INF3610 –Systèmes embarqués

Hiver 2020

Questions TP No. 2

Groupe 1

[1928777] – Mariam Sarwat

[1935516] – Louis-Maxime Bois

Soumis à : Elisabeth Pham

02/04/20

Question 1 :

On doit d'abord connecter les composantes matérielles et mettre en place les structures correctement avant de gérer quoi que ce soit.

asm_vector.r

On initialise la structure _vector_table (Interruption Vector Table a.k.a. Exception Table) qui garde en mémoire comment gérer différentes interruptions logicielles et matérielles. Puisqu'un mot est d'une longueur de 2 bytes et qu'il y a 9 mots dans la structure, nous savons que nous somme en 0x18.

os_cpu_a_vfp-d32.r

La fonction OSIntCtxSw sert à rediriger l'attention du processeur utilisé par vers la prochaine tâche la plus prioritaire à la fin de la gestion d'une interruption. Celle-ci appellera OSTaskSwHook(), puis procèdera à la recherche de la prochaine tâche la plus importante. Pour la recherche de la prochaine tâche, on commence par prendre connaissance de l'état actuelle de la table de tâches prêtes, puis on vérifie si la tâche la plus prioritaire est prête. Si oui, on récupère le stack pointer de la prochaine tâche à partir de sa OS_TCB, puis on restaure les registres de la tâche dans le processeur à partir du stack pointer.

Nous avons la fonction OS_CPU_ARM_ExceptIrqHndler, qui identifie une exception entrante comme OS_CPU_ARM_EXCEPT_IRQ. Ce qui permettra à OS_CPU_ARM_ExceptHndlr de reconnaitre que l'interruption vient de l'IRQ et non pas autre chose.

OS_CPU_ARM_ExceptHndlr a pour but de mettre le CPU en ''mode IRQ''. On commence par sauvegarder tous les registres, puis on impose les registres reliés à l'interruption à gérer. Les lignes 446 à 450 sont l'équivalent de OSIntNesting++;.

OS_CPU_ARM_ExceptHndlr_BreakTask est la fonction appelé juste à la suite de la fonction ci-haut. Les lignes 466 à 469 sont l'équivalent de OSTCBCurPtr->StkPtr = SP; et la ligne 477 réfère à la fonction du même nom décrite dans ucos_int_impl.c. La ligne 485 réfère à la fonction du même nom définit dans os_core.c. Ces deux dernières fonctions ont déjà été expliquées plus haut. Les lignes 487 à 490 servent à restaurer les registres de la tâche dictée par OSIntExit.

ucos_int_impl.c

La fonction UCOS_IntVectSet() est appelé par connect_fit_timer_0. La ligne 343 met la fonction de gestion de l'interruption dans le tableau de gestion d'interruptions. Ce tableau servira à

dicter le comportement d'une interruption quand elle surviendra (table des handlers). La ligne 344 réfère à une table qui garde en mémoire l'ID de chaque interruption (table des ID). On désactive les interruptions juste avant ces lignes pour empêcher l'ISR de prendre le contrôle du processeur pendant la manipulation des tableaux d'interruptions. On empêche ainsi la corruption de ce tableau qui dicte les comportements désirés pendant des interruptions.

La fonction UCOS_IntHandler est définie. Cette fonction interprète une interruption en en extractant la source, puis en redirigeant l'ISR vers le bon handler dans la table des handlers. Les lignes 394 à 399 servent justement à retrouver le bon identificateur d'interruption et à en faire un objet p_isr, qui sera ensuite pris en charge par l'ISR.

On retrouve la définition de OS_CPU_ExceptHndler() qui sert à parcourir le Interruption Vector Table et déterminer le type d'exception en cause du signal. On voit ici (434 à 457) que ce système ne soutient tout simplement pas autre chose que les signaux de l'IRQ ou du FIQ. Ces deux types d'interruption ont comme différence leur priorité, où le FIQ produit des interruptions plus prioritaire que l'IRQ. On peut les identifier par un ID différent qui sera assigné par OS_CPU_ARM_ExceptIrqHndlr.

xintc_l.c

XIntc_DeviceInterruptHandler() est le gestionnaire principal des interruption dans le driver. Il sert tout simplement à accueillir directement toutes les interruptions, de déterminer la prochaine interruption à gérer et que toutes les interruptions sont gérées. Les lignes 220 et 221 servent à s'assurer de gérer chaque interruption qui ont survenue avec une boucle. Il peut y avoir plus qu'une interruption active en même temps, mais il faut les gérer une par une. On parcourt CfgPtr afin de tout couvrir. Les lignes 254 et 255 servent à l'appel du handler en charge de la prochaine interruption.

bsp_init.c

La fonction connect_fit timer_0 se chargera de faire le lien entre l'axi et le GIC. Dans ce même fichier, au lignes 84 à 88, on fournit l'ID et le handler de l'interruption à gérer pour remplir l'Interruption Table Vector.

os_core.c

On retrouve la définition de la fonction OSIntExit(). Elle effectue la transition entre la fin de la gestion d'une interruption dans monISR() et la prise en charge d'une tâche prioritaire au niveau des registres du CPU. Elle s'occupe de désactiver les interruptions, puis de décrémenter la valeur

de OSIntNesting, ce qui signifie la fin de la gestion de cette interruption. On s'assure ensuite qu'il n'y a pas d'autres interruptions à gérer (si OSIntNesting == 0) et si oui, on restaurera les registres de la prochaine tâche prête la plus prioritaire. Si non, on gère la prochaine interruption en ligne. La fonction appellera OS_TRACE_ISR_EXIT(), CPU_INT_EN(), OS_TaskStkRedzoneChk(), OSRedzoneHitHook(), OS_PrioGetHighest(), OS_TRACE_ISR_EXIT(), OS_TRACE_TASK_SWITCHED_IN(), OS_TLS_TaskSw() et OS_TRACE_ISR_EXIT_TO_SCHEDULER().

simAvion.c

Ici se trouve les routines d'exécutions des deux timers du TP ainsi que le gpio_isr. On y utilise OSTimeDlyHMSM() afin d'avoir un OSSemPost() de la sémaphore en lien avec le timer avec le paramètre OS_OPT_TIME_HMSM_STRICT afin d'avoir un signal à prendre à chaque seconde.

Déroulement général

Le fit_timer_0 produit l'interruption qui passera par l'axi_intc_0. L'axi_intc_0 gère toutes les autres interruptions matérielles qui ne sont pas des FIQ. Celle-ci relayera cette interruption au GIC, qui en fait un paquet qui regroupe toutes les interruptions que le GIC a reçu. Ce paquet ira vers la puce, ou plus précisément l'adresse 0x18 de la table des exceptions. Cette adresse, comme vu dans l'asm_vector.r, est relié à la gestion des interruptions de type IRQ, soit l'IRQHandler. On arrête le déroulement de la tâche en cours. On appelle ensuite OS_CPU_ARM_ExceptHndler qui va sauvegarder les registres courant du CPU, puis incrémenter le OSIntNesting et ainsi, indiquer l'arrivée d'une nouvelle interruption. OS_CPU_ARM_ExceptIrqHndlr identifie l'interruption comme OS_CPU_ARM_EXCEPT_IRQ, puis UCOS_IntHandler() recoit l'interruption. On envoie ensuite le tout à XIntc_DeviceInterruptHandler() qui détermine l'origine de l'interruption qui en l'occurrence, est le fit_timer_1s_isr. Sachant ceci, on gère l'interruption comme dicté dans simAvion.c, puis on rétablie les registres de la prochaine tâche courante avec OSIntExit().

Question 2 :

Afin de déterminer ce délai d'attente, il sera intéressant de prendre avantage du paramètre &ts de la fonction OSSemPend. En effet, lors d'un post d'un sémaphore, OS-III lit un timestamp et envoie cette valeur au OSSemPend(). Ceci nous permettra de connaitre quand le post est arrivé. Dans notre cas, le sémaphore sera le semStatistique. Par la suite, dans la fonction statistique, juste après le OSSemPend(&semStatistique, …), OS_TS_GET() pourrait être appeler pour trouver le timestamp actuel. Finalement on retrouve la différence entre la valeur retourner par OS_TS_GET et la variable ts.