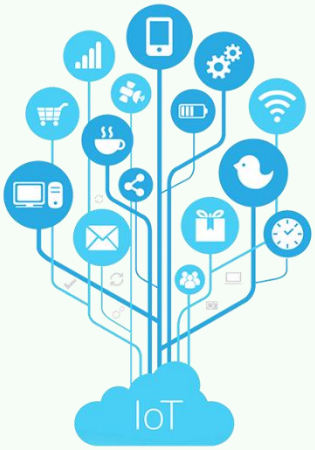
UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

TP n°1 :
Programmation RTOS

MASTER RÉSEAUX ET TÉLÉCOM



VERSION : 10 juin 2021

Florent NOLOT
UNIVERSITÉ DE REIMS CHAMPAGNE ARDENNE

Table des matières

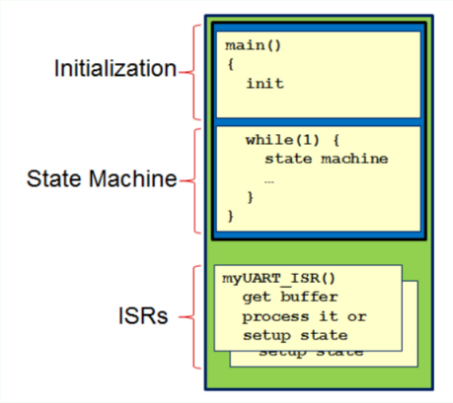
I. Introduction aux RTOS

A. Bare-Metal vs RTOS

1. Principes

Les programmes principaux *main()* d'applications simples sont souvent découpés en 3 parties.

- L'initialisation : Instructions d'initialisation des composants hardware & software
- La boucle principale : Instructions qui doivent être répétées en boucles
- ISR (Interrupt Service Routine) : Instructions exécutées lors des interruptions matérielles



Par exemple voici un code qui reprend la structure Bare-Metal et qui fait clignoter la LED rouge du LaunchPad toute les secondes et fait basculer l'état de la LED verte à chaque interruption (pression du bouton 1).

```c
#include <stdint.h>             /* For uint32_t */
#include <NoRTOS.h>             /* For NoRTOS_start() */
#include <unistd.h>             /* For sleep */
#include <ti/drivers/GPIO.h>    /* For Driver Header files */
#include <ti/drivers/Board.h>   /* For Example/Board Header files */

#include "Board.h"              /* For board configuration */

void interruptServiceRoutine(uint_least8_t index){
    GPIO_toggle(Board_GPIO_LED1);
}
int main(void){
    uint32_t time = 1;

    /* Call driver init functions */
    Board_init();
    GPIO_init();

    /* LEDs Initialization */
    GPIO_setConfig(Board_GPIO_LED0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
    GPIO_setConfig(Board_GPIO_LED1, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
    GPIO_write(Board_GPIO_LED1, Board_GPIO_LED_OFF);

    /* Button Initialization */
    GPIO_setConfig(Board_GPIO_BUTTON0, GPIO_CFG_IN_PU | GPIO_CFG_IN_INT_FALLING);
    GPIO_setCallback(Board_GPIO_BUTTON0, interruptServiceRoutine);
    GPIO_enableInt(Board_GPIO_BUTTON0);

    /* Start NoRTOS */
    NoRTOS_start();

    /* Loop forever Blinking */
    while (1) {
        sleep(time);
        GPIO_toggle(Board_GPIO_LED0);
    }
}
```

ISR

Initialisation

Boucle P.

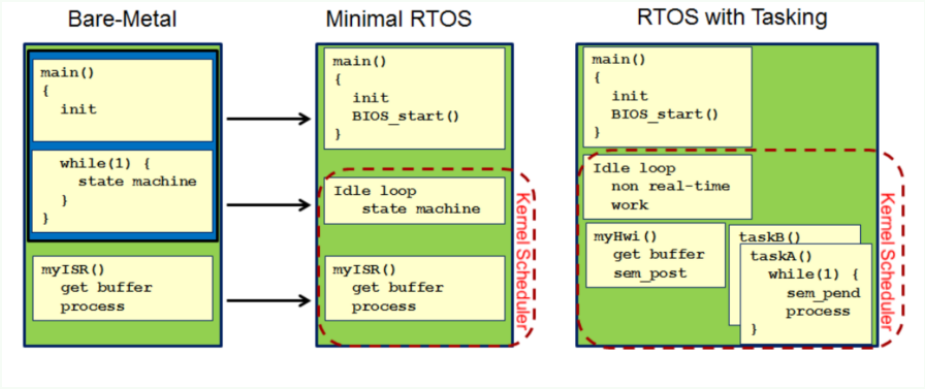
Pour des applications plus complexes et où le contexte requière des vitesse plus rapides, on utilise des systèmes d'exploitation dit « temps réel ». Un Système d'exploitation temps réel ou Real Time Operating System (RTOS) « est un système d'exploitation est un système d'exploitation pour lequel le temps maximum entre un stimulus d'entrée et une réponse de sortie est précisément déterminé ». (Wikipedia.org)

Ainsi l'objectif d'un RTOS est de :

- Réduire le temps de latence (C'est du temps réel après tout)
- D'améliorer la structure en la divisant « pour mieux régner »
- Déterminer le temps d'exécution d'une tâche et poser des Deadlines
- Passer d'une application simple à un autre plus complexe
- Manager différents aspect du système permettant aux développeurs de se concentrer sur leur application.

Les programmes RTOS se structurent de la manière suivante :

- La fonction main pour l'initialisation et la fonction permettant de démarrer le Bios
- L'Idle Loop qui exécute les instructions possédant la plus basse priorité
- L'ISR (Interrupt Service Routine) : Instructions exécutées lors des interruptions matérielles
- Des tâches qui sont exécutées par des Threads.



2. RTOS dans CCS

CCS propose différent exemples de codes, ceux-ci peuvent être séparer en plusieurs parties :

- No RTOS : Applications n'utilisant pas un OS temps réel
- FreeRTOS : Application utilisant un RTOS portable, préemptif et open source
- Portable : Applications multi-threadées utilisant différents drivers de SimpleLink SDK
- Portable Native : Applications multi-threadées utilisant l'API native de TI
- TI-RTOS : Applications utilisant le RTOS de TI

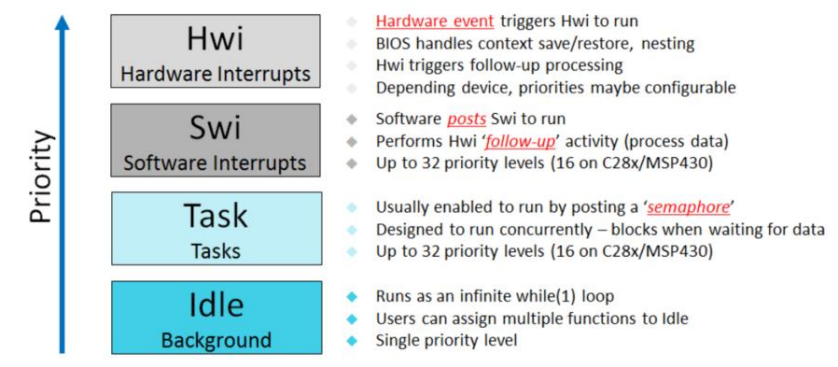
B. TI – RTOS

Texas Instrument a inventé un son propre RTOS permettant au développeur de se contenter sur le développement de leur application sans devoir création des fonctions logicielles de zéro. TI complète son offre de RTOS en incluant des solutions de pile de protocole, communication multi-core, des drivers and management d'alimentation. (Tout ceci est expliqué en détail dans les parties suivantes). TI-RTOS connue aussi par SYS/BIOS est géré par un module appelé Kernel (noyau).

C. Threads et priorité

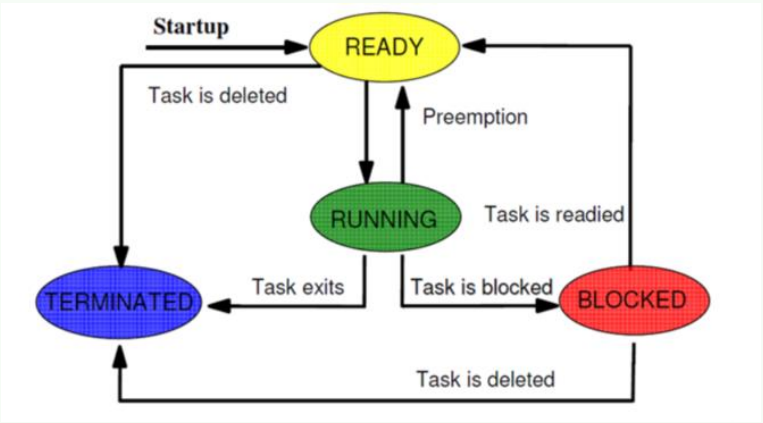
1. Règles

La fonction principale du kernel c'est le L'ordonnanceur (scheduler), celui-ci manage les tâches et les interruptions (exécutées par des Threads) pour que celle possédant la priorité la plus grande s'exécute avant les autres.



- Hwis : Les interruptions matérielles ne peuvent pas s'exécuté indéfiniment. Elles peuvent être préempté par une autre interruption matérielle possédant une priorité plus grande. Toutes les interruptions matérielle possède la même pile système.
- Swis : Les interruptions logicielles sont similaires aux Hwis mais sont déclenchées virtuellement par le programme. Elles possèdent la même pile système que les Hwis threads. En ayant une priorité inférieures aux Hwis les Swis sont souvent utilisées pour exécuter des instructions qui aurait dû être exécutée lors d'une interruption matériel, cela permet de minimiser le temps de latence d'une interruption.
- Tasks : Comme évoquée pour précédemment les tâches sont exécutées par des Threads, chaque Threads possède sa propre pile (ce qui maintient son état). Chaque Thread partage le même espace mémoire que les autres Threads, donc attention à la concurrence. Un Thread peut être bloqué et sont souvent utilisé pour s'exécuter indéfiniment.

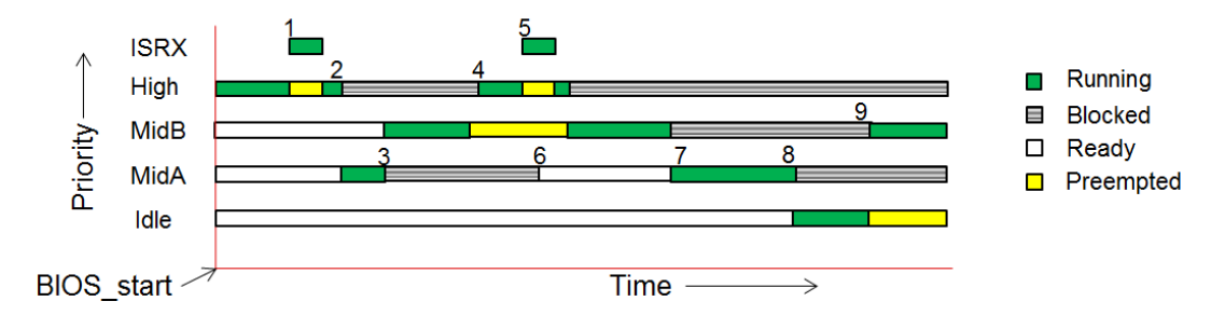
Les tâches et les interruptions possèdent tous une priorité et peuvent ainsi dans différents états.



- Idle : Idle est une tâche spéciale qui possède la priorité la plus basse (0 : priorité la plus basse, 1 : la prochaine la priorité la plus basse…, -1 : La plus haute priorité). L'idle task s'exécute des instruction en fond, comme la vérification de pile système ou la détermination du chargement du CPU…

2. Exemple

- ISRX : Insterrupt Service Routine
- MidA : tâche initialisée en premier dans le main() avec une priorité de 4
- MidB : tâche initialisée en second dans le main() avec une priorité de 4
- High : tâche initialisée en dernier dans le main() avec une priorité de 8



Une fois que l'ordonnanceur du noyau (kernel) s'exécute (dans ce cas grâce à BIOS_start() dans le main), toutes les tâches sont prêtes à démarrer. High démarre en premier car elle possède la priorité la plus grande.

1. Une interruption est déclenchée donc High est préempté puis l'ISR s'exécute.
2. Une fois que l'ISR est terminée, la tâche High continue son exécution jusqu'au moment ou elle passe dans l'état bloqué dû à une pause (sleep).
3. Ainsi la tâche MidA s'exécute car elle a été déclarée en premier dans le main(), puis elle est bloqué car elle attend l'autorisation pour accéder à une ressource (semaphore), donc MidB s'exécute.
4. MidB est préempté par High qui sort de sa pause.
5. Une nouvelle interruption matérielle est déclenchée, donc High est aussi préempté.
6. MidA passe en mode *ready* pendant l'exécution de l'ISR met ne s'exécute pas. Lorsque l'ISR est terminée High qui était préempté s'exécute avant MidB (aussi préempté) car elle possède une priorité supérieure, puis est de nouveau bloqué et MidB s'exécute.
7. MidB est bloqué donc MidA s'exécute.
8. Toutes les tâches sont désormais bloquées, ainsi il n'y a plus de Threads qui s'exécute ou qui sont prêt à être exécutées donc l'Idle s'exécute finalement.
9. L'Idle est préempté car MidB sort de sa pause et s'exécute.

D. Concurrence et communication

Les Threads possèdent leur propre pile d'exécution et peuvent ainsi exécuter des instructions indéfiniment. Toutefois, ils partagent leur mémoire avec les autres Threads, il est donc nécessaire de gérer la concurrence.

1. Sensibilisation à la concurrence entre Threads

Les deux Threads de ce programme sont initialisés avec une priorité de 1, donc l'ordonnanceur devra exécuté en premier le Thread 1 car c'est celui qui déclaré en premier dans le *main().* Le rôle de chaque thread est d'incrémenter une variable globale et d'afficher sa valeur dans la console. Le *sleep()* est présent pour simuler l'exécution de d'autres instructions.

```c
/* ======== main ========  */
int main(void){
    Task_Params taskParams;


    /* Call driver init functions */
    Board_init();

    /* Construct writer/reader Task threads */
    Task_Params_init(&taskParams);
    taskParams.stackSize = THREADSTACKSIZE;

    /* Initialization taks 1 + priority : 1 */
    taskParams.priority = 1;
    if( (task1_id = Task_create((Task_FuncPtr)function_task1, &taskParams, Error_IGNORE)) == NULL ){
        while (1); /* Error creating task */
    }

    /* Initialization taks 2 + priority : 1 */
    taskParams.priority = 1;
    if( (task2_id = Task_create((Task_FuncPtr)function_task2, &taskParams, Error_IGNORE)) == NULL ){
        while (1); /* Error creating task */
    }

    /* Initialize the GPIO since multiple threads are using it */
    GPIO_init();

    /* Start the TI-RTOS scheduler */
    BIOS_start();

    return (0);
}
```

Tâche 1 : Priorité 1

Tâche 2 : Priorité 1

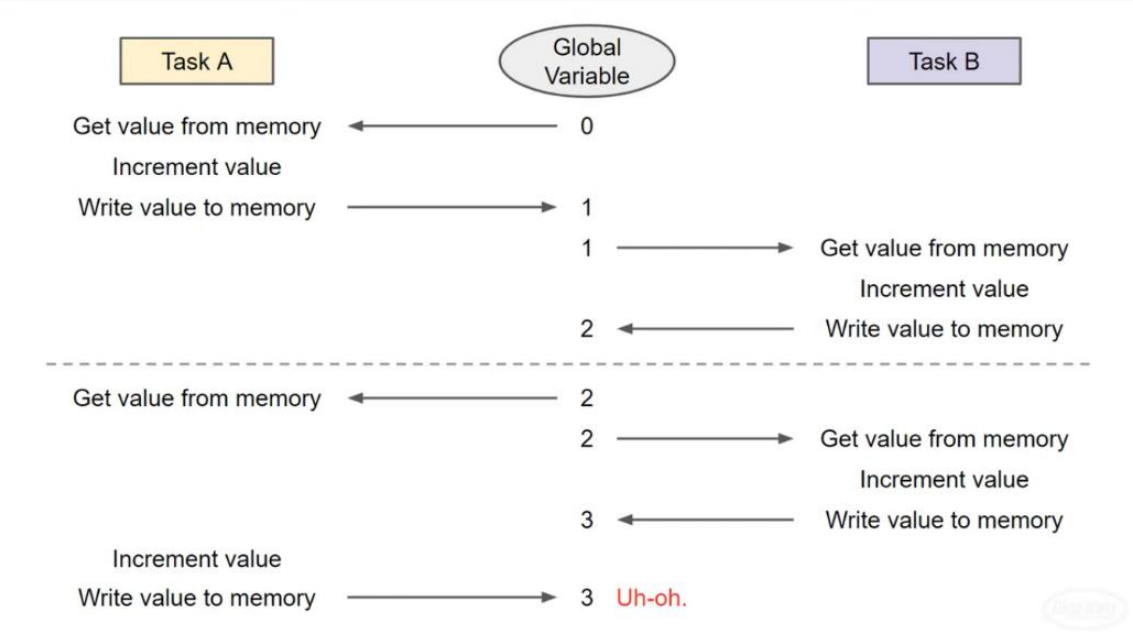
```c
/* RTOS header files */
#include <xdc/std.h>
#include <xdc/runtime/Error.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/drivers/GPIO.h>        /* For Driver header files */
#include <ti/drivers/Board.h>       /* For Example/Board Header files */
#include <unistd.h>                 /* For usleep */
#include <stdio.h>                  /* For printf */

#define THREADSTACKSIZE    1024     /* Stack size in bytes. Large enough in case debug kernel is used. */

Task_Handle task1_id;
Task_Handle task2_id;
int global = 0;

void function_task1(){
    int local;
    while(1){
        local = global;
        local++;
        sleep(1);
        global = local;
        printf("Task1 : global = %d\n",global);
    }
}

void function_task2(){
    int local;
    while(1){
        local = global;
        local++;
        sleep(1);
        global = local;
        printf("Task2 : global = %d\n",global);
    }
}
```

Tâche 1 : global ++ Affichage « global »

Tâche 2 : global ++ Affichage « global »

```
Concurrency_threads:CIO
[Cortex_M3_0] Task1 : global = 1
Task2 : global = 1
Task1 : global = 2
Task2 : global = 2
Task1 : global = 3
Task2 : global = 3
Task1 : global = 4
```

Finalement, on remarque que la variable globale n'est pas incrémentée de 1 à chaque affichage. Comme l'illustre le schéma ci-dessous, lorsqu'une tâche récupère le contenue de la variable globale et l'incrémente de 1, une autre tâche peut exécuter la même instruction plus rapidement et donc l'affichage et le contenue de la variable est incorrect.



2. TI-RTOS : Outils IPC

Les solutions apportées par TI-RTOS sont les « IPC » (Inter-process communication) ou outils de communications inter-processus. Ci-dessous une sélection des outils utilisés en programmation TI-RTOS est présentée.

a) Sémaphores

Les sémaphores permettent à un ou plusieurs Thread(s) d'entrer dans une section critique de code exécutant des instructions permettant de manipuler une ressource. Un sémaphore contient une valeur qui est décrémentée d'une certaine valeur à chaque fois qu'un Thread accède à une ressource et incrémentée d'une certaine valeur à chaque fois qu'un Thread libère la ressource. En utilisant le principe des sémaphores l'information de l'état de la section critique est partagée par l'ensemble des Threads. Il existe les sémaphores binaire avec lesquels un seul Thread à la fois peut avoir accès à la ressource.

b) Mailboxes

Les Mailboxes sont appelé « Queues » par des programmes utilisant Free-RTOS (autre RTOS), elles permettent de transmettre des données entre les Threads.

c) Queues

Les Queues ou files de message permettent de transmettre des données entre les Thread de manière bidirectionnelle (Full duplex).

d) Events

La traditionnel définition d'un Event permet de spécifier qu'une interruption est un Event, tout comme l'incrémentation de la valeur d'un sémaphore, ainsi un Event pourrait être résumé par une action logicielle ou matérielle déclenchant une modification de l'état du système. Si on reprend le principe des sémaphores, on peut déterminer un producteur (l'interruption matérielle qui déclenche l'incrémentation d'un sémaphore) et un consommateur (Le Thread qui attendais l'interruption et a

désormais l'accès à la ressource et qui décrémente la valeur du sémaphore), dans le cas où un Thread (consommateur) est attente de plusieurs producteurs ou une combinaison d'entre eux, il est judicieux d'utiliser les Events.

E. Timers & horloges

Il existe principalement trois type de timer gérés par TI-RTOS

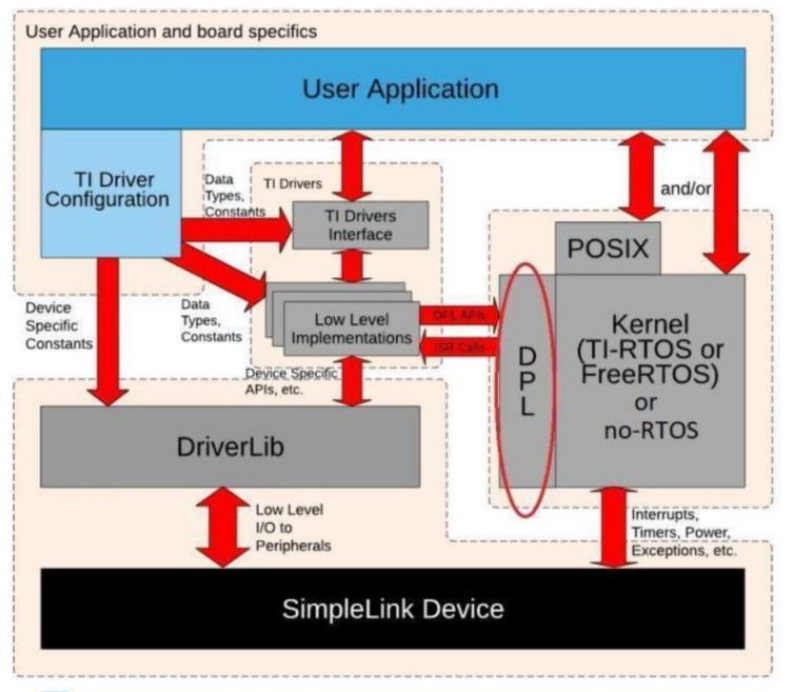
- Timer : Module permettant de gérer un timer matériel. Par défaut TI-RTOS utilisé par TI-RTOS utilise Timer pour les services de gestion du temps des tâches
- Clock : Module dont ses fonctions sont exécutées au bout d'une durée spécifiée par l'utilisateur, cette durée peut être périodique
- Seconds : Module utilisant la RTC du MCU

F. APIs

Une interface de programmation d'application (API) ou interface de programmation applicative, est un ensemble normalisé de classes, de méthodes, de fonctions et de constantes qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels. *(Wikipédia.org)*

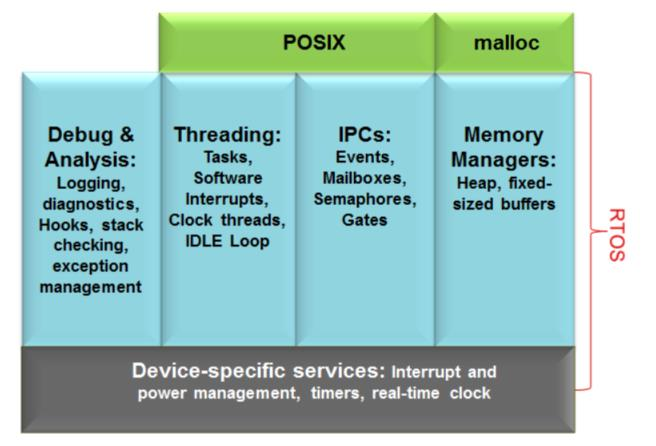
1. API TI-RTOS

Les TI drivers sont développés pour être utilisés par le Driver Porting Layer (DPL). La SimpleLink SDK inclut un DPL pour être implémenté dans FreeRTOS et TI-RTOS.



2. API POSIX

POSIX est une API fournissant un support de développement portable fonctionnant à la fois avec TI-RTOS et avec FreeRTOS. Par exemple, Lors d'une création de Thread en utilisant l'API POSIX (pthread_create()), une TI-RTOS tâche est en finalement créée (Task_create()).



II. Exercices

Rappel : Les liens direct vers les ressources sont dans l'annexe.

A. Exercice 1 – Tâches et priorités

Importer depuis Ressource Explorer : SimpleLink CC13x0 SDK – 4.10… → Examples → Developpement Tools → CC1350 → TI-Drivers → empty → TI-RTOS → CCS Compiler → empty

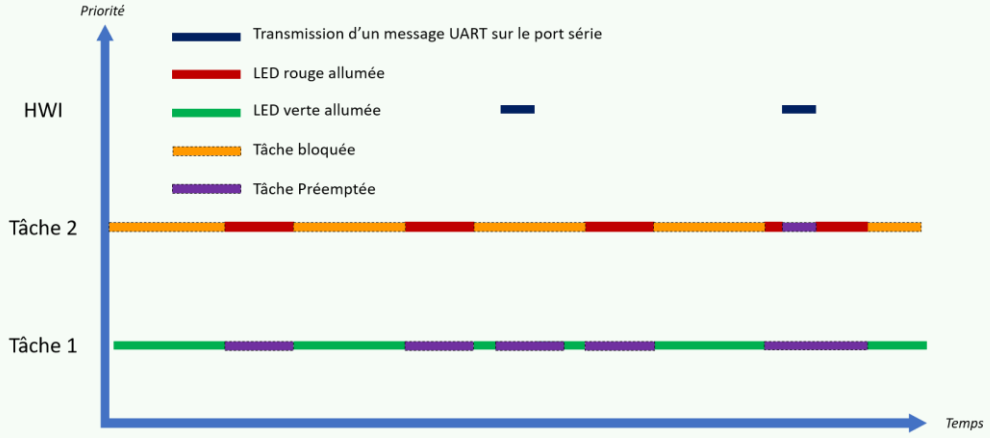
Supprimer « empty.c » et effacer le contenu dans la fonction *main()* du fichier « main_tirtos.c »

1. En utilisant les APIs TI-RTOS et TI Drivers, écrivez un programme possédant 2 tâches et une ISR. La tâche 1 doit avoir une priorité inférieure à la seconde tâche.
    - Quand la tâche 1 est en mode « Running », la LED verte doit être allumée et éteinte dans les autre modes.
    - Quand la tâche 2 est en mode « Running », la LED rouge doit être allumée et éteinte dans les autre modes.
    - L'ISR envoie un message sur le port série

    N'oubliez pas d'ajouter les headers/modules correspondant aux fonctions utilisées et la gestion d'erreur.

    Astuces :

    - Pour ajouter un délai sans positionner la tâche en « bloqué », implémentez un compteur, comptant de 0 à 100 000 (minimum)
    - Utilisez un baud rate de 115200
    - Pour la fonction *task_sleep(), multipliez par 100 000 la durée choisie et divisez le tout par « Clock_tickPeriod » (Include : <ti/sysbios/Clock.h>*
    - A la création d'une tâche il est possible de créer un objet manageur « statique » (Task_construct()) ou « dynamique (Task_create()).



2. Développez la même application avec les APIs POSIX

B. Exercice 2 – Sémaphores

En implémentant un sémaphore binaire modifiez un des programmes de l'exercice précédent (TI-RTOS ou TI-POSIX) afin de débloquer un thread (lors d'une pression du bouton 0) qui fait clignoter la LED rouge en continue.

C. Exercice 3 – Mailbox

Modifiez le programme de l'exercice précédent, afin d'obtenir une ISR et deux tâches.

- L'ISR doit incrémenter un sémaphore de 1 à chaque pression sur le bouton 0
- La tâche 1 décrémente le même sémaphore, lis l'état de la LED rouge et transmet la valeur à l'aide d'une Mailbox à la seconde tâche après un délai. (task_sleep(), ou boucle).
- La tâche 2 lis le message de reçu dans la Mailbox et positionne l'état de la LED verte au même état que celui de la LED rouge

D. Exercice 4 – Horloge

Modifiez le programme de l'exercice précédent et remplacez l'interruption (ISR) par une horloge.

III. Annexes

A. Obtenir codes exemples

1. Bare-Métal & Clignotement LED

Importer depuis Ressource Explorer : SimpleLink CC13x0 SDK – 4.10… → Examples → Developpement Tools → CC1350 → TI-Drivers → gpiointerrupt → No-RTOS → CCS Compiler → gpiointerrupt

Supprimer le fichier « gpiointerrupt.c » et copier/coller le code fournis dans le fichier « Bare_Metal_Example.c » dans « main_notos.c ».

2. Concurrence et Threads

Importer depuis Ressource Explorer : SimpleLink CC13x0 SDK – 4.10… → Examples → Developpement Tools → CC1350 → Demos → Portable Native → TI-RTOS → CCS Compiler → portableNative

Copier / Coller le code fournis dans le fichier « Concurrency_threads.c » dans le fichier « main_tirtos.c ».

B. Ajouter des headers supplémentaires dans un projet

Cliquer droit sur le nom du projet → Properties → Build → Arm Compiler → Include Options

Cliquer sur le bouton « Add » → Browse et sélectionner le dossier dans lequel les fichiers d'en-tête sont localisés.
Cliquer sur « Apply & Close »

C. Vidéo d'introduction TI-RTOS (Bonus)

https://www.youtube.com/watch?v=_4Cq2GXTK9c&list=PLM3nAhWK5eVKzVteAGFtugF6XP5Y3aOXx&index=1

D. Ressources

1. APIs TI Drivers Runtime

https://dev.ti.com/tirex/explore/node?node=AOng5xFsavzvQ16.KytQHg__eCfARaV__LATEST

2. APIs TI-RTOS Kernel Runtime

https://dev.ti.com/tirex/explore/node?node=ALK7.RCEsk55Rp9wysRkGw__eCfARaV__LATEST

3. APIs TI-POSIX

http://software-dl.ti.com/simplelink/esd/simplelink_msp432e4_sdk/2.40.00.11/docs/tiposix/Users_Guide.html#overview

https://hpc-tutorials.llnl.gov/posix/