ENSEIRB-MATMECA

MISE EN ŒUVRE DU NOYAU TEMPS REEL µC/OS II SUR PROCESSEUR BLACKFIN

Patrice KADIONIK
http://kadionik.vvv.enseirb-matmeca.fr/

TABLE DES MATIERES

1. BUT DES TRAVAUX PRATIQUES

Le but de ces Travaux Pratiques est d'étudier la mise en œuvre du noyau Temps Réel µC/OS II sur une carte d'évaluation Analog Devices Blackfin BF537. Ces TP ont été créés en 1999 initialement avec une carte maison à base du microcontrôleur Motorola 68HC11. Ils ont été portés sur la carte Blackfin en 2013 afin de pouvoir entre autre travailler avec le *bootloader* u-boot et être compatibles avec l'environnement VirtualBox.

La mise en œuvre de µC/OS II sur la carte Blackfin a fait l'objet d'un sujet de « projets avancés » de l'option Systèmes Embarqués SE. Je tiens ainsi à remercier Frédéric Kozlowski, Matthieu Juan, Daniel Joffe et Tamara Guilbert de la promotion SE 2012-2013 pour leur travail et leur contribution à l'amélioration constante de l'enseignement de l'option SE…

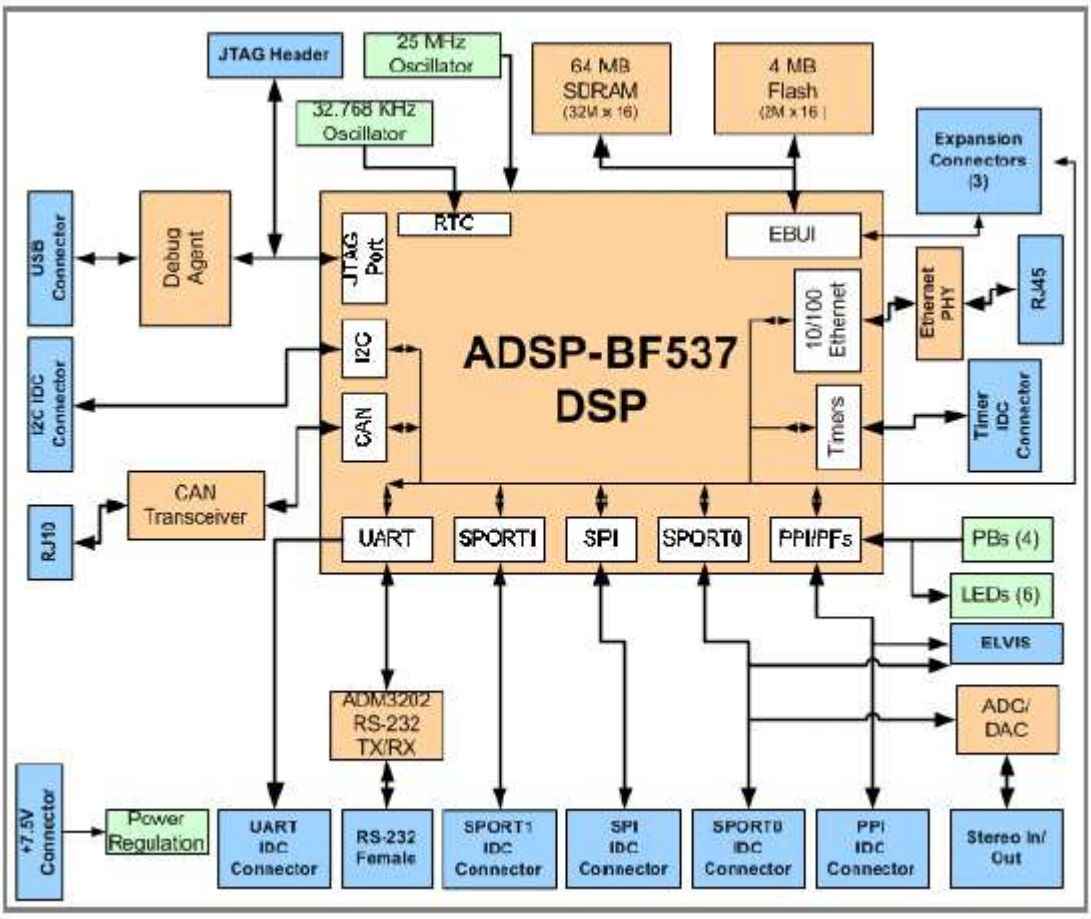
A tout moment, on se référera à l'aide contenue en annexe dans ce manuel ou bien à l'aide en ligne :

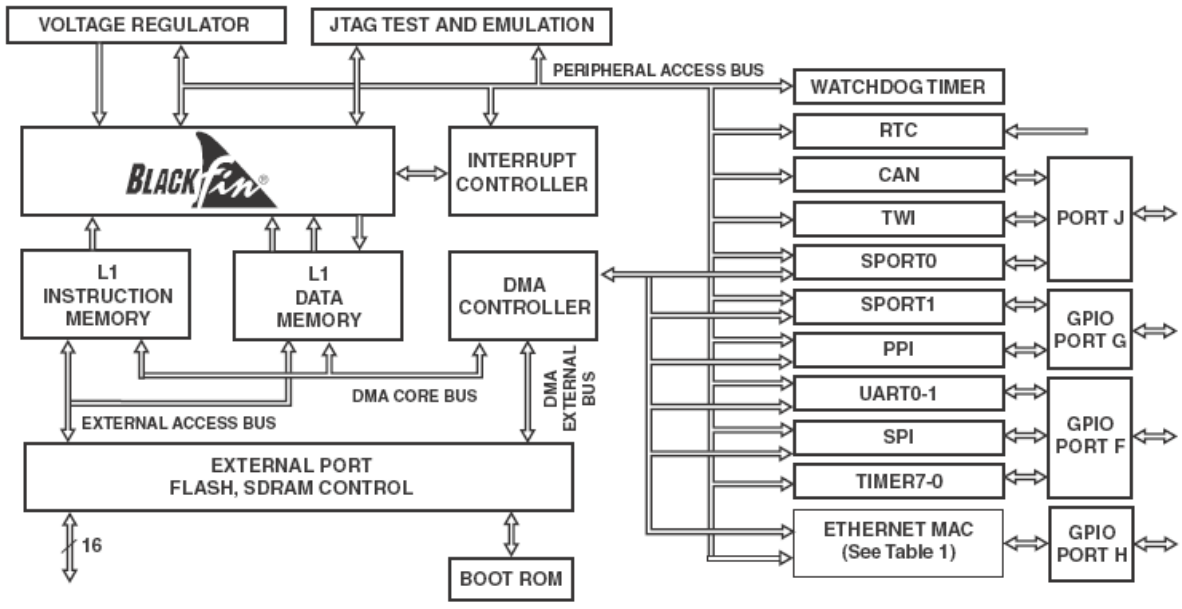
```
% man …
```

2. INFORMATIONS ESSENTIELLES SUR LA CARTE BLACKFIN BF537 EZ-KIT LITE

La carte cible Blackfin BF537 EZ-KIT Lite est une carte d'évaluation d'Analog Devices permettant de tester le processeur de traitement du signal Blackfin. Le processeur Blackfin supporte µC/OS II que nous allons utiliser durant ces TP.

L'architecture du processeur Blackfin BF537 est donnée sur la figure suivante :





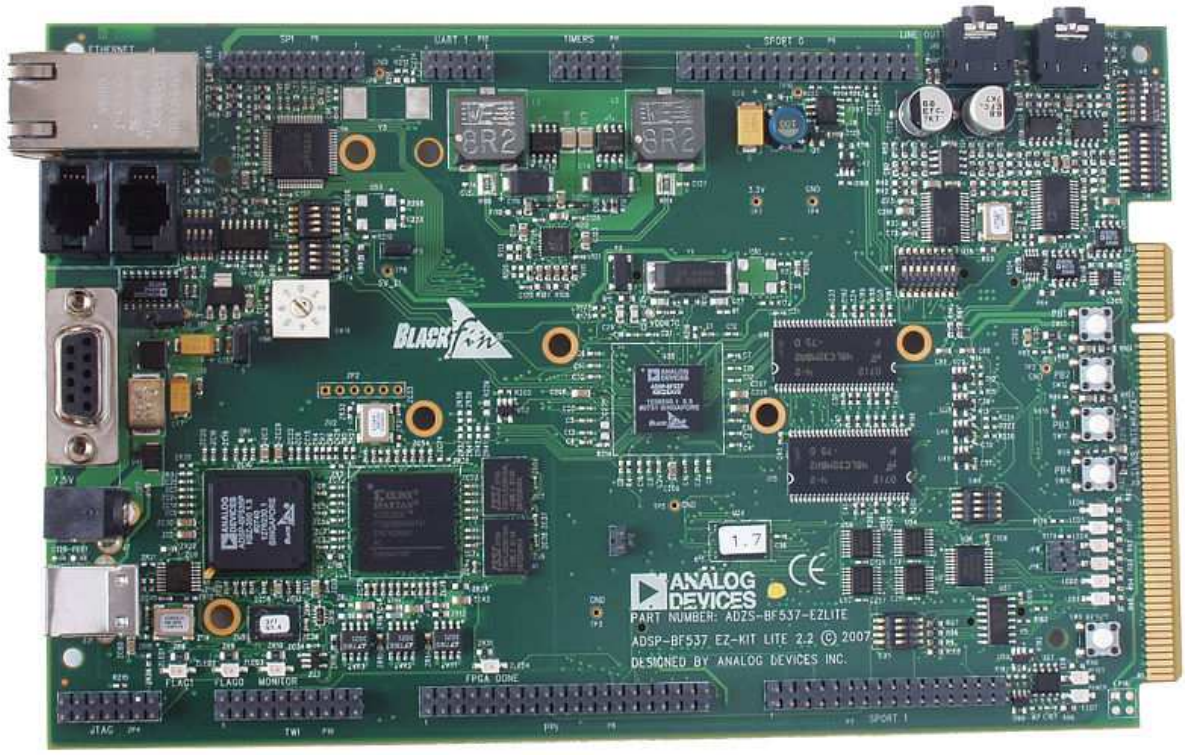
Architecture du processeur Blackfin BF537

La carte cible possède les fonctionnalités suivantes :

- Analog Devices ADSP-BF537 Blackfin processor
  - Core performance up to 600 MHz
  - External bus performance to 133 MHz
  - 182-pin mini-BGA package
  - 25 MHz crystal
- Synchronous dynamic random access memory (SDRAM)
  - MT48LC32M8 – 64 MB (8M x 8-bits x 4 banks) x 2 chips
- Flash memory
  - 4 MB (2M x 16-bits)
- Analog audio interface
  - AD1871 96 kHz analog-to-digital codec (ADC)
  - AD1854 96 kHz digital-to-audio codec (DAC)
  - 1 input stereo jack
  - 1 output stereo jack
- Ethernet interface
  - 10-BaseT (10 Mbits/sec) and 100-BaseT (100 Mbits/sec) Ethernet Media Access Controller (MAC)
  - SMSC LAN83C185 device
- Controller Area Network (CAN) interface
  - Philips TJA1041 high-speed CAN transceiver
- Universal asynchronous receiver/transmitter (UART)
  - ADM3202 RS-232 line driver/receiver
  - DB9 female connector
- Leds
  - 10 leds: 1 power (green), 1 board reset (red), 1 USB (red)
  - 6 general-purpose (amber), and 1 USB monitor (amber)
- Push buttons
  - 5 push buttons: 1 reset, 4 programmable flags with debounce logic
- Expansion interface
  - All processor signals
- Other features
  - JTAG ICE 14-pin header

La carte cible possède donc :

- 4 Mo de mémoire Flash.
- 64 Mo de RAM SDRAM.



Carte cible Blackfin BF537 EZ-KIT Lite

Le mapping mémoire de la carte cible est le suivant :

Start Address		End Address	Content
External Memory	0x0000 0000	0x03FF FFFF	SDRAM bank 0 (SDRAM). See "SDRAM Interface" on page 1-9.
	0x2000 0000	0x200F FFFF	ASYNC memory bank 0. See "Flash Memory" on page 1-10.
	0x2010 0000	0x201F FFFF	ASYNC memory bank 1. See "Flash Memory" on page 1-10.
	0x2020 0000	0x202F FFFF	ASYNC memory bank 2. See "Flash Memory" on page 1-10.
	0x2030 0000	0x203F FFFF	ASYNC memory bank 3. See "Flash Memory" on page 1-10.
	0x203F 0000		MAC address
	All other locations		Not used
Internal Memory	0xFF80 0000	0xFF80 3FFF	Data bank A SRAM 16 KB
	0xFF80 4000	0xFF80 7FFF	Data bank A SRAM/CACHE 16 KB
	0xFF90 0000	0xFF90 7FFF	Data bank B SRAM 16 KB
	0xFF90 4000	0xFF90 7FFF	Data bank B SRAM/CACHE 16 KB
	0xFFA0 0000	0xFFA0 7FFF	Instruction bank A SRAM 32 KB
	0xFFA1 0000	0xFFA1 3FFF	Instruction bank B SRAM 16 KB
	0xFFA0 8000	0xFFA0 BFFF	Instruction SRAM/CACHE 16 KB
	0xFFB0 0000	0xFFB0 0FFF	Scratch pad SRAM 4 KB
	0xFFC0 0000	0xFFDF FFFF	System MMRs 2 MB
	0xFFE0 0000	0xFFFF FFFF	Core MMRs 2 MB
	All other locations		Reserved

Mapping mémoire de la carte cible Blackfin

On notera que :
- La mémoire RAM va de $0000 0000 à $03FF FFFF.
- La mémoire Flash contenant u-boot va de $2000 0000 à $203F FFFF.

Le processeur Blackfin BF537 a 48 signaux GPIO répartis sur 3 ports de contrôle PF, PG et PH.
On retrouve entre autres les leds et boutons poussoirs de la carte cible connectés sur ces GPIO comme l'indique pour partie la figure suivante.

Processor Pin	Other Processor Function	EZ-KIT Lite Function
PF0	GPIO/DMAR0	UART0 transmit
PF1	GPIO/DMAR1	UART0 receive
PF2	UART1_TX/TMR7	Push button (SW13). See "Programmable Flag Push Buttons (SW10–13)" on page 2-19.
PF3	UART1_RX/TMR6/TACI6	Push button (SW12). See "Programmable Flag Push Buttons (SW10–13)" on page 2-19.
PF4	TMR5/SPI_SSEL6	Push button (SW11). See "Programmable Flag Push Buttons (SW10–13)" on page 2-19.
PF5	TMR4/SPI_SSEL5	Push button (SW10). See "Programmable Flag Push Buttons (SW10–13)" on page 2-19.
PF6	TMR3/SPI_SSEL4	LED (LED1). See "LEDs and Push Buttons" on page 1-14 and "Push Button Enable Switch (SW5)" on page 2-11 for information on how to disable the push button.
PF7	TMR2/PPI_FS3	LED (LED2). See "LEDs and Push Buttons" on page 1-14 and "Push Button Enable Switch (SW5)" on page 2-11 for information on how to disable the push button.
PF8	TMR1/PPI_FS2	LED (LED3). See "LEDs and Push Buttons" on page 1-14 and "Push Button Enable Switch (SW5)" on page 2-11 for information on how to disable the push button.
PF9	TMR0/PPI_FS1	LED (LED4). See "LEDs and Push Buttons" on page 1-14 for information on how to disable the push button.
PF10	SPI_SSEL1	LED (LED5). See "LEDs and Push Buttons" on page 1-14 for information on how to disable the push button.
PF11	SPI_MOSI	LED (LED6). See "LEDs and Push Buttons" on page 1-14 for information on how to disable the push button.
PF12	SPI_MISO	Audio reset
PF13	SPI_SCK	CAN ERR

Ports et GPIO (pour partie) du processeur Blackfin BF537

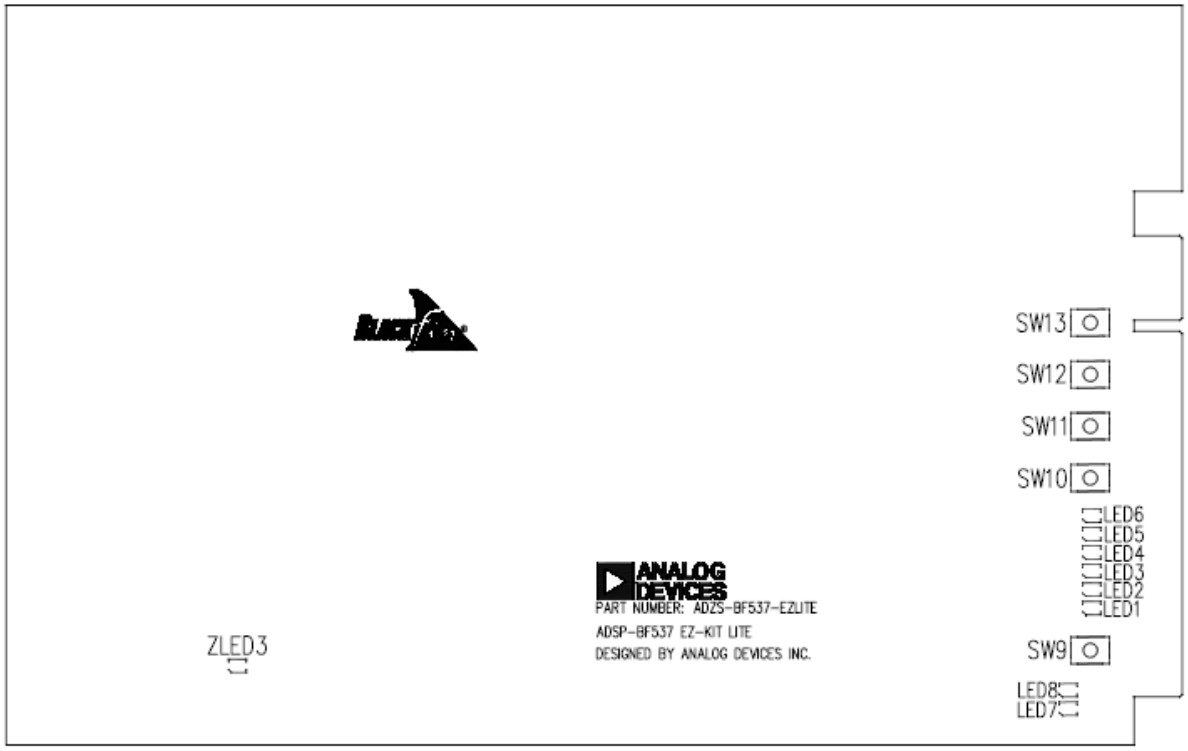
Pour pouvoir faire les TP pilotant les leds de la carte cible, il faut savoir spécifiquement sur quel port elles sont connectées et comment contrôler le port depuis les registres de contrôle et de données du processeur Blackfin.

Les 6 leds LED1 à LED6 sont connectées à 6 GPIO du port F comme suit :

LED Reference Designator	Processor Programmable Flag Pin
LED1	PF6
LED2	PF7
LED3	PF8
LED4	PF9
LED5	PF10
LED6	PF11

6 leds utilisateur de la carte cible

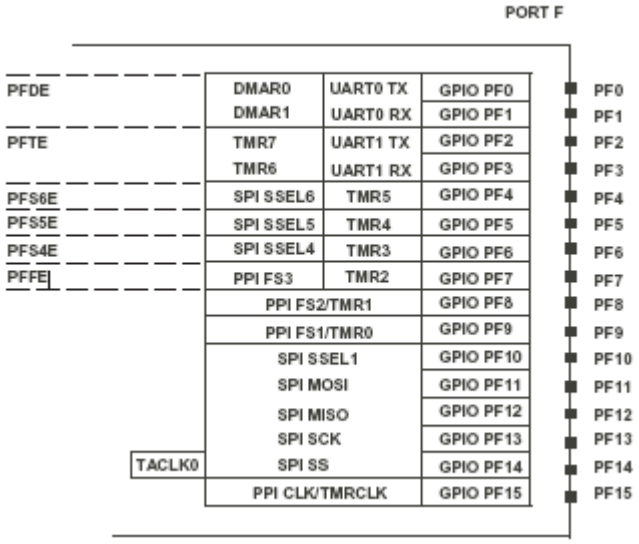
Les 6 leds sont situées sur la carte cible comme suit :



Position des 6 leds utilisateur sur la carte cible

La led est allumée quand on écrit 1 dans le bit correspondant du registre de données du port F.

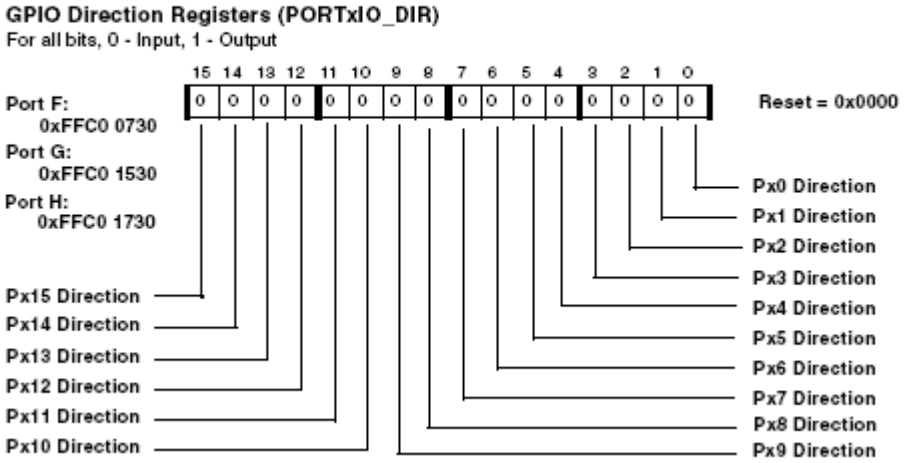
Le port F est agencé comme suit :



Fonctionnalités du port F du processeur Blackfin BF537

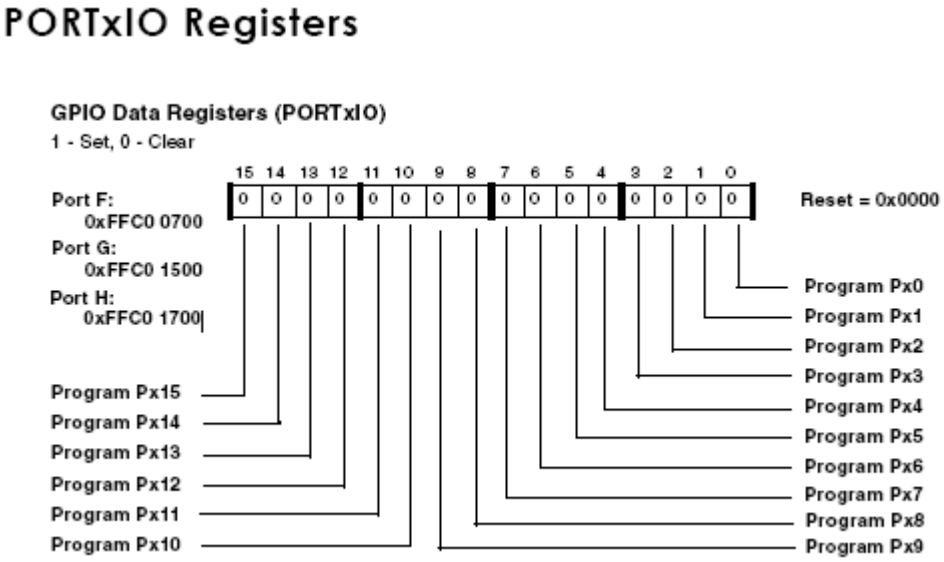
Le registre 16 bits de direction du port F se trouve à l'adresse $FFC0 0730 comme suit :

PORTxIO_DIR Registers



Registre 16 bits de direction du port F du processeur Blackfin BF537

Le registre 16 bits de données 16 bits du port F se trouve à l'adresse $FFC0 0700 comme suit :



Registre 16 bits de données du port F du processeur Blackfin BF537

On pourra par exemple contrôler les leds depuis u-boot. Il faut positionner PF6 à PF11 en sortie via le registre 16 bits de direction du port F puis écrire 1 ou 0 dans le registre 16 bits de données du port F :

```
bfin> mw.w FFC00730 ffff tout en sortie
bfin> mw.w FFC00700 0000 éteint les 6 leds
bfin> mw.w FFC00700 0040 allume la led 1
bfin> mw.w FFC00700 0080
bfin> mw.w FFC00700 0100
bfin> mw.w FFC00700 0200
bfin> mw.w FFC00700 0400
bfin> mw.w FFC00700 0800 allume la led 6
```

Afin de contrôler les leds de la carte Blackfin avec µC/OS II, on utilisera les fonctions développées lors du portage de µC/OS II pour la carte Blackfin et intégrées dans le BSP (*Board Support Package*) décrit ci-après.

3. NOYAU TEMPS REEL µC/OS II

3.1. Notion de Temps Réel

Pour expliquer brièvement le rôle d'un système multitâche Temps Réel, nous allons présenter le noyau µC/OS-II conçu et mis au point par J. Labrosse.
µC/OS II est un noyau Temps Réel qui permet d'effectuer une exécution de plusieurs tâches sur un processeur.
On peut trouver sur le site Internet de µC/OS II les différentes versions de µC/OS II portées sur les différents processeurs.

Ce noyau multitâche est fourni gratuitement sur le site Internet sans *royalties* à des fins d'enseignement et un livre manuel écrit par l'auteur est disponible.

La notion de Temps Réel correspond à la façon dont les tâches sont exécutées dans le temps : le temps d'exécution des tâches étant déterminant pour la commutation des tâches, le noyau Temps Réel exécute en premier toujours la tâche prête de plus forte priorité dont le temps d'exécution est critique.

On peut alors connaître à priori le temps d'exécution de telles tâches.

« Etre Temps Réel, c'est pouvoir ainsi traiter une information dans un temps maximum garanti pour que l'information ait toujours un sens ».

Le fonctionnement du noyau préemptif est illustré à la figure suivante :

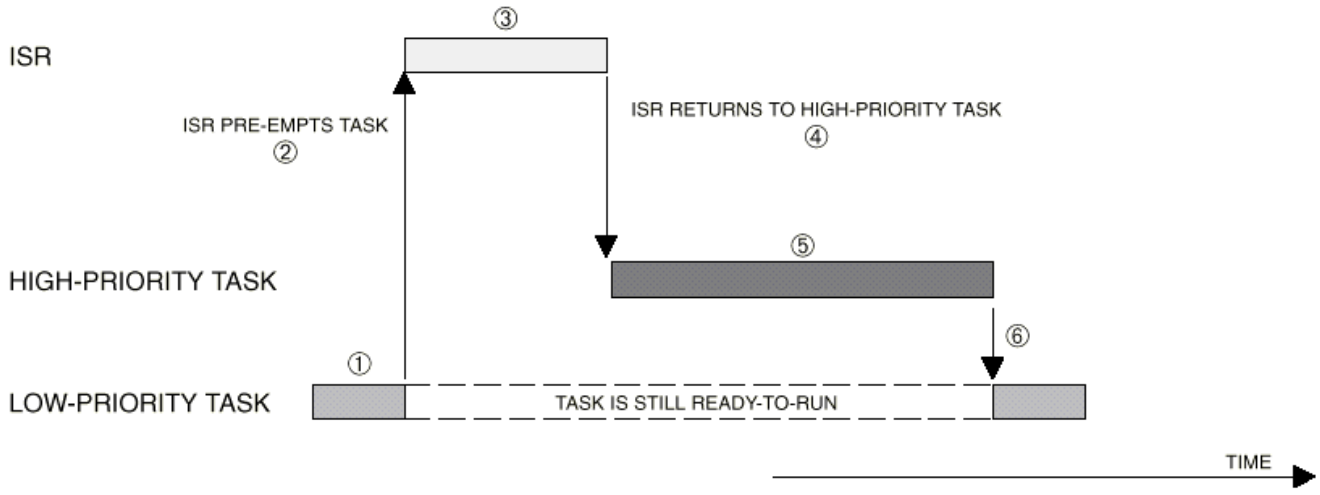


Illustration d'un noyau préemptif

Comme nous le voyons, une tâche de faible priorité est exécutée en (1) alors qu'une interruption asynchrone intervient en (2). Le microprocesseur saute donc au vecteur d'interruption exécutant alors la routine d'interruption qui va faire passer à l'état prêt une tâche de plus forte priorité que la tâche précédente (3) par l'intermédiaire d'une routine du noyau

(libération d'un sémaphore, envoi d'un message, réveil d'une tâche…). A la fin de la routine d'interruption, l'ISR invoque une primitive du noyau (4) qui active l'ordonnanceur décidant de donner la main à la tâche de plus forte priorité prête et non à la tâche précédente de faible priorité. A la fin de l'exécution de la tâche de plus forte priorité, le noyau reprend l'exécution de la tâche de plus faible priorité qui est prête depuis son interruption (6).
Comme on peut le voir, le noyau garantit que les tâches dont le temps d'exécution est critique sont effectuées en premier.

Un noyau Temps Réel assure principalement 2 fonctionnalités :
- L'ordonnancement.
- Le changement de contexte.

L'ordonnancement permet de déterminer quelle est la tâche de plus forte priorité prête qui aura accès ensuite au processeur. Quand une tâche de plus forte priorité doit être exécutée, le noyau doit sauvegarder toutes les informations nécessaires de la tâche en cours d'exécution afin de permettre son éventuelle continuation. Les informations sauvegardées (appelées contexte de la tâche) sont généralement les registres et le pointeur de pile de la tâche qui va être préemptée.
Lors de l'exécution de nouvelle tâche prête de plus forte priorité, le contexte de la nouvelle tâche est chargé puis l'exécution reprend là où elle s'était interrompue.
Cette opération de sauvegarde/restauration de contexte correspond à l'opération de changement de contexte.

Le changement de contexte est illustré à la figure suivante :

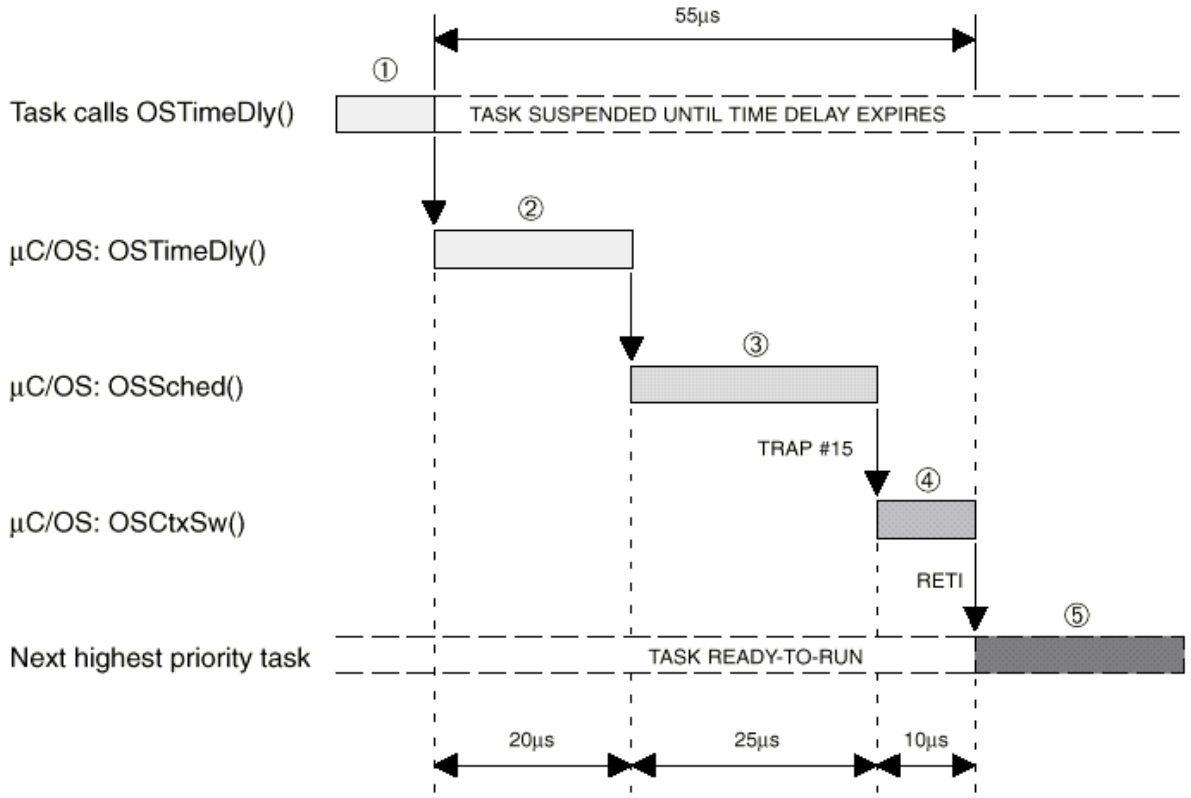


Illustration du changement de contexte de tâches

Nous avons vu précédemment que le noyau est préemptif, c'est-à-dire qu'il exécute toujours la tâche de plus forte priorité qui est prête.
Un exemple de changement de contexte peut être provoqué grâce à la primitive `OSTimeDly(n)` où n représente un nombre de quantum de temps (ou *tick*), c'est à dire le temps durant lequel une tâche prête est exécutée (1). L'appel de ce service du noyau place la tâche dans l'état attente (WAITING) (2) et appelle l'ordonnanceur (*scheduler*). L'ordonnanceur va rechercher la prochaine tâche à exécuter (choix effectué selon le degré de priorité et l'état des tâches). Seules, les tâches prêtes (READY) sont éligibles (3). Un changement de contexte est effectué par la primitive `OStxSw()` du noyau qui est spécialement écrite pour le processeur employé. En effet, le changement de contexte consiste d'une part en la sauvegarde de l'adresse de retour (PC), le pointeur de pile, les registres du processeur et l'adresse du haut de la pile et d'autre part au rétablissement du contexte de la nouvelle tâche c'est-à-dire le chargement du pointeur de pile, la pile, le PC et enfin le reste des registres du processeur.

Les temps reportés sur la figure ci-dessus correspondent à un microcontrôleur Philips XA avec une horloge de 24 MHz. On remarque que le changement de contexte ne prend que 10 µs. Le "basculement" d'une tâche à l'autre a nécessité 55 µs entre le moment de la suspension de la tâche et l'exécution de la nouvelle tâche.
Généralement, un bon noyau ne consomme pour l'ordonnancement des tâches moins de 5 % du temps CPU.

Le noyau µC/OS II nécessite une source d'interruption périodique générée ici par le *tick timer* du processeur afin de définir le quantum de temps ou *tick*. Ce paramètre (OS_TICK_OC_CNTS) est configurable selon les critères de temps des tâches mises en jeu et permet d'accroître l'efficacité du noyau (il ne faut pas passer plus de temps à commuter les tâches qu'à les exécuter, c'est à dire que le temps de commutation de tâche doit rester négligeable par rapport à la durée du *tick* (sur la figure précédente quelques dizaines de µs par rapport à quelques dizaines de ms !).

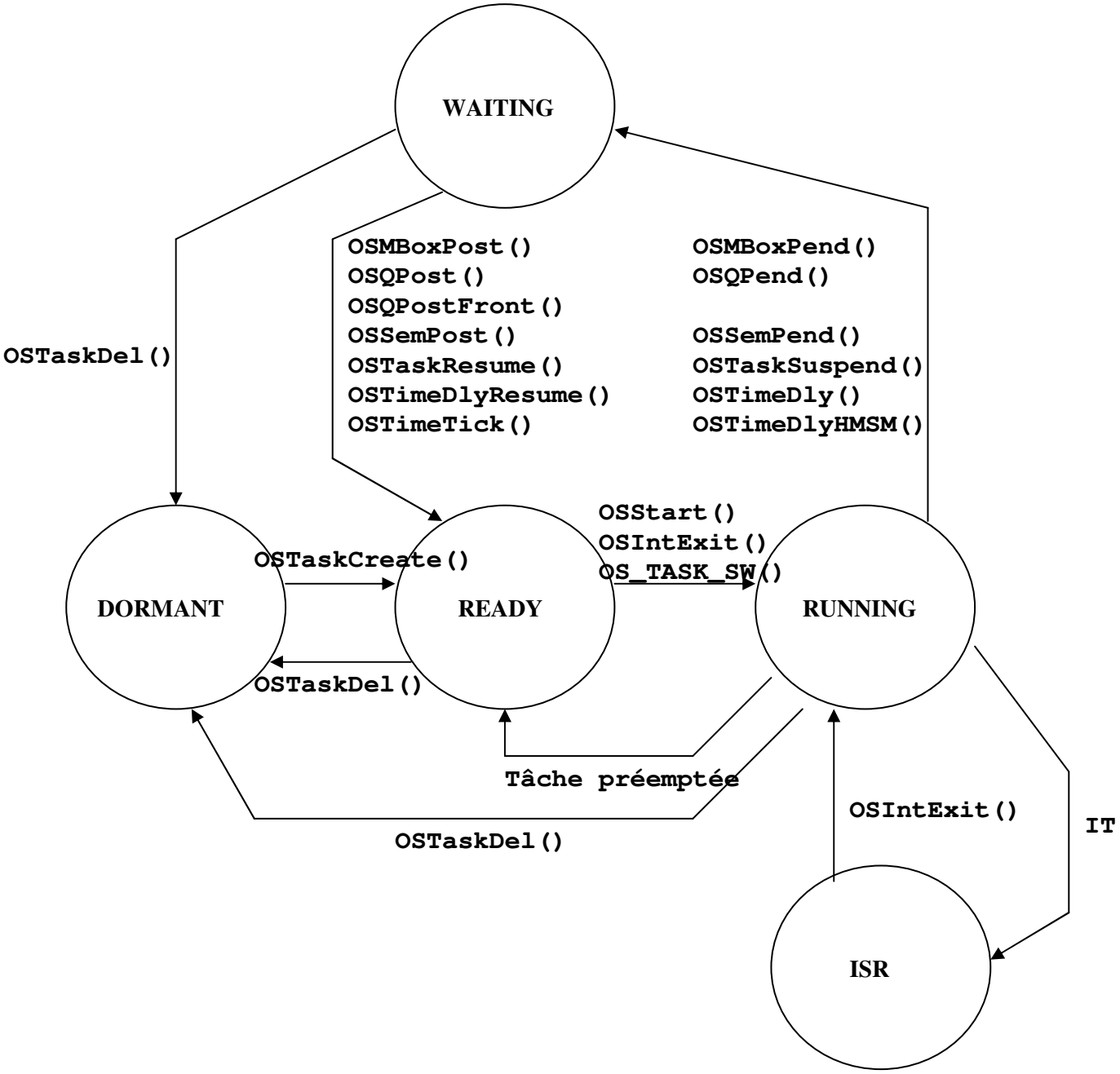
On peut ainsi conclure que la tâche de plus forte priorité prête sera exécutée durant le *tick* suivant. L'interruption suivante du *tick timer* recherchera la tâche de plus forte priorité prête qui sera alors exécutée durant le *tick* suivant et ainsi de suite. On voit ainsi que si l'on n'utilise pas des primitives du noyau µC/OS II pour changer l'état de la tâche de plus forte priorité, celle-ci aura toujours la main au détriment des autres tâches prêtes de plus faible priorité. Il faut donc pour partager le temps CPU changer l'état de cette tâche de temps en temps pour que les autres tâches prêtes aient une chance d'être exécutées. Ce changement d'état peut se faire à partir d'une ISR ou à partir de la tâche de plus forte priorité elle-même (je m'endors…).

3.2. Noyau µC/OS II

Le multitâche dans un environnement monoprocesseur permet d'effectuer à l'échelle humaine plusieurs tâches simultanément. En effet, l'utilisateur peut alors diviser son projet en plusieurs tâches indépendantes. Au niveau du processeur, une seule tâche est effectuée à la fois. En revanche, le multitâche permet d'éliminer les temps CPU inutilisés (boucle d'attente, de scrutation ou *polling*).

Une tâche peut être alors vue comme un programme en cours d'exécution avec son propre contexte (état de la tâche, valeur courantes des registres du processeur…). Il est alors possible d'avoir 2 tâches dans ces conditions qui exécutent le même programme.

La figure suivante précise les différents états possibles d'une tâche µC/OS II.



Les différents états des tâches de µC/OS II et les primitives utilisées

Le fonctionnement du noyau Temps Réel µC/OS II est le suivant :

- Après l'initialisation des ressources internes du noyau µC/OS II (primitive `OSInit()`), les différentes tâches de l'utilisateur sont créées. Le programmeur doit alors spécifier le point d'entrée de la tâche, l'emplacement des données pour cette tâche, l'adresse haute de la pile de la tâche (dans le cas du processeur Blackfin, on empile vers les adresses décroissantes et dépilent vers les adresses croissantes) et la priorité de la tâche. Ainsi la tâche de plus forte priorité prête sera toujours exécutée par le processeur. On dit que le noyau est préemptif.
- Lancement de l'ordonnanceur ou *scheduler* (primitive `OSStart()`).


Remarque : *avec le noyau* µC/OS II*, l'utilisateur dispose de 64 tâches (62 sont réellement disponibles) où chaque priorité correspond à une seule tâche, c'est-à-dire que 2 tâches ne peuvent pas avoir le même niveau de priorité (pas de round robin ou méthode dite du tourniquet pour l'ordonnancement des tâches). Le niveau de priorité sert donc comme identificateur de tâche (TID ou Task ID).*
Le niveau de priorité le plus fort est 0, le plus faible est 63 (seules, les valeurs comprises entre 0 et OS_LOWEST_PRIO-2 (OS_LOWEST_PRIO vaut en général 63) sont utilisables car le noyau crée 2 tâches : une tâche idle de priorité OS_LOWEST_PRIO (en général 63) et éventuellement une tâche de mesure de statistiques de priorité OS_LOWEST_PRIO-1 (en général 62). La tâche idle est exécutée quand aucune tâche n'est dans l'état prêt (READY). Il faut bien que le processuer fasse quelque chose !

Les caractéristiques essentielles du noyau Temps Réel µC/OS II sont les suivantes :
- Création et gestion de 62 tâches au maximum.
- Création et gestion de sémaphores binaires et à compteur.
- Changement de priorité des tâches.
- Destruction de tâches.
- Suspension et réveil de tâches.
- Envoi de messages depuis une routine d'interruption ISR (*Interrupt Service Routine*) ou d'une tâche vers une autre tâche.

Les tâches peuvent ainsi communiquer avec d'autres tâches (grâce aux sémaphores, boîtes aux lettres, files d'attentes…)
Un périphérique peut communiquer avec une tâche grâce aux ISR**.**

3.3. Création d'une tâche µC/OS II

La création d'une tâche se fait grâce à la primitive suivante :
```
OSTaskCreate(AppTask1, (void *)0, (void *)&AppTask1Stk[255], 10);
```

Le premier paramètre est le point d'entrée du programme utilisateur (nom de l'étiquette), le second paramètre indique l'adresse des données passées à la tâche, le troisième paramètre indique l'adresse du haut de la pile de la tâche (256 octets de taille ici) et le dernier paramètre indique la priorité de la tâche (ici 10).

Le corps de la tâche est constitué de la manière suivante :
- Une zone d'initialisation de la tâche.
- Une zone permettant d'initialiser les variables du programme utilisateur.
- Une zone constituée d'une boucle infinie où l'utilisateur place le code de son programme.
- Une instruction changeant l'état de la tâche de READY à WAITING généralement (exemple `OSTimeDly(n)`) au vu des remarques du paragraphe précédent.

3.4. Primitives du noyau µC/OS II

Les primitives de base du noyau Temps Réel employées sont les suivantes :

- `OSinit()` : initialisation globale du noyau.
- `OSTaskcreate (pointeur sur le programme utilisateur, pointeur données, adresse haut de la pile, priorité)` : permet de créer une tâche. Le pointeur "données" permet de faire passer des données à une tâche lors de son initialisation. La priorité est définie telle que plus le nombre est faible, plus la priorité est élevée.
- `OSStart()` : permet de lancer l'ordonnanceur avec la tâche de plus forte priorité prête. Il faut donc créer au préalable au moins une tâche en appelant `OSTaskCreate()` avant d'utiliser cette primitive.

Le fonctionnement correct de µC/OS II exige qu'une tâche doive obligatoirement appeler une primitive de service du noyau afin de :

- Endormir la tâche d'un délai de n ticks (`OSTimeDly()`).
- Attendre sur un sémaphore.
- Attendre un message d'une autre tâche ou d'une ISR.
- Suspendre l'exécution de cette tâche.

Le corps d'une tâche en langage C ressemble donc à cela :

```
void maTache(void *pdata)
{
     pdata = pdata ;

     for( ; ; ) {
     …
/* Utilisation d'au moins une de ces primitives */
     OSMboxPend() ;
     OSQPend() ;
     OSSemPend() ;
     OSTaskDel(OS_PRIO_SELF) ;
     OSTaskSuspend(OS_PRIO_SELF) ;
     OSTimeDly(…) ;
     OSTimeDlyHMSM(…) ;
     …
     }
}
```

Squelette général d'une tâche µC/OS II

Les primitives du noyau µC/OS II commencent toutes par les lettres « OS » (*Operating System*), les lettres suivant OS définissent les familles de primitives de µC/OS II. La liste des primitives de µC/OS II est la suivante :

- Initialisation : `OSinit, OSStart`.
- Gestion des tâches : `OSTaskCreate, OSTaskDel, OSTaskDelReq, OSTaskChangePrio, OSTaskSuspend, OSTaskResume, OSSchedlock, OSSchedUnlock`.
- Gestion du temps : `OSTimeDly, OSTimeDlyResume, OSTimeSet, OSTimeGet`.

- Gestion des sémaphores : `OSSemCreate`, `OSSemAccept`, `OSSemPost`, `OSSemPend`, `OSSemInit`.
- Gestion des boîtes aux lettres : `OSMboxcreate`, `OSMboxAccept`, `OSMboxPost`, `OSMboxPend`.
- Gestion des files d'attente : `OSQCreate`, `OSQAccept`, `OSQPost`, `OSQPend`.
- Gestion des interruptions : `OSIntEnter`, `OSIntExit`.

3.5. Communication entre tâches µC/OS II

Voici un bref descriptif des fonctionnalités qu'offre µC/OS II pour la communication entre tâches.

❖ Les sémaphores

Un sémaphore est un objet qui permet de résoudre les problèmes de synchronisation entre plusieurs tâches concurrentes et pour l'accès à des ressources partagées par plusieurs tâches.

Un sémaphore est constitué d'un compteur à valeur entière (valeur du sémaphore) et d'une file d'attente. Sur le compteur sont définies 2 opérations qui sont Prendre (P) et Vendre (V). Une valeur positive du sémaphore désigne le nombre d'accès disponibles à un instant donné ; une valeur négative ou nulle représente par la valeur absolue le nombre de processus en attente de libération de la ressource. En effet, avant d'accéder à une donnée, un processus doit utiliser l'opération P. Si la valeur du sémaphore est 0, alors le processus doit attendre que cette valeur devienne supérieure à 0. Si la valeur est supérieure strictement à 0, alors l'opération P décrémente la valeur du sémaphore et le processus continue son exécution. Après l'accès aux données, le processus doit appeler l'opération V qui incrémente le sémaphore autorisant ainsi aux autres processus l'accès aux données.
Les sémaphores permettent donc soit de synchroniser des processus (échanges de données), soit d'exclure des processus concurrents (accès à une variable partagée).
Les primitives sont préfixées `OSSemxxx`.

❖ Les boîtes aux lettres (*mailbox*)

Une boîte à lettre est une zone d'échange entre 2 tâches, ici une adresse (par exemple un pointeur). Les boîtes aux lettres permettent de synchroniser la communication entre des tâches asynchrones.
Les primitives sont préfixées `OSMboxxxx`.

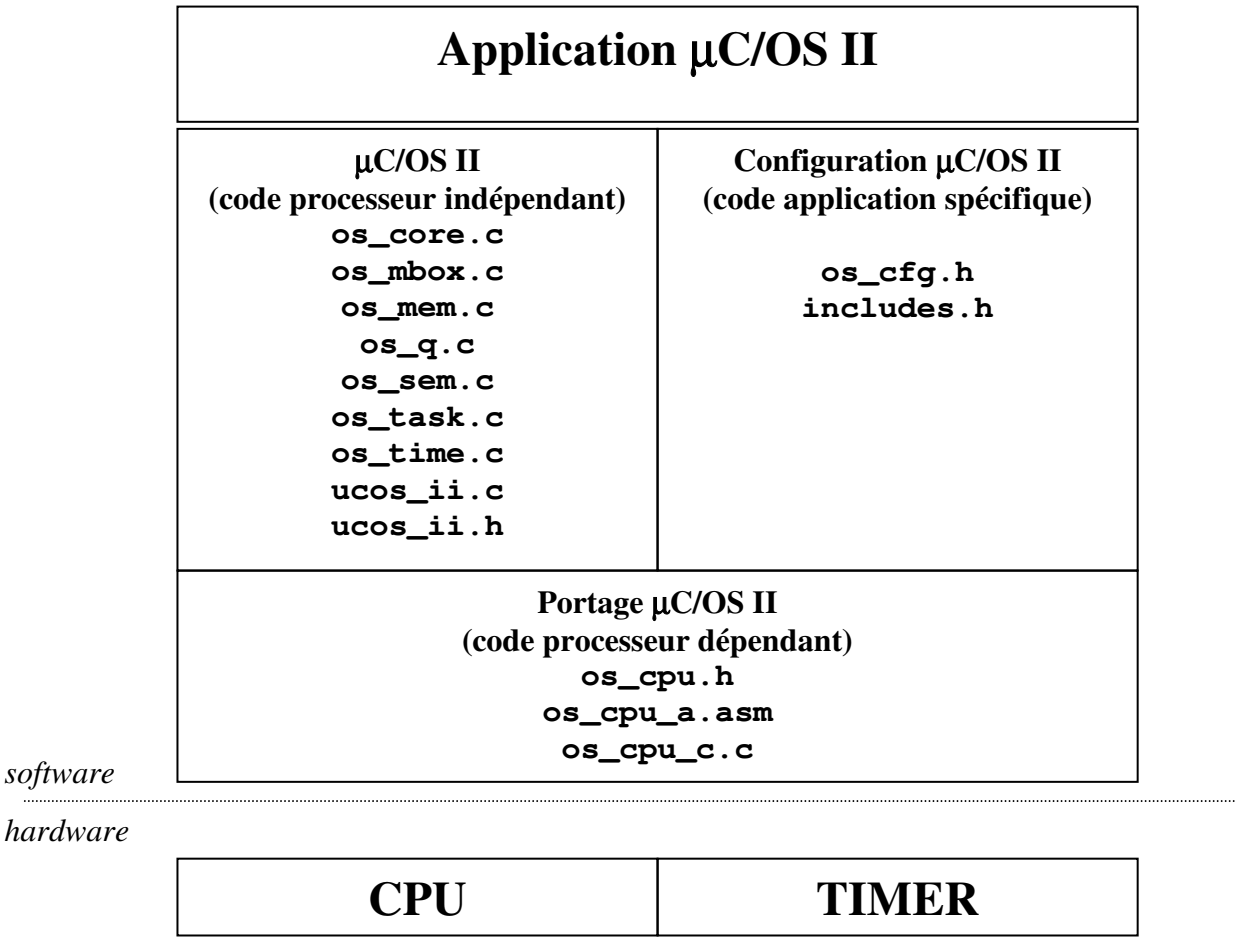
❖ Les files d'attente

Une file d'attente est un sur ensemble des boîtes aux lettres. C'est une liste de boîtes aux lettres. Cela correspond à la file de messages UNIX.
Les primitives sont préfixées `OSQxxx`.

4. PORTAGE DU NOYAU µC/OS II SUR LA CARTE BLACKFIN

Le noyau Temps Réel µC/OS II est majoritairement écrit en langage C. Seules les primitives spécifiques au processeur sont écrites en assembleur.
Un portage a déjà été réalisé pour le processeur Blackfin BF537 dans l'environnement Gnu gcc sous Linux.

La figure suivante montre l'ensemble de différents fichiers sources du noyau µC/OS II.



Architecture des fichiers sources du noyau µC/OS II

```
host% ls -R bf537-uCOS-II

bf537-uCOS-II/tp/ex0:
app.h  bsp.h        include  Makefile  os_cfg.h
bsp.c  crt0_bfin.S  main.c   obj       printf.c


bf537-uCOS-II/uCOS-II/Ports:
os_cpu_a.S  os_cpu_c.c  os_cpu_c.c.org  os_cpu.h

bf537-uCOS-II/uCOS-II/Source:
os_cfg_r.h     os_dbg_r.c  os_mem.c    os_sem.c   os_tmr.c
os_core.c      os_flag.c   os_mutex.c  os_task.c  ucos_ii.c
os_core.c.org  os_mbox.c   os_q.c      os_time.c  ucos_ii.h
```

Dans le répertoire `bf537-uCOS-II/`, nous avons le :

- Sous-répertoire `bf537-uCOS-II/uCOS-II/Source` : fichiers sources de µC/OS II processeur indépendant.
- Sous-répertoire `bf537-uCOS-II/uCOS-II/Ports` : fichiers sources du portage spécifique de µC/OS II sur le processeur Blackfin.
- Sous-répertoire `bf537-uCOS-II/tp/ex0` : exemple d'une application µC/OS II fonctionnelle pour la carte Blackfin.

Le noyau Temps Réel µC/OS II utilise 2 ressources du processeur Blackfin :

- *Timer Core Timer* avec l'interruption de niveau 6 (IVG6) pour l'interruption périodique du *tick timer* (`OSTimeTick()`)
- Interruption logicielle *trap* avec l'interruption de niveau 14 (IVG14) pour le changement de contexte de tâches (`OsctxSw()`).

5. PROGRAMMATION DE TACHES µC/OS II POUR LA CARTE BLACKFIN

Dans le sous-répertoire `bf537-uCOS-II/tp/ex0` se trouve un exemple fonctionnel d'application µC/OS-II pour la carte Blackfin.

Le fichier source `main.c` est présenté ci-après :

```
#include <string.h>
#include <stdlib.h>
#include "cfg.h"
#include "cdefBF537.h"
#include "ucos_ii.h"
#include "bsp.h"

#define TASK1_PRIO 10
#define TASK2_PRIO 15
#define ROOT_PRIO  20

#define STACKSIZE 1024

unsigned long AppStack[APPSTACKSIZE];
OS_STK stack[STACKSIZE];
OS_STK stack1[STACKSIZE];
OS_STK stack2[STACKSIZE];

int counter1;
int counter2;

/*
 * Task 1
 */
void task1(void *arg) {
INT8U OS_result;

OS_result = 0;
counter1 = 0;

while(1) {
      counter1++;

      OSTimeDly(10);
}
}


/*
 * Task 2
 */
void task2(void *arg) {
INT8U OS_result;

OS_result = 0;
counter2 = 0;

while(1) {
      counter2++;

      OSTimeDly(10);
}
}
```

```
/*
 * Root Task: main task
 */
void rootTask(void *arg) {
INT8U OS_result;

OS_result = 0;

if((OS_result     =     OSTaskCreateExt(task1,     (void     *)NULL,
&stack1[STACKSIZE-1], TASK1_PRIO, TASK1_PRIO, &stack1[0], STACKSIZE,
(void *)0, 0 )))
          printf("OSTaskCreateExt task1 failed\n");

if    ((OS_result    =    OSTaskCreateExt(task2,    (void    *)NULL,
&stack2[STACKSIZE-1], TASK2_PRIO, TASK2_PRIO, &stack2[0], STACKSIZE,
(void *)0, 0 )))
          printf("OSTaskCreateExt task2 failed\n");

while(1) {
     BSP_toggleLED (6);
     printf(".");

     OSTimeDly(100);
}
}


int main() {
INT8U OS_result;

BSP_initLED();

OSInit();

printf("uC/OS II for BF537\n");

if    ((OS_result    =    OSTaskCreateExt(rootTask,    (void    *)NULL,
&stack[STACKSIZE-1],  ROOT_PRIO,  ROOT_PRIO,  &stack[0],  STACKSIZE,
(void *)0, 0 )))
          printf("OSTaskCreateExt rootTask failed\n");

OSStart();

return(0);
}


/* app_init():
 * Called by OSInitHookEnd(void) in os_cpu_c.c
 */
void app_init(void) {

/* Initialization of tick timer */
BSP_CoreTmrInit();
}
```

```
/* register_handler_ex():
 * This function is called a few times from os_cpu_c.c.
 */
void register_handler_ex(int inum,void (*func)(void),int mode) {

switch(inum) {
      case 6:
            *pEVT6 = func;
            *pIMASK |= EVT_IVTMR;
            break;
      case 14:
            *pEVT14 = func;
            *pIMASK |= EVT_IVG14;
            break;
      default:
             printf("ERROR: unexpected inum value\n");
}
}


int Cstart(void) {

main();

return(0);
}
```

Fichier source main.c

Le point d'entrée est la fonction `main()` qui initialise les leds de la carte Blackfin (`BSP_initLED()`), initialise µC/OS II (`OSInit()`), crée la tâche principale `rootTask` et lance µC/OS II (`OSStart()`).

Au niveau des tâches µC/OS II :

- La tâche `rootTask` crée les deux tâches `task1` et `task2` puis fait clignoter la led 6 de la carte Blackfin et émet un point sur la liaison série tous les 100 *ticks*.
- La tâche `task1` incrémente le compteur `counter1` tous les 10 *ticks*.
- La tâche `task2` incrémente le compteur `counter2` tous les 10 *ticks*.

Le fichier `Makefile` permet de créer le fichier exécutable `app.elf` au format ELF (*Executable and Linkable Format*) directement exécutable depuis le *bootloader* u-boot.

Le fichier `app.sym` aussi créé correspond à la table des symboles.

```
host% cd bf537-uCOS-II/tp/ex0
host% make
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D
__ADSPLPBLACKFIN__ -I ../../uCOS-II/Source -I ../../uCOS-
II/Ports -I ./include -I . -o obj/os_core.o ../../uCOS-
II/Source/os_core.c
. . .
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D
__ADSPLPBLACKFIN__ -I ../../uCOS-II/Source -I ../../uCOS-
II/Ports -I ./include -I . -o obj/os_tmr.o ../../uCOS-
II/Source/os_tmr.c
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D
__ADSPLPBLACKFIN__ -I ../../uCOS-II/Source -I ../../uCOS-
II/Ports -I ./include -I . -D _LANGUAGE_ASM -o obj/os_cpu_a.o
../../uCOS-II/Ports/os_cpu_a.S
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D
__ADSPLPBLACKFIN__ -I ../../uCOS-II/Source -I ../../uCOS-
II/Ports -I ./include -I . -o obj/os_cpu_c.o ../../uCOS-
II/Ports/os_cpu_c.c
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D
__ADSPLPBLACKFIN__ -I ../../uCOS-II/Source -I ../../uCOS-
II/Ports -I ./include -I . -o obj/crt0_bfin.o crt0_bfin.S
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D
__ADSPLPBLACKFIN__ -I ../../uCOS-II/Source -I ../../uCOS-
II/Ports -I ./include -I . -o obj/main.o main.c
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D
__ADSPLPBLACKFIN__ -I ../../uCOS-II/Source -I ../../uCOS-
II/Ports -I ./include -I . -o obj/bsp.o bsp.c
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D
__ADSPLPBLACKFIN__ -I ../../uCOS-II/Source -I ../../uCOS-
II/Ports -I ./include -I . -o obj/printf.o printf.c
bfin-elf-ld -e start -o app.elf -Ttext 0x100000 obj/os_core.o
obj/os_dbg_r.o obj/os_flag.o obj/os_mbox.o obj/os_mem.o
obj/os_mutex.o obj/os_q.o obj/os_sem.o obj/os_task.o
obj/os_time.o obj/os_tmr.o obj/os_cpu_a.o obj/os_cpu_c.o
obj/crt0_bfin.o obj/main.o obj/bsp.o obj/printf.o \
              -lc `bfin-elf-gcc --print-libgcc-file-name`
bfin-elf-nm app.elf > app.sym

host% grep counter app.sym
001088cc B _counter1
001084c8 B _counter2
host% cp app.elf /tftpboot

bfin> tftp app.elf
bfin> bootelf
```

6. FONCTIONS UTILITAIRES DU BOARD SUPPORT PACKAGE POUR LA CARTE BLACKFIN

Une bibliothèque de fonctions utilitaires a été écrite pour initialiser et utiliser les ressources matérielles de la carte Blackfin. Il s'agit en fait du BSP (*Board Support Package*). Le BSP correspond aux deux fichiers `bsp.c` et `bsp.h`.

Voici la description des fonctions utilitaires et leur prototype :

```
/*
**
** Fonction: BSP_CoreTmrInit()
**               entree(s) : rien
**               sortie(e) : rien
** Description :
** Initialisation du Core Timer du processeur Blackfin
**
**/
void  BSP_CoreTmrInit (void);


/*
**
** Fonction: BSP_initLED()
**               entree(s) : rien
**               sortie(e) : rien
** Description :
** Initialisation des 6 leds de la carte Blackfin qui sont
** éteintes
**
**/
void  BSP_initLED  (void);


/*
**
** Fonction: BSP_setLED()
**               entree(s) : numéro de la led de 1 à 6
**               sortie(e) : rien
** Description :
** Allumage d'une led de la carte Blackfin
**
**/
void  BSP_setLED    (INT8U lite);
```

```
/*
**
** Fonction: BSP_clrLED()
**              entree(s) : numéro de la led de 1 à 6
**              sortie(e) : rien
** Description :
** Extinction d'une led de la carte Blackfin
**
**/
void  BSP_clrLED    (INT8U lite);


/*
**
** Fonction: BSP_toggleLED()
**              entree(s) : numéro de la led de 1 à 6
**              sortie(e) : rien
** Description :
** Changement de l'état d'une led de la carte Blackfin
**
**/
void  BSP_toggleLED (INT8U lite);


/*
**
** Fonction: putchar()
**              entree(s) : caractère
**              sortie(e) : rien
** Description :
** Envoi d'un caractère sur la liaison série de la
** carte Blackfin
**
**/
void  putchar (char c);


/*
**
** Fonction: getchar()
**              entree(s) : rien
**              sortie(e) : caractère
** Description :
** Lecture d'un caractère éventuellement reçu sur la liaison
** série de la carte Blackfin. Attention, la fonction
** est bloquante
**
**/
char  getchar ();
```

```
/*
**
** Fonction: printf() et compagnie
**              entree(s) :
**              sortie(e) :
** Description :
** Fonctions classiques de la famille printf(). Les caractères
sont envoyés sur la liaison série de la carte Blackfin
**
**/
void printchar(char **str, int c);
int prints(char **out, const char *string, int width, int pad);
int printi(char **out, int i, int b, int sg, int width, int
pad, int letbase);
int print(char **out, int *varg);
int printf(const char *format, ...);
int sprintf(char *out, const char *format, ...);
```

7. TP 0 : PRISE EN MAIN

- Démarrer le PC sous Linux. Se connecter sous le nom **guest**, mot de passe : **guest** ☺.

- Se créer un répertoire de travail à son nom et s'y placer :
```
host% cd
host% mkdir mon_nom
host% cd mon_nom
```

- Etablir le schéma de l'environnement de développement :
  - Matériels.
  - Liaisons : série, réseau…
  - Logiciels et OS utilisés.
  - Adresses IP du PC de développement (hôte ou *host*) et de la carte cible (cible ou *target*).

- Se connecter à la carte d'évaluation Blackfin (cible) en utilisant l'outil `minicom`.
  Pour sortir de `minicom`, il suffit de taper la combinaison de touches : CTRL A, Z pour accéder au menu et taper q pour quitter.

- Retrouver les commandes du *bootloader* u-boot de la cible pour :
  - Télécharger par le réseau Ethernet via TFTP un fichier ELF `app.elf`.
  - Lancer et exécuter le fichier ELF `app.elf` précédemment chargé en mémoire RAM.
  - Mettre à zéro une zone mémoire.

- Quelle est l'adresse de point d'entrée de l'exécutable précédemment téléchargé ?

Par la suite, on adoptera les conventions suivantes :
Commande Linux PC hôte :
```
host% commande Linux
```
Commande u-boot :
```
bfin> commande u-boot
```

8. TP 1 : TESTS. MISE EN ŒUVRE DU NOYAU µC/OS II

- Dans son répertoire à son nom, recopier tous le fichier `bf537-uCOS-II.tgz` sous `~kadionik/` :
  ```
  host% cp -r /home/kadionik/bf537-uCOS-II.tgz .
  ```

- Décompresser et installer le fichier `bf537-uCOS-II.tgz` :
  ```
  host% tar -xvzf bf537-uCOS-II.tgz
  ```

- Se placer ensuite dans le répertoire `bf537-uCOS-II/tp/`. L'ensemble du travail sera réalisé à partir de ce répertoire ! **Les chemins seront donnés par la suite en relatif par rapport à ce répertoire...**

- Recopier le répertoire `ex0/` dans le répertoire `ex1/` et s'y placer :
  ```
  host% cp -r ex0 ex1
  host% cd ex1
  ```

- Analyser le fichier `main.c`. Que réalisent les tâches `task1`, `task2` et `rootTask` ? Que réalise la fonction `main()` ?

- Analyser le fichier `Makefile`. Quelle est l'adresse du point d'entrée de l'application ? Quel est le compilateur croisé utilisé ? Quel est le format du fichier binaire produit ? Dans quel fichier se trouve la table des symboles ?

- Compiler l'application de test :
  ```
  host% make
  ```

- Quels sont les fichiers créés lors de la compilation ?

- Analyser le fichier de la table des symboles. Retrouver l'adresse de la valeur courante des compteurs `counter1` et `counter2` ? Mettre à zéro les deux zones mémoire avec u-boot hébergeant la valeur courante des compteurs `counter1` et `counter2` :
  ```
  bfin> mw.b @adresse 0 4
  ```

- Recopier le fichier binaire produit sous `/tftpboot/` pour pouvoir le télécharger dans la carte Blackfin :
  ```
  host% cp app.elf /tftpboot
  ```

- Télécharger le fichier binaire produit et l'exécuter depuis u-boot :
  ```
  bfin> tftp app.elf
  bfin> bootelf
  ```

- Vérifier le bon fonctionnement de l'application de test avec le clignotement de la led 6 et des traces reçues sur le port série.

- Après reset de la carte Blackfin, vérifier par analyse de la mémoire que la valeur courante des compteurs `counter1` et `counter2` s'est bien incrémentée. Expliquer pourquoi la valeur courante des compteurs `counter1` et `counter2` est la même :
  `bfin> md @adresse`

9. TP 2 : MULTITACHE. ECHO SUR RS.232 ET PLUS UN AVEC LE NOYAU µC/OS II

Le but de ce TP est d'écrire un programme d'écho sur le port série de la carte Blackfin avec en concurrence l'incrémentation des 2 compteurs.
On modifiera le fichier `main.c` avec les conditions suivantes :

- Tâche `rootTask` : lancement des tâches `task1` et `task2` et ensuite programme d'écho sur la liaison série RS.232.
- Tâche `task1` : inchangée, fait du « plus 1 ».
- Tâche `task2` : inchangée, fait du « plus 1 ».

- Recopier le répertoire `ex0/` dans le répertoire `ex2/` et s'y placer :
```
host% cp -r ex0 ex2
host% cd ex2
```

- Modifier le programme `main.c` suivant les conditions précédentes. L'ensemble des autres fichiers est donné (`Makefile`, fichiers .h et .c).

- Compiler l'ensemble à l'aide de la commande `make` :
```
host% make
```

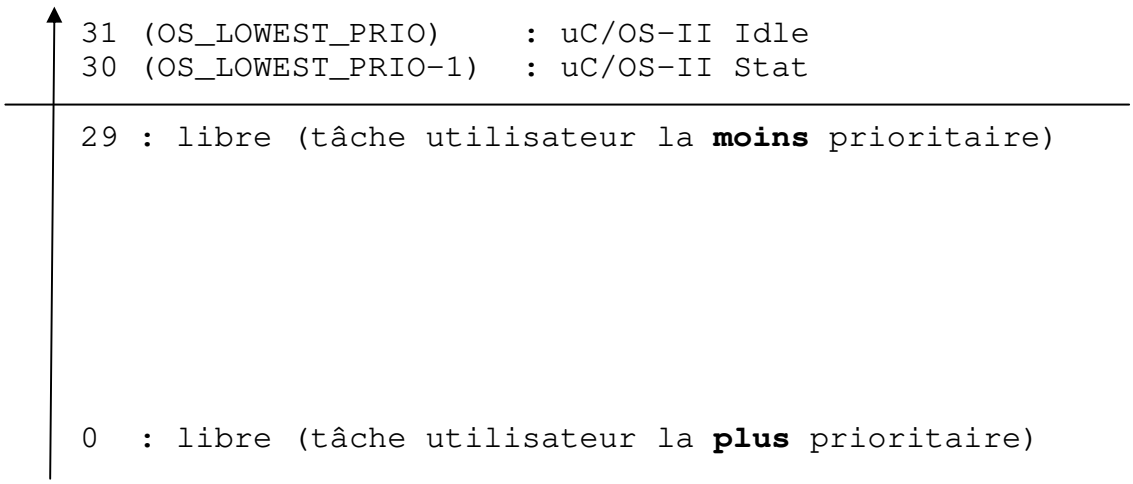
- Analyser le fichier `app.sym`. Retrouver les adresses des étiquettes `counter1` et `counter2`.

- Mettre à zéro le contenu des adresses correspondant à `counter1` et `counter2`.

- Télécharger le fichier binaire `app.elf` dans la carte Blackfin et tester l'application. Vérifier le bon fonctionnement de l'ensemble.

Remarque :

Pour les TP, la priorité d'une tâche µC/OS II doit être prise dans la plage 0 à 29 (`OS_LOWEST_PRIO-2`).



10. TP 3 : MULTITACHE. ECHO SUR RS.232, CHENILLARD ET PLUS UN AVEC LE NOYAU µC/OS II

Le but de ce TP est de faire exécuter par le noyau 3 tâches distinctes.
On modifiera le fichier `main.c` avec les conditions suivantes :

- Tâche `rootTask` : lancement des tâches `task1` et `task2` et ensuite programme d'écho sur la liaison série RS.232.
- Tâche `task1` : chenillard sur les leds 1 à 6.
- Tâche `task2` : inchangée, fait du « plus 1 ».

- Recopier le répertoire `ex0/` dans le répertoire `ex3/` et s'y placer :
```
host% cp -r ex0 ex3
host% cd ex3
```

- Modifier le programme `main.c` suivant les conditions précédentes. L'ensemble des autres fichiers est donné (`Makefile`, fichiers .h et .c).

- Compiler l'ensemble à l'aide de la commande `make` :
```
host% make
```

- Télécharger le fichier binaire `app.elf` dans la carte Blackfin et tester l'application. Vérifier le bon fonctionnement de l'ensemble.

11. TP 4 : SEMAPHORES BINAIRES. GESTION DE L'ACCES EXCLUSIF A UNE RESSOURCE PARTAGEE

Le but de ce TP est de gérer l'accès exclusif à une ressource partagée. Celle-ci sera représentée par la led 1.
On modifiera le fichier `main.c` avec les conditions suivantes :

- Tâche `rootTask` : création d'un sémaphore binaire `sem` et lancement des tâches `task1` et `task2` bloquées sur le sémaphore `sem`. Libération du sémaphore `sem`.
- Tâche `task1` : bloquée sur le sémaphore binaire `sem`. A sa libération, changement d'état de la led 1 puis libération de `sem`.
- Tâche `task2` : bloquée sur le sémaphore binaire `sem`. A sa libération, changement d'état de la led 1 puis libération de `sem`.

- Recopier le répertoire `ex0/` dans le répertoire `ex4/` et s'y placer :
```
host% cp -r ex0 ex4
host% cd ex4
```

- Modifier le programme `main.c` suivant les conditions précédentes. L'ensemble des autres fichiers est donné (`Makefile`, fichiers .h et .c).

- Compiler l'ensemble à l'aide de la commande `make` :
```
host% make
```

- Télécharger le fichier binaire `app.elf` dans la carte Blackfin et tester l'application. Vérifier le bon fonctionnement de l'ensemble.

12. TP 5 : SEMAPHORES BINAIRES. SYNCHRONISATION DE TACHES. RENDEZ-VOUS

Le but de ce TP est de synchroniser 2 tâches à un instant donné dans l'exécution de leur code. Cela s'appelle un rendez-vous.
On modifiera le fichier `main.c` avec les conditions suivantes :

- Tâche `rootTask` : création d'un sémaphore binaire `sem` et lancement des tâches `task1` et `task2`.
- Tâche `task1` : bloquée sur le sémaphore binaire `sem`.
- Tâche `task2` : donne un RDV à la tâche `task1` par libération du sémaphore binaire `sem` qui allumera en conclusion la led 1.

- Recopier le répertoire `ex0/` dans le répertoire `ex5/` et s'y placer :
```
host% cp -r ex0 ex5
host% cd ex5
```

- Modifier le programme `main.c` suivant les conditions précédentes. L'ensemble des autres fichiers est donné (`Makefile`, fichiers .h et .c).

- Compiler l'ensemble à l'aide de la commande `make` :
```
host% make
```

- Télécharger le fichier binaire `app.elf` dans la carte Blackfin et tester l'application. Vérifier le bon fonctionnement de l'ensemble.

13. TP 6 : SEMAPHORES BINAIRES. RECAPITULATIF

Le but de ce TP est de réaliser un *dispatching* d'un travail par une tâche.
On modifiera le fichier `main.c` avec les conditions suivantes :

- Tâche `rootTask` : création de 3 sémaphores binaires `sem1`, `sem2` et `sem3` et lancement des tâches `task1`, `task2` et `task3` bloquées sur sémaphore. Libération des sémaphores un par un pour activer chaque tâche.
- Tâche `task1` : bloquée sur le sémaphore binaire `sem1`. A sa libération par `rootTask`, allumage de la led 1.
- Tâche `task2` : bloquée sur le sémaphore binaire `sem2`. A sa libération par `rootTask`, allumage de la led 2.
- Tâche `task3` : bloquée sur le sémaphore binaire `sem3`. A sa libération par `rootTask`, allumage de la led 3.

- Recopier le répertoire `ex0/` dans le répertoire `ex6/` et s'y placer :
```
host% cp -r ex0 ex6
host% cd ex6
```

- Modifier le programme `main.c` suivant les conditions précédentes. L'ensemble des autres fichiers est donné (`Makefile`, fichiers .h et .c).

- Compiler l'ensemble à l'aide de la commande `make` :
```
host% make
```

- Télécharger le fichier binaire `app.elf` dans la carte Blackfin et tester l'application. Vérifier le bon fonctionnement de l'ensemble.

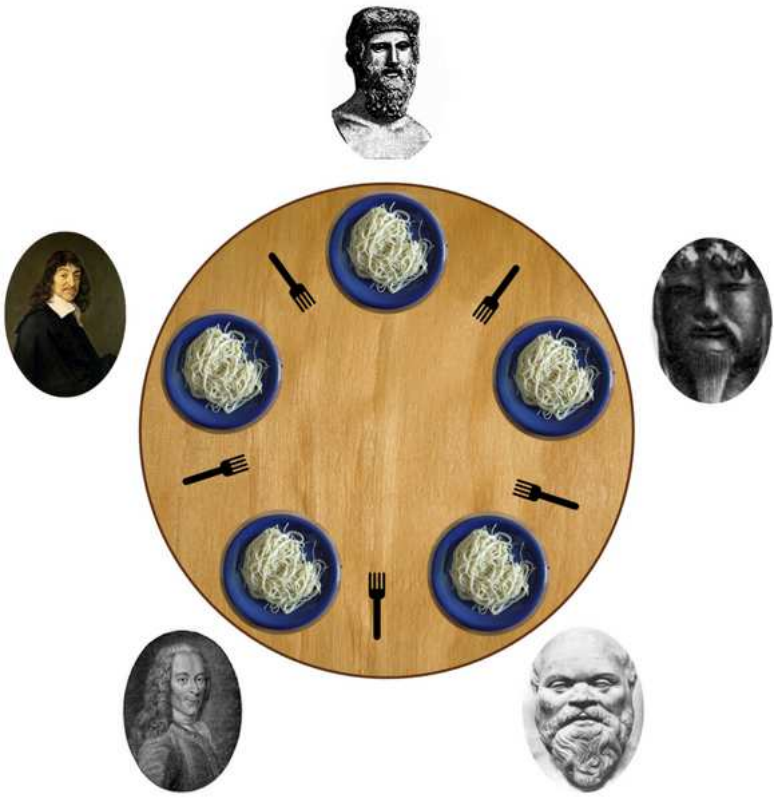
14. TP 7 : SEMAPHORES BINAIRES. PROBLEME DES PHILOSOPHES

Le but de ce TP est traiter le problème des philosophes.

Le problème des philosophes et des spaghettis est un problème classique en théorie informatique et plus particulièrement pour les questions d'ordonnancement des processus. Ce problème a été énoncé par Edsger Dijkstra, l'inventeur du concept des sémaphores.

La situation est la suivante :

- 5 philosophes se trouvent autour d'une table.
- Chacun des philosophes a devant lui un plat de spaghettis.
- A gauche de chaque assiette se trouve une fourchette.



Un philosophe n'a que deux états possibles :

- Penser pendant un temps déterminé.
- Manger.

Des contraintes extérieures s'imposent à cette situation :

- Pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette et celle qui se trouve à gauche de celle de son voisin de droite (soit les deux fourchettes qui entourent sa propre assiette).
- Si un philosophe n'arrive pas à s'emparer d'une fourchette, il se met à penser pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème est le suivant : si tous les philosophes essayent en même temps de manger :

1. Ils vont tous vouloir saisir les mêmes fourchettes en même temps.
2. Tous vont échouer.
3. Tous vont se mettre à penser pendant un certain temps.
4. Tous vont renouveler leur tentative en même temps.
5. etc à l'infini.

L'une des principales solutions à ce problème est celle du sémaphore, proposée, comme ce problème, par Edsger Dijkstra.

- Recopier le répertoire `ex0/` dans le répertoire `ex7/` et s'y placer :
```
host% cp -r ex0 ex7
host% cd ex7
```

- Modifier le programme `main.c` suivant les conditions précédentes. L'ensemble des autres fichiers est donné (`Makefile`, fichiers .h et .c).

- Compiler l'ensemble à l'aide de la commande `make` :
```
host% make
```

- Télécharger le fichier binaire `app.elf` dans la carte Blackfin et tester l'application. Vérifier le bon fonctionnement de l'ensemble.

15. TP 8 : GESTION DU TEMPS

Le but de ce TP est de faire exécuter par le noyau 4 tâches distinctes qui affichent un message à des périodes différentes.
On modifiera le fichier `main.c` avec les conditions suivantes :

- Tâche `rootTask` : lancement des tâches `task1`, `task2` et `task3`.
- Tâche `task1` : affichage d'un message toutes les 1 secondes.
- Tâche `task2` : affichage d'un message toutes les 5 secondes.
- Tâche `task3` : affichage d'un message toutes les 10 secondes.

- Recopier le répertoire `ex0/` dans le répertoire `ex8/` et s'y placer :
```
host% cp -r ex0 ex8
host% cd ex8
```

- Modifier le programme `main.c` suivant les conditions précédentes. L'ensemble des autres fichiers est donné (`Makefile`, fichiers .h et .c).

- Compiler l'ensemble à l'aide de la commande `make` :
```
host% make
```

- Télécharger le fichier binaire `app.elf` dans la carte Blackfin et tester l'application. Vérifier le bon fonctionnement de l'ensemble.

16. TP 9 : GESTION D'UNE MAILBOX

Le but de ce TP est de mettre en œuvre une file de messages ou *mailbox*. Une tâche envoye périodiquement des messages à une autre tâche qui, sur réception du message, change l'état courant de la led x en fonction du contenu du message (x=1 pour led 1 à x=6 pour led 6). On modifiera le fichier `main.c` avec les conditions suivantes :

- Tâche `rootTask` : création d'une file de messages `mbox` et lancement des tâches `task1` et `task2`.
- Tâche `task1` : postage d'un message dans `mbox` précisant le numéro de la led (1 à 6) et son état (on ou off) toutes les secondes.
- Tâche `task2` : attente d'un éventuel message dans `mbox` et changement de l'état de la led considérée en conséquence en cas de réception d'un message.

- Recopier le répertoire `ex0/` dans le répertoire `ex9/` et s'y placer :
```
host% cp -r ex0 ex9
host% cd ex9
```

- Modifier le programme `main.c` suivant les conditions précédentes. L'ensemble des autres fichiers est donné (`Makefile`, fichiers .h et .c).

- Compiler l'ensemble à l'aide de la commande `make` :
```
host% make
```

- Télécharger le fichier binaire `app.elf` dans la carte Blackfin et tester l'application. Vérifier le bon fonctionnement de l'ensemble.

- On en profitera pour réaliser ainsi un chenillard.

17. TP 10 : EXERCICE FINAL

On réalisera un chronomètre au 1/10 ème de seconde affichant le temps courant sur la liaison série avec gestion du chronomètre :

- Start : touche `s`.
- Stop : touche `t`.
- Remise à zéro : touche `r`.

18. TP 11 : MINIPROJET

On réalisera un interpréteur de commandes pour récupérer des statistiques sur les tâches µC/OS II qui tournent sur la carte cible.

Cet interpréteur sera aussi un *shell* de type UNIX.

Le prompt sera : « `ucos%` ».

- On implémentera la commande `uname` qui affichera la version de µC/OS II :
  `ucos% uname`
  On utilisera la primitive `OSVersion()` pour cela.

- On implémentera la commande `cpu` qui affichera le temps CPU consommé, le nombre de tâches en cours et le nombre moyen de changements de contextes :
  `ucos% cpu`
  Les variables gobables suivantes de µC/OS II seront exploitées :
  - `INT8U OSTaskCtr` : nombre de tâches en cours d'exécution.
  - `INT8S OSCPUUsage` : pourcentage CPU consommé en %.
  - `INT32U OSCtxSwCtr` : nombre global de changements de contextes. Cette variable sera remise à zéro puis lue au bout d'une seconde pour avoir le nombre moyen de changements de contextes par seconde.

- On implémentera la commande `ps` qui affichera la liste des tâches en cours et des informations sur ces tâches :
  `ucos% ps`
  La constante `OS_LOWEST_PRIO` donne la priorité de la tâche de plus faible priorité. La valeur est fixée à 31 dans le portage pour le processeur Blackfin. On créera une tâche `rootTask` de priorité `(OS_LOWEST_PRIO-2)` qui sera le `shell` µC/OS II et une tâche `blink` de priorité `(OS_LOWEST_PRIO-3)` qui fera clignoter la led 6 de la carte toutes les secondes. La priorité `(OS_LOWEST_PRIO -1)` est réservée à la tâche système de statistiques `uC/OS-II Stat` et la priorité `(OS_LOWEST_PRIO)` est réservée à la tâche système de fond `uC/OS-II Idle`.
  Cela donne alors le mapping des priorités suivant :
  - `OS_LOWEST_PRIO (31) : uC/OS-II Idle`
  - `OS_LOWEST_PRIO-1 (30) : uC/OS-II Stat`
  - `OS_LOWEST_PRIO-2 (29) : rootTask`
  - `OS_LOWEST_PRIO-3 (28) : blink`

  La primitive `OSTaskQuery()` permet de récupérer des informations sur une tâche :
  `INT8U OSTaskQuery(INT8U prio, OS_TCB *pdata)`
  - `pdata.OSTCBCtxSwCtr` : renvoie le nombre de changements de contextes pour la tâche considérée.

  La primitive `OSTaskNameGet()` permet de récupérer des informations sur une tâche :
  `INT8U OSTaskNameGet(INT8U prio, char *pname, INT8U *err)`
  La primitive `OSTaskStkChk()` permet de récupérer des informations de statistiques sur une tâche :
  `INT8U OSTaskStkChk(INT8U prio, OS_STK_DATA *pdata)`

- `pdata.OSFree+pdata.OSUsed` : quantité mémoire totale de la pile de la tâche considérée.
- `pdata.OSFree` : quantité mémoire disponible de la pile de la tâche.
- `pdata.OSUsed` : quantité mémoire consommée de la pile de la tâche.

La commande `ps` affichera donc pour chaque tâche :

- Le nom de la tâche.
- La priorité de la tâche.
- Le nombre de changements de contextes de la tâche.
- La quantité de mémoire totale, libre et consommée de la pile de la tâche.

- On implémentera la commande `spawn` qui créera une tâche de priorité `i` (comprise entre 0 et 9 inclus) qui exécutera en boucle la fonction fournie `compute()` dont le code source est dans le fichier `main.c` pour consommer du temps CPU :
  `ucos% spawn i  0 <= i <=9`
  Le nom de la tâche `i` sera `taski`. La primitive `OSTaskNameSet()` permet d'attribuer un nom symbolique à une tâche :
  `void OSTaskNameSet(INT8U prio, char *pname, INT8U *err)`

- On implémentera la commande `kill` qui tuera la tâche de priorité `i` (comprise entre 0 et 9 inclus) :
  `ucos% kill i  0 <= i <=9`
  La primitive `OSTaskDel()` permet de tuer une tâche :
  `INT8U OSTaskDel(INT8U prio)`

On pourra utiliser la fonction `myscans()` dont le code source est dans le fichier `main.c` pour lire une chaîne de caractères saisie au clavier jusqu'à l'appui sur la touche `entrée` :
`void myscans(char *str)`

Les traces suivantes montrent un enchaînement typique des commandes du *shell* µC/OS II :

```
ucos% cpu
OSTaskCtr       OSCPUUsage      OSCtxSwCtr
04              001%            0022
ucos% ps
task           priority        switch          total_mem       free            used
blink          28              0012            4096            3800             296
rootTask       29              0015            4096            3276             820
uC/OS-II Stat  30              0006            512             196              316
uC/OS-II Idle  31              0007            1600            1300             300
ucos% spawn 0
ucos% ps
task          priority         switch          total_mem       free            used
task00         00              0300            4096            3808             288
blink          28              0022            4096            3800             296
rootTask       29              0324            4096            3276             820
uC/OS-II Stat  30              0006            512             196              316
uC/OS-II Idle  31              0007            1600            1300             300
ucos% cpu
OSTaskCtr       OSCPUUsage      OSCtxSwCtr
05              053%            0218
```

```
ucos% spawn 1
ucos% ps
task           priority        switch          total_mem       free            used
task00         00              1327            4096            3808             288
task01         01              0182            4096            3604             492
blink          28              0032            4096            3800             296
rootTask       29              1216            4096            3276             820
uC/OS-II Stat  30              0011            512             196              316
uC/OS-II Idle  31              0057            1600            1300             300
ucos% cpu
OSTaskCtr       OSCPUUsage      OSCtxSwCtr
06              079%            0272
ucos%
```

19. REFERENCES

- ADSP-BF537 EZ-KIT Lite Evaluation System Manual. http://www.analog.com/static/imported-files/eval_kit_manuals/ADSP-BF537_EZ-KIT_Lite_Manual_Rev_2_4.pdf
- ADSP-BF537 Blackfin Processor Hardware Reference. http://www.analog.com/static/imported-files/processor_manuals/bf537_hwr_Rev3.2.pdf
- µC/OS-II and the Blackfin Processor (ADSP-BF537). Application Note AN-1530.

20. ANNEXE 1 : PRÉSENTATION D'U-BOOT

21. ANNEXE 2 : PORTAGE DE µC/OS II SUR LE PROCESSEUR BLACKFIN BF537

22. ANNEXE 3 : CONFIGURATION RESEAU HOTES ET CIBLES

	POSTE PC01	
	NOM	**ADRESSE IP**
HOTE	rodirula	10.7.4.223
CIBLE	blackfin001	10.7.2.118

	POSTE PC02	
	NOM	**ADRESSE IP**
HOTE	sarracenia	10.7.4.225
CIBLE	blackfin002	10.7.2.119

	POSTE PC03	
	NOM	**ADRESSE IP**
HOTE	utricularia	10.7.4.227
CIBLE	blackfin003	10.7.2.120

	POSTE PC04	
	NOM	**ADRESSE IP**
HOTE	ibicella	10.7.2.112
CIBLE	blackfin004	10.7.2.121

	POSTE PC05	
	NOM	**ADRESSE IP**
HOTE	nepenthes	10.7.2.114
CIBLE	blackfin005	10.7.2.122

	POSTE PC06	
	NOM	**ADRESSE IP**
HOTE	pinguicula	10.7.2.116
CIBLE	blackfin006	10.7.2.123

Masque de sous réseau : **255.255.248.0**

Exemple : configuration réseau de la carte cible `blackfin001` :

```
target# ifconfig eth0 10.7.2.118 netmask 255.255.248.0
```