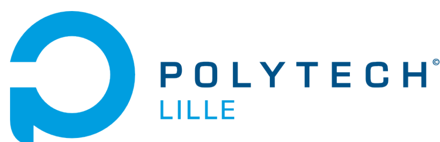
Mémoire de projet de fin d'études

Département IMA

Évaluation du coût des threads dans un système temps réel

Laboratoire d'informatique fondamentale de Lille



Soutenu par :	**Encadrants :**
LELAURE Mélanie	M FORGET Julien
Année : 2013/2014	M BERTOUT Antoine

Table des matières

Introduction **5**

I Présentation **6**

1 Équipe d'accueil **6**
- 1.1 Le laboratoire de recherche : le LIFL 6
- 1.2 L'IRCICA 6
- 1.3 L'équipe DART 7
- 1.4 Le groupe Émeraude 7

2 Description du sujet et organisation du projet **7**
- 2.1 Le task clustering 7
- 2.2 Cahier des charges 8
- 2.3 Organisation du projet 8

II Matériel utilisé **8**

3 Présentation du matériel et des logiciels **9**
- 3.1 Le matériel : micro-contrôleur TMS570 9
- 3.2 Les logiciels 10
  - 3.2.1 Génération du code avec HALCoGen 10
  - 3.2.2 L'environnement de développement Code Composer Studio 10
  - 3.2.3 FreeRTOS un système d'exploitation temps réel 10

4 Prise en main **11**
- 4.1 Installation des logiciels 11
- 4.2 Générer un programme d'exemple utilisant FreeRTOS avec HALCoGen 12
- 4.3 Modifier le programme d'exemple avec CCS 12
  - 4.3.1 Création d'un projet 12
  - 4.3.2 Ajout de tâches temps réel 14

III Expérimentations **14**

5 Coûts mémoire **15**
- 5.1 La gestion de la mémoire avec FreeRTOS 15
  - 5.1.1 Allocation statique 15
  - 5.1.2 Allocation dynamique avec best fit 15
  - 5.1.3 Wrapper des fonstions C 15
  - 5.1.4 Allocation dynamique avec best fit et compactage 15
  - 5.1.5 Choix de la gestion statique 16
- 5.2 Le protocole de test 17
- 5.3 Les résultats obtenus 17

6 Temps d'exécution **19**
6.1 Utilisation de l'outil Run Time Statistics 19
6.2 Les problèmes rencontrés 19
6.3 Différentes solutions envisageable 20
6.3.1 Utiliser le timer interne 20
6.3.2 Configurer avec HALCoGen 20
6.3.3 Utiliser un autre logiciel : Tracealyzer 20

Conclusion **22**

Bibliographie **23**

Table des figures

1 Diagramme de Gantt du projet 9
2 Un système d'exploitation temps réel dans son environnement . . 11
3 Configuration du compilateur lors de la création d'un nouveau projet avec CCS 13
4 Paramètres de la fonction *vTaskCreate* 14
5 Coût mémoire théorique de FreeRTOS 16
6 Allocation de la mémoire RAM lors de la création d'un tâche . . 16
7 Coûts mémoire d'une tâche en fonction de la mémoire allouée . . 18

Introduction

Ce rapport s'inscrit dans le cadre du projet de fin d'études réalisé en 5ème année à l'école Polytech Lille.

L'objectif de ce projet est de mesurer l'impact de l'utilisation d'un grand nombre de threads sur un système embarqué. Il s'inscrit parmi l'un des sujets de recherche de l'équipe DART (Dynamic Adaptation and Real-Time) du LIFL (Laboratoire d'Informatique Fondamentale de Lille) qui est le regroupement de tâches nommé aussi task clustering.

Je souhaite tout d'abord adresser mes remerciements aux personnes qui m'ont aidée dans la réalisation de ce projet. Ainsi, je remercie MM Julien Forget, maître de conférence au LIFL/Polytech Lille et Antoine Bertout, doctorant au LIFL/Université Lille 1. En tant qu'encadrants, ils m'ont guidée dans mon travail et m'ont aidée à trouver des solutions pour avancer.

Ce rapport est divisé en trois parties. La première présente le contexte, tant l'équipe d'accueil que le sujet du projet et son organisation. Ensuite, une partie est consacrée aux matériels utilisés et à leur prise en main. Enfin les expérimentations et leurs résultats sont présentés dans la dernière partie.

Première partie
Présentation

1 Équipe d'accueil

Le projet a été réalisé au sein du groupe Émeraude de l'équipe DART du laboratoire d'informatique fondamentale de Lille (LIFL) et de l'IRCICA.

1.1 Le laboratoire de recherche : le LIFL

Le LIFL (Laboratoire d'Informatique Fondamentale de Lille) est le laboratoire de recherche en informatique de Lille. [1] Le LIFL est une Unité Mixte de Recherche avec comme tutelles le CNRS (Centre National de la Recherche Scientifique) et l'Université Lille 1 et comme établissements partenaires l'Université Lille 3 et Inria. Il est rattaché à l'Institut des Sciences de l'Information et de leurs Interactions (INS2I) du CNRS.

Créé en 1983, le laboratoire accueille actuellement plus de 250 personnes, dont une centaine de chercheurs et enseignants-chercheurs, trente ingénieurs, techniciens et administratifs et une centaine de doctorants.

Le LIFL est également membre fondateur de l'institut de recherche interdisciplinaire IRCICA.

1.2 L'IRCICA

L'IRCICA (Institut de Recherche en Composants et systèmes pour l'Information et la Communication Avancée) est un institut de recherche interdisciplinaire créé en 2003. [2] Il dispose de quatre unités partenaires principales :

- l'IEMN (Institut d'Électronique de Microélectronique et de Nanotechnologie)
- le LIFL
- le PhLAM (laboratoire de Physique des Lasers, Atomes et Molécules)
- le L2EP(Laboratoire d'Électrotechnique et d'Électronique de Puissance de Lille)

L'IRCICA est un hôtel à projets sans équipes résidentes, les chercheurs restent attachés à leur unité, qui a pour objectif de privilégier les recherches à caractère exploratoire ou se situant à l'interface logiciel-matériel et de rapprocher les communautés locales du logiciel et du matériel. Ainsi le programme scientifique de l'IRCICA est organisé autour de trois axes principaux :

- Réseaux de capteurs et Internet des objets
- Dispositifs photoniques
- Nouveaux services et dispositifs pour l'intelligence ambiante

Actuellement l'IRCICA accueille trois platformes : Fibertech (centrale technologique de fibres à cristaux photoniques), platforme télécom et réseaux de capteurs et PIRVI (Plateforme Interactions-Réalité Virtuelle-Image), et six équipes de recherches : Photonique, CSAM, 2XS ,MINT, FOX et DART.

1.3 L'équipe DART

L'équipe DART (Dynamic Adaptation and Real-Time) fait partie du LIFL et s'intéresse aux systèmes embarqués et plus particulièrement aux architectures massivement parallèles reconfigurables et aux architectures hétérogènes et variables. [3] L'équipe s'intéresse notamment aux moyens de les programmer efficacement (temps d'exécution, temps de développement, consommation énergétique, sûreté de fonctionnement) avec la prise en compte du temps à tous les niveaux de la conception à l'exécution.

L'équipe DART est composée de dix membres permanents : deux professeurs, deux chargés de recherche, cinq maîtres de conférences et un ingénieur. et de onze membres non permanents : deux post-doctorants, huit doctorants et un ingénieur.

Elle est divisée en deux groupes : Dreampal (Dynamically REconfigurable Massivelly Parallel Architectures and Languages) et Émeraude (Embedded Real-Time Adaptive Virtualization for Post-Moore Architectures).

1.4 Le groupe Émeraude

Le groupe Émeraude étudie des architectures matérielles et leur utilisation efficace combinant des caractéristiques des circuits actuels et des caractéristiques de nanocomposants émergeants. [4] L'objectif à long terme est d'obtenir un modèle de programmation nouveau pour des applications mobiles sur ces nouvelles machines, qui seront vraisemblablement massivement parallèles, hétérogènes, et auront des contraintes de tolérance aux pannes, et de consommation d'énergie à minimiser.

En parallèle avec ces travaux, le groupe explore la conception de nouvelles architectures neuromorphiques. Avec le soutien de l'IRCICA, il initie des recherches en partenariat avec des chercheurs de l'IEMN pour simuler des réseaux de neurones matériels à base de memristors. Le but est de proposer des architectures à la fois utiles pour le calcul et réalisables.

2 Description du sujet et organisation du projet

2.1 Le task clustering

Le task clustering, ou regroupement de tâches, d'un système temps réel embarqué est l'un des sujets actuellement étudiés par le groupe Émeraude [5].

En effet, les systèmes temps réel sont généralement composés de plusieurs sous-systèmes ayant chacun une fonctionnalité propre et des contraintes de temps. Une solution simple est d'implémenter chaque fonctionnalité dans un thread différent. Cependant, dans certains cas comme dans l'aéronautique, ces fonctionnalités sont très nombreuses (plusieurs centaines voir milliers) et l'utilisation d'un thread par tâche entraîne des surcoûts en terme de mémoire, de temps d'exécution et de changement de contexte. Le groupe Émeraude étudie le regroupement automatique de ces tâches afin de minimiser le nombre de threads tout en respectant les contraintes de temps du système.

2.2 Cahier des charges

L'objectif de ce projet est de vérifier l'impact du task clustering sur un système temps réel. Y a-t-il un réel gain de mémoire et de temps à regrouper les tâches afin de limiter le nombre de threads ? Le but est donc de mesurer et comparer le coût mémoire et le temps d'exécution d'un thread pour un système composé d'un grand nombre de tâches et pour un autre de seulement quelques tâches, les deux systèmes ayant les mêmes fonctionnalités.

Pour ce faire, on dispose d'un micro-contrôleur dédié aux systèmes temps réel critiques. Après une prise en main du micro-contrôleur en y intégrant un système d'exploitation temps réel, il est demandé de mesurer le coût mémoire et le temps d'exécution des threads utilisés pour exécuter différentes tâches. Puis, afin de comparer ces résultats avec un grand nombre de tâches, de définir deux ensembles de tâches réalisant la même fonction, par exemple des calculs mathématiques, le premier ayant un faible nombre de tâches (une dizaine), le second un grand nombre (une centaine ou un millier). Pour ce faire, il faut mettre en place un moyen de générer un grand nombre de tâches de manière systématique, un script shell par exemple.

2.3 Organisation du projet

Pour cause de départ à l'étranger au semestre précédent, ce projet a été réalisé en un mois seulement.

Le projet ayant démarré fin janvier, certains choix avaient été pris en amont, notamment le choix du micro-contrôleur et de l'utilisation d'un système d'exploitation temps réel tel que FreeRTOS.

La réalisation du projet peut se diviser en quatre grandes phases visibles sur le diagramme de Gantt de la FIGURE 1. Dans un premier temps, une phase de prise en main de la platforme de développement et ses outils afin d'être capable d'exécuter un programme d'exemple avec ou sans tâche temps réel s'est avéré nécessaire. À cause de problème de compilation et d'exécution des programmes d'exemple fournis cette phase a duré environ une semaine et demie. Ensuite, il a été possible de réaliser des mesures sur le coût mémoire des threads grâce aux outils fournis par Texas Instruments et FreeRTOS. Cette seconde phase a duré quelques jours. Après avoir mesuré les coûts mémoire, la mesure des temps d'exécution est la suite logique du projet. FreeRTOS propose un outil permettant d'effectuer ces mesures, cependant, il nécessite une bonne compréhension du système d'exploitation et une modification dans la configuration du micro-contrôleur. Cette phase de compréhension et recherche a duré environ une semaine et demie. Après cette phase de recherche sur l'outil et le système d'exploitation, il restait seulement une semaine avant la date de rendu du dossier. Il a donc été décidé de se concentrer sur le dossier et autres livrables demandés pour le projet durant cette dernière semaine.

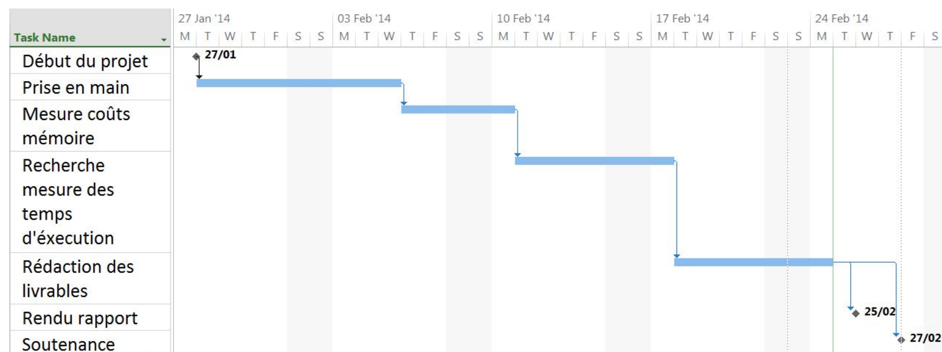


FIGURE 1 – Diagramme de Gantt du projet

Deuxième partie
Matériel utilisé

3 Présentation du matériel et des logiciels

Le matériel utilisé pour ce projet est un kit de développement TMS570LS31x ainsi que les logiciels HALCoGen et Code Composer Studio développés par Texas Instruments. Il a également été choisi d'utiliser le système d'exploitation temps réel FreeRTOS.

3.1 Le matériel : micro-contrôleur TMS570

Le TMS570 est un micro-contrôleur temps réel développé par Texas Instruments (TI) de la famille Hercules. Il est destiné aux systèmes temps réel critiques tels que les systèmes de freinage dans les domaines de l'automobile, de l'aéronautique et du transport ferroviaire.

Nous disposons ici d'un kit de développement TMS570LS31x [6] qui embarque notamment un micro-contrôleur TMS570LS3137 [7], un capteur de température, un capteur de lumière et quelques LEDs. Le micro-contrôleur est quant à lui, composé d'un processeurs dual core 32 bits ARM Cortex-R4, de 2MB de mémoire flash et de 160KB de RAM. La platforme de développement a la particularité d'être fournie avec un logiciel (HALCoGen) permettant de générer le code de configuration pour les couches basses du micro-contrôleur et un environnement de développement (Code Composer Studio).

Ce micro-contrôleur a été choisi car il est récent et est destiné à de vraies applications temps réel critiques. De plus, il dispose d'une interface USB ce qui le rend facilement programmable. Il est compatible avec des systèmes d'exploitation temps réel disponibles et gratuits, par exemple FreeRTOS et Micrium. Il est également multi-core ce qui pourrait s'avérer utile par la suite. Enfin, son prix est abordable car il coûte moins de cent euros.

3.2 Les logiciels

Plusieurs logiciels ont été utilisés, l'ensemble de ces logiciels combinés permet de créer une application temps réel sur le micro-contrôleur. En effet, HALCoGen permet de générer le code de configuration, puis Code Composer Studio (CCS) permet de créer une application incluant le code généré qui sera ensuite chargée sur le micro-contrôleur. Enfin, FreeRTOS est le système d'exploitation du micro-contrôleur, il est intégré à l'application sous la forme d'un ensemble de fichiers sources compilés avec l'application par CCS.

3.2.1 Génération du code avec HALCoGen

HALCoGen (Hardware Abstraction Layers Code Generator) est un logiciel permettant de configurer les différents paramètres du micro-contrôleur : périphériques, interruptions, horloge et de générer le code configuration pour les couches basses de celui-ci. [8] Ce code peut également être considéré comme le driver du micro-contrôleur.

En effet, HALCoGen permet à l'utilisateur de configurer de façon graphique différents micro-contrôleurs distribués par Texas Instruments comme le TMS570LS3137. Une fois le micro-contrôleur configuré l'utilisateur peut générer le code pour la couche d'adaptation hardware (HAL : Hardware Adaptation Lawer). Ce code est un ensemble de fichiers écrits en assembleur et en C. Puis, en important le projet dans Code Composer Studio l'utilisateur peut créer son application en C dans le fichier *sys_main.c*.

Plusieurs exemples de configuration sont fournis par l'aide du logiciel, ainsi que la procédure à suivre pour les reproduire. De plus, pour chaque micro-contrôleur supporté le logiciel propose une configuration incluant le système d'exploitation temps réel FreeRTOS.

Ce logiciel est uniquement disponible pour Windows, mais peut être installé sous Linux avec Wine. De plus, le code obtenu peut être utilisé avec Code Composer Studio sous Windows comme Linux puisque qu'il dépend du micro-contrôleur et non du système d'exploitation de l'ordinateur utilisé.

3.2.2 L'environnement de développement Code Composer Studio

CCS (Code Composer Studio) est un environnement de développement (IDE) basé sur Éclipse fourni par Texas Instruments pour leur processeurs embarqués. [9] Il comprend une suite d'outils pour développer et débugger des applications embarquées tels que les compilateurs nécessaires aux différents micro-contrôleurs TI, un éditeur de code source, un debuggeur, des simulateurs ...

CCS permet également de charger le programme sur le micro-contrôleur. Grâce au débuggeur il est possible de lancer une exécution pas à pas du programme sur le micro-contrôleur et d'observer les différentes variables, ce qui s'est avéré être très utile pour les mesures de coût mémoire. Il est disponible sous Windows et Linux.

3.2.3 FreeRTOS un système d'exploitation temps réel

FreeRTOS est un système d'exploitation temps réel (RTOS : Real Time Operating System) qui est porté sur une trentaine d'architectures différentes. [10]

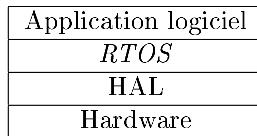


Figure 2 – Un système d'exploitation temps réel dans son environnement

Un RTOS a la particularité d'agir de façon déterministe, c'est-à-dire qu'on est capable de prédire quelles tâches seront exécutées, dans quel ordre et leur temps d'exécution maximum. Ce n'est pas le cas d'un OS classique tel que Windows ou Linux, car l'OS peut décider d'exécuter des tâches de fond comme le compactage de la mémoire à n'importe quel moment qui vont influencer l'ordonnancement des tâches. Ce type de système d'exploitation simplifie la création d'applications temps réel.

Le RTOS vient s'ajouter à la couche d'adaptation hardware comme le montre le modèle de la Figure 2. Cependant, dans le cas de FreeRTOS, le système d'exploitation peut être ajouté au code de configuration créé par HALCoGen. Malgré tout, les fichiers correspondant à l'OS sont distingués par leur nom dans la liste des fichiers générés par HALCoGen, ils commencent tous par *os_*.

Ce système d'exploitation a la particularité d'être léger, il est composé de moins 7000 lignes de code C et assembleur. Ces fichiers sont compilés en même temps que le reste du code source pour former une seule et même application sous la forme d'un binaire qui est ensuite chargée sur le micro-contrôleur.

De plus, FreeRTOS permet la création de tâches temps réel de différentes priorités et un ordonnanceur est déjà implémenté. L'ordonnanceur gère les tâches selon leur priorité puis il suit l'algorithme du *round robin* pour des tâches ayant la même priorité.

FreeRTOS est disponible sous une licence Open Source GPL (GNU General Public License) modifiée, cela signifie que FreeRTOS est un logiciel libre qui peut être utilisé à des fins commerciales gratuitement sans avoir besoin de publier le code source propriétaire. Une licence commerciale est également disponible sous le nom de OpenRTOS, celle-ci est payante et ne fait aucune mention de la licence GPL.

4 Prise en main

4.1 Installation des logiciels

Avant la prise en main du matériel, il faut commencer par installer les logiciels HALCoGen et CCS.

HALCoGen et CCS sont téléchargeables sur le site de Texas Instruments. HALCoGen est seulement disponible sous Windows mais peut être installé sous Linux avec Wine malgré quelques bugs d'affichage. CCS est disponible pour Windows et Linux. Il est cependant nécessaire d'installer certaines bilbiothèques avant d'installer CSS sous Linux. Un script shell vérifiant la présence des packages nécessaires est disponible à l'adresse suivante : `http://processors.wiki.ti.com/index.php/Linux_Host_Support`. Néanmoins comme indiqué sur le site web précédent, le script est une application 32 bits, il donc est nécessaire d'in-

staller des bibliothèques supplémentaire pour les utilisateurs de Linux 64 bits (32 bits library support package).

4.2 Générer un programme d'exemple utilisant FreeRTOS avec HALCoGen

Plusieurs exemples d'applications avec le TMS570 sont disponibles, tous ne sont pas fonctionnels. Le moyen le plus sûr d'obtenir un code fonctionnel est de le générer soi-même avec HALCoGen puis de l'importer dans CCS afin de le modifier. Il est notamment possible avec HALCoGen d'intégrer FreeRTOS lors de la génération du code pour le TMS570LS31x.

L'exemple *freeRTOSBlinky* fourni pour la série TSM570LS31x indique pas à pas la marche à suivre pour générer le code de configuration du micro-contrôleur avec FreeRTOS dans le but de faire clignoter les LEDs. Les étapes de configuration sont les suivantes :

- activer le port GPIO parmi les drivers
- configurer la gestion des interruptions en ajoutant le gestionnaire de FreeRTOS
- configurer la RAM du module de vecteur d'interruption (VIM : Vectored Interrupt Module) en ajoutant le gestionnaire pour le timer de FreeRTOS et pour le SSI
- configurer les chaînes du VIM, activer VIM et le lier aux interruptions matérielles (IRQ : Interrupt Request)
- configurer la fréquence du timer de l'OS

Il reste ensuite à lancer la génération du code de configuration, fait par HALCoGen, et l'importer dans CCS. La fonction principale *main* se trouve dans le fichier *sys_main.c*. Elle est laissée vide par HALCoGen, c'est avec CCS qu'on peut ensuite implémenter une application.

4.3 Modifier le programme d'exemple avec CCS

Afin de pouvoir modifier le programme d'exemple, il faut tout d'abord créer un projet dans lequel le code de configuration généré sera ajouté. On pourra ensuite créer des tâches temps réel et lancer l'ordonnanceur.

4.3.1 Création d'un projet

Lors de la création du nouveau projet de type *CCS project*, il faut que le projet soit situé au même endroit que celui créé avec HALCoGen et ait le même nom. Ainsi, le code généré avec HALCoGen sera automatiquement ajouté au projet CCS. De plus, lors de la création du projet il faut faire attention à la configuration du compilateur du projet. En effet, il faut préciser le micro-contrôleur utilisé, la FIGURE 3 montre la configuration requise pour le kit de développement TMS570LS31x.

Afin que le compilateur retrouve bien les headers situé dans le dossier *include*, il faut ajouté le chemin *${PROJECT_LOC}/include* dans les options d'inclusion du compilateur. Ensuite, le code donné dans l'exemple *freeRTOSBlinky* d'HALCoGen peux être copié collé dans le fichier *sys_main.c*. Cependant, pour que ce code fonctionne correctement et que les LEDs clignotent, il est nécessaire d'ajouter la bibliothèque *gio.h* dans *sys_main.c*.

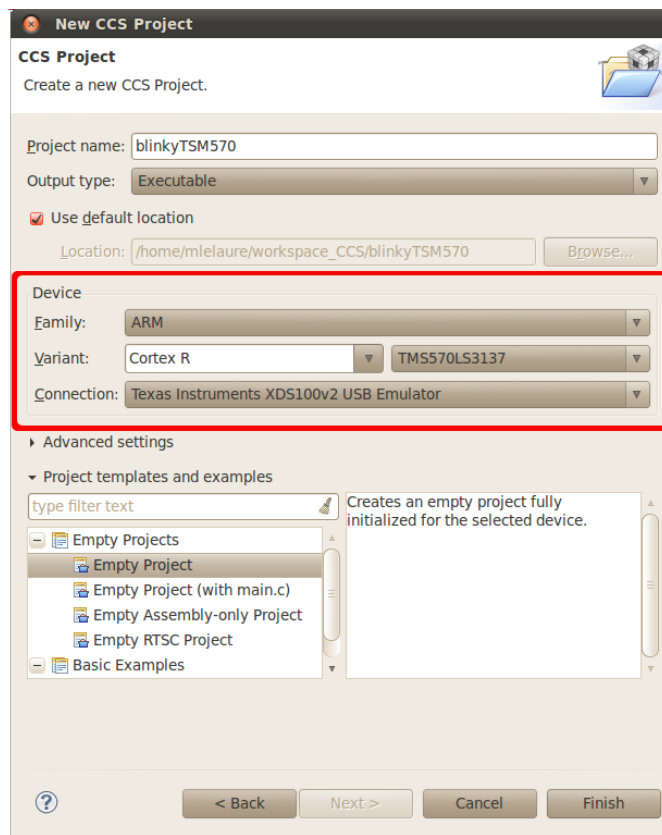


FIGURE 3 – Configuration du compilateur lors de la création d'un nouveau projet avec CCS

pvTaskCode	Pointeur de fonction. La tâche ne doit jamais se terminer (ie boucle sans fin)
pcName	Nom utilisé pour décrire la tâche. Il sert principalement pour faciliter le débuggage. Sa taille est définit par la constante *configMAX_TASK_NAME_LEN*
usStackDepth	Taille de la pile qui sera allouée à la tâche en nombre de mots.
pvParameters	Pointeur vers la structure qui est passé en paramètre lors de l'appel de la fonction.
uxPriority	Priorité de la tâche (0 étant la priorité la plus faible).
pvCreatedTask	Utilisé pour passer un handler afin de référencer la tâche.

FIGURE 4 – Paramètres de la fonction *vTaskCreate*

4.3.2 Ajout de tâches temps réel

Voici quelques fonctions de base qui permettent de créer une tâche, la supprimer et lancer l'ordonnanceur.

La fonction *xTaskCreate* permet la création d'une tâche. Sa déclaration est donnée ci-après et le détail de ses paramètres est donnée par la FIGURE 4. [11] La fonction retourne *pdPASS* si la tâche a bien été créé sinon un code d'erreur.

```
BaseType_t xTaskCreate(
                            TaskFunction_t pvTaskCode,
                            const char * const pcName,
                            unsigned short usStackDepth,
                            void *pvParameters,
                            unsigned BaseType_t uxPriority,
                            TaskHandle_t *pvCreatedTask
                          );
```

Une fois les tâches créées, il suffit de lancer l'ordonnanceur grâce à la fontion *xStartScheduler* pour obtenir une application temps réel. Ensuite, la fonction *vTaskDelete* permet de supprimer une tâche. Elle prend en paramètre le *TaskHandle_t* correspondant à la tâche qui doit être supprimée. Les déclarations de ces deux fonctions sont données ci-après.

```
void vTaskStartScheduler();
void vTaskDelete( TaskHandle_t xTask );
```

Il existe d'autre fonctions disponible dans la documentation qui permettent notamment d'arrêter ou suspendre l'ordonnanceur, de créer ou supprimer des sémaphores ou des mutexes.

Troisième partie

Expérimentations

5 Coûts mémoire

5.1 La gestion de la mémoire avec FreeRTOS

FreeRTOS propose quatre manières de gérer la mémoire, ou plutôt le tas. En effet, quatre fichiers *os_heap* numérotés de 1 à 4 sont disponibles. [12] Un seul de ces quatres fichiers peut être inclu lors de la compilation.

5.1.1 Allocation statique

os_heap_1 est la version la plus simple de gestion de mémoire, elle correspond à l'allocation statique. Seule la fonction *malloc* est implémentée, *free* ne l'est pas. Cette version peut donc être utilisée que pour des applications qui ne suppriment jamais de tâche, de file d'attente ou de sémaphore. *os_heap_1* est le type de gestion mémoire utilisé lorsqu'on utilise HALCoGen, les autres ne sont pas implémentés dans le code généré par celui-ci mais restent disponibles au téléchargement sur le site de FreeRTOS.

5.1.2 Allocation dynamique avec best fit

os_heap_2 est plus élaboré que *os_heap_1*, il correspond à l'allocation dynamique avec best fit. En effet, il utilise un algorithme qui correspond mieux à la mémoire pour *malloc* (*best fit*) et permet de la désallouer et donc de supprimer des tâches, des files d'attente ou sémaphores. L'algorithme du *best fit* s'assure que le bloc de mémoire alloué est celui qui correspond le mieux à la taille désirée, les blocs disponibles sont de 5, 25 ou 100 octets. Cependant, *os_heap_2* ne permet pas de combiner plusieurs blocs adjacent, pas de compactage de la mémoire possible, ce qui peut entraîner une fragmentation de celle-ci. Dans le cas où les blocs de mémoires qui sont alloués et désalloués sont de même taille, il n'y a pas de problème de fragmentation. *os_heap_2* peut-être utilisé pour des applications qui créent et suppriment souvent des tâches à condition que la quantité de mémoire allouée reste la même.

5.1.3 Wrapper des fonstions C

os_heap_3 est en fait un wrapper des fonctions *malloc* et *free* du langage C. Les fonctions font appel à celles existantes en C. Ce type de gestion augmente la taille du code du kernel de freeRTOS qui doit contenir les bibliothèques C nécessaires aux fonctions *free* et *malloc*.

5.1.4 Allocation dynamique avec best fit et compactage

os_heap_4 utilise le même type d'algorithme que *os_heap_2* avec en plus la possibilité de compacter la mémoire. Plusieurs blocs de mémoire adjacents peuvent donc être combinés. Le compactage permet de limiter la fragmentation de la mémoire.

Élément	*Nombre d'octets utilisés*
Ordonnanceur	236 octets
Pour chaque file d'attente créé	76 octets + capacité de stockage allouée par l'utilisateur
Pour chaque tâche créée	64 octets + capacité de mémoire allouée par l'utilisateur

FIGURE 5 – Coût mémoire théorique de FreeRTOS

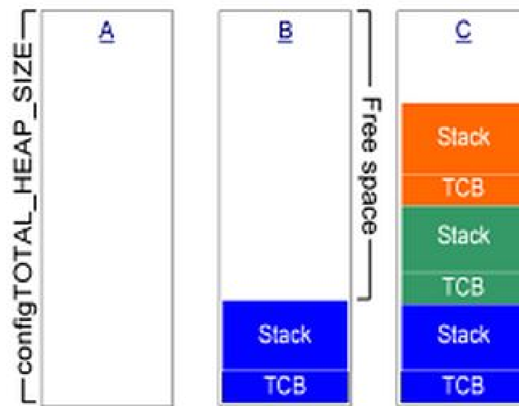


FIGURE 6 – Allocation de la mémoire RAM lors de la création d'un tâche

5.1.5 Choix de la gestion statique

Dans le cadre du projet, les tâches ne sont jamais supprimées, elles sont cependant appelées régulièrement. La création des tâches est gérée de manière statique, *os_heap_1* est donc suffisant pour nos mesures. Avant de réaliser les mesures on peut d'ores et déjà s'attendre à quelques valeurs concernant l'utilisation de la RAM grâce aux informations fournies par FreeRTOS données par la FIGURE 5. [13]

Intéressons nous donc de plus près à *os_heap_1*. La mémoire RAM allouée pour chaque tâche est divisée en deux parties. La première étant la mémoire nécessaire au système d'exploitation pour la gestion de la tâche (TCB : Task Control Block), elle est constante quelle que soit la tâche. La seconde est la quantité de mémoire allouée par l'utilisateur au moment de la création de la tâche. Elle sert au fonctionnement de la tâche en elle-même, cette valeur est donnée en nombre de mots (de 4 octets pour le TMS570 puisque c'est un 32 bits). La FIGURE 6 illustre l'allocation de mémoire à trois moments différents :

- A montre l'état de la mémoire avant la création de tâche
- B montre l'état de la mémoire après la création d'une tâche
- C montre l'état de la mémoire après la création de trois tâches

On souhaite par la suite vérifier ces valeurs par l'expérimentation.

5.2 Le protocole de test

Afin de mesurer l'impact mémoire des différentes tâches, la fonction *xPortGetFreeHeapSize* fournie par FreeRTOS a été utilisée. Elle permet de donner la quantité d'espace mémoire en octets restant sur le tas et s'utilise de la manière suivante.

```
size_t free_heap_size = xPortGetFreeHeapSize();
```

Afin de mesurer l'espace utilisé par chaque tâche, cette fonction est appelée avant et après la création des tâches puis après le lancement de l'ordonnanceur. Il n'est pas possible d'imprimer le résultat sur le terminal puisque l'application est exécutée sur le micro-contrôleur. Cependant, grâce au débuggeur il est possible d'exécuter le code pas à pas, de mettre des *breaking points* à des endroits stratégiques et de visualiser la valeur des variables du programme, c'est cela qui a été utilisé ici. Des *breaking points* ont été mis après chaque appel de la fonction *xPortGetFreeHeapSize* afin de lire le résultat stocké dans une variable locale.

L'exemple utilisé est composé d'un ensemble de six fonctions qui font chacune clignoter une LED différente du micro-contrôleur. Plusieurs variables ont été modifiées au cours des différents tests comme les valeurs des périodes des tâches et de la mémoire allouée pour celle-ci.

FreeRTOS possède un fichier de configuration appelé *FreeRTOSConfig.h*. Certaines valeurs qui déterminent les résultats suivants sont définies dans ce fichier, notamment *configMINIMAL_STACK_SIZE* qui définit la taille minimale de la pile en nombre de mots. Toutes les constantes commençant par *config* sont définies dans ce fichier. Il est également possible de configurer la taille du tas avec *configTOTAL_HEAP_SIZE* ou le nombre maximum de priorité avec *configMAX_PRIORITIES*.

5.3 Les résultats obtenus

Comme l'indique la Figure 6, FreeRTOS indique le nombre d'octets utilisés par chaque tâche, cette valeur sera nommée valeur théorique par la suite. Elle se calcule de la manière suivante :

$$valeurTheorique = 64 + memoireAllouee * 4$$

L'objectif de ces expérimentations est donc de vérifier cette valeur théorique.

Tout d'abord, afin de vérifier la répétabilité des valeurs obtenues, un unique test a été lancé à plusieurs reprises dans lequel la fonction était appelée avant la création de la première tâche et avant le lancement de l'ordonnanceur. Les valeurs obtenues étant toujours identiques à l'octet près, on peut en déduire que le résultat obtenu est invariable lorsque le code est inchangé, pour le reste des mesures il n'est donc pas nécessaire de réaliser le même test plusieurs fois.

Ensuite, pour s'assurer que la création de variables, l'utilisation de la fonction *xGetPortHeapSize()* et du débuggeur n'influencent pas les résultats, plusieurs tests ont été lancés avec une ou plusieurs créations de variables et l'utilisation de la fonction à divers endroits. De même que le test précédent, les valeurs obtenues sont identiques. Ce résultat est conforme aux attentes puisque l'allocation de la mémoire est statique.

Mémoire allouée (en octet)	Valeur théorique (en octet)	Valeur mesurée (en octet)	Ecart (en octet)
64	320	408	88
128	576	664	88
199	860	952	92
200	864	952	88
201	868	960	92
202	872	960	88
203	876	968	92
204	880	968	88
255	1084	1176	92

FIGURE 7 – Coûts mémoire d'une tâche en fonction de la mémoire allouée

Puis, afin de vérifier la valeur théorique une série de tests avec des quantités de mémoire allouée différentes a été mise en place. Plusieurs variations de l'exemple de base ont été étudiées, différentes quantités de mémoire allouée, périodes et priorités. Le nombre de tâches et de priorités maximum avec *configMAX_PRIORITIES* ont également été modifiées, ainsi que la taille minimale de la pile, *configMINIMUM_STACK_SIZE*. Afin de vérifier quel paramètre engendre une modification des résultats chacun des paramètres a été modifié un par un.

Les résultats obtenus sont visibles sur la FIGURE 7. On remarque un écart constant entre la valeur théorique et la valeur obtenue qui est soit de 88 octets si la quantité de mémoire est paire, soit de 92 octets sinon. On peut donc en déduire que la quantité de mémoire allouée par le système d'exploitation est prévisible et varie selon la quantité de mémoire allouée par l'utilisateur lors de la création de la tâche.

On peut ensuite se demander : À quoi correspond cet écart ? Quelques tests supplémentaires en modifiant d'autres paramètres tels que l'utilisation des préemptions ou l'utilisation des *runTimeStats* pouvaient faire varier ces résultats de quelques octets. De plus, afin de faire les mesures l'application est compilée en mode débug, il se peut que cela influence la quantité de mémoire dont ont besoin les tâches créées.

De plus, il s'avère que l'ordonnanceur utilise également une partie de mémoire allouée sur le tas. En effet, des mesures avant et après le lancement de l'ordonnanceur ont montré qu'une partie de mémoire est allouée pour lui. De plus, cette quantité de mémoire suit la même loi que les tâches créées avec pour nuance que la mémoire allouée lors de la création de la tâche est ici la constante *configMINIMAL_STACK_SIZE*.

Pour conclure sur les résultats du coûts mémoires des tâches, ce coût varie principalement en fonction de la quantité de mémoire donnée par l'utilisateur lors de la création de la tâche et suis la loi suivante avec *memoireConfig* qui vaut 88 ou 92 avec la configuration utilisée ici.

$$valeurMesuree = 64 + 4 * memoireAllouee + memoireConfig$$

Ainsi, l'utilisation d'un grand nombre de tâches implique un surcoût mémoire

d'environ 150 octets par tâche. Ce coût est faible comparé à la quantité totale de mémoire utilisée par tâche. Cependant, rassembler des tâches ensemble avec le task clustering permet de gagner un peu d'espace mémoire. Pour que ce coût soit significatif il faudrait diviser par 10 le nombre de tâches, par exemple, passer de 100 à 10 permettrait de gagner environ 13,5 KB.

6 Temps d'exécution

On souhaite mesurer le temps d'exécution de chacune des tâches, FreeRTOS permet cela avec la fonction *vTaskGetRunTimeStats*. Elle permet, de la même manière que *xPortGetFreeHeapSize*, d'obtenir le temps d'exécution de chaque tâche. Cependant, son utilisation n'est pas aussi simple.

6.1 Utilisation de l'outil Run Time Statistics

FreeRTOS est capable de collecter des informations concernant le temps passé à exécuter chaque tâche. La fonction *vTaskGetRunTimeStats* renvoie ces informations dans un buffer avec pour chacune des tâches le temps absolu et le pourcentage du temps total. [14] Cette fonction s'utilise de la manière suivante :

```
static char cBuf[200];
vTaskGetRunTimeStats((signed char*) cBuf);
```

Avant de pouvoir utiliser cette fonction quelques configurations sont nécessaires. Elles peuvent être faites dans le header *FreeRTOSconfig*.

Il faut tout d'abord activer la génération des statistiques en mettant la constante *configGENERATE_RUN_TIME_STATS* à 1. Puis, définir deux fonctions qui vont permettre le calcul des données statistiques.

```
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() configTimer()
#define portGET_RUN_TIME_COUNTER_VALUE() getTime()
```

La première fonction doit configurer le timer qui sera utilisé pour faire les mesures. FreeRTOS recommande que ce timer soit entre 10 et 100 fois plus rapide que celui générant les ticks d'interruptions (*configTICK_RATE_HZ*). La seconde fonction doit renvoyer le nombre de ticks effectués par le timer depuis le lancement de celui-ci. Le lancement du timer se fait pendant sa configuration au moment du lancement de l'ordonnanceur. Ensuite, la fonction *portGET_RUN_TIME_COUNTER_VALUE* est appelée à chaque changement de contexte afin de calculer le temps réellement passé à exécuter la tâche.

6.2 Les problèmes rencontrés

Dès les premiers essais d'utilisation de la fonction *vTaskGetRunTimeStats*, les résultats obtenus étaient étranges. En effet, une seule fonction prenait 100% du temps, quant aux autres leur temps absolu était nul.

La première solution a donc été d'ajouter une boucle dans les différentes tâches, en plus du clignotement des LEDs, afin que leur temps d'exécution soit mesurable. Les résultats étaient toujours nuls, mais les LEDs qui devaient clignoter en même temps se désynchronisaient. On peut donc en déduire que les

calculs prenaient bel et bien du temps d'exécution mais qu'ils n'influençaient pas les statistiques. Le problème venait donc d'ailleurs.

Après plusieurs tests et une étude plus approfondie du code de FreeRTOS correspondant à la fonction de statistique, il s'est avéré que les fonctions utilisées étaient mal défini. Il faut en fait créer un autre timer et écrire la configuration du micro-contrôleur. La configuration d'un timer est fastidieuse et aucun exemple pertinent permettant se mise en place n'a été trouvé. Le temps restant pour le projet étant compté il fallait trouver une autre solution.

6.3 Différentes solutions envisageable

Plusieurs solutions sont envisageables afin de parer ce problème de configuration du timer.

6.3.1 Utiliser le timer interne

FreeRTOS doit déjà disposer d'un timer interne afin de générer les ticks d'interruptions des tâches. Il est envisageable d'utiliser ce timer plutôt que d'en configurer un autre. Cependant, le résultat sera moins précis puisque FreeRTOS conseil de configurer le timer de telle sorte qu'il soit entre 10 et 100 fois plus rapide que les ticks d'interruptions.

La partie du code correspondant aux ticks d'interruptions n'est pas disponible, cela complique donc son utilisation.

6.3.2 Configurer avec HALCoGen

Il est possible avec HALCoGen d'ajouter les statistiques de temps d'exécution à la configuration du micro-contrôleur. Il s'avère qu'HALCoGen ne fait que mettre la valeur de *configGENERATE_RUN_TIME_STATS* à 1. Il est envisageable de configurer un timer avec celui-ci et de l'utiliser pour faire les statistiques. Cependant, une connaissance plus approfondie d'HALCoGen est nécessaire et la durée du projet ne l'a pas permise.

6.3.3 Utiliser un autre logiciel : Tracealyzer

Tracealyzer est un logiciel fabriqué par Percepio, une entreprise suédoise. Il permet entre autres de visualiser les appels aux différentes tâches et leur durés. Il existe une version de ce logiciel adapté à FreeRTOS qui s'appelle FreeRTOS+Trace. [15] Elle a notamment été utilisée par des étudiants faisant une master thesis pour mesurer le temps d'exécution des tâches sur un TMS570LS3137 avec FreeRTOS.[16]

Cependant, FreeRTOS+Trace est disponible uniquement pour Windows et ne fonctionne pas sous Linux, même avec Wine. Selon le service technique, une version Linux devrait être disponible dans un ou deux mois. Il existe une version Linux de Tracealyzer mais c'est un exécutable, il ne peut donc fonctionner que sous Windows. De même, il ne fonctionne pas sous Linux avec Wine.

Malgré cela, l'installation sous Windows est possible et a permis de voir que le logiciel permet de visualiser la trace créer par l'application. Il est en effet possible de configurer FreeRTOS avoir d'obtenir la trace de l'application. Il est également possible qu'il faille utiliser une API particulière afin de créer le type de trace attendue par Tracealyzer.

Cependant, le manque de temps et le fait de devoir changer de système d'exploitation pour utiliser Windows, ont fait que cette possibilité n'a pas été approfondie.

Conclusion

L'objectif de ce projet est de mesurer l'impact de l'utilisation d'un grand nombre de threads dans un système temps réel en terme de coût mémoire, de temps d'exécution et changements de contexte.

A l'issue de ce projet, on peut donner une réponse en ce qui concerne le coût mémoire. La création de chaque thread a un coût mémoire. En effet, chaque thread créé a besoin d'une certaine quantité de mémoire en plus de celle nécessaire à l'exécution de la tâche. Cependant, ce coût reste faible par rapport à la quantité de mémoire nécessaire à la tâche elle-même.

Pour ce qui est des temps d'exécution, on ne peut rien conclure car les mesures n'ont pas pu être réalisées. Cependant, plusieurs solutions ont été identifiées afin d'être en mesure de les faire. La question des changements de contexte n'a pas été abordée.

Malgré une longue prise en main du matériel ce projet fut très intéressant. Il m'a permis notamment d'approfondir ma connaissance des systèmes temps réel. Plus précisément, ce projet m'a permis d'étudier un système d'exploitation temps réel et de rencontrer les problèmes liés à sa mise en place sur un microcontrôleur. J'ai donc pu aborder quelques problématiques liées à l'utilisation de systèmes temps réel critiques.

Bibliographie

Références

[1] LIFL, Présentation. [Online] http://www.lifl.fr/Presentation/

[2] IRCICA. [Online] http://www.ircica.univ-lille1.fr/?lang=fr

[3] LIFL, Equipe DART. [Online] http://www.lifl.fr/Recherche//Equipes/Presentation?eid=7

[4] Emeraude, Présentation. [Online] http://www.lifl.fr/emeraude/

[5] A. Bertout, J. Forget and R. Olejnik, A heuristic to minimize the cardinality of a real-time task set by automated task clustering. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC 2014)*, Gyeongju, Korea, March 2014.

[6] Texas Instruments, TMS570LS31x Hercules USB Stick Development Kit. [Online] http://www.ti.com/tool/tmdx570ls31usb#descriptionArea

[7] Texas Instruments, TMS570LS3137 16/32-Bit RISC Flash Microcontroller. [Online] http://www.ti.com/lit/ds/symlink/tms570ls3137.pdf

[8] Texas Instruments, HAL Code Generator tool. [Online] http://www.ti.com/tool/halcogen

[9] Texas Instruments, Code Composer Studio (CCStudio) Integrated Development Environment (IDE) v5 [Online]. http://www.ti.com/tool/ccstudio

[10] FreeRTOS, About FreeRTOS. [Online] http://www.freertos.org/RTOS.html

[11] FreeRTOS, FreeRTOS API Reference. [Online] http://web.ist.utl.pt/~ist11993/FRTOS-API/group___tasks.html

[12] FreeRTOS, Memory Management. [Online] http://www.freertos.org/a00111.html

[13] FreeRTOS, FreeRTOS FAQ - Memory Usage, Boot Times & Context Switch Times. [Online] http://www.freertos.org/FAQMem.html#RAMUse

[14] FreeRTOS, Run Time Statistics. [Online] http://www.freertos.org/rtos-run-time-stats.html

[15] Percepio, FreeRTOS+Trace. [Online] http://percepio.com/docs/FreeRTOS/manual/index.html#FreeRTOS_Trace

[16] A. Angerd and A. Johansson, Design and Implementation of a central control unit in an automotive drive-by-wire system. Master thesis. Chalmers University of Technology, Gothenburg, Sweden, September 2013