPool** nous avons limité la taille de la memoryPool et de la queue à 4 évenements car c'est le pire des cas qu'on pourrait avoir (4 évenement relevé en meme temps).

Création des évenements Pour ce qui est de la création des évenements, nous avons deux **fonctions** liées à deux **threads** permetant d'observer les entrés sur les pins 15,16,19 et 20 comme expliqué dans le paragraphes suivant . Lorsque la conditions de génération d'un événement est rencontrées, alors on demande à la **MemoryPool** l'allocation du case pour y stocker un evenment dont on remplira la date grâce à la fonction précédement introduite ainsi que la nature de l'interuption. Puis on metra l'adresse de cette case dans la **Queue** pour pouvoir y accéder plus tard.

3 Fonctions crées pour répondre à la problematique

3.1 Mesure de l'entrée numérique

Pour la lecture des entrée numérique, nous devons regarder les changements de valeur a chaque 100ms. Par contre, si la valeur changeait, nous devions vérifier 50ms après la lecture que la valeur soit encore la même qu'à la première lecture. Cela nous permet de filtrer les rebonds dans les boutons. Nous avons créé un thread pour chaque entrée numérique. Cela nous permet de bien gérer les rebonds de chaque boutons. A l'aide d'un booléen qui change de valeur a chaque fois que le thread reçoit un signal, on enregistre une valeur temporaire. Si la valeur est différente que l'ancienne, on signal wait le thread courant. Donc au prochain interrupt nous serons encore dans le if. Alors, on lit une nouvelle valeur temporaire et si elle est encore différente que l'ancienne (pas la temporaire), on change l'ancienne pour la nouvelle.

3.2 Mesure de l'entrée analogique

Le Thread qui s'occupe du traitement des données analogiques est réveillé par un signal toute les 250 milisecondes. Lorsque le Thread se réveille, il récupère un échantillon des pins analogiques puis se rendors. Il va effectuer cette opération jusqu'à ce qu'il possède 5 échantillons pour chacun des 2 pins analogiques. Une fois toutes les valeurs récupérées, le Thread va calculer les moyennes et la comparer aux anciennes (qui sont stocké en mémoire). Si la nouvelle moyenne correspond à 12.5% de la différence entre l'ancienne et la nouvelle moyenne, alors un évènement est généré sous la forme d'une date de format : *AA/MM/JJ/HH/MM/SS* et stocké dans une queue. Si les deux moyennes valident la condition de génération d'évènement, alors deux évènements seront générés. A noté que nous avons multiplié par 1000 la valeur du pourcentage (12.5%) et les valeurs récupérées sur les pins analogiques afin d'éviter l'utilisation de variables de type float.

3.3 Collection des évenements

Pour la collecte des évenements nous avons crée une fonction qui sera lié un **thread** de prioritée moindre par rapport à ceux de mesure des entrées. Ce **thread** qu'on appellera **threadColection** ne se reveille que lorsqu'un evenement est disponible dans la **MemoryPool**, les **thread** de mesures des données active un signal lorsqu'il mettent à disposition l'adresse d'un évenement dans la **Queue**. Le **threadCollection** va alors se reveiller (si les deux autres thread ne sont pas actifs eux même), récuppérer l'adresse dans la **Queue** est fait parvenir au PC par un port serie les données liées à l'évenement contenue à cette adresse. Une fois que le PC à récupérer l'information, le **threadCollection** libére la case de la **MemoryPool** occupée par l'évenement.

4 Description des méthodes utilisées pour valider les fréquences d'échantillonnage et analyse de la meilleure méthode

La problématique demandait que nous utilisions l'horloge temps-réel du microcontrôleur **RTC** afin d'étamper les évènements. Par contre, la librairie RTC nous permet seulement de faire des interruptions au secondes avec leur fonction attach(). Donc nous devions nous fier sur l'horloge interne du processeur pour notre échantillonnage. Afin de s'assurer que l'horloge interne soit fiable, nous avons créé un Ticker qui faisait des interruptions au 50ms. Les interruptions faisaient simplement incrémenter un compteur.
Pour le comparer au **RTC**, à chaque seconde, en utilisant le *RTC.attach()* nous incrémentions un autre compteur. Alors à chaque fois que le compteur du **RTC** incrémentait d'un, le compteur du **Ticker** devait avoir un multiple de cinq. Cette méthode nous a permis de confirmer que nous pouvions utiliser la classe **Ticker** de la librairie **mbed** sans perdre de précision dans notre cas.
Tout dépendamment de la fréquence d'échantillonnage, le **RTC** sera plus précis puisqu'il reste stable même si beaucoup de variable externe changent. Par contre si notre systeme est un soft real time, on peut se permettre d'utiliser l'horloge interne du processeur qui est plus sensible au changement externe. Mais, elle nous permet d'avoir des fréquences d'échantillonnage plus petites.

5 Comparaison des différentes performances du code de l'inclinomètre

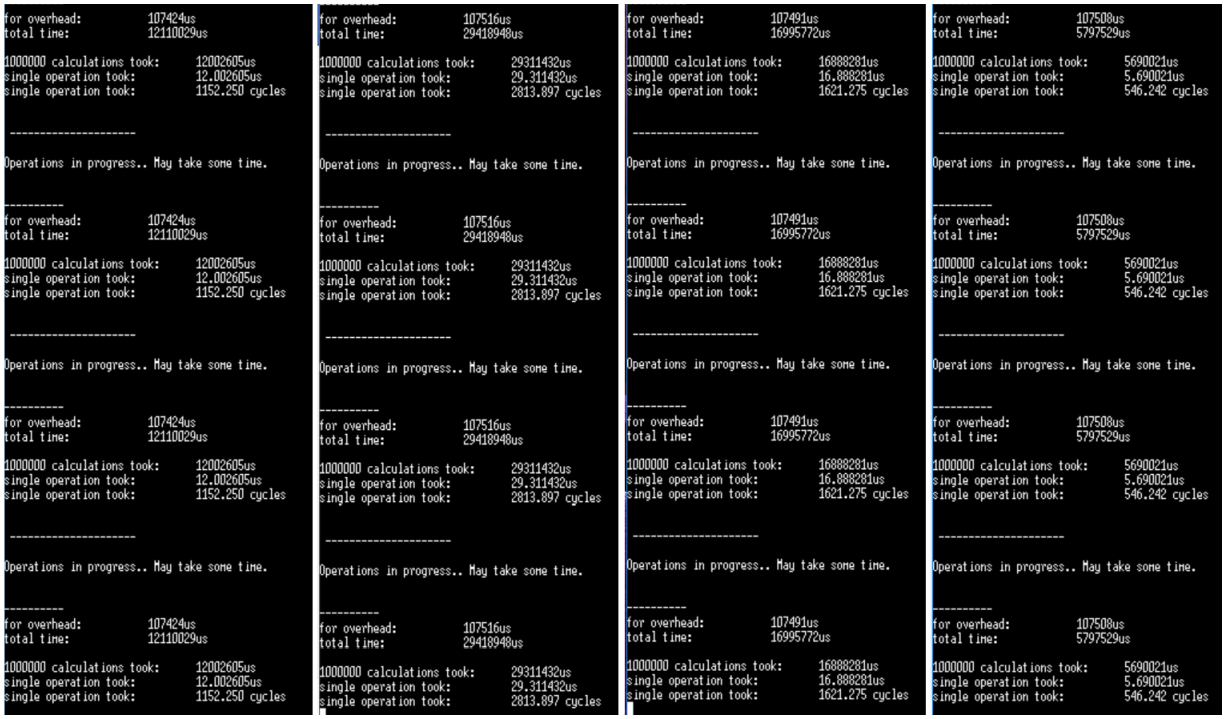
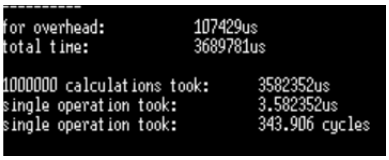


FIGURE 5.1 – Resultats du benchmark

Le programme numéro 4 est le plus rapide des 4 programme pour le calcul d'un angle avec à un accéléromètre. Ce programme utilise une table des valeurs du arc-cosinus précalculées, ce qui permet d'alléger le nombre de calculs que le CPU doit effectuer pour retourner le résultat. Cette méthode nécessite de trouver l'indice du tableau contenant le résultat du arccos en fonction de la valeur de la composante en z de l'accéléromètre. La valeur du module utilisé pour le calcul de l'angle est lui aussi calculé à l'avance. On notera également que ce programme utilise 5 variables volatiles ce qui ralenti le CPU.
Nous avons ensuite le programme numéro 1 qui est environ 2 fois moins rapide que le précédent. Ce dernier n'utilise pas de table de valeurs précalculées de la fonction arc-cosinus mais effectue lui-même le calcul ce qui ralenti le programme. La valeur du module est quant à elle déjà calculée comme pour le programme numéro 4. On notera que pour celui-ci, seulement 4 variables volatiles sont utilisées, la variable *module* est ici déclarée comme un simple float. On peut déjà affirmer que le calcul de arc-cosinus ralenti énormément le programme, passant du simple au double.
En troisième position nous avons le programme numéro 3, qui est aussi long que l'utilisation des 2 programmes précédents l'un à la suite de l'autre. Ce programme-ci ne possède aucune donnée précalculée, ainsi il effectue à la fois le calcul du arc-cosinus, mais égale-

ment le calcul du module en utilisant la fonction racine carrée. On observe ici que, d'une façon générale, les calculs ralentissent plus ou moins fortement un programme. Précalculer les valeurs d'un calcul permet d'augmenter la vitesse d'un programme ce qui peut être une solution non négligeable dans certaines applications.
Enfin nous avons en dernière position le programme numéro 2 qui est presque identique au programme numéro 1 à la différence qu'il ne possède pas la variable contenant la valeur du module. La valeur est directement ajoutée dans le calcul de l'angle sans passer par la variable module. Cette approche rend le code presque 3 fois plus lent qu'avec l'utilisation d'une variable.



En conclusion de cette observation, précalculer la valeur des calculs, utiliser des variables plutôt que de donner directement la valeur et éviter le plus possible l'utilisation des variables volatiles permet d'augmenter la vitesse du programme, donc de ménager le processeur durant l'exécution de celui-ci. On obtient donc, en appliquant tout ce que l'on a découvert un temps d'exécution encore plus rapide que le programme numéro 4.
Si l'on relance plusieurs fois le programme (avec de nouvelles valeurs d'entrées pour l'accéléromètre), on n'observe aucuns changements sur la durée totale du programme. Ceci est dû à la puissance de calcul de note CPU qui est suffisamment élevée pour que la durée ne change pas entre 2 calculs avec différentes entrées. Les entrées ne varient pas énormément ce qui n'affecte pas vraiment la vitesse de calcul du CPU.

6 Conclusion

Cette APP nous a permis de toucher du doigts les systèmes en temps réel et d'avoir une meilleure apréciation de leurs problèmes. Nous avons du créer des thread et metre en place des structures de communication entre eux en evitant de créer des dead lock.
Nous avons aussi vu la limitation en mémoire des système embarqué et avons observer différentes technique pour faire un code optimiser, qui se déroule bien sans prendre trop d'espace.
Comme nos fonctions n'était pas trop longues et que l'échéance de nos threads leurs laissé emplement le temps de s'executer nous n'avons pas encore été coffronté au problème de prioritée et d'ordonnancement des thread.

7 Annexes

FIGURE 7.1 – Architecture du Code

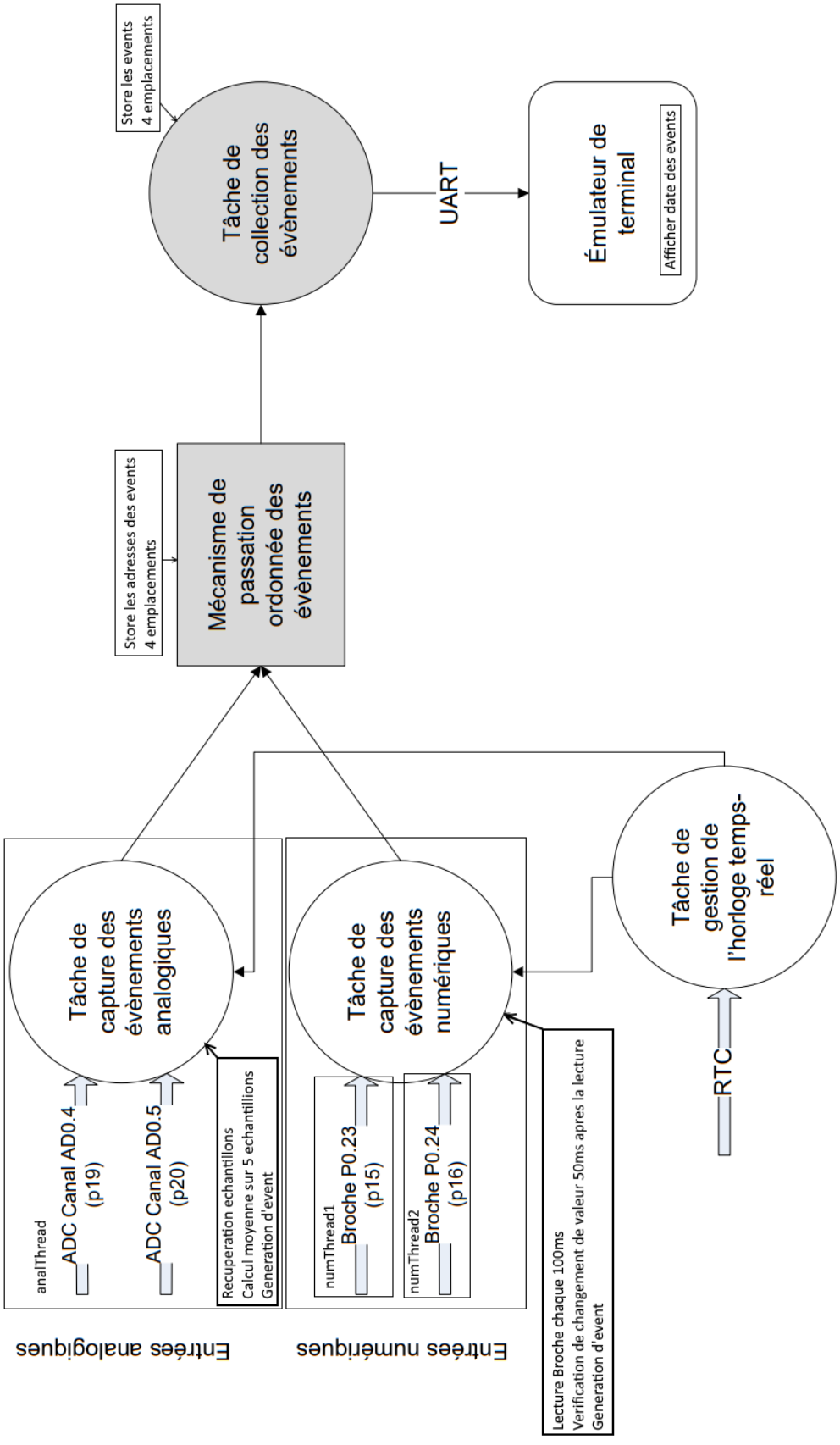


FIGURE 7.2 – diagramme sequentiel du déroulement temporel du programme