TP 4
OS Temps Réel

Objectif

Dans un système temps réel, les contraintes sont souvent défini par les interfaces. Dans le cas que nous verrons dans ce TP, inspiré du projet PLATO, la périodicité d'acquisition des images ainsi que la régularité du débit sortant sont les contraintes.

À l'aide d'un OS temps réel, nous construiront donc un système qui réalise les traitements nécessaires tout en garantissant le respect des contraintes précédentes.

>> Vous produirez un rapport de TP contenant vos réalisations et observations. Les éléments indispensables à ce rapport sont signalés de cette manière.

Créer un projet RTEMS

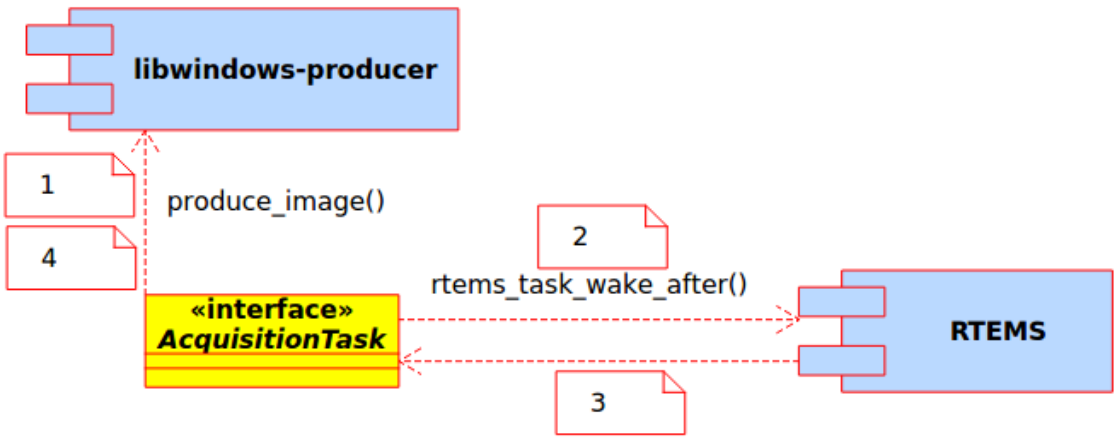
Récupérer le toolchain RCC 1.3 `sparc-rtems-5-gcc-10.2.0-1.3.0` sur `https://www.gaisler.com/anonftp/rcc/bin/` et le décompresser dans `opt/rcc-1.3.0-gcc`

Ajouter `opt/rcc-1.3.0-gcc/bin` au PATH et à la création du projet dans Eclipse, indiquer les informations correspondantes à ce toolchain dans l'écran « Cross GCC Command ». Ajouter également `-qbsp=gr712rc` aux options de compilation et linkage du projet.

Enfin, plutôt que le fichier c créée en même temps que le projet, utiliser `opt/rcc-1.3.0-gcc/src/samples/rtems-tasks.c`

Le programme démarre avec la fonction `Init()`. Celle-ci initialise le temps, crée puis démarre trois tâches ayant pour point d'entrée la fonction `Test_task()`. Enfin, la tâche ayant lancée la fonction Init() est effacée afin de forcer RTEMS à exécuter une autre tâche.

Développer une tâche d'acquisition



Créons une tache qui jouera le rôle d'un driver qui interrogerait le CCD pour acquérir une image par étoile, après un temps de pause de 500ms.

À la manière du TP précédents, intégrer la bibliothèque windows-producer à votre chaîne de compilation.
Rappel : clic droit sur le projet > Properties > C /C++ Build > Settings

Après avoir vérifié que le projet d'exemple compile bien, appeler `rtems_task_create()` avec des arguments définis tels que dans le cours et non des valeurs par défaut. Supprimer également les tâches 2 et 3. Modifier le point d'entrée de la tâche 1. Cette nouvelle fonction devra :

1. via windows-producer, initialiser le producteur d'images avec un tableau global pouvant enregistrer 100 imagettes de 36 pixels,
2. dans une boucle infini, appeler `produce_images()` puis attendre 500ms (cf. `rtems_task_wake_after()`).

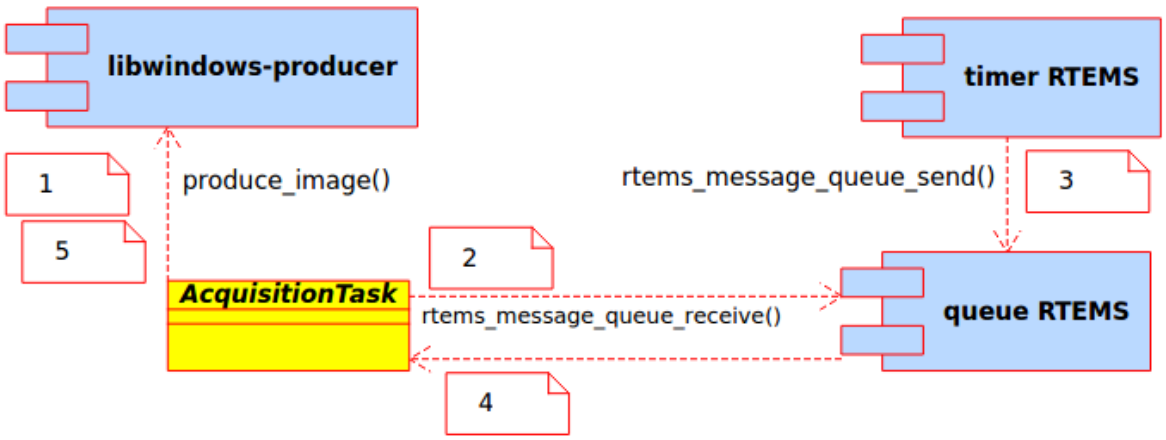
Pour que la tâche puisse utiliser des nombres flottant, la tâche doit être créée avec l'attribut `RTEMS_FLOATING_POINT`.

Au sein de la boucle ajouter l'appel à une fonction mesurant le temps entre deux appels. Celle-ci se basera sur la fonction `rtems_clock_get_ticks_since_boot()`. Pour connaître cette mesure, la passer en argument à une fonction examinée par un point d'arrêt.

>> Quel est le temps entre deux acquisitions?

Utiliser un timer et une queue de messages

Pour que la période d'acquisition ne soit pas dépendante du temps de réponse de la fonction `produce_images()`, remplaçons l'utilisation des pauses par l'utilisation d'un timer et d'une queue de messages.



Dans la fonction `Init()`, créer un timer (cf. `rtems_timer_create()`), l'armer (cf. `rtems_timer_fire_after()`) pour se déclencher au bout d'une demi-seconde et le lier à une nouvelle fonction. Dans celle-ci, réarmer le timer. La macro suivante est également nécessaires à l'utilisation des timers RTEMS :

```
#define CONFIGURE_MAXIMUM_TIMERS 5
```

Pour être prise en compte, les macros de configuration d'RTEMS doivent se trouver avant l'inclusion de `rtems/confdefs.h`

En plaçant un point d'arrêt, vérifier que le timer est bien appelé périodiquement.

Créons une queue de messages pour synchroniser l'acquisition sur le déclenchement du timer.

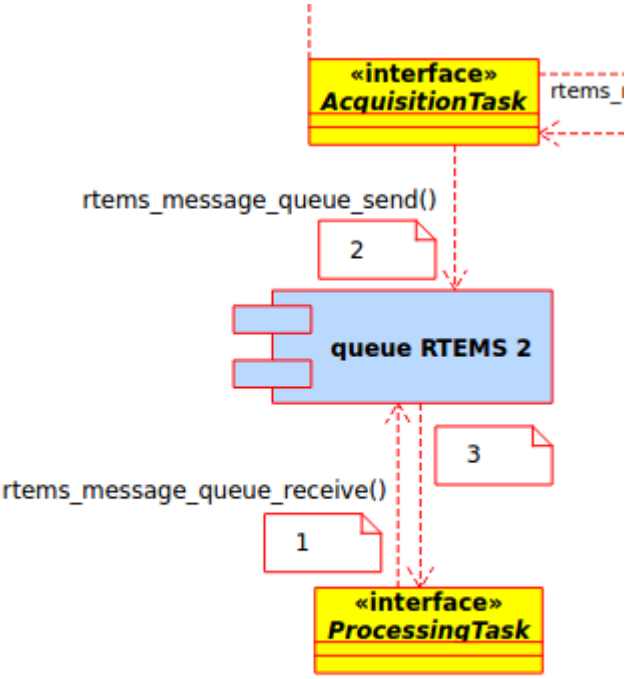
Dans `Init()` créer une queue de messages (cf. `rtems_message_queue_create()`). Dans la fonction appelée par le timer, poster un message dans la queue (`rtems_message_queue_send()`). Dans la tâche d'acquisition, commenter `rtems_task_wake_after()` et lire la queue de messages (cf. `rtems_message_queue_receive()`). La macro suivante est nécessaire à l'utilisation des queues de message RTEMS :

```
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES     10
```

>> Quel est le temps entre deux acquisitions? Comparez le résultat avec celui obtenu avant l'utilisation du timer.

Ajouter une tâche de traitement

Si l'acquisition doit être réalisée dans une tâche de haute priorité pour garantir sa périodicité, le traitement des données n'a pas la même contrainte. Nous réaliserons donc cette opération dans une autre tâche, de moindre priorité.



Créer cette nouvelle tâche dans `Init()`. Créer également une queue de messages qui permettra à la tâche d'acquisition d'indiquer que le buffer d'imagettes a été mis à jour.

Ajouter l'émission du message par la tâche d'acquisition et, au début de la boucle de la tâche de traitement, attendre la réception de ce message. Le message contiendra la valeur du compteur d'acquisitions.

Une fois le message reçu, calculer la photométrie par masques pondérés pour chacune des imagettes. Pour suivre l'évolution de la luminosité de chacune des étoiles, créer instance de la structure `flux` pour chacune d'elles et y sauvegarder la photométrie de 10 acquisitions successives.

```
#define FLUX_LENGTH 10

typedef struct flux{
        uint32_t id_window ;            ///< indice de l'imagette
        uint32_t id_first_acquisition ; ///< valeur du compteur d'acquisition pour measures[0]
        float measures[FLUX_LENGTH]
} flux
```

Transmettre des données entre tâches

Pour transmettre ces données vers une tâche gérant l'émission des télémétries (TM), nous allons poster ces structures dans un buffer circulaire.

Un buffer circulaire est un composant capable de stocker un certain nombre de données puis de les récupérer dans l'ordre où elles ont été ajouté, selon le principe First In First Out (FIFO). Le buffer n'a pas besoin d'être plein pour que des données y soit lues. Il n'a pas non plus besoin d'être vide pour que des données y soient écrites.

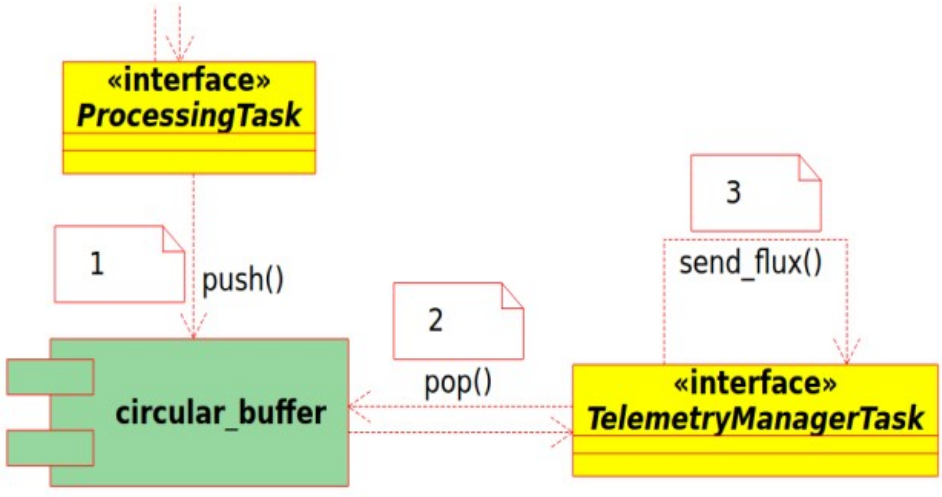
En prévision d'ajout de fonctionnalités d'accès concurrent, dans un fichier à part, encapsuler la structure `buffer_circulaire` fournis dans une autre structure : `buffer_circulaire_partagee`. Encapsuler les appels à `buffer_circulaire` dans de nouvelles méthodes qui utilisent `buffer_circulaire_partagee`.

Créer un `buffer_circulaire_partagee` global, capable de contenir 100 `struct flux`. Dans la fonction `Init()` initialiser le buffer circulaire et dans la tâche de traitement, alimenter le buffer circulaire avec les `struct flux`.

Ajouter une tâche de gestion de télémétrie

Création de la tâche, affichage de TM et mutex

Cette troisième tâche a pour rôle d'envoyer vers l’extérieur les données.



Sur PLATO il s'agit de construire des paquets télémétriques bien formés et de les transmettre à la plateforme pour qu'elle prépare leur émission. Ici nous appeler simplement `send_flux().`

```
/**
 * Simule l'émission d'une séquence de mesures de photométrie.
 * @param f structure enregistrant la séquence
 * @param time moment de l'émission, exprimé en ticks depuis le démarrage du programme .
 */
void send_flux(uint32_t time, flux f){
	static int cpt = 0 ;
	cpt++ ;
}
```

Créer un tâche de priorité plus basse que les 2 autres. Dans une boucle infini, lire le buffer circulaire et s'il n'est pas vide, passe la structure lue à la fonction `send_flux()`.

Pour afficher les résultats, intégrer le point d'arrêt suivant à votre script GDB.

```
break send_flux
commands
silent
	printf "%d \t %d \t %d ", time, f.id_first_acquisition, f.id_window
	printf "\t %f ", f.mesures[0]
	printf "\t %f ", f.mesures[1]
	printf "\t %f ", f.mesures[2]
	printf "\t %f ", f.mesures[3]
	printf "\t %f ", f.mesures[4]
	printf "\t %f ", f.mesures[5]
	printf "\t %f ", f.mesures[6]
	printf "\t %f ", f.mesures[7]
	printf "\t %f ", f.mesures[8]
	printf "\t %f\n", f.mesures[9]
cont
end
```

>> Quelle est la sortie de ce script au cours des 12 premières secondes ?

Pour que la tâche de télémétrie ne soit pas interrompu dans ses lectures de la FIFO par des écritures réalisées par la tâche de traitement, il est nécessaire de protéger cette ressource partagée des accès concurrents. L'outil le plus adapté est le sémaphore binaire, aussi appelé mutex.

Dans les fonctions encapsulant l'accès au buffer circulaire, conditionner les lectures et écritures à l'obtention d'un sémaphore binaire bloquant (RTEMS_WAIT, RTEMS_NO_TIMEOUT).

```
#define CONFIGURE_MAXIMUM_SEMAPHORES		3
```

La macro précédente est nécessaire à l'utilisation des sémaphores.

>> Quel est le code du `buffer_circulaire_partagee` ?

Régulation de débit

Pour ne pas engorger le système en aval, le débit des données doit être régulés. Ajoutons cette fonction à la tâche de gestion de télémétrie.

Définir deux constantes localement à la tâche. La première définissant l'intervalle de régulation, l'autre le nombre maximal de flux à émettre sur un slot de temps. Avec ces deux paramètres, l'on

indique que pas plus de 30 flux doivent être émis toutes les 100ms.

Ajouter deux variables locales. L'une pour conserver l'heure de début du slot (moment d'émission de la première donnée du slot), l'autre pour connaître le nombre de flux déjà émis depuis le début du slot.

Si 30 flux ont été émis depuis de début du slot. Mettre la tâche en pause jusqu'à la fin du slot via utiliser `rtems_task_wake_after()`. Quand un flux est à émettre et que la fin de slot est passé, remettre à 0 le nombre de flux émis et redéfinir le début de slot avec l'heure courante.

>> Quel est le code de la tâche de régulation de télémétrie ?

>> Quelle sont les données émises au cours des 12 premières secondes ?

>> Quel est le comportement du système si la tâche d'émission de TM est de plus haute priorité que la tâche de traitements ? Pourquoi ?