

A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

08/01/2024

Rapport

Programmation en temps réel

Table des matières

Introduction	3
Objectif.....	3
Préparation de l'environnement de travail	3
LINUX, IPC ET OUTILS DE SYNCHRONISATION	4
Objectif.....	4
Déroulement	4
Gestion par threads et processus	4
Partage de données et deadlock.....	5
Exclusion mutuelle	6
OS Temps Réel et programmation de tâches	9
Objectif.....	9
Introduction a « Zephyr »	9
Récupération des projets exemples.....	10
Comprendre "NoGUI"	11
Modifier "NoGUI"	12
Jouer avec le bouton de "NoGUI"	15
Explication du programme liée au bouton	15
Tests des différents scénarios.....	19
Algorithmes de partage de ressources	21
Objectif.....	21
L'algorithme Producteur/Consommateur avec des LEDs	21
Comprendre le projet "WithGUI"	25
L'algorithme Lecteur/Rédacteurs avec un écran.....	29
Ordonnancement Temps Réel	34
Définitions et jeu de tâches	34
Comprendre "PeriodicTasks" être produire l'exemple.....	35
Et si τ_2 est plus contrainte ?.....	40
Et si en plus, τ_3 est plus longue ?.....	42
Problèmes	43
Enoncé.....	43
Mise en place de l'expérience.....	43
Code	44

Résultat expérimental.....	47
Questionnements	48

Introduction

Objectif

L'objectif de ces travaux pratiques est la mise en œuvre d'applications simples basées sur un système d'exploitation temps réel. Les différents exercices auront pour objectif d'étudier l'importance de la gestion des priorités des différents threads d'un processus ainsi que l'intérêt de travailler "sous des interruptions". En particulier, nous découvrirons l'intérêt et les principes de programmation par tâches.¹

Préparation de l'environnement de travail

Pour ces travaux pratique, nous allons prendre en main certains concepts en utilisant un environnement Linux, puis utiliser une carte de développement (STMicroelectronics 32F429I DISCOVERY).

Pour l'environnement de travail, nous utilisons une image de machine virtuelle sur Oracle VirtualBox trouvable sur Moodle d'ESIEA. Les accès sont :

- Utilisateur : "user"
- Mot de passe : "user"

Nous utiliserons l'IDE *VSCode* et *PlatformIO*

¹ [Sources sur sujet du Travaux Pratique](#)

LINUX, IPC ET OUTILS DE SYNCHRONISATION

Objectif

Dans cette section, notre objectif principal est de mettre à l'épreuve la bibliothèque en ce qui concerne les threads. Pour ce faire, nous nous pencherons sur le code de trois fichiers spécifiques : « DataAndThread.c », « DataShare.c » et « Makefile ». Notre démarche consistera à rectifier les éventuelles erreurs dans ces codes et à identifier les problèmes liés à la protection ou à la non-protection de la manipulation des variables.

Déroulement

Gestion par threads et processus

Lors de l'exécution du programme par le fichier « Makefile », nous obtenons des erreurs multiples.

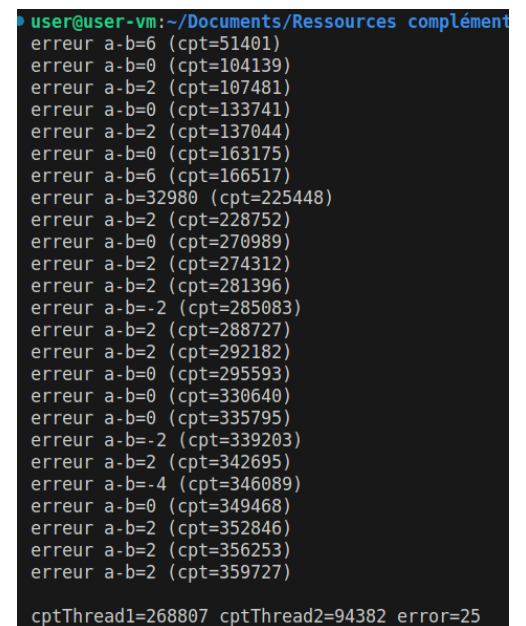
Après recherche, l'origine du problème provient sur l'utilisation de la mémoire lors de l'exécution du programme. Lors de l'exécution, les threads modifient la valeur des variables provoquant des problèmes lors de la manipulation des variables.

Pour résoudre ce problème, il faut protéger les variables lors de la manipulation, autant en écriture et lecture. Ainsi, nous utilisons la méthode Mutex pour protéger les variables. On bloque au début des calculs jusqu'à la lecture et comparaison pour éviter toute erreur. On laisse écrire sur « cpt » pour compter le nombre de threads qui utilise la fonction.

En voici le code :

```
void calc(void) {
    static long cpt=0; /*nombre total d'appels de la fonction */
    cpt++;

    pthread_mutex_lock(&ghMutex2);
    c=a+2;
    b=c;
    a=c;
    if (a!=b) {
        printf("erreur a-b=%d (cpt=%ld)\n", a-b,cpt);
        error++;
    }
    pthread_mutex_unlock(&ghMutex2); }
```



```
user@user-vm:~/Documents/Ressources complément
erreur a-b=6 (cpt=51401)
erreur a-b=0 (cpt=104139)
erreur a-b=2 (cpt=107481)
erreur a-b=0 (cpt=133741)
erreur a-b=2 (cpt=137044)
erreur a-b=0 (cpt=163175)
erreur a-b=6 (cpt=166517)
erreur a-b=32980 (cpt=225448)
erreur a-b=2 (cpt=228752)
erreur a-b=0 (cpt=270989)
erreur a-b=2 (cpt=274312)
erreur a-b=2 (cpt=281396)
erreur a-b=-2 (cpt=285083)
erreur a-b=2 (cpt=288727)
erreur a-b=2 (cpt=292182)
erreur a-b=0 (cpt=295593)
erreur a-b=0 (cpt=330640)
erreur a-b=0 (cpt=335795)
erreur a-b=-2 (cpt=339203)
erreur a-b=2 (cpt=342695)
erreur a-b=-4 (cpt=346089)
erreur a-b=0 (cpt=349468)
erreur a-b=2 (cpt=352846)
erreur a-b=2 (cpt=356253)
erreur a-b=2 (cpt=359727)

cptThread1=268807 cptThread2=94382 error=25
```

Figure 1: Retour erreur

Partage de données et deadlock

Concernant le programme « DataShare », il existe un problème variable qui apparaît dans certains cas. Ce problème est celui d'un Deadlock. Dans certains cas, on a une situation où deux programmes se bloquent mutuellement, car chacun attend une réponse de l'autre ou la libération d'un verrou détenu par l'autre.

On voit dans ce code que les manipulations sur les variables sont finies, il passe dans un autre programme avant qu'il ne rende la clé Mutex pour d'autres threads puis traiter leurs tâches. Or, dans les deux programmes, il se passe la même démarche ce qui rend un deadlock.

```
void calc2(int p_numThread) {
    pthread_mutex_lock(&ghMutex2);
    printf("2");
    myData2.a++;
    myData2.b--;
    if (p_numThread==2)
    {
        calc1(2);
    }
    pthread_mutex_unlock(&ghMutex2);
}
void calc1(int p_numThread) {
    pthread_mutex_lock(&ghMutex1);
    printf("1");
    myData1.a++;
    myData1.b--;
    if (p_numThread==1) {
        calc2(1);
    }
    pthread_mutex_unlock(&ghMutex1);
}
```

Exclusion mutuelle

Voici un code qui devra contenir une zone mémoire de type SNOMBRE partagée entre 3 threads :

- Premier thread qui viendra écrire toutes les 10 secondes une valeur, par exemple (2, 2, 2),
- Second thread qui viendra écrire toutes les secondes une autre valeur, par exemple (10, 10, 10),
- Dernier thread qui viendra uniquement lire les valeurs et les affichera sur le terminal uniquement si au moins l'une des trois coordonnées a changé.

On mettra en évidence le fait que le thread lecteur peut afficher une donnée incohérente (par exemple, un mélange de 2 et de 10). On ajoute artificiellement des « sleep » dans le programme afin de simuler un code mal conçu.

Puis on modifiera à nouveau notre code pour garantir que, quoi qu'il arrive, la mémoire partagée ne pourra pas être écrite ou lue partiellement.²

Ainsi, on obtient le code suivant :

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// On crée une structure snombre
typedef struct snombre {
    int x;
    int y;
    int z;
} SNOMBRE;

SNOMBRE coor = {2, 2, 2};
int nb_erreur = 0;
// pthread_mutex_t mutex1; // mutex de coordonnées

void *writerThread1(void *p) {
    while (1) {
        sleep(10);
        // pthread_mutex_lock(&mutex1);
        coor.x = 2;
        coor.y = 2;
        coor.z = 2;
        // pthread_mutex_unlock(&mutex1);
    }
    return NULL;
}
```

² [Source sujet de Travaux Pratique](#)

```
void *writerThread2(void *p) {
    while (1) {
        sleep(1);
        // pthread_mutex_lock(&mutex1);
        coor.x = 10;
        // sleep(3); ajout error
        coor.y = 10;
        // sleep(3); ajout error
        coor.z = 10;
        // pthread_mutex_unlock(&mutex1);
    }
    return NULL;
}

void *readerThread(void *p) {
    while (1) {
        //pthread_mutex_lock(&mutex1);
        int X = coor.x;
        int Y = coor.y;
        int Z = coor.z;
        //pthread_mutex_unlock(&mutex1);

        if((X != 2) || (Y != 2) || (Z != 2))
        {
            if((X != 10) || (Y != 10) || (Z != 10))
            {
                printf("x: %d, y: %d, z: %d \n",X,Y,Z);
                nb_erreur++;
            }
        }
        sleep(1);
    }
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t writer1, writer2, reader;

    pthread_create(&writer1, NULL, writerThread1, NULL);
    pthread_create(&writer2, NULL, writerThread2, NULL);
    pthread_create(&reader, NULL, readerThread, NULL);

    sleep(20);

    // pthread_mutex_destroy(&mutex1);
}
```



```
printf("nb errors : %d \n",nb_erreur);
printf("END \n");
return 0;
}
```

Pour utiliser ce code, cela se fait en 3 étapes, on commence par le code avec toutes les modifications en commentaires afin de voir comment ce dernier répond :

```
● user@user-vm:~/Documents/Ressources complémentaires-20240108/TP1$ ./DataPointShare
nb errors : 0
END
```

On peut voir qu'il n'y a aucune erreur, car notre système va assez vite pour les éviter (mais elles peuvent arriver, mais ici, les threads font leur modification assez rapidement.).

Ensuite, on comment pour rajouté des tempo (on décommente les deux sleep(3) dans le thread writeThread2()), afin d'ajouter manuellement une erreurs, car un même élément est accédé et modifié par plusieurs threads en même temps.

```
● user@user-vm:~/Documents/Ressources complémentaires-20240108/TP1$ ./DataPointShare
x: 10, y: 2, z: 2
x: 10, y: 2, z: 2
x: 10, y: 2, z: 2
x: 10, y: 10, z: 2
x: 10, y: 10, z: 2
x: 10, y: 10, z: 2
x: 2, y: 10, z: 2
x: 2, y: 10, z: 2
x: 2, y: 10, z: 2
x: 2, y: 10, z: 10
nb errors : 10
END
```

On peut voir ici qu'il y a eu 10 erreurs où les deux threads ont fait des modifications aux valeurs de la variable « coor ».

On ajoute un mutex afin de protéger en écriture notre variable « coor » et on peut clairement voir que plus aucun souci n'est présent peu importe le temps que met chaque thread a modifié la variable.

```
● user@user-vm:~/Documents/Ressources complémentaires-20240108/TP1$ ./DataPointShare
nb errors : 0
END
● user@user-vm:~/Documents/Ressources complémentaires-20240108/TP1$
```

L'ajout du mutex a pu permettre de protéger notre variable « coor » en écriture et en lecture. Cela implique que peu importe le temps mis à effectuer l'opération sur la variable protégé cette dernière ne peut la corrompre.

OS Temps Réel et programmation de tâches

Objectif

Dans cette section, nous explorons en détail le Système d'exploitation "Zephyr" afin d'acquérir une compréhension approfondie de son fonctionnement. Nous nous plongerons dans les principaux composants qui le constituent, examinant ses caractéristiques fondamentales et ses mécanismes essentiels. En comprenant mieux la structure et les fonctionnalités de Zephyr, nous visons à fournir un aperçu clair et informatif qui permettra aux utilisateurs et aux développeurs de tirer pleinement parti de ce système d'exploitation.

Introduction a « Zephyr »

Le système d'exploitation Zephyr est basé sur un noyau à faible encombrement conçu pour être utilisé sur des systèmes embarqués et à ressources limitées : des simples capteurs environnementaux embarqués et des LED portable aux contrôleurs embarqués sophistiqués, aux montres intelligentes et aux applications sans fil de l'Internet des objets.³

“Multi-threading Services for cooperative, priority-based, non-preemptive, and preemptive threads with optional round robin time-slicing. Includes POSIX pthreads compatible API support.”

Le système d'exploitation Zephyr est également basé sur l'algorithme Round-robin (ou tourniquet en français). Le Round-robin est un algorithme d'ordonnancement courant dans les systèmes d'exploitation adapté aux systèmes travaillant en temps partagés.

Voici comment fonctionne l'algorithme Round Robin dans le contexte de Zephyr :

1. Chaque tâche dans le système a une priorité assignée.
2. Les tâches avec la même priorité sont placées dans une file d'attente circulaire.
3. À chaque cycle d'ordonnancement, la tâche en tête de la file d'attente est sélectionnée pour l'exécution.
4. Cette tâche est exécutée pendant un quantum de temps prédéfini ou jusqu'à ce qu'elle se bloque (par exemple, en attente d'une ressource).
5. Une fois que le quantum de temps est écoulé ou que la tâche se bloque, elle est replacée à la fin de la file d'attente.
6. La tâche suivante dans la file d'attente devient la nouvelle tâche en cours d'exécution.

La performance de round-robin dépend fortement du choix du quantum de base.⁴

³ [Documentation Zephyr](#)

⁴ [Wikipédia](#), [cours](#) et ChatGPT

“A thread’s priority is an integer value, and can be either negative or non-negative. Numerically lower priorities takes precedence over numerically higher values.”

Ainsi, la tâche avec la valeur de priorité la plus faible est à la priorité la plus haute. Les intervalles de valeurs possibles attribuables par le gestionnaire de tâche sont codés en un entier en 32 bits signé donc [-65535 ; 65536].

“A cooperative thread has a negative priority value. Once it becomes the current thread, a cooperative thread remains the current thread until it performs an action that makes it unready.”

“A preemptible thread has a non-negative priority value. Once it becomes the current thread, a preemptible thread may be supplanted at any time if a cooperative thread, or a preemptible thread of higher or equal priority, becomes ready.”⁵

La principale différence entre une tâche préemptée et une tâche coopérative réside dans la manière dont le contrôle est transféré entre les différentes tâches d'un système. Dans une tâche préemptée, le système d'exploitation peut interrompre une tâche en cours d'exécution pour donner la main à une autre tâche de priorité plus élevée. Cela se produit de manière automatique, sans la nécessité de la coopération explicite de la tâche en cours.

En revanche, une tâche coopérative nécessite une coopération explicite de la part de la tâche en cours d'exécution pour transférer le contrôle à une autre tâche. La tâche en cours doit volontairement libérer le processeur ou indiquer qu'elle est prête à céder la main. Les tâches coopératives sont généralement mises en œuvre de manière plus explicite par le programmeur, tandis que les tâches préemptées permettent au système d'exploitation de gérer automatiquement les priorités des tâches.

Récupération des projets exemples

Dans un terminal, on récupérer les projets exemple fourni par M. Pierre Courbin. En voici la commande et le lien : `git clone https://github.com/pcourbin-teaching/Zephyr_STM32` .

Voici les projets dans ce git :

- NoGUI
- PeriodicTask
- Sample
- withGUI

⁵ [Documentation](#)

Pour compiler et téléverser le code via l'environnement Zephyr dans la carte [STM32F429](#), nous utilisons l'extension PlatformIO via VSCode.

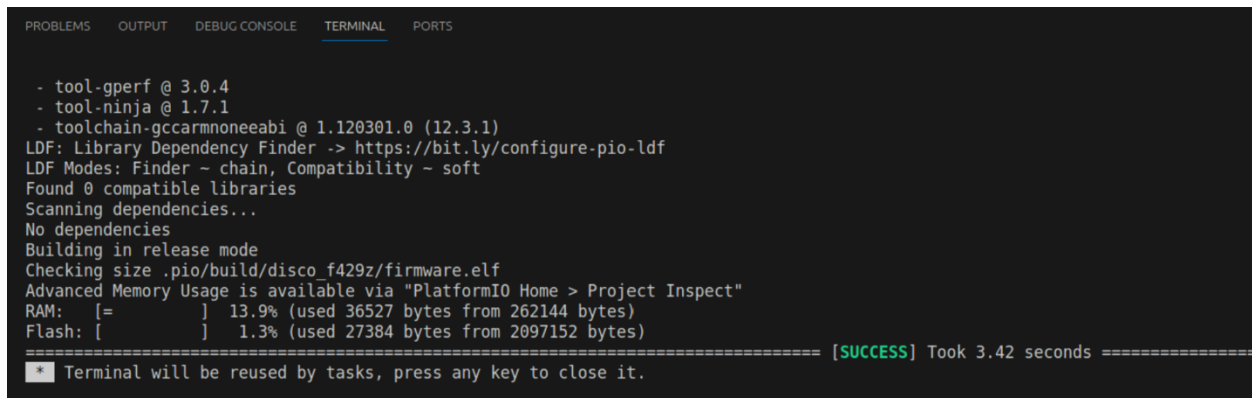
PlatformIO est un outil professionnel multi-plateforme, multi-architecture et multi-cadres destiné aux ingénieurs en systèmes embarqués et aux développeurs de logiciels qui écrivent des applications pour les produits embarqués. PlatformIO est un outil indispensable pour les ingénieurs en systèmes embarqués qui développent des solutions sur plus d'une plate-forme spécifique.

De plus, on nous fournit un CJMCU-680 utilisant un [BME680](#).

Comprendre "NoGUI"

On connecte la carte et lance le mode Debug sur VSCode. On cherche à analyser l'algorithme de gestion des deux leds par les deux threads (on ne parle pas ici du bouton.).

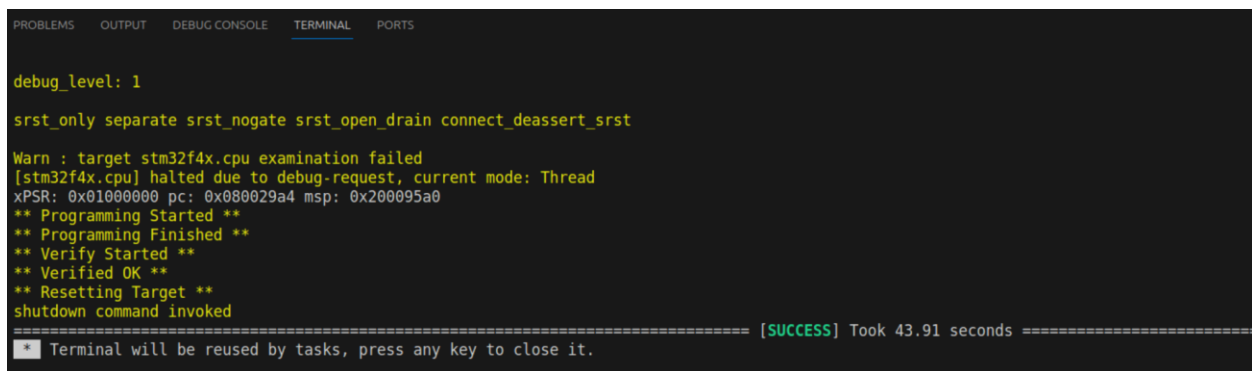
Après importation du projet NOGUI sur VSCODE, on commence par compiler ce dernier afin de vérifier qu'il est bien téléversé sur notre carte.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

- tool-gperf @ 3.0.4
- tool-ninja @ 1.7.1
- toolchain-gccarmnoneabi @ 1.120301.0 (12.3.1)
LDF: Library Dependency Finder -> https://bit.ly/configure-pio-ldf
LDF Modes: Finder ~ chain, Compatibility ~ soft
Found 0 compatible libraries
Scanning dependencies...
No dependencies
Building in release mode
Checking size .pio/build/disco_f429z/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:  [= ] 13.9% (used 36527 bytes from 262144 bytes)
Flash: [= ] 1.3% (used 27384 bytes from 2097152 bytes)
===== [SUCCESS] Took 3.42 seconds =====
* Terminal will be reused by tasks, press any key to close it.
```

Figure 2: Build NOGUI



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

debug_level: 1

srst_only separate srst_nogate srst_open_drain connect_deassert_srst

Warn : target stm32f4x.cpu examination failed
[stm32f4x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080029a4 msp: 0x200095a0
** Programming Started **
** Programming Finished **
** Verify Started **
** Verified OK **
** Resetting Target **
shutdown command invoked
===== [SUCCESS] Took 43.91 seconds =====
* Terminal will be reused by tasks, press any key to close it.
```

Figure 3: Upload NOGUI

On remarque que les temps des clignotements sont périodiques. En regardant le programme, on remarque qu'il y a 2 façons de créer une temporisation d'un état. Le premier est de créer une boucle avec N itérations dans lesquelles on attribue un état identique. Le second est d'utiliser la fonction `k_msleep()` qui met en attente le programme pendant un certain temps en milliseconde.

Le problème dans la première solution, c'est le temps d'attente est variable en fonction du temps de calcul du processeur et de la gestion des priorités des threads, car on est dépendant de l'incréméntation de la variable « i ».

```
static void led1_task(void *p1, void *p2, void *p3)
{
    int i, nbIter = 200000;
    while (1)
    {
        for (i = 0; i < nbIter; i++)
        {
            update_leds(0, 1);
        }
        update_leds(0, 0);
        k_msleep(2000);
    }
}
```

Modifier "NoGUI"

Pourquoi ne pas utiliser la fonction `delay()` ?

La fonction « `k_msleep()` » prend en paramètre un entier qui représente un délai en milliseconde. En comparaison à la fonction « `delay()` », on retrouve le même paramètre (un integer qui nous donne un nombre de millisecondes).

La différence entre ces deux tempos, la fonction « `delay()` » stop l'entièreté du programme alors que « `k_msleep()` » mets uniquement en pause un thread ce qui signifie que le programme continue de fonctionner à côté.

En désactivant l'usage du bouton, on va écrire un programme qui crée trois tâches :

- De priorités différentes,
- De durées différentes (par exemple, une de 1 seconde, une autre de 2 secondes, etc.)
- Qui se partagent les leds tour à tour pour les allumer de manière différente. (Par exemple, une tâche
- Allume uniquement la LED verte, l'autre uniquement la LED rouge et la dernière les deux leds.).

On s'arrangera pour que l'utilisation des leds par une tâche ne soit pas perturbée par une autre.

```

/**
*****
**
* @file    main.c
* @author  P. COURBIN
* @version V2.0
* @date    08-12-2023
* @brief   NoGUI version
*****
**
*/

#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>

#include <zephyr/logging/log.h>
LOG_MODULE_REGISTER(app);

#define STACKSIZE (4096)
static K_THREAD_STACK_DEFINE(led0_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(led1_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(led2_stack, STACKSIZE);

//static K_THREAD_STACK_DEFINE(sw0_stack, STACKSIZE);

/* The devicetree node identifier for the "led0" alias. */
#define LED0_NODE DT_ALIAS(led0)
#define LED1_NODE DT_ALIAS(led1)
#define SW0_NODE DT_ALIAS(sw0)
static const struct gpio_dt_spec led0 = GPIO_DT_SPEC_GET(LED0_NODE, gpios);
static const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(LED1_NODE, gpios);
//static const struct gpio_dt_spec sw0 = GPIO_DT_SPEC_GET_OR(SW0_NODE, gpios,
{0});
//static struct gpio_callback sw0_cb_data;

K_MUTEX_DEFINE(mutexLEDs);
//sint switchPushed = 0;

void update_leds(uint8_t led0_val, uint8_t led1_val)
{
    if (k_mutex_lock(&mutexLEDs, K_FOREVER) == 0)
    {
        gpio_pin_set_dt(&led0, led0_val);
        gpio_pin_set_dt(&led1, led1_val);
        k_mutex_unlock(&mutexLEDs);
    }
}

static void led0_task(void *p1, void *p2, void *p3)
{

```

```

    int i, nbIter = 100000;
    while (1)
    {
        for (i = 0; i < nbIter; i++)
        {
            update_leds(1, 0);
        }
        update_leds(0, 0);
        k_msleep(1000);
    }
}

static void led1_task(void *p1, void *p2, void *p3)
{
    int i, nbIter = 200000;
    while (1)
    {
        for (i = 0; i < nbIter; i++)
        {
            update_leds(0, 1);
        }
        update_leds(0, 0);
        k_msleep(2000);
    }
}

static void led2_task(void *p1, void *p2, void *p3)
{
    int i, nbIter = 500000;
    while (1)
    {
        for (i = 0; i < nbIter; i++)
        {
            update_leds(1, 1);
        }
        update_leds(0, 0);
        k_msleep(3000);
    }
}

uint8_t init_leds()
{
    uint8_t returned = 0;
    if (!device_is_ready(led0.port) || !device_is_ready(led1.port))
    {
        LOG_ERR("Error: LEDs devices are not ready (%s / %s)", led0.port->name, led1.port->name);
        returned = -1;
    }

    if (gpio_pin_configure_dt(&led0, GPIO_OUTPUT_ACTIVE) < 0 ||
        gpio_pin_configure_dt(&led1, GPIO_OUTPUT_ACTIVE) < 0)

```

```

    {
        LOG_ERR("Error: LEDs config failed (%s / %s).", led0.port->name,
led1.port->name);
        returned = -2;
    }
    return returned;
}

void main(void)
{
    struct k_thread led0_t, led1_t, led2_t; // ,sw0_t;
    if (init_leds() < 0) // || init_switches() < 0)
    {
        LOG_ERR("Error: %s", "LED or Switch init failed");
        return;
    }

    k_thread_create(&led0_t, led0_stack, K_THREAD_STACK_SIZEOF(led0_stack),
led0_task, NULL, NULL, NULL,
1, 0, K_NO_WAIT);
    k_thread_create(&led1_t, led1_stack, K_THREAD_STACK_SIZEOF(led1_stack),
led1_task, NULL, NULL, NULL,
2, 0, K_NO_WAIT);

    k_thread_create(&led2_t, led2_stack, K_THREAD_STACK_SIZEOF(led2_stack),
led2_task, NULL, NULL, NULL,
3, 0, K_NO_WAIT);
}

```

Ici, nous avons commencé par retirer le code pour la gestion du bouton, car nous n'en avons pas l'utilité ici. Ensuite, on crée une nouvelle tâche en se basant sur les deux restants permettant de gérer chacune des LEDS.

Ce code nous permet de clairement voir l'effet de préemption, car nos 3 tâches ont des priorités différents allant de 1 pour la tâche led0_t à 3 pour celle que l'on vient de créer.

Cet effet de préemption est donc visible, car on peut voir que la led0 allumer seul ne se fait jamais voler la main tant qu'elle n'a pas fini son temps à l'état haut (allumer) alors que l'allumage des deux leds en même temps (donc notre dernière tâche a la plus basse priorité.) se fait souvent interrompre et revient plus tard pour finir.

Jouer avec le bouton de "NoGUI"

Explication du programme liée au bouton

Dans cette partie, nous allons reprendre le code avant modification du projet « NoGUI », car nous voulons reprendre le code où nous avons la gestion du bouton.

Voici le code pressed callback :


```

void sw0_pressed_callback(const struct device *dev, struct gpio_callback *cb,
uint32_t pins)
{
    LOG_INF("Switch pressed at %d" PRIu32 "\n", k_cycle_get_32());
    if ((pins & (1 << sw0.pin)) != 0)
    {
        switchPushed = 1;
    }
}

```

Cette fonction permet d'ignorer un défaut mécanique du bouton. Le rebondissement des boutons est un phénomène physique qui se produit lorsque vous appuyez ou relâchez un bouton.

En raison de la nature mécanique des interrupteurs, les contacts peuvent rebondir plusieurs fois avant de se stabiliser. Cela peut entraîner des problèmes dans les systèmes qui réagissent aux changements d'état des boutons, car plusieurs transitions peuvent être détectées pour une seule action de l'utilisateur.

Voici le code d'initialisation du switch :

```

uint8_t init_switches()
{
    uint8_t returned = 0;

    if (!device_is_ready(sw0.port))
    {
        LOG_ERR("Error: Switch device %s is not ready.", sw0.port->name);
        returned = -1;
    }

    if (!returned && gpio_pin_configure_dt(&sw0, GPIO_INPUT) != 0)
    {
        LOG_ERR("Error: failed to configure switch %s pin %d.", sw0.port->name, sw0.pin);
        returned = -2;
    }

    if (!returned && gpio_pin_interrupt_configure_dt(&sw0, GPIO_INT_EDGE_TO_ACTIVE) != 0)
    {
        LOG_ERR("Error: failed to configure interrupt on %s pin %d.", sw0.port->name, sw0.pin);
        returned = -3;
    }
}

```

```

    if (!returned)
    {
        gpio_init_callback(&sw0_cb_data,                sw0_pressed_callback,
        BIT(sw0.pin));
        gpio_add_callback(sw0.port, &sw0_cb_data);
    }

    LOG_INF("Set up switch at %s pin %d", sw0.port->name, sw0.pin);
    return returned;
}

```

Cette partie du code permet d'initialiser le bouton sur la carte STM32. Aussi, on retrouve les retours d'erreur, utile lors d'erreur d'association.

Ici, le code de la tâche du bouton :

```

static void sw0_task(void *p1, void *p2, void *p3)
{
    int i, nbIter = 500000;
    while (1)
    {
        if (switchPushed == 1)
        {
            switchPushed = 0;
            for (i = 0; i < nbIter; i++)
            {
                update_leds(1, 1);
            }
            update_leds(0, 0);
        }
        k_yield();
    }
}

```

La tâche est d'allumer les deux leds lors de l'appui du bouton.

Maintenant, que nous avons pu voir le fonctionnement en détails de ce code, nous allons utiliser un oscilloscope numérique (DIGILENT DISCOVERY 2) afin d'observer les Fonctionnement et le délai d'activation des LEDs. La première chose à faire est donc d'aller voir le pin mapping de notre carte (STM32F429) et d'en déduire les pins où l'on va pouvoir brancher notre oscilloscope afin d'observer les signaux des LEDS (LED1 et LED2) et de notre bouton.

Voici ci-dessous notre table de mapping :

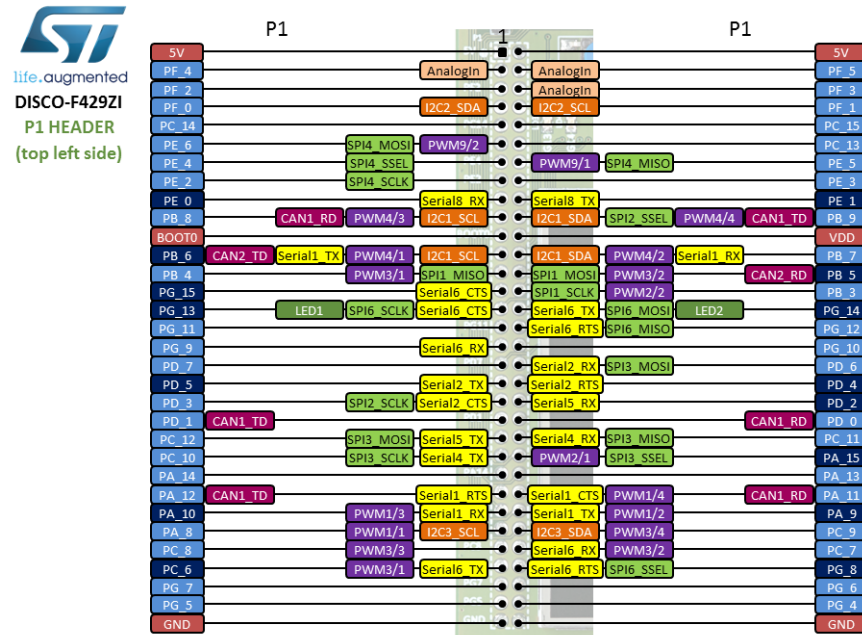


Figure 4: Table des pins

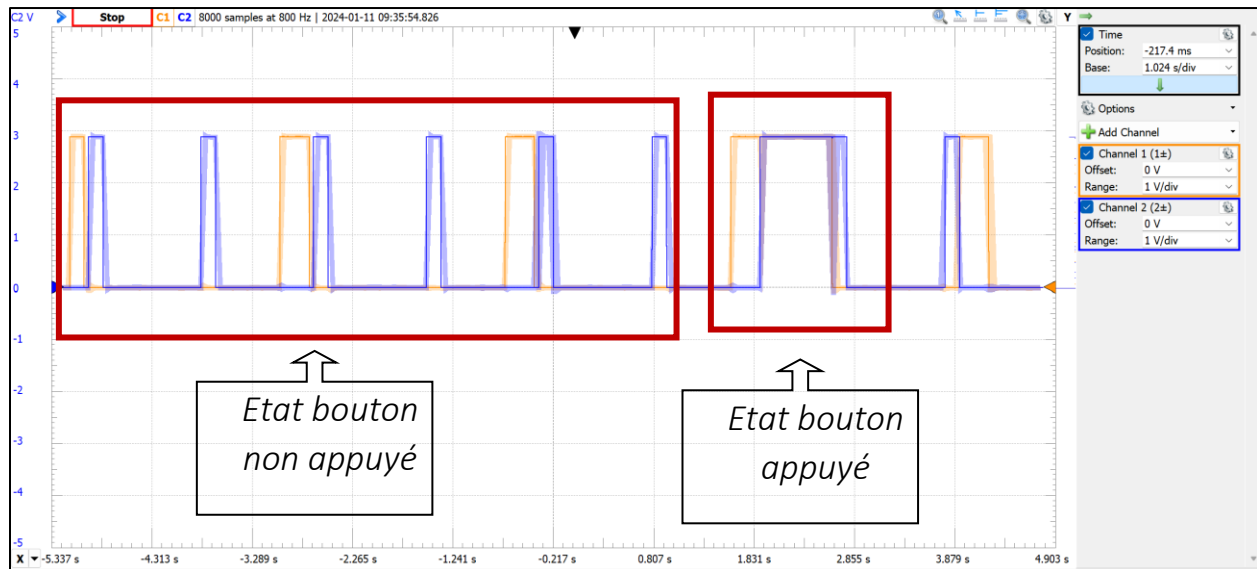
De ce schéma, on peut en déduire que :

- Le signal de la LED1 se trouve sur la pin PG13,
- Le signal de la LED2 se trouver sur la pin PG14,
- Le signal du bouton SW0 se trouver sur la pin PA0

Tests des différents scénarios

Dans cette étude, nous explorons divers scénarios d'attribution de priorité dans le contexte de notre système, en mettant en œuvre des tests pratiques sur la carte. Notre objectif est d'observer le fonctionnement de chaque scénario à travers des analyses à l'oscilloscope, en se concentrant particulièrement sur le temps d'activation des tâches associées aux boutons.

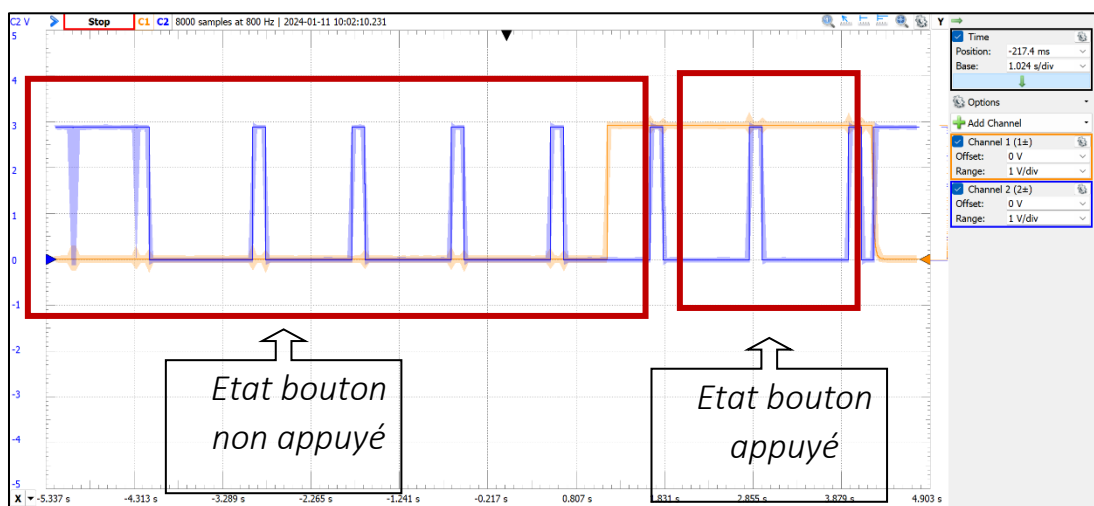
Voici la capture d'écran sur un oscilloscope des tâches des deux LEDs (courbe orange LED2 et courbe bleu LED1) :



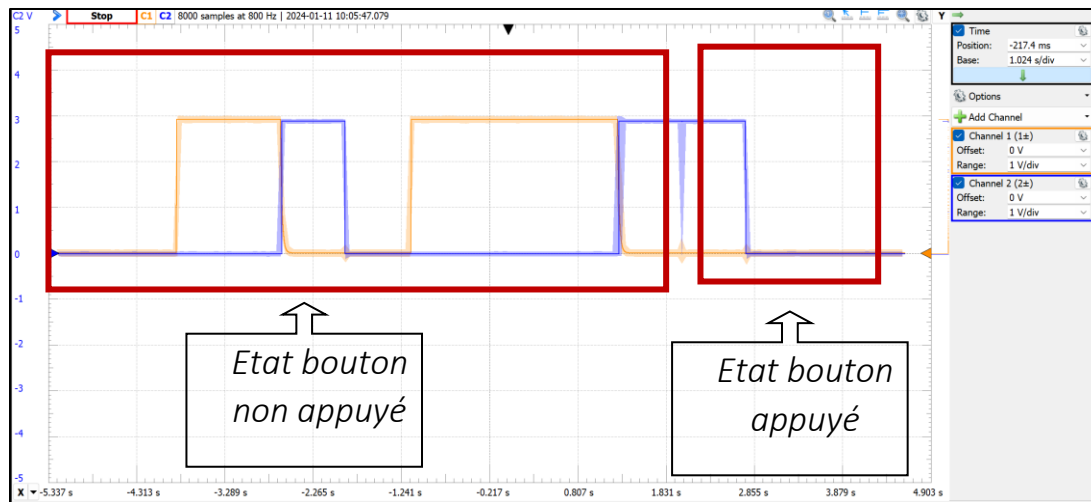
On peut voir qu'un appui sur le bouton les deux LEDs s'allument bien en même temps.

Maintenant, on va chercher à observer le signal de notre bouton en modifiant le code afin de pouvoir observer le délai entre l'appui du bouton et le moment où nos deux LEDs s'allument.

Voici Priorité du bouton à 3 :



Priorité bouton 1 (jamais interrompu toujours en attente de l'appuie) :



Algorithmes de partage de ressources

Objectif

Dans cette partie, nous devons utiliser différents algorithmes d'accès à des ressources partagées, classique en communication inter-processus.

L'algorithme Producteur/Consommateur avec des LEDs

Dans notre contexte, l'incrément et la décrémentation unitaire jouent un rôle crucial dans la gestion de la production et de la consommation.

Production priorité > consommation priorité	Surproduction donc consommation peut agir quand elle veut
Consommation priorité > production priorité	Attente de la production avant consommation
Production prend plus de temps et consommation ne change pas	À un moment la consommation attend la production
Consommation prend moins de temps et production ne change pas	À un moment la consommation attend la production
Consommation prend plus de temps et production ne change pas	Production supérieure à consommation
Production prend moins de temps et consommation ne change pas	Production supérieure à consommation
Production = consommation	Alternance entre les deux leds car équilibre

Lorsque la production a la priorité sur la consommation, cela peut conduire à une surproduction. Dans cette situation, la consommation a la flexibilité d'agir à tout moment.

En revanche, si la consommation a la priorité sur la production, elle doit attendre que la production soit prête avant de pouvoir intervenir. Si la production prend plus de temps que la consommation et que ce dernier reste constant, il en résulte un moment où la consommation attend la production.

De manière similaire, si la consommation prend moins de temps que la production et que ce dernier reste constant, la consommation attendra également la production à un certain moment.

Lorsque la consommation prend plus de temps que la production, cela crée une situation de surproduction, où la production excède la consommation. De même, si la production prend moins de temps que la consommation, on observe également une surproduction.

Lorsque la production équivaut à la consommation, cela entraîne une alternance entre les deux activités, symbolisée par l'allumage alternatif des LED, illustrant un équilibre entre la production et la consommation.

Pour mettre en pratique cela, on crée un programme qui va mettre en place un producteur et un consommateur qui accèdent tous les deux ont une variable partager et qui la modifie, il faudra également que l'on puisse inverser la priorité de ses deux tâches à l'aide d'un bouton.

```
#include <zephyr/kernel.h>
```

```

#include <zephyr/drivers/gpio.h>

#include <zephyr/logging/log.h>
LOG_MODULE_REGISTER(app);

#define STACKSIZE (4096)
static K_THREAD_STACK_DEFINE(product_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(conso_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(sw0_stack, STACKSIZE);

/* The devicetree node identifier for the "led0" alias. */
#define LED0_NODE DT_ALIAS(led0)
#define LED1_NODE DT_ALIAS(led1)
#define SW0_NODE DT_ALIAS(sw0)
static const struct gpio_dt_spec led0 = GPIO_DT_SPEC_GET(LED0_NODE, gpios);
static const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(LED1_NODE, gpios);
static const struct gpio_dt_spec sw0 = GPIO_DT_SPEC_GET_OR(SW0_NODE, gpios,
{0});
static struct gpio_callback sw0_cb_data;

K_MUTEX_DEFINE(mutexLEDs);
K_MUTEX_DEFINE(mutexSupplies);
int switchPushed = 0;
int supplies = 20;

k_tid_t id_product;
k_tid_t id_conso;

void update_leds(uint8_t led0_val, uint8_t led1_val)
{
    if (k_mutex_lock(&mutexLEDs, K_FOREVER) == 0)
    {
        gpio_pin_set_dt(&led0, led0_val);
        gpio_pin_set_dt(&led1, led1_val);
        k_mutex_unlock(&mutexLEDs);
    }
}

static void product_task(void *p1, void *p2, void *p3)
{
    int i, nbIter = 100000;
    while (1)
    {
        for (i = 0; i < nbIter; i++)
        {
            update_leds(1, 0); // allumer LED rouge
        }
        if (k_mutex_lock(&mutexSupplies, K_FOREVER) == 0)
        {
            supplies++;
            k_mutex_unlock(&mutexSupplies);
        }
    }
}

```

```

        update_leds(0, 0);
        k_msleep(300);
    }
}

static void conso_task(void *p1, void *p2, void *p3)
{
    int i, nbIter = 100000;
    while (1)
    {
        for (i = 0; i < nbIter; i++)
        {
            update_leds(0, 1); // allumer LED rouge
        }
        if (k_mutex_lock(&mutexSupplies, K_FOREVER) == 0)
        {
            if(supplies != 0)
            {
                supplies--;
            }
            k_mutex_unlock(&mutexSupplies);
        }
        update_leds(0, 0);
        k_msleep(300);
    }
}

static void sw0_task(void *p1, void *p2, void *p3)
{
    while (1)
    {
        if (switchPushed == 1)
        {
            switchPushed = 0;
            int id_tempo = k_thread_priority_get(id_product);

            k_thread_priority_set(id_product, k_thread_priority_get(id_conso));
            k_thread_priority_set(id_conso, id_tempo);
            printf("priorité conso : %d, priorité product : %d\n", k_thread_priority_get(id_conso), k_thread_priority_get(id_product));
            //LOG_INF("priorité conso : %d, priorité product : %d", k_thread_priority_get(id_conso), k_thread_priority_get(id_product));
        }
        k_yield();
    }
}

void sw0_pressed_callback(const struct device *dev, struct gpio_callback *cb,
uint32_t pins)
{
    LOG_INF("Switch pressed at %d" PRIu32 "\n", k_cycle_get_32());
    if ((pins & (1 << sw0.pin)) != 0)

```



```

    {
        switchPushed = 1;
    }
}

uint8_t init_leds()
{
    uint8_t returned = 0;
    if (!device_is_ready(led0.port) || !device_is_ready(led1.port))
    {
        LOG_ERR("Error: LEDs devices are not ready (%s / %s)", led0.port->name, led1.port->name);
        returned = -1;
    }
    if (gpio_pin_configure_dt(&led0, GPIO_OUTPUT_ACTIVE) < 0 ||
        gpio_pin_configure_dt(&led1, GPIO_OUTPUT_ACTIVE) < 0)
    {
        LOG_ERR("Error: LEDs config failed (%s / %s).", led0.port->name, led1.port->name);
        returned = -2;
    }
    return returned;
}

uint8_t init_switches()
{
    uint8_t returned = 0;

    if (!device_is_ready(sw0.port))
    {
        LOG_ERR("Error: Switch device %s is not ready.", sw0.port->name);
        returned = -1;
    }
    if (!returned && gpio_pin_configure_dt(&sw0, GPIO_INPUT) != 0)
    {
        LOG_ERR("Error: failed to configure switch %s pin %d.", sw0.port->name, sw0.pin);
        returned = -2;
    }
    if (!returned && gpio_pin_interrupt_configure_dt(&sw0, GPIO_INT_EDGE_TO_ACTIVE) != 0)
    {
        LOG_ERR("Error: failed to configure interrupt on %s pin %d.", sw0.port->name, sw0.pin);
        returned = -3;
    }
    if (!returned)
    {
        gpio_init_callback(&sw0_cb_data, sw0_pressed_callback, BIT(sw0.pin));
        gpio_add_callback(sw0.port, &sw0_cb_data);
    }
}

```

```

    LOG_INF("Set up switch at %s pin %d", sw0.port->name, sw0.pin);
    return returned;
}

void main(void)
{
    struct k_thread product_t, conso_t, sw0_t;
    if (init_leds() < 0 || init_switches() < 0)
    {
        LOG_ERR("Error: %s", "LED or Switch init failed");
        return;
    }
    id_product = k_thread_create(&product_t, product_stack,
K_THREAD_STACK_SIZEOF(product_stack),
    product_task, NULL, NULL, NULL,
    1, 0, K_NO_WAIT);
    id_conso = k_thread_create(&conso_t, conso_stack,
K_THREAD_STACK_SIZEOF(conso_stack),
    conso_task, NULL, NULL, NULL,
    2, 0, K_NO_WAIT);
    k_thread_create(&sw0_t, sw0_stack, K_THREAD_STACK_SIZEOF(sw0_stack),
    sw0_task, NULL, NULL, NULL,
    3, 0, K_NO_WAIT);
}

```

Après test, le code fonctionne comme on l'entend avec chacun des threads qui allume une LEDs, celui à la priorité la plus haute allume la sienne le plus souvent et lorsque que l'on appuie sur le bouton, on peut voir l'inversion se faire et l'ordre des LEDs changer.

Les outils utilisés dans ce code sont, un mutex afin que de pouvoir gérer la variable partagée entre nos deux threads, Les GPIO pour la gestion du bouton ainsi que des LEDs.

Comprendre le projet "WithGUI"

La carte STM32F429 possède un écran tactile qui nous permet d'afficher différentes informations et de créer par la même occasion une IHM basique pour cela, on teste avec un projet nommé WithGUI qui comme son nom l'indique, met en place une interface.

Ce capteur va être câblé via I2C, on commence par observer le fichier stm32f429i_disc1.overlay pour savoir quelle PIN, nous allons utiliser.

```

&i2c3 {
    bme680: bme680@77 {
        compatible = "bosch,bme680";
        reg = <0x77>;
        label = "BME680";
    };
};

```

Code stm32f429i_disc1.overlay (I2C3)

On peut donc déduire de ce code que l'I2C utilisé pour le capteur d'humidité/température sur notre carte est l'I2C3.

Avant de commencer à réaliser notre programme en C, il faut d'abord prévoir la manière dont nous allons gérer

```
#include <zephyr/kernel.h>
#include <zephyr/logging/log.h>
LOG_MODULE_REGISTER(app);

#include "display.hpp"
#include "bme680.hpp"

//K_MSGQ_DEFINE(my_msgq, sizeof(char[50]), 10, 1); // message queue

#define STACKSIZE (4096)
static K_THREAD_STACK_DEFINE(display_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(bme680_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(display_esiea_stack, STACKSIZE);

static K_THREAD_STACK_DEFINE(hello_stack, 2048);

#define PRIO_DISPLAY_TASK 1
#define PRIO_BME680_TASK 2
#define PRIO_DISPLAY_ESIEA 3 // bug avec 2
#define PRIO_DISPLAY_HELLO 4

#define PERIOD_DISPLAY_TASK 1000
#define PERIOD_BME680_TASK 500

myDisplay display;
myBME680 bme680;

struct k_thread display_esiea_t;
k_tid_t display_esiea;

char my_msgq_buffer[10 * sizeof(char[50])];
struct k_msgq my_msgq;

K_MUTEX_DEFINE(mutex_msgq); // voir si devoir le mettre en place.

static void display_task(void *p1, void *p2, void *p3)
{
    char name[10] = "DISPLAY";
    k_tid_t tid = k_current_get();
    int period = PERIOD_DISPLAY_TASK;

    char text[50] = {0};
    uint32_t start;

    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
```

```

    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));
    LOG_INF("Run task DISPLAY - Priority %d - Period %d\n",
k_thread_priority_get(tid), period);
    while (1)
    {
        k_timer_status_sync(&timer);
        LOG_INF("START task %s", name);
        start = k_uptime_get_32();
        display.task_handler();
        display.chart_add_temperature(bme680.get_temperature());
        display.chart_add_humidity(bme680.get_humidity());

        // format the string
        sprintf(text, "%d.%02d°C - %d.%02d\n",
bme680.temperature.val1, bme680.temperature.val2 / 10000,
bme680.humidity.val1, bme680.humidity.val2 / 10000);
        k_msgq_put(&my_msgq, &text, K_NO_WAIT);

        LOG_INF("END task %s - %dms", name, k_uptime_get_32() -
start);
    }
}

static void bme680_task(void *p1, void *p2, void *p3)
{
    char name[10] = "BME680";
    k_tid_t tid = k_current_get();
    int period = PERIOD_BME680_TASK;

    uint32_t start;

    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));
    LOG_INF("Run task BME680 - Priority %d - Period %d\n",
k_thread_priority_get(tid), period);
    while (1)
    {
        k_timer_status_sync(&timer);
        LOG_INF("START task %s", name);
        start = k_uptime_get_32();
        bme680.update_values();
        LOG_INF("END task %s - %dms", name, k_uptime_get_32() -
start);
    }
}

static void display_task_esiea(void *p1, void *p2, void *p3)
{
    int period = PERIOD_DISPLAY_TASK;
    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));

```

```

    while(1)
    {
        k_timer_status_sync(&timer);
        // tant que la message queue n'est pas vide
        char receive[50] = {0};
        while(k_msgq_get(&my_msgq, &receive, K_FOREVER) == 0)
        {
            display.text_add(receive);
        }
    }
}

static void display_task_hello(void *p1, void *p2, void *p3)
{
    int period = PERIOD_DISPLAY_TASK;
    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));
    while(1)
    {
        k_timer_status_sync(&timer);

        char text[50] = "Hello!";
        k_msgq_put(&my_msgq, &text, K_NO_WAIT);
    }
}

int main(void)
{
    struct k_thread display_t;
    struct k_thread bme680_t;
    struct k_thread hello_t;

    k_msgq_init(&my_msgq, my_msgq_buffer, sizeof(char[50]), 10);

    display.init(true);
    bme680.init();

    k_thread_create(&display_t, display_stack,
K_THREAD_STACK_SIZEOF(display_stack),
display_task, NULL, NULL, NULL,
PRIO_DISPLAY_TASK, 0, K_NO_WAIT);

    k_thread_create(&bme680_t, bme680_stack,
K_THREAD_STACK_SIZEOF(bme680_stack),
bme680_task, NULL, NULL, NULL,
PRIO_BME680_TASK, 0, K_NO_WAIT);

    display_esiea = k_thread_create(&display_esiea_t,
display_esiea_stack, K_THREAD_STACK_SIZEOF(display_esiea_stack),
display_task_esiea, NULL, NULL, NULL,
PRIO_DISPLAY_ESIEA, 0, K_NO_WAIT);
}

```

```

        k_thread_create(&hello_t,
        K_THREAD_STACK_SIZEOF(hello_stack),
        display_task_hello, NULL, NULL, NULL,
        PRIO_DISPLAY_HELLO, 0, K_NO_WAIT);
    }

```

Code main.cpp projet "WithGUI"

Pour décrire un peu ce code EXPLICATION CODE,

L'algorithme Lecteur/Rédacteurs avec un écran

Dans cette partie, on cherche à programmer l'algorithme des lecteurs/rédacteurs avec un nombre de tâches "rédacteur de message" fixé au départ (Par exemple dans t_nbWriters = 5;) et un seul "lecteur" pour l'affichage. Pour cela, on va reprendre le code.

```

#include <zephyr/kernel.h>
#include <zephyr/logging/log.h>
LOG_MODULE_REGISTER(app);

#include "display.hpp"

#define DELAY_START_TIME_MS 5000

int nbWriters = 5; //nb de thread a créer

#define STACKSIZE (4096)
static K_THREAD_STACK_DEFINE(lecteur_stack, STACKSIZE);
static K_THREAD_STACK_ARRAY_DEFINE(thread_stacks, 10, STACKSIZE);

#define PERIOD_DISPLAY_TASK 1000

myDisplay display;

char my_msgq_buffer[10 * sizeof(char[50])];
struct k_msgq my_msgq;

struct k_thread threads[10];

K_MUTEX_DEFINE(mutex_msgq);

static void lecteur_task(void *p1, void *p2, void *p3)
{
    int period = PERIOD_DISPLAY_TASK;
    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));
    while(1)
    {
        k_timer_status_sync(&timer);
        char receive[50] = {0};
    }
}

```

```

        while(k_msgq_get(&my_msgq, &receive, K_FOREVER) == 0) // tant
que la message queue n'est pas vide
        {
            display.text_add(receive);
        }
    }
}

static void generic_redacteur_taks(void *p1, void *p2, void *p3)
{
    int period = PERIOD_DISPLAY_TASK;

    char text[50] = {0};
    int thread_id = (int)p1;

    uint32_t start;
    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));
    while (1)
    {
        k_timer_status_sync(&timer);
        start = k_uptime_get_32();
        display.task_handler();

        // format the string
        snprintf(text, sizeof(text), "redacteur nb:%d is running",
thread_id);
        k_msgq_put(&my_msgq, &text, K_NO_WAIT);
    }
}

int main(void)
{
    struct k_thread lecteur_t;

    int i;

    k_msgq_init(&my_msgq, my_msgq_buffer, sizeof(char[50]), 10);

    display.init(false); //mise a false pour ne pas avoir de graph

    //création thread lecteur
    k_thread_create(&lecteur_t, lecteur_stack,
K_THREAD_STACK_SIZEOF(lecteur_stack), lecteur_task,
NULL, NULL, NULL, 1, 0, K_NO_WAIT);

    // boucle creation thread
    for (int i = 0; i < nbWriters; i++) {
        k_tid_t thread_id = k_thread_create(&threads[i],
thread_stacks[i],

```

```
K_THREAD_STACK_SIZEOF(thread_stacks[i]),
                                generic_redacteur_taks,
                                (void *)i, NULL, NULL,
                                2, 0, K_NO_WAIT);
    k_thread_name_set(&threads[i], "Thread redacteur");
}
```

Code Lecteur/Rédacteurs avec écran v1

La principale difficulté de ce code a été de rendre générique la création de threads. Pour cela, nous avons créé une fonction thread `generic_redacteur_task()` que l'on va réutiliser pour chacun de nos threads. On retrouve la boucle de création de nos threads « rédacteurs » dans notre main, cette boucle est un `for` qui dépend de la variable `nbWriters`.

Nous allons maintenant créer une deuxième version de ceux où cette fois-ci nous pouvons ajouter des tâches « rédacteurs » à l'aide du bouton-poussoir.


```

static K_THREAD_STACK_DEFINE(sw0_stack, STACKSIZE);

#define SW0_NODE DT_ALIAS(sw0)
static const struct gpio_dt_spec sw0 = GPIO_DT_SPEC_GET_OR(SW0_NODE, gpios,
{0});
static struct gpio_callback sw0_cb_data;

static void sw0_task(void *p1, void *p2, void *p3)
{
    while (1)
    {
        if (switchPushed == 1)
        {
            switchPushed = 0;
            nbWriters++;

        }
        k_yield();
    }
}

uint8_t init_switches()
{
    uint8_t returned = 0;

    if (!device_is_ready(sw0.port))
    {
        LOG_ERR("Error: Switch device %s is not ready.", sw0.port->name);
        returned = -1;
    }
    if (!returned && gpio_pin_configure_dt(&sw0, GPIO_INPUT) != 0)
    {
        LOG_ERR("Error: failed to configure switch %s pin %d.", sw0.port->name, sw0.pin);
        returned = -2;
    }
    if (!returned && gpio_pin_interrupt_configure_dt(&sw0, GPIO_INT_EDGE_TO_ACTIVE) != 0)
    {
        LOG_ERR("Error: failed to configure interrupt on %s pin %d.", sw0.port->name, sw0.pin);
        returned = -3;
    }
    if (!returned)
    {
        gpio_init_callback(&sw0_cb_data, sw0_pressed_callback,
        BIT(sw0.pin));
        gpio_add_callback(sw0.port, &sw0_cb_data);
    }
    LOG_INF("Set up switch at %s pin %d", sw0.port->name, sw0.pin);
    return returned;
}

```

```

}

void sw0_pressed_callback(const struct device *dev, struct gpio_callback *cb,
uint32_t pins)
{
    LOG_INF("Switch pressed at %d" PRIu32 "\n", k_cycle_get_32());
    if ((pins & (1 << sw0.pin)) != 0)
    {
        switchPushed = 1;
    }
}

int main(void)
{
    init_switches();
    k_thread_create(&sw0_t, sw0_stack, K_THREAD_STACK_SIZEOF(sw0_stack),
                    sw0_task, NULL, NULL, NULL,
                    3, 0, K_NO_WAIT);
}

```

Ajout code Lecteur/Rédacteurs avec écran v2

Ici, nous avons seulement ajouté la gestion du bouton, soit :

- Une fonction pour initialiser notre GPIO en tant que bouton (entrée).
- Une fonction de callback pour détecter nos appuies bouton.
- Une fonction thread qui va incrémenter la variable utilisée dans notre boucle qui créait les threads.

Il faut également savoir que dans ce cas nous sommes limités par 2 choses, l'attribution de taille de stack qui n'est pas infini ainsi que la taille de notre tableau de structure de thread.

Ordonnancement Temps Réel

Définitions et jeu de tâches

Reposons nos définitions. Nous considérons que :

- Un jeu de tâches τ composé de n tâches est défini comme :
 - $\tau = \{\tau_1(O_1, C_1, T_1, D_1), \dots, \tau_n(O_n, C_n, T_n, D_n)\}$
- Une tâche $\tau_i = \tau_i(O_i, C_i, T_i, D_i)$ est définie par ce 4-tuplet avec :
 - O_i est le premier instant d'arrivée de τ_i , i.e., l'instant de la première activation depuis le démarrage du système.
 - C_i est le Worst Case Execution Time (WCET) de τ_i , i.e., le temps maximum d'exécution requis pour chacun de ses jobs (pour chacune de ses activations).
 - T_i est la période (périodique) ou inter-arrivée (sporadique) de la τ_i , i.e., l'inter-arrivée exacte (périodique) ou minimum (sporadique) entre deux activations successives de la tâche.
 - D_i est le deadline relatif de τ_i , i.e., le temps avant lequel le job courant doit se terminer relativement à sa date d'activation

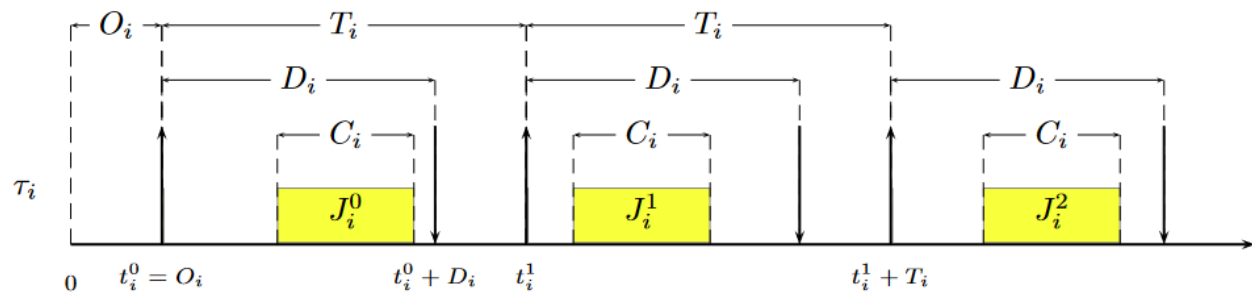


Figure 5.1: Modèle de tâche séquentielle périodique.

Voici le tableau des paramètres des tâches :

5.2	Tâche Réveil (O_i)	Durée (C_i)	Période (T_i)	Deadline (D_i)
τ_1	1	2	10	10
τ_2	0	1	6	6
τ_3	2	8	15	15

Comprendre "PeriodicTasks" être produire l'exemple

Dans cette partie, nous allons utiliser un code qui va nous permettre de générer un jeu de tâches et d'en choisir les différents paramètres tel que, la période, le réveil ou encore la priorité.

```
typedef struct task_t
{
    char name[20];
    struct k_thread thread_p;
    struct z_thread_stack_element *thread_stack;
    int start;
    int period;
    int cpu;
    int priority;
    uint8_t led0;
    uint8_t led1;
} task;
```

main.c du projet "PeriodicTasks" (1)

La première partie importante de ce code est cette structure qui définit les paramètres que vont prendre chacun de nos différents threads, cela va nous permettre de les générer à l'aide d'une boucle et ainsi de pouvoir choisir le nombre de threads souhaité. Les paramètres sont les suivants :

- Le nom du thread (char[20]),
- La ligne de création d'un élément k_thread pour créer un thread avec zephyr,
- La taille du stack de notre thread,
- Le moment de réveil du thread (int),
- La période (int),
- L'utilisation du CPU (int),
- La priorité (int),
- Deux uint8_t pour piloter les LEDs.

```

#define DELAY_START_TIME_MS 5000
#define STACKSIZE (1024)
static K_THREAD_STACK_DEFINE(thread0_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(thread1_stack, STACKSIZE);

task tasks[] = {
    {.name = "T1", .thread_stack = thread0_stack,
     .start = 0000, .period = 1000, .cpu = 400,
     .priority = 1, .led0 = 1, .led1 = 0}, // Bleu
    {.name = "T2", .thread_stack = thread1_stack,
     .start = 1000, .period = 3000, .cpu = 700,
     .priority = 2, .led0 = 0, .led1 = 1}}; // Jaune

```

main.c du projet "PeriodicTasks" (2)

Ici, on crée un tableau qui va contenir la définition des différents threads que l'on veut créer. Pour ce faire, on reprend la structure vue précédemment. Dans notre exemple, deux threads sont créés l'un nommé T1 l'autre T2 :

- T1, est un thread qui commence à 0 ms (dès le départ) avec une période de 3000 ms et une priorité de 1
- T2, est un thread qui commence à 1000 ms avec une période de 3000 ms et une priorité de 1.

```

void burnCPU(uint16_t ms, uint8_t led0, uint8_t led1)
{
    while (ms--)
    {
        for (uint16_t i = 0; i < cal; i++)
        {
            update_leds(led0, led1);
            __asm("nop");
        }
    }
}

void calibrateBurnCPU()
{
    update_leds(1, 1);
    uint32_t objective_ms = 1000;
    uint32_t measured_time = k_uptime_get();
    burnCPU(objective_ms, 1, 1);
    measured_time = k_uptime_get() - measured_time;
    cal = cal * objective_ms / measured_time;
    update_leds(0, 0);
}

```

main.c du projet "PeriodicTasks" (3)

La première fonction ci-dessus permet de gérer l'utilisation du CPU par chacun des threads en fonction de la valeur que l'on a donné au paramètre CPU ci-dessus.

La seconde fonction calibre la fonction précédente burnCPU() pour ce faire elle lance cette dernière avec un temps de 1000 ms puis retiens le temps mis par la fonction burnCPU et calcul une valeur cal afin d'être bien calibrer lors des prochaines utilisations de la fonction.

```
static void generic_task_entry(void *p1, void *p2, void *p3)
{
    char *name = ((task *)p1)->name;
    uint16_t start = ((task *)p1)->start;
    uint16_t period = ((task *)p1)->period;
    uint16_t cpu = ((task *)p1)->cpu;
    uint8_t led0 = ((task *)p1)->led0;
    uint8_t led1 = ((task *)p1)->led1;

    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(start), K_MSEC(period));
    LOG_INF("Run task %s - Priority %d", name, ((task *)p1)->priority);
    while (1)
    {
        k_timer_status_sync(&timer);
        LOG_INF("START task %s", name);
        burnCPU(cpu, led0, led1);
        LOG_INF("END task %s", name);
        update_leds(0, 0);
    }
}
```

main.c du projet "PeriodicTasks" (4)

Ci-dessus, on retrouve la fonction que nous allons appeler pour pouvoir générer nos tâches (celles définissent dans le tableau précédemment).

Cette fonction prend en paramètre l'adresse d'un pointeur p1 qui va nous permettre d'accéder aux informations que l'on veut utiliser pour définir les threads à créer. Cette fonction est cadencée par un timer avec une période qui dépend de la période du thread à l'adresse de p1.

```

void main(void)
{
    uint8_t i, nb_task = sizeof(tasks) / sizeof(tasks[0]);

    if (init_leds() < 0)
    {
        LOG_ERR("Error: %s", "LED init failed");
        return;
    }

    calibrateBurnCPU();
    for (i = 0; i < nb_task; i++)
    {
        LOG_INF("Prepare task %d\n\tstart %d\n\tperiod: %d\n\tcpu: %d\n\tpriority: %d\n",
                i, tasks[i].start, tasks[i].period,
                tasks[i].cpu, tasks[i].priority);

        k_thread_create(&tasks[i].thread_p, tasks[i].thread_stack,
            STACKSIZE,
            generic_task_entry, (void
            *)&tasks[i], NULL, NULL,

            K_PRIO_PREEMPT(tasks[i].priority), 0,
            tasks[i].start));
    }
}

```

main.c du projet "PeriodicTasks" (5)

La dernière partie importante de ce code est le fonction main, qui commence par qui utilise une division astucieuse entre la taille totale de tous les threads et la taille du premier thread afin d'obtenir le nombre de threads que nous allons devoir créer.

Ensuite, on initialise nos LEDs et on appelle la fonction calibrateBurnCPU() (décrite précédemment). Une boucle for (qui boucle en fonction de nb_task) qui fait appel à la fonction k_thread_create() qui va elle-même utiliser le tableau de structure tasks[] afin d'obtenir les paramètres de chacun des threads que l'on veut créer.

On va prendre un le code précédent et le modifier avec les valeurs du tableau ci-dessous afin de tester ce code ainsi que notre jeu de tache :

Taches	Réveil (ms)	Durée (ms)	Période (ms)	Deadline (ms)
T1	1 000	2 000	10 000	10 000
T2	0	1 000	6 000	6 000
T3	2 000	8 000	15 000	15 000

Le paramètre que l'on va devoir déduire pour notre code est la priorité de chaque tâche. Ici, toutes nos deadlines sont égaux à nos périodes, on peut donc se baser sur l'algorithme RM qui nous dit que plus la période d'une tâche est petite plus cette dernière aura une priorité élevée. Finalement

```
task tasks[] = {
    {.name = "T1", .thread_stack = thread0_stack,
     .start = 1000, .period = 10000, .cpu = 2000,
     .priority = 2, .led0 = 1, .led1 = 0}, // Bleu
    {.name = "T2", .thread_stack = thread1_stack,
     .start = 0000, .period = 6000, .cpu = 1000,
     .priority = 1, .led0 = 0, .led1 = 1}, // Jaune
    {.name = "T3", .thread_stack = thread2_stack,
     .start = 2000, .period = 15000, .cpu = 8000,
     .priority = 3, .led0 = 1, .led1 = 1}}; // Jaune et Bleu
```

main.c modifié du projet "PeriodicTasks"

Après le lancement du code avec les paramètres ci-dessus, on peut voir à l'aide des LEDs de notre carte que les priorités ont bien été respectées. Nous allons utiliser un oscilloscope afin de pouvoir vérifier le bon fonctionnement de notre jeu de tache.

Test d'ordonnabilité :

Est-ce ordonnançable avec EDF et RM ?

On suppose que ce système est ordonnable avec Rate Monotonic (RM) Ainsi, l'ordre est τ_2, τ_1, τ_3 . Par le « Necessary and sufficient test » (NS-test) du « Worst Case Response Time » (WCRT), on obtient :

$$\tau_3 \Rightarrow WCRT^5 = 15 \leq D_3 = 15$$

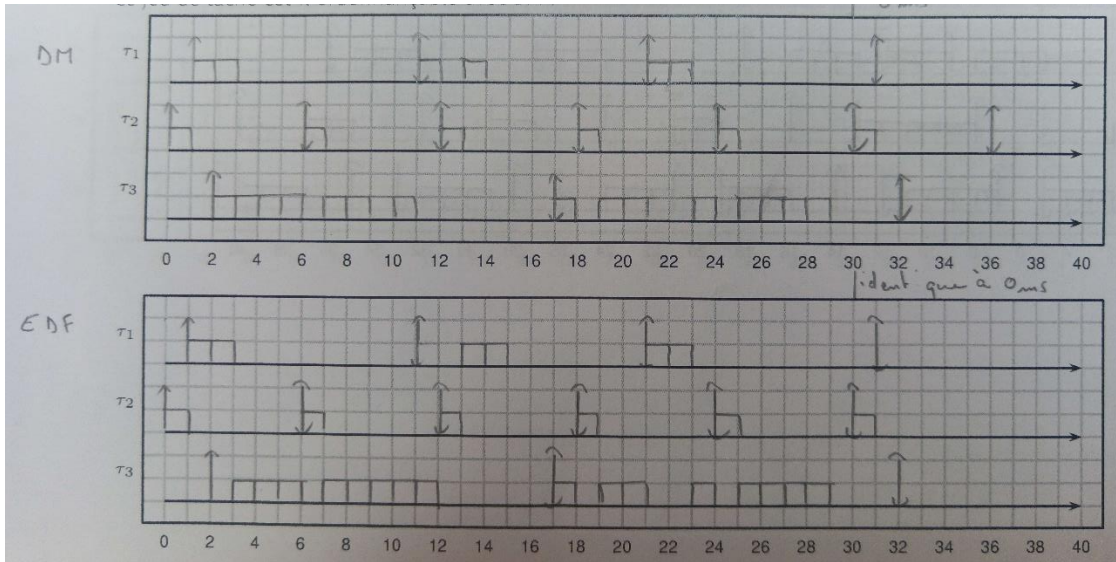
$$\tau_2 \Rightarrow WCRT^5 = 1 \leq D_2 = 6$$

$$\tau_1 \Rightarrow WCRT^5 = 3 \leq D_1 = 10$$

WCRT	τ_3	τ_2	τ_1
1	8	1	2
2	12	1	3
3	14	1	3
4	15	1	3
5	15	1	3

Donc, ce système est ordonnançable avec RM et ainsi que EDF (cas solution optimale).

Voici les graphes de EDF et RM :



Et si τ_2 est plus contrainte ?

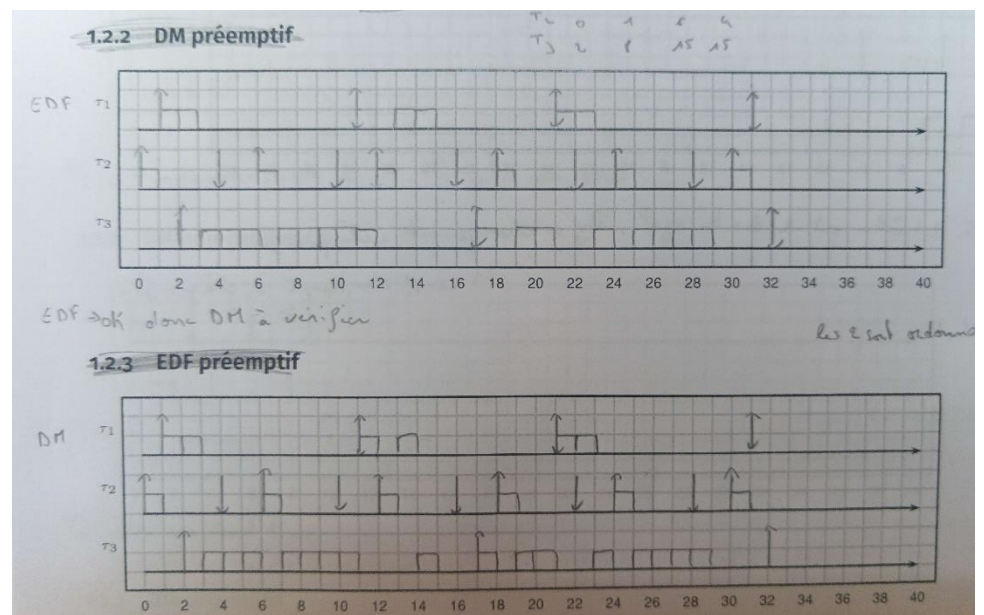
On fixe maintenant le deadline de la tâche τ_2 à 4 (au lieu de 6).

Est-ce ordonnançable avec EDF et DM ?

5,3	Tâche Réveil (O_i)	Durée (C_i)	Période (T_i)	Deadline (D_i)
τ_1	1	2	10	10
τ_2	0	1	6	4
τ_3	2	8	15	15

En change le deadline relatif de τ_2 de 6 à 4, on observe qu'il n'y a pas de changement de priorité pour Deadline Monotonic(DM) par rapport à Rate Monotonic (RM) dans l'ordonnancement des tâches. Ainsi, les valeurs de WCRT ne changent pas. Donc, ce système reste ordonnançable dans le pire cas avec DM et donc EDF.

Voici les graphes de EDF et DM :



Pourquoi est-il plus cohérent d'utiliser DM que RM dans ce cas ?

Dans ce contexte, il est plus logique d'utiliser DM, car nous avons une contrainte de deadline pour τ_2 . Cela n'a donc pas d'impact sur nos priorités, car nos tâche τ_2 avait déjà la haute priorité.

```
task tasks[] = {
    {.name = "T1", .thread_stack = thread0_stack,
     .start = 1000, .period = 10000, .cpu = 2000,
     .priority = 2, .led0 = 1, .led1 = 0}, // Bleu
    {.name = "T2", .thread_stack = thread1_stack,
     .start = 0000, .period = 6000, .cpu = 1000,
     .priority = 1, .led0 = 0, .led1 = 1}, // Jaune
    {.name = "T3", .thread_stack = thread2_stack,
     .start = 2000, .period = 15000, .cpu = 8000,
     .priority = 3, .led0 = 1, .led1 = 1}}; // Jaune et Bleu
```

Et si en plus, τ_3 est plus longue ?

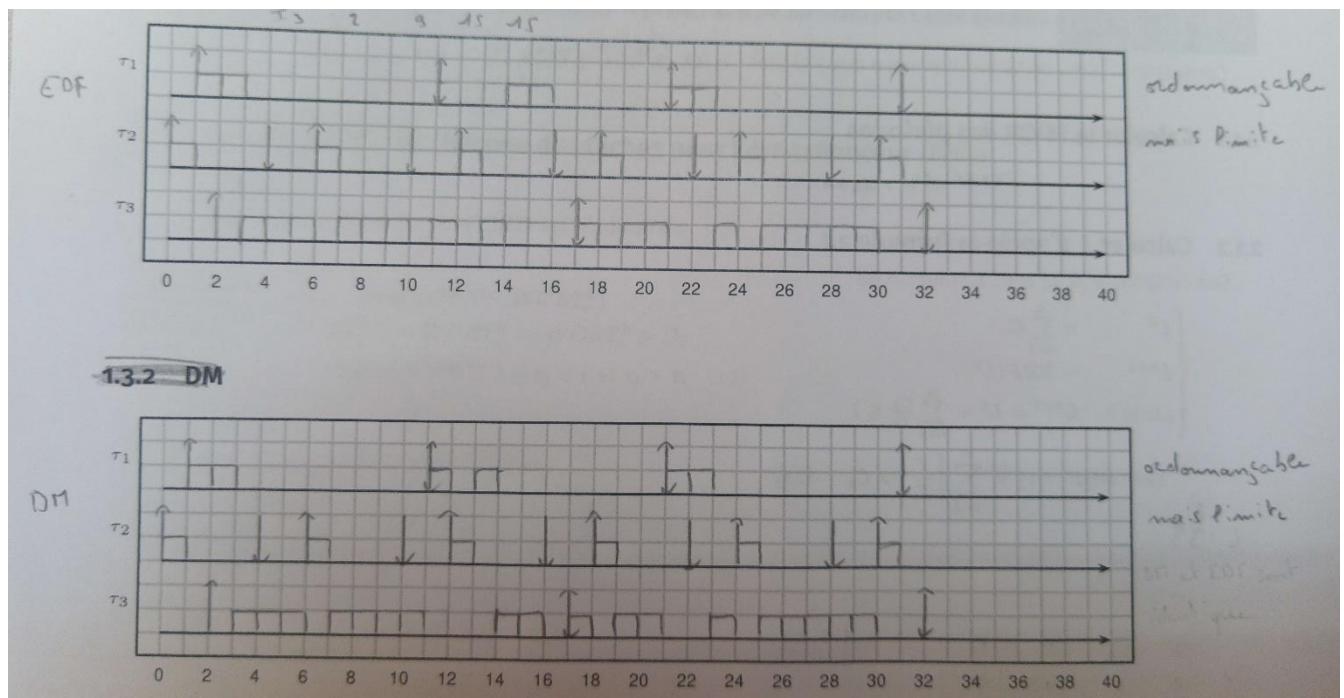
5,4	Tâche Réveil (Oi)	Durée (Ci)	Période (Ti)	Deadline (Di)
τ_1	1	2	10	10
τ_2	0	1	6	4
τ_3	2	9	15	15

Est-ce ordonnançable avec EDF et DM ?

D'après de le NS-test, le système n'est pas ordonnançable dans la pire case de réponse temporelle. Il reste ainsi à savoir si dans notre cas spécifique le système est ordonnançable.

5,4	τ_1	τ_2	τ_3
1	8	1	2
2	13	1	3
3	16	1	3

Dans notre cas, il a l'air possiblement ordonnançable.



Problèmes

Pour montrer notre compréhension des concepts vus en cours (et surtout pour mieux les appréhender), nous allons essayer de répondre à un "Problème ». Dans cette partie, nous allons essayer de répondre à un problème de A à Z.

Enoncé

Mise en place de l'expérience

Depuis le code GUI avec BME, les modifications des tâches pour répondre aux spécifications techniques sont :

1. Les dernières valeurs lues du capteur doivent, à chaque instant, dater de moins de 500 ms.
2. La dernière valeur visible par l'utilisateur sur le graphique doit dater d'au plus 2 secondes.
3. Ajouter une tâche qui permet de lire la valeur de distance d'un capteur infrarouge régulièrement.
4. Ajouter une tâche de contrôle qui, en fonction de la valeur de distance disponible (il ne demandera pas de nouvelle lecture au capteur) décide d'allumer une LED.
5. Fixer un seuil de distance et mesurer le temps entre le moment où la bonne distance est lue par le capteur, et le temps où la tâche de contrôle réagit. (Pour les tests, vous avez le droit d'ajouter du code dans la tâche de lecture pour qu'elle indique elle aussi quand le seuil est dépassé, par exemple sur une autre LED).

Cahier des charges

Dans cette partie, nous allons lister les tâches à générer pour simuler notre système :

- Tâche acquérir valeurs infrarouge (remplacer par un potentiomètre pour les tests)
 - Écrire la valeurs ADC et mettre dans le message queue.
- Tâche d'affichage graphique sur l'écran,
 - Lecture valeurs capteurs (« message queue ») puis affichage graphique (« get »). (Moins prioritaire)
- Tâche pour freiner (allumer LED).
 - Lecture valeur capteur (« message queue », « peak »).

La tâche de freinage et la tâche d'acquisition ont des valeurs infrarouges à une période identique, car la tâche de freinage.

Code

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>
#include <stdio.h>
#include <string.h>

#include "display.hpp"
#include "adc.hpp"

#define LOG_LEVEL CONFIG_LOG_DEFAULT_LEVEL
#include <zephyr/logging/log.h>
LOG_MODULE_REGISTER(app);

myDisplay display;
myADC adc;

#define LED0_NODE DT_ALIAS(led0)
#define LED1_NODE DT_ALIAS(led1)
static const struct gpio_dt_spec led0 = GPIO_DT_SPEC_GET(LED0_NODE, gpios);
static const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(LED1_NODE, gpios);

#define DELAY_START_TIME_MS 5000

#define STACKSIZE (4096)
static K_THREAD_STACK_DEFINE(freiner_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(affichage_stack, STACKSIZE);
static K_THREAD_STACK_DEFINE(update_adc_stack, STACKSIZE);

#define PERIOD_DISPLAY_TASK 1000
#define PERIOD_ACQUIS_TASK 500

static void freiner_task (void *p1, void *p2, void *p3)
{
    char name[10] = "FREINER";
    k_tid_t tid = k_current_get();
    uint32_t start;

    int period = PERIOD_ACQUIS_TASK;
    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));
    while(1)
    {
        k_timer_status_sync(&timer);
        display.task_handler();
        gpio_pin_set_dt(&led0, 0);
        gpio_pin_set_dt(&led1, 0);
        if(adc.get_value() > 30)
        {
            gpio_pin_set_dt(&led0, 1);
            gpio_pin_set_dt(&led1, 1);
        }
    }
}
```

```

    }
    }
    printf("END task %s - %dms", name, k_uptime_get_32() - start);
}

static void affichage_task(void *p1, void *p2, void *p3)
{
    char name[10] = "AFFICHAGE";
    k_tid_t tid = k_current_get();
    uint32_t start;

    int period = PERIOD_DISPLAY_TASK;
    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));
    while(1)
    {
        k_timer_status_sync(&timer);
        display.task_handler();
        char text[30] = {0};
        snprintf(text, sizeof(text), "valeur ADC : %d",
adc.get_value());
        display.text_add(text);
    }
    printf("END task %s - %dms", name, k_uptime_get_32() - start);
}

static void update_adc_task(void *p1, void *p2, void *p3)
{
    char name[10] = "UPDATE";
    k_tid_t tid = k_current_get();
    uint32_t start;

    int period = PERIOD_ACQUIS_TASK;
    struct k_timer timer;
    k_timer_init(&timer, NULL, NULL);
    k_timer_start(&timer, K_MSEC(0), K_MSEC(period));
    while(1)
    {
        k_timer_status_sync(&timer);
        adc.update_value();
    }
    printf("END task %s - %dms", name, k_uptime_get_32() - start);
}

uint8_t init_leds()
{
    uint8_t returned = 0;
    if (!device_is_ready(led0.port) || !device_is_ready(led1.port))
    {
        LOG_ERR("Error: LEDs devices are not ready (%s / %s)", led0.port->name, led1.port->name);
    }
}

```

```

        returned = -1;
    }

    if (gpio_pin_configure_dt(&led0, GPIO_OUTPUT_ACTIVE) < 0 ||
        gpio_pin_configure_dt(&led1, GPIO_OUTPUT_ACTIVE) < 0)
    {
        LOG_ERR("Error: LEDs config failed (%s / %s).", led0.port->name,
            led1.port->name);
        returned = -2;
    }
    return returned;
}

int main(void)
{
    struct k_thread affichage_t, frieren_t, update_t;

    display.init(false);
    adc.init();
    display.task_handler();
    init_leds();

    //creation de nos tâches
    k_thread_create(&affichage_t, affichage_stack,
        K_THREAD_STACK_SIZEOF(affichage_stack),
        affichage_task, NULL, NULL, NULL, 1, 0,
        K_NO_WAIT);

    k_thread_create(&frieren_t, freiner_stack,
        K_THREAD_STACK_SIZEOF(freiner_stack),
        freiner_task, NULL, NULL, NULL, 2, 0,
        K_NO_WAIT);

    k_thread_create(&update_t, update_adc_stack,
        K_THREAD_STACK_SIZEOF(update_adc_stack),
        update_adc_task, NULL, NULL, NULL, 1,
        0, K_NO_WAIT);
}

```

Ce code reprend beaucoup des outils utilisés précédemment tel que les GPIO pour nos LEDs, les fonctions d’affichages de l’écran, etc. Ce code contient 3 threads :

- `Freiner_task()` : cette tâche va vérifier la valeur de notre ADC (capteur infrarouge) puis en fonction de sa valeur allume des LEDs ou non (simulation d’activation du freinage). La période de ce thread est de 1000 ms.
- `Affichage_task()` : pour pouvoir afficher nos valeurs ADC sur l’écran a l’aide de la fonction `display.text_add()`.
- `Update_adc_task()` : cette fonction update les valeurs de l’ADC toutes les 500 ms (période du thread).

Ce code devrait contenir un autre thread pour la gestion d'un graphique et d'un capteur de température et d'humidité que l'on peut retrouver dans les parties précédentes, mais nous avons préféré séparer ces parties.

Résultat expérimental

Nous n'avons pas pu récupérer les valeurs expérimentales pour des raisons diverses. À la place, nous allons nous baser sur des ordres de grandeur temps d'exécution de thread que l'on peut trouver sur ChatGTP ou des forums en ligne.

Le problème de cette solution est la fiabilité de l'information sur notre situation, car les temps d'exécution dépendent de la carte utilisée, la fréquence utilisée. Par exemple pour l'ADC, elle dépend de la résolution de l'ADC, fréquence d'échantillonnage et etc. L'autre problème est l'optimisation de notre système. Ayant des valeurs très théoriques, nous ne pouvons pas optimiser les délais d'exécution des tâches.

Dans ce contexte, nous avons pris la valeur obtenue est multiplier par 2 leurs valeurs d'exécution. Ainsi, nous sommes sûr que dans le pire cas que le temps d'exécution sera dans l'intervalle de 0 à la valeur trouver. Voici le tableau de valeur :

	O_i	C_i	T_i	D_i
BME	X	10	500	500
Freinage	X	8	X	X
ADC	X	10	500	500
Display	X	33	2000	2000

Avec marge d'erreur, on obtient :

	O_i	C_i	T_i	D_i
BME	X	20	500	500
Freinage	X	16	X	X
ADC	X	20	500	500
Display	X	66	2000	2000

Questionnements

En considérant qu'un système de freinage automatique, et que les freins sont parfaits (quand la LED de contrôle s'allume, le robot s'arrête) :

1. Quel est le pire temps de réponse de la tâche de contrôle ?
2. Quelle vitesse maximum conseillez-vous ?
3. Quels sont les paramètres applicables à vos tâches pour augmenter la vitesse possible ?
4. Quels sont les paramètres d'affichage minimum possible pour voir les dernières données le plus souvent possible ?
5. Oh et... ils se passent quoi si on remplace le capteur de distance par un capteur ultrason ?

Nous détaillerons, expliquerons, mesurerons et testerons nos différents scénarios.

Analyse

D'après les résultats expérimentaux, nous pouvons utiliser le WCRT afin de déterminer l'ordonnabilité du système. Tout d'abord, l'ordonnanceur utilise un algorithme à priorité fixe. Il peut être DM ou RM.

Mais il reste quand même à déterminer la fréquence d'exécution. Tout d'abord, je pense que la fréquence d'activation de la fonction de frein devrait au minimum être 2 fois la fréquence de lecture de l'ADC, car en s'inspirant de la loi de Nyquist-Shannon, il faut au minimum 2 fois la fréquence pour pouvoir tester la valeur rafraîchie par le thread de l'ADC à la moins une fois.

On ajoute ensuite une marge d'erreur. De façon arbitraire, nous avons décidé de le multiplier par 2. Tout cela revient à 4 fois la fréquence du thread ADC.

On obtient ces valeurs :

Tâche	O_i	C_i	T_i	D_i	Priorité
BME	X	20	500	500	3
Freinage	X	16	125	125	1
ADC	X	20	500	500	2
Display	X	66	2000	2000	4

WCRT	BME	Freinage	ADC	Display
1	56	16	36	122
2	56	16	36	122
3	56	16	36	122

Une réponse au questionnement

Tout la chaîne d'information prend dans le pire cas la somme des temps des exécution + temps fréquence des tâches. Ce qui nous donne : $125 + 500 + 16 + 20 = 661$

Sachant, que le temps de freinage est parfait donc délai nul, on peut se baser sur la valeur de temps d'exécution du thread et temps d'exécution du capteur. Dans notre cas, on peut supposer que le temps de parcours de la lumière pour le capteur est négligeable par rapport au pire temps de réactivité.

Ainsi, nous pouvons faire un tableau des distances parcouru avant arrêt par le robot en fonction de sa vitesse. Maintenant, il reste à savoir à quelle distance on souhaite s'arrêter pour déterminer la vitesse où l'on souhaite ce déplacé.

Pour augmenter la vitesse du robot, il faut diminuer le temps de réactivité du système. Donc, les paramètres à faire varier seraient par déduction serait la fréquence d'exécution des tâches sachant que ce sont les seules variables modifiables dans nos systèmes. Ainsi, la variable à modifier est la fréquence de lecture de capteur infrarouge, car il n'a pas beaucoup de sens d'augmenter la fréquence d'exécution du freinage si la valeur de l'ADC ne change pas mis à part de diminuer le pire temps de réaction du robot.

Par exemple avec ces paramètres :

Tâche	O_i	C_i	T_i	D_i	Priorité
BME	X	20	500	500	3
Freinage	X	16	23,75	23,8	1
ADC	X	20	95	95	2
Display	X	66	2000	2000	4

WCRT	BME	Freinage	ADC	Display
1	56	16	36	122
2	88	16	52	222
3	104	16	68	306
19	188	16	68	946
20	188	16	68	946

Vitesse(m/s)	Temps réaction(ms)	Distance fait(m)
0	661	0,0
1	661	0,7
2	661	1,3
3	661	2,0
4	661	2,6
5	661	3,3
6	661	4,0
7	661	4,6
8	661	5,3
9	661	5,9
10	661	6,6
20	661	13,2
30	661	19,8

Vitesse	Temps réaction	Distance fait
0	155	0,0
1	155	0,2
2	155	0,3
3	155	0,5
4	155	0,6
5	155	0,8
6	155	0,9
7	155	1,1
8	155	1,2
9	155	1,4
10	155	1,5
20	155	3,1

Sachant que notre thread qui met à jour les valeurs de l'ADC à une période de N ms, cela veut dire que pour voir les dernières données le plus souvent possible, la tâche d'affichage devrait elle aussi prendre une période de $N/2$ ms, autant que le frein. Une période plus petite créerait des doublons dans l'affichage et plus grande réduirait la vitesse à laquelle une nouvelle valeur apparaît sur notre affichage.

Voici un résultat possible obtenues pour un système ordonnançable :

Tâche	O_i	C_i	T_i	D_i	Priorité
BME	X	20	500	500	4
Freinage	X	16	67,5	67,5	1
ADC	X	20	270	270	2
Display	X	66	135	135	3

WCRT	BME	Freinage	ADC	Display
1	56	16	36	122
2	122	16	36	118
3	138	16	36	118
4	154	16	36	118
5	154	16	36	118

Vitesse	Temps réaction	Distance fait
0	411	0,0
1	411	0,4
2	411	0,8
3	411	1,2
4	411	1,6
5	411	2,1
6	411	2,5
7	411	2,9
8	411	3,3
9	411	3,7
10	411	4,1
20	411	8,2
30	411	12,3

Si on remplace le capteur de distance par un capteur ultrason, on multiplie par 874 030,5 le temps de réponse du capteur. Ainsi, on ne peut pas supposer que le temps de parcours du son pour le capteur est négligeable par rapport au pire temps de réactivité. Elle impact directement réactivité, donc la vitesse maximale atteignable sans endommager le robot.

En voici un tableau pour se faire une idée :

Vitesse(m/s)	Temps réaction(ms)	Distance fait sans compter capteur(m)	Délais dernier info son(ms)	Délais dernier info lumière(ms)
0	374	0,0	0,0	0,00000
1	374	0,4	2,2	0,00000
2	374	0,7	4,4	0,00000
3	374	1,1	6,5	0,00001
4	374	1,5	8,7	0,00001
5	374	1,9	10,9	0,00001
6	374	2,2	13,1	0,00001
7	374	2,6	15,2	0,00002
8	374	3,0	17,4	0,00002
9	374	3,4	19,6	0,00002
10	374	3,7	21,8	0,00002
20	374	7,5	43,6	0,00005

Mais ces résultats sont faits pour 343 m/s pour le son dans l'air. On peut multiplier cette vitesse par 4 si notre robot est un aquatique (environ 1500 m/s) et diviser la vitesse de la lumière par $\frac{4}{3}$ dans l'eau.

Cette idée fait juste poser que la solution de l'ultrason n'est pas forcément un mauvais. Cette idée peut-être plus viable de façon budgétaire ou technique en fonction de l'utilisation et de l'environnement d'activité. Par exemple, ce capteur ultrasons n'est pas du tout viable dans l'espace par manque d'un environnement sous pression.

Conclusion

Au final, il est préférable de prendre un capteur infrarouge à courte distance pour les robots terrestre pour optimiser le temps de réaction. Le temps de réactivité du frein va dépendre des autres tâches ajouter au système, car dans la liste de temps d'exécution ne sont que des exemples possibles si notre système de ne contient que ces tâches (ce qui peu probable). Donc, il peut varier en fonction des autres tâches pour que l'ensemble du système reste ordonnable.