6 Systèmes d'exploitation temps réel et multitâches

Description générale

- **Date du TP :** 11ième semaine ;
- **Durée du TP :** 4,5 h (3 semaines) ;
- **Matériels nécessaires :** Carte d'expérimentation, microcontrôleur PIC 18f, programmateur, des composants électroniques, plaque d'essai ;
- **Logiciels utilisés :** RTOS, Complilateur C PCW, Simulateur ISIS, IC Writer ;
- **Public cible :** 4ième niveau d'études de technicien supérieur en informatique industrielle ;
- **Partie du cours en rapport avec le TP :** Classification logicielle des systèmes temps réel, systèmes multitâches.

Objectifs

- Explorer les bases des systèmes embarqués multitâches ;
- Apprendre les systèmes d'exploitation temps réel RTOS ;
- Assimiler les notions par un exemple d'un RTOS utilisé dans des projets simples ;
- Approfondir les acquis en programmation C des microcontrôleurs PIC ;
- Maîtrise des architectures des microcontrôleurs Microchip PIC18.

Pré requis

Notions générales sur :
- Système d'exploitation ;
- L'électronique ;
- Algorithmique ;
- Programmation en C ;
- Les microcontrôleurs.

6.1 introduction

Maintenant, presque tous les systèmes basés sur des microcontrôleurs exécutent plus qu'une activité. Par exemple, un système qui contrôle une température est composé de trois tâches qu'ils se répètent après un court délai, à savoir :

- Tâche 1 : lire la température ;
- Tâche 2 : changer le format de la température
- Tâche 3 :affichage de la température

On signale que la plus part des systèmes complexes peuvent avoir plusieurs tâches complexes. Dans un système multitâches, nombreuses tâches nécessitent le CPU au même temps, mais la on dispose seulement d'un seul CPU on a donc besoin de quelques formes d'organisation et de coordination qui permettent à chaque tâche d'avoir le temps de CPU qu'elle a besoin. Dans la pratique, chaque tâche prend un temps très bref, il paraît donc comme si toutes les tâches sont exécutées en parallèle et simultanément.
Presque tous les systèmes basés sur des microcontrôleurs travaillent en temps réel. Un système temps réel est un système sensible au temps qui peut répondre à son environnement dans le temps le plus court possible. Le temps réel ne veut pas dire nécessairement que le microcontrôleur devrait opérer à haute vitesse. Ce qui est important dans un système temps réel c'est qu'il doit avoir un temps de réponse rapide, bien qu'une haute vitesse puisse aider.
Par exemple, on attend d'un système temps réel basé sur des microcontrôleurs et qui a plusieurs interrupteurs externes de répondre immédiatement quand un interrupteur est activé ou lorsqu'un autre événement se produit. Un système d'exploitation temps réel (RTOS) est un code qui contrôle l'allocation des tâches quand le microcontrôleur opère dans un environnement multitâches. Par exemple un STR décide quelle est la tâche suivante qui sera exécutée ou comment coordonner la priorité des tâches, et comment passer les données et les messages entre les tâches. Cette partie explore les principales bases des systèmes embarqués multitâches et donne des exemples d'un STR utilisé dans des projets simples.
Le code des STR multitâches sont des sujets compliqués et larges, et ce cours décrit des concepts qui concernent ces derniers seulement brièvement. Les lecteurs intéressés devraient faire référence aux nombreux livres et papiers disponibles sur les systèmes d'exploitation, les systèmes multitâches, et les STR.
Il y aplusieurs systèmes STR disponibles sur le marché pour les PIC microcontrôleurs. Les plus connus des systèmes STR de haut niveau pour les PIC sont Salvo (www.pumpkin.com), ce dernier peut être utilisé à partir d'un compilateur C de PIC :High-tech, et le CCS (Custom Computer Services) qui intègrent des systèmes RTOS. Dans cette partie, les exemples des projets STR sont basés sur le compilateur CCS (www.ccsinfo.com), qui est un des compilateurs PIC C les plus utilisés qui est développé pour les séries des microcontrôleurs : PIC16 et PIC18.

6.2 Machines d'état

Les machines d'état sont des constructions simples utilisées pour exécuter plusieurs activités, généralement en suivant une séquence .On remarque plusieurs systèmes temps réel qui appartiennent à cette catégorie. Par exemple, l'opération d'une machine à laver ou d'un lave-vaisselle est décrite simplement avec une instruction des Machines d'état. Peut-être la méthode la plus simple de construire des machines d'état dans le C est d'utiliser une instruction switch-case. Par exemple, notre système qui contrôle la température est composé de trois tâches nommées : Tâche 1, Tâche 2, et Tâche 3, comme il est montré dans la figure 6.1.

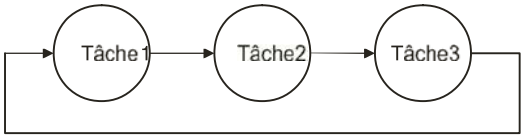

FIG. 6.1 – Implémentation de la machine d'état

```
for(;;)
{
        state = 1;
        switch (state)
        {
                CASE 1:
                        implement TASK 1
                        state++;
                break;
                CASE 2:
                        implement TASK 2
                        state++;
                break;
                CASE 3:
                        implement TASK 3
                        state = 1;
                break;
        }
Delay_ms(n);
}
```

FIG. 6.2 – Implémentation en C de la machine d'état

La figure 6.2 montre les machines de l'état réalisée suivant les trois tâches et qui utilisent des switch-case . L'état initial est 1, et chaque tâche incrémente le nombre de l'état par

un pour sélectionner l'état suivant qui sera exécuté. Le dernier état sélectionne l'état 1, et il y a un délai à la fin de switch-case. La construction de l'état de la machine est exécutée de façon continue dans une boucle sans fin.
Dans plusieurs applications, les états n'ont pas besoin être exécutés en séquence. Plutôt, le prochain état est sélectionné par l'état présent soit directement soit suivant quelques conditions. Cela est montré dans la figure 6.3.

```
for(;;)
{
        state = 1;
        switch (state)
        {
        CASE 1: implement TASK 1
                state = 2;
        break;
        CASE 2: implement TASK 2
                state = 3;
        break;
        CASE 3: implement TASK 3
                state = 1;
        break;
        }
Delay_ms(n);
}
```

FIG. 6.3 – sélectionner le prochain état à partir de l'état courant

La machines d'état, bien qu'elle soit facile à implémenter, elle est primitive et elle a des applications limitées.
Ils peuvent être utilisés seulement dans les systèmes qui ne sont pas vraiment sensibles, où les activités de la tâche sont précis et sans priorité.
De plus, quelques tâches peuvent être plus importantes qu'autres. Nous pouvons vouloir que quelques tâches s'exécutent toutes les fois qu'ils deviennent éligibles.

6.3 Le système d'exploitation Temps réel (RTOS)

Les RTOS (Real Time Operating System) sont construits autour d'un noyau multitâche qui contrôle l'allocation de laps du temps des tâches. Un laps du temps est la période de temps pour l'exécution d'une tâche donnée avant qu'elle sera arrêtée et remplacé par une autre tâche. Ce processus est connu par le changement de contexte, qui se répète de

façon continue. Quand le changement du contexte se produit, la tâche est stoppée, les registres du processeur sont sauvegardés dans la mémoire, les registres du processeur de la prochaine tâche disponible sont chargés dans le CPU et la nouvelle tâche commence sa exécution.
Un RTOS fournit aussi les messages qui passent entre les tâches, synchronisation des tâches et allocation de ressources partagées aux tâches.
Les parties de base d'un RTOS sont :
– l'ordonnanceur (Scheduler ;
– services RTOS ;
– Synchronisation et outils de la messagerie ;

6.3.1 L'ordonnanceur (The Scheduler)

L'ordonnanceur est au noyau de chaque RTOS, car il fournit les algorithmes pour choisir les tâches pour l'exécution. Trois types d'algorithmes d'ordonnancement existent :
– Ordonnancement coopératif
– Ordonnancement en anneau à jeton
– Ordonnancement préemptif
l'ordonnacement coopératif est l'algorithme d'ordonnacement le plus simple et disponible. Chaque tâche s'exécute jusqu'à ce qu'elle soit terminé et renonce à l'unité centrale de traitement CPU volontairement. L'ordonnancement coopératif ne peut pas satisfaire aux besoins du système temps réel, puisqu'elle ne peut pas soutenir la priorité des tâches selon l'importance. En outre, une tâche simple peut utiliser l'unité centrale de traitement trop longue, tout en laissant un peut de temps pour les autres tâches.
Et l'ordonnanceur ne peut pas controler les diverses temps d'exécution des tâches.Une construction de machine d'état est une forme simple d'une technique d'ordonnancement coopérative.
Dans l'ordonnanceur en anneau à jetont, chaque tâche est assignée une part de temps égale du CPU (voir La figure 6.4). Un compteur donne une tranche de temps pour chaque tâche. Quand une tranche du temps de la tâche accomplit, le compteur est mis à zéros et la tâche est placée à la fin du cycle. Et de nouveau, des tâches supplémentaires sont placées à la fin du cycle avec leurs compteurs mis à zéros. Ceci, comme l'ordonnaceur coopératif, n'est pas très utile dans un système en temps réel, car très souvent certains tâches prennent seulement quelques millisecondes tandis que d'autres exigent des centaines de millisecondes ou plus.

FIG. 6.4 – l'ordonnanceur en anneau à jetont

L'ordonnanceur préemptif est considéré comme un algorithme d'ordonnancement en temps réel.Il est basé sur la priorité, pour chaque tâche est accordée une priorité (voir la figure

6.5).La tâche avec la plus haut priorité obtient le temps de CPU et elle se met en exécution. Les systèmes temps réel soutiennent généralement des niveaux de priorité de 0 à 255, où 0 est la priorité la plus élevée et 255 est la plus bas.

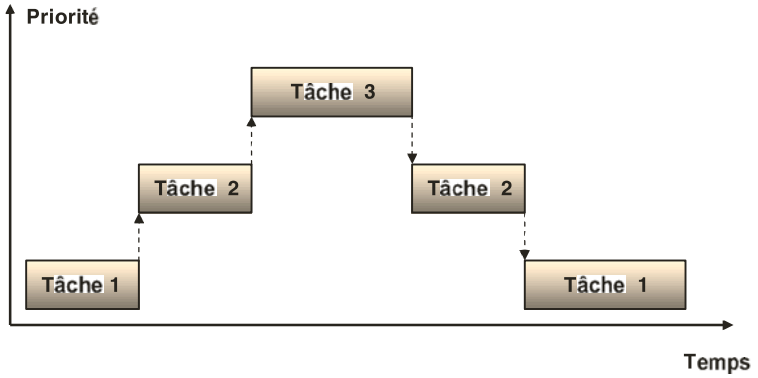

FIG. 6.5 – l'ordonnanceur préemptif

Dans quelques systèmes temps réel plus d'une tâche peut être au même niveau prioritaire, L'ordonnanceur préemptif est combiné à l'ordonnanceur en anneau à jeton. Dans ce cas, les tâches à haute priorité s'exécutent avant ceux qui possèdent une faible priorité, par contre les tâches au même niveau prioritaire s'exécutent par l'ordonnanceur en anneau à jeton. Si une tâche est préemptée par une tâche plus prioritaire, le compteur de temps est sauvegardé puis restaurer quand il regagne le contrôle de CPU.
Dans un système en temps réel, une tâche peut être dans les états suivants (voir la figure 6.6) :
– Prêt
– En exécution
– Bloqué

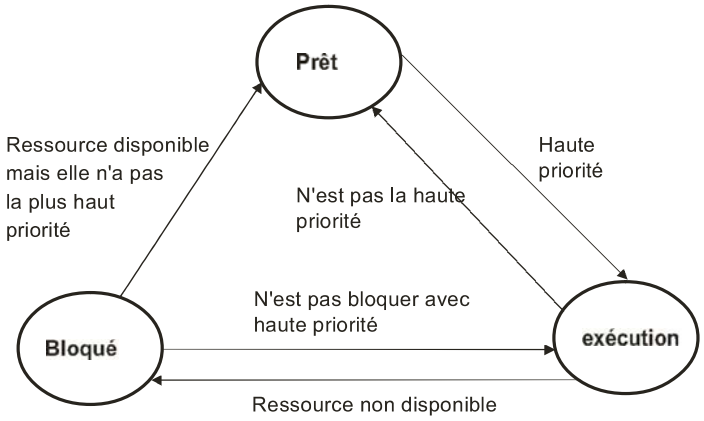

FIG. 6.6 – les états d'une tâche

Quand une tâche est d'abord créée, elle est habituellement à l'état prêt et elle entre dans la liste des tâches. De cet état, selon l'algorithme d'ordonnancement, la tâche peut devenir en exécution. Une tâche en exécution devient une tâche bloquée si elle a besoin d'une ressource qui n'est pas disponible. Par exemple, une tâche peut avoir besoin des données d'un convertisseur A/D alors elle sera bloqué jusqu'à ce qu'elle les reçoit.
Une fois que la ressource peut être accédée, la tâche bloquée devient une tâche en exécution si elle est la plus prioritaire dans le système, si elle n'est pas prioritaire alors elle se déplace à l'état prêt. Seulement une tâche en exécution peut être bloquée. Une tâche prête ne peut pas être bloquée. Quand une tâche se déplace d'un état à l'autre, le processeur sauve le contexte de la tâche courante dans la mémoire, charges le contexte de la nouvelle tâche de la mémoire et exécute alors les nouvelles instructions.
Le noyau temps réel fournit habituellement une interface pour réaliser des opérations sur la tâche.
Les opérations typiques sur une tâche sont :
– Création de la tâche ;
– Suppression d'une tâche ;
– Changement de la priorité d'une tâche ;
– Changement de l'état d'une tâche.

6.3.2 Services de RTOS

Les services de RTOS sont utilités par le noyau pour créer les tâches efficacement. Par exemple, une tâche peut employer des services de temps pour obtenir la date du jour et l'heure.
Certains de ces services sont :
– Services manipulant les interruptions ;
– Services de temps ;
– Services de gestion du matériel ;
– Services de gestion de mémoire ;
– Services d'entrée-sortie

6.3.3 Outils de synchronisation et de transmission de messages

Les outils de synchronisation et de transmission de messages sont construit par le noyau, ils aident à développer et à créer des applications en temps réel.
Certains de ces services sont :
– Sémaphores
– Drapeaux d'événement
– Boîtes aux lettres
– Pipes Files d'attente de message
Les sémaphores sont utilisés pour synchroniser l'accès aux ressources partagées, telles que des secteurs de données communs.
Des drapeaux d'événement sont employés pour synchroniser les activités entre Tâches.
Boîtes aux lettres, pipes, et les files d'attente des messages sont employées pour envoyer

des messages entre les tâches.

6.4 Compilateur C de CCS et RTOS

La langue de CCS fournit les fonctions suivantes de RTOS ainsi que les fonctions normales en C :

- rtos_run() : initialise l'opération de RTOS. Toutes les opérations de contrôle des tâches sont mis en application après avoir appelé cette fonction.
- rtos_terminate() termine l'opération de RTOS. Contrôle les retours au programme original sans RTOS. En faite, cette fonction est comme un retour de rtos_run ().
- rtos_enable() reçoit le nom d'une tâche comme argument. Cette fonction active la tâche et la fonction rtos_run() peut appeler la tâche quand son temps est dû.
- rtos_disable() reçoit le nom d'une tâche comme argument. Cette fonction désactive la tâche ainsi elle ne sera plus appeler par le rtos_run() à moins qu'elle soit appelé à nouveau en appelant rtos_enable().
- rtos_yield(), une fois appelé en dedans d'une tâche, elle retourne le contrôle au dispatcheur. Tous les tâches devraient appeler cette fonction pour libérer le processeur ainsi d'autres tâches peuvent utiliser le temps de processeur.
- rtos_msg_send() reçoit un nom de tâche et un octet comme arguments. La fonction envoie l'octet à la tâche indiquée, où elle est placée dans la file d'attente de message de la tâche.
- rtos_msg_read() lit l'octet situé dans la file d'attente de message de la tâche.
- rtos_msg_ poll() retourne vrai s'il y a des données dans la file d'attente de message de la tâche. Cette fonction devrait s'appeler avant de lire l'octet de la file d'attente de message de la tâche.
- rtos_signal() reçoit le nom de sémaphore et incrémente cette sémaphore.
- rtos_wait() reçoit un nom de sémaphore et attend la ressource liée au sémaphore à devenir disponible. Le compteur de sémaphore est alors ainsi décrémenté la tâche peut réclamer la ressource.
- rtos_await() reçoit une expression comme argument, et la tâche attend jusqu'a l'expression soit vrai.
- rtos_overrun () reçoit un nom de tâche comme argument, et la fonction renvoie vrai si cette tâche a débordé son temps assigné.
- rtos_stats () retourne les statistiques d'une tâche indiquée. Les statistiques peuvent être les temps minimum et maximum d'exécution de la tâche et tout le temps d'exécution de la tâche. Le nom de tâche et le type de statistiques sont indiqués comme arguments à la fonction.

6.5 Préparation de RTOS

En plus des fonctions précédentes, la commande de préprocesseur #use rtos() doit être indiqué au début du programme avant d'appeler n'importe quel fonction de l'RTOS.
Le format de cette commande de préprocesseur est : #use rtos(timer=n, minor_cycle=m)

où le timer est entre 0 et 4 et spécifie au processeur le timer qui sera employé par le RTOS et le minor_cycle est le plus long temps d'exécution de n'importe quelle tâche. Le nombre écrit ici doit être suivi de s, ms, us ou de ns.

6.6 Déclaration d'une tâche

Une tâche est juste déclarée comme n'importe quelle autre fonction de C, mais la tâche dans une application multitâche n'a aucun argument et ne retourne aucune valeur. Avant qu'une tâche soit déclarée, a la commande de préprocesseur #task est nécessaire pour indiquer les options de la tâche.
Le format de cette commande de préprocesseur est : #task(rate=n, max=m, queue=p)
où le paramètre rate indique combien de fois la tâche devrait s'appeler. Le nombre indiqué doit être suivi par s, ms, us ou de ns.
Le paramètre max indique combien de temps de processeur occupé par tâche lors de son exécution. Le temps spécifié ici doit être moins ou égal au temps indiqué par le minor_cycle.
Le paramètre queue est optionnel, et il indique nombre d'octets à réserver pour la tâche pour recevoir des messages d'autres tâches. La valeur par défaut est 0. Dans l'exemple suivant, une tâche appelée my_ticks est appelé chaque 20ms et elle n'emploie pas plus que 100ms de temps de processeur. Cette tâche est indiquée sans l'option de queue :

```
#task(rate=20ms, max=100ms)
void my_ticks()
{
.....
.....
}
```

6.7 Exemple d'application

Dans le projet suivant, basé sur l' RTOS, quatre LED sont reliées à la moitié inférieure de PORTB d'un microcontrôleur de type PIC18F452.ce programme se compose de quatre tâches, où chaque tâche clignote une LED à différents fréquence :
- La tâche 1, appelée le task_B0, clignote la LED reliée au port RB0 à un taux de 250ms.
- la tâche 2 de, appelée le task_B1, clignote la LED reliée au port RB1 à un taux de 500ms.
- La tâche 3, appelée le task_B2, clignote la LED reliée au port RB2 une fois par une seconde.
- La tâche 4, appelée le task_B3, clignote la LED reliée au port RB3 une fois toutes les deux secondes.

La figure 6.7 montre le schéma de circuit du projet. Un cristal 4MHz est employé comme horloge. Les pins RB0-RB3 de PORTB sont reliées aux LED par les résistances de protection.

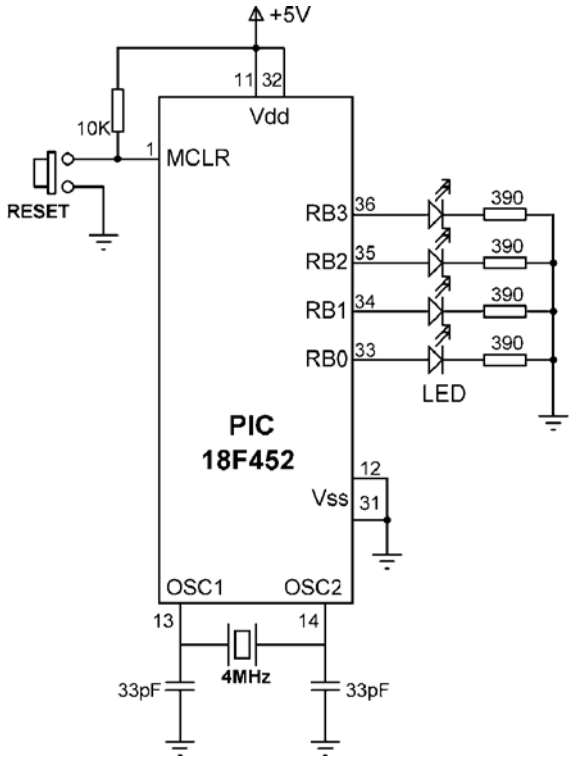

FIG. 6.7 – le circuit du projet

Le logiciel est basé sur le compilateur C de CCS, et le programme est donné dans listing 7.

Dans le programme principale, les pins du PORTB sont déclarées comme des sorties et le RTOS est exécuté en appelant la fonction rtos_run (). Le chémin du fichier de déclaration de CCS RTOS devrait être inclus au début de le programme. La commande de préprocesseur # use delay dit au compilateur que nous employons une horloge 4MHz. Le Timer de RTOS est déclaré comme Timer 0 et le temps de minor_cycle est déclarer à 10ms en utilisant la commande de préprocesseur #use rtos. Le programme se compose de quatre tâches semblables :

- La tâche task_B0 clignote la LED reliée à RB0 à un taux de 250ms, la LED est allumée pendant 250ms, puis éteinte pendant 250ms, et ainsi de suite. La fonction output_toggle est utilisé pour changer l'état de la LED chaque fois que la tâche s'appelle. Dans le compilateur CCS, PIN_B0 se réfère à la pin RB0 du microcontrôleur.
- La tâche task_B1 clignote la LED reliée à RB1 à un taux de 500ms.
- La tâche task_B2 clignote la LED reliée à RB2 chaque seconde.
- Finalement, la tâche task_B3 clignote la LED reliée à RB3 toutes les deux secondes

Le programme donné sur le listing xxx est un programme multitâche où les LED clignotent indépendamment les uns des autres et concurremment.

6.8 Travail demandé

1. Lire la partie théorique pour bien assimiler les notions théoriques sur l'RTOS.
2. Commenter le listing.
3. Compiler le programme avec PCW par la suite vérifier son bon fonctionnement à l'aide de la simulation par ISIS.
4. Câbler le schéma sur une plaque d'essai puis programmer le microcontrôleur.

```
//*************************listing7*********************//
#include <rtos.h>
#use delay (clock=4000000)
//
// Define which timer to use and minor_cycle for RTOS
//
#use rtos(timer=0, minor_cycle=10ms)
//
// Declare TASK 1 - called every 250ms
//
#task(rate=250ms, max=10ms)
void task_B0()
{
output_toggle(PIN_B0); // Toggle RB0
}
//
// Declare TASK 2 - called every 500ms
//
#task(rate=500ms, max=10ms)
void task_B1()
{
output_toggle(PIN_B1); // Toggle RB1
}
//
// Declare TASK 3 - called every second
//
#task(rate=1s, max=10ms)
void task_B2()
{
output_toggle(PIN_B2); // Toggle RB2
}
//
// Declare TASK 4 - called every 2 seconds
//
#task(rate=2s, max=10ms)
void task_B3()
{
output_toggle(PIN_B3); // Toggle RB3
}
//
// Start of MAIN program
//
void main()
{
set_tris_b(0); // Configure PORTB as outputs
rtos_run(); // Start RTOS
}
//*****************************************************//
```