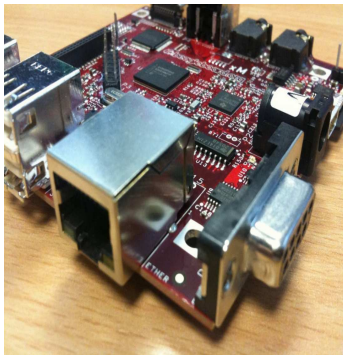


Le système d'exploitation temps réél QNX sous Beagleboard

TABLE DES MATIERES

Mise en œuvre de la carte beagleboard avec Le RTOS QNX

1) **Présentation de la carte beagleboard** **Page N°3**
2) **Câblage de la carte et mise en œuvre logicielle** **Page N°4**

Le système d'exploitation QNX

1) **Historique** **Page N°9**
2) **Présentation** **Page N°9**
3) **Gestion des processus** **Page N°10**
4) **Les techniques d'ordonnancement des processus** **Page N°17**
5) **Création d'une tache sous QNX** **Page N°19**
6) **communication inter-processus** **Page N°24**
7) **La synchronisation des processus et des threads** **Page N°39**
8) **Gestion des entrées sorties sous QNX** **Page N°48**
9) **Gestion des interruptions sous QNX** **Page N°56**
10) **Le Gestionnaire de ressource** **Page N°71**

Mise en œuvre de la carte beagleboard avec Le RTOS QNX

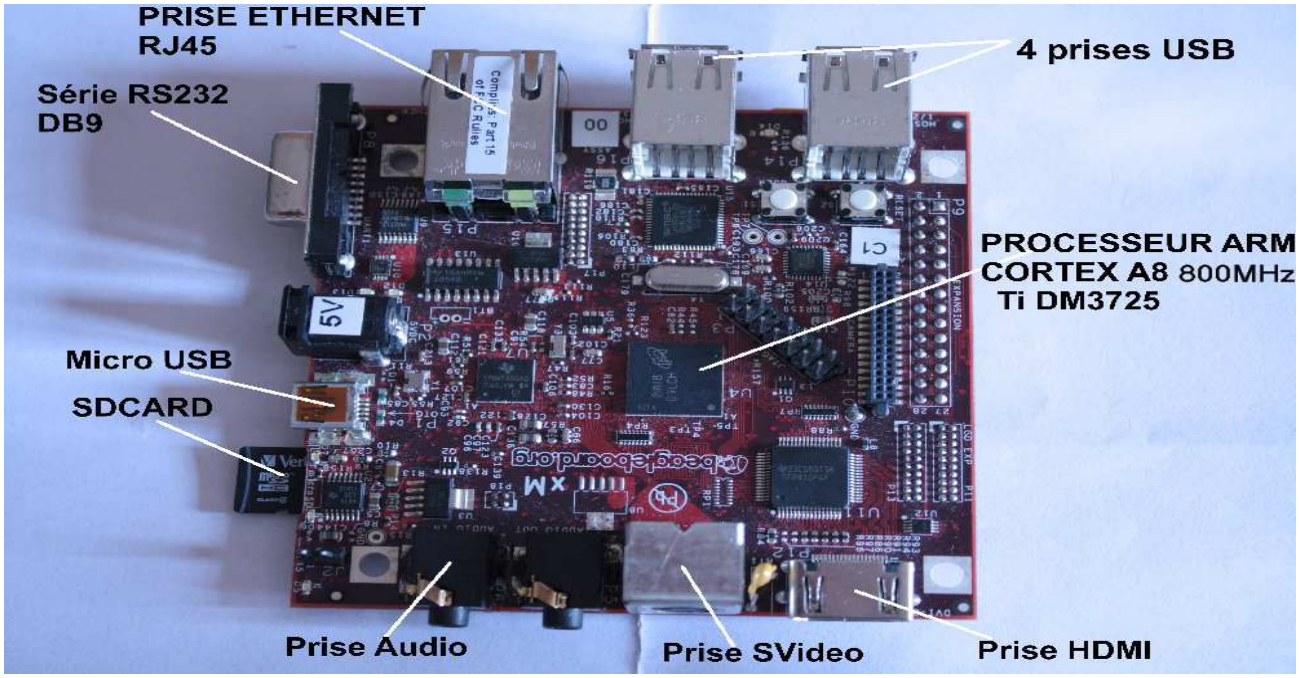
1) Présentation de la carte beagleboard

La BeagleBoard-xM est une carte à base de microprocesseur Cortex A8. Le microcontrôleur est un processeur DM3730 fabriqué et vendu par texas instruments. Ce fleuron du monde de l'électronique embarquée intègre un cœur Risc 32 bits ARM Cortex-A8 cadencé à 800 MHz (avec une unité multimédia Neon), un puissant DSP TMS320C64x cadencé à 430 MHz et un accélérateur graphique 2D/3D Imagination Technologies PowerVR SGX530 compatible avec l'OpenGL ES 2.0 (résolution maximale de 1024 x 768 points en 24 bits). Cette carte est destinée avant tout aux applications *desktop*. L'équipe de développement ayant privilégié la connectique vidéo (HDMI et S-Video) et audio (entrée/sortie stéréo). L'environnement graphique openGL permet de développer des applicatifs en C++ sous la base d'interface homme machine. Les environnements intégrés Qtcréator et photon GUI sont les outils de base pour obtenir des applicatifs à interfaces graphiques.

Le reste de l'équipement comprend 512 Mo de mémoire vive et de la flash avec un lecteur de micro-SDCARD. C'est largement suffisant pour accueillir un système d'exploitation type Linux embarqué, QNX, Android sur cette carte mère.

Linux se taille une part de choix sur cet ordinateur **conçu pour et par des passionnés**. Le processeur est reconnu par les dernières moutures du noyau, tout comme l'unité multimédia Neon et le DSP.

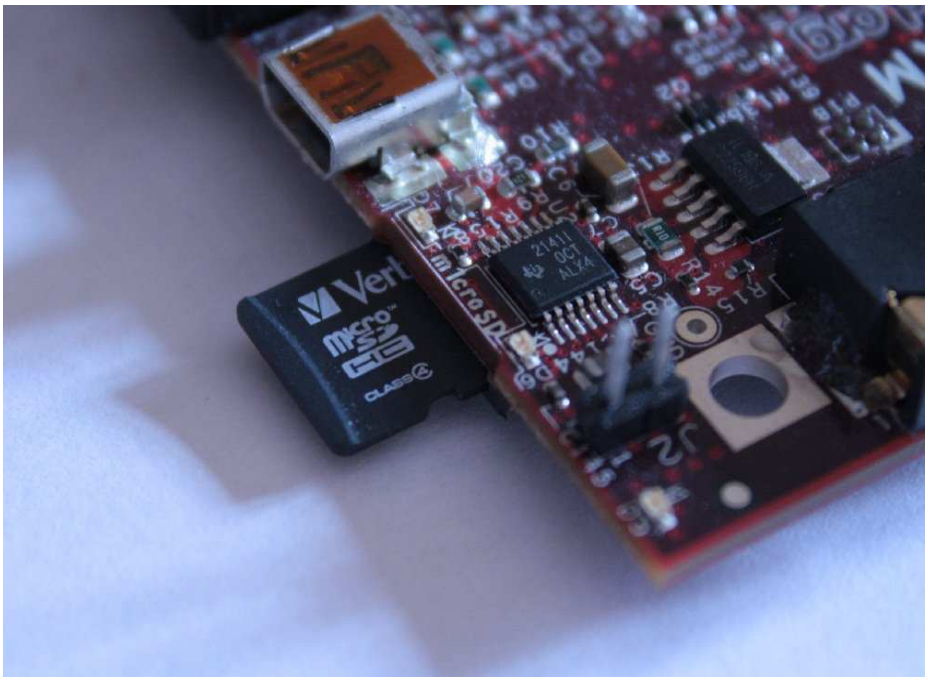
Si la création d'un PC Linux ultra-compact est une évidence, d'autres utilisations sont possibles : plate-forme pour des clients légers ou des bornes interactives, base pour des systèmes de vidéoconférence ou de voix sur IP autonomes, carte graphique externe USB pour les ordinateurs portables, analyseur de trafic USB, *etc.*



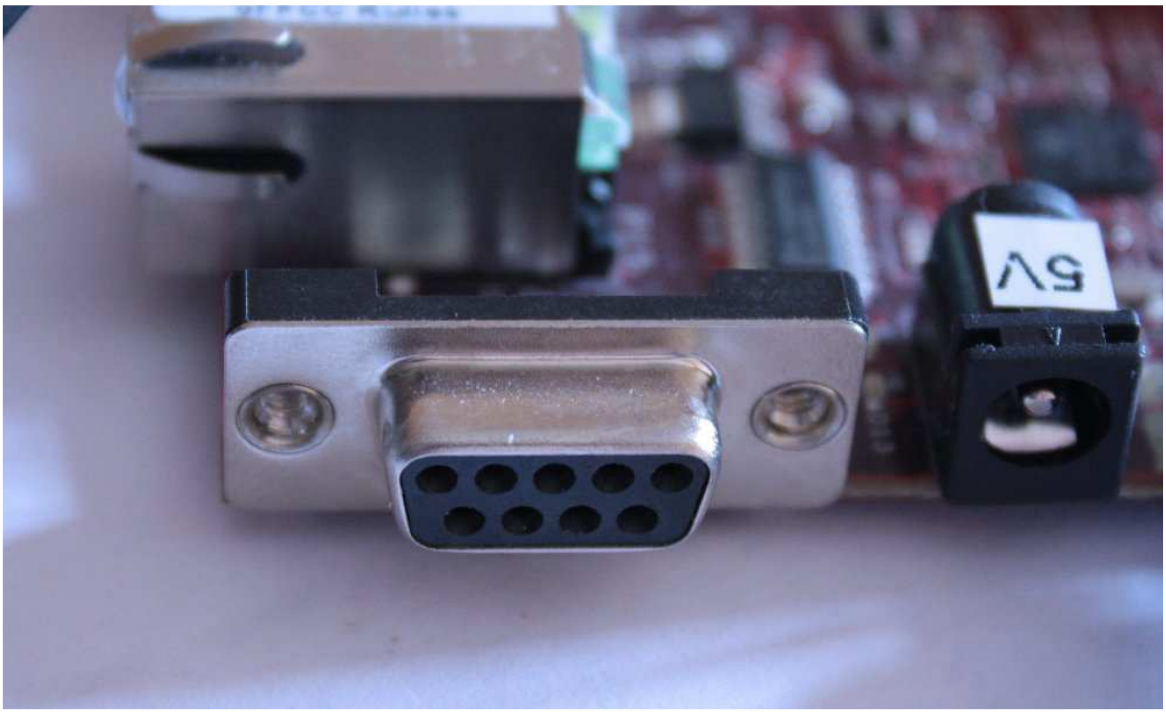
Un port d'extension est disponible, tout comme quatres connecteurs USB 2.0 et un lecteur de cartes au format SD. Ce dernier est compatible aussi bien avec les cartes mémoire que les produits réseau (Wifi, Bluetooth) ou les modules d'extension tiers (GPS, *webcam*…).

2) Câblage de la carte et mise en œuvre logicielle

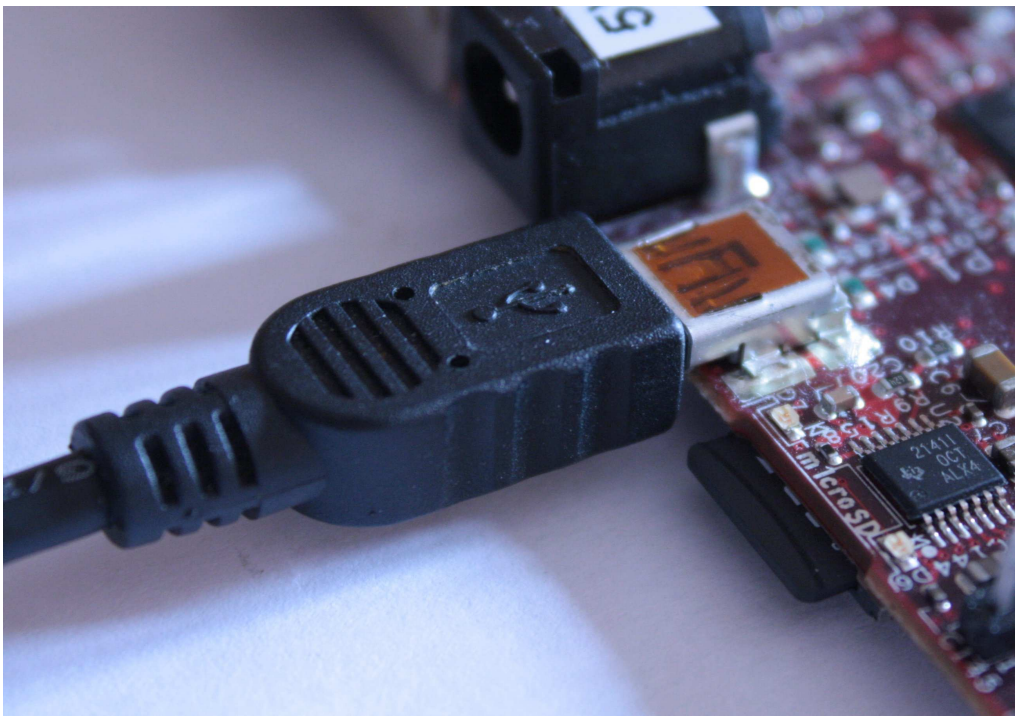
Insérez la carte microSD dans son lecteur. La carte possède le système d'exploitation temps réel QNX et ses applicatifs.



Branchez la prise série sur le connecteur DB9 de la beagleboard et l'ordinateur



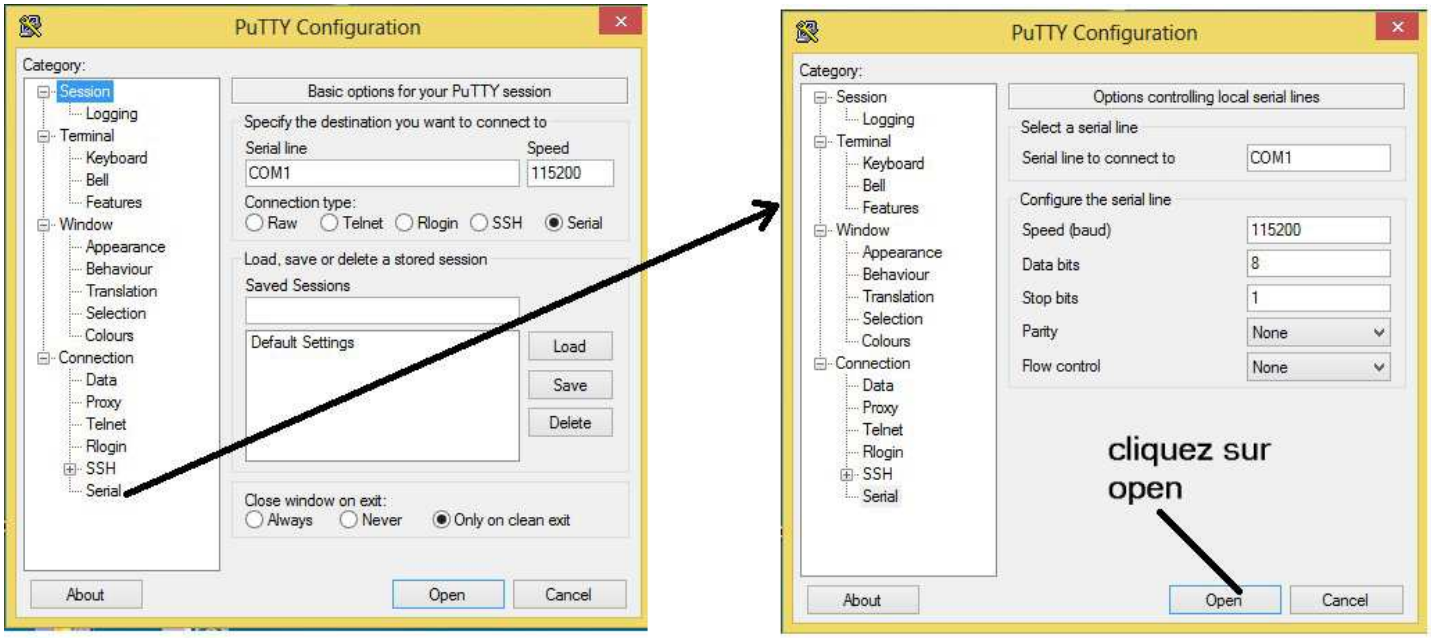
Alimentez la carte avec un cordon USB micro-sd à double prise USBA. La partie alimentation de l'USBA sera branché sur l'ordinateur. Si vous branchez l'autre prise USBA la partie data le système d'exploitation ne se lance pas lors du chargement des pilotes USB.



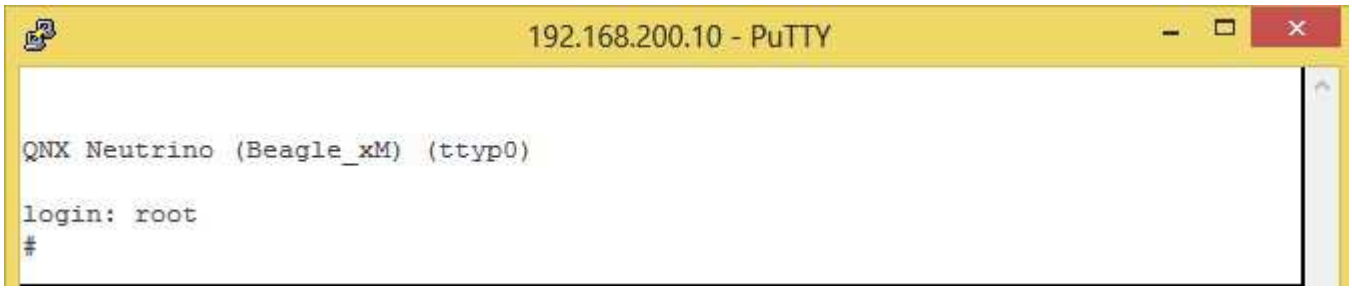
Lancez l'utilitaire putty.exe sur l'ordinateur



Configurez la liaison série de la façon suivante :



La console suivante apparait :



Le prompt # nous indique que l'on s'est loggé en mode administrateur.
La première chose est de changer la configuration réseau, tapez la commande suivante pour voir notre config réseau
nicinfo

```
# nicinfo
en0:
  SMSC9500 Ethernet Controller

  Physical Node ID ........................... 000000 000000
  Current Physical Node ID ................... 02800F 117202
  Current Operation Rate ..................... 100.00 Mb/s full-duplex
  Active Interface Type ...................... MII
    Active PHY address ....................... 1
  Maximum Transmittable data Unit ............ 1514
  Maximum Receivable data Unit ............... 1514
  Promiscuous Mode ........................... Off
  Multicast Support .......................... Enabled

  Packets Transmitted OK ..................... 28
  Bytes Transmitted OK ....................... 1618
  Broadcast Packets Transmitted OK ........... 1
  Multicast Packets Transmitted OK ........... 0

  Packets Received OK ........................ 177
  Bytes Received OK .......................... 16234
  Broadcast Packets Received OK .............. 136
  Multicast Packets Received OK .............. 0
  Memory Allocation Failures on Receive ...... 0

  Single Collisions on Transmit .............. 0
  Multiple Collisions on Transmit ............ 0
  Deferred Transmits ......................... 0
  Late Collision on Transmit errors .......... 0
  Transmits aborted (excessive collisions) ... 0
  Transmit Underruns ......................... 0
  No Carrier on Transmit ..................... 0
  Receive Alignment errors ................... 0
  Received packets with CRC errors ........... 0
  Packets Dropped on receive ................. 0
  Short packets .............................. 0
  Total Frames experiencing Collison(s) ...... 0

#
```

La commande ifconfig permet d'obtenir l'@ IP de la carte réseau

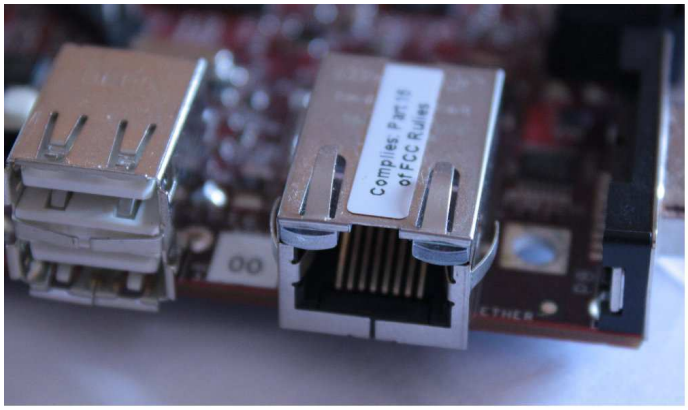
```
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
        inet 127.0.0.1 netmask 0xff000000
en0: flags=80008843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST,SHIM> mtu 1500
        address: 02:80:0f:11:72:02
        media: Ethernet 100baseTX full-duplex
        status: active
        inet 192.168.200.10 netmask 0xffffff00 broadcast 192.168.200.255
#
```

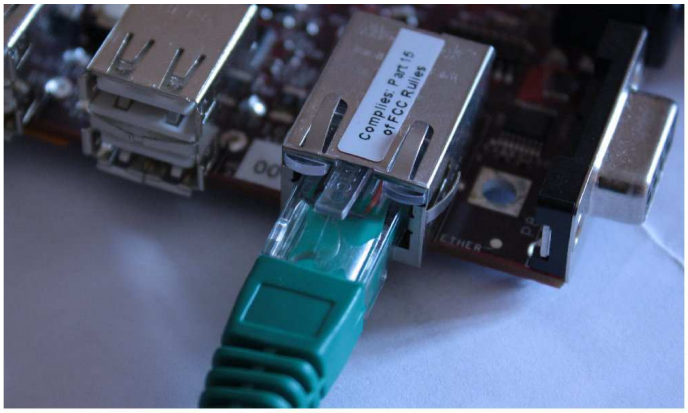
Puis tapez la commande ci-dessous pour changer notre adresse IP et notre masque de sous réseau. En0 est l'interface réseau 192.168.200.10 notre nouvelle adresse IP et netmask 255.255.255.0 le nouveau masque de sous réseau

ifconfig en0 192.168.200.10 netmask 255.255.255.0 up

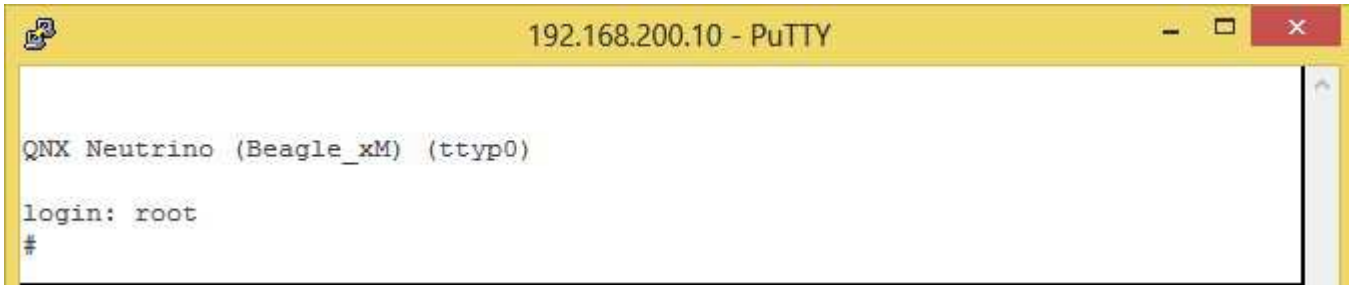
pour rebooter la carte tapez la commande shutdown

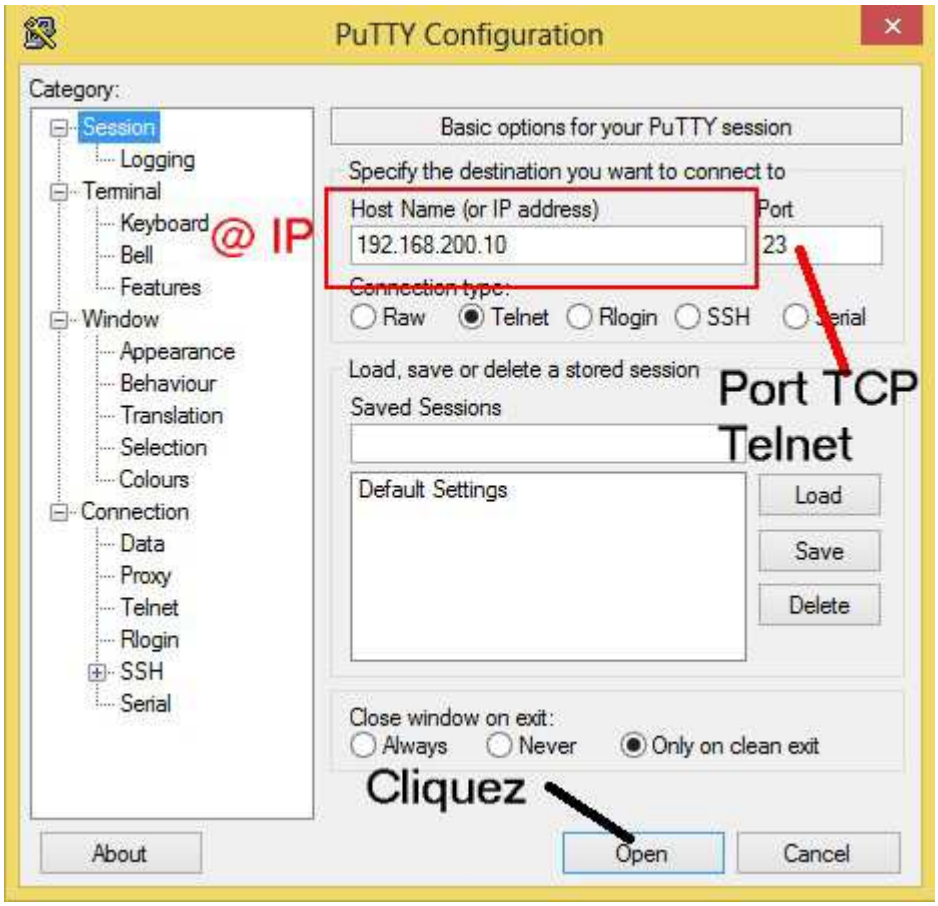
Branchez le câble réseau RJ45 sur la carte et l'ordinateur ou le switche.

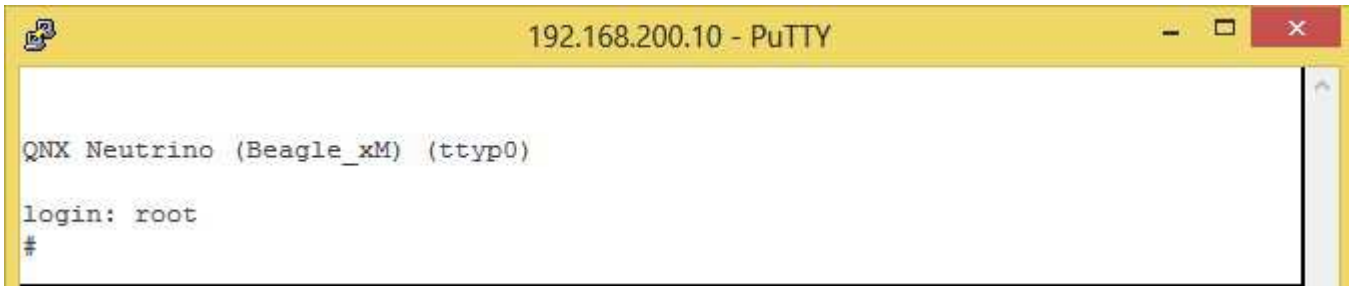




Le système d'exploitation de la carte nous offre différents services réseaux et notamment un serveur telnet
Fermez putty
Relancez putty qui servira de client Telnet afin de se logguer sur la carte. Cliquez sur open et tapez root, la fenêtre console apparait :




PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
Telnet
Rlogin
SSH
Serial
@ IP
Basic options for your PuTTY session
Specify the destination you want to connect to
Host Name (or IP address)
Port
192.168.200.10
23
Connection type:
Raw
Telnet
Rlogin
SSH
Serial
Load, save or delete a stored session
Saved Sessions
Port TCP
Telnet
Default Settings
Load
Save
Delete
Close window on exit:
Always
Never
Only on clean exit
Cliquez
About
Open
Cancel


192.168.200.10 - PuTTY
QNX Neutrino (Beagle_xM) (ttyp0)
login: root
#

Le système d'exploitation QNX

1) Historique

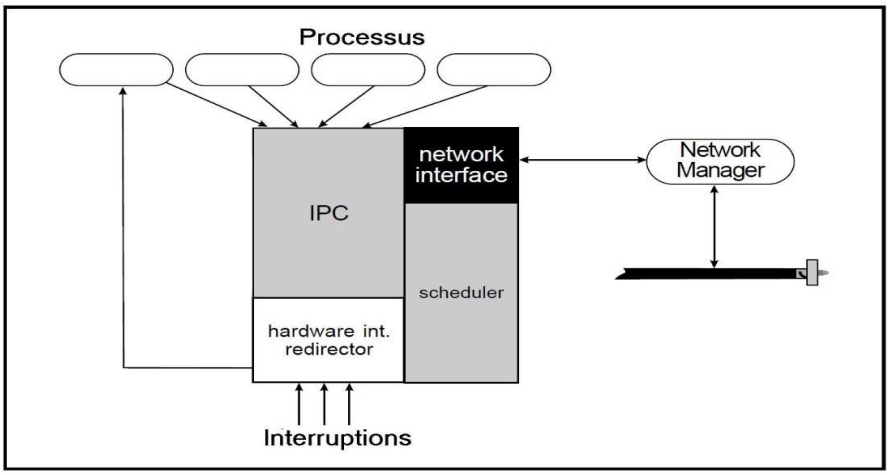
Quantum Unix, nom originel de QNX, a été créé par Dan Dodge et Gordon Bell en 1981. Les premières versions étaient destinées aux processeurs 8808 et 6809. Le nom n'étant pas très original, Quantum Unix deviendra plus tard Quick Unix ou QNX. QNX va alors se doter de fonctionnalités novatrices qui le rapproche des standards du marché, notamment grâce à sa **compatibilité** avec la norme **POSIX**.

2) Présentation

QNX est fondé sur un micro noyau (Microkernel) d'une taille de 10Koctets qui gère un groupe de processus systèmes. Ce noyau de petite taille lui confère fonctionnalité et performance d'un executif temps réel

Le **Microkernel** possède **quatres fonctions** fondamentales :

- ✓ **Transmission de messages** ou **IPC (interprocess communication)** : Le Microkernel manie les messages des processus à travers tout le système.
- ✓ **Gestion des processus** ou **scheduler** : le micro noyau travaille en étroite collaboration avec le gestionnaire de processus. Il est chargé d'attribuer les ressources du système aux processus qui le demande. C'est de cette partie que nous traiterons par la suite.
  Le micro noyau est exécuté suite à un appel système provenant soit d'un autre processus, soit d'une interruption matérielle.
- ✓ **Les interruptions** : toutes les interruptions matérielles sont d'abord soumises au noyau puis passées au driver approprié ou au système.
- ✓ **La communication réseau** de bas niveau, Le noyau délivre tous les messages destinés aux processus distants connectés au réseau.



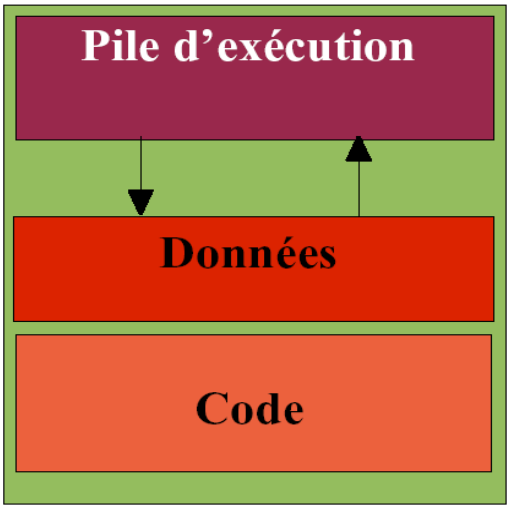
Il y a 14 appels noyaux pour ses différents services.
Le micro noyau est exécuté suite à un appel système provenant soit d'un autre processus, soit d'une interruption matérielle.
A l'exception de ceux fournit par le Microkernel, tous les services de QNX sont gérés par des processus "classiques". Il en est de même pour les quatre processus systèmes suivant :
Gestionnaire de processus (Proc)
Gestionnaire du système de fichier (Fsys)
Gestionnaire de périphériques (Dev)
Gestionnaire de réseau (Net)

Les **processus système** sont pratiquement **identiques aux programmes des utilisateurs** - ils ne possèdent aucune **interface** cachée ou privée qui ne serait pas accessible aux programmes des utilisateurs.
C'est cette architecture qui donne à QNX une très grande flexibilité. Depuis que la plupart des services sont des tâches standards de QNX, il est possible d'augmenter les possibilités du système lui-même en ré-écrivant de nouveaux programmes pour lui donner la possibilité d'offrir de nouveaux services !
Les **drivers** sont eux aussi des **processus standards**. La couche matérielle est donc totalement transparente pour le système. Pour charger un périphérique, il suffit d'exécuter le driver correspondant.

3) Gestion des processus

3.1) Description d'un processus

Un programme est composé d'une ou plusieurs séquences d'instructions, agissant sur un ensemble de données. Un programme est essentiellement statique. Un processus représente l'exécution d'un programme statique. C'est une **entité dynamique** créée en mémoire vive à un instant donné par le système d'exploitation et qui peut disparaître au bout d'un temps fini. De plus un processus peut communiquer avec d'autres processus (IPC Interprocessus communication) . Les processus sont composés d'un espace de travail en mémoire formé de 3 segments :



- **Le code correspond aux instructions, en langage d'assemblage, du programme à exécuter.**
- **La zone de données contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire.**
- **Enfin, les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la pile.**

Un processus est toujours constitué d'une tâche principale (le main) mais qui peut contenir d'autres taches auxiliaires (threads) qui s'exécutent de manière concurrente avec la tâche principale.
Le micronoyau maintient une table pour gérer l'ensemble des processus. Cette table contient la liste de tous les processus avec des informations concernant chaque processus.
L'environnement d'un processus encore appelé *contexte*, est caractérisée par :

- l'état des pointeurs associés à chaque zone mémoire (programme, données, pile).
- des caractéristiques de cette tâche.

Lorsque le processus ou la tâche cesse d'être active, le système d'exploitation effectue une **sauvegarde du contexte** en rangeant toutes ces données **dans la pile du système**. Lorsque la tâche redevient active, **on restaure le contexte** afin de lui redonner la main.

Les tâches sont classées par ordre de priorité dans une file. Ce classement est effectué par **l'ordonnanceur (scheduler)** et destinée au répartiteur **(dispatcher)** organe chargé d'allouer le processeur aux différentes tâches.

3.2) Le mécanisme de gestion des processus

La gestion des processus sous QNX est découpée en deux parties :

- **l'attribution des ressources** aux processus, c'est le rôle du **scheduler**. **la gestion des processus et des ressources** qui leurs ont été attribuées : c'est le rôle du **gestionnaire de processus**. Le scheduler est un service rendu par le micro noyau. Le gestionnaire de processus est un processus système classique.Le **scheduler** et le **gestionnaire de processus** travaillent en étroite collaboration puisqu'ils **partagent leur espace mémoire**.

- Le **gestionnaire de processus** est responsable de la création des nouveaux processus et de la **gestion des ressources attribuées** aux processus. L'ensemble des services du gestionnaire de processus sont demandés grâce à l'**utilisation de messages**. Par exemple, si un processus souhaite créer un processus fils, il envoie un message au gestionnaire de processus en y spécifiant les caractéristiques de celui-ci. Comme l'échange de messages peut se faire à travers le réseau, il est facile de créer un processus sur une machine distante en envoyant un message au gestionnaire de processus de cette machine.

La commande pidin sous une console Telnet QNX, nous permet de lister tous les processus inscrit par le gestionnaire de processus.

```
# pidin
     pid tid name               prio STATE       Blocked
       1   1 boot/procnto-instr   0f READY
       1   2 boot/procnto-instr 255r RECEIVE     1
       1   3 boot/procnto-instr 255r RECEIVE     1
       1   4 boot/procnto-instr  10r RECEIVE     1
       1   5 boot/procnto-instr 255r RECEIVE     1
       1   6 boot/procnto-instr  10r RECEIVE     1
       1   7 boot/procnto-instr  10r RECEIVE     1
       1   8 boot/procnto-instr  10r RECEIVE     1
       1  10 boot/procnto-instr  10r RECEIVE     1
       1  11 boot/procnto-instr  10r RECEIVE     1
       1  12 boot/procnto-instr  10r RECEIVE     1
       1  14 boot/procnto-instr  10r RUNNING
    8194   1 /boot/devc-seromap  10r RECEIVE     1
    8195   1 proc/boot/slogger   10r RECEIVE     1
    8196   1 proc/boot/pipe      10r SIGWAITINFO
    8196   2 proc/boot/pipe      10r RECEIVE     1
    8196   3 proc/boot/pipe      10r RECEIVE     1
    8197   1 proc/boot/random    10r SIGWAITINFO
```

Le **scheduler** quant à lui est chargé de **gérer l'attribution des ressources** du système aux différents processus qui s'exécutent. Il intervient lorsque :

- un processus est prêt à être exécuté
- le temps attribué à un processus est écoulé
- un processus est préempté

3.3) L'état des processus sous QNX

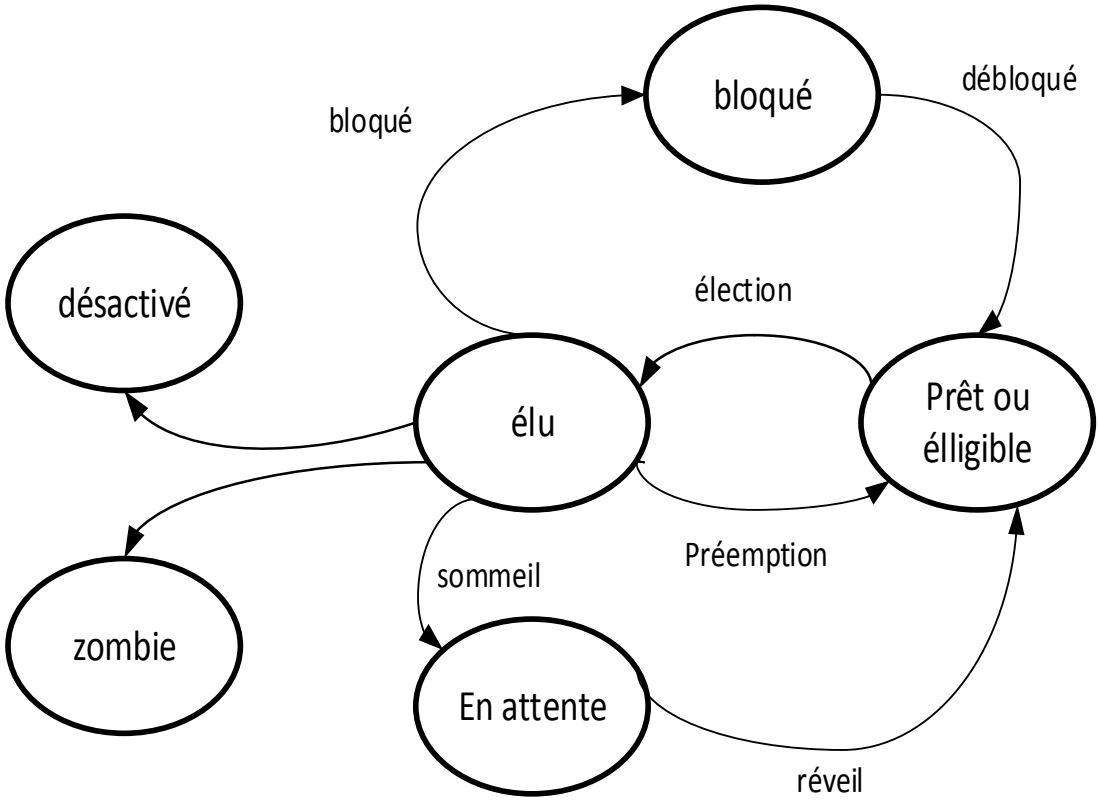
Les processus peuvent se trouver dans les cinq états suivants :

- **prêt** : le processus est en attente, prêt à être exécuté.
- **bloqué** : le processus peut être alors dans les cas suivants :
  bloqué sur un envoi de message
  bloqué sur la réception d'un message
  bloqué sur la réponse à un message
  bloqué par un signal
  bloqué par un sémaphore
- **désactivé** : le processus a reçu un signal SIGSTOP.
- **en attente** : le processus a fait un appel système "wait()" ou "waitpid()".
- **zombie** : le processus a terminé son exécution mais ne peut pas renvoyer sa valeur de retour car le parent est en attente sur un appel système "wait()" ou "waitpid()".

- **élu** (en cours d'exécution) c'est à dire en possession d'un processeur.

- **éligible** (ou prête) : la tâche est alors candidate au processeur. Elle possède toutes les ressources nécessaires à son fonctionnement sauf un processeur. Son lancement ne dépend que de sa priorité par rapport aux tâches en cours d'exécution ou aux tâches prêtes.

- **bloqué** (ou suspendue) : la tâche n'est plus en possession d'un processeur. Elle s'est interrompue avant sa fin sur l'apparition d'une condition de blocage (attente d'une ressource indispensable à son exécution future, mise en sommeil volontaire de la tâche, etc...).



Dans QNX, **chaque processus possède une priorité**. Le scheduler **sélectionne le processus à exécuter en comparant la priorité de chaque processus** se trouvant dans l'état "prêt". Le processus qui possède la **plus haute priorité** sera alors **élu**.
L'ensemble des priorités s'étend de 0 (priorité la plus basse) à 31 (priorité la plus haute). La priorité par défaut d'un processus est héritée du processus parent. Les processus créé par le shell se voient attribué la priorité 10.

3.4) Création d'un processus

Dans des applications embarquées il y a deux approches différentes pour créer un processus :

- Créé un script Shell qui contient les lignes de commande pour commencer un processus
- Créé un processus qui se lance au démarrage et qui démarrent tous les autres processus.

Ils existent sous QNX beaucoup de fonctions qui permettent la création de processus.
Certaines de ses fonctions sont un héritage de l'OS LINUX.
Le composant manager de processus sous QNX s'appelle procnto et il est responsable de la création des processus. Si un processus veut créer un autre processus il fait appel au composant procnto en lui envoyant un message pour la création du processus.

La fonction spawn crée et exécute un nouveau processus fils :

#include <spawn.h>

Déclaration
pid_t spawn(const char * *path*, **int** *fd_count*, **const int** *fd_map[ ]*, **const struct inheritance** ***** *inherit*, **char * const** *argv[ ]*, **char * const** *envp[ ]* **);**

path *: Chaine de caractère qui correspond au chemin de l'exécutable à associer à un processus fils*
fd_count*=0*
fd_map*=NULL*
const struct inheritance * *inherit* Un pointeur sur une structure **`inheritance`** qui indique ce que le processus fil doit hérité du processus père.
char * const *argv[ ]* Pointeur sur un tableau de chaine de caractère qui contient les paramètres en ligne de commande que l'on souhaite faire passer au processus fils
char * const *envp[ ]* Pointeur sur un tableau de chaine de caractère qui contient les paramètres des variables d'environnement du RTOS
pid_t variable du type entier qui contient le numéro d'identification du processus fils qui a été créé

```
#include <spawn.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
main(int argc, char **argv)
{
char *args[] = { "child", NULL };// Processus fils à créer dans le même réperoire
int i, status;
pid_t pid;
struct inheritance inherit;
// create 3 child processes
for (i = 0; i < 3; i++) {
inherit.flags = 0;
if ((pid = spawn("child", 0, NULL, &inherit, args, environ)) == -1)// création processus fils
perror("spawn() failed");
else
printf("spawned child, pid = %d\n", pid);
}

/*
* waitchild.c
*
* This is an example of a parent process that creates some child
* processes and then waits for them to terminate. The waiting is
* done using wait(). When a child process terminates, the
* wait() function returns.
*/
#include <stdio.h>
#include <unistd.h>
main(int argc, char **argv)
{
printf("pausing, terminate me somehow\n");
pause();
}
```

4) Les techniques d'ordonnancement des processus

Nous avons vu que le système exécute le processus prêt qui possède la priorité la plus forte. **L'ordonnancement intervient lorsque plusieurs processus qui possèdent la même priorité sont à l'état prêt.**
QNX propose trois méthodes différentes pour l'élection des processus, on parle d'ordonnancement :

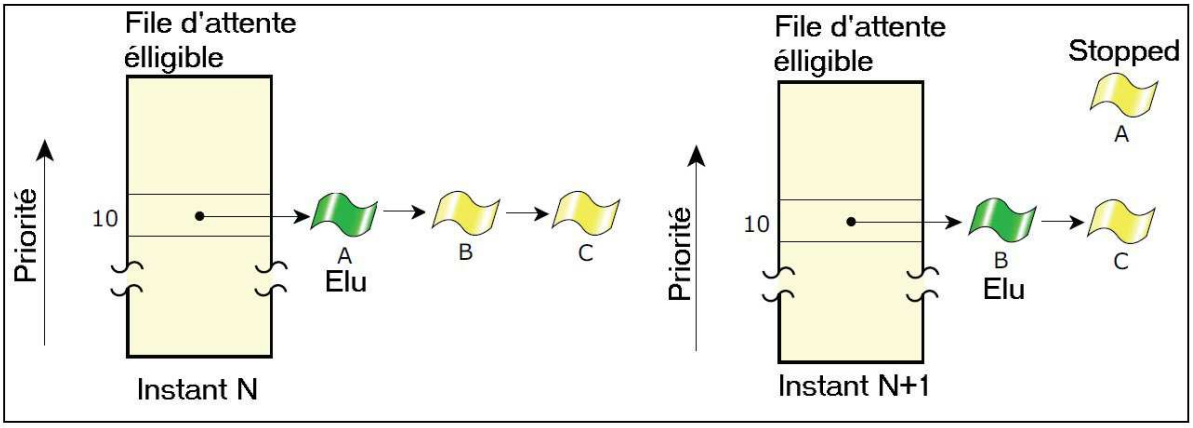
- ✓ **Ordonnancement FIFO**
- ✓ **Ordonnancement Round Robin**
- ✓ **Ordonnancement Adaptatif**

Tous les processus ou les taches dans le système utilise une de ces trois méthodes, mais deux processus différents peuvent utiliser une méthode différente.

4.1) L'ordonnancement FIFO (First In First Out)

Dans le cas de l'ordonnancement FIFO, un processus s'exécute jusqu'à ce qu'il :

- ✓ **libère** le processus (il se termine ou il fait un appel système, par exemple)
- ✓ soit préempté par un processus de priorité supérieure

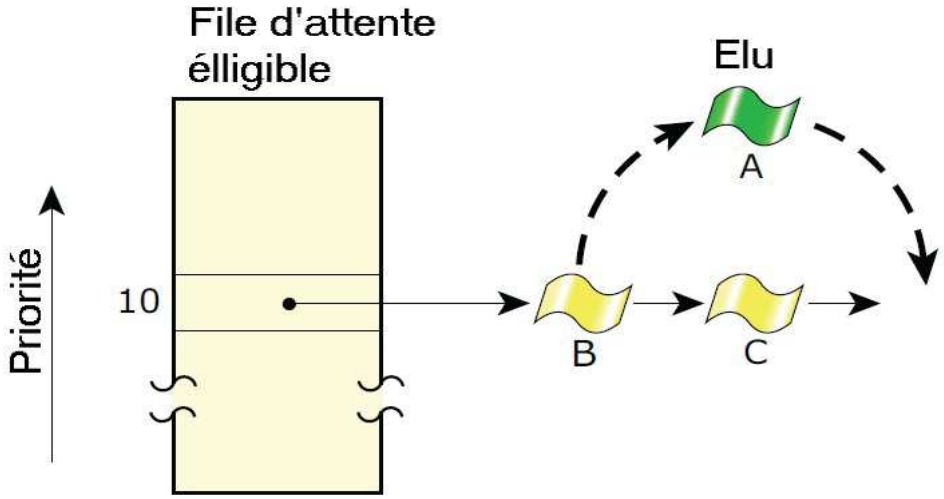


Deux processus qui ont la même priorité peuvent utiliser le FIFO scheduling pour assurer une exclusion mutuelle lors de l'utilisation d'une ressource partagée. Ces deux process ne pourront se déranger mutuellement par une action de préemption alors qu'ils seront en train de s'exécuter. Par exemple s'ils partagent une partie de la mémoire, chacun de ces deux processus pourrons mettre à jour la mémoire sans avoir à utiliser une quelconque forme de sémaphores.

4.2) L'Ordonnancement Round Robin

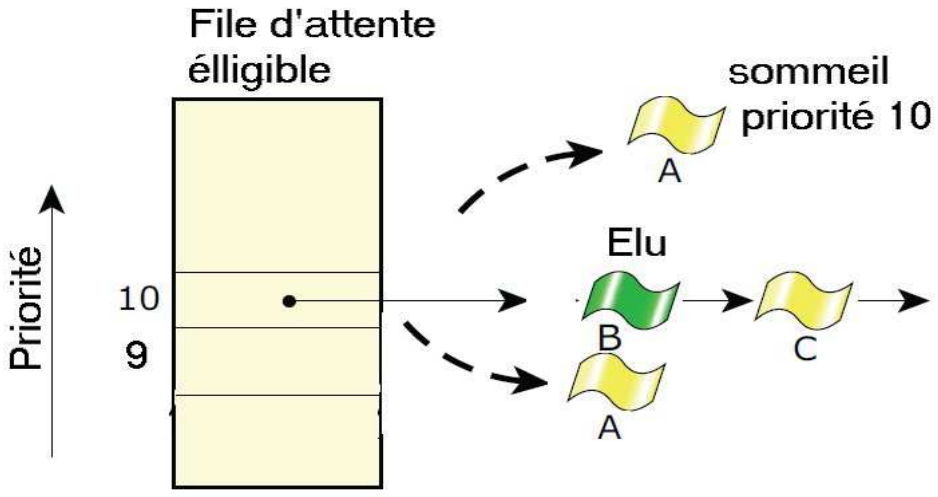
Dans le principe du round-robin, un processus s'exécute jusqu'à ce qu'il :

- ✓ il abandonne volontairement les ressources processeurs
- ✓ il soit preempté par une priorité plus haute
- ✓ il utilise tout son timeslice (durée d'attribution du processeur) sous le contrôle du scheduler



4.3) L'Ordonnancement Adaptatif

Dans le scheduling adaptif un processus se comporte ainsi :
Si un processus consomme tout son timeslice (donc s'il ne se bloque pas), sa priorité est réduite de 1. On appelle cela la dégénérescence de priorité.
Il est à noter que même si sa priorité est abaissée de 1, un process peut encore avoir la priorité la plus haute et donc de nouveau avoir le CPU.
Si un processus bloque, il retour immédiatement à sa priorité originelle ou initiale.



Vous pouvez utiliser un scheduling adaptif dans les environnements possédant un grand nombre de processus en fond. Cette méthode donne un accès suffisant au CPU tout en donnant souvent "la parole" aux autres processus.

5) Création d'une tache sous QNX

Un processus lourd est un programme en cours d'exécution. Chaque processus lourd implique la gestion d'un espace d'adressage virtuel MMU et de nouvelles copies de toutes les variables et ressources nécessaires à l'exécution (pile, registres, fichiers ouverts, verrous etc…).
Le principal avantage de ces processus est la protection mémoire entre les processus du même système.
Par contre, l'utilisation des processus lourds présente les inconvénients suivants :

- leur création nécessite des appels systèmes coûteux en temps,
- le changement de contexte entre processus est une opération lente, en particulier pour de nombreux transferts en mémoire,
- le coût des mécanismes de protection associés au processus,
- l'interaction, la synchronisation ou la communication entre processus nécessite l'utilisation de mécanismes de communication spéciaux (tube communicant appelé "*pipe*", *socket*, boîte aux lettres),
- le partage de mémoire entre processus s'effectue par ajout de mécanismes lourds (bibliothèque de partage de mémoire).

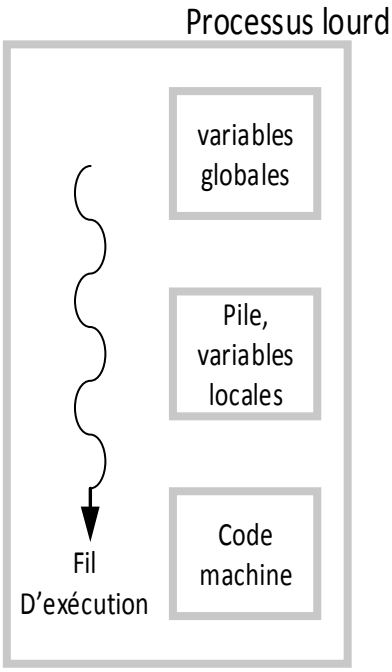
La liste des processus lourd est donnée par le gestionnaire des taches :

```
# pidin
     pid tid name               prio STATE       Blocked
       1   1 boot/procnto-instr   0f READY
       1   2 boot/procnto-instr 255r RECEIVE     1
       1   3 boot/procnto-instr 255r RECEIVE     1
       1   4 boot/procnto-instr  10r RECEIVE     1
       1   5 boot/procnto-instr 255r RECEIVE     1
       1   8 boot/procnto-instr  10r RECEIVE     1
       1  10 boot/procnto-instr  10r RECEIVE     1
       1  11 boot/procnto-instr  10r RECEIVE     1
       1  14 boot/procnto-instr  10r RUNNING
       1  15 boot/procnto-instr  10r RECEIVE     1
       1  17 boot/procnto-instr  10r RECEIVE     1
       1  18 boot/procnto-instr  10r RECEIVE     1
    8194   1 /boot/devc-seromap  10r RECEIVE     1
    8195   1 proc/boot/slogger   10r RECEIVE     1
    8196   1 proc/boot/pipe      10r SIGWAITINFO
    8196   2 proc/boot/pipe      10r RECEIVE     1
    8196   3 proc/boot/pipe      10r RECEIVE     1
```

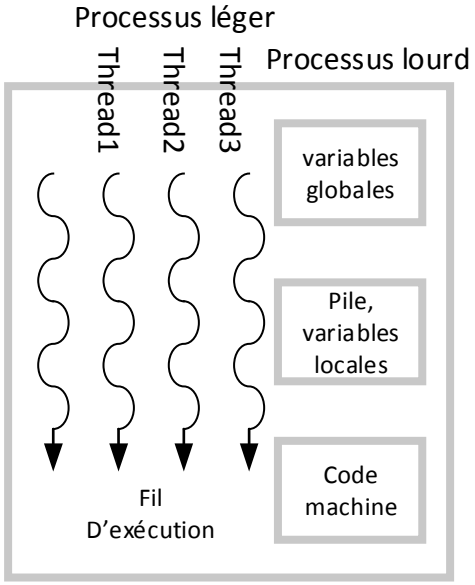
pidin -F "%I %60N %b" // I (PID et TID) b nbre de thread par processus, (d'autres options sont possibles)

```
# pidin -F "%I %60N %b"
pid-tid     name                                                    tid
      1-01 proc/boot/procnto-instr                                   1
      1-02 proc/boot/procnto-instr                                   2
      1-03 proc/boot/procnto-instr                                   3
      1-04 proc/boot/procnto-instr                                   4
      1-05 proc/boot/procnto-instr                                   5
      1-08 proc/boot/procnto-instr                                   8
      1-10 proc/boot/procnto-instr                                  10
      1-11 proc/boot/procnto-instr                                  11
      1-14 proc/boot/procnto-instr                                  14
      1-15 proc/boot/procnto-instr                                  15
      1-17 proc/boot/procnto-instr                                  17
      1-18 proc/boot/procnto-instr                                  18
   8194-01 proc/boot/devc-seromap                                    1
   8195-01 proc/boot/slogger                                         1
   8196-01 proc/boot/pipe                                            1
   8196-02 proc/boot/pipe                                            2
   8196-03 proc/boot/pipe                                            3
```

Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit **monoprogrammé** :
L'exécution du code du processus est réalisée de manière séquentielle par un fil de contrôle (*thread of control*), le programme est monolithique (main+fonctions) .



Les threads ou processus léger permettent de dérouler plusieurs blocs d'instructions, en PARALLELE, à l'intérieur du même processus. Un thread exécute une fonction. Chaque fonction associée à un thread va être exécutée de façon parallèle et indépendante.



Quand tu commences un processus, tu commences automatiquement un thread ou tache. Ce thread ou tâche est appelé la tache main()
Le système d'exploitation gère donc des centaines de thread :

Dans un système monoprocesseur l'illusion sera donnée que chaque thread est exécuté de façon parallèle ou concurrente. En réalité chaque thread sera exécuté à tour de rôle mais très rapidement.

Avantages :

- **Multi-tâche moins coûteux** : puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux threads est moins coûteuse que la commutation de contexte de processus lourd
- **Communication entre threads plus rapide et plus efficace** : grâce au partage de certaines ressources entre threads, IPC (*Inter Processus Communication*) inutile pour les threads
- **Inconvénients** :

Programmation utilisant des threads est toutefois plus difficile : obligation de mettre en place des **mécanismes de synchronisation**.

Création de la tache

#include <pthread.h>
int pthread_create(pthread_t* *thread*, **const pthread_attr_t*** *attr*, **void***
(***start_routine*****)(void*),** **void*** *arg* **);**

pthread_t* *thread : Pointeur sur la variable du type tache*
const pthread_attr_t* *attr NULL par défaut*
void* (***start_routine*****)(void*)** Pointeur sur la routine (flux d'exécution de la tache)
void* *arg NULL par défaut*

Joint une tache

#include <pthread.h>
int pthread_join(pthread_t *thread*, **void**** *value_ptr* **);**

pthread_t *thread* variable du type tache
void** *value_ptr* NULL par défaut

La fonction *pthread_join()* bloque l'appel des autres threads jusqu'à ce que le thread cible soit terminée

Retourne l'identifiant du thread

#include <pthread.h>
pthread_t pthread_self(void);

Cette fonction retourne l'identifiant de la tâche appelée.

Exemples

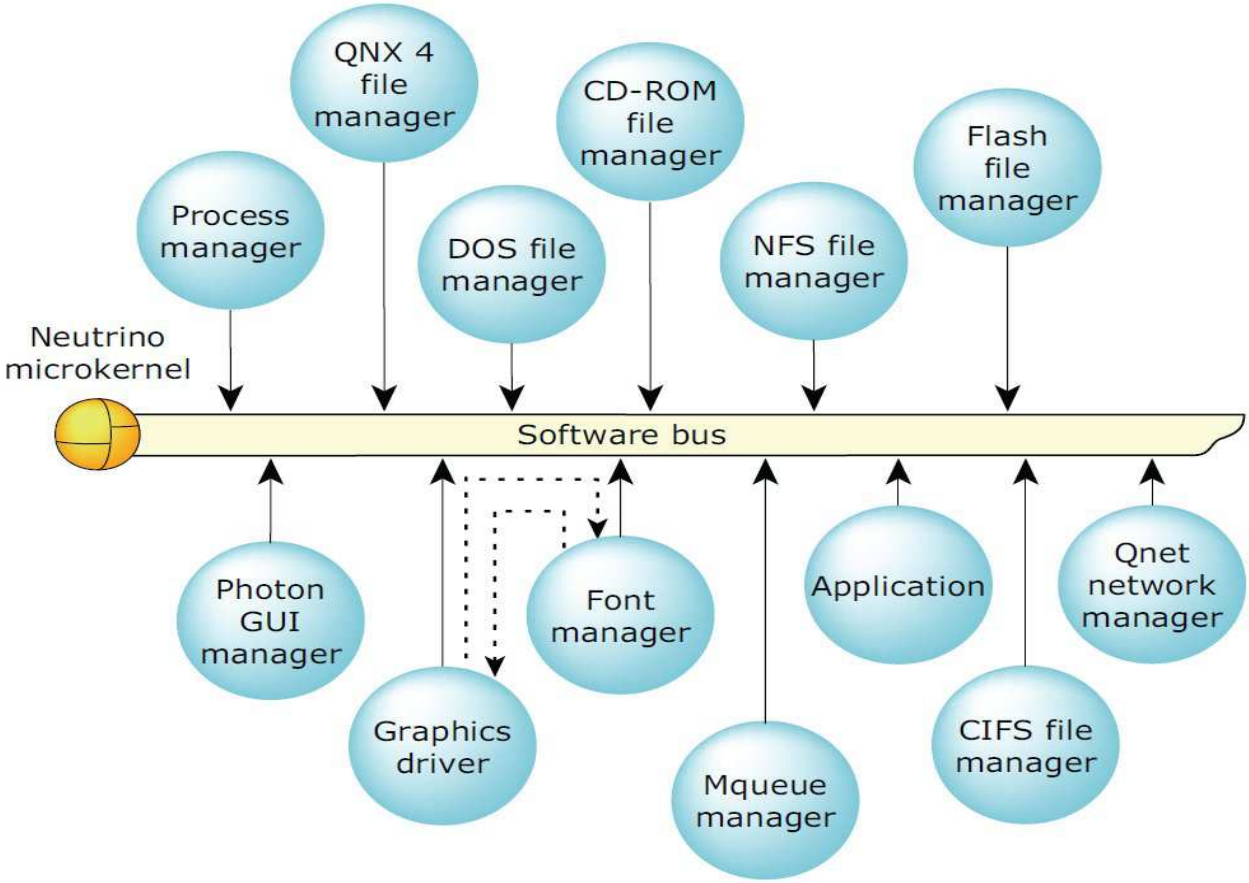
```
# include <unistd.h>
# include <pthread.h>
#include <stdlib.h>
# include <stdio.h>

 int valeur = 0;
void main(void)
{
pthread_t tid1, tid2;// déclaration de 2 thread
void fonc(void);
//pthread_set_concurrency(2);
pthread_create(&tid1, NULL, (void *(*)()) fonc, NULL); //creation du thread 1
pthread_create(&tid2, NULL, (void *(*)()) fonc,NULL); //creation du thread 2
pthread_join(tid1, NULL);// attendre que le thread1 soit terminé
pthread_join(tid2, NULL);
printf("valeur = %d\n", valeur);
}

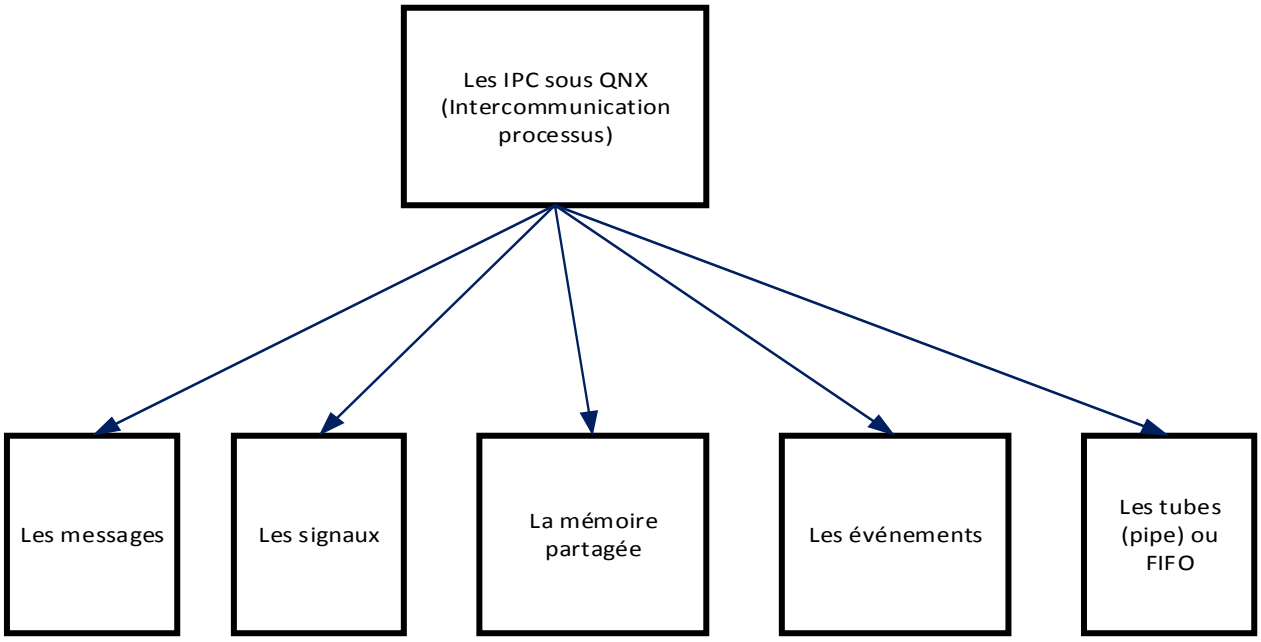

void fonc(void)
{
int i;
pthread_t tid;
for (i=0; i <1000000; i++) valeur = valeur + 1;
tid = pthread_self();// renvoie le thread identifiant
printf("tid : %d valeur = %d\n", tid, valeur);
}
```

6) communication inter-processus

L'architecture de l'OS appelé Neutrino est constituée d'un micronoyau et de beaucoup de processus coopérants. Ces processus communiquent entre eux sous différentes formes de communication interprocessus (IPC). Le passage de message est la forme primaire des IPC dans Neutrino.



Ils existent une multitude de mécanisme permettant la communication inter-processus



Puisque QNX Neutrino gère les threads globalement, la communication entre deux threads situées dans deux processus distincts est alors une communication inter-processus. La communication inter-processus peut être **synchrone** ou **asynchrone**, **bloquante** ou **non-bloquante**.

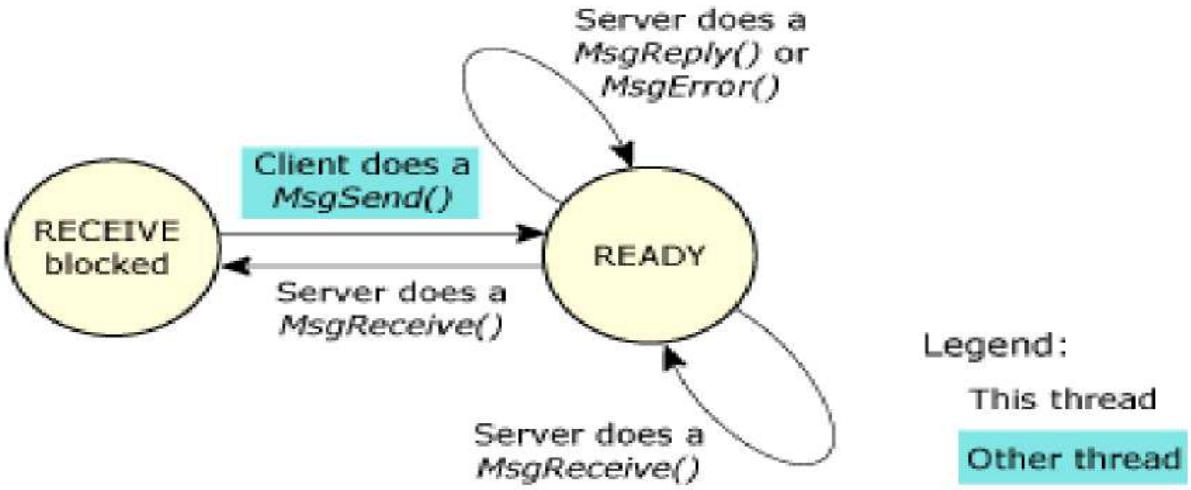
mécanismes de communication	implémenté dans	utilisé par	**POSIX**
passage de messages	Noyaux	Processes	NO
Signals	Noyaux	Processes/Threads	YES
tubes	processus externes	Processes/Threads	YES
FIFOs	processus externes	Processes/Threads	YES
Message Queues	processus externes	Threads	YES
Mémoires partagées	manageur de processus	Processes/Threads	YES

La communication est synchrone lorsque le message ne peut être reçu par le récepteur qu'à un moment précis: lorsque celui-ci se place en attente d'un message (état RECEIVE). La communication est asynchrone lorsque le récepteur peut être interrompu à tout moment par un message. La communication est bloquante lorsque l'émetteur doit attendre une réponse du récepteur (états SEND et REPLY). Elle est non-bloquante si l'émetteur peut continuer son exécution sans se soucier du récepteur.

6.1) La messagerie de QNX Neutrino

La messagerie de QNX Neutrino est le système de communication inter-processus de base directement programmé dans le *micronoyau*.
C'est à la base une communication synchrone et bloquante, Voici son fonctionnement :

Un thread émetteur envoie un message synchrone à un thread récepteur grâce à la fonction *MsgSend*. Le thread émetteur bloquera (état SEND) jusqu'à ce que le thread récepteur appelle la fonction *MsgReceive*. Le thread émetteur passera alors de l'état SEND à l'état REPLY ce qui indique que le thread attend une réponse. Le thread récepteur traite le message et envoie une réponse à l'aide de la fonction *MsgReply*. Ce qui aura pour effet de débloquer le thread émetteur. Si le thread récepteur appelle la fonction *MsgReply* avant que le thread émetteur n'envoie le message, elle bloquera (état RECEIVE). La fonction *MsgReply* permet de retourner un code erreur et un message tandis que la fonction *MsgError* ne retourne qu'un code d'erreur. Voici les différents états par lesquels passent le thread émetteur et le thread récepteur:



Les messages sont copiés de la zone mémoire du thread émetteur vers la zone mémoire du thread récepteur. Il n'y a pas de temporisation. La vitesse de communication est donc très proche de la vitesse de la mémoire vive en autant que les threads résident sur les mêmes postes de travail.
Avant de pouvoir recevoir des messages, un thread récepteur doit créer un canal de communication grâce à la fonction *ChannelCreate*. Lorsque la communication est terminée le récepteur doit détruire le canal avec la fonction *ChannelDestroy*.
Avant de pouvoir envoyer des messages, un thread émetteur doit se connecter à un canal de communication avec la fonction *ConnectAttach*. Lorsque la communication est terminée l'émetteur doit se déconnecter du canal avec la fonction *ConnectDetach*.

Les fonctions

```
#include <sys/neutrino.h>
int ChannelCreate( unsigned flags );
```

Le flag peut être utilisé pour des requêtes de notification de signaux à partir du noyau
Cette fonction renvoie un identifiant de canal sous la forme d'un entier.

L'appel noyau *ChannelCreate() créé un canal qui peut être utilisé pour recevoir des messages ou des signaux*
Les threads communiqueront avec le canal attaché en appelant la fonction *ConnectAttach()*. Les threads peuvent être dans le même processus ou dans un autre processus sur le même nœud (ou dans un nœud distant si le manageur de réseau est démarré)

```
#include <sys/neutrino.h>
#include <sys/netmgr.h>
int ConnectAttach( uint32_t nd, pid_t pid, int chid, unsigned index, int flags );
```

nd : Le descripteur de nœud par defaut NULL
pid : L'identifiant du processus du créateur de canal. Si le PID est nul le processus appelant est concerné
chid : L'identifiant de canal retourné par *ChannelCreate()*
index : Le plus petit identifiant de connexion que l'on peut accepter.
Flags : par défaut null

L'appel noyau *ConnectAttach()* établit une connexion entre le processus appelant et le canal spécifié par la variable chid lancé par le processus repéré par son pid et son nd.
Si une erreur a lieu la fonction retourne -1

```
#include <sys/neutrino.h>
int MsgSend( int coid, const void* smsg, int sbytes, void* rmsg, int rbytes );
```

coid : l'identifiant du canal pour envoyer le message sur lequel tu as établi une connexion en appelant ConnectAttach()
smsg : un pointeur sur un buffer qui contient le message que tu veux envoyer
sbytes : le nombre d'octets à envoyer
rmsg : un pointeur sur un buffer qui contient le réponse que tu recevras
rbytes :dimension du buffer de réponse en octet

Le thread d'émission est bloqué en attente d'une réponse.

The sending thread becomes blocked waiting for a reply.

```
#include <sys/neutrino.h>
int MsgReceive( int chid, void * msg, int bytes, struct _msg_info * info );
```

chid : L'identifiant de canal que tu as établi par *ChannelCreate()*
msg : un pointeur sur un buffer ou la fonction peut stocker les données recues
byte : la dimension du buffer précédent
info : par défaut NULL, ou un pointeur sur la structure _msg_info ou la fonction peut stocker des informations additionnelles à propos du message

Le thread résultant de cette fonction entre dans un état bloqué jusqu'à ce qu'un message arrive.

```
#include <sys/neutrino.h>
int MsgReply( int rcvid, int status, const void* msg, int size );
```

rcvid : l'identifiant de réception que la fonction MsgReceive*() a retourné quand elle a reçu le message
status : l'état à utiliser pour le déblocage de l'appel MsgSend*() dans le thread rcvid
msg : un pointeur sur un buffer qui contient le message que tu veux répondre.
Size : la dimension du message en octets

Retourne -1 en cas d'échec

Exemple :

```
#include <sys/types.h>
#include <process.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <sched.h>
#include <time.h>
#include <sys/neutrino.h>
#include <errno.h>



int main()  {


    struct timespec horloge;
    struct _msg_info info;
    int chid,rcvid,coid;
    int itemp,i;
    char tab[]="bonjour";
    char tab1[]="caijour";
     pid_t processusFils=0; //recoit l'heure et l'affiche  --serveur
    pid_t processusPere=0; //envoie l'heure a intervalle régulier --client
```

```
    processusPere=getpid();
    setprio(getpid(),10);
    if( (chid = ChannelCreate(1))== -1 ) {
            perror( "channel" );
            return EXIT_FAILURE;
            };

    if( (processusFils = fork()) == -1 ) {
            perror( "fork" );
            return EXIT_FAILURE;
            };


    if (processusFils) {

       for(i=0;i<10;i++) {
          rcvid=MsgReceive(chid,tab1,sizeof(tab1),&info); //on recoit un message
          printf("On a recu un message\n");
          printf("nombre de secondes ecoulees depuis le debut du programme : %s \n",tab1);
          MsgReply(rcvid,EOK,NULL,0); // on renvoie un message vide
       }
    }

    if (processusPere) {
          //wait(NULL);
       coid = ConnectAttach(0,processusPere,chid,0,0);
          if (coid==-1) {
          printf("erreur");
          exit(EXIT_FAILURE);
       }
       while (1) {

          if (MsgSend(coid,tab,sizeof(tab),NULL,0)) {
             printf("Erreur durant l'envoie du message\n");
          }
       }
    }
    return 0;
}
```

Les messages bloquants vs non-bloquants

Dans certaines situations, on ne veut pas que le thread émetteur attend une réponse du récepteur. Pour ce faire elle doit envoyer une pulse en utilisant la fonction *MsgSendPulse*. Une pulse ne peut contenir qu'un code sur 8 bits et un message sur 32 bits. Le thread récepteur peut recevoir un pulse à même la fonction *MsgReceive*. La fonction *MsgReceivePulse* est spécialement conçue pour ne recevoir que des pulses.

Pour une utilisation robuste des messages

Pour éviter que deux threads ou plus entrent en inter blocage, il faut:

• établir une hiérarchie de threads, les threads de plus haut niveau exécutant des tâches pour les threads de plus bas niveau;
• envoyer des messages bloquants aux threads de niveaux supérieurs seulement - les messages contenant la description de la tâche à effectuer;
• envoyer des messages non-bloquants (pulse) pour communiquer avec des threads de niveau inférieur - le pulse contenant le résultat de la tâche exécutée.

Si vous respectez ces quelques règles simples, aucun inter blocage (du moins causé par les messages) n'est possible.

6.2) la mémoire partagée

Lorsqu'une grande quantité d'information doit être transférée d'un processus à un autre, le système de messagerie de base de QNX ne suffit plus. Nous pourrions dans ce cas utiliser le système de fichiers mais pour des raisons de performance cette solution est rarement viable. Il faut plutôt placer les données en mémoire vive et demander au système d'exploitation de créer une zone de mémoire partagée.
Des multiples threads à l'intérieur d'un process partagent la mémoire de ce process. Pour partager la mémoire entre les process, tu dois premièrement créé une région de mémoire partagée et alors mappé cette région dans ton espace d'adresse des process.
Pour partager de la mémoire, il faut qu'un processus crée un segment de mémoire partagée ainsi que les structures de contrôle associées. Cette opération est assurée par la primitive `shm_open`

Dans un premier temps, il faut ouvrir (ou créer) une zone de mémoire partagée à l'aide de *shm_open*:

```
fd = shm_open("/nomdelazone", O_RDWR|O_CREAT, 0777);
```

Ensuite, il faut donner la taille voulue à la zone mémoire à l'aide de *ftruncate*:

```
ftruncate( fd, taille );
```

une partie de la mémoire du processus à la zone mémoire partagée à l'aide de *mmap*:

```
addr = mmap(0,taille,PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

Ensuite on peut utiliser librement la mémoire pointée par la variable *addr*:

```
*addr = 1;
```

Lorsqu'on n'a plus besoin de la zone de mémoire partagée, on ferme la zone en utilisant *close*:

```
close(fd);
```

Finalement, on détruit la zone mémoire en utilisant *shm_unlink*:

```
shm_unlink("/nomdelazone");
```

Évidemment, la zone ne sera détruite que si elle est inutilisée. Contrairement au système de messagerie de base de QNX, une mémoire partagée ne permet pas de communiquer entre des postes distants.

Les fonctions

```
#include <fcntl.h>
#include <sys/mman.h>
int shm_open( const char * name, int oflag, mode_t mode );
```

Cette fonction ouvre un segment de mémoire partagée

Name : le nom de l'objet mémoire partagée que tu veux ouvrir
Oflag : définit les autorisations d'accès dans cette mémoire partagée
Mode : définit le niveau d'accès pour accéder à la mémoire partagé (chmod) d'un processus

Cette fonction renvoie le descripteur de cette mémoire partagée

```
#include <unistd.h>
int ftruncate( int fildes, off_t length );
```

Cette fonction tronque un fichier

int *fildes :* descripteur de la mémoire partagée
off_t *length : La dimension que tu souhaites avoir pour le fichier en octet*

Cette fonction renvoie -1 en cas d'erreur

```
#include <sys/mman.h>
void * mmap( void * addr, size_t len, int prot, int flags, int fildes, off_t off );
```

Map une région mémoire dans un espace d'adresse d'un processus

Addr : par défaut NULL ou une adresse sur la région mémoire partagée que tu souhaites exploiter pour tes processus
len : Le nombre d'octets de cet espace mémoire partagée
flags : spécifie le type d'accès à cette zone mémoire
fildes : Le descripteur de la mémoire partagée
off : offset ou tu souhaites démarrer dans la mémoire partagée

```
#include <sys/mman.h>
int shm_unlink( const char * name );
```

Efface un objet de la mémoire partagée

Name : le nom de l'objet mémoire partagée que tu veux effacer

```
#include <sys/mman.h>
int munmap( void * addr, size_t len );Détache les adresses mappées précédemment
```

addr le début de la gamme d'adresse que tu veux
len : La longueur de la gamme d'adresse en octet

Exemples :

Processus dit client

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <semaphore.h>
#include "sc.h"

SEMAPHORE *SEM;
SHMdata         *addr;// 2 octets pour la zone de mémoire data et cmd

 void main(int argc, char *argv[])

{          int fd;
int counter;
int nval;
fd = shm_open(SHMname, O_RDWR, 0777);
if(fd<0) serror("can't open shmem");
addr = (SHMdata * )mmap(0, sizeof(SHMdata) , PROT_READ | PROT_WRITE, MAP_SHARED, fd,0);
if(addr == (void *)-1) serror("mmap failed");
close(fd);
printf("\nclient[%d]: Map address = %6.6X\n",getpid(),addr);
for(counter=1;counter<argc;counter++)
{
                        nval = atoi(argv[counter]);
                        printf("\n\t-client [%d]: count me %d",getpid(),nval);
                        addr->data = nval;
                        addr->cmd = nval;
                        sleep(1);// Le processus serveur fait évoluer la mémoire partagée
                        printf("\n\t-client [%d]: got answer: %d", getpid(),addr->data);
}
addr->cmd = 0;// tué le processus serveur de mémoire partagé
printf("\n**    client [%d]: terminated\n",getpid());
}
```

Processus dit serveur de mémoire partagée

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <semaphore.h>
#include "sc.h"
#define PAGESIZE 4096
SEMAPHORE *SEM;
SHMdata         *addr;

void main(int argc, char *argv[])
{          int fd, nval;
int counter;
fd = shm_open(SHMname, O_RDWR | O_CREAT |O_EXCL, 0777);
if(fd<0)
{          printf("\ncan't open shmem (%s). Try to close it\n",strerror(errno));
                     shm_unlink(SHMname);
                    fd = shm_open(SHMname, O_RDWR | O_CREAT , 0777);
                     if(fd<0) puts("can't open shmem");
            }


if( ftruncate(fd,sizeof(SHMdata))==-1) serror("set shm size error");
addr =(SHMdata * ) mmap(0,sizeof(SHMdata),PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
if(addr == (void *)-1) serror("mmap failed");
close(fd);
printf("\nMap address = %6.6X\n",addr);

counter=0;
addr->cmd = 1;
while(1)
{                    if(addr->cmd==0) break;
            nval = (addr->data)+100;
            (addr->data) = nval;
            counter++;

}
shm_unlink(SHMname);
printf("\n\nSERVER TERMINATED\nTotal queries = %d\n",counter);
}
```

6.3) les signaux

Un signal est une petite quantité d'information (de la taille d'un int) envoyée entre deux processus.. Le signal est la solution la plus simple pour faire communiquer plusieurs processus.
Ils permettent aussi d'avertir simplement un processus qu'un événement est arrivé (Les signaux sont envoyés par le noyau pour indiquer l'arrivée d'un événement).
Il existe environ 32 signaux prédéfinis, ayant chacun une signification. Il n'est pas possible de définir de nouveaux signaux. Les signaux sont envoyés en particulier lorsque le processus tente une opération illégale comme une division par zéro ou un accès à une zone mémoire ne lui appartenant pas.

QNX implémente les signaux standards POSIX. La spécification POSIX originale définie les l'implémentation des signaux au niveau des processus seulement. Les signaux utilisent la communication asynchrone puisque le processus récepteur peut être interrompu à tout moment par un signal provenant du processus émetteur. C'est une interruption logicielle! Il n'y pas de réponse envoyée au thread émetteur et les messages sont prédéterminés par des constantes (voir signal.h).

Voici les signaux les plus importants:

• SIGCHLD: Mort d'un processus enfant
• SIGCONT: Continuer un processus interrompu
• SIGFPE: Erreur de calcul en point flottant
• SIGKILL: Tuer le processus (ne peut être détourné ou ignoré)
• SIGSEGV: Violation de l'espace mémoire
• SIGSTOP: Pause du processus
• SEGTERM: Terminer le processus
• SIGINT: CTRL-C

Un processus peut envoyer un signal à elle-même avec la fonction *raise* ou à un autre processus avec la fonction *kill*. Un processus répond à un signal en installant un gestionnaire de signal avec la fonction *signal ou sigaction* dont voici le prototype:

Les fonctions

```
#include <sys/types.h>
#include <signal.h>
int kill( pid_t pid,int sig );
```

pid : Le descripteur d'identification du processs auquel on vent envoyer un

sig : Le signal que l'on souhaite envoyer limité à 64 pour le RTOS QNX

Cette fonction envoie un signal à un processus identifié par son pid

```
#include <signal.h>
int sigaction( int sig, const struct sigaction * act, struct sigaction * oact );
```

sig : le numéro du signal que l'on souhaite intercepter

act : Pointeur sur une structure sigaction qui spécifie comment tu veux modifier les actions à réaliser pour le signal donné

oact : Null par défaut

Cette fonction spécifie quels sont les actions qui sont associés à la réception d'un signal spécifique (assimilable à une interruption logicielle)

```
#include <stdio.h>
#include <spawn.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#include <signal.h>


int main(void)
{
pid_t pid;
char *args[] = { "signalchild", NULL };
struct inheritance inherit;
puts("exemple passage d'un signal");
inherit.flags = 0;
if ((pid = spawn(args[0], 0, NULL, &inherit, args, environ)) == -1)
perror("spawn() failed");
printf("envoie du signal par le père (pid = %d)\n", getpid());
kill(pid, SIGTERM);
wait(NULL);
return 0;
}


#include <stdio.h>
#include <spawn.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#include <signal.h>
void MyHandler( int sig_number );
// attention ne prend en compte que les 64 premiers signaux
int main(int argc, char *argv[]) {
     struct sigaction action;
     action.sa_handler = &MyHandler; /* pointeur de fonction */
     //sigemptyset(&action.sa_mask); /* ensemble de signaux vide */
     action.sa_flags = 0; /* options par défaut */


     while(1){
               //sigmask(0);
          /*if (signal(0x80, MyHandler) != 0)
          {
          fprintf(stderr, "Erreur interception\n");
          exit(1);
          }*/
               if (sigaction(SIGTERM, &action, NULL) != 0)
               {
               fprintf(stderr, "Erreur sigaction\n");
               exit(1);
               }
```

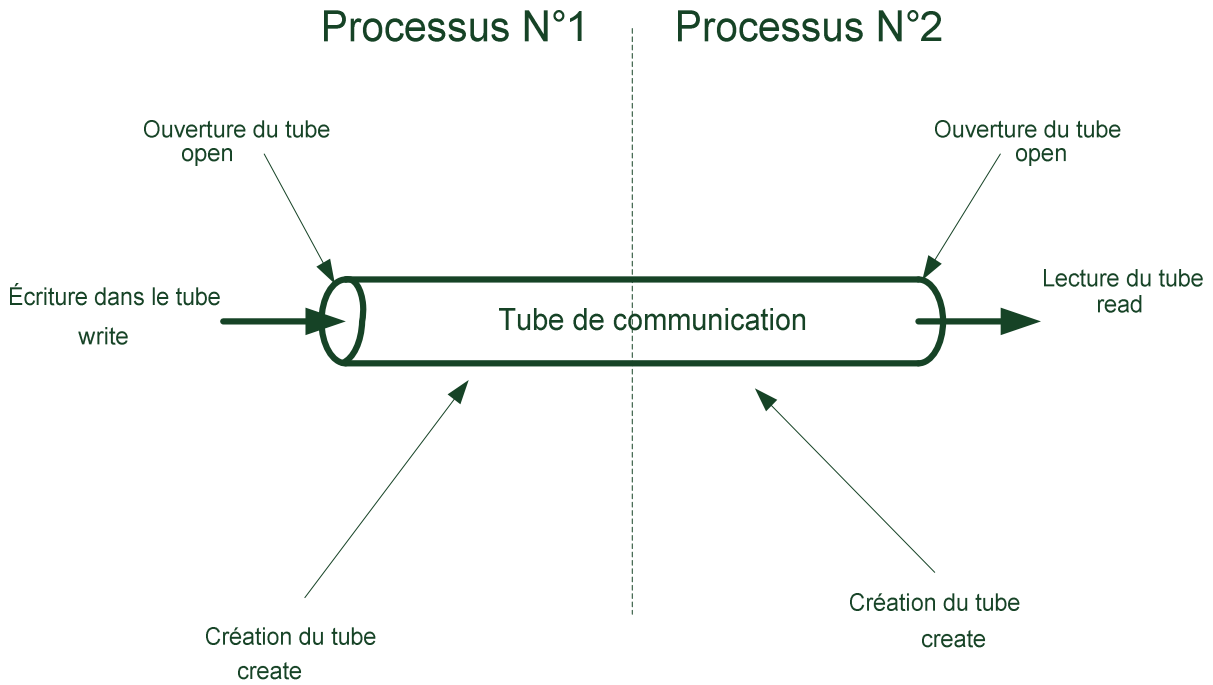
```
          sleep(0);// endormissement du processus
          }
}

/* cette fonction sera lancée à chaque réception d'un signal après cette routine
Le programme continueras dans le main après la fonction sigaction*/

void MyHandler( int sig_number )
     {
     if(sig_number==SIGTERM)
          printf("j'ai bien recu le signal \n");
     exit(0);
     }
```

6.4) Les tubes

Un tuyau est un canal de communication unidirectionnel et temporaire établi entre un premier processus qui écrit dans le tuyau et un deuxième processus qui lit les données du tuyau.
Les tubes (pipes) constituent le mécanisme fondamental de communication entre processus sous UNIX. Ce sont des files d'octets. Il est conseillé de les utiliser de façon unidirectionnelle. Si l'on veut une communication bidirectionnelle entre deux processus, il faut créer deux tubes, chacun étant utilisé dans un sens. Un tube est implémenté de façon proche d'un fichier.



Les tubes non nommés ne possèdent pas de nom. Il n'existe que le temps de l'exécution du processus qui le crée. Plusieurs processus peuvent lire ou écrire dans un même tube, sans qu'on puisse différencier l'origine des informations en sortie. La capacité d'un tube est limitée (PIPE_BUF octets). Les processus communiquant par un tube non nommé doivent avoir **un lien de parenté** (par exemple descendance d'un ancêtre commun ayant créé ce tube : problème classique de l'héritage des descripteurs de fichiers par un fils).
On crée un tube par un appel à la fonction pipe, déclarée dans unistd.h :
int pipe(int descripteur[2]);

La fonction renvoie 0 si elle réussit, et elle crée alors un nouveau tube. La fonction pipe remplit le tableau descripteur passé en paramètre, avec :
• descripteur[0] désigne la sortie du tube (dans laquelle on peut lire des données) ;
• descripteur[1] désigne l'entrée du tube (dans laquelle on peut écrire des données) ;
Le principe est qu'un processus va écrire dans descripteur[1] et qu'un autre processus va lire les mêmes données dans descripteur[0].

Les fonctions :

#include <unistd.h> int pipe(int *fildes*[2]);

fildes[2] : **un tableau ou la fonction peut stocker les descripteurs de fichier pour les terminaisons du tube .**

Création d'un tune non nommé

Cette fonction retourne 0 en cas de succès

Par défaut le flag O_NONBLOCK est inactif, tu peux le prépositionné ce flag en utilisant la fonction *fcntl()*

#include <unistd.h>

ssize_t read(int *fildes*, **void*** *buf*, **size_t** *nbyte* **);**

fildes : Le descripteur de fichier que tu veux lire
buf : Un pointeur sur un buffer ou la fonction peut stocker les données qui sont lues.
Nbyte : Le nombre d'octets que tu veux lire

Cette fonction permet de lire un flux d'octets à partir d'un fichier
Elle retourne le nombre d'octets lus

- Si aucun processus a un tube ouvert pour l'écriture la fonction read retourne 0
- Si un processus a un tube ouvert et un attribut non bloquant O_NONBLOCK prépositionné la fonction read retourne -1 si aucune donné n'a été envoyé dans le tube
- Si un processus a un tube ouvert et un attribut non bloquant O_NONBLOCK non prépositionné la fonction read est bloqué jusqu'à ce que des données soient envoyées dans le tube.

#include <unistd.h>

ssize_t write(int *fildes*, **const void*** *buf*, **size_t** *nbytes* **);**

fildes : Le descripteur de fichier pour l'écriture
buf : Un pointeur sur un buffer ou la fonction stocke les données pour l'écriture
Nbyte : Le nombre d'octets que tu veux écrire

Cette fonction permet d'écrire un flux d'octets à partir d'un fichier

Si le tube ouvert a prépositionné le flag O_NONBLOCK une requête d'écriture peut causer le blocage du processus tant que les données ne sont pas lues.
Si le tube ouvert a effacé le flag O_NONBLOCK une requête d'écriture ne provoque pas le blocage du processus tant que les données ne sont pas lues.

La fonction retourne le nombre d'octets écrit ou – 1 en cas de problème

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
#include <math.h>
#include <stdlib.h>

int main(void)
{
pid_t pid_fils;
int tube[2];
double x, valeurW, valeurR;
puts("Création d'un tube");
if (pipe(tube) != 0) /* pipe */
{
fprintf(stderr, "Erreur dans pipe\n");
exit(1);
}
        switch(pid_fils = fork()) /* fork */
        {
        case -1 :       perror("Erreur dans fork\n");
                        exit(errno);
        case 0 : /* processus fils */
                        close(tube[1]);
                        read(tube[0], &valeurR, sizeof(double));
                        printf("Le fils (%d) a lu : %.2f\n", getpid(), valeurR);
                        break;
        default : /* processus père */
                        printf("Fermeture sortie dans le père (pid = %d)\n", getpid());
                        close(tube[0]);
                        puts("\nEntrez x :");
                        scanf("%lf", &x);
                        valeurW =sin(x);//Right-click on the project: Properties | Linker |Extra Libs | Add. Then
                //type 'm' in the Name box.
                        //valeurW =(x);
                        write(tube[1], &valeurW, sizeof(double));
                        wait(NULL);
                        break;
        }
return 0;
}
```

Un tube nommé possède un nom dans le système. S'il n'est pas détruit, il persiste dans le système après la fin du processus qui l'a créé. Tout processus possédant les autorisations appropriées peut l'ouvrir en lecture ou en écriture. Un tube nommé conserve toutes les propriétés d'un tube : taille limitée, lecture destructive,

lecture/écriture réalisées en mode FIFO. Le tube nommé passe par un fichier sur le disque. L'intérêt est que **les deux processus n'ont pas besoin d'avoir un lien de parenté**. Pour créer un tube nommé, on utilise la fonction mkfifo de la bibliothèque sys/stat.h.

Les fonctions :

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo( const char* path, mode_t mode );
```

Cette fonction crée un tube du type nommé

path : Le nom du chemin ou tu veux utiliser le fichier spécial type FIFO

mode : La permission que tu souhaites donner pour accéder à ce fichier spécial type FIFO

Retourne 0 en cas de succès ou -1 en cas d'echec

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open( const char * path, int oflag );
```

path : Le nom du chemin du fichier que tu veux ouvrir.
oflag : Flag qui spécifie l'état et le mode d'accès au fichier.

Cette fonction retourne le descripteur de fichier

```
#include <stdio.h>
FILE* fdopen( int filedes, const char* mode );
```

Filedes : Le descripteur de fichier que tu veux associer avec un flux binaire
mode : le mode d'accès au fichier

Cette fonction associe un flux de fichier à un descripteur de fichier

Cette fonction retourne un descripteur de flux de fichier en cas de succes – 1 en cas d'échec

```
#include <unistd.h>
int unlink( const char * path );
```

path : le nom du chemin que tu veux effacer

Cette fonction efface un lien à un fichier.

```
/* processus création du tube */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
int fd;
FILE *fp;
char *nomfich="/usr/pipe"; /* nom du fichier attention au nom sous QNX ?*/
if(mkfifo(nomfich, O_NOCTTY | O_NDELAY | O_WRONLY | O_CREAT | O_TRUNC | S_IRUSR |
S_IWUSR | S_IRGRP | S_IWGRP) != 0) /* création du fichier */
{
perror("Problème de création du noeud de tube");
exit(1);
}
fd = open(nomfich, O_WRONLY); /* ouverture en écriture */
fp=fdopen(fd, "w"); /* ouverture du flot */
fprintf(fp, "coucou\n"); /* écriture dans le flot */
unlink(nomfich); /* fermeture du tube */
return 0;
}
```

/* processus lecture du tube */

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
int fd;
FILE *fp;
char *nomfich="/tmp/test.txt", chaine[50];
fd = open(nomfich, O_RDONLY); /* ouverture du tube */
fp=fdopen(fd, "r"); /* ouverture du flot */
fscanf(fp, "%s", chaine); /* lecture dans le flot */
puts(chaine); /* affichage */
unlink(nomfich); /* fermeture du flot */
return 0;
}
```

7) La synchronisation des processus et des threads

Les tâches ou les processus ont besoin **d'échanger des données** pour coopérer durant l'exécution d'une application.
Un processus a besoin de ***ressources*** pour s'exécuter : mémoire, processeur, mais aussi entrées/sorties (fichiers, communications avec d'autres processus s'exécutant ou non sur la même machine).
Une ressource peut être locale à un processus; dans ce cas, il est seul à l'utiliser et il n'y a pas de problèmes de synchronisation. Mais dans la plupart des cas, **les ressources sont partagées**. Le nombre de points d'accès d'une ressource est le nombre de processus qui peuvent simultanément l'utiliser. Par exemple, une imprimante ou un processeur sont des ressources à un seul point d'accès. Pour chaque ressource partagée, il faut mettre en place une politique de synchronisation.

7.1) synchronisation entre processus

généralités

Il existe 2 types de ressources :

- les **ressources matérielles** (mémoires, registres, organe périphérique, etc.)
- les **ressources logicielles** (fichiers, sous-programmes, buffers de données, etc.)

Parmi ces deux types de ressources, il faut distinguer les ressources :

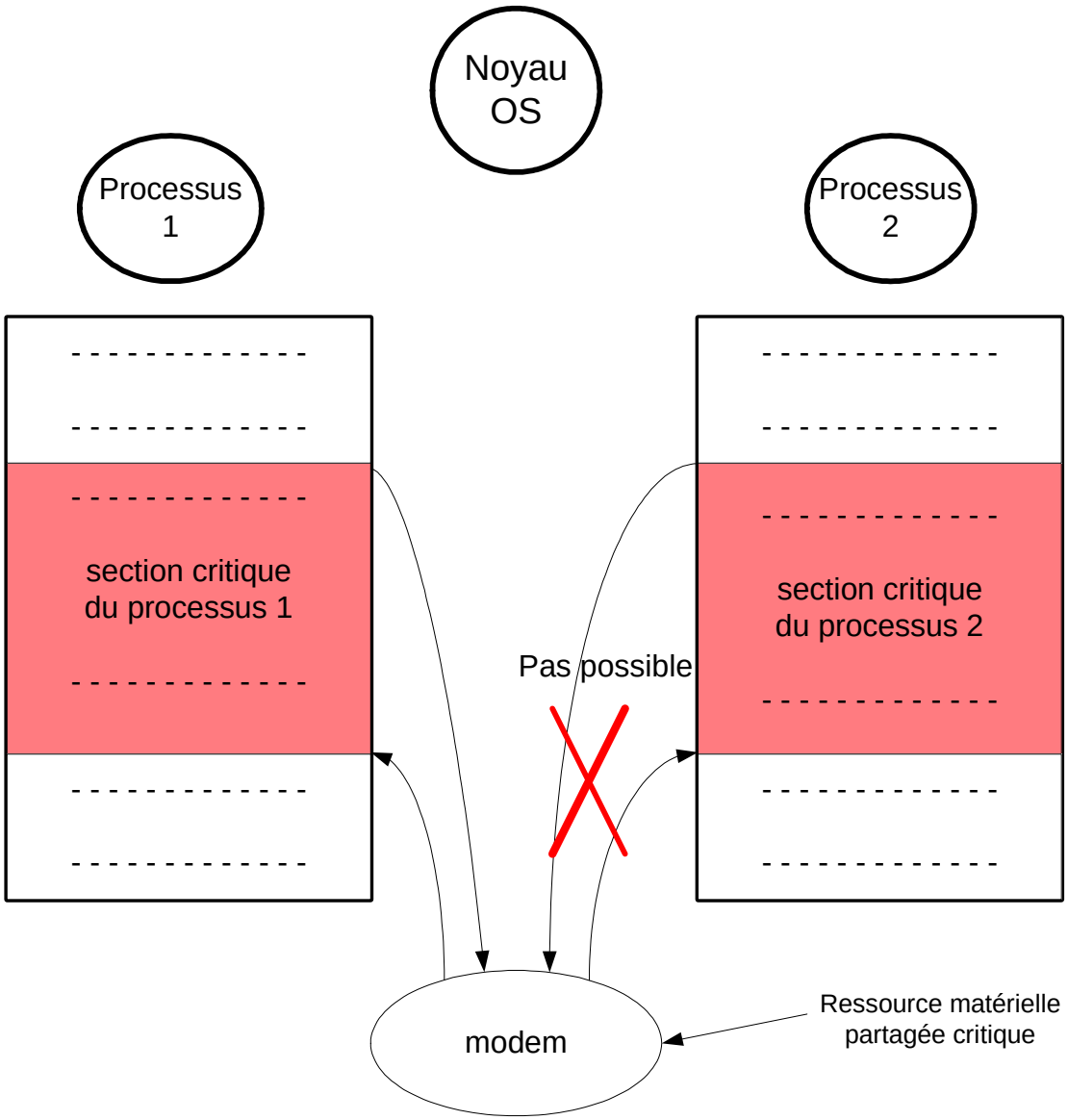
- **locales**, c'est à dire utilisées par une seule tâche.
- **communes**, c'est à dire **partageables** par plusieurs tâches.

Les ressources partageables sont accessibles par plusieurs tâches :

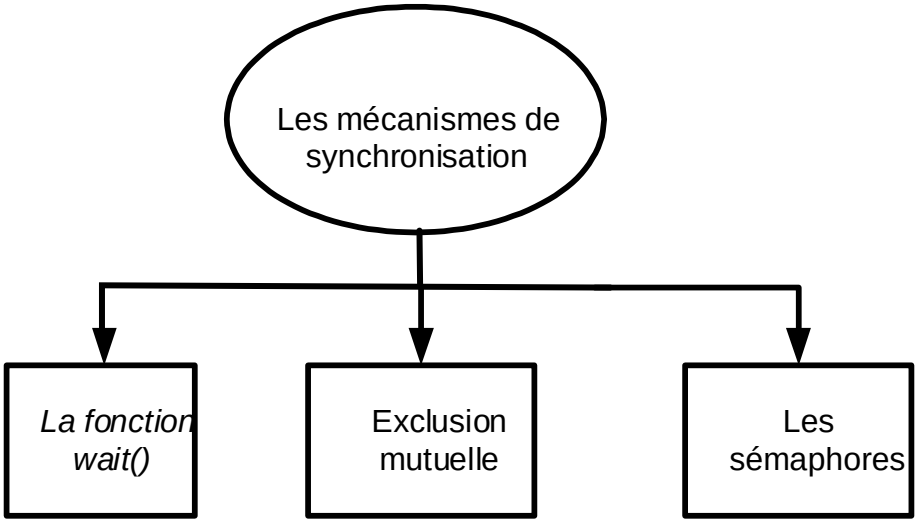
- Si une seule tâche à la fois peut accéder à la ressource, alors la ressource est **critique** (ou **non réentrante**). Ex: une imprimante.

- Si ***n*** tâches peuvent y accéder à la fois, alors elle est à ***n* entrées**. C'est une ressource partageable **réentrante**.

Lorsque la ressource est **partageable et non réentrante** (ressource critique)**,** il faut faire appel au principe d'**exclusion mutuelle**.
Elle empêche l'accès à la donnée si celle-ci est utilisée par une autre tâche.
La partie du programme conduisant à un conflit d'accès à une ressource est appelée ***section critique***.



7.2) Les différentes méthodes de synchronisation entre processus sous QNX



mecanismes de synchronisation	Implanté dans	Utilisé par	POSIX
Semaphore	Kernel	Process/Threads	YES
Exclusions mutuelles	Kernel	Threads	YES

7.3) Donnée partagées et exclusion mutuelle

Lorsqu'un nouveau processus est créé par un fork, toutes les données (variables globales, variables locales, mémoire allouée dynamiquement), sont dupliquées et copiées, et le processus père et le processus fils travaillent ensuite sur des variables différentes.

Dans le cas de threads, la mémoire est *partagée*, c'est à dire que les variables globales sont partagées entre les différents threads qui s'exécutent en parallèle. Cela pose des problèmes lorsque deux threads différents essaient d'écrire et de lire une même donnée.
Deux types de problèmes peuvent se poser :
• Deux threads concurrents essaient en même temps de modifier une variable globale ;
• Un thread modifie une structure de donnée tandis qu'un autre thread essaie de la lire.
Il est alors possible que le thread lecteur lise la structure alors que le thread écrivain a écrit la donnée à moitié. La donnée est alors incohérente.
Pour accéder à des données globales, il faut donc avoir recours à un mécanisme d'exclusion mutuelle, qui fait que les threads ne peuvent pas accéder en même temps à une donnée. Pour celà, on introduit des données appelés *mutex*, de type pthread_mutex_t.
Un thread peut verrouiller un mutex, avec la fonction pthread_mutex_lock(), pour pouvoir accéder à une donnée globale ou à un flot (par exemple pour écrire sur la sortie stdout). Une fois l'accès terminé, le thread dévérouille le mutex, avec la fonction pthread_mutex_unlock().
Si un thread A essaie de vérouiller le un mutex alors qu'il est déjà verrouillé par un autre thread B, le thread A reste bloqué sur l'appel de pthread_mutex_lock() jusqu'à ce que le thread B dévérouille le mutex. Une fois le mutex dévérouillé par B, le thread A verrouille immédiatement le mutex et son exécution se poursuit. Cela permet au thread B d'accéder tranquillement à des variables globales pendant que le thread A attend pour accéder aux mêmes variables.
Pour déclarer et initialiser un mutex, on le déclare en variable globale (pour qu'il soit accessible à tous les threads) :
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
La fonction pthread_mutex_lock(), qui permet de verrouiller un mutex, a pour prototype :
int pthread_mutex_lock(pthread_mutex_t *mutex);

Il faut éviter de verrouiller deux fois un même mutex dans le même thread sans le déverrouiller entre temps. Il y a un risque de blocage définitif du thread.

Certaines versions du système gèrent ce problème mais leur comportement n'est pas portable.
La fonction pthread_mutex_unlock(), qui permet de déverrouiller un mutex, a pour prototype
:
int pthread_mutex_unlock(pthread_mutex_t *mutex);

Dans l'exemple suivant, différents threads font un travail d'une durée aléatoire. Ce travail est fait alors qu'un mutex est verrouillé.

Les fonctions

pthread_mutex_lock

```
#include <pthread.h>
int pthread_mutex_lock( pthread_mutex_t* mutex );
```

mutex : Un pointeur sur l'objet mutex que tu veux verrouiller .

Cette fonction verrouille une zone critique encadrée par un mutex. Si le mutex est déjà verrouillé alors le thread appelant se bloque jusquà ce que le mutex soit déverrouillé.
Si un thread de priorité supérieure essaye de verrouiller un mutex déjà verrouillé alors la priorité effective de la tache (propriétaire courante du mutex) devient plus élevé que celle qui essaye de prendre le mutex
Si il y a plusieurs thread bloqué sur le mutex alors le thread en attente de plus haute priorité est débloqué et devient le prochain propriétaire du mutex

```
#include <pthread.h>
int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

mutex : Un pointeur sur l'objet mutex que tu veux déverrouiller.

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

pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;

void* ma_fonction_thread(void *arg);

int main(void)
{
int i;
pthread_t thread[10];
srand(time(NULL));
      for (i=0 ; i<3 ; i++)
      pthread_create(&thread[i], NULL, ma_fonction_thread, (void*)i);
      for (i=0 ; i<3 ; i++)
      pthread_join(thread[i], NULL);
return 0;
}

void* ma_fonction_thread(void *arg)
{
int num_thread = (int)arg;
int nombre_iterations, i, j, k, n;
nombre_iterations = rand()%8;

      for (i=0 ; i<nombre_iterations ; i++)
      {
      n = rand()%10000;
      pthread_mutex_lock(&my_mutex);
      printf("Le thread numro %d commence son calcul\n", num_thread);
      for (j=0 ; j<n ; j++)
      for (k=0 ; k<n ; k++)
      {}
      printf("Le thread numero %d a fini son calcul\n", num_thread);
      pthread_mutex_unlock(&my_mutex);
      }
      pthread_exit(NULL);
}
```

7.4) Les sémaphores

En général, une section critique est une partie du code où un processus ou un thread ne peut rentrer qu'à une certaine condition. Lorsque le processus (ou un thread) entre dans la section critique, il modifie la condition pour les autres processus/threads.
Par exemple, si une section du code ne doit pas être exécutée simultanément par plus de n threads. Avant de rentrer dans la section critique, un thread doit vérifier qu'au plus n-1 threads y sont déjà. Lorsqu'un thread entre dans la section critique, il modifie la condition.
Sur le nombre de threads qui se trouvent dans la section critique. Ainsi, un autre thread peut se trouver empêché d'entrer dans la section critique.
La difficulté est qu'on ne peut pas utiliser une simple variable comme compteur. En effet, si le test sur le nombre de thread et la modification du nombre de threads lors de l'entrée dans la section critique se font séquentiellement par deux instructions, si l'on joue de malchance un autre thread pourrait tester la condition sur le nombre de threads justement entre l'exécution de ces deux instructions, et deux threads passeraient en même temps dans la section critique.
Il y a donc nécessité de tester et modifier la condition de manière atomique, c'est-à-dire qu'aucun autre processus/thread de peut rien exécuter entre le test et la modification. C'est une opération atomique appelée *Test and Set Lock.*
Les sémaphores sont un type sem_t et un ensemble de primitives de base qui permettent d'implémenter des conditions assez générales sur les sections critiques. Un sémaphore possède un compteur dont la valeur est un entier positif ou nul. On entre dans une section critique si la valeur du compteur est strictement positive. Pour utiliser un sémaphore, on doit le déclarer et l'initialiser à une certaine valeur avec la fonction sem_init.

Définition :

Un sémaphore est l'association d'un **compteur ″c″** (à valeurs entières) avec une **file d'attente ″f″**.
Le compteur peut prendre une valeur soit positive, nulle ou négative.
La file d'attente est de type premier entré premier sorti (FIFO).

Règles de fonctionnement :

Le sémaphore ne peut être modifié que par 2 primitives P(s) (prendre le sémaphore) et V(s) (libérer le sémaphore):

- P(s) décrémente le sémaphore et suspend la tâche appelante s'il est négatif.
- V(s) incrémente le sémaphore et relance une tâche s'il est inférieur ou égal à zéro.

```
P(s):
    DEBUT
    |   c(s) <- c(s) - 1
    |   SI  c(s) < 0
    |   |   ALORS
    |   |       Bloquer la tâche ayant effectué la requête
    |   |       Mettre cette tâche de la file d'attente f(s)
    |   FINSI
    FIN
```

```
V(s) :
    DEBUT
    |   c(s) <- c(s) + 1
    |   SI  c(s) <= 0
    |   |   ALORS
    |   |       Relancer la tâche en attente dans f(s)
    |   FINSI
    FIN
```

Il suffit d'initialiser le sémaphore à 1 et d'encadrer chaque région critique par les deux opérations **P(s)** (*Prendre* la ressource) et **V(s)** (libérer la ressource).

Différents cas de figure :

- Si c(s) > 0 (=1) : aucune tâche n'est en section critique
- Si c(s) < 1 (ou n) : une (ou n) tâche est en section critique:

⇒ si c(s) < 0 alors abs(f(s)) = nombre de tâche dans la file d'attente

si c(s) = 0 alors la file d'attente est vide

```
#include <semaphore.h>
int sem_init( sem_t * sem,int pshared,unsigned value );
```

sem : Un pointeur sur la structure **`sem_t pour le sémaphore que tu souhaites initialisé`** A pointer to the
pshared : Non nulle si tu veux partager le sémaphore entre les processus via une mémoire partagée
value : La valeur initiale du compteur du sémaphore. Une valeur positive indique un sémaphore déverrouillé une valuer de 0 indique un sémaphore verrouillé.

```
#include <semaphore.h>
int sem_wait( sem_t * sem );
```

sem : Un pointeur sur le sémaphore que tu souhaites prendre.

Cette fonction décrémente le sémaphore et essaye de prendre le sémaphore. Si la valeur du compteur du sémaphore est inférieure ou égal à 0 le processus est bloqué à l'instruction **`sem_wait.`**

Elle retourne 0 si le sémaphore a été décrémenté
-1 si l'état du sémaphore est inchangé

```
#include <semaphore.h>
int sem_post( sem_t * sem );
```

sem : Un pointeur sur le sémaphore que tu souhaites libérer.

Cette fonction incrémente le sémaphore
Elle retourne -1 en cas d'erreur.

```
#include <semaphore.h>
int sem_destroy( sem_t * sem );
```

sem : Un pointeur sur le sémaphore que l'on souhaite détruire.

La fonction détruit le sémaphore non nommé

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

sem_t semaphore; /* variable globale : s maphore */
void* ma_fonction_thread(void *arg);

int main(int argc, char **argv)
{
int i;
pthread_t thread[10];
srand(time(NULL));
      if (argc != 2)
      {
            printf("Usage : %s nbthreadmax\n", argv[0]);
            exit(0);
      }
      sem_init(&semaphore, 0, atoi(argv[1])); /* initialisation */
/* cre ation des threads */
      for (i=0 ; i<5 ; i++)
            pthread_create(&thread[i], NULL, ma_fonction_thread, (void*)i);
      /* attente */
            for (i=0 ; i<5 ; i++)
                  pthread_join(thread[i], NULL);
            sem_destroy(&semaphore);
            return 0;
      }



void* ma_fonction_thread(void *arg)
{
int num_thread = (int)arg; /* num ro du thread */
int nombre_iterations, i, j, k, n;
nombre_iterations = rand()%8+1;
      for (i=0 ; i<nombre_iterations ; i++)
      {
            sem_wait(&semaphore);
            printf("Le thread %d entre dans la section critique\n",num_thread);
            sleep(rand()%9+1);// La section critique
            printf("Le thread %d sort de la section critique\n",num_thread);
            sem_post(&semaphore);
            sleep(rand()%9+1);
      }
pthread_exit(NULL);
}
```

8) Gestion des entrées sorties sous QNX

Pour s'interfacer à des entrées-sorties sous QNX :

Il faut valider un privilège pour accéder aux entrées-sorties par l'intermédiaire d'une tache.

ThreadCtl(_NTO_TCTL_IO, 0);

Et pour pouvoir accéder directement aux différents registres d'entrées sorties utilisez la fonction

mmap_device_memory

Pour lire un registre :

val8 = in8(iobase+N) // lit une valeur de 1 octet
val16 = in16(iobase+N) // lit une valeur de 2 octets
val32 = in32(iobase+N) // lit une valeur de 32bits

Pour écrire dans un registre :

val8 = out8(iobase+N) // écrit une valeur de 1 octet
val16 = out16(iobase+N) // écrit une valeur de 2 octets
val32 = out32(iobase+N) // écrit une valeur de 32bits

```
#include <sys/neutrino.h>

int ThreadCtl( int cmd, void * data );
```

cmd : la commande que tu veux executer
data : un pointeur sur la donnée associé à la commande spécifique.

Cet appel noyau permet de faire des changements spécifiques aux taches.

```
#include <hw/inout.h>

uint32_t in32( uintptr_t port );
```

port : Le port sur lequel tu veux lire une valeur de 32 bits

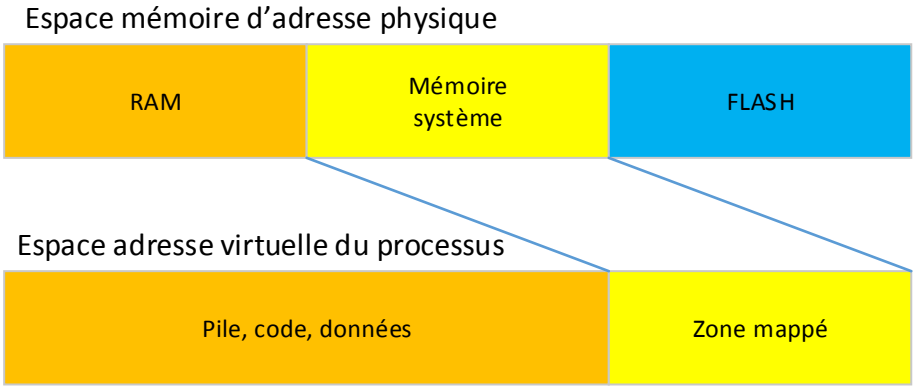
```
#include <sys/mman.h>
int munmap_device_memory( void * addr, size_t len );
```

addr : Le début de la gamme d'adresse que tu veux démapper

len : la longueur de la plage d'adresse en octet.

```
#include <sys/mman.h>
void * mmap_device_memory( void * addr, size_t len, int prot, int flags, uint64_t physical );
```

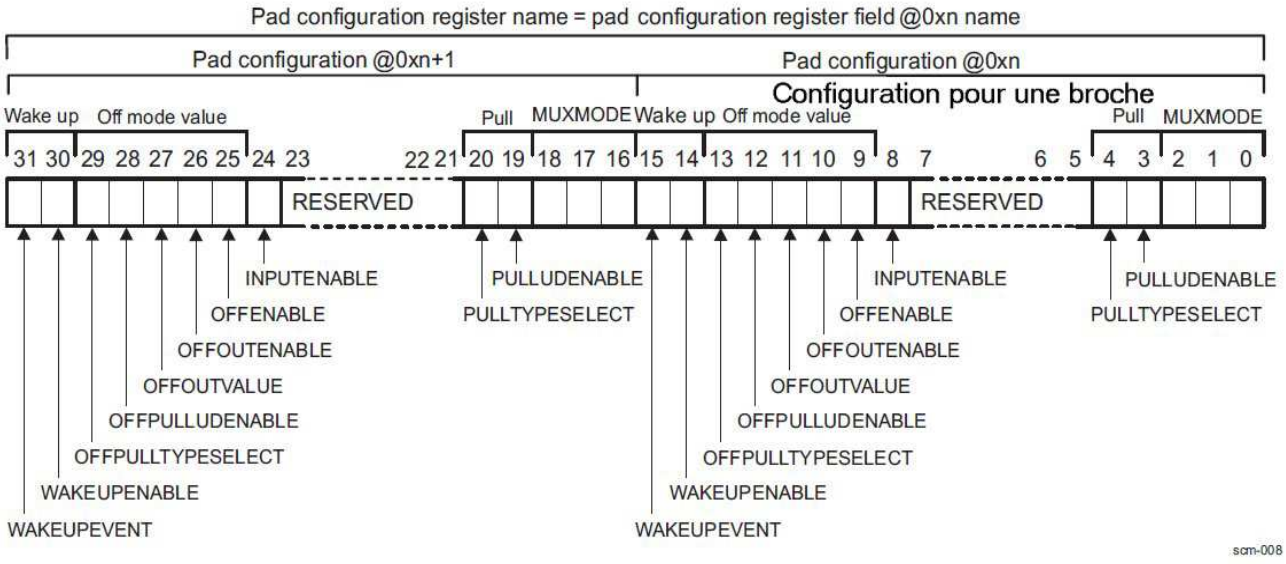
addr : NULL, ou un pointeur à l'adresse ou tu veux mapper dans l'espace d'adresse du processus appelant
len : Le nombre d'octet que tu veux mapper dans l'espace d'adresse
prot : Les restrictions d'accès que tu veux utiliser pour cette région mémoire qui sera mappé
physical : L'adresse physique de la mémoire mappé.
Cette fonction permet de mapper un système de mémoire physique dans un espace d'adresse virtuel du processus



Cette fonction retourne la première adresse de la zone mappée

Configuration des ports d'entrées sorties GPIO

Un port d'entrée sortie (GPIO) est constitué de 32 lignes d'entrée sortie. Après la phase de RESET, le logiciel doit prépositionné les registres de configuration et de mutliplexage de chaque broche. Les données écrites dans ces registres configurent directement le multiplexage de la logique de configuration des broches. Chaque broche est configurable en utilisant le registre de configuration qui a une largeur de 16 bits :
Un registre de configuration fournit un contrôle pour deux broches différentes :



Le registre de détection de front montant GPIO_RISINGDETECT permet de valider ou d'inhiber la détection de front montant sur l'une des 32 broches d'entrées sorties du GPIO lors d'une interruption ou de la détection d'un événement sur une broche :

GPIO_RISINGDE

Physical Address	0x4831 0048	**Instance**	GPIO1
	0x4905 0048		GPIO2
	0x4905 2048		GPIO3
	0x4905 4048		GPIO4
	0x4905 6048		GPIO5
	0x4905 8048		GPIO6
Description	This register is used to enable/disable for each input lines the rising-edge (transition 0=>1) detection to be used for the interrupt request and the wake-up generation.		
Type	RW		

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RISINGEDGE																															

Bits	Field Name	Description	Type	Reset
31:0	RISINGEDGE	Rising Edge Interrupt/wake-up enable	RW	0x00000000
		0x0: Disable IRQ /wake up on rising edge detect		
		0x1: Enable IRQ /wake up on rising edge detect		

Le registre de validation des interruptions du GPIO permet de valider ou d'inhiber une interruption sur l'une des 32 broches d'entrées sorties du GPIO.

GPIO_IRQENABLE1

Address Offset	0x01C		
Physical Address	0x4831 001C	**Instance**	GPIO1
	0x4905 001C		GPIO2
	0x4905 201C		GPIO3
	0x4905 401C		GPIO4
	0x4905 601C		GPIO5
	0x4905 801C		GPIO6
Description	This register provides IRQ 1 enable information.		
Type	RW		

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQENABLE1																															

Bits	Field Name	Description	Type	Reset
31:0	IRQENABLE1	Interrupt 1 Enable Register	RW	0x00000000
		0x0: Disable IRQ generation for channel N		
		0x1: Enable IRQ generation for channel N		

Le registre d'état des interruptions du GPIO permet de détecter une interruption sur l'une des 32 broches d'entrées sorties du GPIO.

GPIO_IRQSTATUS1

Address Offset	0x018		
Physical Address	0x4831 0018	**Instance**	GPIO1
	0x4905 0018		GPIO2
	0x4905 2018		GPIO3
	0x4905 4018		GPIO4
	0x4905 6018		GPIO5
	0x4905 8018		GPIO6
Description	This register provides IRQ 1 status information.		
Type	RW		

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQSTATUS1																															

Bits	Field Name	Description	Type	Reset
31:0	IRQSTATUS1	Interrupt 1 Status Register. Write a 1 in the corresponding bit to clear it to 0. Write 0 in the corresponding bit does not affect its value. 0x0: IRQ channel N not triggered 0x1: IRQ channel N triggered	RW	0x00000000

Attention à remettre le flag à 0 après l'interruption

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <hw/inout.h>
#include <sys/neutrino.h>
#include <sys/mman.h>
#include <errno.h>
#include <string.h>

/* Programme qui permet d'éteindre ou allumé  une LED par action sur le bouton poussoir user */


// documentation
// Base address of GPIO1
#define OMAP35XX_GPIO1_BASE                 0x48310000
// Base address of GPIO2
#define OMAP35XX_GPIO2_BASE                 0x49050000
// Base address of GPIO3
#define OMAP35XX_GPIO3_BASE                 0x49052000
// Base address of GPIO4
#define OMAP35XX_GPIO4_BASE                 0x49054000
// Base address of GPIO5
#define OMAP35XX_GPIO5_BASE                 0x49056000
// Base address of GPIO6
#define OMAP35XX_GPIO6_BASE                 0x49058000
// GPIO register size
#define OMAP35XX_GPIO_SIZE              0x1000
// This register is used to enable the pins output capabilities.
#define OMAP35XX_GPIO_OE                0x034
//This register is used to register the data that is read from the GPIO pins.
#define OMAP35XX_GPIO_DATAIN                0x038
//This register is used for setting the value of the GPIO output pins
#define OMAP35XX_GPIO_DATAOUT               0x03C
//This register is used to register the data that is read from the GPIO pins.
#define OMAP35XX_GPIO_SETDATAOUT        0x090
//This register is used for setting the value of the GPIO output pins
#define OMAP35XX_GPIO_CLEARDATAOUT  0x094

uint64_t gpio_base[] = {
                OMAP35XX_GPIO1_BASE,
                OMAP35XX_GPIO2_BASE,
                OMAP35XX_GPIO3_BASE,
                OMAP35XX_GPIO4_BASE,
                OMAP35XX_GPIO5_BASE,
                OMAP35XX_GPIO6_BASE,
NULL};

uint32_t ptr = 0;
```

```
void gpio_set(int pin)
{
	//out32(ptr + OMAP35XX_GPIO1_CLEARDATAOUT,  (1 << pin));
	out32(ptr + OMAP35XX_GPIO_DATAOUT, in32(ptr + OMAP35XX_GPIO_DATAOUT) | (1 <<
pin));
}

void gpio_reset(int pin)
{
	//out32(ptr + OMAP35XX_GPIO1_SETDATAOUT,  (1 << pin));
	out32(ptr + OMAP35XX_GPIO_DATAOUT, in32(ptr + OMAP35XX_GPIO_DATAOUT) & ~(1 <<
pin));
}

int gpio_read(int direction, int pin)
{
	int pin_value = 0;
	if (direction == 0){
		if (in32(ptr + OMAP35XX_GPIO_DATAOUT) & (1 << pin)){
			pin_value = 1;
		}
	}
	else if (direction == 1){
		if (in32(ptr + OMAP35XX_GPIO_DATAIN) & (1 << pin)){
			pin_value = 1;
		}
	}
	else{
		pin_value = -1;
	}
	return pin_value;
}

void gpio_set_direction(int direction, int pin)
{
	if (direction == 0){
		out32(ptr + OMAP35XX_GPIO_OE, in32(ptr + OMAP35XX_GPIO_OE) & ~(1 << pin));
	} else {
		out32(ptr + OMAP35XX_GPIO_OE, in32(ptr + OMAP35XX_GPIO_OE) | (1 << pin));
	}
}
```

```
int main (int argc, char *argv[])
{
	int opt = 0, verbose = 0, pin = -1, direction = -1, gpio_module = -1;
	int cmd = -1;
	extern char *optarg;

	// Handle commandline arguments
	// -m gpio_module  The GPIO module. Can be a value from 1 to 6
	// -p pin          The GPIO pin you want to set
	// -d direction    The direction of the GPIO pin. Can be: 0 (=write)|1(=read)
	// -c command      Action to perform. Can be: set|reset|read
	pin=22;
	direction = 0; //sortie
	gpio_module=5;
		/* enable this thread to execute i/o functions... */
		ThreadCtl(_NTO_TCTL_IO, 0);

		ptr = mmap_device_memory( 0, OMAP35XX_GPIO_SIZE,
PROT_READ|PROT_WRITE|PROT_NOCACHE, 0, gpio_base[gpio_module - 1]);
		if ( ptr == MAP_FAILED ) {
		   perror( "mmap_device_memory for physical address failed");
		   exit( EXIT_FAILURE );
		}

		gpio_set_direction(pin, direction);

		gpio_reset(pin);
		/*gpio_set(pin);*/
		//if (cmd == 0) gpio_set(pin);
		//if (cmd == 1) gpio_reset(pin);
		//if (cmd == 2) printf("0x%x\n",gpio_read(direction, pin));

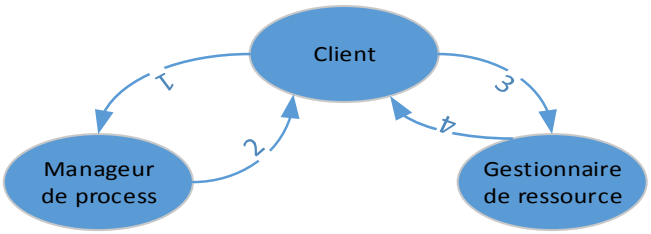
		munmap_device_memory(gpio_base[gpio_module - 1], OMAP35XX_GPIO_SIZE);
		printf("ok\n");//ok
		return 0;
	//} else {

	//	return -1;
	//}
}
```

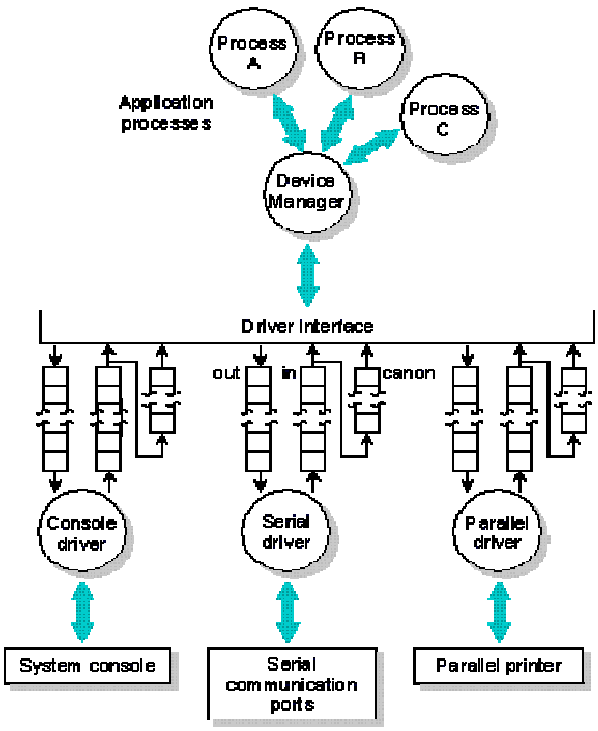
On peut accéder aussi aux périphériques en utilisant le manager de ressources.
C'est un serveur qui manipule certains messages qui peuvent être associés à du matériel ou peut être purement une entité logicielle (/dev/null).
Lorsqu'un applicatif envoie une requête de service.

Fd = open (" /dev/ser1 ", O_RDWR) ; // ouverture du port série

La fonction open envoie un message au manageur de process (1)
Le manageur de process répond en indiquant qui est concerné (2)
Le client ou l'applicatif établit une connexion au gestionnaire de ressource spécifié et renvoie un message d'ouverture contenant le descripteur de ressource (3).
Le gestionnaire de ressource répond par un état ok ou échec (4)



Dans l'exemple précédent, On peut écrire et lire directement sur le port série (printf, scanf) grâce à des fonctions de haut niveau en utilisant le gestionnaire de ressource et les pilotes associés au port série.
Le manageur de système QNX (dev) est l'interface entre les processus et les terminaisons des systèmes. Les terminaisons des systèmes sont positionnées dans l'espace d'entrée sortie avec des noms commençant par dev.
Le processus manager de système (Dev) contrôle le flux de données à partir ou vers les processus d'application.
L'interface matérielle est pilotée par des processus drivers individuels
Les flux de données entre les terminaisons et ses drivers sont gérés par une file d'attente dans une mémoire partagée pour chaque terminaison de système :



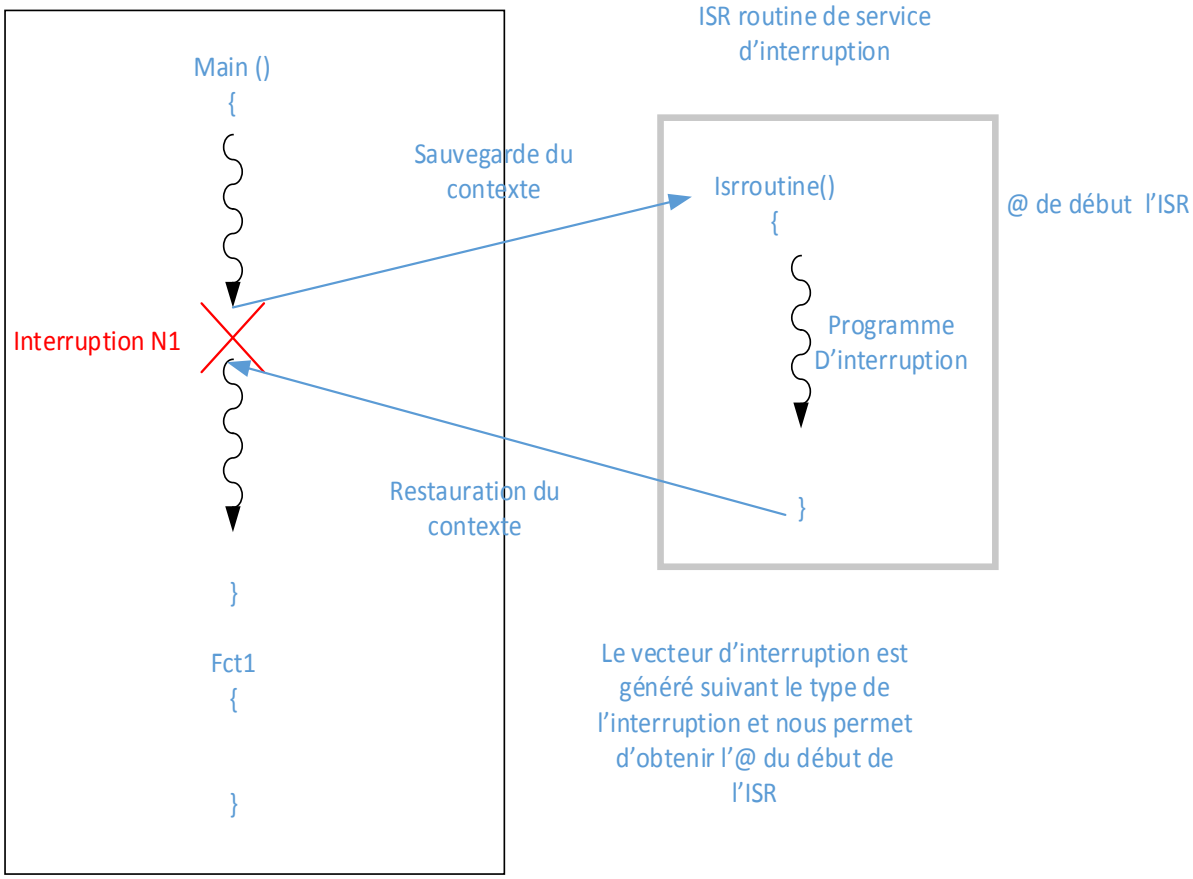
9) Gestion des interruptions sous QNX

9.1) Principe

Les interruptions sont des demandes de service effectuées par des périphériques du microprocesseur
Le microprocesseur dans son environnement est destiné à traiter des informations en "temps réel". L'application est couplée au monde extérieur, par l'échange fréquent de messages et de signaux analogiques à des instants non prévus. Pour répondre aux sollicitations extérieures, le microprocesseur est dans l'obligation de changer d'état en fonction des priorités relatives à l'opération en cours et de celle demandée.
Remarque : utiliser les interruptions permet de décharger le microprocesseur de certaines tâches répétitives (surveiller si un périphérique externe est prêt à placer des données sur le bus de données par exemple) consommatrices de cycles machine. En effet, avec une gestion efficace des interruptions, le microprocesseur n'a plus besoin d'observer son environnement (scrutation) en permanence, grâce aux lignes d'interruption, les interfaces sont capables de stopper le déroulement du programme principal et d'indiquer qu'elles demandent la prise en compte d'informations extérieures.

Les interruptions sont des événements (le plus souvent extérieurs) ou des instructions qui permettent d'interrompre le déroulement du programme principal (après exécution de l'instruction en cours), afin d'effectuer un autre programme (programme d'interruption). A l'issu de ce dernier le programme principal est repris exactement là où il avait été interrompu. Il existe donc sur les microprocesseurs des broches (entrées) dédiées à la réception des demandes d'interruptions. Une interruption est asynchrone : elle peut survenir à tout instant.
Une interruption survient en général pour :

- ✓ signaler un incident sur un périphérique ;
- ✓ demander un dialogue entre un périphérique et le microprocesseur.



Demande d'interruption

Afin d'installer une routine de service d'interruption (ISR) , l'applicatif doit dire à l'OS qu'il désire associer une ISR avec une source particulière d'interruption qui peut être une interruption matérielle *Interrupt Request* line (IRQ) ou une interruption logicielle. Le nombre de source d'interruption dépend de la configuration matérielle de la carte.
Dans beaucoup de cas un thread spécifie quelle source d'interruption il veut associer à l'ISR en utilisant les appels fonctions *InterruptAttach()* or *InterruptAttachEvent(). Quand l'applicatif veut dissocier l'ISR à partir d'une source d'interruptionil peut appeler InterruptDetach().*
Le programme de démarrage doit faire en sorte que toutes les interruptions sont masquées durant l'initialisation du système.
Quand le premier appel à *InterruptAttach() est fait pour un vecteur d'interruption le noyau démasque l'interruption.*
Les threads doivent avoir les privilèges d'entrées sorties

```
#define IRQ3  3 // vecteur d'interruption

Déclaration de  l'ISR */

extern const sigevent *serint (void *, int); // serint l'ISR
...
/*
* Associe le handler d'interruption serint,
* avec  IRQ 3
*/

ThreadCtl( _NTO_TCTL_IO, 0 );// I/O privileges

id = InterruptAttach (IRQ3, serint, NULL, 0, 0);
...
/* traitement à compléter . */
...
/* détache la source d'interruption  */

InterruptDetach (id);
```

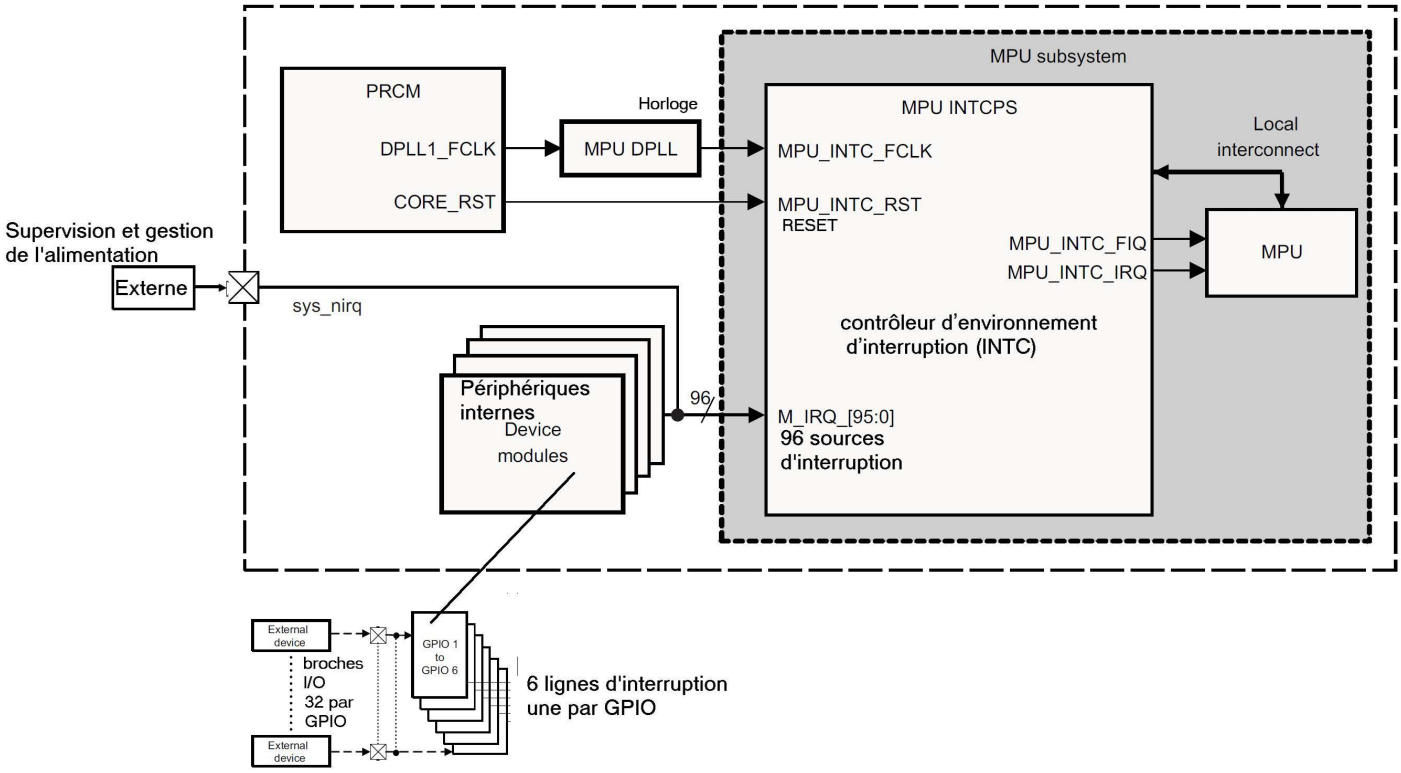
Un moyen simple élégant et rapide de mettre en place un service d'interruption est d'avoir un thread dédié au traitement de l'interruption. Le thread attache les interruptions (via *InterruptAttach()*) et alors celui-ci est en attente d'une interruption. Le blocage est appliqué par l'appel de la fonction *InterruptWait().*

Le contrôleur d'environnement d'interruption (INTC) du cortex A8

L'INTC peut manipuler deux sources d'interruptions à partir d'un système externe et une source en interne :

- Une ligne d'interruption sys_nirq relié à un circuit de gestion et de supervision d'alimentation externe
- Les lignes d'entrées sorties des modules GPIO. Il y a six lignes d'interruptions. Une ligne d'interruption est associée à un module GPIO qui peut générer une simple interruption suite à un événement sur l'une de ses 32 lignes d'entrée sorties
- Les lignes d'interruption de tous les autres modules intégrés dans le cortex A8

En tout il peut manipuler jusqu'à 96 requêtes d'interruption qui peuvent être configurées comme des requêtes d'interruption du type FIQ et IRQ



Le sous systèlme MPU INTCPS travaille à la moitié de l'horloge du MPU.
Une interruption FIQ (Fast interrupt request) est une interruption dont le temps de latence est le plus rapide. Elle est préemptive (peut interrompre une source d'interruption IRQ) . C'est une interruption qui sauvegarde le contexte du microprocesseur dans un ensemble de registres interne et non dans une pile. Du fait de sa rapidité de traitement, elle est utilisée notamment pour des applications critiques. Il n'y a aucune notion de priorité pour les canaux pouvant générer une interruption FIQ.

L'interruption IRQ est une requête normale d'interruption est utilisé pour manipuler des interruptions à usage général. Cette interruption est masquée lorsqu'une interruption FIQ est en cours.

Le niveau de priorité est donné ci-dessous

	Exceptions	Priority	*I* bit	*F* bit
	Reset	1	1	1
	Data Abort	2	1	—
FIQ	Fast Interrupt Request	3	1	1
IRQ	Interrupt Request	4	1	—
	Prefetch Abort	5	1	—
	Software Interrupt	6	1	—
	Undefined Instruction	6	1	—

Les différentes sources d'interruption

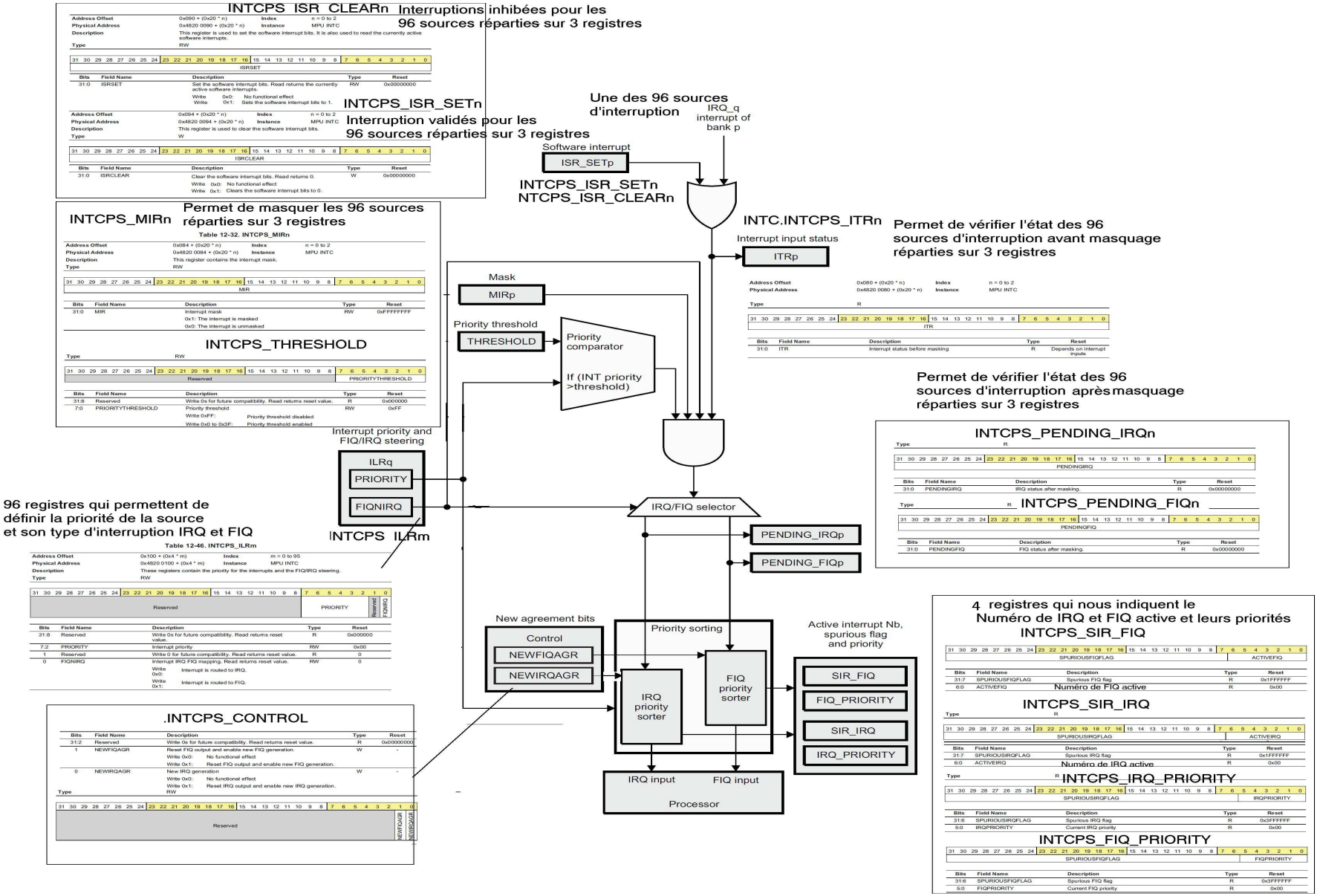
IRQ	Source	Description
M_IRQ_0	EMUINT	MPU emulation(2)
M_IRQ_1	COMMTX	MPU emulation(2)
M_IRQ_2	COMMRX	MPU emulation(2)
M_IRQ_3	BENCH	MPU emulation(2)
M_IRQ_4	MCBSP2_ST_IRQ	Sidetone MCBSP2 overflow
M_IRQ_5	MCBSP3_ST_IRQ	Sidetone MCBSP3 overflow
M_IRQ_6	Reserved	Reserved
M_IRQ_7	sys_nirq	External source (active low)
M_IRQ_8	Reserved	Reserved
M_IRQ_9	SMX_DBG_IRQ	L3 interconnect error for debug
M_IRQ_10	SMX_APP_IRQ	L3 interconnect error for application
M_IRQ_11	PRCM_MPU_IRQ	PRCM module IRQ
M_IRQ_12	SDMA_IRQ_0	System DMA request 0(3)
M_IRQ_13	SDMA_IRQ_1	System DMA request 1(3)
M_IRQ_14	SDMA_IRQ_2	System DMA request 2
M_IRQ_15	SDMA_IRQ_3	System DMA request 3
M_IRQ_16	MCBSP1_IRQ	McBSP module 1 IRQ (3)
M_IRQ_17	MCBSP2_IRQ	McBSP module 2 IRQ (3)
M_IRQ_18	SR1_IRQ	SmartReflex 1
M_IRQ_19	SR2_IRQ	SmartReflex 2
M_IRQ_20	GPMC_IRQ	General-purpose memory controller module
M_IRQ_21	SGX_IRQ	2D/3D graphics module
M_IRQ_22	MCBSP3_IRQ	McBSP module 3(4)
M_IRQ_23	MCBSP4_IRQ	McBSP module 4(4)
M_IRQ_24	CAM_IRQ0	Camera interface request 0
M_IRQ_25	DSS_IRQ	Display subsystem module(4)
M_IRQ_26	MAIL_U0_MPU_IRQ	Mailbox user 0 request
M_IRQ_27	MCBSP5_IRQ	McBSP module 5 (4)
M_IRQ_28	IVA2_MMU_IRQ	IVA2 MMU
M_IRQ_29	GPIO1_MPU_IRQ	GPIO module 1(4) (5)
M_IRQ_30	GPIO2_MPU_IRQ	GPIO module 2(4) (5)
M_IRQ_31	GPIO3_MPU_IRQ	GPIO module 3(4) (5)
M_IRQ_32	GPIO4_MPU_IRQ	GPIO module 4(4) (5)
M_IRQ_33	GPIO5_MPU_IRQ	GPIO module 5(4)
M_IRQ_34	GPIO6_MPU_IRQ	GPIO module 6(4)
M_IRQ_35	Reserved	Reserved
M_IRQ_36	WDT3_IRQ	Watchdog timer module 3 overflow
M_IRQ_37	GPT1_IRQ	General-purpose timer module 1
M_IRQ_38	GPT2_IRQ	General-purpose timer module 2
M_IRQ_39	GPT3_IRQ	General-purpose timer module 3
M_IRQ_40	GPT4_IRQ	General-purpose timer module 4
M_IRQ_41	GPT5_IRQ	General-purpose timer module 5(4)
M_IRQ_42	GPT6_IRQ	General-purpose timer module 6(4)
M_IRQ_43	GPT7_IRQ	General-purpose timer module 7(4)
M_IRQ_44	GPT8_IRQ	General-purpose timer module 8(4)
M_IRQ_45	GPT9_IRQ	General-purpose timer module 9
M_IRQ_46	GPT10_IRQ	General-purpose timer module 10
M_IRQ_47	GPT11_IRQ	General-purpose timer module 11
M_IRQ_48	SPI4_IRQ	McSPI module 4
M_IRQ_49	Reserved	Reserved
M_IRQ_50	Reserved	Reserved
M_IRQ_51	Reserved	Reserved
M_IRQ_52	Reserved	Reserved

IRQ	Source	Description
M_IRQ_53	Reserved	Reserved[4]
M_IRQ_54	MCBSP4_IRQ_TX	McBSP module 4 transmit[4]
M_IRQ_55	MCBSP4_IRQ_RX	McBSP module 4 receive[4]
M_IRQ_56	I2C1_IRQ	I^2C module 1
M_IRQ_57	I2C2_IRQ	I^2C module 2
M_IRQ_58	HDQ_IRQ	HDQ/1-Wire®
M_IRQ_59	McBSP1_IRQ_TX	McBSP module 1 transmit[4]
M_IRQ_60	McBSP1_IRQ_RX	McBSP module 1 receive[4]
M_IRQ_61	I2C3_IRQ	I^2C module 3
M_IRQ_62	McBSP2_IRQ_TX	McBSP module 2 transmit[4]
M_IRQ_63	McBSP2_IRQ_RX	McBSP module 2 receive[4]
M_IRQ_64	Reserved	Reserved
M_IRQ_65	SPI1_IRQ	McSPI module 1
M_IRQ_66	SPI2_IRQ	McSPI module 2
M_IRQ_67	Reserved	Reserved
M_IRQ_68	Reserved	Reserved
M_IRQ_69	Reserved	Reserved
M_IRQ_70	Reserved	Reserved
M_IRQ_71	Reserved	Reserved
M_IRQ_72	UART1_IRQ	UART module 1
M_IRQ_73	UART2_IRQ	UART module 2
M_IRQ_74	UART3_IRQ	UART module 3 (also infrared)[6]
M_IRQ_75	PBIAS_IRQ	Merged interrupt for PBIASlite1 and 2
M_IRQ_76	OHCI_IRQ	OHCI controller HSUSB MP Host Interrupt
M_IRQ_77	EHCI_IRQ	EHCI controller HSUSB MP Host Interrupt
M_IRQ_78	TLL_IRQ	HSUSB MP TLL Interrupt
M_IRQ_79	Reserved	Reserved
M_IRQ_80	UART4_IRQ	UART module 4
M_IRQ_81	MCBSP5_IRQ_TX	McBSP module 5 transmit[6]
M_IRQ_82	MCBSP5_IRQ_RX	McBSP module 5 receive[6]
M_IRQ_83	MMC1_IRQ	MMC/SD module 1
M_IRQ_84	Reserved	Reserved
M_IRQ_85	Reserved	Reserved
M_IRQ_86	MMC2_IRQ	MMC/SD module 2
M_IRQ_87	MPU_ICR_IRQ	MPU ICR interrupt
M_IRQ_88	D2DFRINT	From 3G coprocessor hardware when used in stacked modem configuration
M_IRQ_89	MCBSP3_IRQ_TX	McBSP module 3 transmit[6]
M_IRQ_90	MCBSP3_IRQ_RX	McBSP module 3 receive[6]
M_IRQ_91	SPI3_IRQ	McSPI module 3
M_IRQ_92	HSUSB_MC_NINT	High-Speed USB OTG controller
M_IRQ_93	HSUSB_DMA_NINT	High-Speed USB OTG DMA controller
M_IRQ_94	MMC3_IRQ	MMC/SD module 3
M_IRQ_95	Reserved	Reserved

Le graphique suivant nous indique l'architecture du contrôleur d'interruption

IRQ	Source	Description
M_IRQ_53	Reserved	Reserved[4]
M_IRQ_54	MCBSP4_IRQ_TX	McBSP module 4 transmit[4]
M_IRQ_55	MCBSP4_IRQ_RX	McBSP module 4 receive[4]
M_IRQ_56	I2C1_IRQ	I^2C module 1
M_IRQ_57	I2C2_IRQ	I^2C module 2
M_IRQ_58	HDQ_IRQ	HDQ/1-Wire®
M_IRQ_59	McBSP1_IRQ_TX	McBSP module 1 transmit[4]
M_IRQ_60	McBSP1_IRQ_RX	McBSP module 1 receive[4]
M_IRQ_61	I2C3_IRQ	I^2C module 3
M_IRQ_62	McBSP2_IRQ_TX	McBSP module 2 transmit[4]
M_IRQ_63	McBSP2_IRQ_RX	McBSP module 2 receive[4]
M_IRQ_64	Reserved	Reserved
M_IRQ_65	SPI1_IRQ	McSPI module 1
M_IRQ_66	SPI2_IRQ	McSPI module 2
M_IRQ_67	Reserved	Reserved
M_IRQ_68	Reserved	Reserved
M_IRQ_69	Reserved	Reserved
M_IRQ_70	Reserved	Reserved
M_IRQ_71	Reserved	Reserved
M_IRQ_72	UART1_IRQ	UART module 1
M_IRQ_73	UART2_IRQ	UART module 2
M_IRQ_74	UART3_IRQ	UART module 3 (also infrared)[6]
M_IRQ_75	PBIAS_IRQ	Merged interrupt for PBIASlite1 and 2
M_IRQ_76	OHCI_IRQ	OHCI controller HSUSB MP Host Interrupt
M_IRQ_77	EHCI_IRQ	EHCI controller HSUSB MP Host Interrupt
M_IRQ_78	TLL_IRQ	HSUSB MP TLL Interrupt
M_IRQ_79	Reserved	Reserved
M_IRQ_80	UART4_IRQ	UART module 4
M_IRQ_81	MCBSP5_IRQ_TX	McBSP module 5 transmit[6]
M_IRQ_82	MCBSP5_IRQ_RX	McBSP module 5 receive[6]
M_IRQ_83	MMC1_IRQ	MMC/SD module 1
M_IRQ_84	Reserved	Reserved
M_IRQ_85	Reserved	Reserved
M_IRQ_86	MMC2_IRQ	MMC/SD module 2
M_IRQ_87	MPU_ICR_IRQ	MPU ICR interrupt
M_IRQ_88	D2DFRINT	From 3G coprocessor hardware when used in stacked modem configuration
M_IRQ_89	MCBSP3_IRQ_TX	McBSP module 3 transmit[6]
M_IRQ_90	MCBSP3_IRQ_RX	McBSP module 3 receive[6]
M_IRQ_91	SPI3_IRQ	McSPI module 3
M_IRQ_92	HSUSB_MC_NINT	High-Speed USB OTG controller
M_IRQ_93	HSUSB_DMA_NINT	High-Speed USB OTG DMA controller
M_IRQ_94	MMC3_IRQ	MMC/SD module 3
M_IRQ_95	Reserved	Reserved

Le graphique suivant nous indique l'architecture du contrôleur d'interruption


INTCPS_ISR_CLEARn
Interruptions inhibées pour les 96 sources réparties sur 3 registres
INTCPS_ISR_SETn
Interruption validés pour les 96 sources réparties sur 3 registres
Une des 96 sources d'interruption
IRQ_q interrupt of bank p
Software interrupt
ISR_SETp
INTCPS_ISR_SETn
NTCPS_ISR_CLEARn
INTCPS_MIRn
Permet de masquer les 96 sources réparties sur 3 registres
INTCPS_THRESHOLD
INTC.INTCPS_ITRn
Interrupt input status
ITRp
Permet de vérifier l'état des 96 sources d'interruption avant masquage réparties sur 3 registres
Mask
MIRp
Priority threshold
THRESHOLD
Priority comparator
If (INT priority >threshold)
Permet de vérifier l'état des 96 sources d'interruption après masquage réparties sur 3 registres
INTCPS_PENDING_IRQn
INTCPS_PENDING_FIQn
Interrupt priority and FIQ/IRQ steering
ILRq
PRIORITY
FIQNIRQ
96 registres qui permettent de définir la priorité de la source et son type d'interruption IRQ et FIQ
INTCPS_ILRm
IRQ/FIQ selector
PENDING_IRQp
PENDING_FIQp
New agreement bits
Control
NEWFIQAGR
NEWIRQAGR
Priority sorting
IRQ priority sorter
FIQ priority sorter
Active interrupt Nb, spurious flag and priority
SIR_FIQ
FIQ_PRIORITY
SIR_IRQ
IRQ_PRIORITY
IRQ input
FIQ input
Processor
4 registres qui nous indiquent le Numéro de IRQ et FIQ active et leurs priorités
INTCPS_SIR_FIQ
Numéro de FIQ active
INTCPS_SIR_IRQ
Numéro de IRQ active
INTCPS_IRQ_PRIORITY
INTCPS_FIQ_PRIORITY
.INTCPS_CONTROL

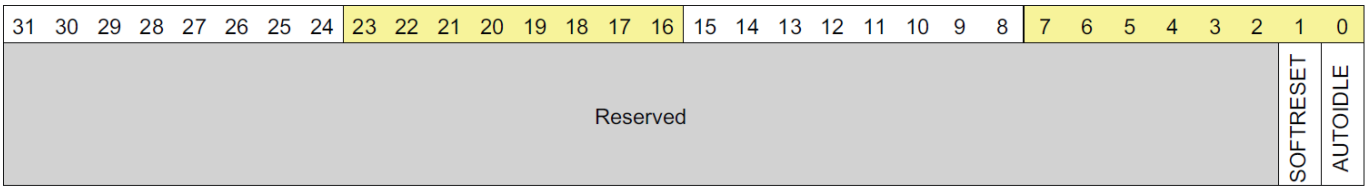
Pour valider un traitement rapide des interruptions de priorité élevé, un seuil de masquage de priorité programmable est fourni INTCPS_THRESHOLD[7:0]. Ce seuil de priorité permet une préemption par des interruptions de priorité supérieur. Toutes les interruptions de priorité égal ou plus petite au seuil sont masqués. Cependant la priorité 0 peut ne jamais être masqué par ce seuil, un seuil de priorité 0 est traité de la même façon qu'une priorité 1. Les valeurs des priorités et du seuil de priorité peuvent être positionné entre 0X0 et 0x3F. 0x00 est la plus haute priorité et 0X3F est la plus basse. Quand un masquage par priorité n'est pas nécessaire, un seuil de priorité de 0XFF desactive le mécanisme de suil de priorité.

Un niveau de priorité 0 est assigné à chaque source d'interruption entrante. A la fois le niveau de priorité et le type d'interruption (IRQ et FIQ) sont configurés par le registre .INTCPS_ILRm. Si plus d'une interruption entrante avec la même priorité et le même type (IRQ et FIQ) a lieu simultanément l'interruption de niveau la plus élevée est prise en compte en premier. Quand un ou plusieurs interruptions entrantes non masqué sont détectés, le registre INTCPS sépare les interruptions IRQ et FIQ en utilisant le bit FIQNIRQ du registre INTCPS_ILRm[0]. Le resultat est placé dans les registres INTCPS_PENDING_IRQn or INTCPS_PENDING_FIQn.
Les priorités sortantes pour les interruptions IRQ and FIQ peuvent s'executer en parallèle.
Chaque priorité IRQ/FIQ sortante détermine le numéro de priorité la plus élevée.
Chaque numéro de priorité est placé dans les registres INTCPS_SIR_IRQ, .INTCPS_SIR_FIQ.
La valeur est préservée jusqu'à ce que les bits de INTCPS_CONTROL NEWIRQAGR or NEWFIQAGR sont positionnés.

Le mode de sauvegarde de l'énergie et notamment la validation de l'horloge d'interface est validé si le bit AUTOIDLE de INTCPS_SYSCONFIG[0]. Quand ce mode est validé et il y a aucune activité sur le bus L'holorge d'interface est désactivée en interne dans le module ceci réduit la consommation de puissance. Quand il y a une nouvelle activité sur le bus d'interface L'horloge d'interface redémarre. Après un RESET ce mode est désactivé.

Table 12-10. INTCPS_SYSCONFIG

Address Offset	0x010		
Physical Address	0x4820 0010	**Instance**	MPU INTC
	0x480C 8010		Modem INTC
Description	This register controls various parameters of the module interface.		
Type	RW		



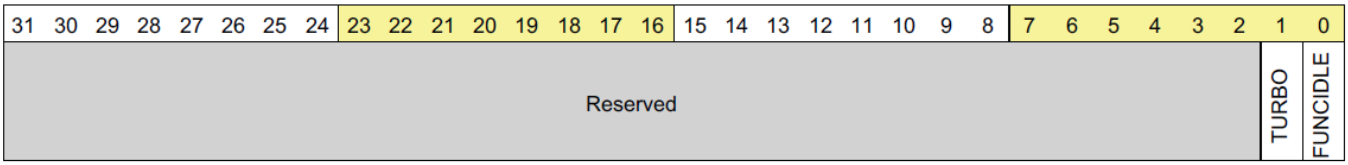
Bits	Field Name	Description	Type	Reset
31:2	Reserved	Write 0s for future compatibility. Read returns reset value.	R	0x00000000
1	SOFTRESET	Software reset. Set this bit to trigger a module reset. The bit is automatically reset by the hardware. Read returns 0. Write 0x0: No functional effect Write 0x1: The module is reset.	RW	0
0	AUTOIDLE	Internal interface clock gating strategy	RW	0

Le mode de sauvegarde de l'énergie et notamment la validation de l'horloge d'auto-IDLE est validé si le bit FUNCIDLE de .INTCPS_IDLE[0] est positionné à 0. Quand ce mode est validé et qu'il il n'ya aucune interruption active, L'horloge de fonctionnement de l'horloge d'auto-IDLE est désactivé afin de réduire la

consommation de puissance. Quand une interruption non masquée est détectée l'horloge fonctionnelle redémarre.
L'entrée d'horloge du synchronisateur dispose d'un mode de sauvegarde de l'énergie qui est validé quand le bit turbo du registre INTCPS_IDLE[1] est positionné. Le synchronisateur d'horloge permet aux interuuptions externes de se resynchroniser avant qu'elle soient masquées.

Table 12-22. INTCPS_IDLE

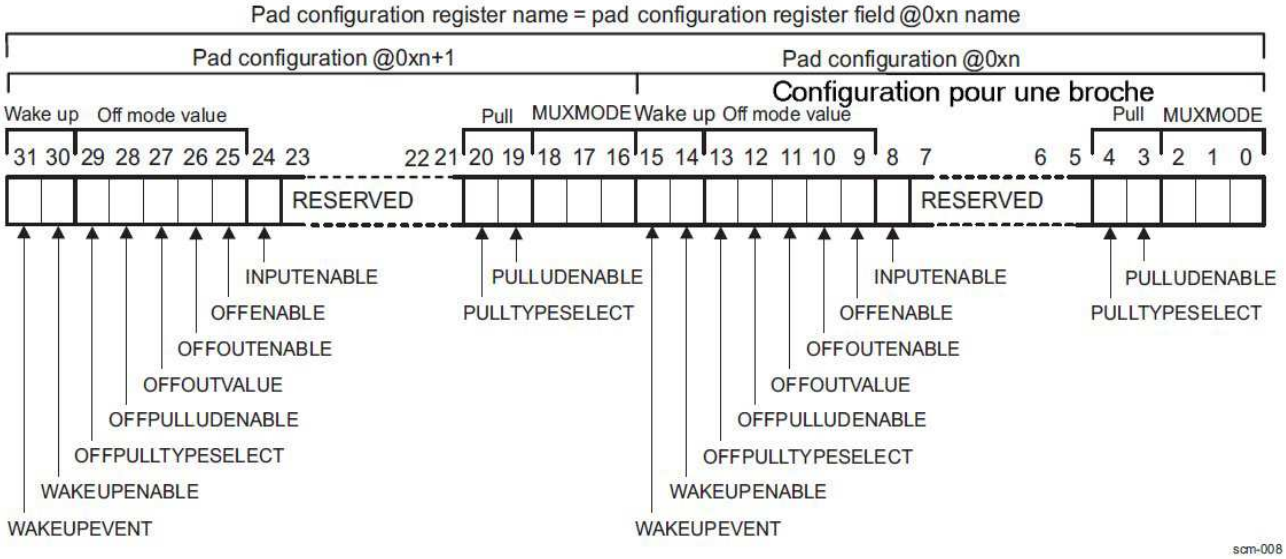
Address Offset	0x050		
Physical Address	0x4820 0050	**Instance**	MPU INTC
	0x480C 8050		Modem INTC
Description	This register controls the functional clock auto-idle and the synchronizer clock auto-gating.		
Type	RW		



Bits	Field Name	Description	Type	Reset
31:2	Reserved	Write 0s for future compatibility. Read returns reset value.	R	0x00000000
1	TURBO	Input synchronizer clock auto-gating	RW	0
		0x0: Input synchronizer clock is free-running (default).		
		0x1: Input synchronizer clock is auto-gated based on interrupt input activity.		
0	FUNCIDLE	Functional clock idle mode	RW	0
		0x0: Functional clock gating strategy is applied (default).		
		0x1: Functional clock is free-running.		

Configuration des ports d'entrées sorties GPIO

Un port d'entrée sortie (GPIO) est constitué de 32 lignes d'entrée sortie. Après la phase de RESET, le logiciel doit prépositionné les registres de configuration et de mutliplexage de chaque broche. Les données écrites dans ces registres configurent directement le multiplexage de la logique de configuration des broches. Chaque broche est configurable en utilisant le registre de configuration qui a une largeur de 16 bits :
Un registre de configuration fournit un contrôle pour deux broches différentes :



Le registre de détection de front montant GPIO_RISINGDETECT permet de valider ou d'inhiber la détection de front montant sur l'une des 32 broches d'entrées sorties du GPIO lors d'une interruption ou de la détection d'un événement sur une broche :

GPIO_RISINGDE

Physical Address	0x4831 0048	**Instance**	GPIO1
	0x4905 0048		GPIO2
	0x4905 2048		GPIO3
	0x4905 4048		GPIO4
	0x4905 6048		GPIO5
	0x4905 8048		GPIO6
Description	This register is used to enable/disable for each input lines the rising-edge (transition 0=>1) detection to be used for the interrupt request and the wake-up generation.		
Type	RW		

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RISINGEDGE																															

Bits	Field Name	Description	Type	Reset
31:0	RISINGEDGE	Rising Edge Interrupt/wake-up enable 0x0: Disable IRQ /wake up on rising edge detect 0x1: Enable IRQ /wake up on rising edge detect	RW	0x00000000

Le registre de validation des interruptions du GPIO permet de valider ou d'inhiber une interruption sur l'une des 32 broches d'entrées sorties du GPIO.

GPIO_IRQENABLE1

Address Offset	0x01C		
Physical Address	0x4831 001C	**Instance**	GPIO1
	0x4905 001C		GPIO2
	0x4905 201C		GPIO3
	0x4905 401C		GPIO4
	0x4905 601C		GPIO5
	0x4905 801C		GPIO6
Description	This register provides IRQ 1 enable information.		
Type	RW		

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQENABLE1																															

Bits	Field Name	Description	Type	Reset
31:0	IRQENABLE1	Interrupt 1 Enable Register 0x0: Disable IRQ generation for channel N 0x1: Enable IRQ generation for channel N	RW	0x00000000

Le registre d'état des interruptions du GPIO permet de détecter une interruption sur l'une des 32 broches d'entrées sorties du GPIO.

GPIO_IRQSTATUS1

Address Offset	0x018		
Physical Address	0x4831 0018	**Instance**	GPIO1
	0x4905 0018		GPIO2
	0x4905 2018		GPIO3
	0x4905 4018		GPIO4
	0x4905 6018		GPIO5
	0x4905 8018		GPIO6
Description	This register provides IRQ 1 status information.		
Type	RW		

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQSTATUS1																															

Bits	Field Name	Description	Type	Reset
31:0	IRQSTATUS1	Interrupt 1 Status Register. Write a 1 in the corresponding bit to clear it to 0. Write 0 in the corresponding bit does not affect its value. 0x0: IRQ channel N not triggered 0x1: IRQ channel N triggered	RW	0x00000000

Attention à remettre le flag à 0 après l'interruption

Les fonctions d'interruption

#include <sys/neutrino.h>

int InterruptAttach(int *intr***, const struct sigevent * (*** *handler***)(void *, int), const void *** *area***, int** *size***, unsigned** *flags* **);**

intr : le numéro de vecteur que tu veux attacher à l'ISR
handler : Un pointeur sur l'adresse de la fonction ISR
area : Un pointeur sur une zone de communication de ton process , or NULL si tu ne veux pas de zone de communication
size : La dimension de la zone de communication 0 si nul
flags : Flags qui spécifie comment tu veux attacher ton gestionnaire d'interruption.

L' appel noyaux *InterruptAttach() attache la fonction d'interruption (ISR) à une interruption matériel spécifique (numéro de vecteur). Cette fonction valide automatiquement les niveaux d'interruption*

Avant d'appeler la fonction *InterruptAttach(), un thread doit faire une requête de privilège d'entrée sortie par l'appel :*

ThreadCtl(_NTO_TCTL_IO, 0);

#include <sys/neutrino.h>

int InterruptWait(int *flags***, const uint64_t *** *timeout* **);**

flag et timeout paramètres nuls

Cet appel noyau est en attente d'une interruption matérielle. Le thread appelant attachera la fonction ISR à l'interruption.
L'appel **InterruptWait est bloquant.**

Cette fonction retourne -1 en cas d'échec
EOK en cas de succès.

```
/* prise en compte d'une interruption suite à l'action sur le boutton user, commutation de l'état de la
LED sur la carte */
#include <cstdlib>
#include <iostream>
//#include "GPIO/GPIO.h"
#include <hw/inout.h>
#include <sys/mman.h>
#include <pthread.h>
#include <sys/neutrino.h>
#include <errno.h>
//#include <stdio.h>
//#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
//#include <hw/inout.h>
#include <sys/neutrino.h>
//#include <sys/mman.h>
//#include <errno.h>
#include <string.h>
#define OMAP35XX_GPIO5_BASE                0x49056000 // pour accéder au GPIO5
#define OMAP35XX_GPIO_DATAOUT              0x03C
#define GPIO_4    0x00000010
#define GPIO_29 0x20000000

void ConfigInterrupt(void);
void ConfigGPIOInterrupt (void);
void ResetIRQStatus (void);
void ConfigLed (void);
uintptr_t MapUserspace (unsigned int len,unsigned long address);
void *intr_thread(void*);
void *wait_thread(void*);
const struct sigevent* isr_userbutton(void* arg, int id);
volatile bool isLedOn = 1;
   volatile bool ResetStatus = 0;
   volatile bool isIntrOn = 0;
   volatile bool error = 0;
   struct sigevent event;


int main(int argc, char *argv[]) {
event.sigev_notify = SIGEV_INTR;
std::cout<<"Carotte"<<std::endl;
std::cout<<"Create interrupt thread"<<std::endl;
pthread_create(NULL, NULL, &intr_thread, NULL );// start up a thread that is dedicated to interrupt
processing
delay(5);
while (!error);
std::cout<<"Exiting program"<<std::endl;
return EXIT_SUCCESS;
}
```

```
const struct sigevent* isr_userbutton(void* arg, int id){// ceci est l' ISR
        //uint32_t mmap_addr;
isLedOn = isLedOn ^ 1;
return (&event);
}

void *intr_thread(void*){// ce  thread est dédié à manipuler et à manager les  interruptions
uint32_t mmap_addr;
uint32_t mem;
unsigned int numerovector=29;
int numinter;
if (ThreadCtl(_NTO_TCTL_IO,0)== - 1)// enable I/O privilege
std::cout << "Thread control failed"<< std::endl;
else{
/*****************************************************/

std::cout << " Before Interrupt Attach"<<std::endl; // Attach ISR routine
//SIGEV_INTR_INIT(&event);
if((numinter=InterruptAttach(numerovector,&isr_userbutton,0,0,_NTO_INTR_FLAGS_TRK_MSK))
== -1){// attache l' ISR au numéro de vecteur 29  GPIO1 isr_userbutton fct interrupt
        //if((numinter=InterruptAttach(numerovector,&isr_userbutton,NULL,0,_NTO_INTR_FLAGS_E
ND)) == -1 //attache l' ISR au numéro de vecteur 29  GPIO1 isr_userbutton fct interrupt
        std::cout<<"InterruptAttach() failed"<<std::endl;
error = 1;
}
else{

        ConfigLed();
           ConfigInterrupt();
           ConfigGPIOInterrupt();
           while(1){
             std::cout<<"Before Interrupt Wait"<<std::endl;
             mmap_addr = MapUserspace(4,0x48310018);// IRQstatus 1
             std::cout<<"IRQ Status : "<<std::hex<<in32(mmap_addr)<<std::endl;//IRQstatus 1 Clear
Status GPIO1

             mmap_addr = MapUserspace(4,0x48200090);// INTCPS_ISR_SET
             std::cout<<"INTCPS_ISR_SET :
"<<std::hex<<in32(mmap_addr)<<std::endl;//INTCPS_ISR_SETn This register is used to set the
software interrupt bits.

             InterruptWait(0,0);//thread en attente d'inter après inter on continue à l'instruction suivante
             std::cout<<"Interrupt Wait"<<std::endl;
               if (isLedOn){
                 mmap_addr = MapUserspace(4,0x4905603C);
                 out32(mmap_addr,in32(mmap_addr)& ~(1 << 22));
                 std::cout<<"Off"<<std::endl;
               }
               else{
```

```
                    mmap_addr = MapUserspace(4,0x4905603C);
                    out32(mmap_addr,in32(mmap_addr)| (1 << 22));// acces GPIO150 set
                    std::cout<<"On"<<std::endl;
                  }
                  mmap_addr = MapUserspace(4,0x48310018);//IRQstatus 1 Clear Status GPIO1
                  std::cout<<"IRQ Status : "<<std::hex<<in32(mmap_addr)<<std::endl;

                  ResetIRQStatus();
              }
            }
return EXIT_SUCCESS;
}
}

 uintptr_t MapUserspace (unsigned int len,unsigned long address){
      uintptr_t mmap_addr;
      mmap_addr = mmap_device_io(len,address);
      if (mmap_addr == MAP_DEVICE_FAILED){
        std::cout << " mmap_device_io failed" << std::endl;
        error = 1;
      }
      return mmap_addr;
    }



    void ConfigInterrupt(void){
      uintptr_t mmap_addr;

      mmap_addr = MapUserspace(4,0x48200010); //Sysconfig Interface clock is free-running.
      out32(mmap_addr,0);// INTCPS_SYSCONFIG pas de gestion d'énergie

      mmap_addr = MapUserspace(4,0x48200050); //INTCPS_IDLE Idle Functional clock is free-
running..
      out32(mmap_addr,1);// INTCPS_IDLE pas de gestion d'énergie

      mmap_addr = MapUserspace(4,0x48200174); //INTCPS_ILRm ILRm IRQ priorité 0
      out32(mmap_addr,0);// INTCPS_ILRm

      mmap_addr = MapUserspace(4,0x48200088); // INTCPS_MIR_CLEARn Clear MIRn Clears the
MIR mask bit to 0 pour numéro vecteur 29
      out32(mmap_addr,GPIO_29);// INTCPS_MIR_CLEARn

      std::cout<<"ConfigInterrupt Done"<<std::endl;
    }
```

```
void ConfigGPIOInterrupt (void){
    uintptr_t mmap_addr;
    uint32_t buffer;

    mmap_addr = MapUserspace(4,0x48310064); //Set IRQ Enable 1
    out32(mmap_addr,GPIO_4);

    mmap_addr = MapUserspace(4,0x48310048); //Edge (rising = 0x48, falling = 0x4C)
    buffer = in32(mmap_addr) | GPIO_4;
    out32(mmap_addr,buffer);

    mmap_addr = MapUserspace(4,0x48310018); //IRQ Status 1
    out32(mmap_addr,GPIO_4);


    std::cout<<"ConfigGPIOInterrupt Done"<<std::endl;
  }

  void ResetIRQStatus (void){
    uintptr_t mmap_addr;
    mmap_addr = MapUserspace(4,0x48310018); //IRQ Status 1
    out32(mmap_addr,GPIO_4);

    mmap_addr = MapUserspace(4,0x48200088); //Clear MIR
    out32(mmap_addr,GPIO_29);// INTCPS_MIR_CLEARn

    std::cout<<"ResetIRQStatus Done"<<std::endl;

  }

  void ConfigLed (void){
    uintptr_t mmap_addr;
    uint32_t buffer;

    mmap_addr = MapUserspace(4,0x48002A0C);//PADCONF user button //PADCONF userbutton
réservation mémoire registre 0x48002A0C CONTROL_PADCONF_SYS_BOOT1[15:0]
    buffer = (in32(mmap_addr) & 0x0000FFFF) | 0x050C0000;//// SYS_BOOT2/GPIO_4 interrupteur
sur bloc GPIO1 broche N°4 dit SYS_BOOT2
    out32(mmap_addr,buffer);//ONTROL_PADCONF_SYS_BOOT1[32:16]registre configuration Pin
SYS_BOOT2/GPIO_4
  }
```

10) Le Gestionnaire de ressource matérielle

10.1) Présentation

Un manageur de ressource est un processus qui enregistre un nom dans l'espace des noms du système de fichier. Les autres processus utilisent ce chemin pour communiquer avec le manageur de ressource. Le manageur de ressource a la responsabilité de présenter une interface aux différents types de système. Ces systèmes peuvent être matériels (ports séries, ports parallèles , cartes réseaux , disques durs) ou virtuels (comme **/dev/null**, un système de fichier réseau). Dans beaucoup d'OS, cette fonctionnalité est traditionnellement associée avec les drivers systèmes. Au contraire des drivers systèmes le manageur de ressource ne nécéssite aucun arrangement spécial avec le noyau. En fait le manageur de ressource commence comme un processus qui est séparé du noyau et peut être appelé par les programmes aux niveaux utilisateur.Un manageur de ressource accepte les messages à partir d'autres programmes et communique avec le matériel. Il enregistre un prefix pour les noms de chemin dans l'espace des noms de chemin (e.g. **/dev/ser1**), et d'autres processus peuvent ouvrir le nom en utilisant les fonctions standard du C

- *open()*
- *read()*
- *write()*

Un manageur de ressource n'est pas restreint à simplement manipuler les fonctions *open(), read(),* and *write(), il peut supporter n'importe quelles fonctions qui sont basées sur un descripteur de fichier aussi bien que d'autres formes d'IPC*
Par exemple un port série peut être piloté par un manageur de ressource qui peut être appellé **/dev/ser1 dans l'espace des noms de chemin.**
Quannd un processus fait une requête de service sur le port série il a juste à ouvrir le port série dans ce cas **/dev/ser1**

```
fd = open("/dev/ser1", O_RDWR);
for (packet = 0; packet < npackets; packet++)
{
write(fd, packets[packet], PACKET_SIZE);
}
close(fd);
```

Les avantages d'écrire son propre manageur de ressource :

- Le client respecte la nome POSIX et tous les programmateurs sont familiers des fonctions *open(), read(),* and *write(). On s'éloigne de la machine et de la programmation très fastidieuse des registres en fournissant une couche logicielle de plus haut niveau au client .*
- *On peut réduire le nombre de type d'interface et les normaliser pour tous les applicatifs (un seul driver unique à tous les processus clients).*
- *On peut utiliser les lignes de commandes en mode console pour communiquer avec le manageur de ressource*

Il y a 2 types de manageur de ressource

- ✓ Les manageurs de ressources matérielles
- ✓ Les manangeurs de ressources pour les systèmes de fichier

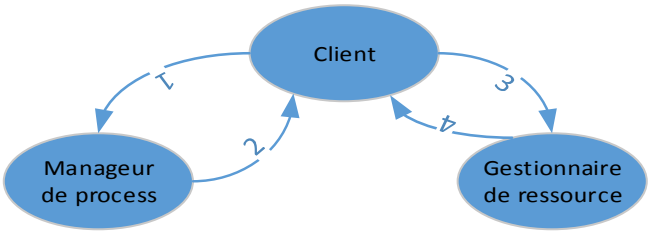
On ne s'intéressera qu'au premier cas.

10.2) Les mécanismes de communication d'un client avec un gestionnaire de ressource

Lorsqu'un applicatif client envoie une requête de service.

Fd = open (" /dev/ser1 ", O_RDWR) ; // ouverture du port série

La fonction open envoie un message au manageur de process (1)
Le manageur de process répond en indiquant qui est concerné (2)
Le client ou l'applicatif établit une connexion au gestionnaire de ressource spécifié et renvoie un message d'ouverture contenant le descripteur de ressource (3).
Le gestionnaire de ressource répond par un état ok ou échec (4)



10.3) Développement d'un manageur de ressources

Le manageur de ressource est un serveur qui utilise le protocole de messagerie Neutrino send/receive/reply pour recevoir et répondre aux messages. Son algoritme de fonctionnement est donné ci-dessous

Initialise le manageur de ressource
Enregistre le nom dans l'espce des noms de chemin avec le manageur de process
Tant que (1 différent de 0)
    Faire
    Recevoir un message
        Suivant le type de message faire
        _IO_CONNECT : appelle de la fonction io_open ;
        _IO_READ : appelle de la fonction io_read
        _IO_WRITE : appelle de la fonction io_write
        Fsuivant
Fin tant que

Les fonctions pour le manageur de ressource

a) Initialise les attributs pour le manageur de ressource

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>
dispatch_t *dispatch_create( void );
```

Nous avons besoin de prépositionner un mécanisme afin que les applicatifs clients puissent envoyer des messages au manageur de ressource. Ceci est fait grâce à la fonction *dispatch_create() qui crée et retourne la structure dispatch. Cette structure contient l'identifiant de canal. A noté que l'identifiant de canal n'est pas actuellement créé jusqu'à ce que tu attaches quelque chose comme resmgr_attach(), message_attach(),* and *pulse_attach().*

Cette fonction fait partie de la couche dispatch du manageur de ressource

b) Initialise les fonctions utilisées par les messages

```
#include <sys/iofunc.h>

void iofunc_func_init(unsigned nconnect,resmgr_connect_funcs_t *connect,unsigned nio,
resmgr_io_funcs_t *io );
```

nconnect : le nombre d'entrée de la table de connexions que tu veux typiquement. Tu passes l'argument _RESMGR_CONNECT_NFUNCS.
connect : Un pointer sur un structure du type **resmgr_connect_funcs_t** que tu veux avec les fonctions par défaut de connexion
nio : Le nombre d'entrée de la table d'entrée sortie dont tu as besoin typiquement tu passes l'argument _RESMGR_IO_NFUNCS.
io : Un pointeur sur une structure **resmgr_io_funcs_t** dont tu as besoin avec les fonctions par défaut d'entrée sorties

Quand tu appelles *resmgr_attach()*, tu lui fais passer une structure de contrôle **resmgr_attr_t** qui correspond aux attributs du manageur de ressource
Cette structure de contrôle est initialisée comme ceci :

```
memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;
```

Nous configurons ainsi :

- Combien de structures d'entrée sorties sont disponibles pour la réponse du serveur
- La dimension du buffer de réception

La fonction **iofunc_func_init permet de définir quelles fonctions sont appellées quand un message particulier arrive dans notre cas :**

• *Les fonctions de connexions resmgr_attach(),*

• *les fonctions d'entrée sorties*

c) Initialise la structure d'attribut utilisée par le système

```
#include <sys/iofunc.h>

void iofunc_attr_init (iofunc_attr_t *attr, mode_t mode, iofunc_attr_t *dattr,
struct _client_info *info );
```

attr : un pointeur sur la structure d'attribut **iofunc_attr_t** que tu veux initialiser.
mode : le type de permission d'accès que tu veux utiliser pour la ressource
dattr : NULL, ou un pointeur sur la structure **iofunc_attr_t** que tu veux utiliser pour initialiser la structure pointée par attr
info : Null ou un pointeur sur la structure **_client_info** qui contient les informations de connexion du client

La structure d'attribut contient des informations à partir de notre système associé au nom **/dev/sample,** il contient au moins les informations suivantes :

- La permission et les types de service
- Les utilisateurs et les identifiants de groupe

d) Enregistrement d'un nom dans l'espace des noms

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>
```

```
int resmgr_attach (dispatch_t *dpp,resmgr_attr_t *attr,const char *path,enum _file_type
file_type,unsigned flags,const resmgr_connect_funcs_t *connect_funcs,const resmgr_io_funcs_t
*io_funcs, RESMGR_HANDLE_T *handle );
```

Avant qu'un manageur de ressource puisse recevoir les messages à partir des programmes client, il a besoin d'informer les autres programmes (via le manageur de process) qu'il est responsable d'un espace de nom particulier (ici **"/dev/sample).** Ceci est réalisé par l'enregistrement des noms de chemin. Quand le nom est enregistré les processus clients peuvent trouver et se connecter au manageur de ressource en utilisant le nom enregistré

Les paramètres d'entrées de la fonction sont les suivants :

```
id = resmgr_attach(dpp, /* dispatch handle */
                   &resmgr_attr, /* resource manager attrs */
                   "/dev/sample", /* device name */
                   _FTYPE_ANY, /* open type */
                   0, /* flags */
                   &connect_funcs, /* connect routines */
                   &io_funcs, /* I/O routines */
                   &attr); /* handle */
if(id == -1) {
fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
return EXIT_FAILURE;
}
```

e) Mise en place d'une structure de contexte

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>
```

```
dispatch_context_t * dispatch_context_alloc ( dispatch_t * dpp );
```

La structure de contexte contient un buffer ou les messages seront reçus. La dimension du buffer a été mise en place quand nous avons initalisé la structure d'attribut du manageur de ressource. La structure de contexte contient un buffer d'entrée sortie que la librairie peut utiliser pour répondre aux messages (reply). Le nombre d'entrée sortie a été mise en place quand nous avons initialisé la structure d'attribut du manageur de ressource.

f) Démarrage de la boucle de message du manageur de ressource

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>
```

```
dispatch_context_t * dispatch_block ( dispatch_context_t * ctp );
```

Une fois que le manageur de ressource a établi son nom, il reçoit les messages des programmes clients *open(), read(), write()*) sur ce nom. Le client construit un message du type _IO_CONNECT et l'envoie à notre manageur de ressource. Le manageur de ressource reçoit le message à l'intérieur de la fonction *dispatch_block()*

Quand nous appellons la fonction *dispatch*, celle-ci decode le message et appelle la fonction appropriée basée sur le type de connexion et la table des fonctions d'entrée sorties.
Cette fonction est définie dans la table des fonctions d'entrée sorties soit la structure io_funcs champ devctl

```
io_funcs.devctl = io_devctl;
```

C'est cette fonction qui sera appelée à chaque message du type _IO_CONNECT

g) Réception et exploitation d'un message issu d'un client

La structure **io_devctl_t contient les messages** the _IO_ message recus par le manageur de ressource

```
struct _io_devctl {
uint16_t type; // le type IO_CTL
uint16_t combine_len;// utile pour des messages combinés
int32_t dcmd; // type de commande qui a été envoyer
int32_t nbytes;// Le nombre d'octet du message
int32_t zero;
/* char data[nbytes]; *// s'il y a un message entrant il est placé dans un tableau
};
struct _io_devctl_reply {
uint32_t zero;
int32_t ret_val;// valeur de retour en réponse
int32_t nbytes;// Le nombre d'octet du message retourné
int32_t zero2;
/* char data[nbytes]; */*// s'il y a un message sortant de réponse  il est placé dans un tableau
} ;
typedef union {
struct _io_devctl i;// Message entrant
struct _io_devctl_reply o;// message sortant
} io_devctl_t;
```

Les structures de message d'entrées sorties **io_devctl_t** sont une union du message entrant i (venant du manageur de ressource et indirectement du programme client) et du message sortant o de réponse (qui partira vers le client).

```
#include <sys/iofunc.h>
int iofunc_devctl_default( resmgr_context_t *ctp,io_devctl_t *msg,iofunc_ocb_t *ocb );
```

*ctp : un pointeur sur la structure **resmgr_context_t** que le manageur de ressource utilise pour passer les informations de contexte*
*msg : Un pointeur sur la structure **io_devctl_t qui contient le message que le manageur de ressource a reçu de l'applicatif client***
*ocb : Un pointeur sur la structure **iofunc_ocb_t** pour les blocs de contrôle d'ouverture quand le client a ouvert la ressource*

Cette fonction *iofunc_devctl_* implémente les actions par défaut pour les messages du type _IO_DEVCTL

Cette fonction est suivie par la macro *DEVCTL_DATA()* qui renvoie un pointeur sur les données qui suivent le message entrant **char data[nbytes]**

data = _DEVCTL_DATA (msg->i); // Récupération des données pour le message entrant

Pour connaitre les commandes envoyées par le client et agir en conséquence on utilise le switch:

```
switch (msg->i.dcmd) {
//case MY_DEVCTL_SETVAL: global_integer = rx_data->data32; nbytes = 0; break;
case INITI2C: value=rx_data->data.txdata[0];rx_data->data.rxdata[0] = value; /* See note
4 */nbytes = sizeof(rx_data->data.rxdata);break;
case WRITEREGISTERI2C: adress = rx_data->data.txdata[1];value = rx_data-
>data.txdata[0];nbytes = sizeof(rx_data->data.rxdata);rx_data->data.rxdata[0] =
adress;rx_data->data.rxdata[1]=0xFF;break;
case READREGISTERI2C:break;
// See note 4. The rx data overwrites the tx data for this command.
default:return(ENOSYS);
}
```

Il faut au préalable avoir défini ces différentes commandes sous forme binaire à la fois dans le serveur gestionnaire de ressource et le client applicatif.

```
#define MY_CMD_CODE 1
#define INITI2C __DIOTF(_DCMD_MISC, MY_CMD_CODE + 0, union _my_devctl_msg)
//#define MY_DEVCTL_SETVAL __DIOT(_DCMD_MISC, MY_CMD_CODE + 1, int)
#define WRITEREGISTERI2C __DIOTF(_DCMD_MISC, MY_CMD_CODE + 1, union _my_devctl_msg)
#define READREGISTERI2C __DIOTF(_DCMD_MISC, MY_CMD_CODE + 2, union _my_devctl_msg)
```

La macro `__DIOTF`

__DIOTF(*class*, *cmd*, *data*)

class : les types de catégories pour les commandes, pour notre gestionnaire _DCMD_MISC.
Cmd : la commande spécifique dans la classe
Data : le type de donnée à passer vers le gestionnaire de ressource

Cette fonction passe des informations au système.

h) Réponse du manageur de ressource à l'applicatif client

```
#include <sys/resmgr.h>
#define_RESMGR_PTR( resmgr_context_t ctp, void msg, size_t nbytes ) ...
```

ctp : un pointeur sur la structure **resmgr_context_t** que le manageur de ressource utilise pour passer les informations de contexte
msg : Un pointeur sur la structure **io_devctl_t qui contient le message de réponse à l'applicatif client**
nbytes : Le nombre d'octet que tu souhaites envoyer en réponse.

Cette fonction va permettre de compléter la structure de sortie **io_devctl_t (réponse du serveur à l'applicatif)**

10.4) Développement de l'applicatif client faisant appel au gestionnaire de ressource

Ouverture d'un descripteur de fichier

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

int open(const char * path**, int** oflag**, ...);**

path : le nom du chemin du fichier que tu veux ouvrir .
oflag : flags qui spécifie l'état et les modes d'accès aux fichiers.

Cette fonction retourne le descripteur de fichier

Emission d'un message lié à une commande spécifique

```
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>
int devctl( int filedes, int dcmd, void * dev_data_ptr, size_t n_bytes, int * dev_info_ptr );
```

filedes : Un pointeur sur un descripteur de fichier que tu as obtenu par la fonction open (ouverture du système)
dcmd : une commande spécifique pour le système
dev_data_ptr : dépend de la commande, ce paramètre peut être :
• Un pointeur sur un buffer qui contient le message à faire passer au gestionnaire de ressource
• une zone tampon de données de réception venant du gestionnaire de ressource
• les 2 cas précédents
• NULL.

La fonction devctl() envoie une commande spécifique au système ou au gestionnaire de ressource

```
// ddki2cclient création d'une messagerie entre un applicatif client et le manageur de ressource  /dev/sample
// émission  du code commande
// 01 pour INITI2C __DIOTF
// 02  pour WRITEREGISTERI2C
// 03 pour READREGISTERI2C
// retour d'une information pour le code cmd 01 de la part du driver
//retour de 2 informations  pour le code cmd 02 de la part du driver


#include <errno.h>
        #include <stdio.h>
        #include <stddef.h>
        #include <stdlib.h>
        #include <unistd.h>
        #include <sys/iofunc.h>
        #include <sys/dispatch.h>
        #include <errno.h>
        #include <sys/types.h>
        #include <devctl.h>
        #include <fcntl.h>
typedef union _my_devctl_msg {
int txdata[2]; /* Filled by client on send */
int rxdata[2]; /* Filled by server on reply */
} data_t;
#define MY_CMD_CODE 1
#define MY_INIT_GETVAL __DIOTF(_DCMD_MISC, MY_CMD_CODE + 0, union _my_devctl_msg)
//#define MY_DEVCTL_SETVAL __DIOT(_DCMD_MISC, MY_CMD_CODE + 1, int)
#define MY_DEVCTL_GETVAL __DIOTF(_DCMD_MISC, MY_CMD_CODE + 1, union _my_devctl_msg)

int main(int argc, char **argv) {
int fd, ret, val;
data_t data;
if ((fd = open("/dev/sample", O_RDONLY)) == -1) {
return(1);
}
/* Find out what the value is set to initially */
//val = -1;
//ret = devctl(fd, MY_DEVCTL_GETVAL, &val, sizeof(val), NULL);
//printf("GET returned %d w/ server value %d \n", ret, val);
/* Set the value to something else */
//val = 25;
//ret = devctl(fd, MY_INIT_GETVAL, &val, sizeof(val), NULL);
//printf("SET returned %d \n", ret);
/* Verify we actually did set the value */
//val = -1;
//ret = devctl(fd, MY_DEVCTL_GETVAL, &val, sizeof(val), NULL);
//printf("GET returned %d w/ server value %d == 25? \n", ret, val);
/* Now do a set/get combination */
//data.txdata=0x01;

memset(&data, 0, sizeof(data));
```

```
data.txdata[0]=0x01;
data.txdata[1]=0x02;
//data.tx = 50;
// envoie du code commande 01 pour drivers et des infos 01 et 02 pour l'union data
ret = devctl(fd,  MY_INIT_GETVAL, &data, sizeof(data), NULL);
// réponse du driver au code cmd dans l'union data data.rxdata[0]
printf("SETGET returned with %d w/ server value %d == 25?\n",ret, data.rxdata[0]);
/* Check set/get worked */
//val = -1;
//data.txdata=0x02;
// envoie du code commande 02 pour drivers
ret = devctl(fd, MY_DEVCTL_GETVAL, &data, sizeof(data), NULL);
printf("GET returned %d w/ server value %d == 50? \n", ret, data.rxdata[0]);
printf("derniere valeure %d \n",  data.rxdata[1]);
// réponse du driver au code cmd dans l'union data data.rxdata[0]data.rxdata[1]
return(0);
}
```

Le programme serveur gestionnaire de ressource

/* Création d'un gestionnaire de ressource dont le nom de chemin est /dev/sample et qui répond à 3 types de messages */

```
        #include <errno.h>
        #include <stdio.h>
        #include <stddef.h>
        #include <stdlib.h>
        #include <unistd.h>
        #include <sys/iofunc.h>
        #include <sys/dispatch.h>
        #include <errno.h>
        #include <sys/types.h>
        #include <devctl.h>
typedef union _my_devctl_msg {
        int txdata[2]; /* Filled by client on send */
        int rxdata[2]; /* Filled by server on reply */
} data_t;
#define MY_CMD_CODE 1
#define INITI2C __DIOTF(_DCMD_MISC, MY_CMD_CODE + 0, union _my_devctl_msg)
//#define MY_DEVCTL_SETVAL __DIOT(_DCMD_MISC, MY_CMD_CODE + 1, int)
#define WRITEREGISTERI2C __DIOTF(_DCMD_MISC, MY_CMD_CODE + 1, union _my_devctl_msg)
#define READREGISTERI2C __DIOTF(_DCMD_MISC, MY_CMD_CODE + 2, union _my_devctl_msg)
static resmgr_connect_funcs_t connect_funcs;
static resmgr_io_funcs_t io_funcs;
static iofunc_attr_t attr;
int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg,RESMGR_OCB_T *ocb);
int global_integer = 0;
static char *buffer = "Hello world\n";


main(int argc, char **argv)
{
/* declare variables we'll be using */
resmgr_attr_t resmgr_attr;
dispatch_t *dpp;
dispatch_context_t *ctp;
int id;

/* initialize dispatch interface */
if((dpp = dispatch_create()) == NULL) {
fprintf(stderr,"%s: Unable to allocate dispatch handle.\n",argv[0]);
return EXIT_FAILURE;
}

/* initialize resource manager attributes */
memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;
```

```
/* initialize functions for handling messages */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,_RESMGR_IO_NFUNCS, &io_funcs);
io_funcs.devctl = io_devctl;

/* initialize attribute structure used by the device */
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);
attr.nbytes = strlen(buffer)+1;

/* attach our device name */
id = resmgr_attach(dpp, /* dispatch handle */&resmgr_attr, /* resource manager attrs */"/dev/sample", /*
device name */_FTYPE_ANY, /* open type */0, /* flags */&connect_funcs, /* connect routines */&io_funcs,
/* I/O routines */&attr); /* handle */
if(id == -1) {
fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
return EXIT_FAILURE;
}
/* allocate a context structure */
ctp = dispatch_context_alloc(dpp);

/* start the resource manager message loop */
while(1) {
if((ctp = dispatch_block(ctp)) == NULL) {
fprintf(stderr, "block error\n");
return EXIT_FAILURE;
}
dispatch_handler(ctp);
}
}
```

```
int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg,RESMGR_OCB_T *ocb) {
int nbytes, adress, value,status;
union { /* See note 1 */
data_t data;
//int data32;// ... other devctl types you can receive
} *rx_data;
if ((status = iofunc_devctl_default(ctp, msg, ocb)) !=_RESMGR_DEFAULT) {
return(status);
}
status = nbytes = 0;
rx_data = _DEVCTL_DATA(msg->i);// rx_data pointeur sur union= i structure _io_devctl
switch (msg->i.dcmd) {

case INITI2C: value=rx_data->data.txdata[0];rx_data->data.rxdata[0] = value;
nbytes = sizeof(rx_data->data.rxdata);break;
case WRITEREGISTERI2C: adress = rx_data->data.txdata[1];
value = rx_data->data.txdata[0];
nbytes = sizeof(rx_data->data.rxdata);
rx_data->data.rxdata[0] = adress;
rx_data->data.rxdata[1]=0xFF;
break;
case READREGISTERI2C:break;
// See note 4. The rx data overwrites the tx data for this command.
default:return(ENOSYS);
```

```
}
memset(&msg->o, 0, sizeof(msg->o)); //O structure du type _io_devctl_reply ;
msg->o.ret_val = status;
msg->o.nbytes = nbytes;
return(_RESMGR_PTR(ctp, &msg->o, sizeof(msg->o) + nbytes));
}
```

POSIX : POSIX est le nom d'une famille de standards définie par l'IEEE et formellement désignés IEEE 1003. Ces standards ont émergé d'un projet de standardisation des APIs des logiciels destinés à fonctionner sur des variantes du système d'exploitation UNIX. Le terme POSIX a été suggéré par Richard Stallman en réponse à la demande de l'IEEE pour un nom facilement memorisable. C'est un acronyme de Portable Operating System Interface, dont le X exprime l'héritage UNIX de l'API.