TP2 Free RTOS

Charger sur moodle les différents fichiers nécessaires

1 Créer le projet free RTOS :

Il faut structuer votre projet : créer un nouveau projet pour une cible LPC1769 depuis le répertoire démo/ARM7_LPC1769_KEIL_RDVS

Il faut en Keil5 inclure les fichiers de démarrage necessaire au processeur à la création du projet :

Attention à avoir des chemins complets cours et sans caractères étranges (espace, accents …)

Il faut utiliser software Packs et sélectionner le lpc1769.

Puis rajouter (cocher) CMSIS/core Device / Startup

Il faut maintenant rajouter tous les fichiers necessaires au projet...en les rangeant par thématique

(faire Add group (clic contextuel sur Target 1) pour créer les répertoires)

– dans un répertoire FREERTOS :

inclure dans le projet tous les fichiers C du répertoire source : list.c , queue.c, task.c

inclure du repertoire source/MemMang : heap2.c

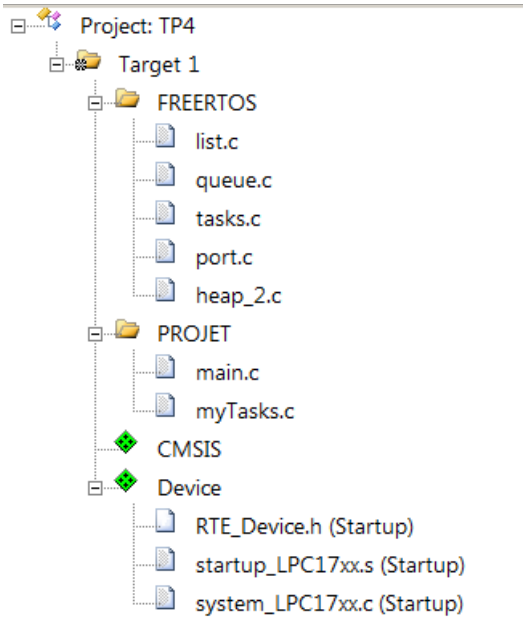
inclure du repertoire Source/portable/RVDS/ARM_CM3 : port.c et portmacro.c

– dans un repertoire Projet

inclure Demo/ARM7_LPC1769_Keil_RVDS : main.c myTasks.c

il faut bien sur déclarer le répertoire listing_output comme cible pour les fichiers générés

- les fichiers liés au projet keil LPC1769 apparaitront avec le manage run time environnement



Le premier objectif est de pouvoir compiler ce projet et de le tester simplement en rajoutant une seule tâche et de vérifier comme pour notre propre OS que le système démarre et passe bien la main à notre tâche.

Vous allez rencontrer 7 erreurs et aucun warning à priori (si vous êtes en compilateur version 5)

Les erreurs à la compilation vont être provoquées par l'absence de chemins déclarés pour les fichiers .h que l'on va rajouter dans Options for Target (ALT F7), on éditera l'onglet C/C++ ligne include Paths pour rajouter les 3 chemins manquants.

FreeRTOS.h est dans source/include

FreeRTOSConfig.h se trouve dans le même répertoire que votre main.c

Portmacro.h se trouve dans le même répertoire que port.c

Si vous voyez C/C++(AC6) comme titre, passez en compilateur version 5 (ALT F7 onglet Target) sous peine de ne pas arriver à compiler à la fin (67 erreurs).

La déclaration des tâches utilise l'API de FREERTOS , il faut donc garder ouvert en permanence le document FreeRTOS_Reference_Manual_V10 … et imiter les exemples qu'il donne pour chaque fonction

On suivra rapidement le démarrage de l'OS pour visualiser qu'il se passe les mêmes choses qu'avec le notre mis à part l'usage de listes chaînées.

2 Réalisation d'un système temps réel simple

Le but de ce TP est de réaliser un jouet , petit véhicule de type char commandé par boutons, un codeur incrémental et muni à l'avant d'une animation à base de leds à l'avant dont la vitesse dépend d'un potentiomètre.

Le jouet émet du son quand il recule à l'aide du DAC du micro-contrôleur dont le volume, la fréquence sera réglable à l'aide de deux potentiomètres scruptés périodiquement.

Vous disposez de code C exemples dans lequel prendre les initialisations des différents périphériques nécessaires et les fonctions d interruptions nécessaires.

Toutes ces initialisations devront être appelées depuis prvSetupHardware(void) du fichier main.c

2.1 Chenillard décoratif à l'avant du véhicule

Il sera muni à l'avant d'un chenillard de leds de type K2000 avec 5 leds P2.3 à P2.7 dont la vitesse de défilement devra être définie dans une variable val_attente.

la séquence sera la suivante si le véhicule roule, elle se joue en boucle

P2.7	P2.6	P2.5	P2.4	P2.3
ON	OFF	OFF	OFF	OFF
OFF	ON	OFF	OFF	OFF
OFF	OFF	ON	OFF	OFF
OFF	OFF	OFF	ON	OFF
OFF	OFF	OFF	OFF	ON
OFF	OFF	OFF	ON	OFF
OFF	OFF	ON	OFF	OFF
OFF	ON	OFF	OFF	OFF

On pourra implémenter cela via un tableau contenant les valeurs des divers états...

Pensez à utiliser FIOMASK pour pouvoir écrire sur le port2 en agissant seulement sur les bits qui concernent le chenillard. On pourra lire le manuel LPCxx_UM.pdf chapitre 9 page 139 pour comprendre son usage.

la séquence sera la suivante si le véhicule est à l'arrêt, elle se joue aussi en boucle

P2.7	P2.6	P2.5	P2.4	P2.3
ON	OFF	OFF	OFF	OFF
ON	ON	OFF	OFF	OFF
ON	ON	ON	OFF	OFF
ON	ON	ON	ON	OFF
OFF	ON	ON	ON	ON
OFF	OFF	ON	ON	ON
OFF	OFF	OFF	ON	ON
OFF	OFF	OFF	OFF	ON
OFF	OFF	OFF	ON	ON
OFF	OFF	ON	ON	ON
OFF	ON	ON	ON	ON
ON	ON	ON	ON	OFF
ON	ON	ON	OFF	OFF
ON	ON	OFF	OFF	OFF

On sait si le véhicule roule ou pas à l'aide de la variable globale vitesse_moyenne.

Le changement d'animation se fait uniquement dans les étapes 0 et 4 de la première animation et dans les étapes correspondantes dans l'animation 2 (0 et 8)

Chaque animation sera décrite par un tableau, dans lequel on se déplacera à l'aide d'un index

Pour simplifier, on peut doubler chaque état du tableau de l'animation 1 pour avoir deux tableaux de même taille et alors une seule tâche peut gérer les deux animation facilement.

- Cette première tâche met à jour les leds en lisant soit sequence1[index] soit sequence2[index] , avance d'un cran dans l'animation en cours (incrémentation de l'index) , se met en attente à l'aide de vTaskDelay fonction non bloquante, dont le paramêtre sera mis à jour par une autre tâche via une variable globale initialisée à une valeur permettant d'avoir 2 étapes de chenillard par seconde . Quand l'index atteindra les valeurs clefs de changement d'animation, il sera décidé pour les huit prochaines étapes quelle animation sera utilisée en testant la vitesse_moyenne modifiée par la seconde tâche.
- Cette seconde tâche scrutera 25 fois par seconde le bouton **P0.0** et le bouton **P0.1**

  Le premier bouton accélère en marche avant (et ralentit vite si on est en marche arrière)

  Le second bouton accélère en marche arrière (et ralentit vite si on est en marche avant)

  Si aucun bouton n'est enfoncé on ralentit lentement de 1 en 1, l'accélération se faisant de 5 en 5 ,

  la vitesse maximale étant de 150

  La vitesse sera stockée dans la variable vitesse_moyenne et le sens dans la variable marche_recul

Déclarer ces 2 taches dans void vInit_myTasks(UBaseType_t uxPriority) de myTasks.c

On s'inspirera des exemples du guide de freeRTOS pour la création de tache.

A chaque fois qu'on sera confronté à une notion x fois par seconde, on traduira cela par un endormissement de la tâche via un vTaskDelay....

Tester ce code simple pour voir si les deux tâches se lancent bien et si le chenillard fonctionne bien.

On pourra tricher sur le nombre de ticks par seconde pour raccourcir la simulation.

2.2 Accélération, Ralentissement, Rotation du jouet

On utilisera un codeur incrémental pour incrémenter et décrémenter une variable differentiel_vitesse qui sera bridée à +/- vitesse_max pour gérer la rotation. Le codeur utilise les entrées **P2.11** , **P2.12** et **P2.10** pour son bouton.

On le scrutera à 200HZ.

Conseil pour aller très vite : l'état des deux pattes forme un nombre binaire "codeur " entre 0 et 3

si on tourne dans un sens CW, on a la séquence 0 1 3 2 0 1 3 2.

Si on tourne dans l'autre sens CCW on a la séquence 0 2 3 1 0 2 3 1.

En mémorisant l'ancienne valeur "old_codeur" on sait donc si on tourne dans un sens où dans l'autre

Vous pouvez implémenter cela dans un switch (old_codeur) , et dans chaque cas savoir s'il faut :

- incrémenter (si on est pas à la dernière case du tableau) si codeur est le suivant en sens CW
- décrémenter (si on est pas à la première case du tableau) pour le suivant en sens CCW
- ne pas bouger si codeur vaut old_codeur (inutile de le tester)
- augmenter la vitesse de scrutation si on a sauté une valeur ou ne rien faire...

L'appui sur le bouton du codeur fera aller droit s'il existe un différentiel de vitesse, et arrêtera brutalement le robot s'il n'y a plus de différentiel de vitesse lors d'un second appui.

La tâche qui gèrera l'actualisation des PWM sera notifiée qu'une mise à jour est nécessaire à chaque modification soit de la valeur moyenne, soit de la vitesse différentielle.

Les consignes des PWMs s'obtiendront en ajoutant differentiel_vitesse à vitesse_moyenne pour une roue, et en soustrayant differentiel_vitesse à vitesse_moyenne pour l'autre roue.

Attention à tenir compte du sens de rotation de chaque roue : la roue sera à l'arrêt si le PWM est à 50 % , tournera dans un sens pour les consignes supérieures à 50 % et dans l'autre sens pour les consignes inférieurs à 50 %. on fera attention à rester dans la gamme 0 / 100 % pour chaque PWM.

On actualisera alors les pwm des roues en mettant à jour les registres PWM1MR1 et PWM1MR2 sans oublier d'activer PMW1LER pour la mise à jour des PWMS.

2.3 Gestion des capteurs analogiques :

Récupérer dans le fichier necessaire_ADC.C la routine d'initialisation et le handler de fin de conversion

Créer une tache gere_conversion qui va lancer périodiquement la conversion 50 fois par seconde(pour savoir comment faire, voir le commentaire final de init_adc()). On commutera la sortie P0.9 à chaque lancement de conversion.

Toutes les fins de conversion (dans l'interruption ADC) on pourra actualiser la vitesse du chenillard en mettant à jour la variable qui définit le delay dans la tâche qui gère les chenillards de manière à avoir une vitesse de défilement de 1 à 6hz pour les valeurs converties. On commutera la sortie P0.10 à chaque fin de conversion.

On notifiera une tâche qui calculera aussi un buffer rampe_volume[256] pour rejoindre le nouveau volume sonore (en 256 étapes) et un buffer rampe_vitesse[256] pour rejoindre la nouvelle vitesse en 256 étapes (valeur exploitée par la fonction calcul_tableau disponible dans ADC.C). On commutera la sortie P0.11 pour toute la durée du traitement de cette tâche.

Quand ces buffers seront prêts, on signalera via une variable globale qu'on peut les utiliser.

la valeur convertie sur 12 bits sera traduite pour la vitesse de lecture sur une gamme allant de 64 à 1024 (vitesse de lecture divisée par 4 à vitesse de lecture multipliée par 4)

la valeur convertie sur 12 bits pour le volume sonore sera exploitée pleine echelle pour un volume de 0 à 1

2.4 Gestion du son par DAC et interruption timer

Le Son devra être généré par le DAC et une interruption timer dans laquelle on mettra le DAC à jour (le DAC n'est pas capable de générer une interruption). Malheureusement écrire dans le DAC affecte directement la sortie Son à l'instant d'écriture, il faut donc être extrèmement précis sur l'instant de sortie de l'échantillon de son, sans retard tolérable sous peine de déformer le son... Il est donc impossible d'être compatible OS.

De temps en temps, tous les 256 échantillons, à chaque changement de buffer audio, on va devoir prévenir pourtant l'OS qu'il faut calculer un nouveau buffer son.

Dans un premier temps utiliser la fonction de notification de l'OS pour comprendre les dégats ...on augmentera si besoin la demande à tous les 8 échantillons pour augmenter la probabilité d'interrompre l'OS à un mauvais moment et voir l'OS se planter...

On ne pourra donc pas utiliser directement l'API de l'OS, on utilisera une astuce pour communiquer avec une tâche compatible OS : on lancera un timer qui débordera très vite (1us, le temps de sortir d'interruption incompatible OS par exemple) et déclenchera une interruption compatible OS qui sera traitée au plus vite par l'OS. Cette interruption compatible OS notifiera une tâche pour recalculer le buffer audio.

L'idée est de calculer le son avec une vitesse variable et un volume variable. vous pourrez vous inspirer de l'exemple donné dans necessaire_DAC.c

Pour calculer le volume sonore, il faudra faire le calcul sans nombre à virgule en restant positif : l'ecart à 512 est amplifié en le multipliant par la valeur de 0 à 4095 puis divisé par 4096 (décalage de 12bits) pour être alors recentré à 512.

Pour ajuster la fréquence et le volume, soit on utilisera les buffers de rampe, soit on utilisera la dernière valeur du buffer de rampe si on a plus de changement. On pensera à aquitter l'indication de disponibilité.

Cette solution avec les buffers de rampe est imparfaite, vous expliquerez ce qui se passe si on change le volume lentement sur une seconde, et vous proposerez votre solution pour améliorer.

Pour l'instant le son est joué tout le temps, on pensera à rajouter les éléments permettant au passage à zéro de décider si on joue le son ou pas selon si le véhicule se déplace en marche avant ou marche arrière.

ANNEXE : programmation en Free Rtos

relire "FreeRTOS_Reference_Manuel_V10.0.0.pdf" disponible sur Moodle section "references".

pour créer une nouvelle tâche : regarder l exemple page 37

xTaskCreate (nom_fonction , "NOM", taille_pile, adresse_parametres, priorite, pointeur_TCB)

- le premier champ est un pointeur de fonction qui contient l'adresse d'une fonction donc passer le nom de la fonction en paramètre revient à passer l'adresse où la fonction est implémentée par le compilateur.
- pointeur_TCB sera mis à jour avec l'adresse mémoire du TCB contrôlant la tâche nouvellement créée. Si on a pas besoin de savoir où est stockée les informations de la tâche , on passe NULL en paramètres.

BIEN REGARDER LES EXEMPLES POUR NE PAS SE TROMPER SUR LES TYPES DE PARAMETRES.

Pour toutes les fonctions ci dessous , la documentation donne un exemple simple d utilisation après la description de la fonction.

pour endormir une tâche un certain temps: vTaskDelay()

Pour mettre la tâche en pause : vTaskSuspend()

Pour notifier une tâche : xTaskNotify() ou vTaskNotifyGive()

Pour notifier une tâche depuis une interruption: xTaskNotifyFromISR() ou vTaskNotifyGiveFromISR()

Pour se mettre en attente de notification: ulTaskNotifyTake() ou xTaskNotifyWait()