



Le système d'exploitation temps réel VXWORKS7

TABLE DES MATIERES

Le système d'exploitation temps réel Vxworsks

1) Historique page N°3

2) Présentation page N°3

3) La plateforme matérielle embarquée Raspberry PI4 page N°6

3.1) Architecture

3.2) L'accès au système d'exploitation

4) Les taches temps réels dans l'espace noyau de vxworks page N°14

5) Les processus temps réel RTP page N°26

6) L'ordonnancement sous VxWorks page N°31

7) Séquence de boot vxworks7 sur la plateforme Raspberry PI4 page N°36

8) Le device tree page N°42

9) Les pilotes de périphériques page N°44

9.1) L'architecture VxBus

9.2) Description en langage C d'un périphérique

9.3) La gestion des ressources mémoires

9.4) La gestion des ressources physiques

10) Association d'un périphérique à un système d' entrées sorties page N°58

10.1) Principe

10.2) Les périphériques de type caractère

10.3) ouverture d'un périphérique d'entrées sorties

10.4) lecture d'un périphérique d'entrées sorties

11) Création d'un environnement autonome sur SDCARD page N°65

11.1) Présentation

11.2) La ROMFS

11.3) Lancement d'un DKM au démarrage

11.4) Lancement d'un RTP au démarrage

12) Programmation du GPIO : Les broches d'entrées sorties page N°70

13) Le bus I2C page N°74

14) Le bus SPI page N°83

15) Le contrôleur PWM page N°87

Le système d'exploitation temps réel Vxworsks

1) Historique

VxWorks est un système d'exploitation temps réel multitâche, généralement utilisé dans les systèmes embarqués.

Ce système d'exploitation propriétaire, développé par la société Wind River (acquise par Intel en 2009), a été employé par la NASA pour les missions spatiales

VxWorks va se doter de fonctionnalités novatrices qui le rapproche des standards du marché, notamment grâce à sa compatibilité avec la norme POSIX, python, la containerisation. Il est principalement employé par la recherche et l'industrie (aéronautique, automobile, transport, télécommunication, militaires)

2) Présentation

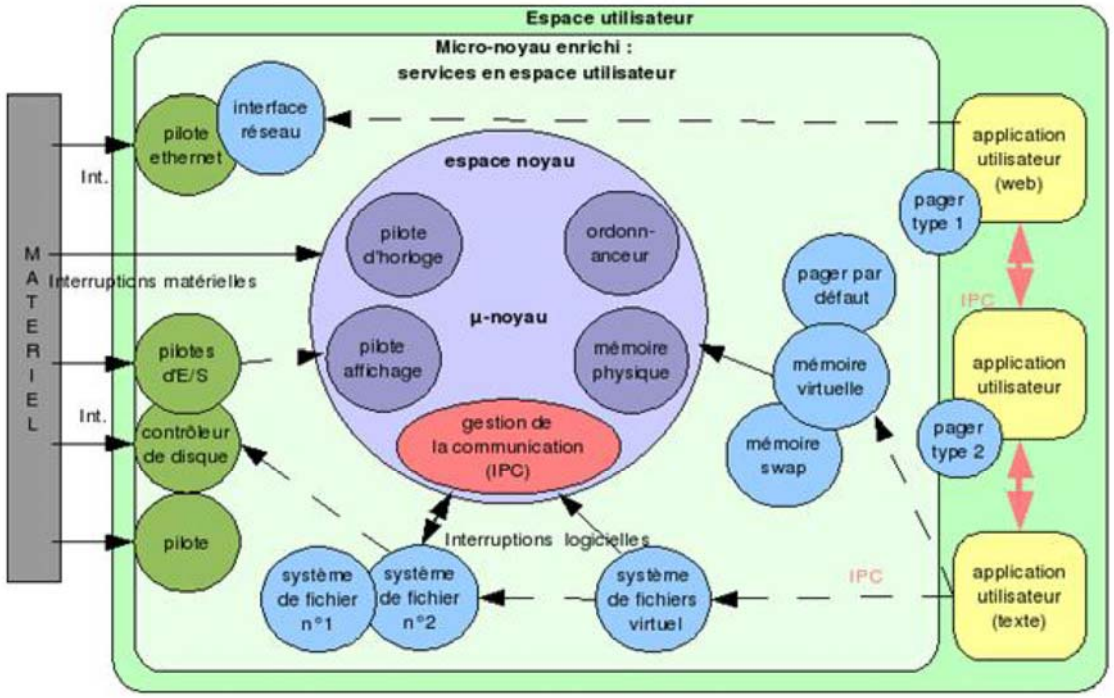
Vxworks est fondé sur un micro noyau (Microkernel) d'une taille de quelques Koctets qui gère un groupe de processus systèmes. Ce noyau de petite taille lui confère fonctionnalité et performance d'un executif temps réel Le Microkernel possède quatre fonctions fondamentales :

Transmission de messages ou IPC (interprocess communication) : Le Microkernel manie les messages des processus à travers tout le système.

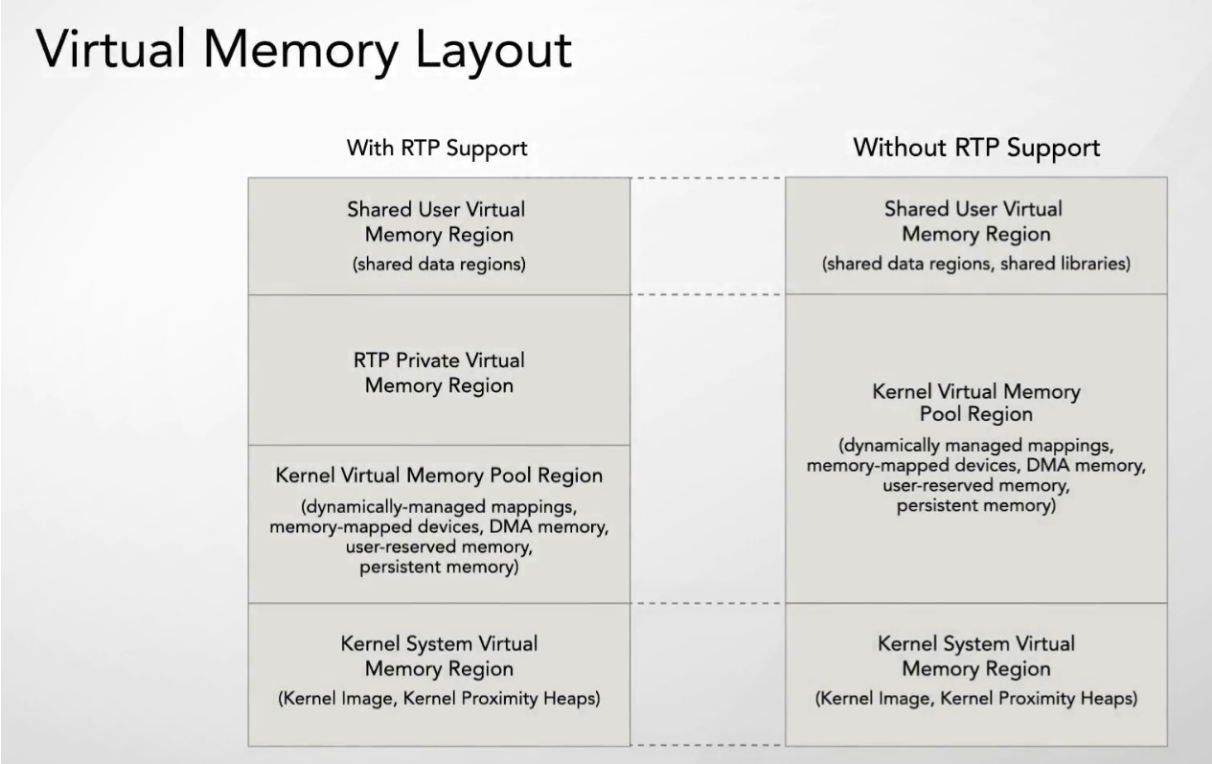
Gestion des taches ou scheduler : le micro noyau travaille en étroite collaboration avec le gestionnaire de tache. Il est chargé d'attribuer les ressources du système aux taches qui le demande. C'est de cette partie que nous traiterons par la suite. Le micro noyau est exécuté suite à un appel système 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.



L'Adressage est virtuelle mais pas de protection mémoire dans l'espace noyau . Il existe un espace d'adressage noyau pour les taches utilisateurs noyau DKM et un espace utilisateur pour les processus temps réels (RTP) dans le système.



Un seul espace d'adressage unique au niveau du noyau implique :

- ✓ Une Corruption possible des données de l'exécutif par les applications tout le monde voit tout. Pas de protection mémoire.
- ✓ Une Cohérence des données partagées entre les tâches nécessite donc des outils de synchronisation. D'où les abstractions suivantes de l'exécutif :
- Tâches.
- Sémaphores.
- File de messages et pipes.

Pourquoi un seul espace d'adressage noyau :

- ✓ Contexte de tâche plus petit.
- ✓ Pas d'appel système : les services sont invoqués par un simple appel de fonction .
- ✓ Toutes les tâches applicatives sont des tâches "superviseurs/privilégiées".
- ✓ Communication entre tâches sans traverser le système d'exploitation.

= gain de temps

= gain de "place" mémoire, de ressources

= gain de prédictiblité (cf. contraintes systèmes embarqués temps réel).

Les conseils à suivre face aux exigences du multitache temps réel

• Comportement déterministe

- Une réponse tardive est une mauvaise réponse

- Une réponse hative peut être une mauvaise réponse

• Conçu pour une vitesse élevée - Réponse rapide - Faible surcharge

• Code correctement écrit

- N'utilise le temps CPU qu'en cas de besoin

Retarde ou suspend lorsqu'il n'y a aucun travail à faire

- Pas de scrutation continue ou d'attente infinie

3) La plateforme matérielle embarquée Raspberry PI4

3.1) Architecture

En TP on utilisera la plateforme :

- RAspberry PI 4 pour vxworks

La RAspberry PI 4 est une carte CPU autonome à faible cout pour des systèmes embarqués non industriels. Elle fait partie d'un ensemble de nano-ordinateurs qui ont su évoluer au fil des années.

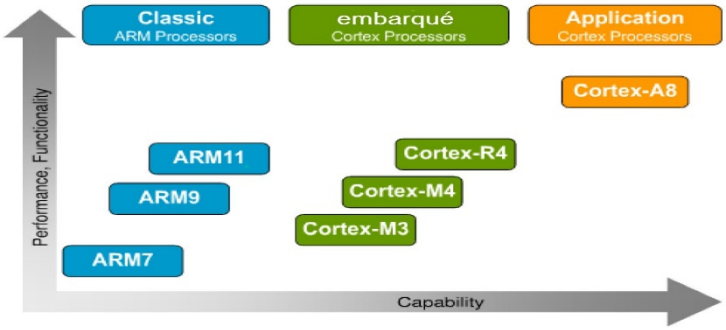
	Raspberry Pi 3 Model B	Raspberry Pi Zero	Raspberry Pi 2 Model B	Raspberry Pi Model B+
Introduction Date	2/29/2016	11/25/2015	2/2/2015	7/14/2014
SoC	BCM2837	BCM2835	BCM2836	BCM2835
CPU	Quad Cortex A53 @ 1.2GHz	ARM11 @ 1GHz	Quad Cortex A7 @ 900MHz	ARM11 @ 700MHz
Instruction set	ARMv8-A	ARMv6	ARMv7-A	ARMv6
GPU	400MHz VideoCore IV	250MHz VideoCore IV	250MHz VideoCore IV	250MHz VideoCore IV
RAM	1GB SDRAM	512 MB SDRAM	1GB SDRAM	512MB SDRAM
Storage	micro-SD	micro-SD	micro-SD	micro-SD
Ethernet	10/100	none	10/100	10/100
Wireless	802.11n / Bluetooth 4.0	none	none	none
Video Output	HDMI / Composite	HDMI / Composite	HDMI / Composite	HDMI / Composite
Audio Output	HDMI / Headphone	HDMI	HDMI / Headphone	HDMI / Headphone
GPIO	40	40	40	40

Raspberry Pi 4B vs 3B+

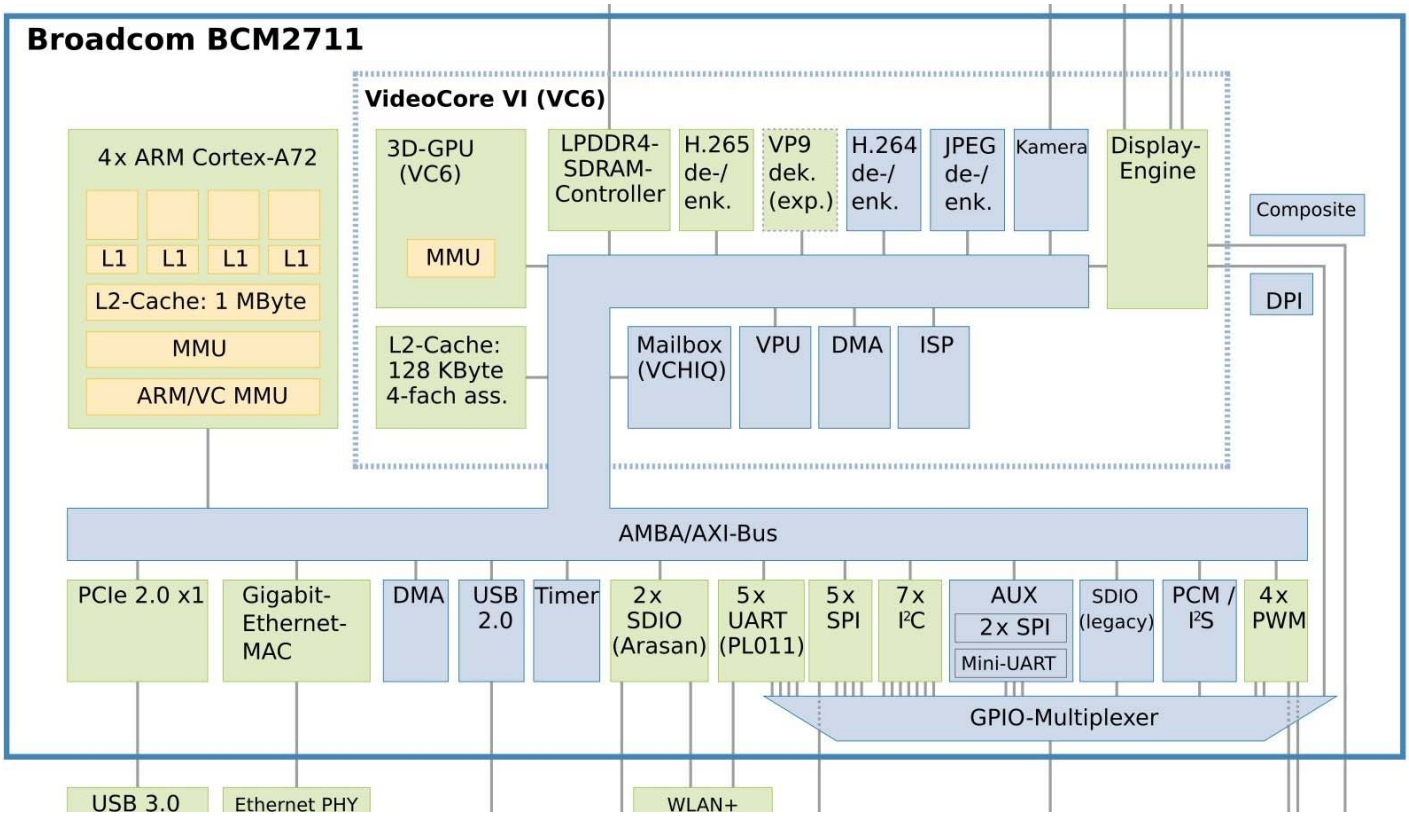


Features/Specs	Raspberry Pi 4 Model B	Raspberry Pi 3 Model B+
Release Date	24th June 2019	14th March 2018
SoC Type (Processor)	Broadcom BCM2711 (with metal cover)	Broadcom BCM2837B0 (with metal cover)
Core Type	Cortex-A72 64-bit (ARMv8)	Cortex-A53 64-bit (ARMv8)
No. of Cores	Quad-Core	
GPU	VideoCore VI	VideoCore IV
Multimedia	H.265 decode (4Kp60) H.264 decode (1080p60) H.264 encode (1080p30) OpenGL ES 1.1, 2.0, 3.0 Graphics	H.264, MPEG-4 decode (1080p30) H.264 encode (1080p30) OpenGL ES 1.1, 2.0 Graphics
CPU Clock	1.5 GHz	1.4 GHz
Memory/OS storage	microSD	
RAM	LPDDR4: 1GB, 2GB, 4GB and 8GB options	LPDDR2 1GB
Ethernet	True Gigabit Ethernet	Gigabit over USB 2.0 (Max 300Mbps)
USB Port	2 x USB 3.0 + 2 x USB 2.0	4 x USB 2.0
HDMI	2 x micro HDMI support Dual Display	1 x full size HDMI
WiFi	802.11 b/g/n/ac (2.4GHz+5GHz & Shielded)	
Bluetooth	5.0 + BLE (Shielded)	4.2 + BLE (Shielded)
Antenna	PCB Antenna (Similar to Rpi Zero W)	
GPIO	40 pins (Fully backwards-compatible with previous boards)	
Operating System	Raspbian (> 24 June 2019)	Raspbian (> March 2018)
Dimension	85mm x 56mm	
Power Input	5V via USB Type C (upto 3A) 5V via GPIO header (upto 3A) Power over Ethernet, requires PoE HAT	5V via USB Micro B (upto 2.5A) 5V via GPIO header (upto 3A) Power over Ethernet, requires PoE HAT

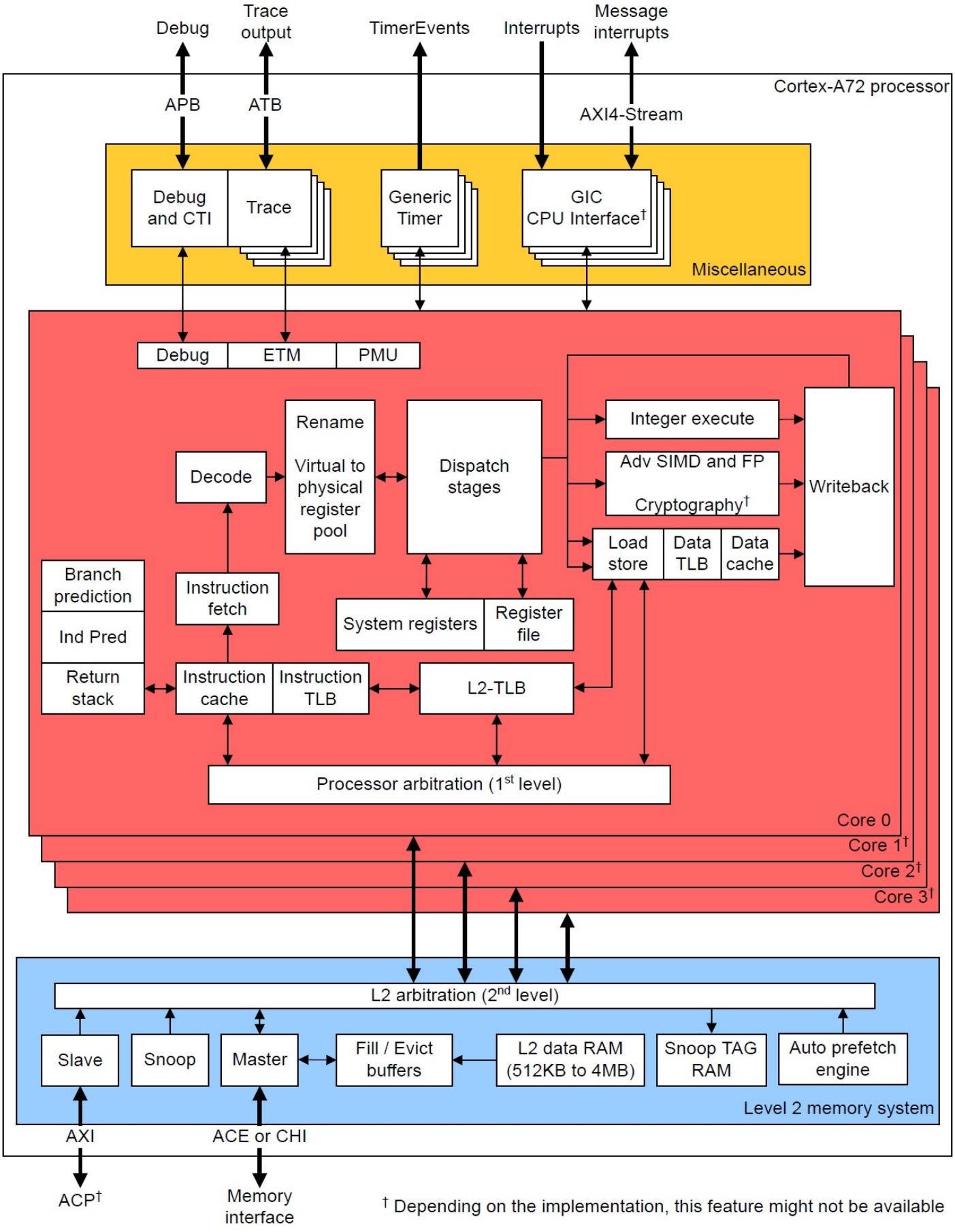
Le processeur utilisé sur la PI4 est compatible avec les processeurs ARM cortex A-7 A-8 (des processeurs d'application). La société ARM propose une gamme importante de microprocesseur répondant à une multitude d'application.

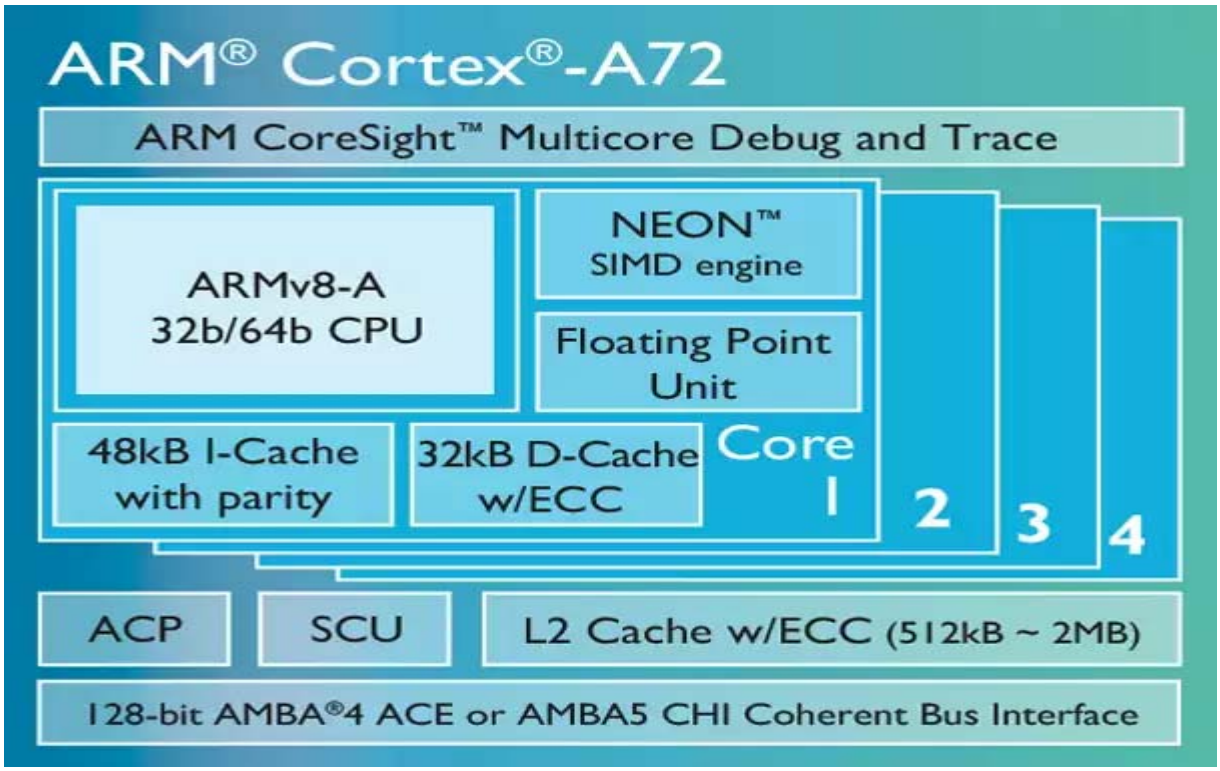


Le processeur BCM2711 sur la carte Raspberry PI 4 fait partie de la famille de systèmes sur puces **(SOC)** et basée sur une architecture ARM à faible consommation L'architecture du BCM2711 est donnée ci-dessous :



Le SOC **BCM2711** est doté de quatre processeurs ARM CORTEX A72 , avec des unités de calcul à virgule flottante.

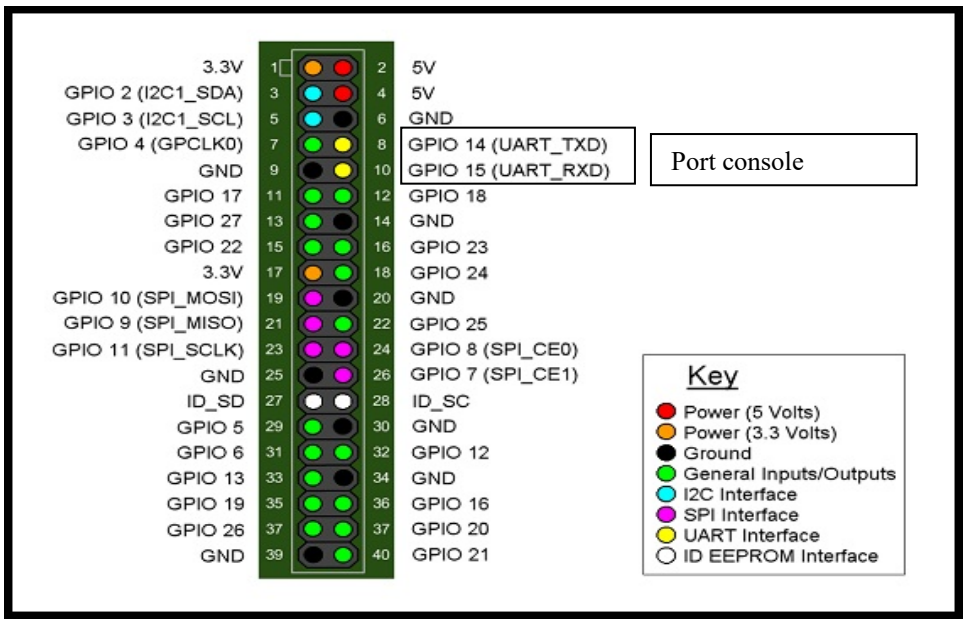

Debug
Trace output
TimerEvents
Interrupts
Message interrupts
APB
ATB
AXI4-Stream
Cortex-A72 processor
Debug and CTI
Trace
Generic Timer
GIC CPU Interface†
Miscellaneous
Debug
ETM
PMU
Rename
Virtual to physical register pool
Decode
Dispatch stages
Integer execute
Adv SIMD and FP
Cryptography†
Writeback
Load store
Data TLB
Data cache
Branch prediction
Ind Pred
Return stack
Instruction fetch
System registers
Register file
Instruction cache
Instruction TLB
L2-TLB
Processor arbitration (1st level)
Core 0
Core 1†
Core 2†
Core 3†
L2 arbitration (2nd level)
Slave
Snoop
Master
Fill / Evict buffers
L2 data RAM (512KB to 4MB)
Snoop TAG RAM
Auto prefetch engine
Level 2 memory system
AXI
ACE or CHI
ACP†
Memory interface
† Depending on the implementation, this feature might not be available



Tous ces éléments sont intégrés sur la même puce et conviennent pour une multitude d'application dont du traitement vidéo HD. Ces applications peuvent être :

- Systèmes de navigation
- Lecteur multimédia
- Systèmes de surveillance médicale
- Vision industriel
- Communication portable
- Tablette graphique

Son connecteur d'extension de 40 broches a le format suivant :

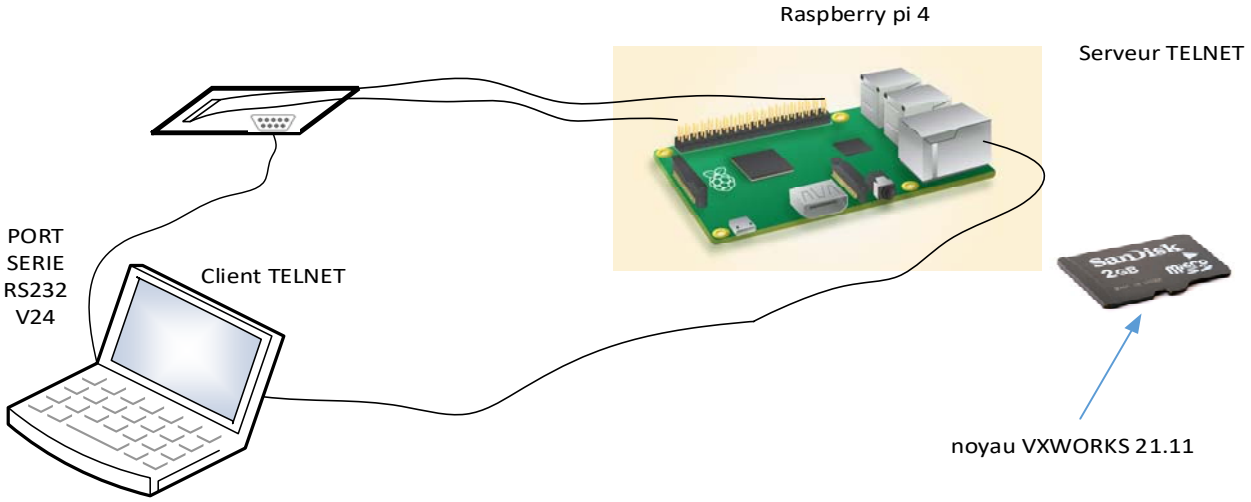


Cette carte peut accueillir un système d'exploitation type Linux embarqué Xenomai Android …

3.2) L'accès au système d'exploitation

On peut comme beaucoup d'environnement embarqué (calculateur) prendre la main à distance sur la carte grâce à des services réseaux préinstallés :

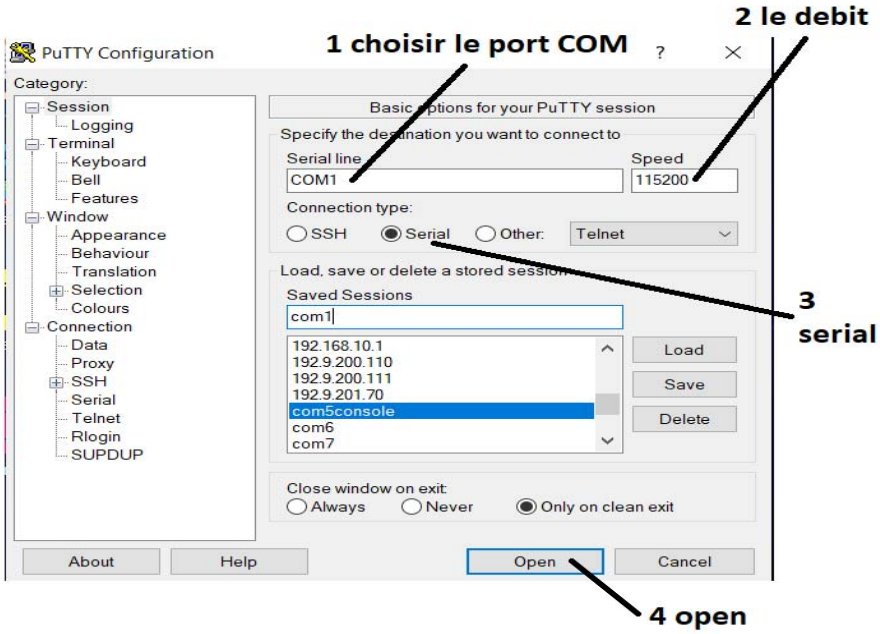
- En mode console client et serveur TELNET
- En utilisant le port console sur une liaison série RS232 V24



Ouvrir Putty sur l'environnement windows :



Configurez putty avec la configuration de la liaison série RS232



Le chargeur de boot et le noyau Vxworks se lancent :

```
 _______          _______
 \.......\        /......./
  \.......\      /......./
   \.......\    /......./
    \.......\  /......./
     \.......\ \...../             VxWorks SMP 64-bit
      \.......\ \.../
       \.......\ \./     Release version: 21.11
        \.......\  -     Build date: Mar 27 2022 10:25:38
         \.......\
          \......./       Copyright Wind River Systems, Inc.
           \...../  -                1984-2022
            \.../  /.\
             \./  /...\
              -  -------

                 Board: Raspberry Pi 4 Model B - ARMv8
             CPU Count: 4
         OS Memory Size: ~883MB
```

L'interpréteur de code est en attente. Celui-ci attend du code écrit en langage C

```
->
```

Pour passer sur l'interpréteur de commande tapez cmd

```
-> cmd
[vxWorks *]#
```

Pour connaitre notre configuration réseau afin d'utiliser le service TELNET tapez la commande suivante :

```
[vxWorks *]# ifconfig -a
lo0     Link type:Local loopback
        inet 127.0.0.1  mask 255.255.255.255
        inet6 unicast fe80::1%lo0  prefixlen 64  automatic
        inet6 unicast ::1  prefixlen 128
        UP RUNNING LOOPBACK MULTICAST NOARP ALLMULTI
        MTU:1500  metric:1  VR:0  ifindex:1
        RX packets:87 mcast:0 errors:0 dropped:0
        TX packets:87 mcast:0 errors:0
        collisions:0 unsupported proto:0
        RX bytes:26432 (26 k)  TX bytes:26432 (26 k)

genet0  Link type:Ethernet  HWaddr dc:a6:32:e3:45:bc
        inet 192.9.201.70  mask 255.255.255.0  broadcast 192.9.201.255
        inet6 unicast fe80::dea6:32ff:fee3:45bc%genet0  prefixlen 64  automatic
        inet6 unicast 3ffe:1:2:3::4  prefixlen 64
        UP RUNNING SIMPLEX BROADCAST MULTICAST DHCP
        MTU:1500  metric:1  VR:0  ifindex:2
        RX packets:0 mcast:0 errors:0 dropped:0
        TX packets:0 mcast:7 errors:0
        collisions:0 unsupported proto:0
        RX bytes:0  TX bytes:0

[vxWorks *]#
```

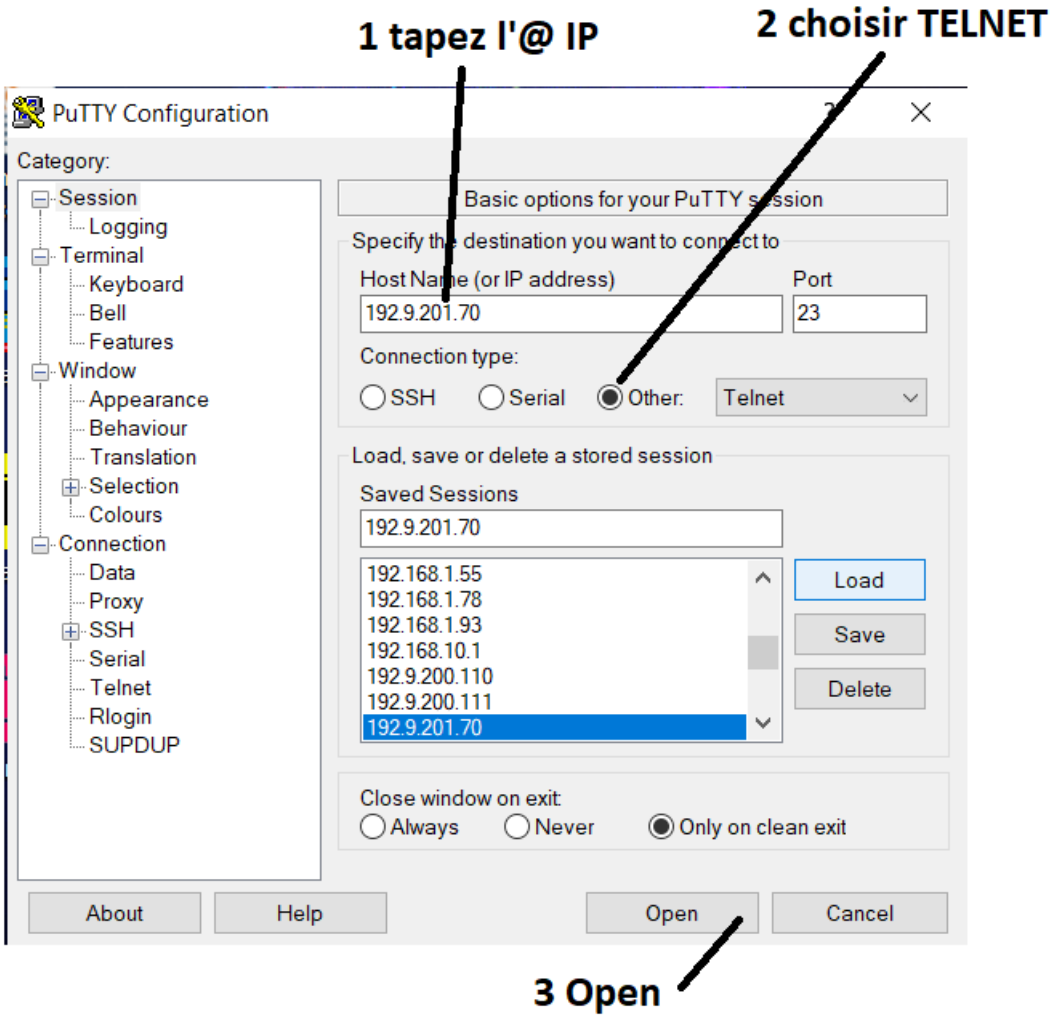
Notre adresse IP apparait dans l'encadré. Celle-ci est statique et codée en dure lors de la construction du noyau Vxworks. Le système de fichier utilisé est du type FAT et on peut voir les répertoires de la racine du noyau :

```
[vxWorks *]# cd /
[vxWorks *]# ls
.
..
dev
tyCo
I2C
SPI
PWM
ram
ram0
tmp
ttyS0
ttyS1
romfs
fifos
null
tty
shm
devm
vxbus
[vxWorks *]#
```

Pour revenir à l'interpréteur de code tapez C

```
[vxWorks *]# C
->
```

On va par la suite se connecter avec le service réseau TELNET



L'interpréteur de code est en attente. Celui-ci attend du code écrit en langage C

```
->
```

En terme de programmation VxWorks dispose de deux types de tache :

- ✓ Les taches noyaux ou DKM (downloadable module kernel)
- ✓ Les taches main ou thread encapsulées dans un processus temps réel (RTP real time processus)

La suite de ce polycopié va nous permettre d'aborder ces différents éléments logiciels

Les taches temps réels dans l'espace noyau de vxworks

4) Présentation

Elles sont créés comme des DKM (downloadable kernel modules)

tache = unité exécution du code
Un fil exécution : le PC Registres du processeur
Un point entrée , Un état ,une priorité

- La tâche n'est pas le code

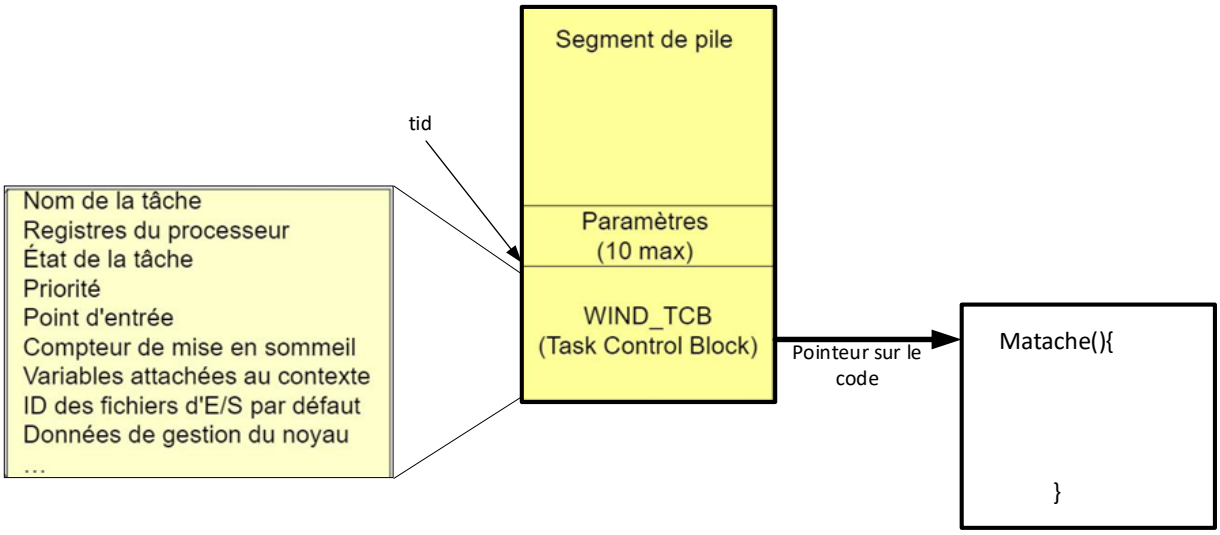
Deux taches distinctes peuvent être associées au même segment de code (mais segments de données et de pile différents)

- Toutes les taches partagent le même espace d'adressage celle du noyau

Les taches sont gérées par leur ID (Identificateur TID)

- Chaque tache a son propre contexte de sauvegarde dans le TCB (Task Control Block)

Une tache d'un DKM en mémoire peut se représenter de la façon suivante :



L'utilisation de l'espace du noyau donne un accès direct aux fonctionnalités et aux E/S pour une vitesse maximale ;

Pour basculer une nouvelle tâche en exécution, le noyau doit :

- Sauvegarder le contexte de la tâche en cours d'exécution dans la structure **TCB** de celle-ci.

- Restaurer le contexte de la nouvelle tâche à exécuter à partir de sa structure **TCB**.

La structure **TCB** (Task Control Block) est définie par la structure de données **WIND_TCB** dans le fichier **taskLib.h.** Elle contient les informations sur le contexte CPU, pc, sp, registres cpu, registres fpu ainsi que sur la tâche, priorité, flags, etc…

Le noyau gère une file particulière pour chacun des états possibles des tâches.

```
-> i

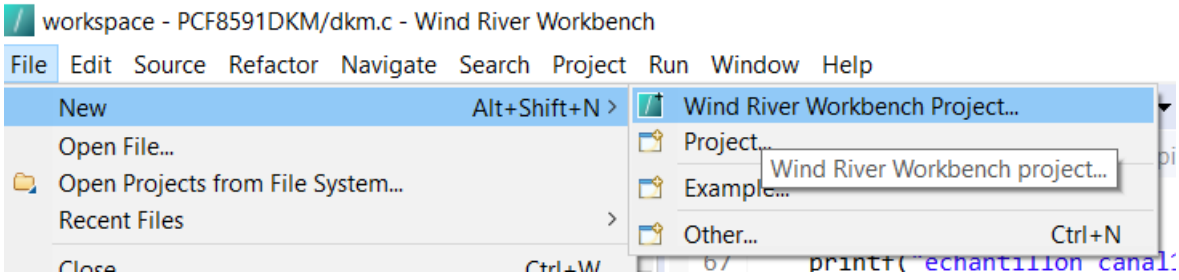
  NAME            TID       PRI   STATUS        PC           ERRNO  CPU #
---------- ---------------- --- ---------- ---------------- ------- -----
tIsr0       ffff800000034f00   0 PEND       ffffffff805cc8c8       0     -
tJobTask    ffff80000005dac0   0 PEND       ffffffff804e7cf4       0     -
tExcTask    ffffffff80a23a88   0 PEND       ffffffff805cc8c8       0     -
tLogTask    ffff800000060180   0 PEND       ffffffff805c9f4c       0     -
/sd0XbdSvc  ffff8000003b6a90   1 PEND       ffffffff805cc8c8       0     -
tShell0     ffff800000444d90   1 PEND       ffffffff805cc8c8       0     -
tShellRem1> ffff8000004569a0   1 PEND       ffffffff805cc8c8       0     -
tShellRem1> ffff800000453c90   1 READY      ffffffff805dfd3c       0     2
```

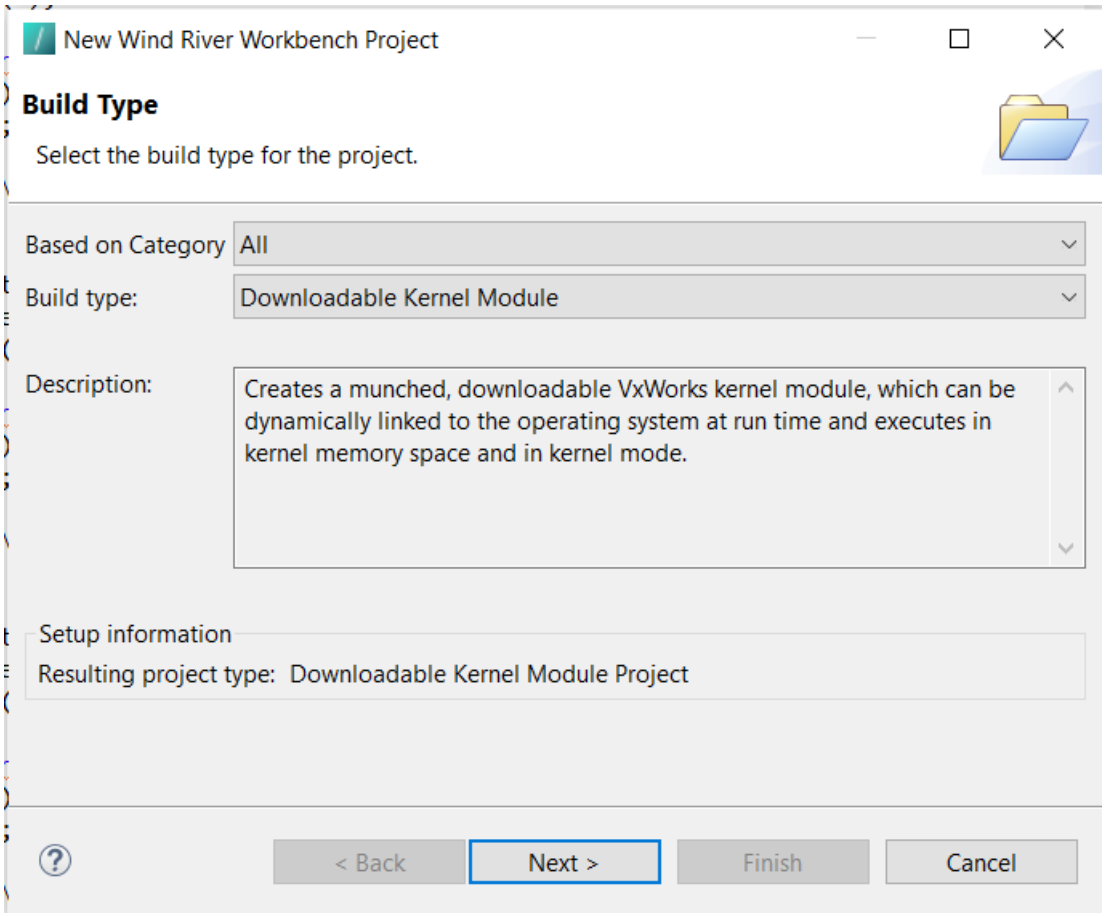
Pour créer une tache noyau

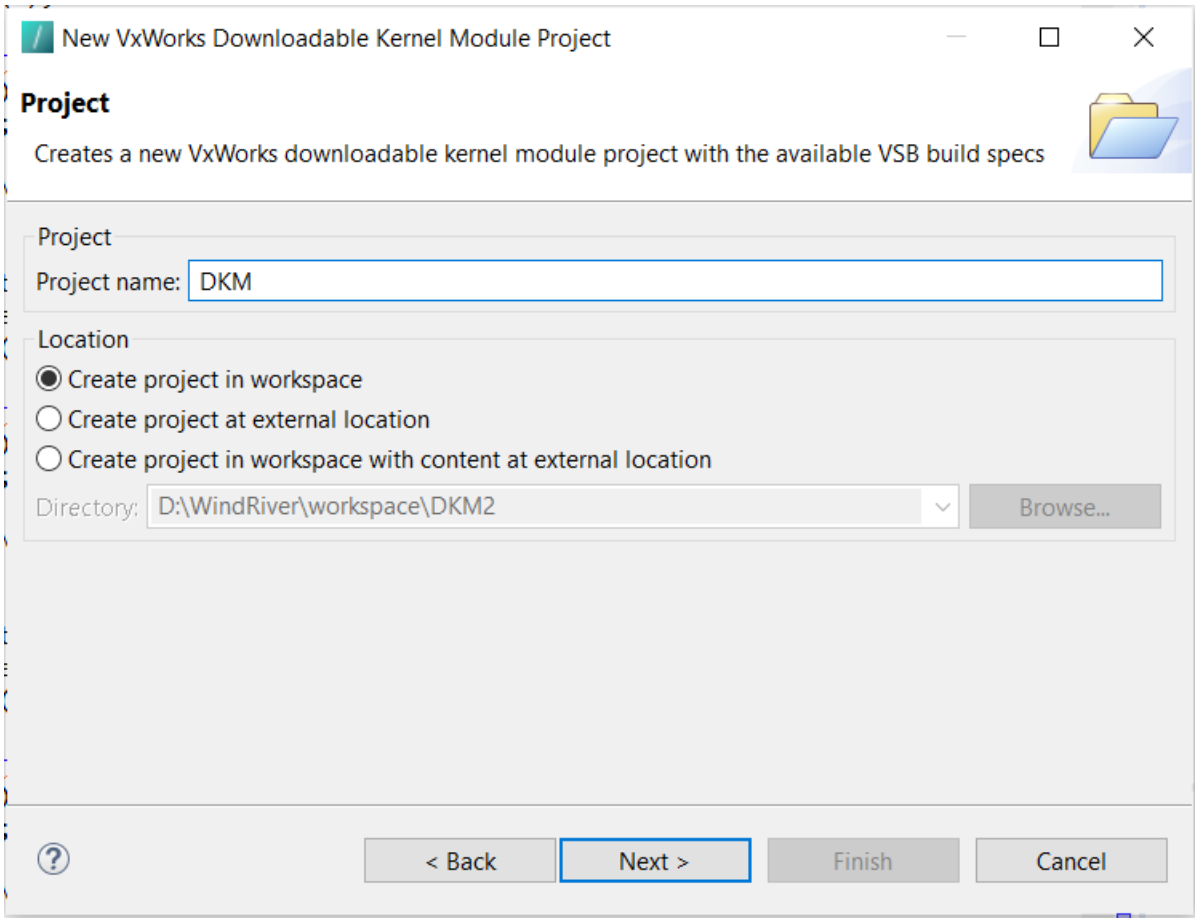
```
int taskSpawn
(
char * name, /* name of new task (stored at pStackBase) */
int priority, /* priority of new task */
int options, /* task option word */
int stackSize, /* size (bytes) of stack needed plus name */
FUNCPTR entryPt, // address of code to execute
int arg1, /* 1st of 10 req'd task args to pass to func */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8,
int arg9,
int arg10
)
```

Mise en place d'une tache noyau avec l'outil workbench

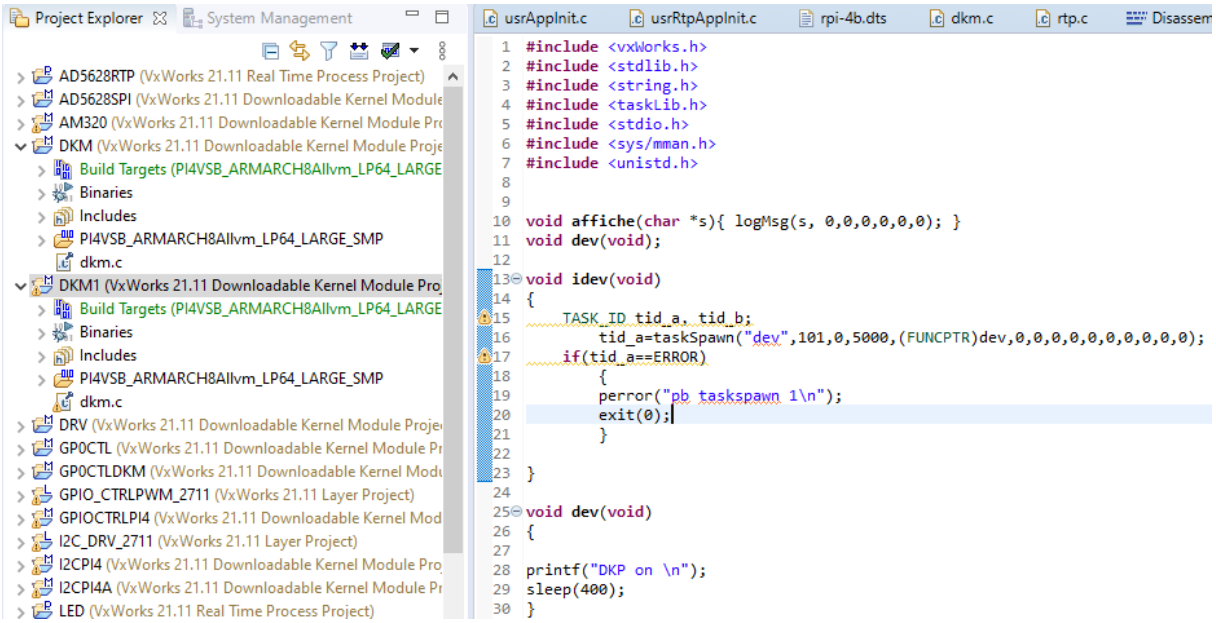
Sous workbench, créez un projet pour un module noyau DKM




New Wind River Workbench Project
Build Type
Select the build type for the project.
Based on Category
All
Build type:
Downloadable Kernel Module
Description:
Creates a munched, downloadable VxWorks kernel module, which can be dynamically linked to the operating system at run time and executes in kernel memory space and in kernel mode.
Setup information
Resulting project type: Downloadable Kernel Module Project
< Back
Next >
Finish
Cancel

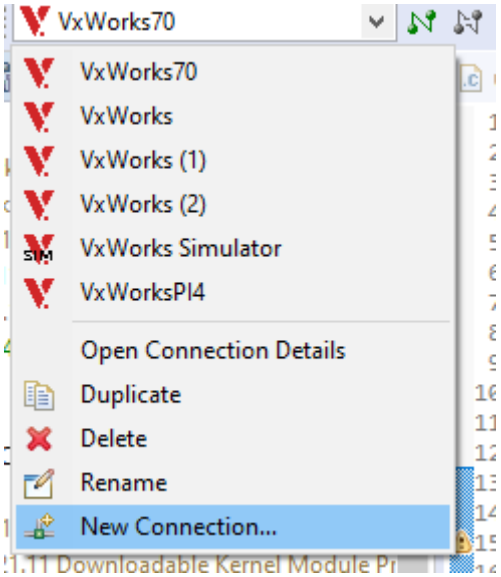

New VxWorks Downloadable Kernel Module Project
Project
Creates a new VxWorks downloadable kernel module project with the available VSB build specs
Project
Project name:
DKM
Location
Create project in workspace
Create project at external location
Create project in workspace with content at external location
Directory:
D:\WindRiver\workspace\DKM2
Browse...
< Back
Next >
Finish
Cancel

Complétez le code source suivant

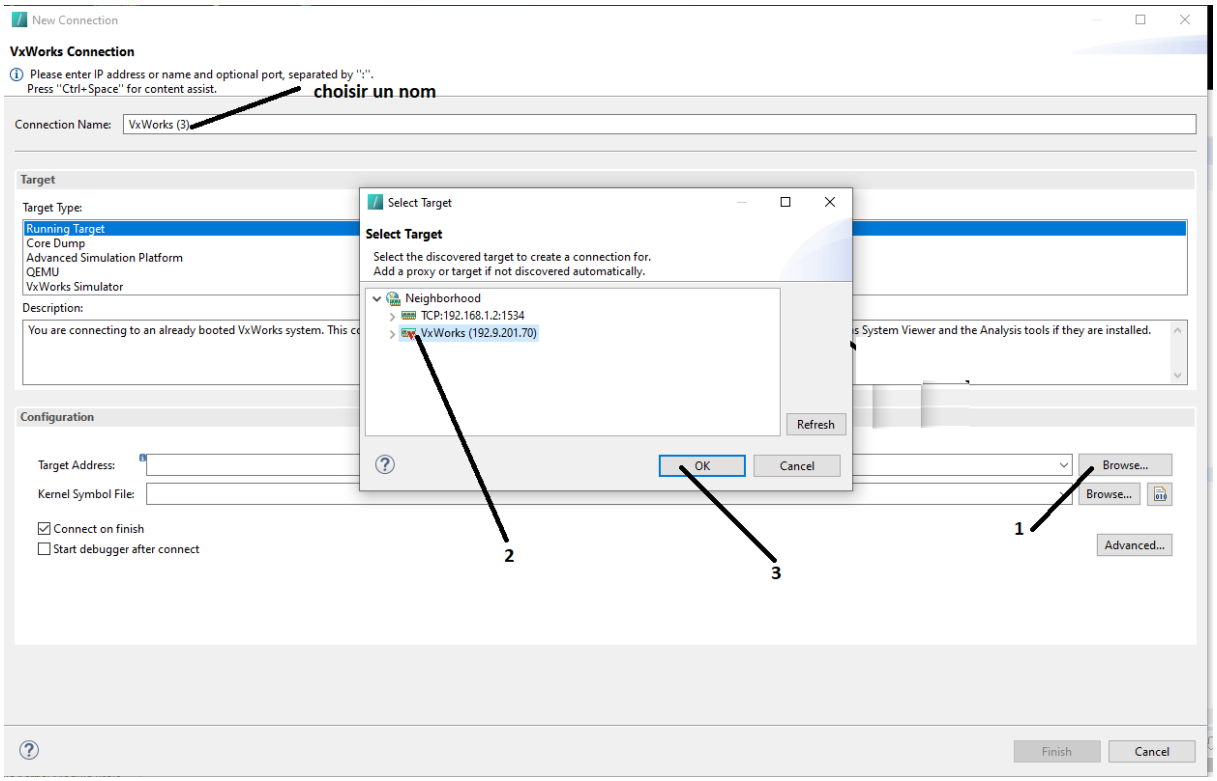


- Quelle est la dimension de la pile ?
- Quelle est la priorité de la tache ?
- Comment s'appelle la fonction associée à cette tache ?

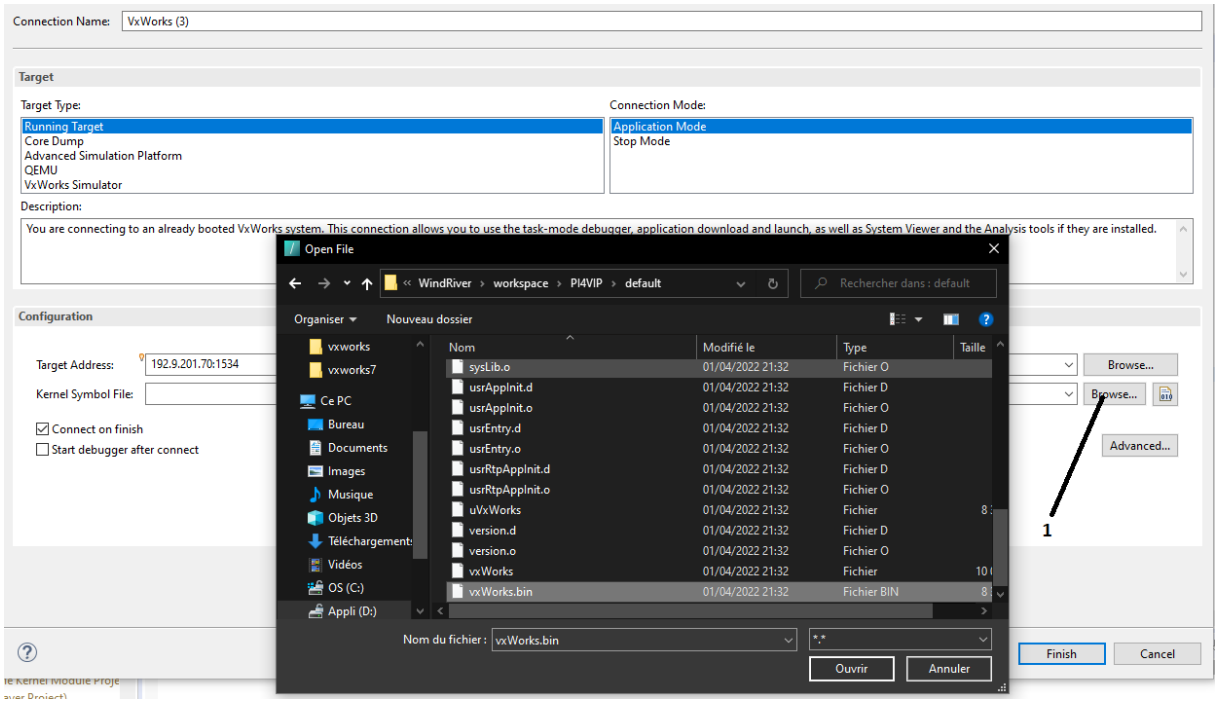
Cliquez en haut de la page sur la fenêtre de connexion, Sélectionnez new connection



Choisir les 4 étapes suivantes



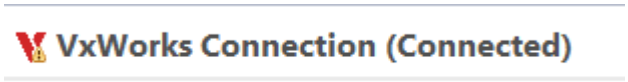
Indiquez ou se situe l'image du noyau vxworks.bin dans PI4VIP



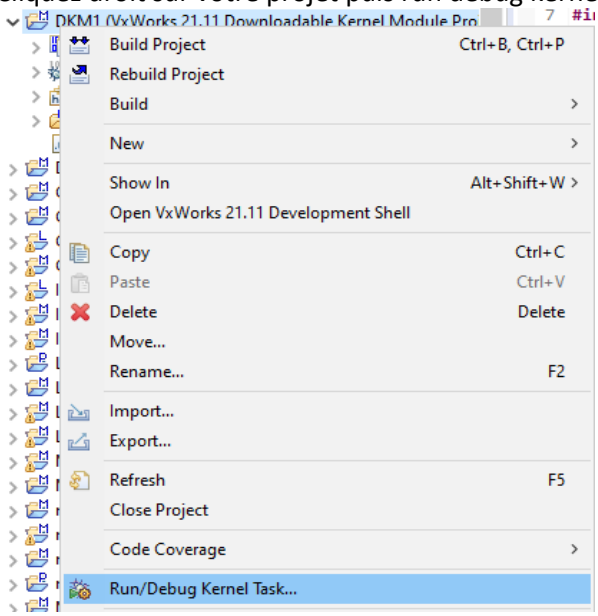
Cliquez sur finish



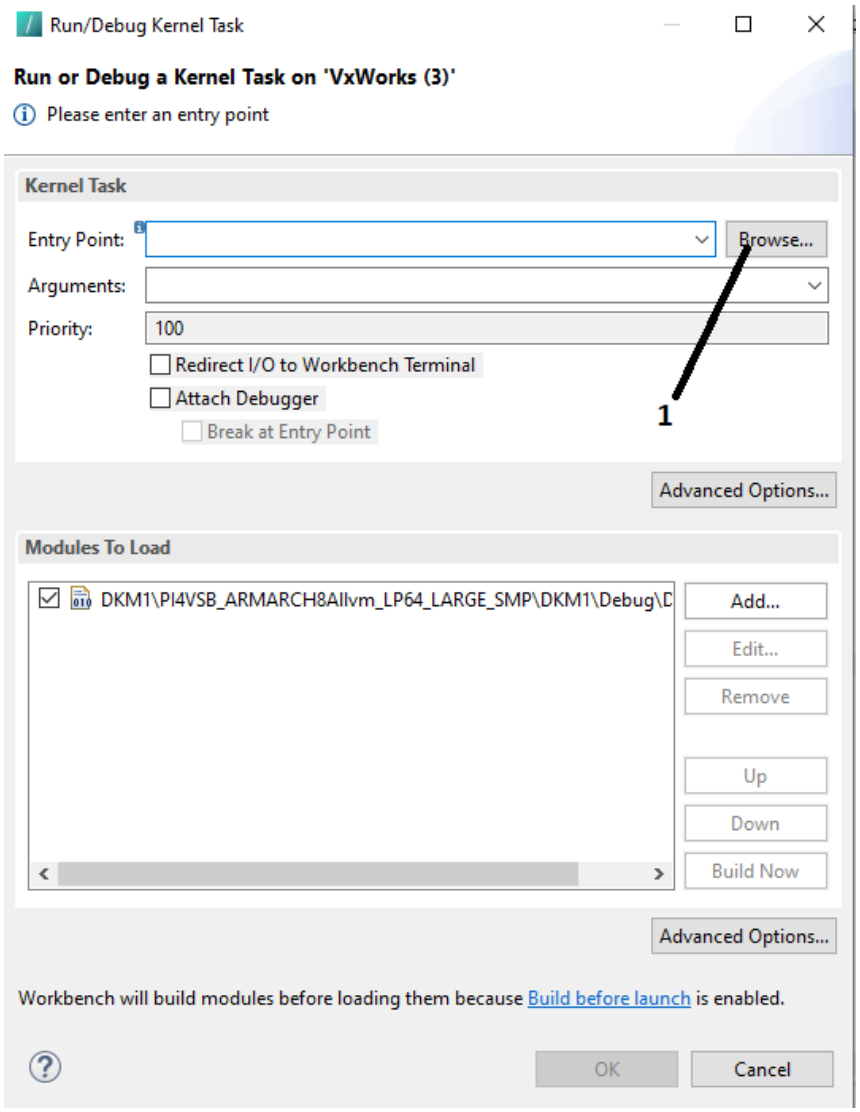
La connexion a réussi



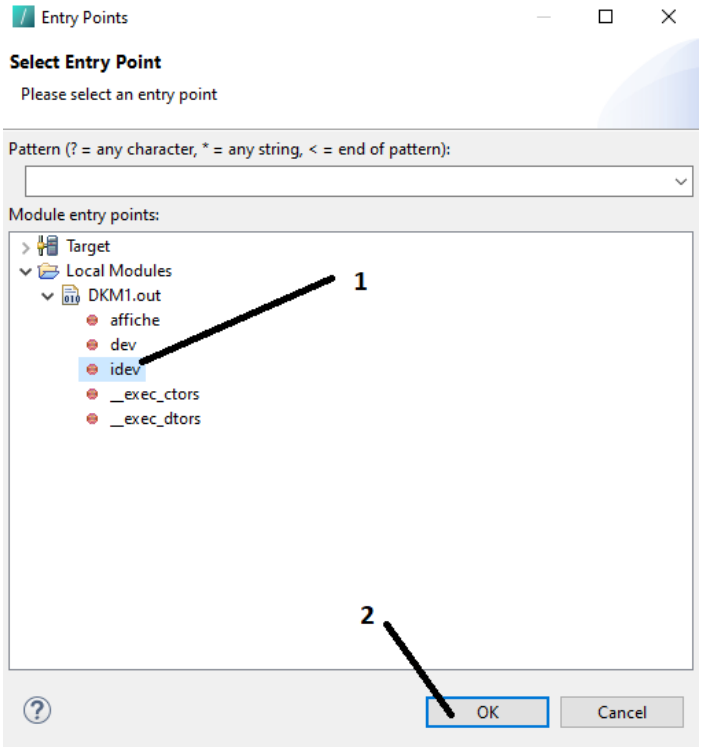
Cliquez droit sur votre projet puis run debug kernel



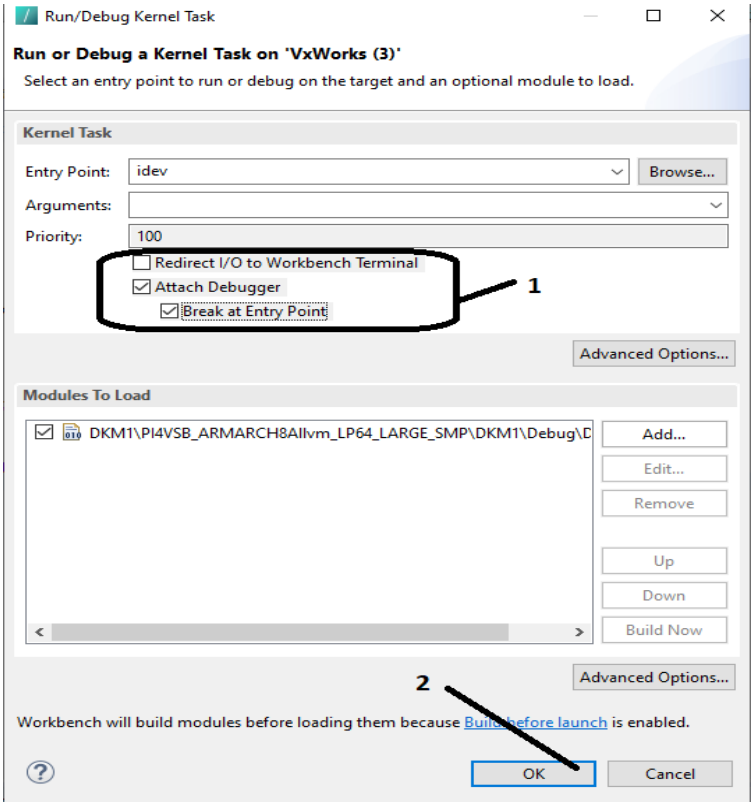
Sélectionnez notre module



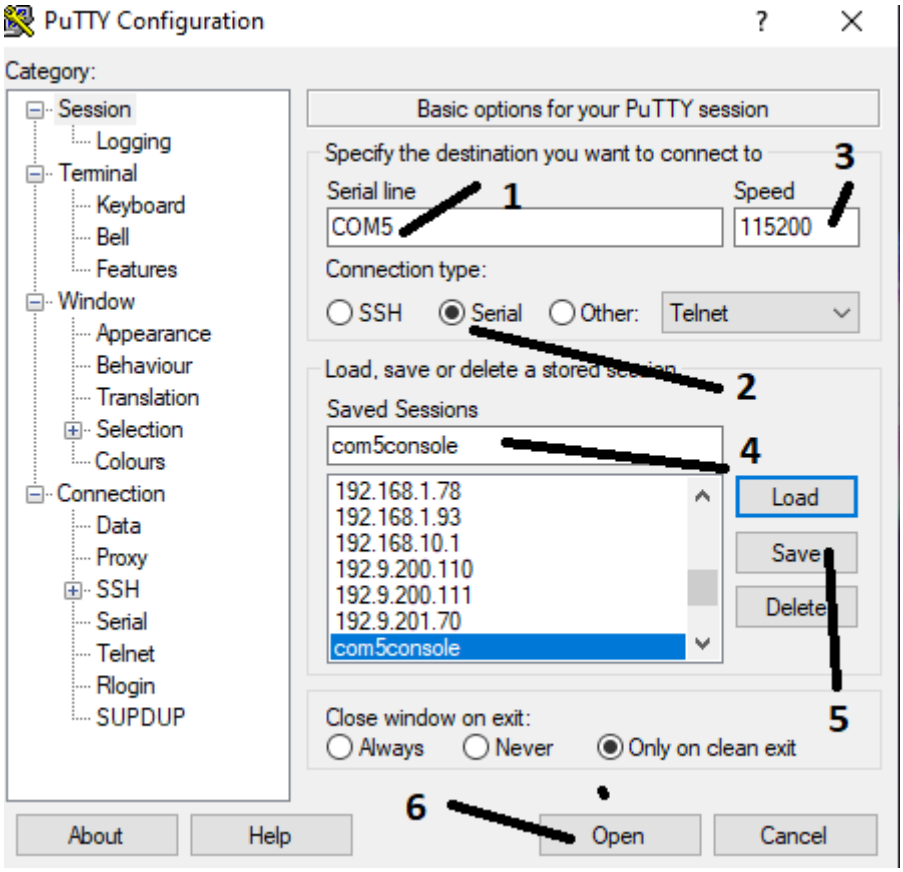
Sélectionnez le nom de la tache principal



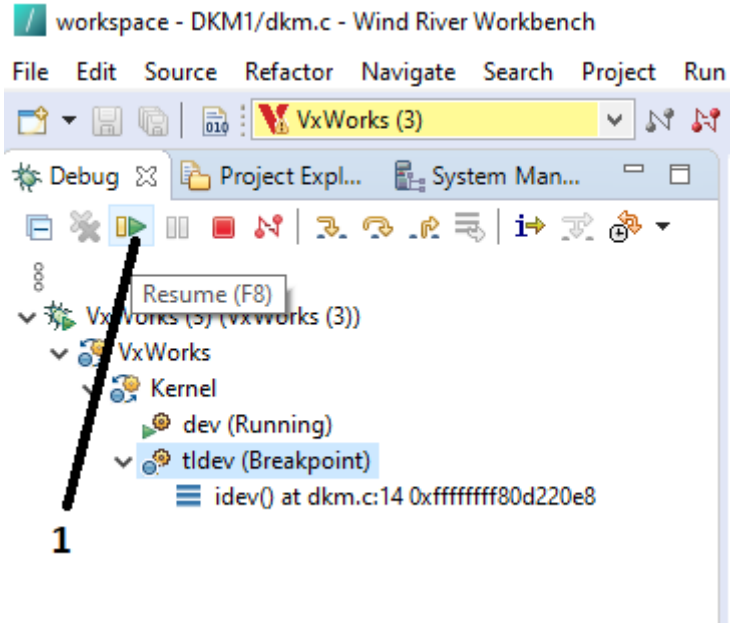
Suivez les deux étapes



Se connectez sur le port série , le port console avec l'utilitaire putty



Lancez notre tache





On va se déplacer vers l'image du module présent sur le PC

```
-> cmd
[vxWorks *]# cd /
[vxWorks *]# ls
.
..
dev
tyCo
I2C
SPI
PWM
ram
ram0
tmp
ttyS0
ttyS1
romfs
fifos
hostfs
null
tty
shm
devm
vxbus
[vxWorks *]#
```

```
[vxWorks *]# cd hostfs
[vxWorks *]# cd ws
[vxWorks *]#
```

```
[vxWorks *]# ls
.metadata
2.txt
AD5628RTP
AD5628SPI
AM320
dev.c
DKM
dkm.c
dkm.o
```

Pour lancer le module charger celui-ci

```
[vxWorks *]# ld dkm.o
0xffff8000091da000
[vxWorks *]#
```

Revenir sur l'interpréteur de code

```
[vxWorks *]# C
-> moduleShow

        MODULE NAME             MODULE ID        GROUP #      SEGMENT ADDRESSES
-------------------------- -------------------- ------- -------------------------
DKM1.out                   0x00ffff8000092c8000       1 0xffffffff80d22000 (TEXT)
                                                       0xffffffff80d23000 (DATA)
                                                       0xffffffff80d23030 (BSS)
dkm.o                      0x00ffff8000091da000       2 0xffffffff80d26000 (TEXT)
                                                       *    NO SEGMENT    * (DATA)
                                                       *    NO SEGMENT    * (BSS)
value = 0 = 0x0
```

Lancer alors la tache associée à ce module

```
-> idev
value = -140737485908928 = 0xffff800000255440DKP on
 = buffer + 0xffff800000255410
```

Compilation et exécution d'un module noyau par l'intermédiaire d'un cross compilateur.

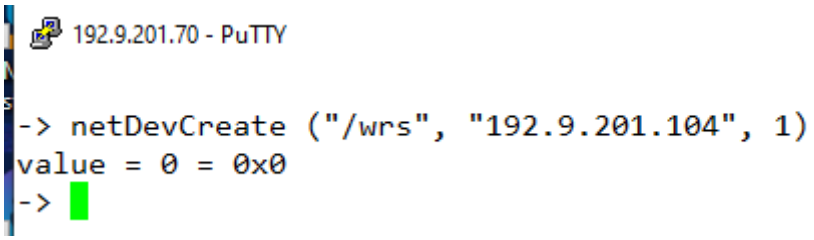
Lancer la machine virtuelle xubuntu et ouvrir une console sous LINUX

```
petitpa@ubuntu:~$ ls
alca.jpg    Music             supervis3.jpg                 Templates
Desktop     Pictures          supervis4.jpg                 Videos
Documents   Public            supervis_boucle_hirshman1.jpg vxworks
Downloads   snmp4j-2.8.3.jar  supervis_boucle_hirshman.jpg  vxworks7
foo.vxe     SNMP4JV3E.jar     supervis.jpg
linaro      supervis1.jpg     Sycobs5
petitpa@ubuntu:~$ cd vxworks7
petitpa@ubuntu:~/vxworks7$ cd wrsdk-vxworks7-raspberrypi4b/
petitpa@ubuntu:~/vxworks7/wrsdk-vxworks7-raspberrypi4b$ ls
a.out      foo.vxe    MLP115RTP.c    pilote.h       setcpu.c    tools
compilers  fork.c     MLP115RTP.vxe  README.md      setcpu.vxe  vxsdk
demo.h     gpioLib    modules        rtp.c          tdrv.c
docs       gpioLib.c  NCLA.txt       rtp.vxe        tdrv.vxe
examples   gpioLib.h  paams1.c       sdkenv.sh      thr.c
foo.c      Makefile   paams.c        sdkenv.sh.org  thr.vxe
petitpa@ubuntu:~/vxworks7/wrsdk-vxworks7-raspberrypi4b$ source sdkenv.sh
petitpa@ubuntu:~/vxworks7/wrsdk-vxworks7-raspberrypi4b$
```

Lancer la compilation de notre module noyau et copiez le dans un répertoire dédié au transfert FTP

```
petitpa@ubuntu:~/vxworks7/wrsdk-vxworks7-raspberrypi4b$ source sdkenv.sh
petitpa@ubuntu:~/vxworks7/wrsdk-vxworks7-raspberrypi4b$ $CC -dkm gpioLib/gpioLib.c modules/dkm.c -o modules/dkm.o
petitpa@ubuntu:~/vxworks7/wrsdk-vxworks7-raspberrypi4b$ sudo cp modules/dkm.o /home/target/ftp
petitpa@ubuntu:~/vxworks7/wrsdk-vxworks7-raspberrypi4b$
```

Créer un lien réseau vers le serveur FTP sur la PI4 en utilisant l'adresse IP de la VM xubuntu



Se positionner dans le répertoire wrs de la racine de la PI4 grâce à l'interpréteur de commande

```
-> cmd
[vxWorks *]# cd /wrs
[vxWorks *]#
```

Charger le module

```
[vxWorks *]# ld dkm.o
0xffff800009f82000
[vxWorks *]#
```

Lancer la fonction associée à la tâche du module noyau

```
[vxWorks *]# C
-> start

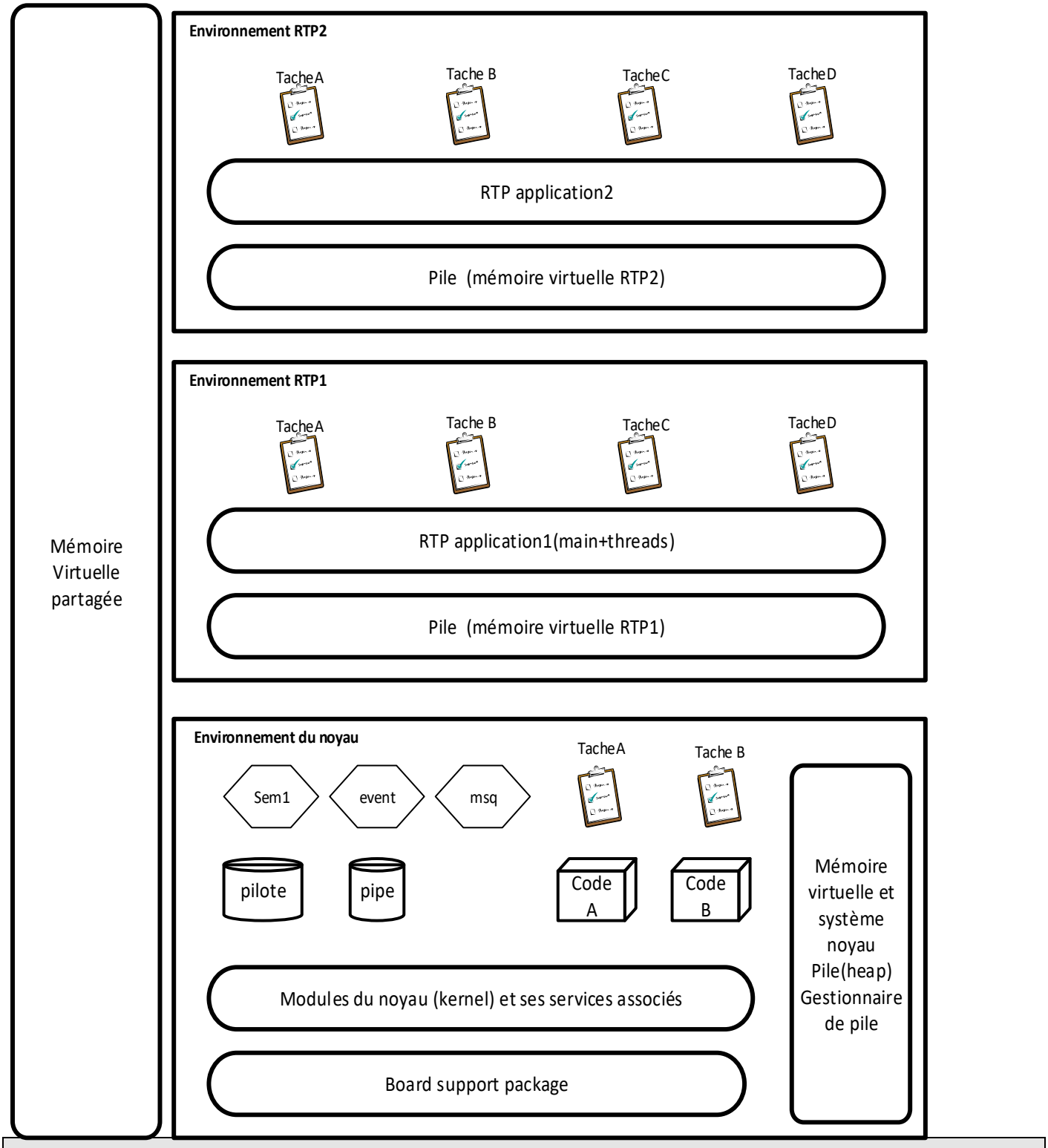

 On

 Off
```

Les processus temps réel RTP

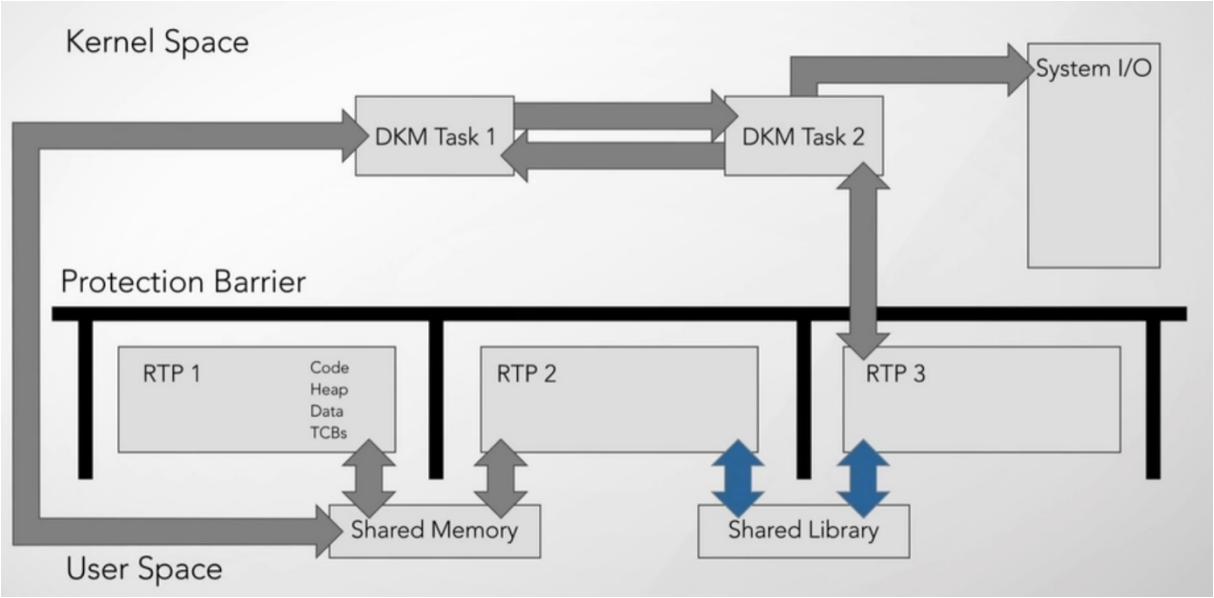
5.1) Présentation

Un RTP est juste un conteneur de taches. Les taches RTP sont additionnées dans une file de tache au même titre que les DKM
Un processus temps réel a son propre espace mémoire à l'intérieur de l'espace utilisateur
L'instance RTP est créé à partir d'un fichier .vxe
La protection et la séparation mémoire est renforcée par le MMU le MMU crée un espace mémoire virtuel pour chaque RTP



Chaque processus a son propre espace d'adressage et est protégé des autres processus

L'espace noyau est protégé des processus dans l'espace utilisateur



Le déroulement des processus temps réel est dépendant des événements externes au calculateur. Ces événements n'étant pas sous le contrôle du système, plusieurs taches peuvent se trouver en concurrence pour l'octroi du processeur. L'échange de données entre l'espace noyau et l'espace RTP se fait par une mémoire partagée

```
RTP_ID rtpSpawn(
rtpFilename, // Path to the vxE executable
argv,      //Argument list to be passed to main
envy,      //          Environment variable list
priority, // Task priority for the initial task
uStackSze,// Maximum stack size for the initial task options,  Options for the RTP
taskOptions // Options for the initial task
)
```

La valeur retournée RTP_ID est ok pour succes ou NULL pour erreur

Le vxworks shell peut être utilisé pour commencer un processus RTP :

/home/myvxapp.vxe first second third

ou rtp exec /home/myvxapp.vxe first second third

Pour l'interpréteur de code C

rtpSp « myvxapp.vxe first second third »

ou rtpSpawn("myvxapp.vxe", ----------)

L'utilisation de processus peut partitionner le système pour une plus grande fiabilité

• L'utilisation du modèle de mémoire protégée signifie que les zones où la vitesse est essentielle peuvent utiliser l'espace du noyau, tandis que d'autres codes peuvent utiliser l'espace de processus pour la sécurité

Le fonctionnement normal des tâches planifiées par VxWorks, et non des RTP

• Un RTP doit avoir au moins une tâche pour continuer à exister

• La tâche dans un RTP a une valeur de priorité, tout comme une tâche du noyau, et est placée dans la même file d'attente prête

• Une tâche dans l'espace utilisateur a une partie de son bloc de contrôle des tâches dans l'espace noyau que le planificateur peut utiliser, et cette partie n'est pas accessible par le RTP

• Les exceptions de l'espace utilisateur RTP sont gérées par le noyau

Avantages pour le projet RTP

• Un système utilisant des projets RTP est plus résistant aux défauts logiciels.
•La protection de la mémoire RTP ne se fait pas au détriment du déterminisme en temps réel.
• Il est plus facile de porter des logiciels LINUX vers des projets VxWorks RTP que vers des projets VxWorks Downloadable Kernel Module (DKM)

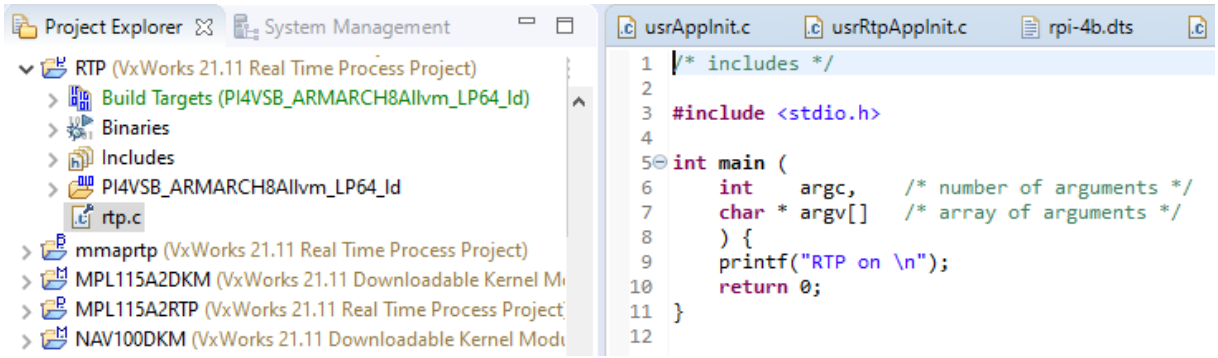
Inconvénients du projet RTP

• Les appels d'API du noyau de processus RTP en mode utilisateur ont une latence plus élevée.
• L'API du noyau RIP en mode utilisateur est un sous-ensemble de l'API de l'application du noyau.
• L'utilisation de projets RTP augmente l'empreinte mémoire du noyau VxWorks et du système
La tâche initiale dans en RTP commence à main() et doit rester en vie
• Tâches dans un RTP

- Partager du code et des données au sein du RTP
- Peut générer plus de tâches dans le RTP
- Impossible d'accéder au code ou aux données en dehors du RTP
- Utilise des appels système pour accéder aux fonctionnalités du noyau telles que les sémaphores
- Peut communiquer avec d'autres RTP et tâches du noyau à l'aide de données partagées d'objets publics

Un processus RTP n'a pas un accès direct au descripteur de fichier de l'espace noyau

Créer un nouveau projet du type RTP et compléter votre fichier source de la manière suivante :



Comment obtenir la liste des processus en mémoire sous vxworks

rtpShow() ;

```
-> rtpShow

       NAME                  ID              STATE       OPTIONS    TASK CNT
-------------------- ------------------ --------------- ---------- --------
< P/NonDebug/RTP.vxe 0xffff8000001ff970 STATE_NORMAL           0x1        1

RTP address space: overlapped
RTP code region:   8796090925056 bytes (0x200000-0x80000000000)

value = 1 = 0x1
-> 
```

Visualisez ainsi l'espace mémoire géré par vxworks

adrSpaceShow

```
-> adrSpaceShow

Global RAM Pool Info:
---------------------
  Allocation unit size:                      0x1000     4.0kB
  Total size:                             0x3b400000   948.0MB
  Allocated:                               0xcce2000   204.8MB
  Free:                                   0x2e71e000   743.1MB
  Largest contiguous free block:          0x2e688000   742.5MB


Shared User Virtual Memory Region Info:
---------------------------------------
  Allocation unit size:                      0x1000     4.0kB
  Total size:                         0x80000000000     8.0TB
  Allocated:                                     0     0.0
  Free:                               0x80000000000     8.0TB
  Largest contiguous free block:      0x80000000000     8.0TB


RTP Private Virtual Memory Region Info:
---------------------------------------
  RTP component included:                      Yes
  RTP Address Space:                       Overlapped
  RTP code region base:           0x0000000000200000
  RTP code region size:                0x7ffffe00000     7.9TB


Kernel System Virtual Memory Info:
----------------------------------
  System Region Base:             0xffffffff80000000
  System Region Size:                      0x80000000     2.0GB


Kernel Reserved Memory Region Info:
-----------------------------------
  Reserved Region Base:           0xfffff00000000000
  Reserved Region Size:                0x80000000000     8.0TB


Kernel Virtual Memory Pool Region Info:
---------------------------------------
  Allocation unit size:                      0x1000     4.0kB
  Total size:                         0x80000000000     8.0TB
  Allocated:                               0xa546000   165.2MB
  Free:                               0x7fff5aba000     7.9TB
  Largest contiguous free block:      0x7fff5aba000     7.9TB
value = 0 = 0x0
```

Visualiser la liste des descripteurs de périphériques gérée par le noyau vxworks

```
-> devs
drv refs name
  1 [ 3] /
  4 [ 3] /I2C/0
  6 [ 3] /PWM/0
  5 [ 3] /SPI/0
  0 [ 3] /dev/null
  9 [ 3] /devm
 19 [ 3] /fifos
 23 [ 3] /hostfs
  0 [ 3] /null
 11 [ 3] /ram
 11 [ 3] /ram0
 18 [ 3] /romfs
  8 [ 3] /shm
 17 [ 5] /stdio_pty_0xffff800000263b60.M
 16 [ 5] /stdio_pty_0xffff800000263b60.S
 11 [ 3] /tmp
 20 [ 3] /tty
  3 [ 5] /ttyS0
  3 [ 3] /ttyS1
  2 [ 3] /tyCo/0  ==>  /ttyS0
  2 [ 3] /tyCo/1  ==>  /ttyS1
  7 [ 3] /vxbus
 22 [ 3] host:
value = -140737485864688 = 0xffff800000260110 = buffer + 0xffff8000002600e0
```

✍ le port console est géré par un périphérique du type série RS232 /ttyS0 quel est son numéro de pilote ?

Visualiser la liste des descripteurs de fichiers gérée par le noyau vxworks

```
-> iosFdShow
 fd name                                  drv
  3 /ttyS0                                  3 in out err
  4 (anonpipe)                             10
  5 (socket)                               21
  6 (socket)                               21
  7 (socket)                               21
  8 (socket)                               21
  9 (socket)                               21
 10 (socket)                               21
 11 (socket)                               21
 12 (socket)                               21
 13 (socket)                               21
 14 (socket)                               21
 15 (socket)                               21
 16 /stdio_pty_0xffff800000263b60.M        17
 17 (socket)                               21
 18 (socket)                               21
 19 /stdio_pty_0xffff800000263b60.S        16
value = 50 = 0x32 = '2' = buffer + 0x2
->
```

Les différentes taches DKM ou RTP (main + thread) sont ordonnancées dans un contexte multitâche SMP (symetric multiple processor) avec 4 cœurs du type CORTEX A72. Ces taches sont distribuées aux quatre cœurs grâce à 2 techniques d'ordonnancement que l'on va présenter par la suite.

L'ordonnancement sous VxWorks

6.1) Fonctionnement de l'ordonnanceur.

L'ordonnanceur *(scheduler)* constitue le cœur du noyau temps réel . Il est invoqué à chaque commutation de taches ainsi qu'à chaque appel à une fonction système (sémaphore, signal, file d'attente, ...)

Il y a ordonnancement si :

- La tâche en exécution passe dans l'état attente, retardée ou suspendue.
- Une tâche de plus haute priorité passe dans l'état prêt (préemption).
- Ecoulement du time-slice si au moins une autre tâche de même priorité est prête.

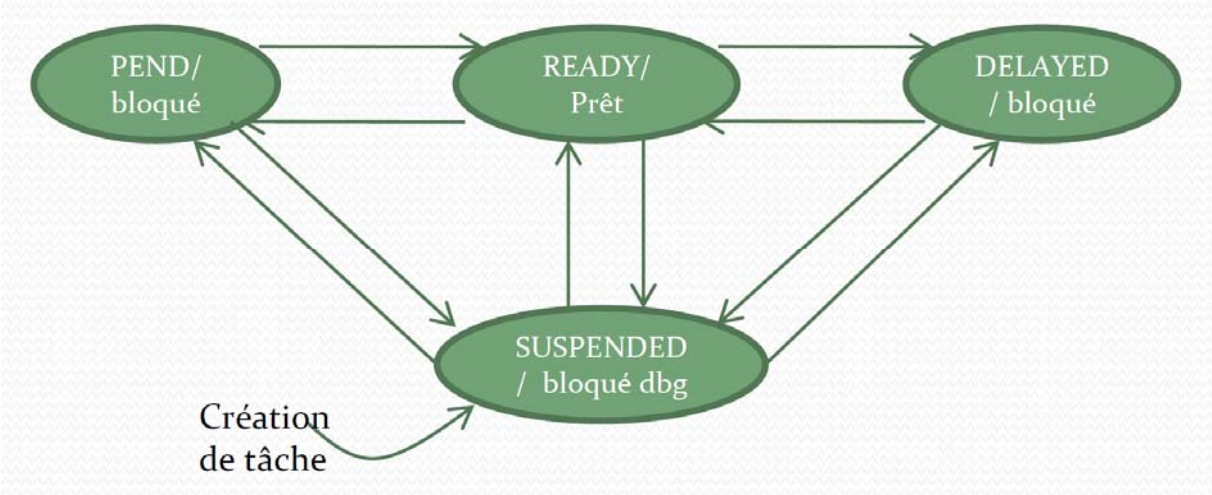
Les priorités.

La priorité d'une tâche est codée sur un octet :

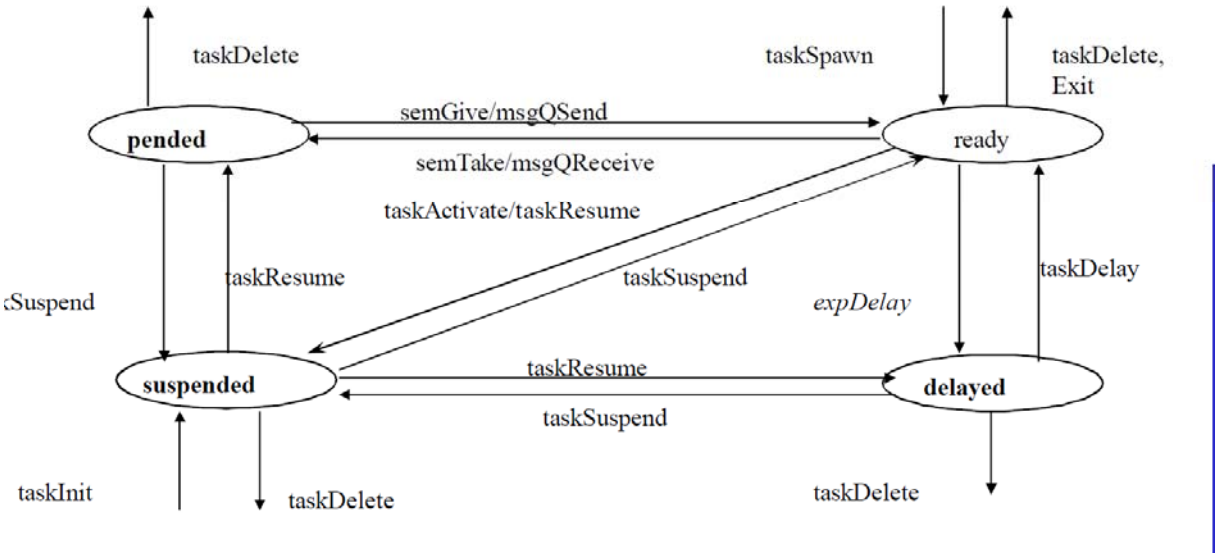
- 0 = priorité maximale
- 255 = priorité minimale
- 100 = priorité par défaut

VxWorks n'implémente pas la gestion dynamique des priorités. Si une tâche A n'est jamais bloquée ou en attente ou retardée, alors une tâche B moins prioritaire ne sera jamais exécutée.
Si plusieurs tâches sont de même priorité, il est alors possible de faire fonctionner l'ordonnanceur en mode "round robin" entre celles-ci. (ordonnancement par priorités avec files multiples).

Prête	$READY$	État d'une tâche qui n'attend aucune autre ressource que le CPU.
En attente	$PEND$	État d'une tâche bloquée suite à l'indisponibilité d'une ressource (matérielle, logicielle, information, ...)
Endormie	$DELAY$	État d'une tâche endormie pour une durée déterminée.
Suspendue	$SUSPEND$	État d'une tâche inéligible. Elle n'est jamais exécutée. Cet état est principalement utilisé pour la mise au point (*debugging*). La suspension d'un tâche n'empêche pas le changement d'état logique, seulement l'exécution. Ainsi, une tâche **en attente** et **suspendue** peut être débloquée. De même, une tâche **endormie** et **suspendue** peut être réveillée. Dans les deux cas elle reste **suspendue**.
Endormie-suspendue	$DELAY + S$	État d'une tâche à la fois **endormie** et **suspendue**.
En attente-suspendue	$PEND + S$	État d'une tâche à la fois **en attente** et **suspendue**.
En attente-avec «time-out»	$PEND + T$	État d'une tâche **en attente**, mais avec un *time out*
En attente-suspendue-avec «time-out»	$PEND + S + T$	État d'une tâche **en attente** avec un *time out*, et en plus **suspendue**. À l'issue du *time out* elle reste **suspendue**.



- états
  - prêt / **bloqué**



6.2) Les techniques d'ordonnancement des taches

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

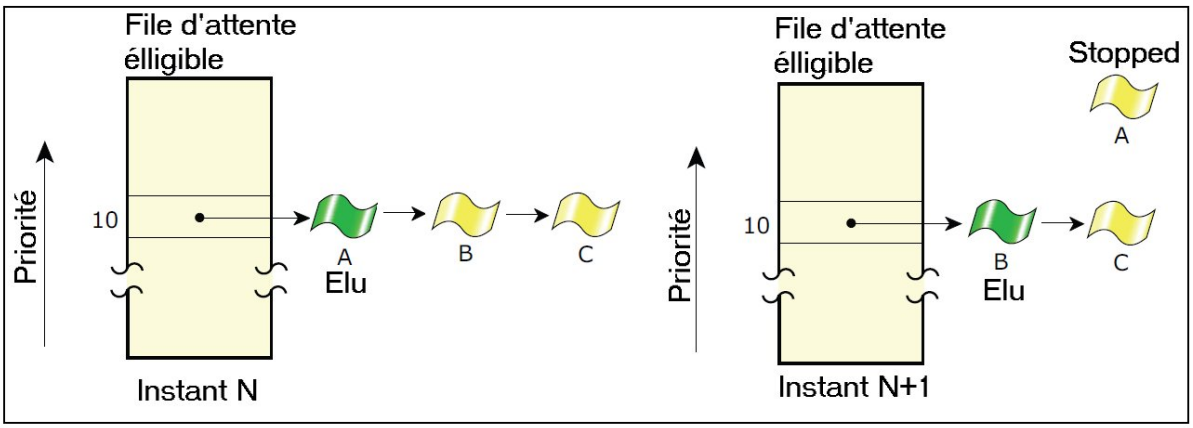
- ✓ **Ordonnancement FIFO**
- ✓ **Ordonnancement Round Robin**

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

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

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

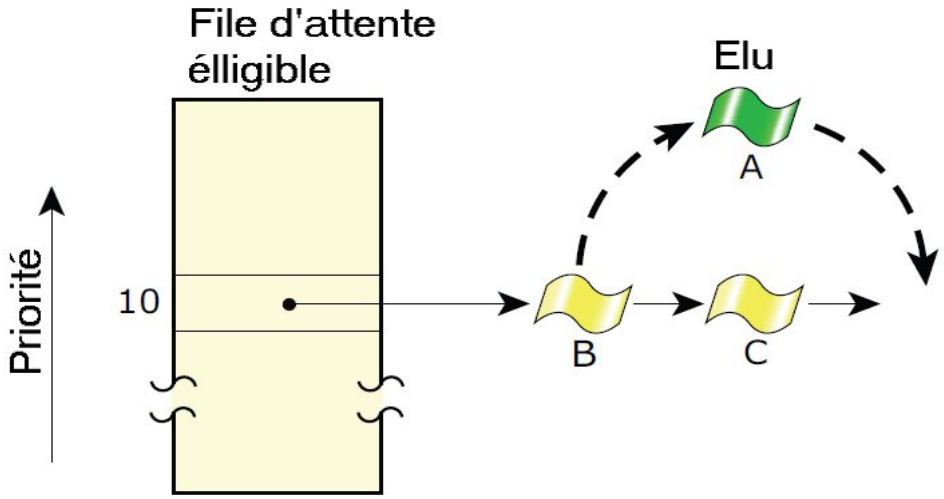
- ✓ **libère** le processeur (il se termine ou il fait un appel système, par exemple)
- ✓ soit préempté par un processus de priorité supérieure



b) L'Ordonnancement Round Robin

Dans le principe du round-robin, une tache s'exécute jusqu'à ce qu'elle :

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



c) Le changement de contexte

•Une interruption (provenant d'un périphérique externe ou d'une minuterie, par exemple) entraînera un changement de contexte

• Le changement de contexte est synchrone s'il est contrôlé par la tâche, par exemple, une attente ou un retard, ou une tâche de priorité plus élevée étant prête à s'exécuter

• Le changement de contexte est asynchrone s'il est causé par une interruption, telle qu'un timer

Pour activer le découpage temporel.
Où les ticks sont le nombre de temps système par tranche (slice)

```
STATUS kernelTimeSlice
    (
    int ticks   /* time-slice in ticks or 0 to disable round-robin */
    )
```

Cette routine permet une sélection circulaire parmi les tâches de même priorité et définit la tranche de temps système (ticks) attribuée . le round robin est désactivée par défaut. Une tranche de temps de zéro ticks désactive le round robin.

• Pour désactiver le découpage temporel :

```
kernelTimeSlice(0)
```

• Pour trouver la fréquence d'horloge du système :

```
_Vx_freq_t sysClkRateGet(void)
```

Retourne Le nombre de ticks par seconde de l'horloge système.

• Pour trouver la période d'une tranche de temps en ticks :

```
STATUS kernelTimeSliceGet
    (
    ULONG       * pTimeSlice   /* current time-slice in ticks */
    )
```

Cette routine renvoie la tranche de temps en ticks pour le système.

• En multitâche coopératif pour abandonner le CPU temporairement, le programmeur appelle

```
STATUS taskDelay
    (
    _Vx_ticks_t ticks   /* number of ticks to delay task */
    )
```

Cette routine amène la tâche appelante à abandonner le CPU pendant la durée spécifiée en ticks.

L'état de la tâche est changé en retardé (non prêt) jusqu'à ce que le nombre spécifié d'interruptions de tick se soit produit. Par exemple, un taskDelay( ) avec une valeur de tick de 1 attendra jusqu'à la prochaine interruption de l'horloge système ; dans ce cas, le temps pendant lequel la tâche est retardée est inférieur à une période de tick. Une fois le délai expiré, l'état de la tâche revient à prêt (sauf si elle a été suspendue ou arrêtée pendant le retard) et devient éligible pour être planifiée pour exécution.

L'appel de taskDelay() avec une valeur de tick de 0 force l'ordonnanceur à mélanger la file d'attente prête, permettant aux tâches de la même priorité que la tâche appelante d'avoir une chance de s'exécuter. S'il n'y a pas de tâche de même priorité que la tâche appelante dans la file d'attente prête, la tâche appelante reprendra immédiatement son exécution.

La liste des taches :

```
-> i

  NAME             TID       PRI   STATUS         PC           ERRNO  CPU #
---------- ---------------- --- ---------- ---------------- ------- -----
tIsr0       ffff8000000099a0   0 PEND       ffffffff804138c0       0     -
tJobTask    ffff8000001e1440   0 PEND       ffffffff8036d3bc       0     -
tExcTask    ffffffff809c3488   0 PEND       ffffffff804138c0       0     -
tLogTask    ffff8000001f2950   0 PEND       ffffffff80411e50       0     -
tShell0     ffff80000020f530   1 PEND       ffffffff804138c0       0     -
tShellRem1> ffff80000025e210   1 READY      ffffffff8041ef08       0     2
tErfTask    ffff800000007c40  10 PEND       ffffffff804147a8       0     -
tVxdbgTask  ffff8000002566c0  25 PEND       ffffffff804138c0       0     -
tTcfEvents  ffffffff809ce0b0  49 PEND+T     ffffffff80414f78  3d0004     -
tTcf        ffff8000002581c0  49 DELAY      ffffffff8041d4dc       0     -
tTcf        ffff800000251aa0  49 PEND       ffffffff804138c0       0     -
tTcf        ffff800000252a10  49 PEND       ffffffff804138c0       0     -
tAioIoTask> ffff8000001f6d40  50 PEND       ffffffff804147a8       0     -
tAioIoTask> ffff8000001f7280  50 PEND       ffffffff804147a8       0     -
tNetConf    ffff8000001fc1a0  50 PEND       ffffffff804138c0       0     -
ipcom_teln> ffff8000001dfe00  50 PEND       ffffffff804138c0       0     -
iptftps     ffff800000253a80  50 PEND       ffffffff804138c0       2     -
ipcom_teln> ffff8000002137e0  50 PEND+T     ffffffff804138c0       0     -
tStdioProx> ffff800000213d00  50 READY      ffffffff804138c0       0     -
tLogin2137> ffff80000025dcf0  50 PEND       ffffffff804147a8       0     -
tAioWait    ffff8000001f6640  51 PEND       ffffffff804138c0       0     -
tNet0       ffff800000008500 100 READY      ffffffff80413530       0     0
tTffsPTask  ffff80000021c6c0 100 DELAY      ffffffff8041d4dc       0     -
miiBusMoni> ffff80000020e9e0 252 DELAY      ffffffff8041d4dc       0     -
tIdleTask0  ffffffff80cf5000 287 READY      ffffffff80413190       0     -
tIdleTask1  ffffffff80d02000 287 READY      ffffffff80413190       0     1
tIdleTask2  ffffffff80d0f000 287 READY      ffffffff80413190       0     -
tIdleTask3  ffffffff80d1c000 287 READY      ffffffff80413190       0     3
```

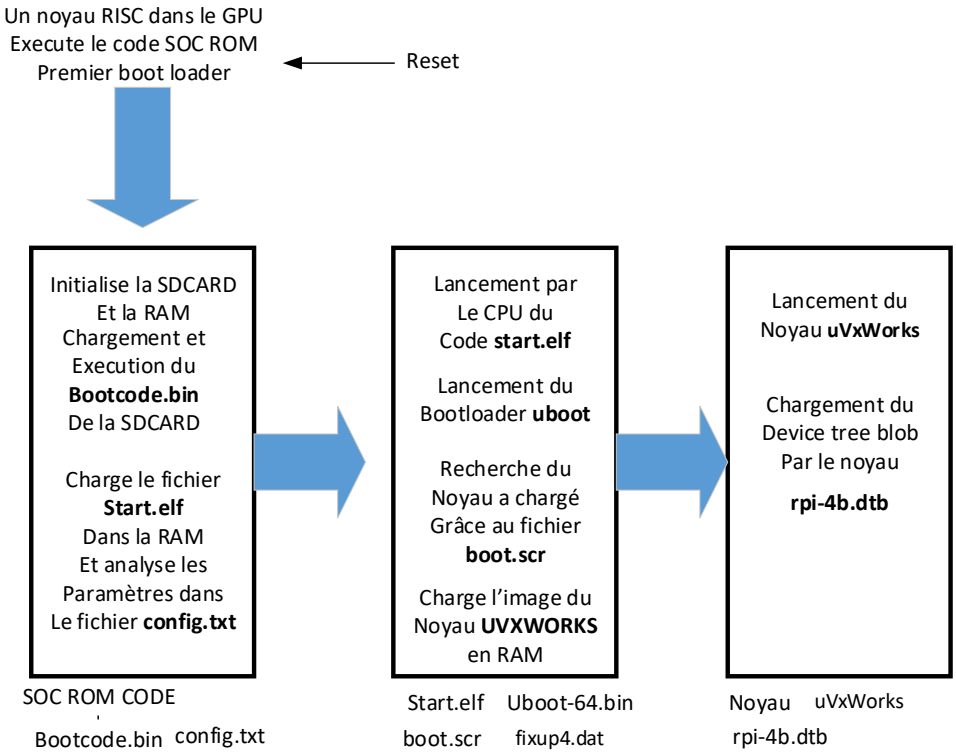
Tisr0 : tâche qui exécute pour une routine de service d'interruption ISR
tjobtask : tâche qui exécute des fcts pour d'autres taches
tlogtask : tâche qui exécute des fcts pour les messages logs
tshell0 : tâche qui exécute le shell affiche données sur l'écran
terftask : les nouveaux drivers de stockage utilisent cette tache pour s'interfacer entre le système et vxworks
tvxdbgtask : utilisé par des routines de debug (breakpoint)
tache : **tcf tcfevent** pour l 'agent
Tnetzero : tâche qui permet d'accéder directement à la pile réseau et et au drivers réseau
Tai0 : permet de gérer la queue pour les demandes de lecture et d'écriture E/S
Tnetconf : écoute les interfaces qui essaye de se connecter à la pile réseau et de les configurer
la tache **troot** : permet d'effectuer les premiers tests quand il boote après troot le système est en SMP
Tuer la tache tNet0

```
NAME             TID       PRI   STATUS         PC           ERRNO  CPU #
-------- ---------------- --- ---------- ---------------- ------- -----
tNet0       ffff800000008500 100 READY      ffffffff80413530       0     0
tTffsPTask  ffff80000021c6c0 100 DELAY      ffffffff8041d4dc       0     -
miiBusMoni> ffff80000020e9e0 252 DELAY      ffffffff8041d4dc       0     -
tIdleTask0  ffffffff80cf5000 287 READY      ffffffff80413190       0     -
tIdleTask1  ffffffff80d02000 287 READY      ffffffff80413190       0     1
tIdleTask2  ffffffff80d0f000 287 READY      ffffffff80413190       0     -
tIdleTask3  ffffffff80d1c000 287 READY      ffffffff80413190       0     3
value = 0 = 0x0
-> td tNet0
```

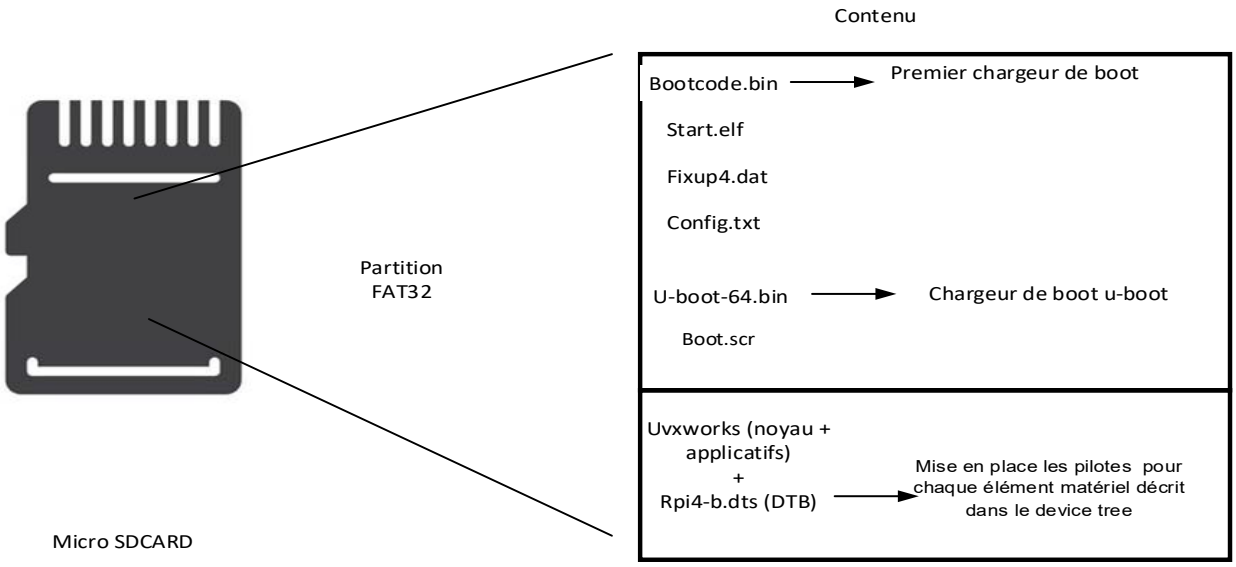
✍ Que remarquez vous ?

Séquence de boot vxworks7 sur la plateforme Raspberry PI4

Avant que le noyau de Vxworks se lance plusieurs bootloader propre à la Raspberry PI4 et au processeur ARM sont nécessaires :



Nom	Rôle	Format
Bootcode.bin	bootloader	Binaire
Start.elf	Code ARM exécuté par le CPU	Binaire
Fixup4.dat	Configure la SDRAM GPU/CPU	Binaire
Config.txt	Paramètre de configuration pour u-boot	texte
rpi-4b.dtb	Device tree blob	texte
uVxWorks	Noyau	Binaire
Boot.scr	Ligne de commande pour charger et lancer le noyau	texte
u-boot-64.bin	bootloader	binaire



Bien que le processus global de démarrage de VxWorks 7 soit similaire aux anciennes versions, les détails ont considérablement changé. Les principales différences sont :

VxWorks lui-même n'inclut plus de code ROMable pour s'exécuter directement à partir de la réinitialisation et amorcer le système.

VxWorks 7 utilise une arborescence de périphériques de style Linux pour définir le matériel ;

l'arborescence des périphériques doit généralement être chargée en mémoire en tant que "blob" binaire **DTB (device tree blob)** avant le démarrage du noyau VxWorks.

Au lieu de cela, le matériel doit avoir un chargeur de démarrage indépendant installé pour s'exécuter à partir de la réinitialisation et démarrer le système.

Si vous utilisez une conception matérielle commerciale prête à l'emploi, le chargeur de démarrage sera probablement U-Boot. Par exemple, il peut démarrer d'autres systèmes d'exploitation tels que Linux.

Lorsque le Raspberry Pi 4 modèle B s'allume, le processeur ARM est hors ligne et le GPU démarrera le SOC avec le firmware spécifique au GPU (**Bootcode.bin** dans la partition de démarrage de la carte SD) .

Les fichiers **start4.elf** et **fixup4.dat**

start4.elf est le Firmware du GPU Pi4. Fixup_file (fixup4.dat) Spécifie l'emplacement de mémoire dans le start_file(start4.elf) pour qu'il corresponde au fichier de fractionnement de mémoire GPU. Notez que start_file et fixup_file sont des paires de fichiers correspondants et intimement liées.
start4.elf est responsable aussi du chargement et de la modification des fichiers dtB (taille de la mémoire, adresse Ethernet...).), qui réalise des fonctions de personnalisation personnalisées simples.

Ce chargeur de démarrage démarrera à son tour l'exécutable ARM image. U-boot est utilisé comme image exécutable ARM initiale pour démarrer le code ARM.

Le DTB stocke la ligne de démarrage transmise au noyau VxWorks. Plus précisément, la ligne de démarrage est écrite dans le nœud choisi de l'appareil arbre. Le noyau VxWorks extrait ces informations de l'arborescence des périphériques et les analyse comme nécessaire.

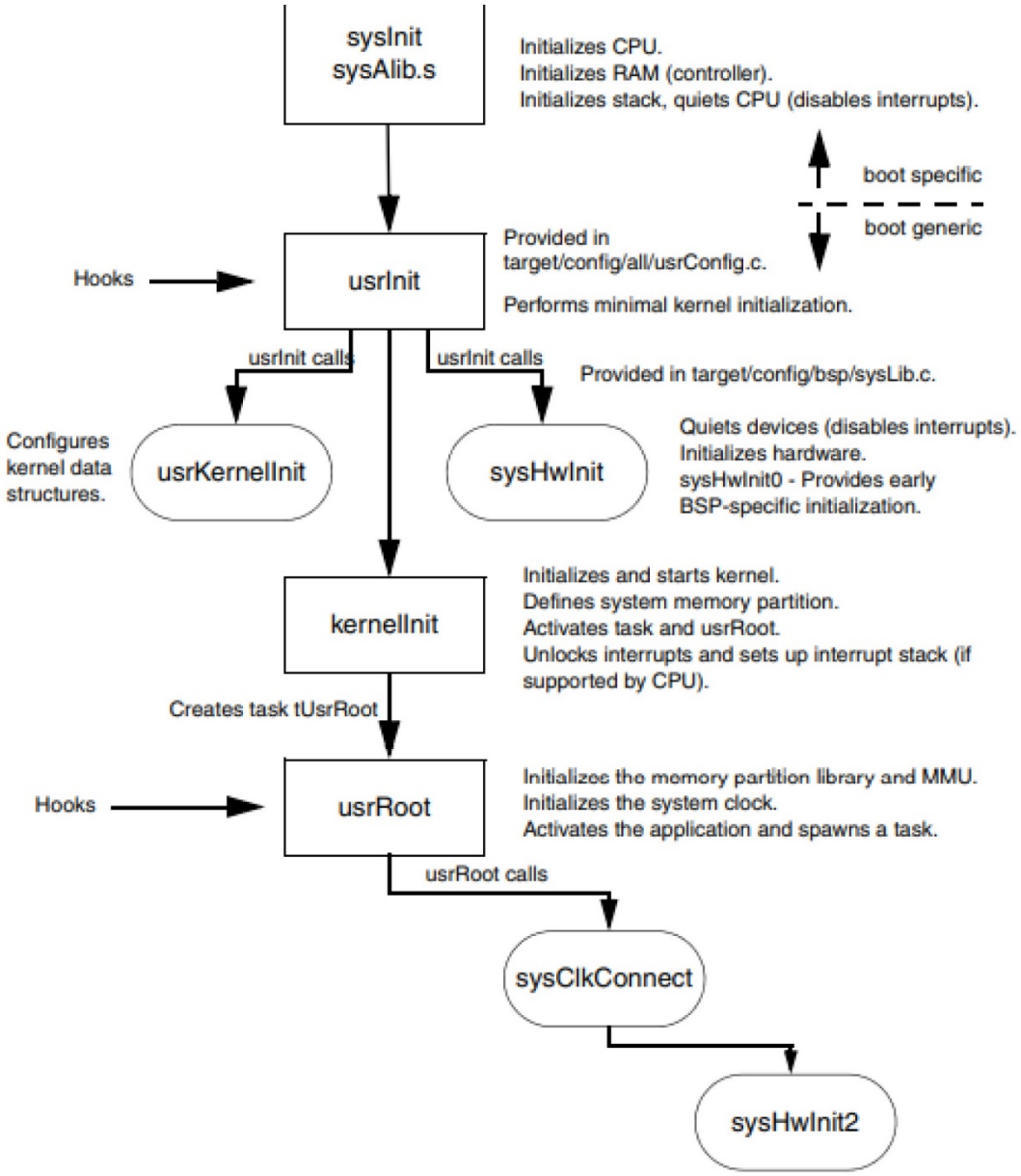
```
bootargs = "usb2End(0,0)host:vxWorks h=192.9.201.7 e=192.9.201.70:ffffff00 g=192.9.201.8 u=target
pw=vxTarget o=genet0 f=0x40";
        stdout-path = "serial1";
```

U-Boot peut démarrer deux types d'images VxWorks : uVxWorks et le raw fichier bin vxWorks.bin. Le uVxWorks est utilisé avec le U-Boot "bootm" commande et nécessite un fichier DTB indépendant . le vxWorks.bin est utilisé avec la commande U-Boot "go" et nécessite un DTB intégré au noyau.

Le fichier **boot.scr** va permettre à uboot de charger et d'exécuter le noyau
Il contient les commandes suivantes pour charger et exécuter un Noyau VxWorks sur une carte ARM à partir d'une carte SD.

fatload mmc 0:1 0x10000000 uVxWorks
bootm 0x10000000

Le point d'entrée VxWorks est _sysInit( ) dans sysALib.s. Cette routine d'assemblage définit l'état initial du matériel puis saute à usrInit( ) dans usrConfig.c.
usrInit( ) est la première routine C exécutée à partir d'une image VxWorks.
Cette routine initialise le cache, initialise la table vectorielle, effectue une initialisation spécifique à la carte, puis démarre le noyau multitâche avec un utilisateur- tâche de démarrage.

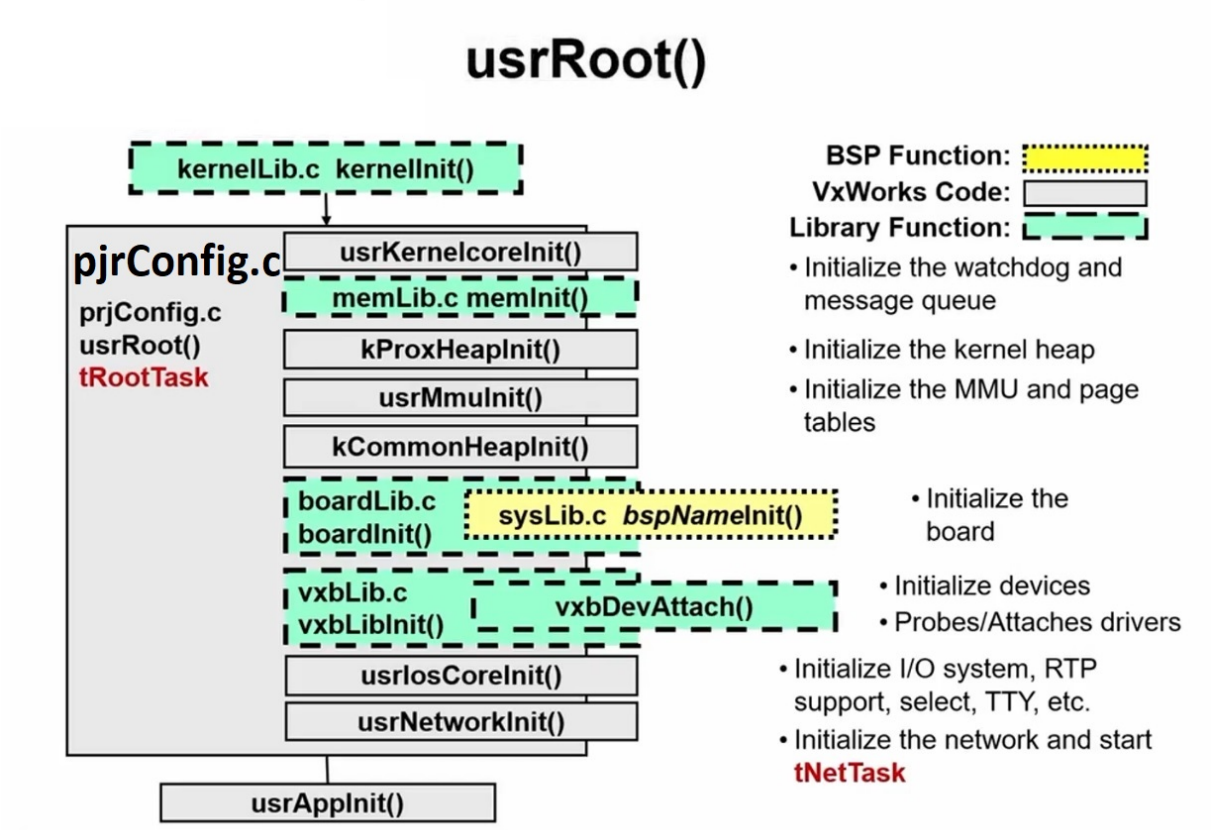


Le but de la routine usrInit( ) est d'initialiser complètement le CPU et d'arrêter tout autre matériel, préparant ainsi le chemin pour que le noyau s'initialise et démarre lui-même. Cette routine se trouve dans prjConfig.c mais appelle des routines dans plusieurs autres fichiers, dont certains doivent être fournis par vous et d'autres sont fournis par Wind River. Normalement,
il n'est pas nécessaire de modifier la routine usrInit( ) fournie par Wind River. Cependant, la routine sysHwInit( ) qui doit être appelée depuis usrInit( ) nécessite généralement une modification. La routine sysHwInit( ) garantit que le matériel dépendant de la carte est au repos. sysHwInit( ) est fourni dans le BSP ou le modèle de référence et se trouve dans sysLib.c.
La routine usrInit( ) (dans prjConfig.c) enregistre les informations sur le type d'amorçage, gère toutes les initialisations qui doivent être effectuées avant le démarrage effectif du noyau, puis démarre l'exécution du noyau. C'est le premier code C à s'exécuter dans VxWorks.
Cette routine est appelée en mode superviseur avec toutes les interruptions matérielles verrouillées.
De nombreuses fonctionnalités ne peuvent pas être appelées à partir de cette routine. Il n'y a pas encore de contexte de tâche - c'est-à-dire, pas de bloc de contrôle de tâche (TCB) et pas de pile de threads - par conséquent, les fonctionnalités qui nécessitent un contexte de tâche ne peuvent pas être appelées. Cela inclut toute fonctionnalité qui peut entraîner la préemption de l'appelant (telle que les sémaphores) ou toute fonctionnalité qui utilise elle-même une fonctionnalité de ce type, telle que printf(). Au lieu de cela, la routine usrInit( )
ne fait que ce qui est nécessaire pour créer le thread initial, usrRoot( ). usrRoot( ) termine alors le démarrage. Les opérations d'initialisation effectuées dans usrInit( ) incluent les éléments suivants :

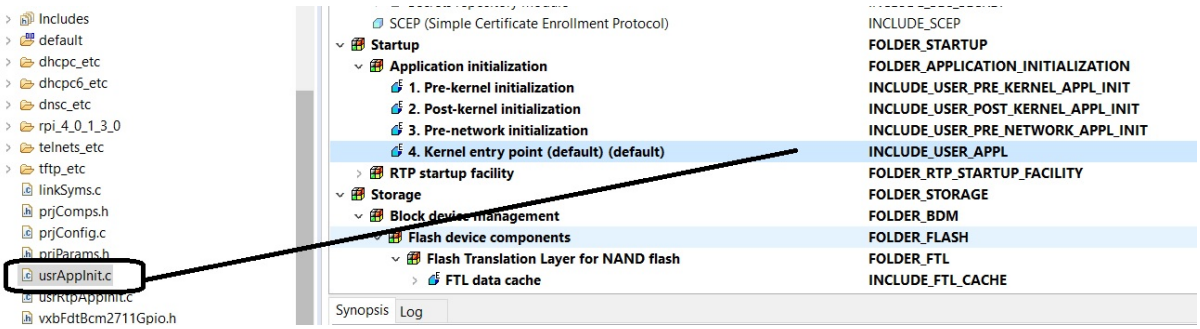
■Initialisation du cache.
Le code au début de usrInit() initialise les caches, définit le mode des caches et place les caches dans un état sûr. A la fin de usrInit(),les caches d'instructions et de données sont activés par défaut.
■ Remise à zéro du segment BSS du système.
Le langage C spécifie que toutes les variables non initialisées (stockées dans bss) doivent avoir des valeurs initiales de 0.
Comme usrInit( ) est le premier code C à s'exécuter, il efface la section de mémoire contenant bss.
■ Initialisation des vecteurs d'interruption.
Les vecteurs d'exception doivent être initialisés avant d'activer les interruptions et de démarrer le noyau.
Tout d'abord, intVecBaseSet() est appelé pour établir l'adresse de base de la table vectorielle.
Après l'appel de intVecBaseSet( ), la routine excVecInit( ) initialise tous les vecteurs d'exception
aux gestionnaires par défaut. Ces gestionnaires interceptent et signalent en toute sécurité les exceptions causées par des erreurs de programme ou des interruptions matérielles inattendues.
■ Initialisation du matériel système à un état de repos.
L'appel de la routine système sysHwInit( ) initialise le matériel du système. L'initialisation consiste principalement à réinitialiser et à désactiver les périphériques matériels. Sinon, lorsque le noyau est démarré et que les interruptions sont activées, ces périphériques peuvent provoquer des interruptions inattendues. Dans VxWorks, les ISR (pour les périphériques d'E/S, les horloges système, etc.) ne sont pas connectés à leurs vecteurs d'interruption tant que l'initialisation du système n'est pas terminée par la tâche usrRoot( ). Il s'agit d'une exigence car le pool de mémoire n'est pas encore initialisé. Vous ne devez pas connecter un gestionnaire d'interruption à une interruption pendant l'appel sysHwInit( ) ; cela nécessite une allocation de mémoire, qui n'est pas disponible pour le moment. La plupart des connexions d'interruption se produisent plus tard dans la routine sysHwInit2( ) située dans sysLib.c.
■ Appel de kernelInit( ).
Les bibliothèques VxWorks contiennent le code pour kernelInit(). Par conséquent, il n'est normalement pas disponible pour les développeurs BSP sous forme source. La routine kernelInit( ) lance l'environnement multitâche et ne revient jamais.

La routine kernelInit( ) est fournie par Wind River dans un fichier d'archive de la bibliothèque VxWorks. Le but de cette routine est de rendre le noyau opérationnel afin que toute initialisation ultérieure puisse être effectuée en tant que tâche s'exécutant sous le noyau.
Le nom de cette tâche est tRootTask, et la routine qu'elle exécute est généralement usrRoot().
La routine kernelInit( ) appelle intLockLevelSet( ), désactive le mode de planification round-robin et crée une pile d'interruptions(si elle est prise en charge par l'architecture). La routine crée ensuite une pile racine et un bloc de contrôle de tâche (TCB)
à partir du haut du pool de mémoire, génère le thread racine usrRoot() et termine le thread d'exécution usrInit(). A ce moment, les interruptions sont activées. Il est essentiel que toutes les sources d'interruption soient désactivées par usrInit() et que toutes les interruptions en attente soient effacées.

Le but de la routine usrRoot( ) est de terminer l'initialisation du noyau et de tout le matériel, puis de lancer n'importe quel code d'application.
Cette routine est fournie par Wind River dans le fichier prjConfig.c et la copie d'origine ne doit pas être modifiée.



Au cours du développement, prjConfig.c est souvent temporairement copié dans le répertoire BSP et des modifications de débogage sont apportées à la copie temporaire. prjConfig.c est également configurable avec les macros définies dans config.h. Elle initialise les services noyaux événements, sémaphores, messages …
La routine usrRoot( ) appelle la routine memInit(pour initialiser la mémoire. Facultativement, usrRoot( ) peut appeler les routines memShowInit( )et usrMmuInit( ). Une fois que le système est multitâche, le BSP appelle sa première routine, sysClkConnect(). sysClkConnect( ) appelle immédiatement la routine sysHwInit2( ). sysHwInit2( ) est responsable de toute initialisation de carte non terminée dans sysHwInit( ), telle que la connexion de sources d'interruption à l'aide de intConnect( ). Ensuite, la routine usrRoot( ) continue l'initialisation de l'horloge.
Il définit la fréquence d'horloge par défaut sur la valeur de la macro SYS_CLK_RATE, généralement 60 Hz. usrRoot( ) active ensuite l'horloge système avec un appel à sysClkEnable( ). Une fois l'horloge initialisée et en cours d'exécution, plusieurs modules du noyau, tels que selectLib, le sous-système d'E/S et la console, sont initialisés. Elle appelle les fonctions d'initialisation du board support package, le vxbus pour sonder (probe) les périphériques et attacher les pilotes (attach) . Elle termine par l'appel de la fonction usrAppInit pour initialiser les applications.

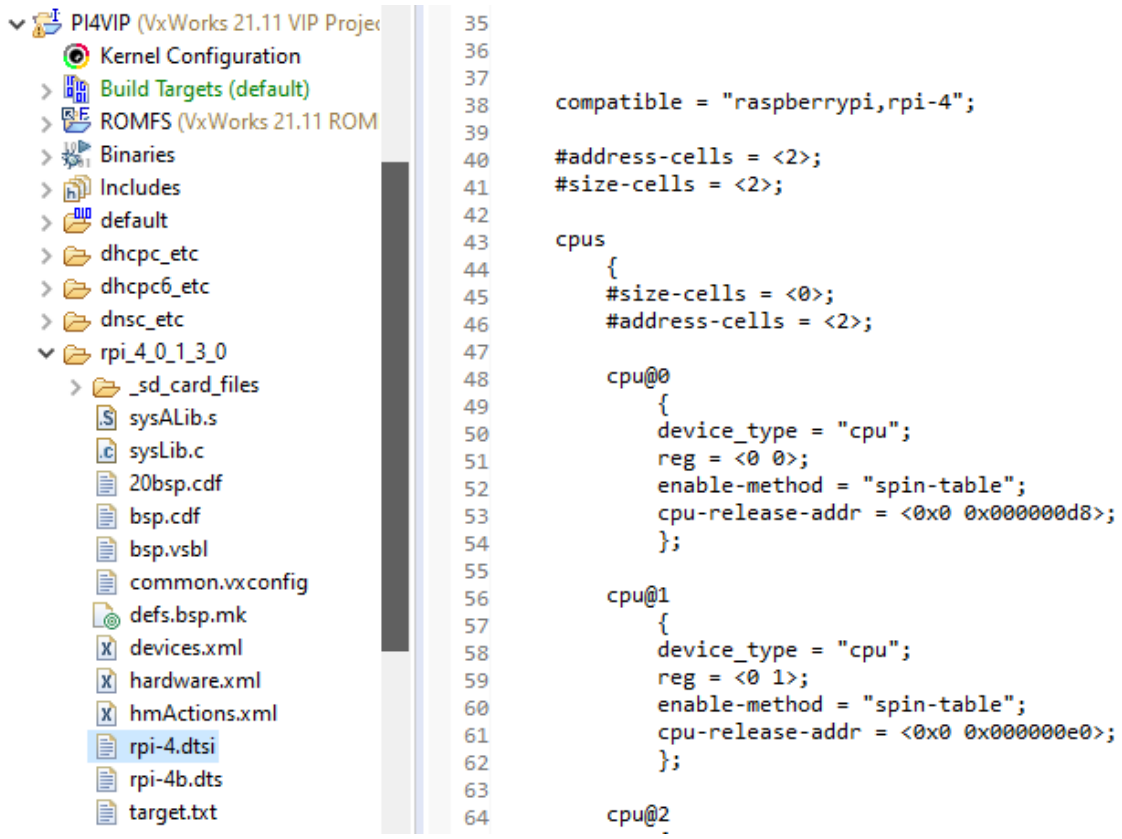


8) Le device tree

Depuis plusieurs années, le support de l'architecture ARM dans le noyau est passé progressivement au « Device Tree » pour la description du matériel.

Avant l'introduction du Device Tree, une quantité très importante de code C dans ces répertoires du noyau arch/arm/ servait à décrire l'ensemble des périphériques de chaque SoC, puis l'ensemble des périphériques de chaque carte électronique embarquant un SoC. l'architecture ARM ne fournissant pas de mécanisme d'énumération dynamique (USB PCI), il est nécessaire pour le kernel de connaître à l'avance l'ensemble des périphériques du système et leurs caractéristiques. Le noyau doit décrire, sous la forme de code C, l'ensemble des périphériques de chaque SoC et de chaque carte électronique ce qui devient très lourd.
Le passage au Device Tree s'est donc inscrit dans ce grand mouvement de nettoyage et de refactoring du support de l'architecture ARM dans le noyau .
Le principe fondamental du Device Tree est de décrire le matériel d'un système et de passer cette description du matériel au système d'exploitation lors du démarrage d'une plateforme.

Avec le Device Tree, chaque plateforme matérielle se retrouve décrite par un fichier appelé Device Tree Source (DTS ou DTSI).



Lors de la compilation du noyau , ces fichiers DTS sont transformés dans un format binaire appelé Device Tree Blob (DTB) grâce à un compilateur spécifique appelé Device Tree Compiler (DTC). Cette transformation sert principalement à fournir un format plus efficace à parser qu'un format texte.

Démarrage avec U-Boot

Pour démarrer un noyau utilisant le Device Tree pour décrire une plateforme matérielle, il faudra donc au niveau du chargeur de démarrage charger deux fichiers :
l'image du noyau uVxworks et le Device Tree Blob rpi-4.dtb .
Notons que le chargeur de démarrage est autorisé à adapter le DTB avant de le passer au kernel, par exemple pour lui passer des informations comme la ligne de commandes du noyau ou l'adresse MAC des différentes interfaces réseau.

Comme son nom l'indique, un Device Tree est un arbre, composé d'un ensemble de noeuds (node) imbriqués les uns avec les autres. Chacun de ces noeuds peut contenir un ensemble de propriétés (properties) et de sous-noeuds. Voici un extrait d'un Device Tree, provenant du fichier rpi-4.dtsi

```
uart0: serial@fe201000
            {
            compatible = "brcm,bcm2711-pl011", "arm,pl011", "arm,primecell";
            reg = <0x0 0xfe201000 0x0 0x0200>;
            interrupts = <153 0 4>;
            interrupt-parent = <&intc>;
            clock-frequency = <48000000>;
          pinmux-0 = <&uart0_pins>;
            fifo-disabled = <1>;
            status = "disabled";
            };
```

Au démarrage, le noyau reçoit dans le registre l'adresse du DTB et va donc pouvoir le parcourir pour découvrir le matériel de la plateforme sur laquelle il s'exécute. En particulier, pour chaque noeud dans le Device Tree qui est un fils d'un noeud identifié comme un simple-bus, le noyau va instancier une structure platform_device représentant ce périphérique.

```
-> vxFdtDtbDump
root @ {
   compatible = "raspberrypi,rpi-4";
   #address-cells = <0x00000002>;
   #size-cells = <0x00000002>;
   model = "Raspberry Pi 4 Model B - ARMv8";
   cpus {
      #size-cells = <0x00000000>;
      #address-cells = <0x00000002>;
      cpu@0 {
         device_type = "cpu";
         reg = <0x00000000 0x00000000>;
         enable-method = "spin-table";
         cpu-release-addr = <0x00000000 0x000000d8>;
      }
      cpu@1 {
         device_type = "cpu";
         reg = <0x00000000 0x00000001>;
         enable-method = "spin-table";
         cpu-release-addr = <0x00000000 0x000000e0>;
      }
```

9) Les pilotes de périphériques

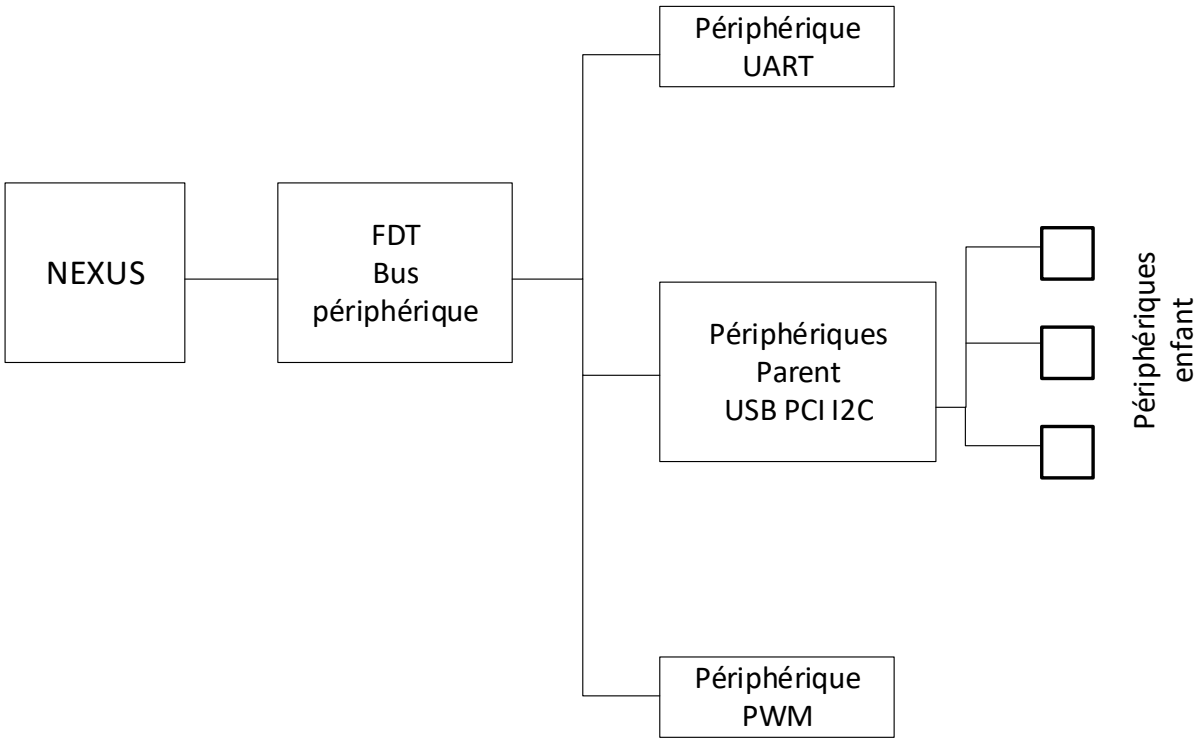
9.1) L'architecture VxBus

L'architecture des pilotes est basée sur VxBus (Vxworks bus) . VxBus fournit un modèle de pilote de périphérique cohérent qui est à la fois indépendant de l'architecture et du BSP

VxBus est destiné à fournir aux développeurs des pilotes de périphériques les fonctionnalités suivantes :

- La possibilité d'organiser les appareils dans une topologie simple.
- La possibilité de créer des pilotes pouvant être chargés et déchargés à la volée.
- Un mécanisme de gestion des ressources de l'appareil (BARs, IRQs).
- Un mécanisme de gestion des relations parent/enfant.
- Amélioration de la portabilité du pilote.
- Indépendance du BSP et de l'architecture.

Vxbus se sert du device tree afin de gérer les appareils dans une topologie simple :

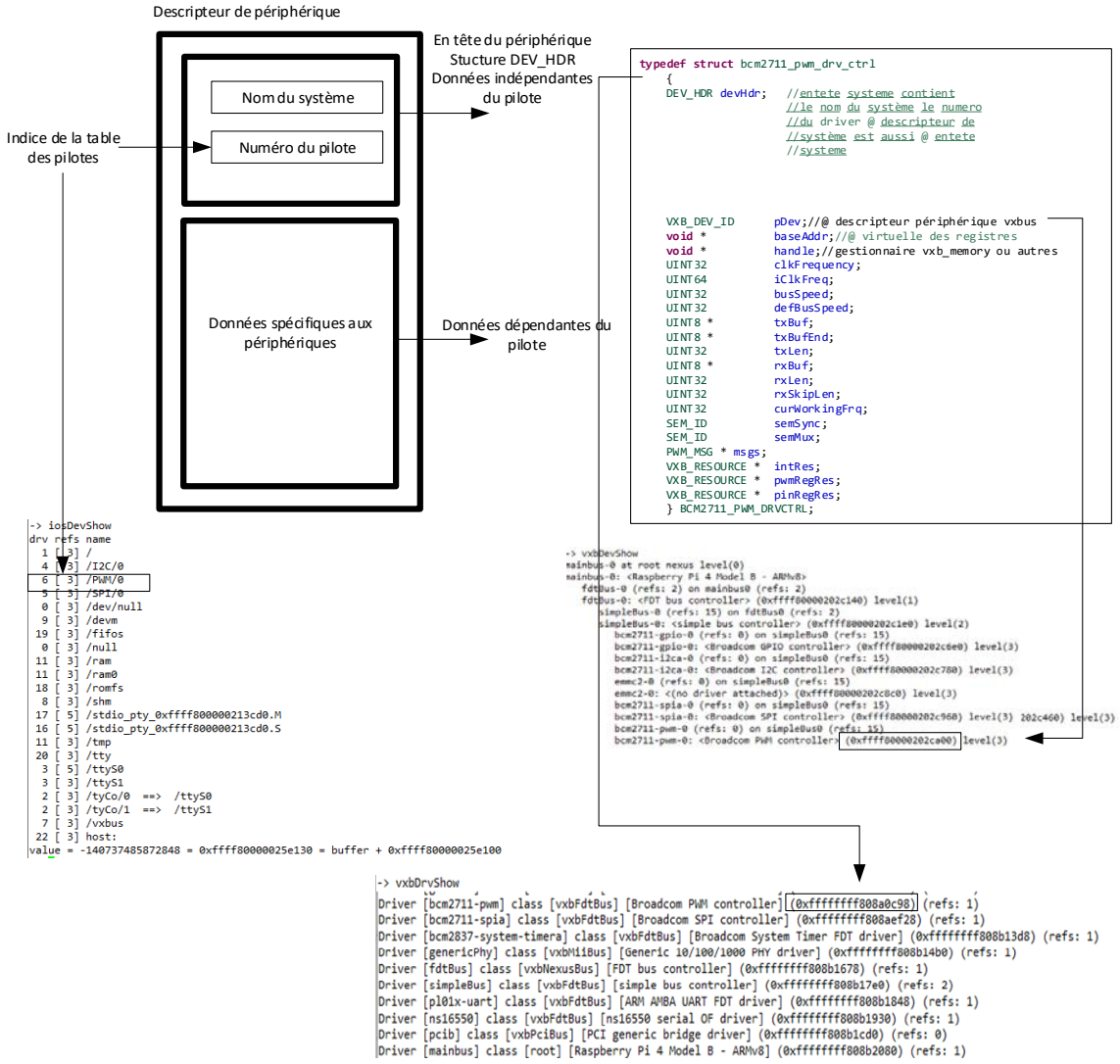


9.2) Description en langage C d'un périphérique

Les descripteurs de périphériques

Chaque périphérique est décrit par une structure avec deux parties :

- ✓ Une partie indépendante du périphérique DEV_HDR le nom, le numéro l'@ du descripteur système
- ✓ Une partie dépendante du périphérique registres, interruption, broches, fréquence …



La structure de pilote et de gestion de pilote `VXB_DRV`

Les pilotes dans VxBus sont définis à l'aide d'une structure VXB_DRV qui comprend des informations d'identification et une table de méthodes. Cette structure est utilisée pour gérer les pilotes dans le système VxWorks.

Initialisation de la structure

```
VXB_DRV vxbFdtBcm2711GP0CtlDrv =
    {
        { NULL } ,
        "bcm2711-gpoctl",        /* Name */
        "Broadcom GPO CLK  controller",       /* Description */
             VXB_BUSID_FDT,      /* Class */
        0,                  /* Flags */
        0,                  /* Reference count */
             bcm2711GP0CtlDeviceMethods             /* Method table */
    } vxbSampleDrv;
```

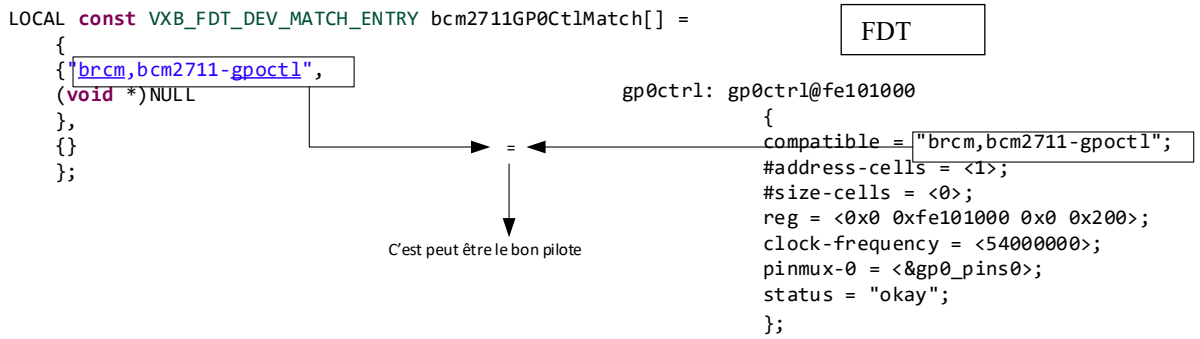
```
LOCAL VXB_DRV_METHOD bcm2711GP0CtlDeviceMethods[] =
    {
    /* DEVICE API */
    { VXB_DEVMETHOD_CALL(vxbDevProbe), vxbBcm2711GP0CtlProbe },
    { VXB_DEVMETHOD_CALL(vxbDevAttach), vxbBcm2711GP0CtlAttach },
    { VXB_DEVMETHOD_CALL(vxbDevShutdown), vxbBcm2711GP0CtlShutdown },
    { VXB_DEVMETHOD_CALL(vxbDevDetach), vxbBcm2711GP0CtlDetach },
       VXB_DEVMETHOD_END
    };
```

Un tableau de méthodes contient le plus souvent les méthodes de sonde `vxbDevProbe` , d'attachement, d'arrêt et de détachement et leurs correspondances avec les méthodes du fichier source.
La liste se termine par une entrée vide. La définition du pilote vxbSampleDrv inclut un pointeur vers cette table.
La macro VXB_DRV_DEF( ) a une fonction spéciale ; elle déclare la structure du pilote comme faisant partie d'un ensemble de liens.

```
VXB_DRV_DEF(vxbFdtBcm2711GP0CtlDrv);
```

C' est cet ensemble de liens que le noyau va utiliser pour appeler toutes les méthodes probe des pilotes et comparer le nom du périphérique du device tree avec celui fournit par la méthode probe

un tableau de correspondance est utilisé par la méthode probe du pilote pour faire cette comparaison.



La fonction vxbDevProbe( ) fait correspondre le périphérique avec un pilote existant en appelant la méthode vxbFdtDevMatch( ) ou vxbPCIDevMatch( ) pour du PCI

```
LOCAL STATUS vxbBcm2711GP0CtlProbe
    (
    VXB_DEV_ID pDev
    )
    {
    /*
     * Probe logic really depends on the device.
     */
       //SPI_DBG (SPI_DBG_INFO, "probing for device match.\n");

           return vxbFdtDevMatch (pDev, bcm2711GP0CtlMatch, NULL);
    }
```

la fonction vxbFdtDevMatch( ) fait correspondre le périphérique du DTB avec le pilote.
Un pilote non attaché

```
gp0ctrl-0 (refs: 0) on simpleBus0 (refs: 15)
gp0ctrl-0: <(no driver attached)> (0xffff80000202cb40) level(3)
```

La méthode vxbDevAttach( ) permet d'attacher le périphérique au driver en :

- ✓ Allouant des ressources mémoires pour le pilote
- ✓ Récupérer les informations du device tree
- ✓ Appelle la fonction d'initialisation du périphérique
- ✓ Manipule les interruptions
- ✓ Appelle ttyDevCreate() pour créer un terminal de périphérique

9.3) La gestion des ressources mémoires

L'objectif de la gestion des ressources est de fournir une structure de ressources standard pour décrire les différents types de ressources (mémoires entrées sorties, interruption) et de les stocker dans un objet référencé par un pointeur dans la structure VXB_DEV.

VxBus GEN2 introduit le concept d' abstraction VXB_RESOURCE pour gérer les banques de registres et les interruptions. Chaque ressource a une structure parent VXB_RESOURCE commune et un enfant VXB_RESOURCE_ADR ou VXB_RESOURCE_IRQ structure.

```
typedef struct vxbresourceadr {
ULONG start;
ULONG end;
Void * virtual;
Virtual address;
Void *pHandle;
} VXB_RESOURCE_ADR ;
```

```
typedef struct vxbresourceirq {
UINT32 hvec; //vecteur matériel numéro irq
UINT32 lvec; //vecteur logique numéro de vecteur
Void *extra;
} VXB_RESOURCE_IRQ ;
```

Une fois les ressources initialisées, une structure enfant peut y accéder à l'aide de la fonction vxbResourceAlloc( )

```
VXB_DEV_ID pDev;
VXB_RESOURCE * pRes;
pRes = vxbResourceAlloc (pDev, VXB_RES_IRQ, 0);
```

fonction d'allocation de ressource mémoire

```
VXB_RESOURCE * vxbResourceAlloc
    (
    VXB_DEV_ID pDev,    /* périphérique à qui on souhaite allouer la ressource */
    UINT16     type,    /* type de ressource à allouer */
    UINT16     idx      /* identifier of resource to allocate */
    )
```

La structure VXB_RESOURCE_ADR utilisée avec les ressources de la banque de registres contient l'adresse de début physique de la banque de registres, son adresse virtuelle et sa taille. Les deux types de ressources sont VXB_RES_MEMORY et VXB_RES_IO. VXB_RES_IO n'est utilisé que pour le PCI sur la plateforme ARM

Exemple : Pilote I2C

```
VXB_RESOURCE_ADR *   pResAdr = NULL;// structure de ressource mémoire pour les I/O
VXB_RESOURCE *       pRes = NULL;// structure de ressource mémoire pour le drivers
pRes = vxbResourceAlloc(pDev, VXB_RES_MEMORY, 0);// allocation des ressources mémoire pour la
structure VXB_RESOURCE information ressource mémoire drivers
    //pRes = vxbResourceAlloc(pDev, VXB_RES_IO, 0);
    if (pRes == NULL)
        {
        I2C_DBG (I2C_DBG_ERR, "failed to alloc memory resource\n");
        goto errOut;
```

```
        }

    pResAdr = (VXB_RESOURCE_ADR *)pRes->pRes;//the member pRes de VXB_RESOURCE est un objet de
type VXB_RESOURCE_ADR
    if (pResAdr == NULL)
        {
        I2C_DBG (I2C_DBG_ERR, "failed to alloc memory address resource\n");
        goto errOut;
        }
```

intRes	VXB_RESOURCE *	{node={next=0x0000000000000000}, id=0x01030000, p..
node	SL_NODE	{next=0x0000000000000000}
id	UINT32	0x01030000
pRes	void *	0xffff80000202026f0
owner	VXB_DEV_ID	0xffff80000202c6e0
vxbSerial	VXB_KEY	0x0000000000000012
pClk	VXB_CLK_ID	{clkNode={next=0x0000000000000000, previous=0xfff..
pResAdr	VXB_RESOURCE_ADR *	{start=0x00000000fe804000, virtAddr=0xffff8000068..
start	PHYS_ADDR	0x00000000fe804000 — @phys obtenue par FDT registreS I2C
virtAddr	VIRT_ADDR	0xffff80000683f000 — @ virtuelle MMU I2C
size	size_t	0x0000000000001000 — TAILLE MEMOIRE REGISTRES
pHandle	void *	0x0000000000000020 — TYPE VXB_RES_MEMORY
pRes	VXB_RESOURCE *	{node={next=0xffff800002022720}, id=0x01010000, p..
node	SL_NODE	{next=0xffff800002022720}
id	UINT32	0x01010000
pRes	void *	0xffff800002028760
owner	VXB_DEV_ID	0xffff80000202c6e0
vxbSerial	VXB_KEY	0x0000000000000011
pFdtDev	VXB_FDT_DEV *	{name=[0x69, 0x32, 0x63, 0x00, 0x33, 0x66, 0x38, ..
i2clnClk	UINT64	0x000000001dcd6500
pValue	void *	0xffffffff8001110c
polled_mode	UINT32	0x00000000

```
i2c1: i2c@fe8040000
            {
            compatible = "brcm,bcm2711-i2ca";
            reg = <0x0 0xfe804000 0x0 0x1000>;
            interrupts = <149 0 4>;
            interrupt-parent = <&intc>;
            def-bus-frequency = <100000>;
            clock-frequency = <500000000>;
           pinmux-0 = <&i2c1_pins>;
            #address-cells = <1>;
            #size-cells = <0>;
            polled-mode = <0>;
            status = "okay";
            };
```

9.4) La gestion des ressources physiques

a) La gestion des broches d'entrée sorties

Certains périphériques disposent d'entrées sorties qui sont déclarées par la variable pinmux-o dans le device tree de la manière suivante

```
i2c1: i2c@fe8040000
            {
            compatible = "brcm,bcm2711-i2ca";
            reg = <0x0 0xfe804000 0x0 0x1000>;
            interrupts = <149 0 4>;
            interrupt-parent = <&intc>;
            def-bus-frequency = <100000>;
            clock-frequency = <500000000>;
          pinmux-0 = <&i2c1_pins>;
            #address-cells = <1>;
            #size-cells = <0>;
            polled-mode = <0>;
            status = "okay";
            };
```

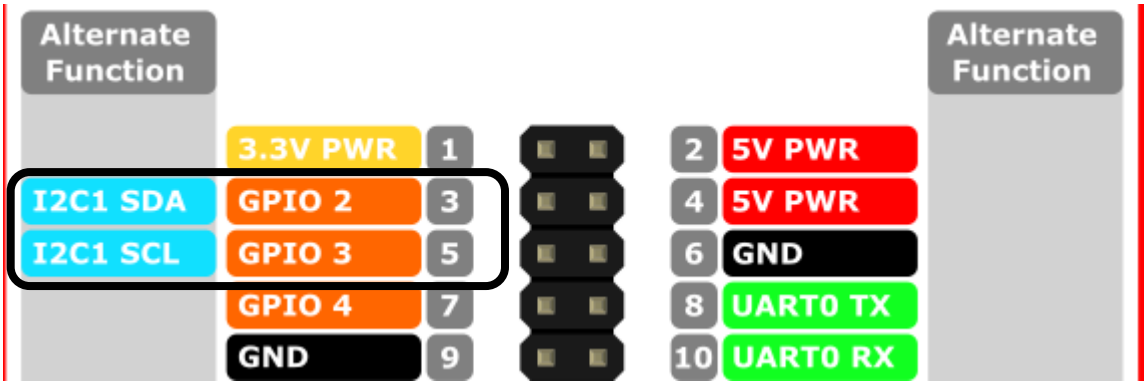
La variable pinmux-o est associée à l'adresse **i2c1_pins** qui est définie par :

```
i2c1_pins: i2c1Pins
              {
              pin-set = <2 4 1  /* gpio2 = sda, ALT0, PullUp */
                        3 4 1>; /* gpio3 = scl, ALT0, PullUp */
              };
```

Numéro du gpio

```
* Pin Mode selection:
                *
                * 0 = float
                * 1 = pull-up
                * 2 = pull-down
                *
```

```
/*
              * Function selection:
              *
              * 0 = GPIO pin is an input
              * 1 = GPIO pin is an output
              * 4 = GPIO pin takes alternate function 0
              * 5 = GPIO pin takes alternate function 1
              * 6 = GPIO pin takes alternate function 2
              * 7 = GPIO pin takes alternate function 3
              * 3 = GPIO pin takes alternate function 4
              * 2 = GPIO pin takes alternate function 5
              *
```



GPIO	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	PCLK	SPI3_CE0_N	TXD2	SDA6
GPIO1	High	SCL0	SA4	DE	SPI3_MISO	RXD2	SCL6
GPIO2	High	SDA1	SA3	LCD_VSYNC	SPI3_MOSI	CTS2	SDA3
GPIO3	High	SCL1	SA2	LCD_HSYNC	SPI3_SCLK	RTS2	SCL3
GPIO4	High	GPCLK0	SA1	DPI_D0	SPI4_CE0_N	TXD3	SDA3

Afin de vérifier la conformité de ses broches d'entrées sorties la fonction **vxbPinMuxEnable( )** est utilisée.

vxbPinMuxEnable( ) - enable the pinmux for the default state of a device
STATUS vxbPinMuxEnable
(
VXB_DEV_ID pDev /* pinmux device handle */
)

Cette routine essaie d'activer le pinmux pour l'état par défaut d'un appareil spécifique. Normalement, un appareil client n'en a besoin que pour activer sa configuration PINMUX par défaut lors de l'initialisation au démarrage.

b) La gestion des horloges

Certains périphériques sont reliées aux horloges internes du circuit. On peut déclarer la fréquence de des horloges dans le device tree

```
i2c1: i2c@fe8040000
            {
            compatible = "brcm,bcm2711-i2ca";
            reg = <0x0 0xfe804000 0x0 0x1000>;
            interrupts = <149 0 4>;
            interrupt-parent = <&intc>;
            def-bus-frequency = <100000>;
            clock-frequency = <500000000>;
          pinmux-0 = <&i2c1_pins>;
            #address-cells = <1>;
            #size-cells = <0>;
            polled-mode = <0>;
            status = "okay";
            };
```

vxbClkEnableAll( ) activer toutes les horloges utilisées par l'appareil. Cette routine active toutes les horloges utilisées par l'appareil.

Les informations d'horloge sont liées aux nœuds de l'arborescence des périphériques.

```
if (vxbClkEnableAll (pDev) == ERROR)
        {
        I2C_DBG (I2C_DBG_ERR, "failed to enable clocks\n");
        return ERROR;
        }
```

OK si toutes les horloges sont activées, ERROR si l'activation de toutes les horloges a échoué.

vxbClkGet( ) – retourne le gestionnaire d'horloge

```
VXB_CLK_ID vxbClkGet
  (
  VXB_DEV_ID   pDev,     /* pointer to VxBus device */
  const char * pClkName   /* name to lookup         */
  )
```

Cette routine utilise deux manières pour trouver un gestionnaire d'horloge : d'abord, elle essaiera de rechercher tous les registres d'horloge dans la liste d'horloge fdt qui correspondent au nom de l'horloge. si cela échoue, elle essaiera de trouver dans la plate-forme domaines d'horloge le nom de l'horloge. une fois trouvée, le gestionnaire de l'horloge est renvoyée.

```
pClk = vxbClkGet (pDev, NULL);
    if (pClk == NULL)
        {
        I2C_DBG (I2C_DBG_ERR, "failed to get clock ID.\n");
        (void) vxbClkDisableAll (pDev);
        return ERROR;
        }
```

vxbClkRateGet( ) - Cette routine essaie d'obtenir la fréquence d'horloge

```
UINT64 vxbClkRateGet
  (
  VXB_CLK_ID pClk   /* clock handle */
  )
i2cInClk = vxbClkRateGet (pClk);//500 Mhz dans dtb
    if (i2cInClk == CLOCK_RATE_INVALID)
        {
        I2C_DBG (I2C_DBG_ERR, "failed to get clock rate. Using default frequency\n");
        i2cInClk = BCM2711_I2C_ICLK_FREQ;
        }
```

c) La gestion des interruptions

L'allocation de la ressource mémoire est aussi nécessaire avec comme type de ressource VXB_RES_IRQ

Pour les fonctions d'interruption, les fonctions suivantes sont disponibles, qui utilisent toutes le gestionnaire de ressource d'interruption VxBus :

```
VXB_RESOURCE *  intRes ;
pDrvCtrl->intRes = vxbResourceAlloc (pDev, VXB_RES_IRQ, 0);
        if (pDrvCtrl->intRes == NULL)
            {
            I2C_DBG (I2C_DBG_ERR, "vxbResourceAlloc VXB_RES_IRQ fail.\n");
            goto errOut;
            }
```

La fonction vxbIntConnect Lie un ISR à une ressource d'interruption de périphérique.

```
STATUS vxbIntConnect
    (
    VXB_DEV_ID      pDev,    /* descripteur de périphérique              */
    VXB_RESOURCE  * pRes,    /* pointeur sur structure de ressource mémoire */
    VOIDFUNCPTR     pIsr,    /* pointeur sur ISR routine d'interruption      */
    void *          pArg     /* paramètre d'entrée ISR                   */
    )
```

La fonction vxbIntEnable valide une ressource d'interruption (au niveau du PIC programmable interrupt controler).

```
STATUS vxbIntEnable
    (
    VXB_DEV_ID      pDev,    /* descripteur de périphérique
    VXB_RESOURCE  * pRes     /* pointeur sur structure de ressource mémoire */
    )
```

Dissocie un ISR d'une ressource d'interruption de périphérique.
vxbIntDisconnect( )
Masque une ressource d'interruption (au niveau PIC).
vxbIntDisable( )
Dans un système multicœur, achemine une interruption vers un cœur spécifique.
vxbIntReroute( )

```
pDrvCtrl->intRes = vxbResourceAlloc (pDev, VXB_RES_IRQ, 0);
 if (pDrvCtrl->intRes == NULL)
            {
            I2C_DBG (I2C_DBG_ERR, "vxbResourceAlloc VXB_RES_IRQ fail.\n");
            goto errOut;
            }
 else
     {
 if (OK != vxbIntConnect (pDev, pDrvCtrl->intRes,(VOIDFUNCPTR) vxbBcm2711I2cIsr,pDev))
                {
                 I2C_DBG (I2C_DBG_ERR, "vxbIntConnect failed.\n");
                 goto errOut;
                }
            else
                {
                if (OK != vxbIntEnable (pDev, pDrvCtrl->intRes))
                    {
                    I2C_DBG (I2C_DBG_ERR, "vxbIntEnable fail.\n");
                    (void) vxbIntDisconnect (pDev, pDrvCtrl->intRes);
                    goto errOut;
                    }
                printf("I2C Interrupt was  installed.\n");
                }
            }
```

Attention dans l'ISR pas d'accès possible sur les entrées sorties

d) Les interruptions dans le device tree

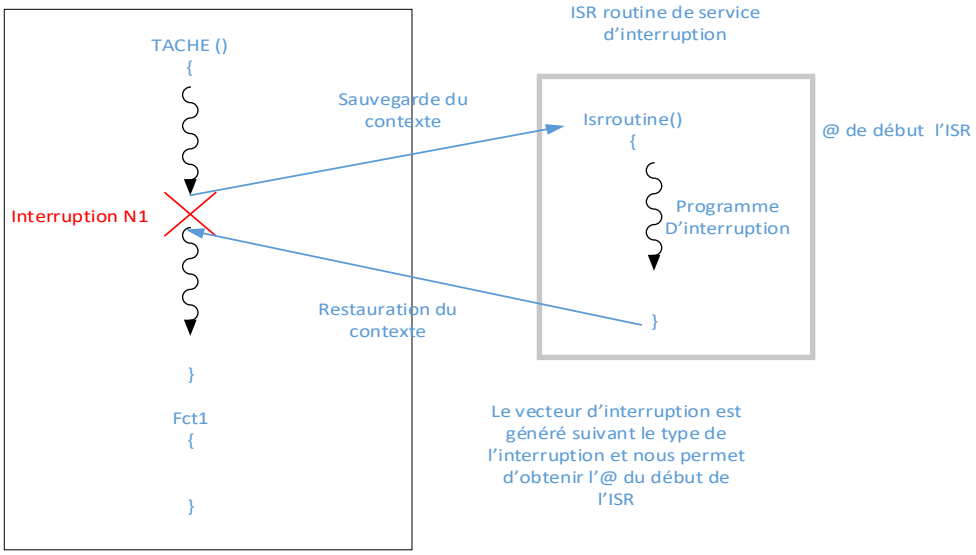
d.1) notion

Les interruptions sont des demandes de service effectuées par des périphériques du SOC
Le SOC 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 SOC 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 SOC 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 SOC 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 de la tache en cours (après exécution de l'instruction en cours), afin d'effectuer un autre programme (programme d'interruption). A l'issu de ce dernier la tache reprend exactement là où elle a été interrompu. Il existe donc sur les SOC des broches du GPIO, I2C, SPI, ETHERNETdédiées à la réception des demandes d'interruptions. Une interruption est le plus souvent asynchrone : elle peut survenir à tout instant.

Une interruption survient en général pour :

- ✓ signaler un événement sur un périphérique ;
- ✓ demander un dialogue entre un périphérique et le cœur de CPU qui déroule les taches .



d.2) Les mots clés du device tree:

#interrupt-cells

Cette propriété spécifie le nombre de paramètres qu'elle reçoit d'un périphérique utilisateur d'interruption.
Pour cette version de ce pilote, il peut s'agir de 1 ou 3.

Si #interrupt-cells a un 1 paramètre, alors le paramètre est le numéro de vecteur d'interruption.
exemple : interruptions = 34 ; cela signifie que le numéro d'interruption est 34.

Si #interrupt-cells a 3 paramètres, le premier paramètre est le numéro de vecteur d'interruption, **le second est le type d'interruption (0 pour SPI, 1 pour PPI),**

le dernier correspond aux drapeaux d'interruption :

bit[3:0] :
type de déclenchement d'interruption. 0x1 pour déclenchement sur front montant, 0x4 pour déclenchement sur niveau haut actif.

bit[15:8] : **interrupt CPU mask.** indicate which CPU will handle this interrupt.
each bit represents one core, starting from core 0. If none bit is set, core 0 will be chosen to handle this interrupt.
Note: for SPI, only one CPU target is allowed.
bit[23:16]: **interrupt priority.**

Example: **interrupts = 34 0 4; cela signifie que le vecteur d'interruption est 34, le type est SPI, la priorité est 0, la liaison au processeur 0, le déclenchement niveau haut actif.**

```
max-int-lvl
Cette propriété spécifie le nombre maximal de demandes d'interruption prises en charge par la
puce.

interrupt-controller

Cette propriété spécifie que ce nœud est un contrôleur d'interruption du SOC.

Exemple

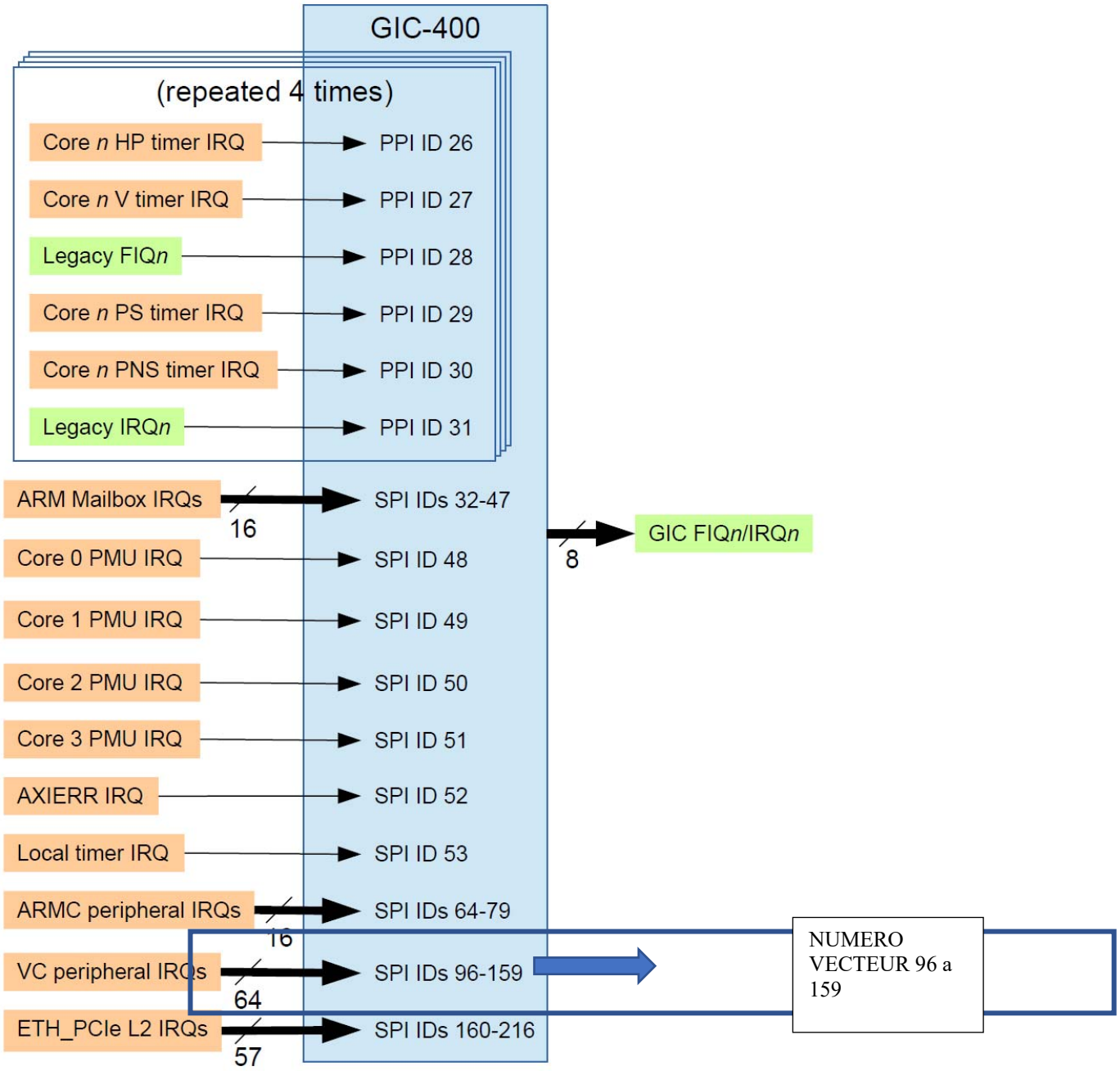
intc: interrupt-controller@00a01000
    {
    compatible = "arm,gic";
    interrupt-controller;
    #interrupt-cells = <3>;
    reg = <0x00a01000 0x1000>,
          <0x00a00100 0x100>;
    max-int-lvl = <160>;
    };
explanation of some abbreviations

  CPC - Cross Processor Call
  IPI - Inter Processor Interrupt
  GIC - General Interrupt Controller
  SGI - Software Generated Interrupt
  PPI - Private Peripheral Interrupt
  SPI - Shared Peripheral Interrupt


gpio: gpio@fe200000
            {
            compatible = "brcm,bcm2711-gpio";
          gpio-controller;
            #gpio-cells = <4>;
            reg = <0x0 0xfe200000 0x0 0x00f4>;
            interrupts = <145 0 1
                          146 0 1>;
            interrupt-parent = <&intc>;
            pinmux-0 = <&gpio_pins0>;
            status = "okay";
            };
```

d.3) Les sources d'interruption

La raspberry PI4 (SOC BROADCOM 2711) dispose d'un contrôleur d'interruption GIC-400 du type PPI et SPI

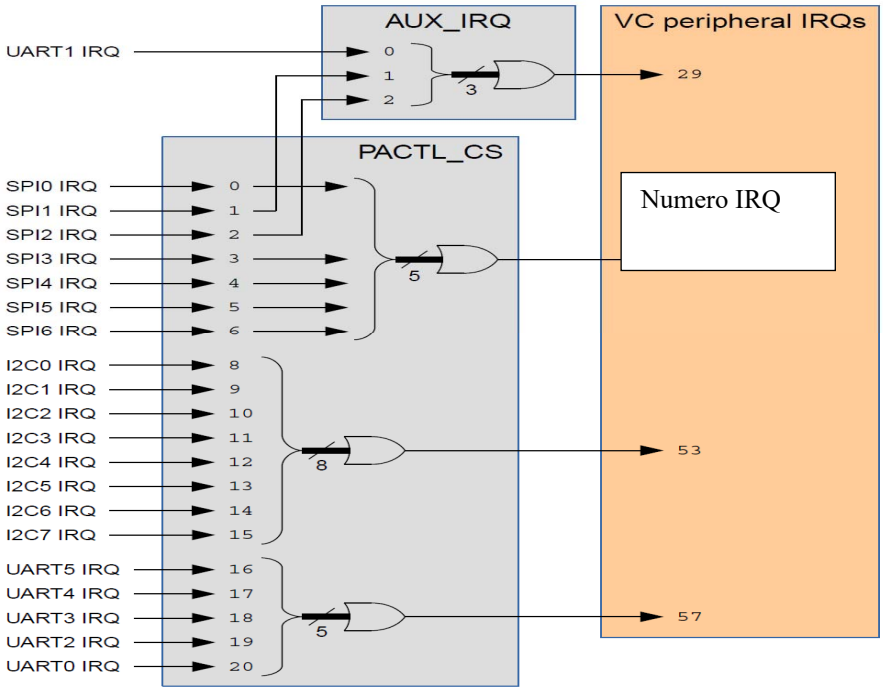


VC = videocore 64 NUMERO IRQ

Le videocore dispose de 64 lignes d'interruption (Numero IRQ)

#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0	Timer 0	16	DMA 0	32	HDMI CEC	48	SMI
1	Timer 1	17	DMA 1	33	HVS	49	GPIO 0
2	Timer 2	18	DMA 2	34	RPIVID	50	GPIO 1
3	Timer 3	19	DMA 3	35	SDC	51	GPIO 2
4	H264 0	20	DMA 4	36	DSI 0	52	GPIO 3
5	H264 1	21	DMA 5	37	Pixel Valve 2	53	**OR of all I2C**
6	H264 2	22	DMA 6	38	Camera 0	54	**OR of all SPI**
7	JPEG	23	**DMA 7 & 8**	39	Camera 1	55	PCM/I2S
8	ISP	24	**DMA 9 & 10**	40	HDMI 0	56	SDHOST
9	USB	25	DMA 11	41	HDMI 1	57	**OR of all PL011 UART**
10	V3D	26	DMA 12	42	Pixel Valve 3	58	**OR of all ETH_PCIe L2**
11	Transposer	27	DMA 13	43	SPI/BSC Slave	59	VEC
12	Multicore Sync 0	28	DMA 14	44	DSI 1	60	CPG
13	MultiCore Sync 1	29	**AUX**	45	Pixel Valve 0	61	RNG
14	MultiCore Sync 2	30	ARM	46	**Pixel Valve 1 & 4**	62	**EMMC & EMMC2**
15	MultiCore Sync 3	31	DMA 15	47	CPR	63	ETH_PCIe secure

Certaines sources partagent le même numéro d'IRQ et donc le même numéro de vecteur (bus SPI I2C et UART). Il faudra donc chercher dans les flags des registres d'interruptions des périphériques la source de l'interruption si plusieurs périphériques se partagent le même numéro de vecteur.



Pour l'I2C1 on aura donc le vecteur 96+53=149

```
i2c1: i2c@fe8040000
            {
            compatible = "brcm,bcm2711-i2ca";
            reg = <0x0 0xfe804000 0x0 0x1000>;
            interrupts = <149 0 4>;
            interrupt-parent = <&intc>;
            def-bus-frequency = <100000>;
            clock-frequency = <500000000>;
          pinmux-0 = <&i2c1_pins>;
            #address-cells = <1>;
            #size-cells = <0>;
            polled-mode = <0>;
            status = "okay";
            };
```

La commande isrshow visualise les routines de service d'interruption (ISR)mise en œuvre avec le numéro de vecteur associé (tag)

```
-> isrShow
ISR ID             Name                    Tag        HandlerRtn
------------------ ----------------------- ---------- ----------------------------
0xffff800000009560 isr1                    27         ffffffff801a4404
0xffff800000009660 isr2                    99         bcmSystTimerInt
0xffff8000000098a0 isr3                    125        ns16550vxbInt
0xffff800000009f80 isr4                    153        ffffffff803f73e0
0xffff80000000a080 isr5                    145        ffffffff80314834
0xffff80000000a340 isr6                    149        ffffffff802d5b90
0xffff80000000a6c0 isr7                    150        ffffffff803a7cdc
0xffff80000000ab00 isr8                    189        ffffffff80316160
0xffff8000001db820 isr9                    256        ffffffff80300504
0xffff8000001db900 isr10                   257        ffffffff802cf604
0xffff80000021bf20 isr11                   258        windRescheduleNotify
value = 0 = 0x0
->
```

La commande vxbIntShow visualise les paramètres d'interruption de chaque périphérique

```
-> vxbIntShow
Number of registered interrupt controllers: 2
Vector entry lVec 145: ──────────  numero de vecteur
Name: [bcm2711-gpio:0]
Interrupt service routine: 0xffffffff8040fc8c
ISR argument: 0xffff80000000a080
LVec value in interrupt resource is correct
Resource owner: [bcm2711-gpio0]
Resource ID: 0x01030000
Resource hVec: 145
Resource lVec: 145
Resource ISR pointer: 0xffffffff80314834
Resource ISR context: 0xffff800002266000
Resource flags: 0x00000001
CPU mask: 0x00000001  Triggering: edge  Polarity: active high  Priority: 0
Flags: 0x00000001
Entry is: [CONNECTED] [NOT ENABLED]
Interrupt controller device: [armgic0]
```

Les interruptions sont plébiscitées afin d'éviter le gaspillage de ressource processeur mais peu appréciées pour le déterminisme global de l'OS car une tache peut être préemptée à tout moment par une routine de service d'interruption. Dans un contexte monotache on peut demander dans une zone critique d'une tache à verrouiller les interruptions afin de garder ce déterminisme global (fonction intLock)

int intLock (void)

Cette routine désactive les interruptions. Il peut être appelé depuis le niveau interruption ou tâche. La routine intLock( ) renvoie une clé de verrouillage dépendante de l'architecture représentant le niveau d'interruption avant l'appel ; cette clé peut être passée à intUnlock() pour réactiver les interruptions.
L'appel d'une routine système VxWorks avec des interruptions verrouillées peut entraîner la réactivation des interruptions pendant une période de temps indéterminée. Si la routine appelée se bloque ou si une tâche de priorité plus élevée devient éligible à l'exécution (READY), les interruptions seront réactivées pendant l'exécution d'une autre tâche ou pendant que le noyau est inactif.
Pour empêcher les interruptions d'être réactivées dans le cas où une routine appelée entraîne l'éligibilité d'une tâche de priorité supérieure à l'exécution, la primitive taskLock() peut être utilisée pour désactiver la replanification. Notez que si une tâche est bloquée ou suspendue, le planificateur sélectionnera toujours la tâche prête la plus prioritaire à exécuter (ou à devenir inactive),que la tâche ait ou non verrouillé la préemption via taskLock(), et ainsi les interruptions seront réactivées.
Le niveau de verrouillage d'interruption est un attribut d'une tâche, c'est-à-dire qu'il fait partie du contexte de la tâche. Ainsi, si une tâche désactive les interruptions et invoque ensuite une routine système VxWorks qui provoque le blocage de la tâche appelante ou la préparation d'une tâche de priorité supérieure, le niveau de verrouillage d'interruption sera restauré lorsque l'exécution de la tâche sera reprogrammée ultérieurement. Par exemple, dans le fragment de code suivant, les interruptions seront désactivées après le retour de l'invocation de
taskDelay( ) :

```
lockKey = intLock ();

   ... (work with interrupts locked out)

  taskDelay (sysClkRateGet() * 10);   /* delay for 10 seconds */

   ... (work with interrupts locked out)
```

Finally, the above descriptions only applies for tasks since ISRs are not permitted to block or suspend, and task rescheduling only occurs when an ISR completes execution.

EXAMPLES

```
  lockKey = intLock ();

   ... (work with interrupts locked out)

  intUnlock (lockKey);
```

To lock out interrupts and task scheduling as well (see WARNINGS above):

```
  if (taskLock() == OK)
    {
    lockKey = intLock ();

    ... (critical section)

    intUnlock (lockKey);
    taskUnlock();
    }
   else
    {
    ... (error message or recovery attempt)
    }
```

intUnlock( )

```
void intUnlock
  (
  int lockKey   /* lock-out key returned by preceding intLock() */
  )
```

Cette routine réactive les interruptions qui ont été désactivées par intLock( ). Le paramètre lockKey est une clé de verrouillage dépendante de l'architecture renvoyée par un appel intLock( ) précédent.

d.4) La gestion des interruptions en C

L'allocation de la ressource mémoire est aussi nécessaire avec comme type de ressource VXB_RES_IRQ
Pour les fonctions d'interruption, les fonctions suivantes sont disponibles, qui utilisent toutes le gestionnaire de ressource d'interruption VxBus :

```
VXB_RESOURCE *  intRes ;
pDrvCtrl->intRes = vxbResourceAlloc (pDev, VXB_RES_IRQ, 0);
        if (pDrvCtrl->intRes == NULL)
            {
            I2C_DBG (I2C_DBG_ERR, "vxbResourceAlloc VXB_RES_IRQ fail.\n");
            goto errOut;
            }
```

La fonction vxbIntConnect Lie un ISR à une ressource d'interruption de périphérique.

```
STATUS vxbIntConnect
    (
    VXB_DEV_ID      pDev,   /* descripteur de périphérique              */
    VXB_RESOURCE  * pRes,   /* pointeur sur structure de ressource mémoire */
    VOIDFUNCPTR     pIsr,   /* pointeur sur ISR routine d'interruption      */
    void *          pArg    /* paramètre d'entrée ISR                   */
    )
```

La fonction vxbIntEnable valide une ressource d'interruption (au niveau du PIC programmable interrupt controler).

```
STATUS vxbIntEnable
    (
    VXB_DEV_ID      pDev,   /* descripteur de périphérique
    VXB_RESOURCE  * pRes    /* pointeur sur structure de ressource mémoire */
    )
```

Dissocie un ISR d'une ressource d'interruption de périphérique.
vxbIntDisconnect( )
Masque une ressource d'interruption (au niveau PIC).
vxbIntDisable( )
Dans un système multicœur, achemine une interruption vers un cœur spécifique.
vxbIntReroute( )

```
pDrvCtrl->intRes = vxbResourceAlloc (pDev, VXB_RES_IRQ, 0);
 if (pDrvCtrl->intRes == NULL)
            {
            I2C_DBG (I2C_DBG_ERR, "vxbResourceAlloc VXB_RES_IRQ fail.\n");
            goto errOut;
            }
 else
     {
 if (OK != vxbIntConnect (pDev, pDrvCtrl->intRes,(VOIDFUNCPTR) vxbBcm2711I2cIsr,pDev))
                {
                 I2C_DBG (I2C_DBG_ERR, "vxbIntConnect failed.\n");
                 goto errOut;
                }
```

```
            else
                {
                if (OK != vxbIntEnable (pDev, pDrvCtrl->intRes))
                    {
                    I2C_DBG (I2C_DBG_ERR, "vxbIntEnable fail.\n");
                    (void) vxbIntDisconnect (pDev, pDrvCtrl->intRes);
                    goto errOut;
                    }
                printf("I2C Interrupt was  installed.\n");
                }
            }
```

Attention dans l'ISR pas d'accès possible sur les entrées sorties

L'accessibilité des méthodes dans le pilote

On rajoute dans le descripteur de périphérique une nouvelle structure qui nous permettra de pointer sur les différentes fonctions du pilote

```
typedef struct bcm2711_GP0Ctl_drv_ctrl
    {
      DEV_HDR             devHdr;//entete systeme contient le nom du système le numero du driver @ descripteur de système est aussi @ entete systeme
      VXB_DEV_ID      pDev;
    void *          baseAddr;//pRegBase
    void *          handle;
    VXB_RESOURCE *  gp0ctlRegRes;
    VXB_RESOURCE *  pinRegRes;
    UINT32          clkFrequency;
    struct vxbGp0ctlFunctionality pGp0ctlFunc;
    } BCM2711_GP0CTL_DRVCTRL;


struct vxbGp0ctlFunctionality
    {
     BOOL    allocated;      /* flag used to denote timer allocation */
      STATUS (*GP0Ctlinit)
        (
                    void *      pCookie, UINT32 div
        );

    STATUS (*GP0CtlDisable)
            (
                    void *      pCookie
            );
        };
```

Dans la fonction d'attachement on fera le lien entre méthodes atteignables par l'utilisateur et déclarées dans la structure précédente et les méthodes du pilote

```
LOCAL STATUS vxbBcm2711GP0CtlAttach
    (
    VXB_DEV_ID pDev
    )
    {
      BCM2711_GP0CTL_DRVCTRL * pDrvCtrl;
      struct vxbGp0ctlFunctionality * pTFunc;
pTFunc = &pDrvCtrl->pGp0ctlFunc;
pTFunc->GP0Ctlinit = vxbBcm2711GP0CtlInit;
pTFunc->GP0CtlDisable=vxbBcm2711GP0CtlDisable;
```

Cas particulier du contrôleur GPIO

Les deux librairies vxbFdtBcm2711Gpio associé à la librairie générique vxbGpioLib permettent la mise en place des interruptions sur le GPIO sans déclaration explicite du descripteur de périphérique du GPIO. Les fonctions sont les suivantes :

La fonction vxbGpioIntConnect Lie un ISR à une ressource d'interruption de périphérique.

```
STATUS vxbGpioIntConnect
    (
    UINT32     id,    /* GPIO ID */
    VOIDFUNCPTR pIsr,   /* ISR */
    void *     pArg    /* parameter */
    )
```

La fonction vxbGpioIntConfig permet de configurer l'interruption

```
STATUS vxbGpioIntConfig
        (
        UINT32        id,     /* GPIO ID */
        INTR_TRIGER   trig,   /* trigger mode */
        INTR_POLARITY pol     /* polarity */
        )
```

Cette routine configure le mode de déclenchement d'interruption et sa polarité.

L'argument trig spécifie le mode de déclenchement. cet argument pourrait être n'importe laquelle des valeurs suivantes : INTR_TRIGGER_CONFORM :0 - ne change pas le mode de déclenchement du device tree
. INTR_TRIGGER_EDGE :1 - interruption déclenchée par front.
INTR_TRIGGER_LEVEL :2 - interruption déclenchée par le niveau.

L'argument pol spécifie une polarité d'interruption. cet argument pourrait être n'importe laquelle des valeurs suivantes : INTR_POLARITY_CONFORM :0 - ne change pas l'interruption du device tree
polarité. INTR_POLARITY_HIGH :1 - niveau logique haut/front montant.
INTR_POLARITY_LOW :2 - niveau logique bas/front descendant.

La fonction vxbGpioIntEnable valide une ressource d'interruption

```
STATUS vxbGpioIntEnable
        (
        UINT32      id,     /* GPIO ID */
        VOIDFUNCPTR pIsr,   /* ISR */
        void *      pArg    /* parameter */
        )
```

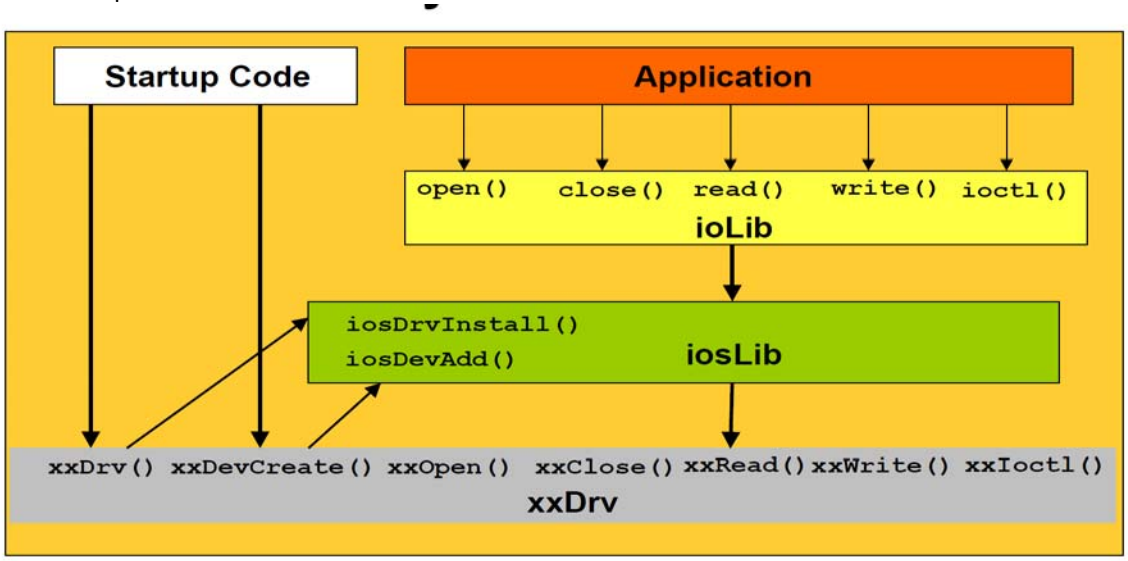
10) Association d'un périphérique à un système d' entrées sorties

10.1) Principe

on va associer un périphérique par l'intermédiaire d'un descripteur de fichier. L'appel des pilotes du périphérique se fera grâce aux méthodes :

• open/creat/close/remove
• read/write/ioctl

– un périphérique sera associé à un et un seul pilote. Le pilote gère les requêtes sur le périphérique par l'intermédiaire de ses méthodes xxopen xxwrite …..



L'utilisation d'un périphérique comme un périphérique d' entrées sorties a comme Objectif de :

– masquer les contraintes matérielles

– découpler l'application du matériel

– protéger/partager l'accès

Les Principes retenus sont :

– un périphérique est vu comme un fichier

– L'utilisation du système d'entrée/sortie (IOS)

– primitives standardisées de l'IOS

appel par périphérique

open/creat/close/remove

read/write/ioctl

– un périphérique est associé à un et un seul pilote

– le pilote gère les requêtes sur le périphérique

La commande devs permet d'obtenir la liste des descripteurs de périphériques

```
[vxWorks *]# devs
drv refs name
  1 [ 3] /
  2 [ 3] /bin  ==>  /sd0a/sysroot/bin
  2 [ 3] /boot  ==>  /sd0a/sysroot/boot
  2 [ 3] /dev  ==>  /
  2 [ 3] /etc  ==>  /sd0a/sysroot/etc
 12 [ 3] /fifos
 17 [ 3] /host.host
 15 [ 3] /input/event
  2 [ 3] /lib  ==>  /sd0a/sysroot/lib
  0 [ 3] /null
 11 [ 3] /ram0
 14 [ 3] /random
  6 [ 3] /sd0a
  4 [ 3] /shm
  9 [ 3] /stdio_pty_0xffff800000452110.M
  8 [ 3] /stdio_pty_0xffff800000452110.S
  9 [ 3] /stdio_pty_0xffff800000452bf0.M
  8 [ 3] /stdio_pty_0xffff800000452bf0.S
  9 [ 3] /stdio_pty_0xffff8000004531f0.M
  8 [ 3] /stdio_pty_0xffff8000004531f0.S
 23 [ 3] /test
 11 [ 3] /tmp
  3 [ 3] /ttyS0
  3 [ 3] /ttyS1
 14 [ 3] /urandom
  2 [ 3] /usr  ==>  /sd0a/sysroot/usr
 17 [ 3] /wrs
 13 [ 3] /zero
[vxWorks *]#
```

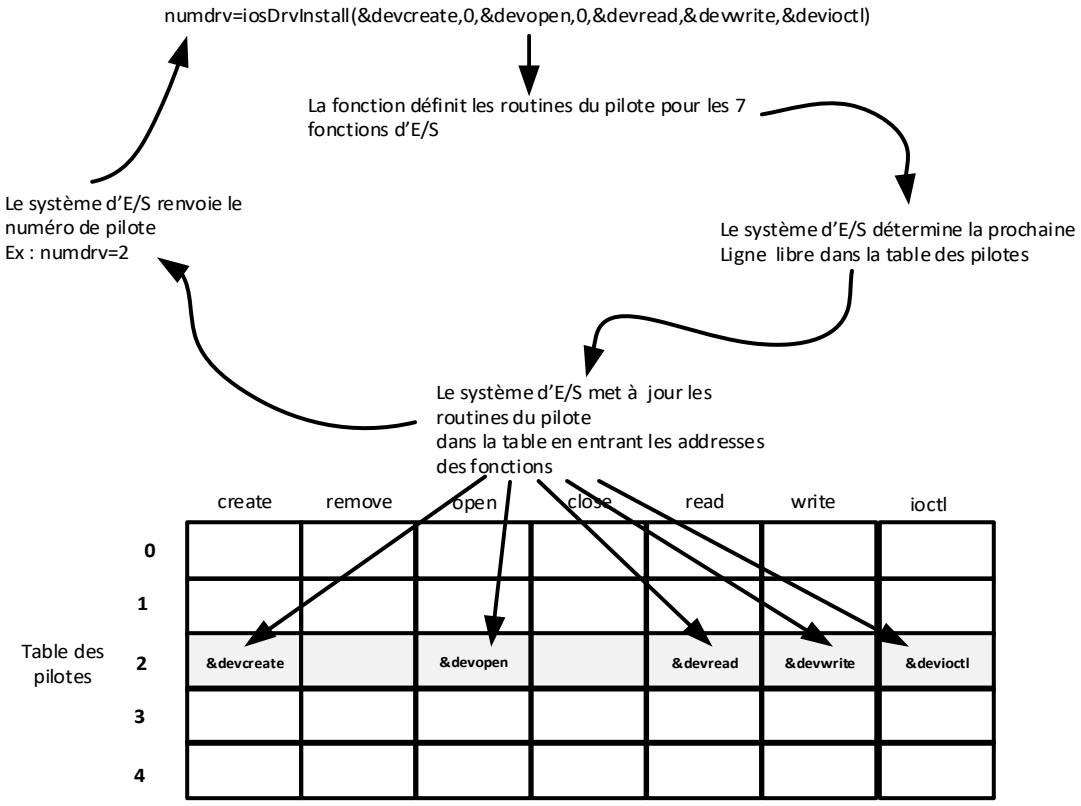
10.2) Les périphériques de type caractère

Installation d'un pilote d'entrées sorties

iosDrvInstall( ) - installe un pilote noyau d'entrées sorties

```
Cette routine doit être appelée une fois par chaque pilote d'E/S.
elle connecte les différents appels de service d'E/S aux routines de service du pilote et
attribue un numéro au pilote en ajoutant le pilote à la table des pilotes. Elle retourne le
numéro du pilote
```

```
int iosDrvInstall
    (
    DRV_CREATE_PTR pCreate,    /* unused */
    DRV_REMOVE_PTR pRemove,    /* pointer to driver remove function */
    DRV_OPEN_PTR   pOpen,      /* pointer to driver open function */
    DRV_CLOSE_PTR  pClose,     /* pointer to driver close function */
    DRV_READ_PTR   pRead,      /* pointer to driver read function */
    DRV_WRITE_PTR  pWrite,     /* pointer to driver write function */
    DRV_IOCTL_PTR  pIoctl      /* pointer to driver ioctl function */
    )
```

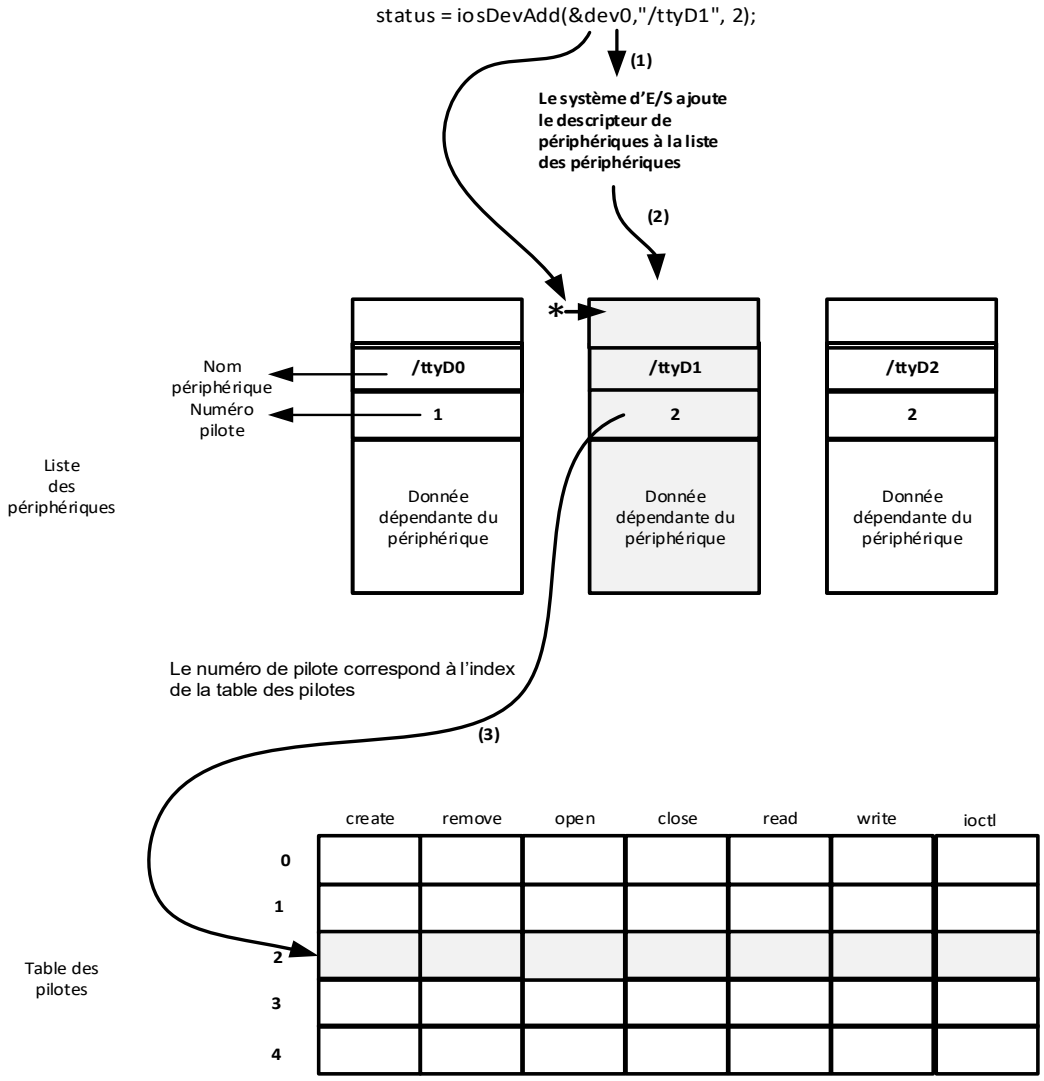



```
-> iosDrvShow( )
drv            remove            open              close             read              write             ioctl
  0  0x0               0xffffffff802d1fa0 0xffffffff802d21e0 0xffffffff802d2184 0xffffffff802d1f44 0xffffffff802d203c
  1  0xffffffff803caa94 0xffffffff803cab34 0xffffffff803caec4 0xffffffff803caef8 0xffffffff803caf2c 0xffffffff803caf60
  2  0xffffffff806eafec 0xffffffff806eb074 0xffffffff806eb394 0x0               0x0               0xffffffff806eb3c8
  3  0x0               0xffffffff802b8d38 0xffffffff802b8da4 0xffffffff802bade8 0xffffffff802ba60c 0xffffffff802b8df8
  4  0xffffffff80456ff0 0xffffffff804571ec 0xffffffff804575e4 0xffffffff804576a0 0xffffffff804576a0 0xffffffff804576d4
  5  0x0               0xffffffff802b50e0 0xffffffff802b5230 0xffffffff802b52b0 0xffffffff802b53d0 0xffffffff802b550c
  6  0xffffffff8037fd3c 0xffffffff8037ce4c 0xffffffff8037eca4 0xffffffff80380acc 0xffffffff80380b04 0xffffffff8037d8f4
  7  0x0               0xffffffff80367588 0xffffffff80367760 0xffffffff80367864 0xffffffff80367bd0 0xffffffff80367fc4
  8  0x0               0xffffffff802b6804 0xffffffff802b68b0 0xffffffff802b6940 0xffffffff802b6a68 0xffffffff802b6ad0
  9  0x0               0xffffffff802b6c30 0xffffffff802b6cb4 0xffffffff802b6d4c 0xffffffff802b6ddc 0xffffffff802b6eb4
 10  0x0               0xffffffff803bbbe8 0xffffffff803bbce8 0xffffffff803bbd3c 0xffffffff803bbda8 0xffffffff803bbdc8
 11  0xffffffff803c37b4 0xffffffff803c2904 0xffffffff803c3cb0 0xffffffff803c38e0 0xffffffff803c3ab0 0xffffffff803c4a1c
 12  0xffffffff802bbea4 0xffffffff802bc09c 0xffffffff802bc4e4 0xffffffff802bc568 0xffffffff802bc660 0xffffffff802bc750
 13  0x0               0xffffffff802d18c0 0xffffffff802d1914 0xffffffff802d192c 0xffffffff802d1960 0xffffffff802d197c
 14  0x0               0xffffffff802d1460 0xffffffff802d14b4 0xffffffff802d14cc 0xffffffff802d1590 0xffffffff802d1608
 15  0x0               0xffffffff80350038 0xffffffff80350154 0xffffffff80350278 0xffffffff803503c0 0xffffffff803504d0
 16  0x0               0x0               0xffffffff8052ed2c 0xffffffff8052ed94 0xffffffff8052eddc 0xffffffff8052ee24
 17  0xffffffff80220ebc 0xffffffff80221160 0xffffffff802217b4 0xffffffff802218e0 0xffffffff80221cac 0xffffffff80221e48
 18  0xffffffff803a2e34 0xffffffff803a2fcc 0xffffffff803a4078 0xffffffff803a41d0 0xffffffff803a44e4 0xffffffff803a4804
 19  0xffffffff803a8a90 0xffffffff803a8af4 0xffffffff803a9b34 0xffffffff803a9df4 0xffffffff803aa120 0xffffffff803aa454
 20  0x0               0xffffffff8056d0b4 0xffffffff8056d388 0xffffffff8056d474 0xffffffff8056d56c 0xffffffff8056d6b8
 21  0x0               0xffffffff80578b68 0xffffffff80578e10 0xffffffff80578ebc 0x0               0xffffffff8057903c
 22  0xffffffff8064c394 0xffffffff8064c5ac 0xffffffff8064ccd8 0xffffffff8064ceb8 0xffffffff8064d1a4 0xffffffff8064d470
 23  0x0               0xffffffff80d631f4 0xffffffff80d63230 0xffffffff80d63260 0xffffffff80d632d4 0xffffffff80d63340
value = 123 = 0x7b = '{' = buffer + 0x4b
->
```

Ajout du descripteur de périphériques dans La liste des périphériques

Les périphériques de caractère sont ajoutés dynamiquement à la liste système d'E/S en appelant la routine d'E/S interne iosDevAdd()

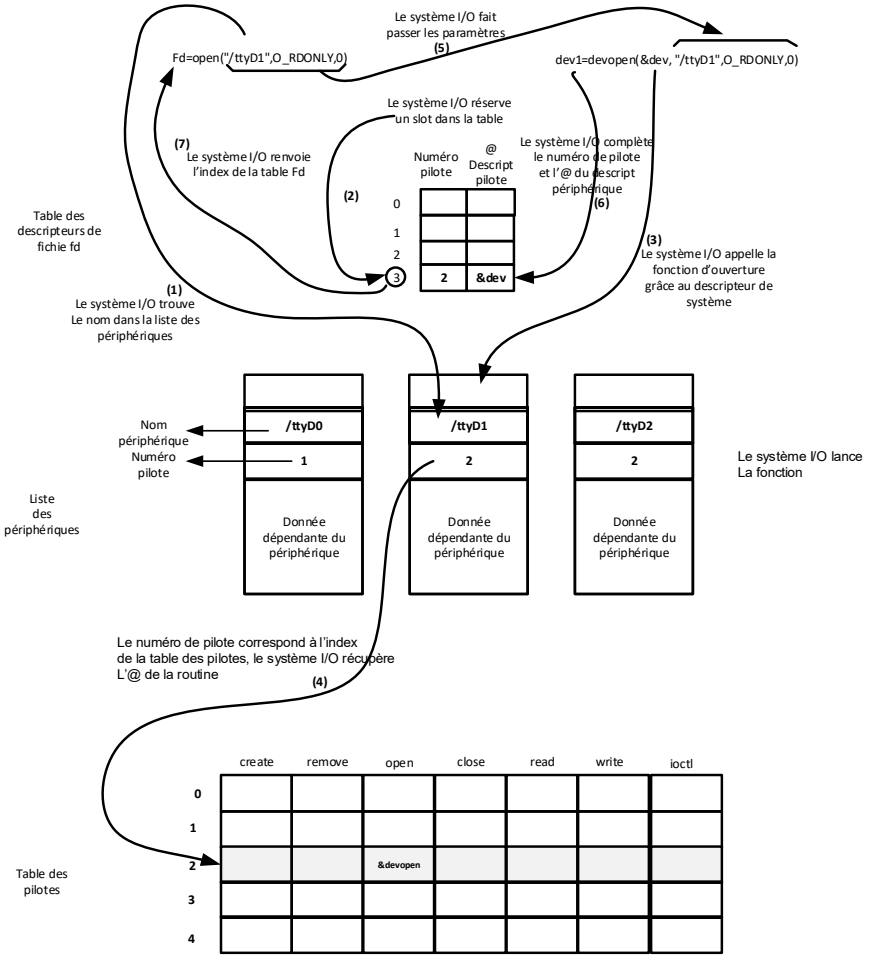
```
STATUS iosDevAdd
    (
    DEV_HDR *    pDevHdr,   /* descripteur de périphérique                */
    const char * name,      /* nom du système d'entrée sorties */
    int          drvnum     /* numéro du pilote*/
                            /* retourné par iosDrvInstall() */
    )
```



```
-> iosFdShow
 fd name                                  drv
  3 /ttyS0                                  3 in out err
  4 (anonpipe)                             10
  5 (socket)                               21
  6 (socket)                               21
  7 (socket)                               21
  8 (socket)                               21
  9 (socket)                               21
 10 (socket)                               21
 11 (socket)                               21
 12 (socket)                               21
 13 (socket)                               21
 14 (socket)                               21
 15 (socket)                               21
 16 /stdio_pty_0xffff800000263b60.M         17
 17 (socket)                               21
 18 (socket)                               21
 19 /stdio_pty_0xffff800000263b60.S         16
value = 50 = 0x32 = '2' = buffer + 0x2
->
```

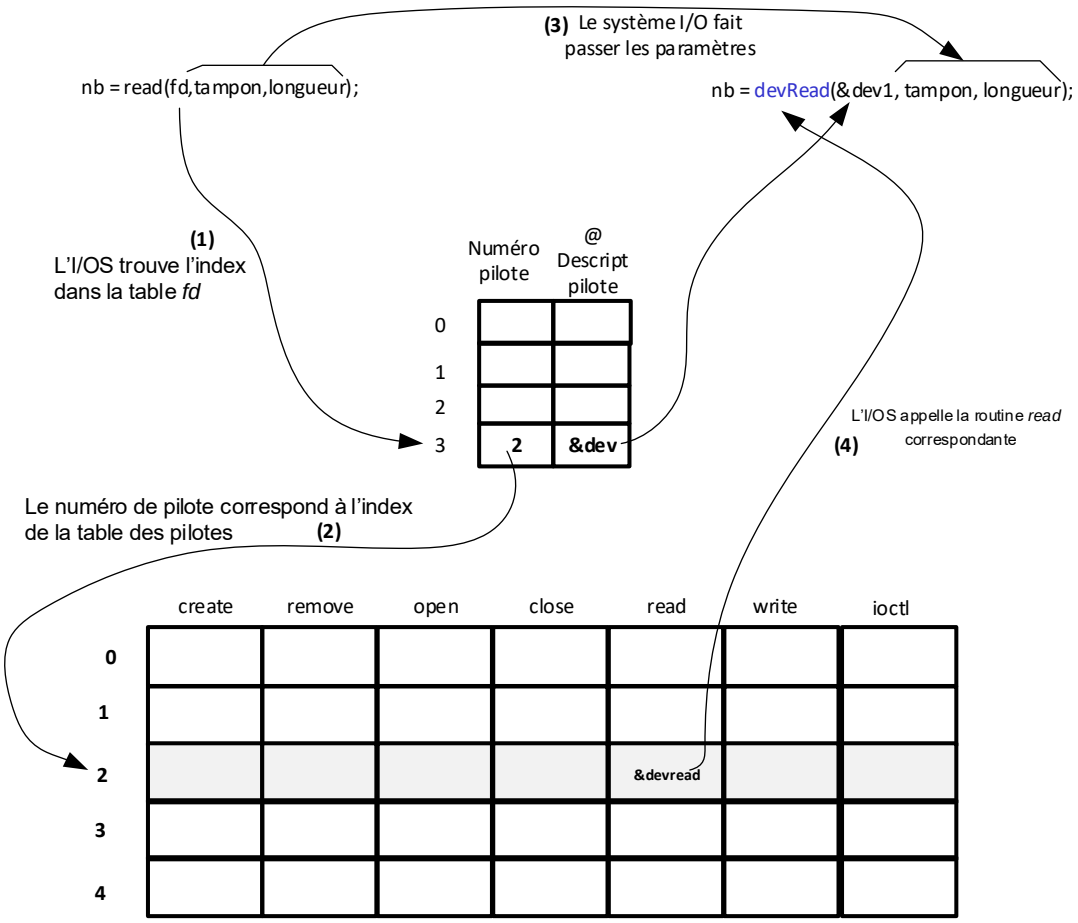
10.3) ouverture d'un périphérique d'entrées sorties

Le mécanisme d'ouverture d'un périphérique est décrit en 7 étapes



10.4) lecture d'un périphérique d'entrées sorties

Le mécanisme de lecture d'un périphérique est décrit en 4 étapes



Désinstallation et retrait d'un périphérique

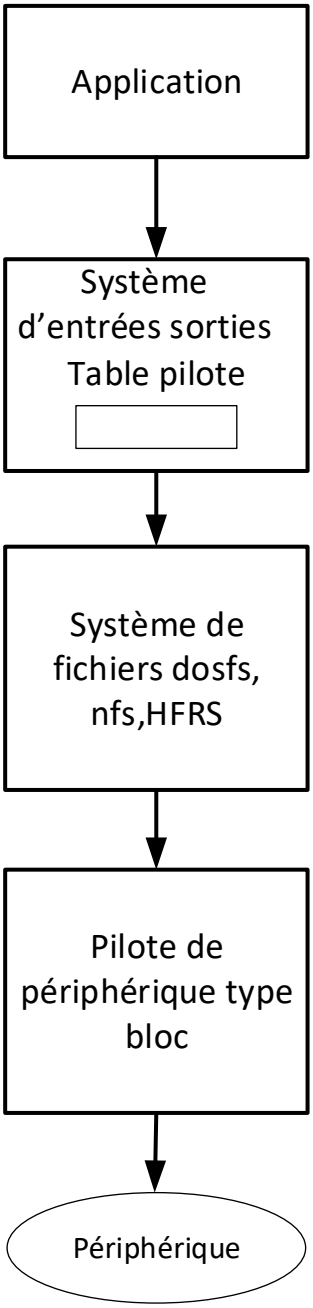
Int ret = iosDrvRemove (int drvNumber,int protOpen)

ret= FALSE : pilote non désinstallé si un périphérique est ouvert
ret=TRUE : un périphérique ouvert est fermé et détruit implicitement, le piloté est désinstallé

retrait

status = iosDevDelete(DEV_HDR * pDevHdr);

les pilotes de périphériques en bloc interagissent avec un système de fichiers



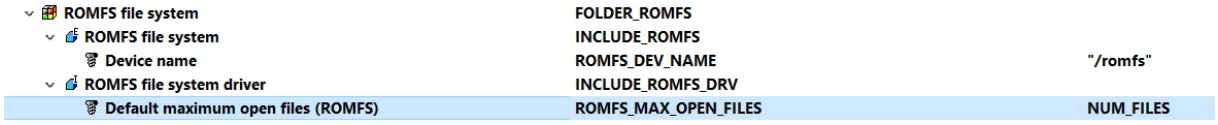
Création d'un environnement autonome sur SDCARD

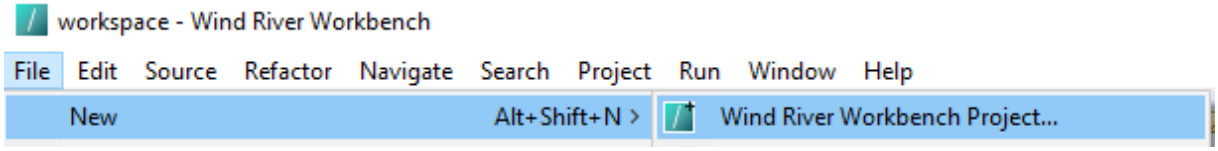
11.1) Présentation

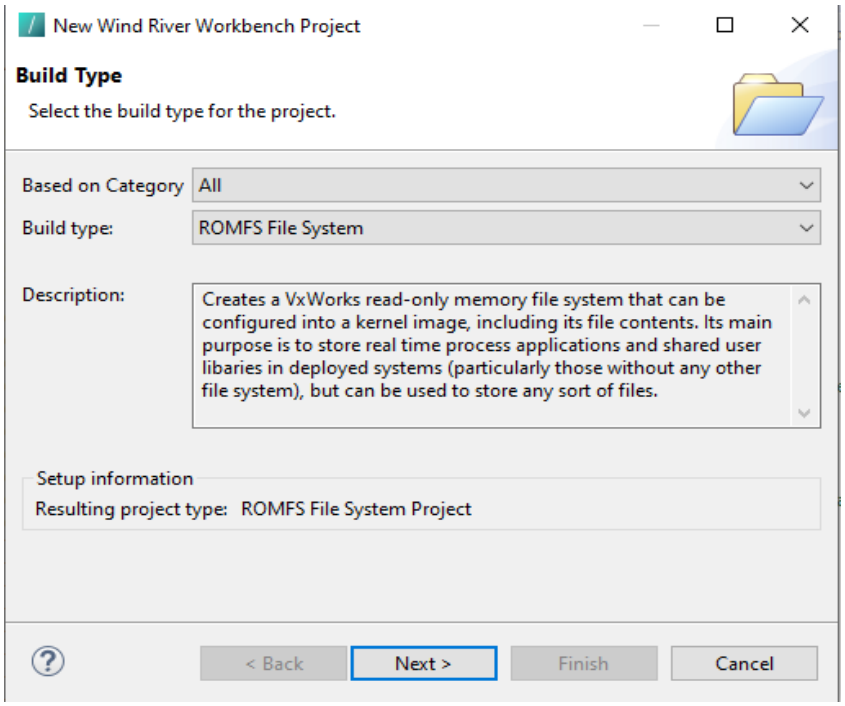
Il est intéressant dans un environnement embarqué de disposer d'une carte qui soit tout à fait autonome sans à proximité un PC. Une carte capable de lancer des applicatifs en mode standalone. Plusieurs méthodes existent sous Vxworks lors de la création du noyau

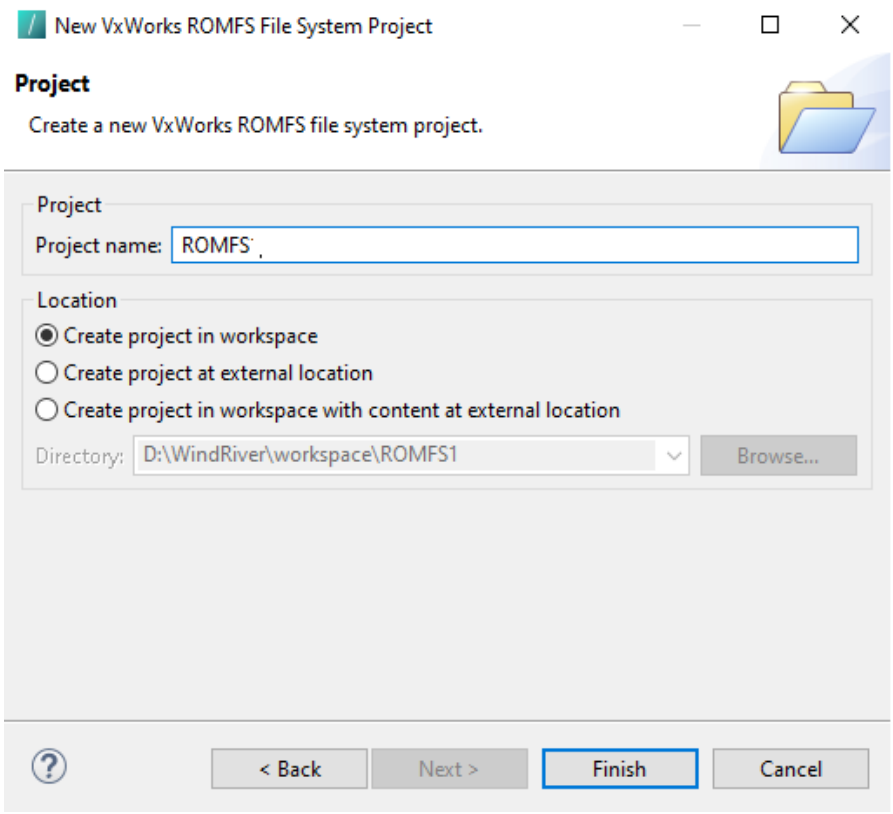
11.2) La ROMFS

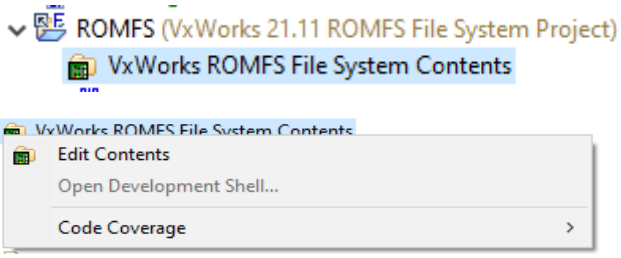
Un projet de système de fichiers ROMFS offre la possibilité de regrouper des applications
et des fichiers VxWorks avec le système d'exploitation sans le besoin d'un disque local ou d'une connexion réseau à un disque distant. Lorsque VxWorks est configuré avec le système de fichiers ROMFS, les fichiers de tout type peuvent être inclus dans l'image du système d'exploitation en ajoutant vers un répertoire ROMFS sur le système hôte. La construction produit une image système unique qui inclut à la fois VxWorks et les fichiers du répertoire du système de fichiers ROMFS.
Lorsque VxWorks est démarré avec cette image, le système de fichiers ROMFS et les fichiers qu'il contient sont chargés avec le noyau. De plus, RTP les applications peuvent être couplées à une fonction de démarrage automatisé afin qu'elles s'exécutent automatiquement au démarrage. Un système de fichiers ROMFS offre ainsi la possibilité de créer des systèmes entièrement autonomes.



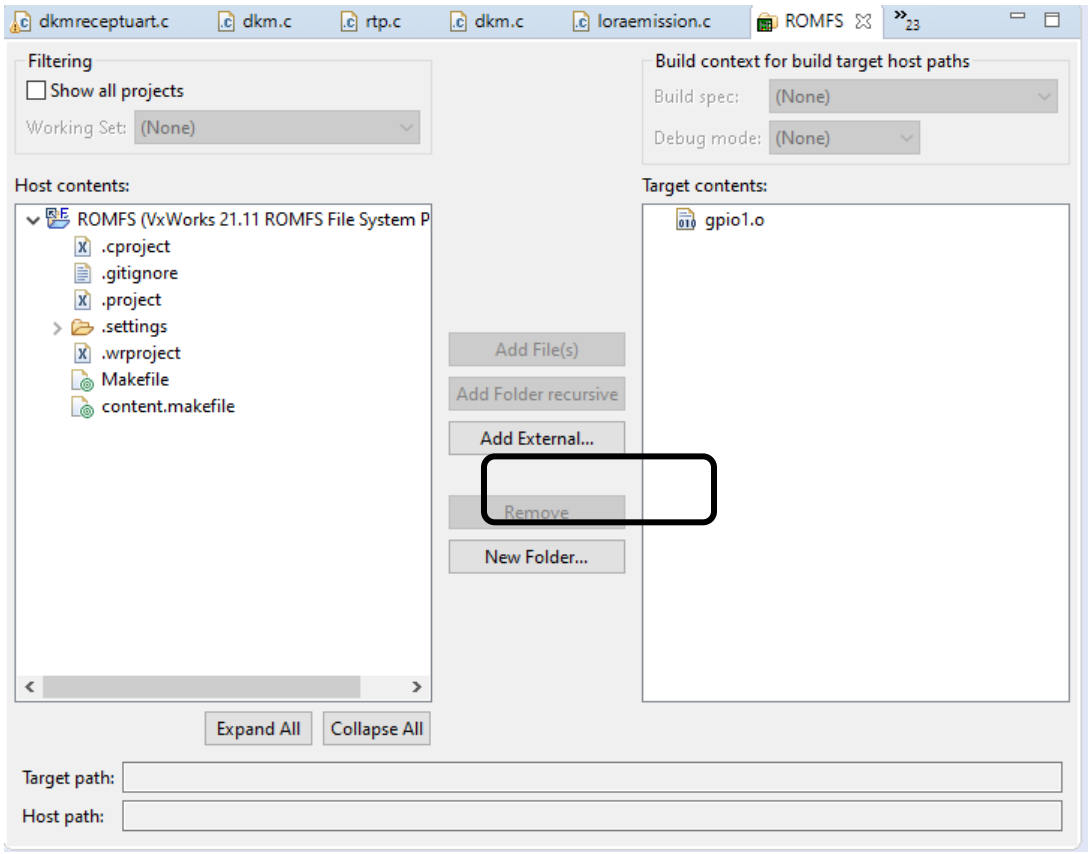





New VxWorks ROMFS File System Project
Project
Create a new VxWorks ROMFS file system project.
Project
Project name: ROMFS
Location
Create project in workspace
Create project at external location
Create project in workspace with content at external location
Directory: D:\WindRiver\workspace\ROMFS1
Browse...
< Back
Next >
Finish
Cancel

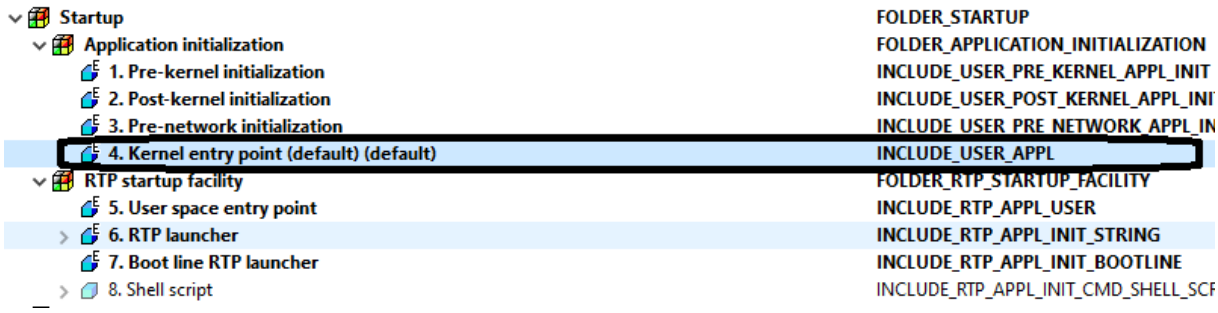

ROMFS (VxWorks 21.11 ROMFS File System Project)
VxWorks ROMFS File System Contents

Edit Contents
Open Development Shell...
Code Coverage

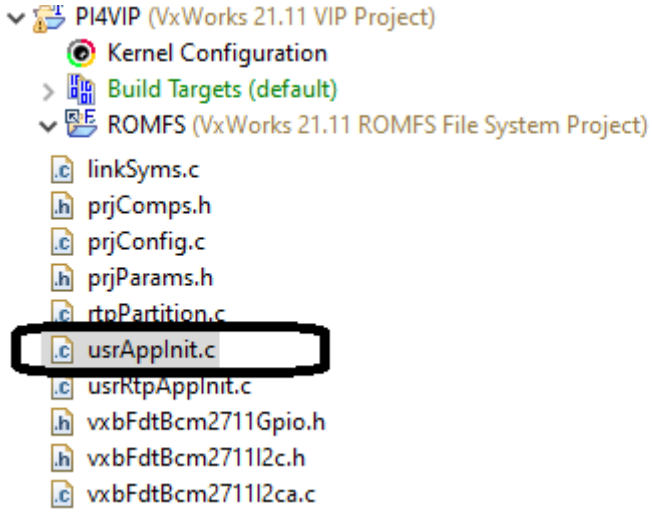


11.3) Lancement d'un DKM au démarrage

Avoir le composant **INCLUDE_USER_APPL** cochez dans le VIP



Additionne un appel à la fonction point d'entrée de l'application **myAppStartup** dans le fichier usrAppInit.c




```
/*******************************************************************************
*
* usrAppInit - initialize the users application
*/

void usrAppInit (void)
    {
#ifdef USER_APPL_INIT
      USER_APPL_INIT;           /* for backwards compatibility */
#endif

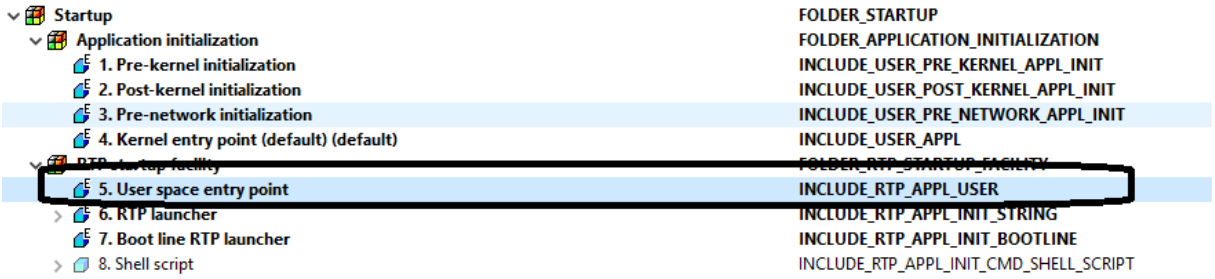
    /* TODO: add application specific code here */
myAppStartup();

    }
```

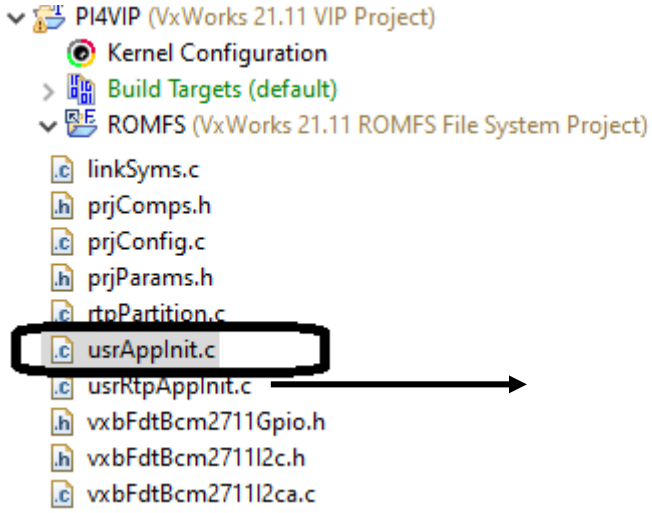
Lancement d'un RTP au démarrage

Vous pouvez utiliser la fonction de démarrage de l'application VxWorks conjointement avec la fonction d'initialisation usrRtpAppInit( ) pour démarrer les applications automatiquement au démarrage de VxWorks.

Pour utiliser cette méthode, VxWorks doit être configuré avec le composant INCLUDE_RTP_APPL_USER, qui inclut le fichier dans votre projet automatiquement.



Pour chaque application que vous souhaitez démarrer, ajoutez un appel rtpSpawn( ) et le code associé au stub de la fonction usrRtpAppInit( ).



L'exemple suivant démarre une application appelée myVxApp avec trois arguments :

```
void usrRtpAppInit (void)
{
char * vxeName = "c:/tmp/myVxApp.vxe";
char * argv[5];
RTP_ID rtpId = NULL;
/* set the application's arguments */
argv[0] = vxeName;
argv[1] = "first";
argv[2] = "second";
argv[3] = "third";
argv[4] = NULL;
/* Spawn the RTP. No environment variables are passed */
if ((rtpId = rtpSpawn (vxeName, argv, NULL, 220, 0x10000, 0, 0)) == NULL)
{
printErr ("Impossible to start myVxApp application (errno = %#x)", errno);
}
}
```

Dans l'exemple ci-dessus, l'exécutable de l'application myVxApp.vxe est stocké sur le système hôte dans c:\tmp.

Vous pouvez également stocker l'exécutable dans ROMFS sur le système cible où l'instruction d'affectation identifie l'exécutable comme suit :

```
char * vxeName = "/romfs/myVxApp.vxe";
```

Ils existent d'autres méthodes pour démarrer un processus RTP lors du boot de la carte :

- Des lignes de boot peuvent être inclues dans le noyau
- Un fichier script peut être créé

12) Programmation du GPIO : Les broches d'entrées sorties à usage générale

Les broches d'entrées sorties du GPIO de la Raspberry PI4 vont de GPIO0 à GPIO27 (GPIO0 et GPIO1 sont réservées). Certaines broches ne sont pas utilisables car partagées avec d'autres périphériques (fonctions alternées ALT0 à ALT5).

GPIO	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	PCLK	SPI3_CE0_N	TXD2	SDA6
GPIO1	High	SCL0	SA4	DE	SPI3_MISO	RXD2	SCL6
GPIO2	High	SDA1	SA3	LCD_VSYNC	SPI3_MOSI	CTS2	SDA3
GPIO3	High	SCL1	SA2	LCD_HSYNC	SPI3_SCLK	RTS2	SCL3
GPIO4	High	GPCLK0	SA1	DPI_D0	SPI4_CE0_N	TXD3	SDA3

GPIO	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO5	High	GPCLK1	SA0	DPI_D1	SPI4_MISO	RXD3	SCL3
GPIO6	High	GPCLK2	SOE_N / SE	DPI_D2	SPI4_MOSI	CTS3	SDA4
GPIO7	High	SPI0_CE1_N	SWE_N / SRW_N	DPI_D3	SPI4_SCLK	RTS3	SCL4
GPIO8	High	SPI0_CE0_N	SD0	DPI_D4	BSCSL / CE_N	TXD4	SDA4
GPIO9	Low	SPI0_MISO	SD1	DPI_D5	BSCSL / MISO	RXD4	SCL4
GPIO10	Low	SPI0_MOSI	SD2	DPI_D6	BSCSL SDA / MOSI	CTS4	SDA5
GPIO11	Low	SPI0_SCLK	SD3	DPI_D7	BSCSL SCL / SCLK	RTS4	SCL5
GPIO12	Low	PWM0_0	SD4	DPI_D8	SPI5_CE0_N	TXD5	SDA5
GPIO13	Low	PWM0_1	SD5	DPI_D9	SPI5_MISO	RXD5	SCL5
GPIO14	Low	TXD0	SD6	DPI_D10	SPI5_MOSI	CTS5	TXD1
GPIO15	Low	RXD0	SD7	DPI_D11	SPI5_SCLK	RTS5	RXD1
GPIO16	Low	<reserved>	SD8	DPI_D12	CTS0	SPI1_CE2_N	CTS1
GPIO17	Low	<reserved>	SD9	DPI_D13	RTS0	SPI1_CE1_N	RTS1
GPIO18	Low	PCM_CLK	SD10	DPI_D14	SPI6_CE0_N	SPI1_CE0_N	PWM0_0
GPIO19	Low	PCM_FS	SD11	DPI_D15	SPI6_MISO	SPI1_MISO	PWM0_1
GPIO20	Low	PCM_DIN	SD12	DPI_D16	SPI6_MOSI	SPI1_MOSI	GPCLK0
GPIO21	Low	PCM_DOUT	SD13	DPI_D17	SPI6_SCLK	SPI1_SCLK	GPCLK1
GPIO22	Low	SD0_CLK	SD14	DPI_D18	SD1_CLK	ARM_TRST	SDA6
GPIO23	Low	SD0_CMD	SD15	DPI_D19	SD1_CMD	ARM_RTCK	SCL6
GPIO24	Low	SD0_DAT0	SD16	DPI_D20	SD1_DAT0	ARM_TDO	SPI3_CE1_N
GPIO25	Low	SD0_DAT1	SD17	DPI_D21	SD1_DAT1	ARM_TCK	SPI4_CE1_N
GPIO26	Low	SD0_DAT2	<reserved>	DPI_D22	SD1_DAT2	ARM_TDI	SPI5_CE1_N
GPIO27	Low	SD0_DAT3	<reserved>	DPI_D23	SD1_DAT3	ARM_TMS	SPI6_CE1_N

Lors de la configuration du noyau et du device tree on peut préciser pour chaque broche du GPIO la fonction alternée retenue.

```
/*
            * Function selection:
            *
            * 0 = GPIO pin is an input
            * 1 = GPIO pin is an output
            * 4 = GPIO pin takes alternate function 0
            * 5 = GPIO pin takes alternate function 1
            * 6 = GPIO pin takes alternate function 2
            * 7 = GPIO pin takes alternate function 3
            * 3 = GPIO pin takes alternate function 4
            * 2 = GPIO pin takes alternate function 5
            *
            * Pin Mode selection:
            *
            * 0 = float
            * 1 = pull-up
            * 2 = pull-down
            *
            */


gpio_pins0: gpioPins0
               {
               pin-set =  <17 0 1      /* gpio17 = input, pull up */
                          27 1 0>;     /* gpio27 = output*/

                  };

gpio: gpio@fe200000
            {
            compatible = "brcm,bcm2711-gpio";
          gpio-controller;
            #gpio-cells = <4>;
            reg = <0x0 0xfe200000 0x0 0x00f4>;
            interrupts = <145 0 1
                          146 0 1>;
            interrupt-parent = <&intc>;
            pinmux-0 = <&gpio_pins0>;
            status = "okay";
            };
```

La programmation des broches en langage C

```
#include "../gpioLib/gpioLib.h"
```

Assignation d'une broche d'entrées sorties

void pinMode(int numerobroche, int type)

int type : IN ou OUT

void vxbGpioSetDir(int numerobroche, int type);

int type : IN=0 ou OUT=1

Ecriture d'un niveau logique sur une broche entrées-sorties

void gpiowrite(int numerobroche, int valeur)

int valeur : HIGH ou LOW

Lecture d'un niveau logique sur une broche entrées-sorties

int gpioread(int numerobroche, int valeur)

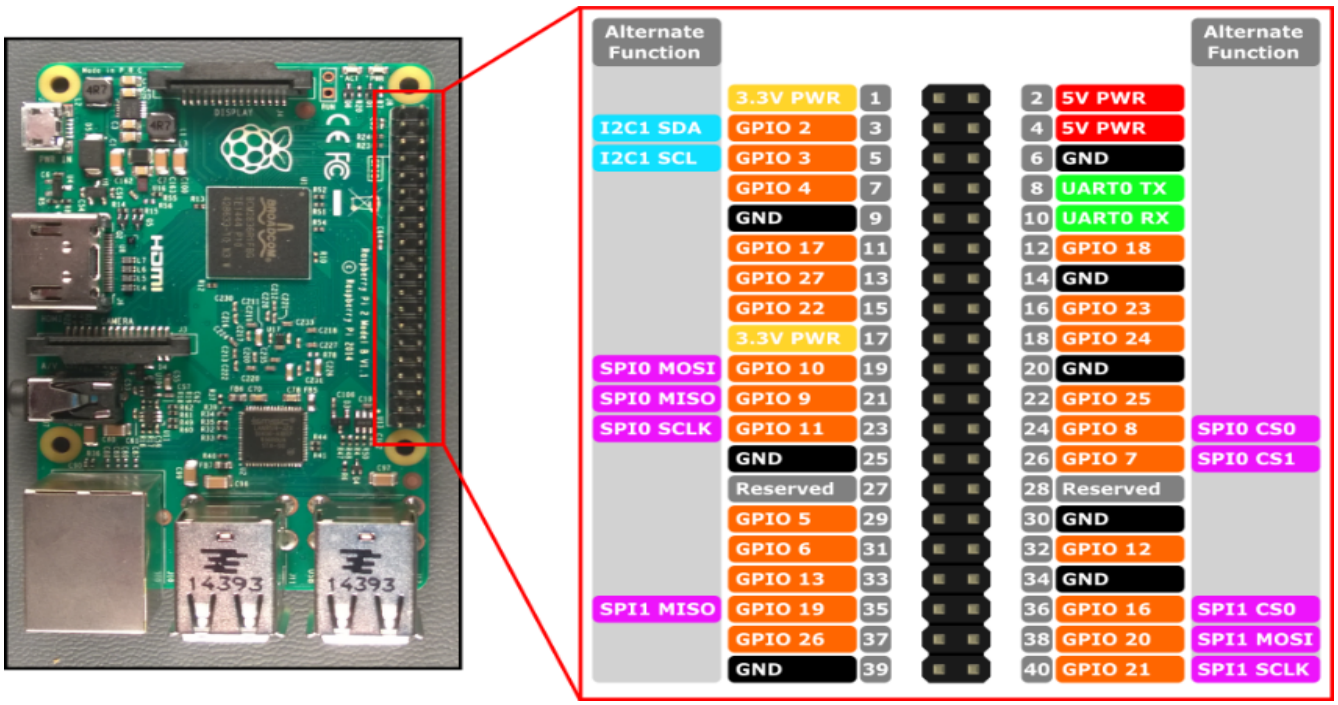
valeur de retour 0 ou 1 suivant le niveau logique de la broche

Libération d'une broche d'entrée sortie

void gpiofree(int numerobroche)

Le numéro de broche correspond au chiffre qui suit le GPIO

Exemple GPIO17 nous donne numerobroche=17



La communication et synchronisation entre taches et processus

Les fonctions intertâches et interprocessus de VxWorks fournissent des mécanismes de synchronisation de l'activité de différentes tâches ainsi que de communication entre elles. Pour la communication interprocessus et noyau-processus, les sémaphores VxWorks et les files d'attente de messages, les tubes et les événements (ainsi que les sémaphores et événements POSIX) peuvent être créés en tant qu'objets publics pour assurer la communication intertâche à travers les limites de la mémoire (entre le noyau et les processus, et entre différents processus).

Structures de données partagées (dkm rtp)

Fournir un moyen pour les tâches s'exécutant dans le même espace mémoire (noyau ou RTP) de communiquer en accédant aux structures de données.

Le verrouillage des interruptions(dkm noyau seulement)
Fournir un moyen de désactiver les interruptions, et donc d'empêcher la préemption par les ISR. Le verrouillage d'interruption n'est pas un objectif général mécanisme, et ne doit être utilisé que de manière très restreinte.

Le verrouillage des taches (dkm noyau seulement)
Fournir un moyen de désactiver la préemption par d'autres tâches. Le verrouillage des tâches est un mécanisme dangereux général et ne doit être utilisé que de manière très restreinte.

Les Semaphores (dkm rtp)

Fournir le principal moyen de synchronisation des tâches et d'exclusion mutuelle. Les variantes à usage général et spécialisées sont disponibles. Les sémaphores peuvent être créés en tant qu'objets publics, ce qui permet leur utilisation dans la communication interprocessus.

Les messages de file d'attente du noyau (dkm)

Fournir un mécanisme de niveau supérieur pour la communication directe des messages entre les tâches.
Les files d'attente de messages peuvent être créées en tant qu'objets publics, ce qui permet leur utilisation dans la communication interprocessus

Les tubes (dkm, rtp)

Fournir une interface de messagerie alternative à la fonction de file d'attente de messages. Les tubes fonctionnent via le système d'E/S.

Les événements (dkm)

Fournir un moyen de communication et de synchronisation entre les tâches et d'autres tâches, interrompre les routines de service (ISR) et tâches, sémaphores et tâches, et files d'attente de messages et tâches.

Nom	communication	synchronisation	accès
Structures de données partagées	x	x	DKM RTP

Le verrouillage des interruptions		×	DKM
Le verrouillage des taches		×	DKM
Les Semaphores		×	DKM RTP
Les messages de file d'attente	×		DKM
Les tubes	×		DKM RTP
Les événements		×	DKM

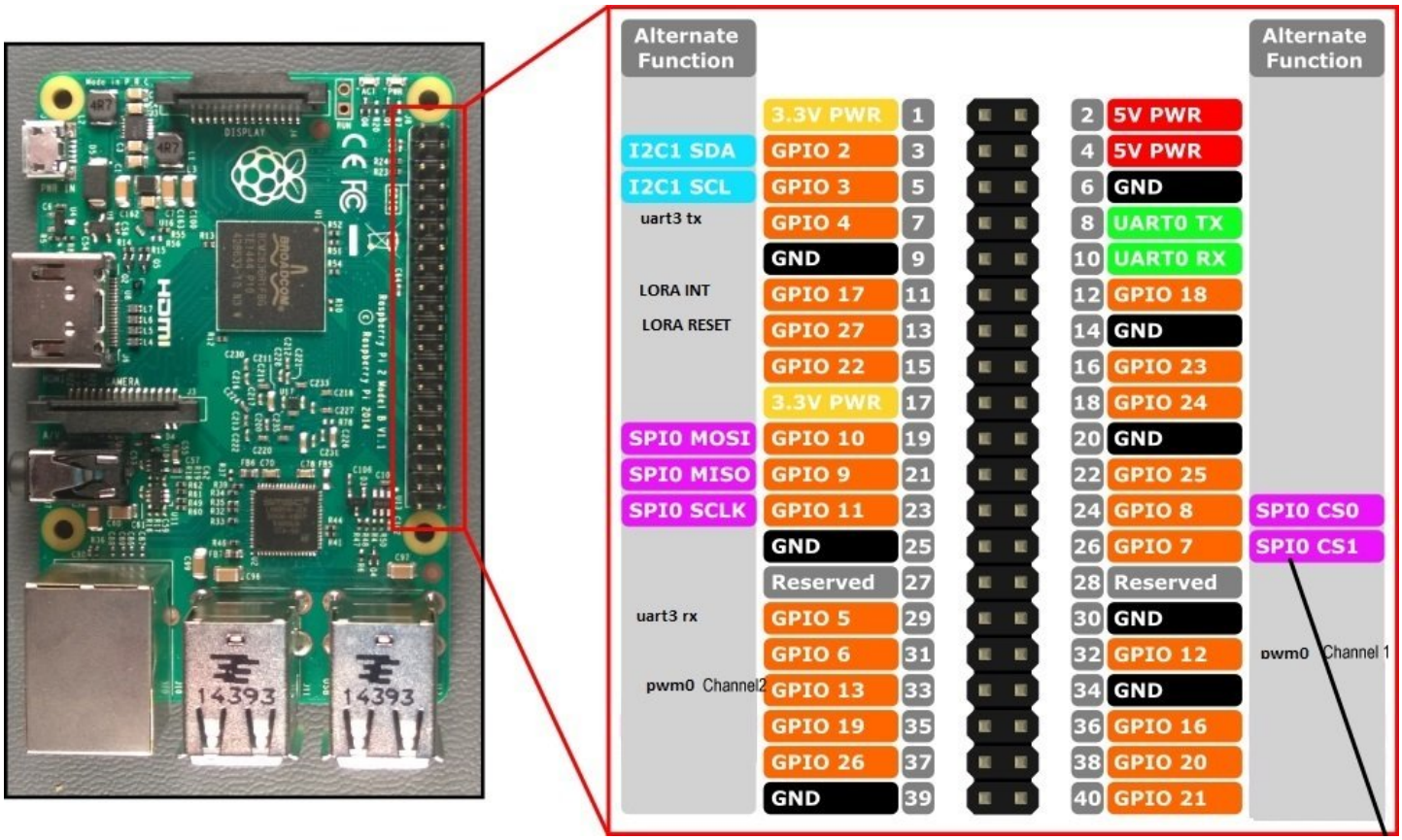
taskLock( )

NAME

***taskLock*( )** - disable task rescheduling

```
STATUS taskLock (void)
```

DESCRIPTION
Cette routine désactive le changement de contexte de tâche. La tâche qui appelle cette routine sera la seule tâche qui est autorisée à s'exécuter, à moins que la tâche n'abandonne explicitement le CPU en se faisant n'est plus prêt. Généralement, cet appel est associé à taskUnlock( ); ensemble, ils entourent un section critique du code. Ces verrous de préemption sont implémentés avec une variable de comptage qui autorise les verrous de préemption imbriqués. La préemption ne sera pas déverrouillée tant que taskUnlock()a été appelé autant de fois que taskLock().
Cette routine ne verrouille pas les interruptions ; utilisez intLock( ) pour verrouiller les interruptions.
Un taskLock() est préférable à intLock() comme moyen d'exclusion mutuelle, car une interruption les verrouillages ajoutent une latence d'interruption au système
.Un semTake() est préférable à taskLock() comme moyen d'exclusion mutuelle
La routine taskLock( ) ne peut pas être appelée à partir des routines de service d'interruption.



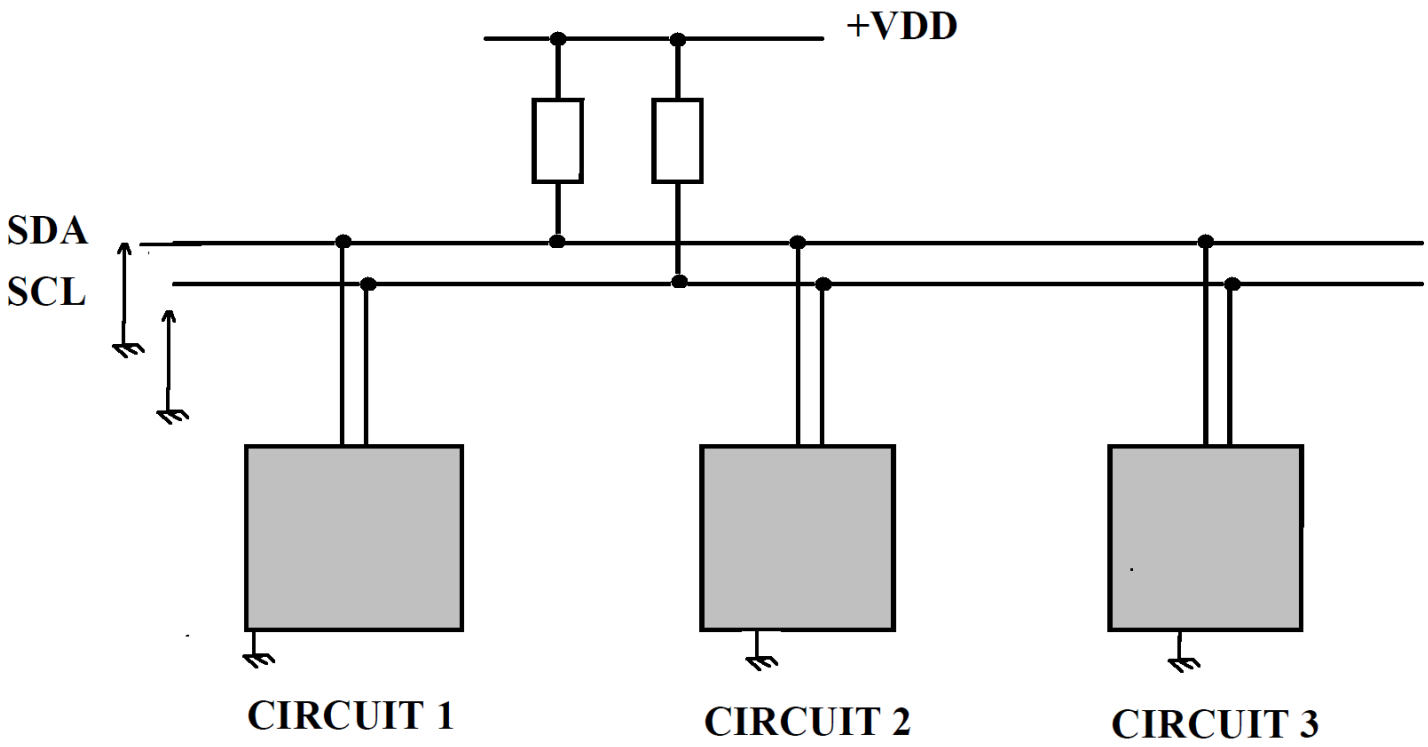
LE BUS I2C FICHE RESUME

Présentation:

I2C est un bus série permettant de transmettre des informations de façon asynchrone entre divers circuits connectés sur le bus.

Le protocole de la liaison est du type MAITRE/ESCLAVE. Chaque circuit est reconnu par son adresse et peut être soit transmetteur soit receveur de l'information.

Deux fils SDA (Serial DAta) et SCL (Serial CLock) véhiculent les informations entre les différents circuits. SDA et SCL sont des lignes bidirectionnelles, connectées à plus VCC par l'intermédiaire de deux résistances de tirage.



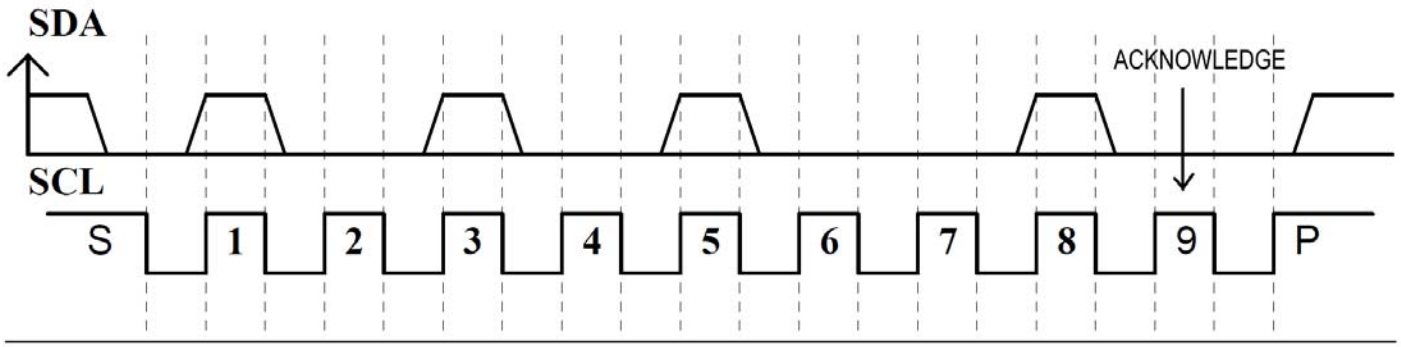
Quand le bus est libre, c'est à dire quand il n'y a pas de transfert de données les deux lignes sont à l'état haut. Pour que le transfert d'un bit soit valide la donnée sur la ligne SDA doit être stable pendant que l'horloge est haute sur SCL.

La valeur de SDA peut uniquement changer quand SCL est à l'état bas.

Un message débute par une condition de START (S) et se termine par une condition de STOP (P).

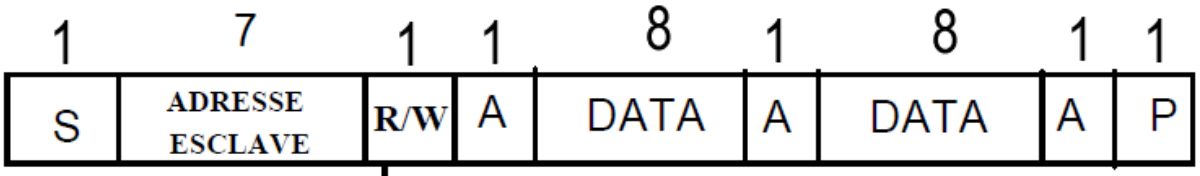
Condition de START: Passage de l'état haut à l'état bas de SDA pendant que SCL est à l'état haut.

Condition de STOP: Passage de l'état bas à l'état haut de SDA pendant que SCL est à l'état haut.



Transfert de données sur le bus: Les données sont transférées par paquets de 8 bits (1 octet). Chaque octet doit être suivi d'un bit d'acknowledge.

Dans le premier octet transféré les sept premiers bits représentent l'adresse de l'esclave et le huitième bit est le bit de read/write

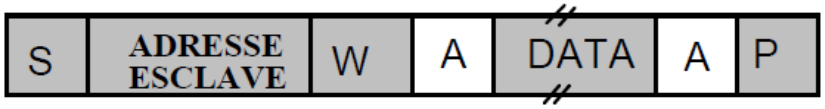


S: START
A: ACK
P: STOP

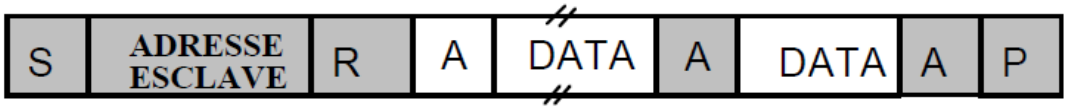
0 les données sont transférées du maître vers l'esclave

1 les données sont transférées de l'esclave vers le maître

Mode maître transmetteur



Mode maître receveur



Du maitre vers l'esclave

De l'esclave vers le maitre

S : Condition de START

P : Condition de stop

Sur le bus I2C, il y aura toujours un et un seul contrôleur maitre et c'est celui-ci qui synchronise les échanges en fournissant les tops d'horloge sur SCL

Aspect Logiciel : utilisation de l'I2C sous Vxworks 7

On va utiliser le pilote de périphérique associé aux entrée sorties. Le descripteur de fichier qui permet l'échange se nomme /I2C/0 (voir la commande devs)

Ouverture :

Dans le domaine noyau (DKM) On va donc ouvrir ce pilote

```
fd = open("/I2C/0", 3, 555))
```

fd =-1 en cas de problème sinon fd=la valeur du descripteur de fichier

configuration du bus I2C

on utilise la fonction IOCTL avec un code fonction associé à la commande

ioctl(int fd, int numerocode, type paramètre) ;

numeroCode	Type paramètre	description
510	int	Programmation de la vitesse du bus en bits/s
511	UINT8	Programmation @ de l'esclave
512	UINT8	Nombre de cycle d'horloge à attendre après le front descendant de SCL avant l'écriture du prochain bit de données
513	UINT8	Nombre de cycle d'horloge à attendre après le front montant de SCL avant la lecture du prochain bit de données

Exemple :

```
ioctl(fd,510,100000);//prog freq 100000hz
```

Lecture ou écriture sur le bus I2C

Ordre d'écriture avec transmission de l' adresse esclave configurée précédemment

```
write(int fd, UINT8 command[] ,int commandLength)

int fd: descripteur de fichier
UINT8 command[]: tableau d'octet à envoyer
int commandLength: nombre d'octet à envoyer

si commandLength=0 l'adresse de l'esclave est envoyée
```

ordre de lecture

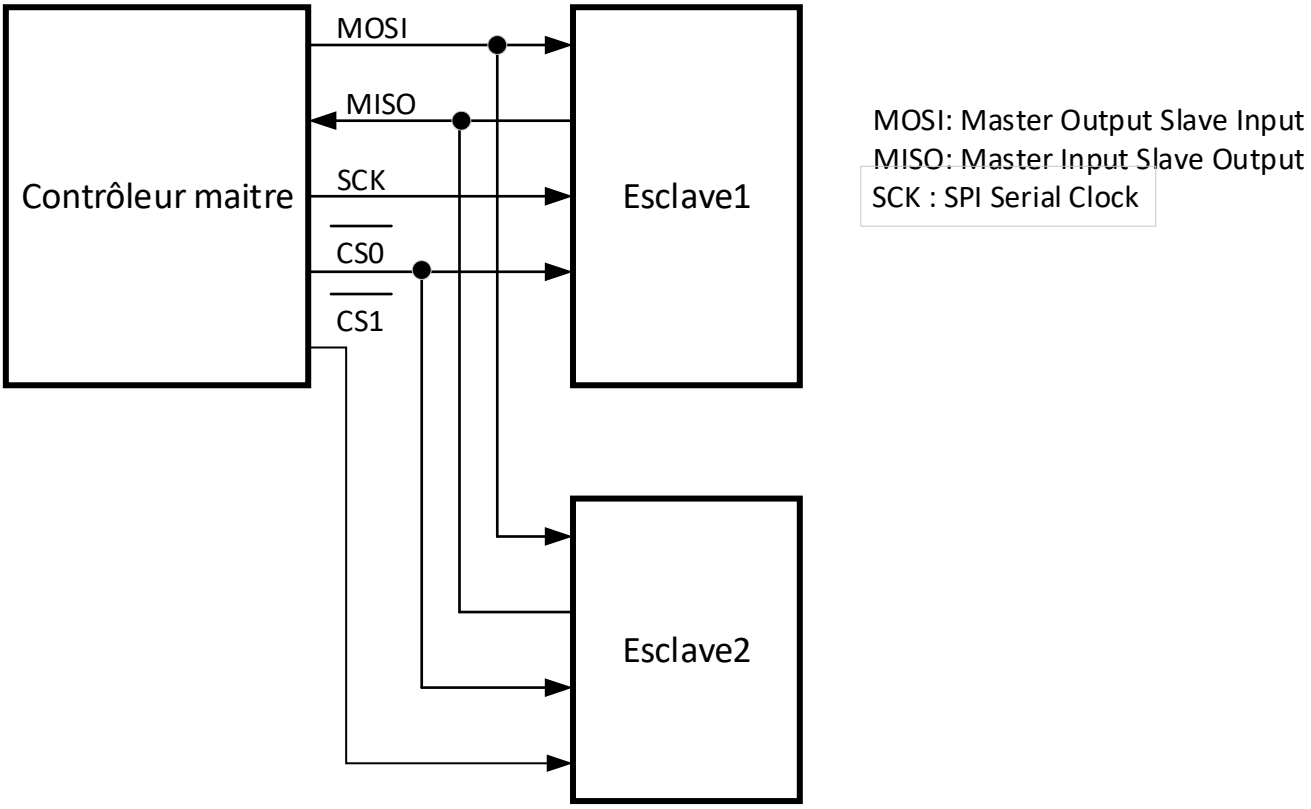
```
read(int fd, UINT8 buffer[] ,int Lengthbuffer)

int fd: descripteur de fichier
```

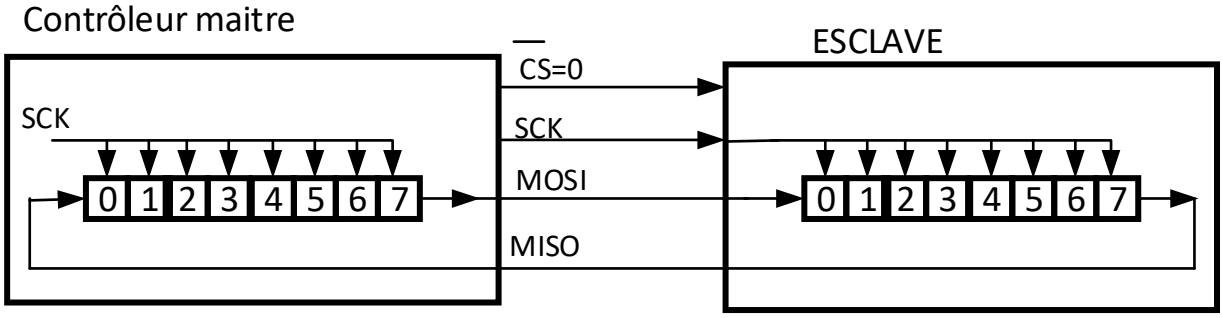
```
UINT8 buffer[]: tableau d'octet ou se loge chaque octet lu
int Lengthbuffer: nombre d'octet à lire
```

LE BUS SPI en mode full duplex : FICHE RESUME

Le bus SPI (Serial Peripheral Interface) est un bus série développé par Motorola. Il est constitué d'un contrôleur maitre et de un ou plusieurs esclaves sélectionné par une ligne chip select CS



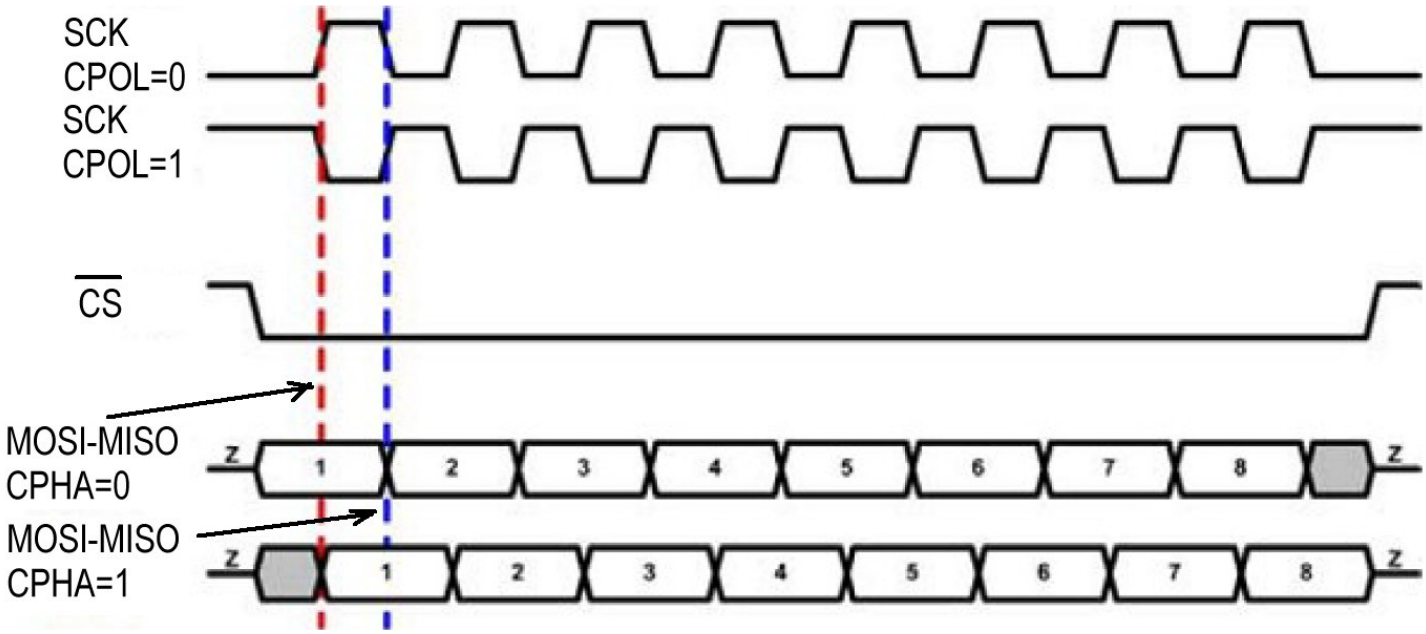
Deux registres a décalages sont cascadés ensemble de façon à ce que lorsque le contrôleur maitre envoie une donnée synchronisé par SCK l'esclave lui transmet en même temps une réponse synchronisée aussi par SCK .



Ils existent quatres modes de transmission :

CPOL=0 L'horloge au repos est à l'état bas

CPOL=1 L'horloge au repos est à l'état haut
CPHA=0 le bit est pris en compte sur un front montant de l'horloge
CPHA=1 le bit est pris en compte sur un front descendant de l'horloge



Aspect Logiciel : utilisation du SPI sous Vxworks 7

On va utiliser le pilote de périphérique associé aux entrée sorties. Le descripteur de fichier qui permet l'échange se nomme /SPI/0 (voir la commande devs)

Ouverture :

Dans le domaine noyau (DKM) On va donc ouvrir ce pilote

```
fd = open("/SPI/0", 3, 555))
```

fd =-1 en cas de problème sinon fd=la valeur du descripteur de fichier

configuration du bus SPI

on utilise un tableau de 11 entier long non signés comme dernier paramètre de la fonction write pour configurer le bus SPI

```
UINT32 mode[11]={0x00};// tableau mode de configuration
```

L'octet 4 case 3 du tableau permet de programmer la vitesse du bus en bits/s

```
mode[3]=50000; //50000hz
```

L'octet 5 case 4 permet de configurer pour chaque bit le bus. Des constantes sont prédéfinies et l'association de celles-ci par des ou ou des et logiques permette de configurer le bus SPI

```
#define SPI_CSPOL_L   0xFB        (bit 2 à 0 logique)
#define SPI_CSPOL_H   0x04      (bit 2 à 1 logique)
#define SPI_CLK_POLH   0x02     (bit 1 à 1 logique)
#define SPI_CLK_POLL   0xFD     (bit 1 à 0 logique)
#define SPI_CLK_PHASEM   0xFE   (bit 0 à 0 logique)
#define SPI_CLK_PHASES   0x01   (bit 0 à 1 logique)
#define FULL_DUPLEX   0x20      (bit 1 à 1 logique)
#define FULL_NODUPLEX   0xDF    (bit 5 à 0 logique)
#define INTERRUPT   0xBF        (bit 6 à 0 logique)
#define POLLING   0x40          (bit 6 à 1 logique)
```

```
SPI_CSPOL_H :CS=1 au repos
SPI_CSPOL_L :CS=0 au repos
SPI_CLK_CPOL_H :CPOL=1 au repos
SPI_CLK_CPOL_L :CPOL=0 au repos
SPI_CLK_PHASEM :CPHA=0
SPI_CLK_PHASES :CPHA=1
FULL_DUPLEX : mode de transmission full duplex
FULL_NODUPLEX   : mode de transmission non full duplex
INTERRUPT : gestion par interruption
```

POLLING : gestion par polling

Exemple d'utilisation

mode[4]=mode[4]|SPI_CSPOL_H|SPI_CLK_POLH | FULL_DUPLEX;
mode[4]=mode[4]& SPI_CLK_PHASEM;
mode[4]=mode[4]&INTERRUPT;

mode[5] =0 // broche CS0 active sur PI4
mode[5] =1 // broche CS1 active sur PI4

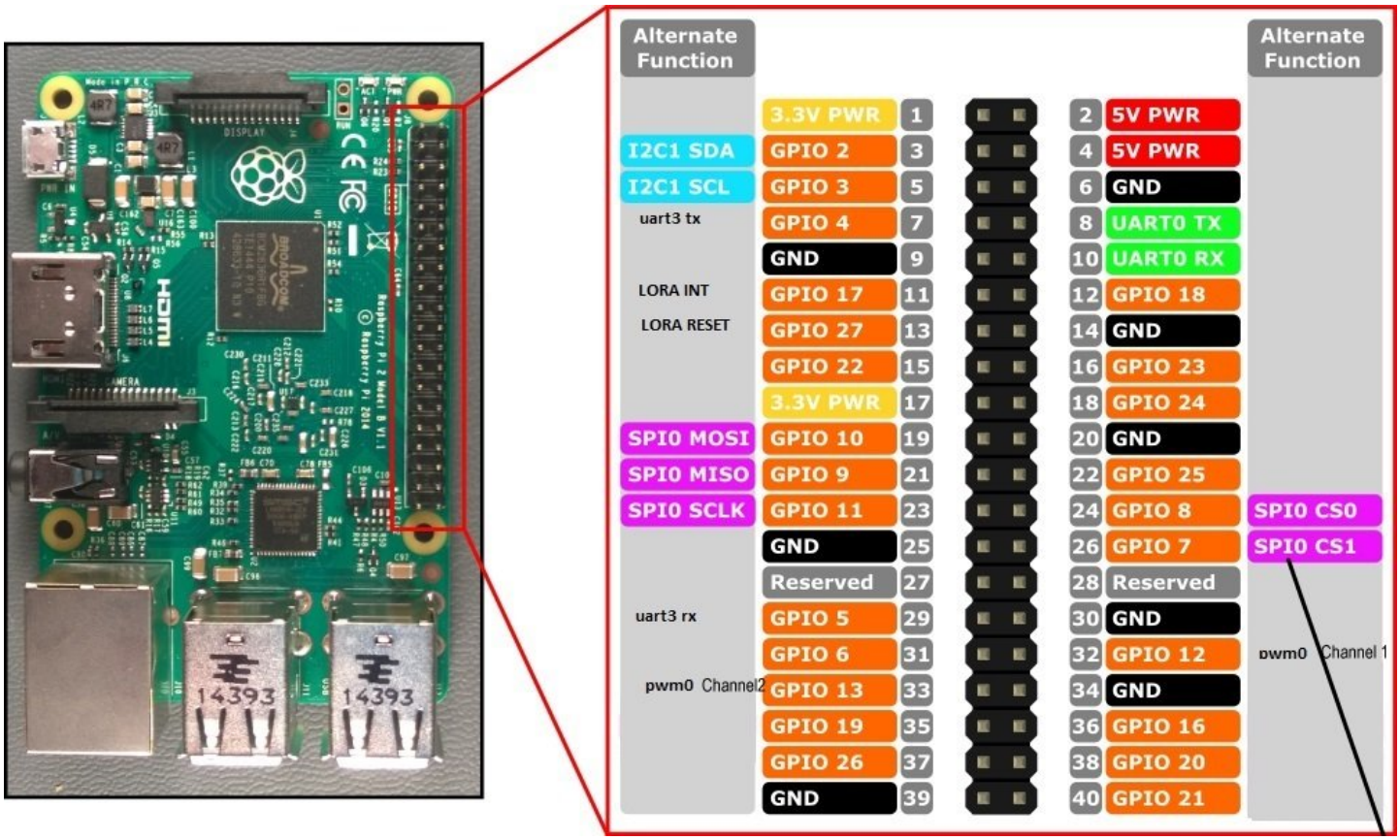
ordre de lecture et d'écriture sur SPI:

write(int fd , uint8_t buffer[], int lenghtbuffer, uint8_t bufrec[],int lenghtbufrec, UINT32 mode[11])

int fd: descripteur de fichier
uint8_t buffer[]: tableau d'octet à envoyer
int lenghtbuffer: nombre d'octet à envoyer
UINT8 bufrec[]: tableau d'octet ou se loge chaque octet lu
int int lenghtbufrec: nombre d'octet à lire
UINT32 mode[11] : tableau mode de configuration

Le contrôleur PWM

La Raspberry PI4 dispose d'un contrôleur PWM a deux canaux (channel1, channel2 voir ci-dessous). Pour disposer de canaux supplémentaires il faudra rajouter un autre contrôleur sur le bus SPI ou I2C



La programmation dans le domaine noyau reste la suivante :

Ouverture :

Dans le domaine noyau (DKM) On va donc ouvrir ce pilote

```
fd = open("/PWM/0", 3, 555))
```

```
fd =-1 en cas de problème sinon fd=la valeur du descripteur de fichier
```

configuration de la PWM

on utilise un tableau de 4 entier long non signés comme dernier paramètre de la fonction write pour configurer le bus SPI

```
// tableau mode de configuration
```

```
UINT32 buffer[4];

       buffer[0]=50;// frequency channel1 hz
       buffer[1]=50;// frequency channel2 hz
       buffer[2]=6;// ratio channel1 %
       buffer[3]=2;// ratio channel2 %

if (write(fd,buffer, 4)) {
              close(fd);
          return false;
         }
```