ENSEIRB-MATMECA

MISE EN ŒUVRE DE LINUX EMBARQUE SUR PROCESSEUR BLACKFIN

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

TABLE DES MATIERES

1 BUT DES TRAVAUX PRATIQUES

Le but de ces Travaux Pratiques est d'étudier la mise en œuvre de Linux embarqué sur une carte d'évaluation Analog Devices Blackfin BF537. Ces TP ont été créés en 2001 initialement sur une carte d'évaluation Motorola ColdFire EVB5407C3. Ils ont été portés sur la carte Blackfin en 2011 afin de pouvoir entre autre travailler avec le *bootloader u-boot*.

La mise en œuvre de Linux embarqué sur la carte Blackfin a fait l'objet d'un sujet de « projets avancés » de l'option Systèmes Embarqués SE. Je tiens ainsi à remercier Damien Espitallier, Nicolas Fruleux, Mathieu Goutel et Nicolas Lesteven de la promotion SE 2010-2011 pour leur travail et leur contribution à l'amélioration constante de l'enseignement de l'option SE…

On verra les points suivants :

- Mise en œuvre de µClinux.
- Mise en œuvre du *bootloader u-boot*.
- Développement d'applications sous µClinux (Hello World, application Client/Serveur).
- Mise en œuvre d'un client SMTP embarqué sous µClinux. Application au pilotage par SMTP de systèmes embarqués.
- Mise en œuvre d'un serveur Web embarqué sous µClinux. Application au pilotage par Web de systèmes embarqués.
- Mise en œuvre du système de fichiers JFFS2 pour les mémoires Flash.
- Debug d'applications µClinux par gdbserver.

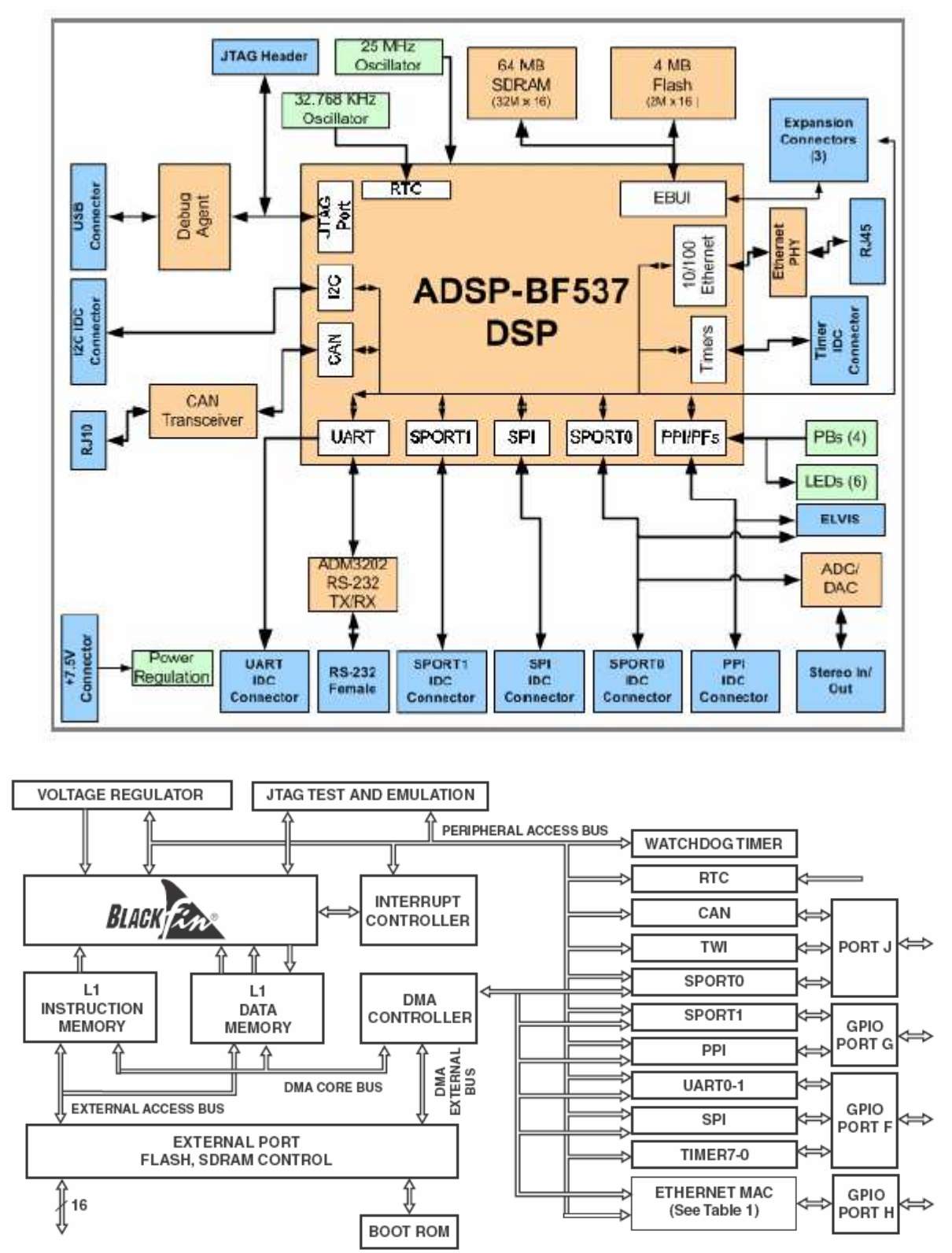
A tout moment, on se référera à l'aide contenue en annexe dans ce manuel ou bien à l'aide en ligne :

```
% man …
```

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

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

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



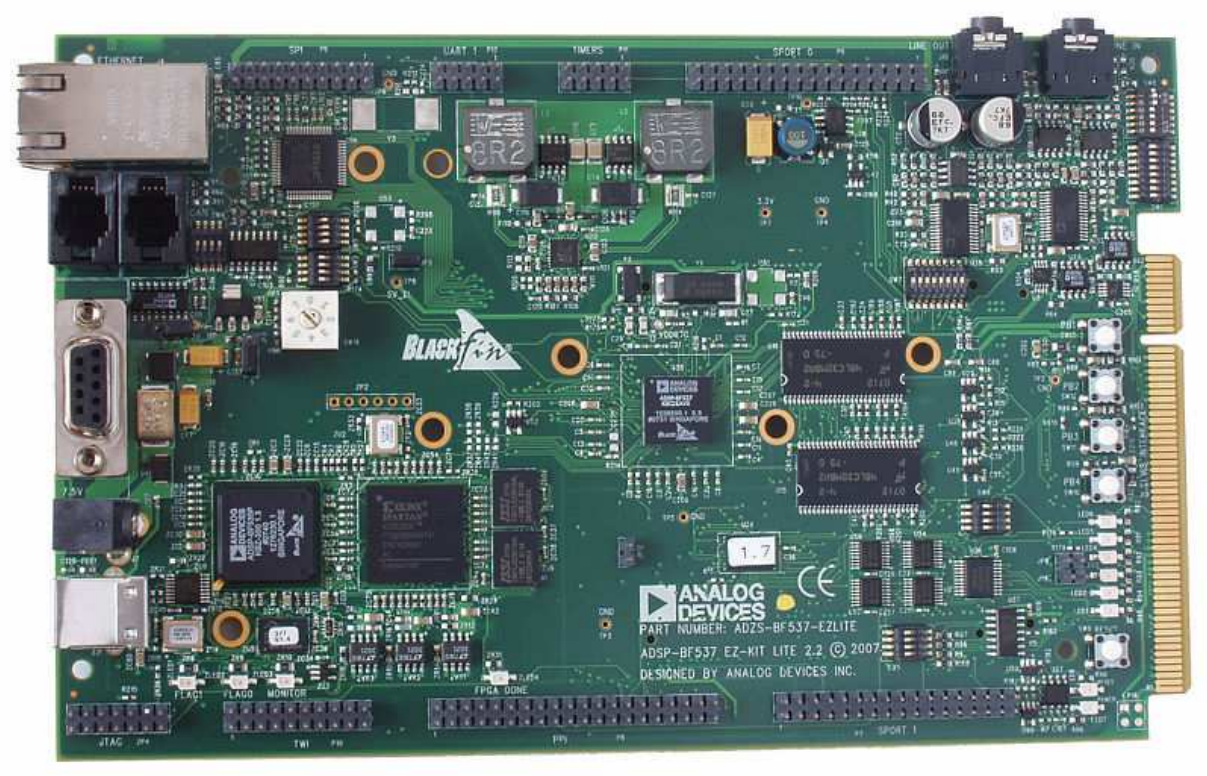
Architecture du processeur Blackfin BF537

La carte cible possède les fonctionnalités suivantes :

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

La carte cible possède donc :

- 4 Mo de mémoire Flash : on y mettra le *bootloader u-boot*, le noyau Linux et le système de fichiers *root*.
- 64 Mo de RAM SDRAM.



Carte cible Blackfin BF537 EZ-KIT Lite

Le mapping mémoire de la carte cible est le suivant :

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

Mapping mémoire de la carte cible Blackfin

On notera que :

- La mémoire RAM va de $0000 0000 à $03FF FFFF.
- La mémoire Flash contenant *u-boot* (et éventuellement le noyau Linux et le système de fichiers *root*) va de $2000 0000 à $203F FFFF.

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

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

Ports et GPIO (pour partie) du processeur Blackfin BF537

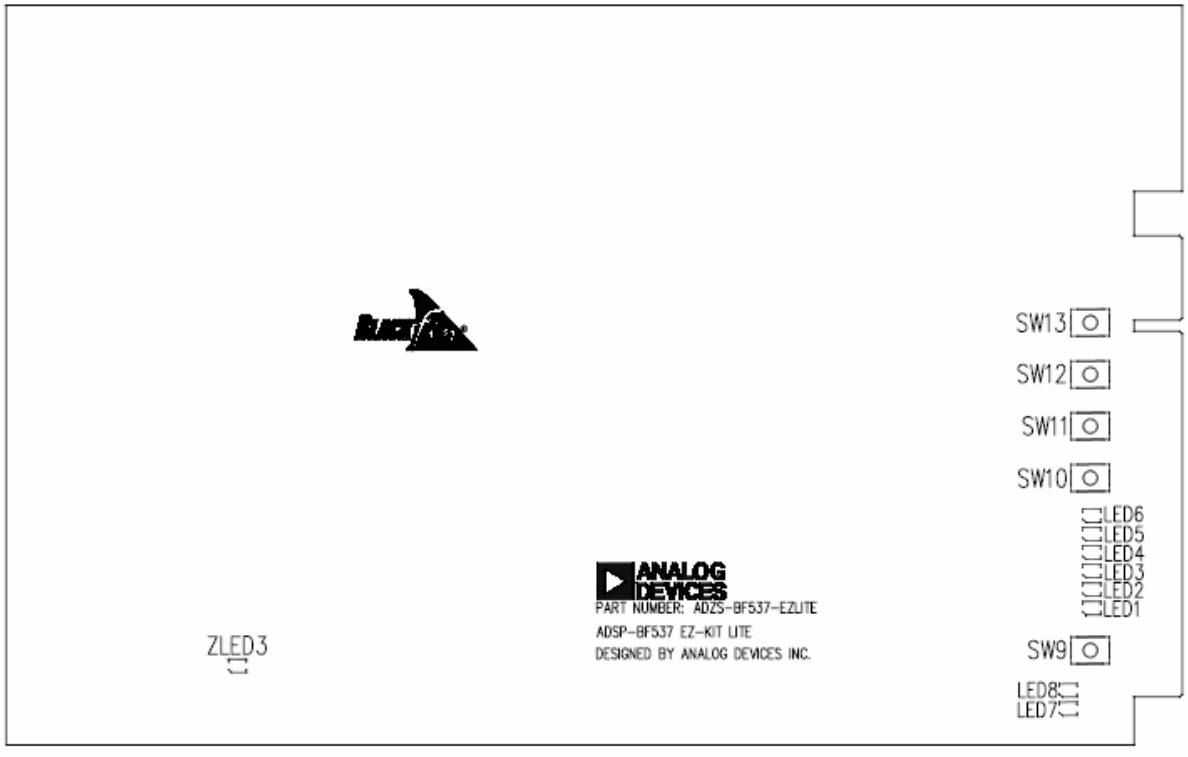
Pour pouvoir faire les TP pilotant les leds de la carte cible, il faut savoir spécifiquement sur quel port elles sont connectées et comment contrôler le port depuis les registres de contrôle et de données du processeur Blackfin.

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

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

6 leds utilisateur de la carte cible

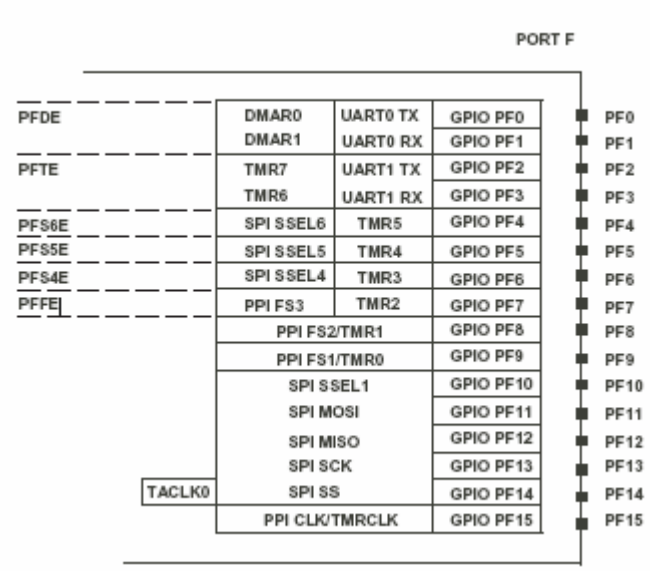
Les 6 leds sont situées sur la carte cible comme suit :



Position des 6 leds utilisateur sur la carte cible

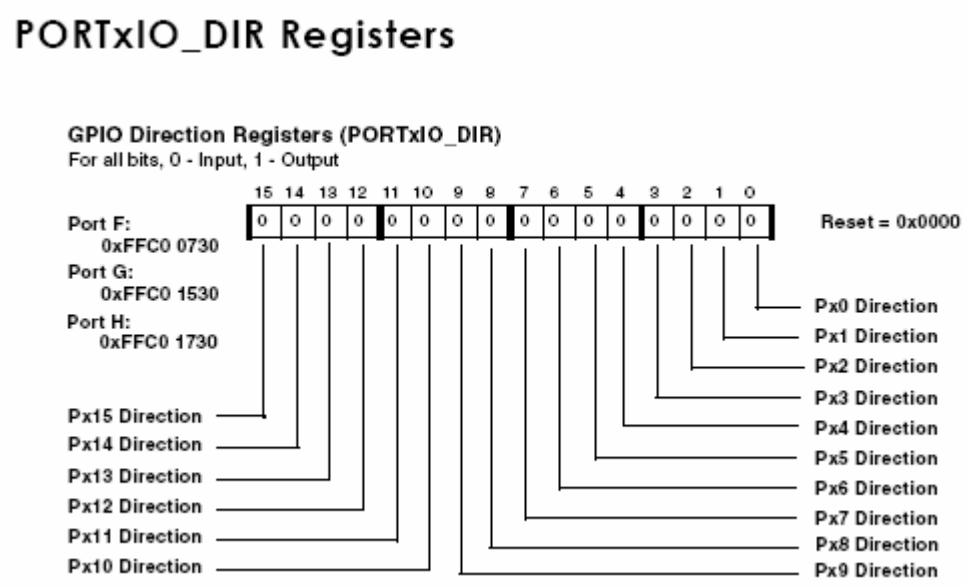
La led est allumée quand on écrit 1 dans le bit correspondant du registre de données du port F.

Le port F est agencé comme suit :



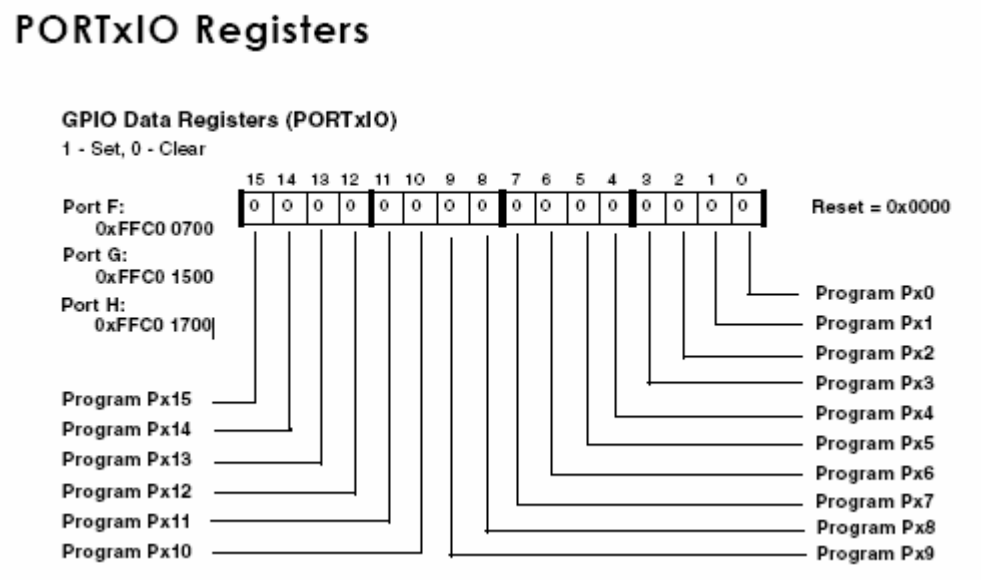
Fonctionnalités du port F du processeur Blackfin BF537

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



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

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



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

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

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

Le processeur Blackfin BF537 n'a pas de MMU mais possède une MPU (*Memory Protection Unit*) qui protège par blocs les accès mémoire. On ne pourra pas piloter directement les leds depuis Linux avec un programme utilisateur. Il faudra au préalable pour cela utiliser un pilote de périphérique ou *driver*.
Le pilote de périphérique exécuté en mode noyau peut accéder directement aux leds car la MPU est court circuitée. On pourra alors utiliser par exemple le code source C suivant :

```
unsigned short *PFDIR = (unsigned short *) 0xffc00730 // Registre de
direction du port F

unsigned short *PFDAT = (unsigned short *) 0xffc00700 //Registre de
données du port F

*PFDIR &= 0x0FC0 ; // PF6 à PF11 en sortie

LED 1      ON    : PF6      = 1        *PFDAT |= 0x0040;
LED 1      OFF   : PF6      = 0        *PFDAT &= 0xFFBF;

LED 2      ON    : PF7      = 1        *PFDAT |= 0x0080;
LED 2      OFF   : PF7      = 0        *PFDAT &= 0xFF7F;


LED 3      ON    : PF8      = 1        *PFDAT |= 0x0100;
LED 3      OFF   : PF8      = 0        *PFDAT &= 0xFEFF;

LED 4      ON    : PF9      = 1        *PFDAT |= 0x0200;
LED 4      OFF   : PF9      = 0        *PFDAT &= 0xFDFF;

LED 5      ON    : PF10     = 1        *PFDAT |= 0x0400;
LED 5      OFF   : PF10     = 0        *PFDAT &= 0xFBFF;

LED 6      ON    : PF11     = 1        *PADAT |= 0x0800;
LED 6      OFF   : PF11     = 0        *PADAT &= 0xF7FF;
```

3 DERNIÈRES MINUTES

Les TP sont maintenant virtualisés avec `VirtualBox`. On ne pourra pas faire les manipulations avec NFS notamment le TP3 qui est laissé à titre documentaire…

4 TP 0 : PRISE EN MAIN

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

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

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

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

- Retrouver à l'aide des informations données en annexe, la syntaxe des commandes du *bootloader u-boot* de la cible pour :
  - Télécharger par le réseau Ethernet via TFTP un fichier binaire `uImage`.
  - Lancer et exécuter le fichier précédemment chargé en mémoire RAM.

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

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

5 TP 1 : TELECHARGEMENT D'UN NOYAU µCLINUX DANS LA CIBLE

- Télécharger par le réseau dans la cible le fichier image `uImage-bf`.
- Lancer le noyau µClinux ainsi téléchargé. Observer les traces sur la console au boot. Quelle est la puissance de la cible en Mips ?
- Sous quel utilisateur est-on connecté par défaut ?
- Quels sont les services µClinux (processus) actifs sur la cible ?
- Configurer l'interface Ethernet (voir en annexe) de la cible et tester la connectivité IP avec la commande `ping` entre la cible et l'hôte.
- Tester l'accès au serveur Web `boa` sur la cible depuis l'hôte avec le navigateur.

6 TP 2 : GENERATION D'UN NOYAU µCLINUX ET TESTS

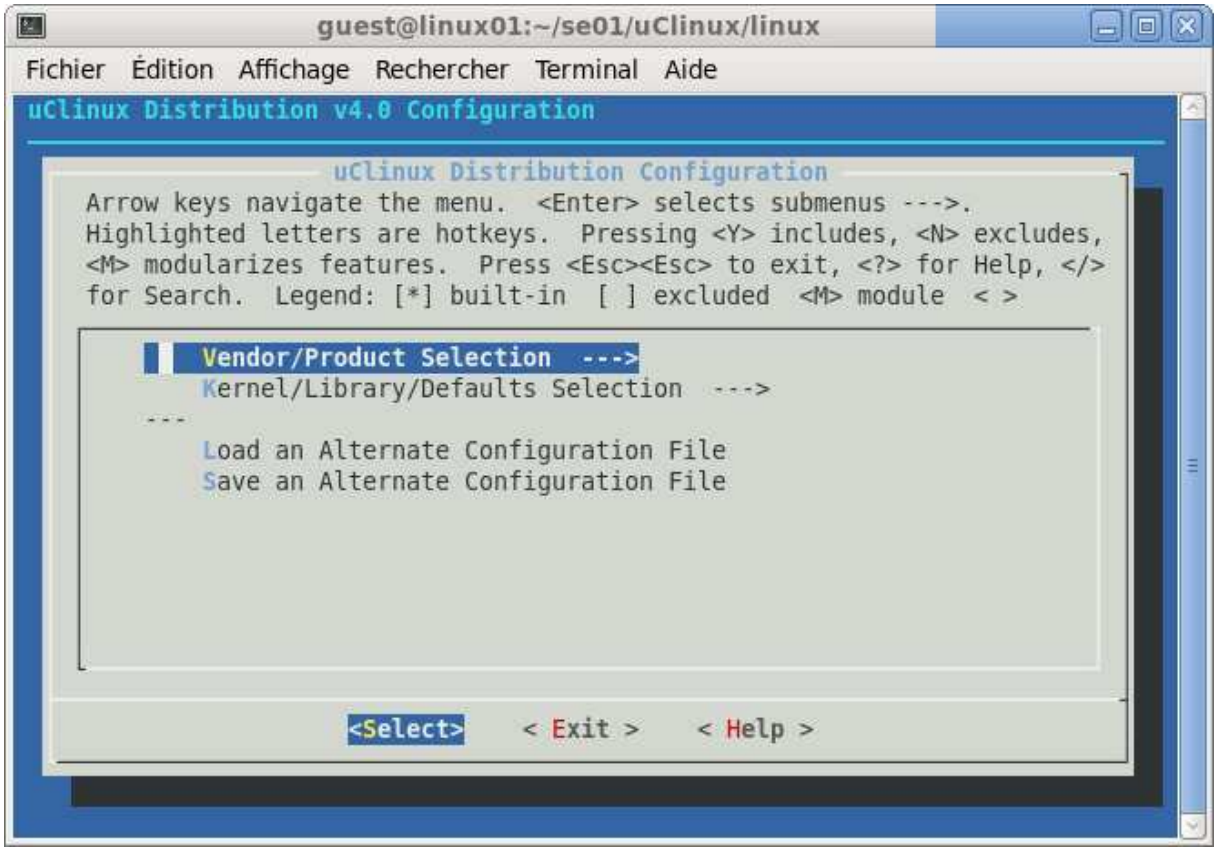
- Dans son répertoire à son nom, recopier le fichier `uClinux.tgz` sous `~kadionik/` :
  `host% cp –r /home/kadionik/uClinux.tgz .`

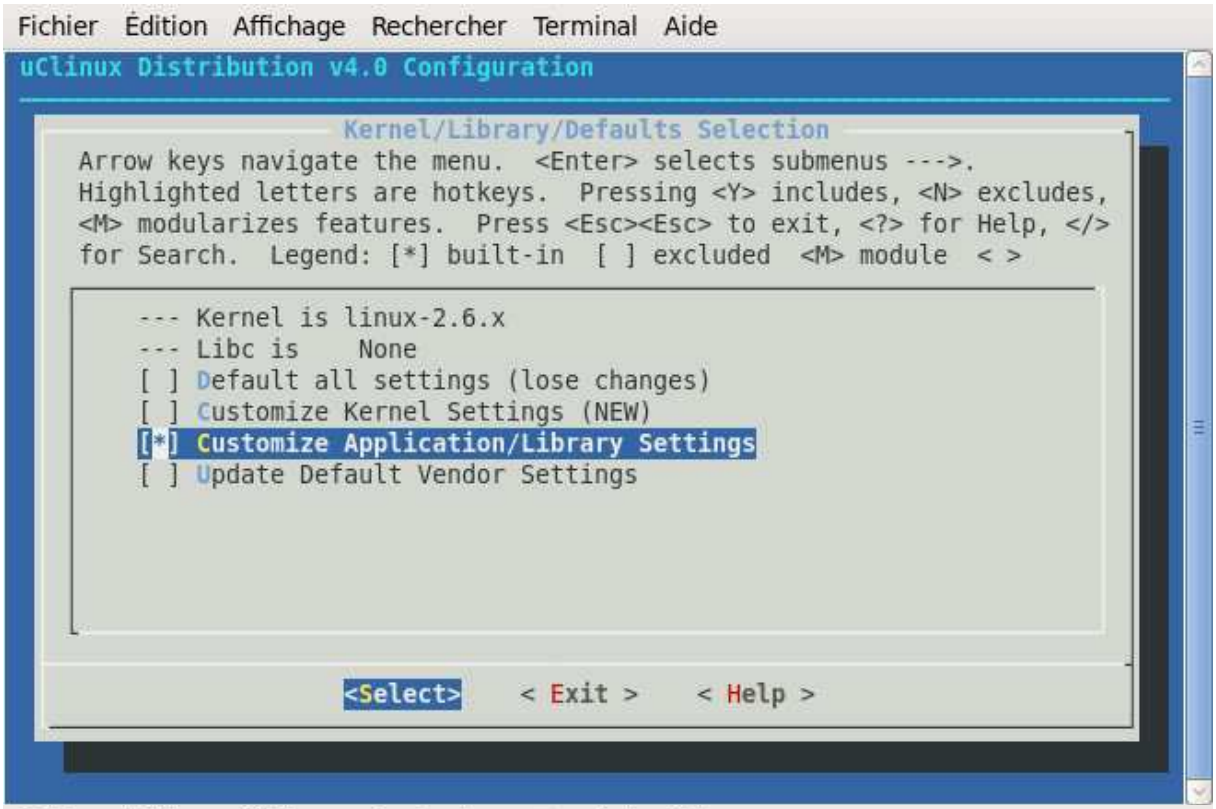
- Décompresser et installer le fichier `uClinux.tgz` :
  `host% tar –xvzf uClinux.tgz`

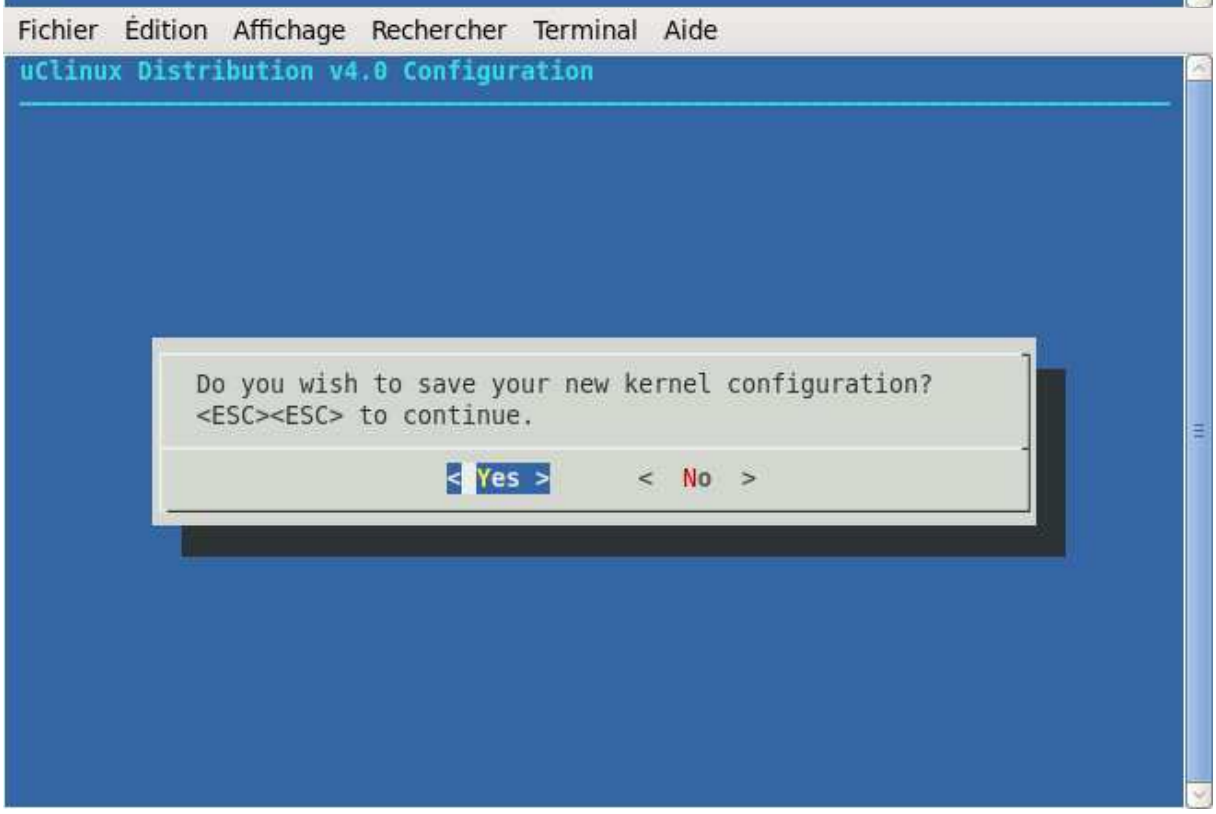
- Se placer ensuite dans le répertoire `linux/` (lien symbolique vers le répertoire `blackfin-linux-dist/`). **L'ensemble du travail sera réalisé à partir de ce répertoire ! Les chemins seront donnés par la suite en relatif par rapport à ce répertoire...**

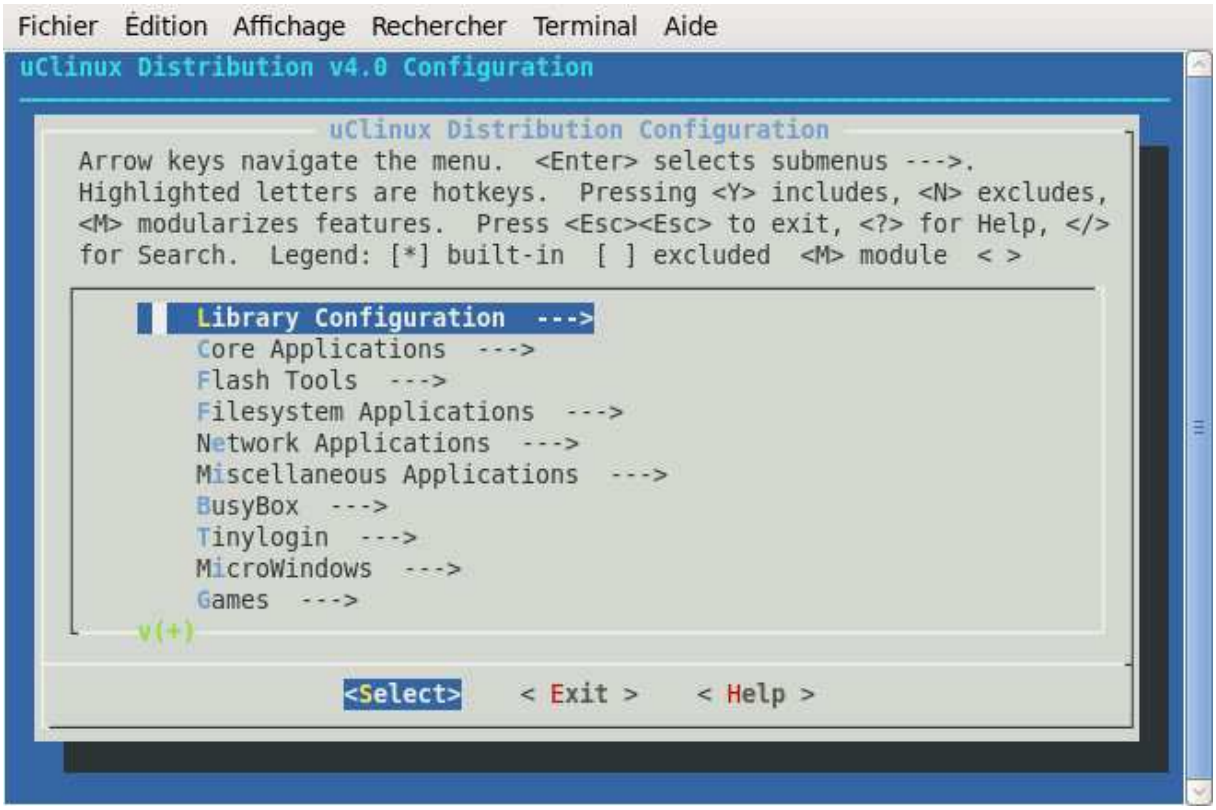
- Le noyau µClinux a été préalablement configuré ainsi que les applications minimales pour le système de fichiers *root*. Configurer les applications µClinux pour la carte Blackfin en utilisant la commande suivante :
  `host% make menuconfig`

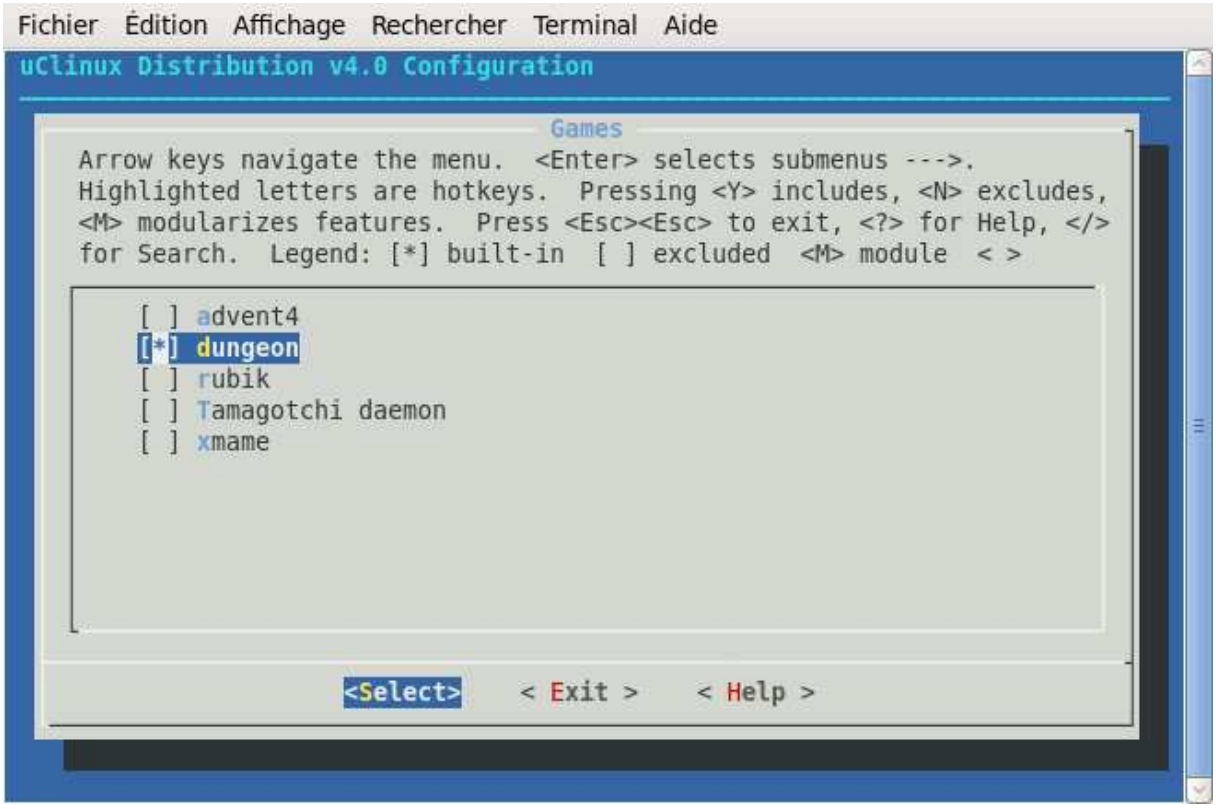
- Dans le menu `Games`, choisir le package `dungeon`. On a alors la succession des menus suivants :

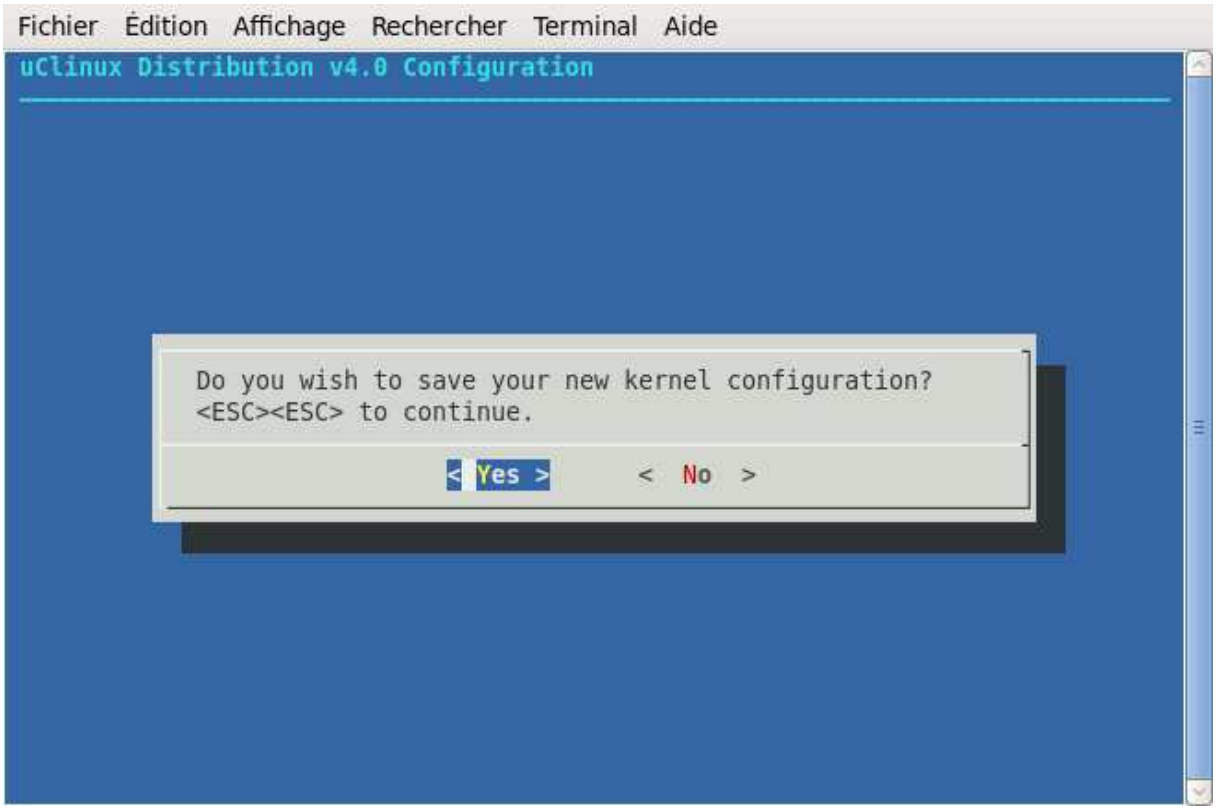



Fichier Édition Affichage Rechercher Terminal Aide
uClinux Distribution v4.0 Configuration
Kernel/Library/Defaults Selection
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >
--- Kernel is linux-2.6.x
--- Libc is None
[ ] Default all settings (lose changes)
[ ] Customize Kernel Settings (NEW)
[*] Customize Application/Library Settings
[ ] Update Default Vendor Settings
<Select> < Exit > < Help >


Fichier Édition Affichage Rechercher Terminal Aide
uClinux Distribution v4.0 Configuration
Do you wish to save your new kernel configuration?
<ESC><ESC> to continue.
< Yes > < No >


Fichier Édition Affichage Rechercher Terminal Aide
uClinux Distribution v4.0 Configuration
uClinux Distribution Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >
Library Configuration --->
Core Applications --->
Flash Tools --->
Filesystem Applications --->
Network Applications --->
Miscellaneous Applications --->
BusyBox --->
Tinylogin --->
MicroWindows --->
Games --->
v(+)
<Select> < Exit > < Help >


Fichier Édition Affichage Rechercher Terminal Aide
uClinux Distribution v4.0 Configuration
Games
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >
[ ] advent4
[*] dungeon
[ ] rubik
[ ] Tamagotchi daemon
[ ] xmame
<Select> < Exit > < Help >



- Compiler pour générer le fichier image `uImage` du noyau µClinux :
  ```
  host% make
  ```
  La première compilation prend une vingtaine de minutes… Il est temps de prendre un café ! Lors des compilations suivantes, le temps de compilation sera de moins de 5 minutes grâce à l'outil `make` qui gère les dates de modification des fichiers. Seuls, les fichiers modifiés seront à nouveau compilés, ce qui sera le cas des applications *userland* que l'on développera par la suite pour les exercices suivants.

- Où se trouve installé le fichier image `uImage` ainsi créé sur l'hôte ?

- Recopier le fichier `uImage` ainsi généré sous `/tftpboot/` pour pouvoir le télécharger depuis la cible avec *u-boot* :
  ```
  host% cp images/uImage /tftpboot
  ```

- Télécharger depuis *u-boot* le fichier `uImage` par le réseau. Lancer le noyau µClinux :
  ```
  bfin> tftp 1000000 uImage
  bfin> bootm 1000000
  ```

- Vérifier que l'application jeu `dungeon` est disponible. Que conclure sur la configurabilité de µClinux ?

Le *bootloader* étant préconfiguré, on pourra par la suite simplement rentrer les commandes *u-boot* suivantes :

```
bfin> tftp uImage
bfin> bootm
```

7 TP 3 : MISE EN ŒUVRE DE NFS SOUS µCLINUX

Passer au TP suivant car impossible sous VirtualBox…

Cet exercice permet de mettre en œuvre NFS côté client sur la cible. On en verra l'intérêt au fil du TP…

- Par analyse du fichier `/etc/exports` sur le PC hôte, retrouver le point de montage NFS depuis la cible.
- Télécharger la nouvelle image `uImage` puis lancer le noyau µClinux. Configurer l'interface réseau de la cible puis effectuer le montage NFS depuis la cible en utilisant comme répertoire de montage `/mnt/` :
  ```
  target# mount -o nolock -t nfs @IP_host:/home/guest /mnt
  ```
- Tester le système de fichiers NFS : création d'un fichier, exécution d'un fichier.
- Quel est l'intérêt d'un montage NFS client sur la cible dans le processus de développement d'une application ?

8 TP 4 : CREATION D'UNE APPLICATION USERLAND SOUS µCLINUX

On désire maintenant développer une application µClinux exécutée par la cible. Il s'agit de la célèbre application « Hello World ! » de K&R.
On suivra le HOWTO `Adding-User-Apps-HOWTO` donné en annexe.

- Création du répertoire `hello` dans le *userland* sous `linux/user/` à partir du répertoire `ledcon` s'y trouvant :
```
host% cd linux/user
host% cp –r ledcon hello
host% cd hello
```

- Modification du fichier `linux/user/hello/Makefile`.

- Modification du fichier `linux/user/Makefile` par ajout de la directive :
```
dir_$(CONFIG_USER_HELLO_HELLO)  += hello
```

- Modification du fichier `linux/user/Kconfig` par ajout de la directive dans le menu `Miscellaneous Applications` :

```
config USER_HELLO_HELLO
bool "hello"
  help
     The Hello World
```

- Création du fichier « Hello World ! » `linux/user/hello/hello.c`.

- On générera le noyau µClinux avec la validation de l'application utilisateur (*userland*) `hello` :
```
host% make menuconfig
```

- Compiler et télécharger le noyau dans la cible puis le lancer.

- Tester la commande locale `hello` sous `/bin/` de la cible.

9 TP 5 : DEVELOPPEMENT D'UN PILOTE DE PERIPHERIQUE POUR L'ACCES AUX LEDS

Le processeur Blackfin présent sur la carte cible ne possède pas de MMU. Cependant, la présence d'une MPU (*Memory Protection Unit*) empêche l'utilisateur d'accéder à certaines adresses mémoire comme celles correspondant aux registres des périphériques. Pour pouvoir piloter les leds, il est nécessaire de concevoir et d'utiliser un pilote de périphérique ou *driver*.

- Copier les fichiers canevas du TP5 :
```
host% cd linux
host% cp –r ../tp/tp5/leds_driver .
host% cd leds_driver
```

- Editer le fichier `leds.c` et apporter les modifications nécessaires pour piloter les leds en fonction d'une valeur significative sur 6 bits et des informations du chapitre II.

- Observer le contenu du fichier `Makefile` et compiler le module.
```
host% make
```

- Quel est le nom du fichier module généré ?

- Il convient maintenant que l'interface réseau soit convenablement configurée pour tous les TP suivants. On se réfèrera à l'annexe 6 pour les valeurs d'adresse IP à utiliser. A titre d'exemple, la configuration réseau de la carte cible `blackfin001` sera :
```
target# ifconfig eth0 10.7.2.118 netmask 255.255.248.0
```

- On pourra télécharger le fichier module en utilisant la commande `tftp` :
```
host% cp leds.ko /tftpboot
target# tftp –g –r leds.ko @IP_host
```

- Insérer le module dans le noyau :
```
target# insmod leds.ko
```

- Créer le point d'entrée dans le système de fichiers `/dev/` :
```
target# mknod /dev/leds c 100 0
```

- Tester le *driver* en écrivant sur `/dev/leds` :
```
target# echo -ne \\00  >/dev/leds
target# echo -ne \\77  >/dev/leds
```

- Modifier l'application `hello` du TP 5 pour réaliser un chenillard en utilisant les appels système classiques `open(), write() et close()` pour interagir avec le driver.

Par la suite, on supposera que le driver est chargé et le point d'entrée `/dev/leds` créé pour faire les TP suivants. Pour ce faire, on pourra copier le module `leds.ko` dans le ROM *File System* et le fichier `leds.ko` sera incorporé dans la génération des prochains fichiers image `uImage` :

```
host% cp leds.ko ../romfs
```

Astuce !

Pour éviter de recompiler à chaque fois le noyau µClinux avec les applications *userland* dont les vôtres et réduire le temps de recompilation, on pourra modifier le fichier `Makefile` de sa propre application *userland* pour faire une compilation croisée en positionnant la variable d'environnement CC :

```
host% cd linux
host% cd user/mon_appli
host% gedit Makefile
```

Rajouter alors au début du fichier `Makefile` :

```
CC=bfin-uclinux-gcc
```

On fera alors une compilation croisée puis on recopiera sous `/tftpboot` l'exécutable ainsi produit :

```
host% make
host% cp mon_appli_exe /tftpboot
```

On pourra alors télécharger l'exécutable en utilisant la commande `tftp` et lancer l'application :

```
target# tftp –g –r mon_appli_exe @IP_host
target# chmod u+x mon_appli_exe
target# ./mon_appli_exe
```

10 TP 6 : DEVELOPPEMENT D'UNE APPLICATION CLIENT/SERVEUR

On désire mettre en œuvre une application Client/Serveur TCP/IP pour le contrôle des leds de la carte cible. Ce TP met ici en valeur la connectivité IP à l'aide d'une interface de contrôle propriétaire (la sienne) d'un système embarqué.

- Créer une application *userland* `myserver` que l'on intègrera sous µClinux. On développera un serveur TCP qui acceptera les requêtes envoyées par un client TCP dans la *socket* de connexion. La requête envoyée par le client comportera 2 octets :
  Octet 1 : numéro de la led à contrôler (entre 1 et 6)
  Octet 2 : valeur de la led (0 : led éteinte, 1 : led allumée)
  Le serveur enverra en retour un octet comme code de retour :
  0 : OK
  1 : ERREUR

```
host% cd linux/user
host% cp –r ../../tp/tp6/myserver .
host% cd myserver
```

- On pourra utiliser le fichier squelette `myserver.c` comme base de travail (structure de données, enchaînement des appels systèmes). On utilisera le driver d'accès aux leds précédent.

- Configurer le noyau µClinux pour utiliser l'application *userland* `myserver` :
```
host% make menuconfig
```

- Compiler le noyau µClinux :
```
host% make
```

- Après compilation, téléchargement et lancement du noyau µClinux dans la cible, on lancera l'application `myserver` sur le numéro de port 2000 :
```
target# myserver 2000
```

- Créer l'application cliente `myclient.c` qui réalisera un chenillard. On pourra utiliser le fichier squelette `myclient.c` comme base de travail (structure de données, enchaînement des appels systèmes).
```
host% cd linux
host% cp –r ../tp/tp6/myclient .
host% cd myclient
```

- Compiler le fichier `myclient.c` et tester le tout :
```
host% myclient nom_cible 2000
```

- Que faut-il penser de cette méthode de connectivité IP ?

11 TP 7 : MISE EN ŒUVRE D'UN SERVEUR WEB EMBARQUE SOUS µCLINUX

Ce TP se propose de mettre en œuvre un serveur Web sous Linux embarqué. C'est généralement cette solution qui est mise en œuvre sur un système embarqué pour assurer une connectivité IP. Le but est de piloter par le Web les 6 leds globalement de la carte cible…

- Configurer le noyau µClinux pour utiliser le package `boa` (fait a priori par défaut) avec le support des scripts CGI génériques (application *userland* `cgi_generic` commun à tous les serveurs Web portés sous µClinux) :
  ```
  host% make menuconfig
  ```
  et :
  ```
  uClinux Distribution Configuration> Network Applications> valider boa
  uClinux Distribution Configuration> Miscellaneous Configuration> valider generic cgi
  ```

- Se placer dans le répertoire `linux/` et recopier le patch `mywwwpatch` pour la configuration du serveur `boa` avec support des CGI et la compilation du CGI (exécutable) `enseirb` pour le pilotage des leds de la cible :
  ```
  host% cd linux
  host% cp –r ../tp/tp9/mywwwpatch .
  ```

- Se placer dans `linux/mywwwpatch` et exécuter le shell script de patch :
  ```
  host% cd mywwwpatch
  host% ./mywwwpatch
  ```

- Le fichier source CGI pour le pilotage des leds de la cible est le fichier C `enseirb.c` situé sous dans `linux/user/cgi_generic`. Modifier ce fichier C pour traiter les commandes CGI envoyées par le navigateur Web. Dans un CGI, les E/S sont redirigées. Il convient simplement d'utiliser des appels systèmes `printf()` pour générer l'entête HTTP et les données de la réponse renvoyée vers le navigateur. Une requête CGI est de la forme `http://@IP_cible:port/cgi-bin/mon_cgi?param1`. Le script CGI a accès à des variables d'environnement CGI et la variable d'environnement `QUERY_STRING` contient le paramètre de la requête envoyée par le navigateur. Modifier le fichier C `enseirb.c` pour contrôler l'allumage (`param=1`) ou l'extinction (`param=0`) des 6 leds **globalement** de la carte cible. On pourra récupérer la valeur courante de la variable d'environnement `QUERY_STRING` grâce à l'appel système `getenv()` :
  ```
  char *param;
  param = getenv("QUERY_STRING");
  ```

- Après compilation, téléchargement et lancement du noyau µClinux dans la cible.

- On lancera le serveur Web `boa` :
  ```
  target# boa –c /etc
  ```

- On vérifiera que le serveur Web boa est bien actif :
  ```
  target# ps
  ```

- A l'aide d'un navigateur Web (*Firefox*), exécuter le script CGI `printenv` en forgeant l'URL adéquate. On observera l'évolution de la variable d'environnement `QUERY_STRING` :
  ```
  http://@IP_cible/cgi-bin/printenv
  http://@IP_cible/cgi-bin/printenv?1
  http://@IP_cible/cgi-bin/printenv?0
  ```

- Contrôler alors l'allumage et l'extinction des 6 leds de la carte cible en forgeant l'URL adéquate.

- Que faut-il penser de cette méthode de connectivité IP ?

12 TP 8 : MISE EN ŒUVRE D'UN CLIENT SMTP SOUS LINUX

On va dans un premier temps tester le serveur de mails SMTP du PC hôte sous Linux et comprendre le dialogue entre un client et un serveur SMTP et constater que :

- Il est du type commande/réponse.
- Il est orienté flux d'octets structurés sous forme de chaînes de caractères ASCII.

- Se connecter par *telnet* au service mail du PC hôte. Les commandes envoyées au serveur de mail (protocole SMTP *Simple Mail Transfer Protocol*, RFC821) sont structurées sous forme de lignes de commandes ASCII. Un exemple d'échanges de commandes SMTP est proposé ci-après (C: commande à taper, R: réponse du serveur SMTP) :

```
R: 220 linux01.localdomain ESMTP Postfix

C: MAIL FROM:<guest@localhost>
R: 250 2.1.0 Ok

C: RCPT TO:<guest@localhost>
R: 250 2.1.5 Ok

C: DATA
R: 354 End data with <CR><LF>.<CR><LF>
un test
.

R: 250 2.0.0 Ok: queued as E3382A50439

C: QUIT
R: 221 2.0.0 Bye
```

- En s'aidant de l'exemple, envoyer manuellement un mail à l'utilisateur `guest` :
  ```
  host% telnet localhost 25
  ```

- Vérifier sa bonne réception en utilisant la commande de lecture de mail `mail` :
  ```
  host% mail
  ```

Pour développer une application cliente SMTP sous Linux, il convient donc de développer une application réseau cliente comme dans le TP précédent en formatant le flux de données échangées sous forme de chaînes de caractères ASCII respectant le protocole SMTP.
Pour cela, on va se simplifier la vie et partir d'un client SMTP existant issu des sources du livre de T. Jones cité en référence. On consultera les fichiers sources donnés en annexe.

- Se placer dans le répertoire `linux/` et recopier le répertoire `mysmtp/` :
  ```
  host% cd linux
  host% cp –r ../tp/tp7/mysmtp .
  host% cd mysmtp
  ```

- Ce répertoire contient un fichier C principal `main.c` et un fichier bibliothèque de fonctions `emstp.c` et son fichier .h associé ainsi qu'un fichier `Makefile`. Par analyse du code source, modifier les fichiers C pour envoyer un mail de format texte et de format HTML à l'utilisateur `guest` du PC hôte.

- Compiler et tester le client SMTP `mysmtp`.

13 TP 9 : MISE EN ŒUVRE D'UN CLIENT SMTP EMBARQUE SOUS µCLINUX

Il convient maintenant de transformer l'exemple précédent en application *userland* µClinux.

- Créer une application *userland* `mysmtp` que l'on intégrera sous µClinux :
```
host% cd linux/user
host% cp –r ../../tp/tp7/mysmtp/ .
host% cd mysmtp
```

- Intégrer l'application `mysmtp` dans la distribution µClinux. Modifier les programmes sources pour que le client SMTP émette un email à destination de l'utilisateur `guest` du PC hôte.

- On générera le noyau µClinux avec la validation de l'application `mysmtp` :
```
host% make menuconfig
```

- Compiler :
```
host% make
```

- Télécharger le fichier `uImage` dans la cible puis le lancer.

- Envoyer manuellement un mail à l'utilisateur `guest` du PC hôte :
```
target# telnet @IP_host 25
```

- Vérifier sa bonne réception en utilisant la commande de lecture de mail `mail` :
```
host% mail
```

- Tester la commande locale `mysmtp` sous `/bin/` de la cible. On ajustera au préalable le fichier `/etc/hosts` du système de fichiers *root* de la cible pour ajouter une entrée correspondant à l'adresse IP du PC hôte (voir annexe 6), par exemple comme ceci :
```
target# echo "@IP_host linux01" >> /etc/hosts
```

- Vérifier sa bonne réception en utilisant la commande de lecture de mail `mail` :
```
host% mail
```

- Que faut-il penser de cette méthode de connectivité IP ?

14 TP 10 : SYSTEME DE FICHIERS JFFS2 POUR MEMOIRE FLASH

Le système de fichiers JFFS2 (*Journaling Flash File System 2*) est un système de fichiers Linux développé spécialement pour être utilisé sur une mémoire de type Flash que l'on retrouve fréquemment dans un système embarqué. Cela peut être un circuit intégré de mémoire Flash, une carte Compact Flash, une clé USB. La particularité d'une mémoire Flash est sa programmation par secteur (512 octets à quelques Ko) qui nécessite d'effacer complètement le secteur pour y programmer ne serait-ce qu'un seul octet. De plus, il convient de supporter les pannes d'alimentation durant la phase de reprogrammation. JFFS2 est prévu pour cela : c'est un système de fichiers journalisé qui stocke toutes les versions d'un fichier modifié, ce qui permet d'accélérer la phase de boot d'un système Linux (pas de `fsck` intempestif) et d'assurer l'intégrité des fichiers stockés. Il intègre un processus de ramasse miettes (*Garbage Collector*) quand le taux d'occupation du système de fichiers est proche de 80-90 % afin de libérer de la place par suppression des trop anciennes versions d'un fichier mais cela a un coût : le temps d'exécution du GC.
On va regarder dans ce TP la mise en œuvre de JFFS2. On va stocker des fichiers dans le système de fichiers JFFS2 et vérifier leur existence même après coupure de l'alimentation de la carte cible.

La carte cible possède une mémoire Flash 16 bits (ST M29W320DT) mappée dans l'espace d'adressage de $2000 0000 à $2040 0000 exclus.

Le partionnement de la mémoire Flash est le suivant :

Plage partition	**Taille partition**	**Taille partition**	**Numéro partition MTD**	**Nom partition**
$203F FFFF $203F 0000	$10000 octets	64 Ko	3	`MAC Address`
$203E FFFF $2016 0000	$290000 octets	2624 Ko	2	`RootFS PK`
$2015 FFFF $2004 0000	$120000 octets	1152 Ko	1	`Kernel PK`
$2003 FFFF $2000 0000	$40000 octets	256 Ko	0	`Bootloader PK`

- Le noyau est déjà configuré pour le support de JFFS2. Configurer le noyau µClinux pour utiliser les outils *Flash Tools* :
  ```
  host% make menuconfig
  ```
  et
  ```
  uClinux Distribution Configuration> Flash Tools> valider mtd-utils puis valider flash_erase et mkfs.jffs2
  ```

- Compiler le noyau µClinux :
  ```
  host% make
  ```

- Après compilation, téléchargement et lancement du noyau µClinux dans la cible, on testera le système de fichiers JFFS2. On peut voir que 4 partitions MTD utilisables avec JFFS2 ont été créées (`/dev/mtd0 à /dev/mtd3`). On le vérifiera aussi dans les traces de boot du noyau. On utilisera par la suite la partition MTD 2 « `RootFS PK` » pour les manipulations suivantes :
  ```
  target# cd /proc
  target# cat mtd
  dev:    size   erasesize  name
  mtd0: 00040000 00010000 "Bootloader PK"
  mtd1: 00120000 00010000 "Kernel PK"
  mtd2: 00290000 00010000 "RootFS PK"
  mtd3: 00010000 00010000 "MAC Address"
  ```

- Construire un système de fichiers JFFS2 sommaire à installer dans la partition MTD 2 (`/dev/mtd2`):
  ```
  target# cd /tmp
  target# mkdir -p jffs2/bin
  target# cd jffs2
  target# echo coucou > file1
  target# cat file1
  coucou
  target# cd ..
  target# mkfs.jffs2 -d jffs2 -o jffs2.img
  target# flash_erase /dev/mtd2
  target# cp jffs2.img /dev/mtd2
  ```

- Monter la partition JFFS2 ainsi créée sur `/mnt/` :
  ```
  target# mount -t jffs2 /dev/mtdblock2 /mnt
  target# cd /mnt
  target# ls
  bin    file1
  target# mount
  rootfs on / type rootfs (rw)
  proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
  sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
  mdev on /dev type tmpfs (rw,nosuid,relatime,mode=0755,size=10M)
  devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,mode=600)
  var on /var type ramfs (rw,relatime)
  tmp on /tmp type tmpfs (rw,nosuid,nodev,relatime)
  debugfs on /sys/kernel/debug type debugfs (rw,relatime)
  /dev/mtdblock2 on /mnt type jffs2 (rw,relatime)
  ```

- Démonter la partition JFFS2 :
```
target# cd
target# umount /mnt
```

- Rebooter le noyau µClinux et vérifier de la présence intacte du système de fichiers JFFS2 sur la partition MTD 2.

15 TP 11 : DEBUG D'APPLICATIONS µCLINUX AVEC GDBSERVER

On désire maintenant étudier le debug d'une application µClinux. On a à disposition 2 méthodes :

- Utilisation du port JTAG couplé au debugger GNU GDB. Cela nécessite une sonde JTAG spécifique connectée. Cette méthode permet un debug en Temps Réel. On n'utilisera pas cette méthode.
- Utilisation de l'accès réseau Ethernet de la carte cible. On utilise alors une application client/serveur GDB sur l'hôte et `gdbserver` sur la cible. Cette méthode est à utiliser pour déverminer une application « marchant à peu près ».

En cas de gros plantage, on peut revenir à la première méthode plus lourde ou utiliser aussi un émulateur ! On mettra en œuvre la méthode de debug par le réseau plus simple.

- Créer une application *userland* `mybug` que l'on intègrera sous µClinux :
```
host% cd linux/user
host% cp –r ../../tp/tp11/mybug/ .
```

- Configurer le noyau µClinux pour utiliser l'application *userland* `mybug` et `gdbserver` (a priori fait par défaut) :
```
host% make menuconfig
uClinux Distribution Configuration> Miscellaneous Applications> valider gdbserver
```

- Compiler le noyau µClinux :
```
host% make
```

- Après compilation, téléchargement et lancement du noyau µClinux dans la cible, on lancera l'application `mybug` :
```
target# /bin/mybug
```

- Que se passe-t-il ?

- Lancer d'abord l'application serveur `gdbserver` sur la cible :
```
target# gdbserver :3000 /bin/mybug
```

- Le debug se fera sur le port socket 3000. Lancer sur l'hôte l'application cliente GDB depuis le répertoire de compilation de `mybug.c` pour bénéficier du debug au niveau symbolique :
```
host% cd linux/user/mybug
host% bfin-uclinux-gdb mybug.gdb
```

- Au prompt GDB, se connecter au serveur `gdbserver` :
```
(gdb) target remote @IP_cible:3000
```

- Utiliser alors les commandes GDB pour localiser le bug et la cause du plantage (commandes `s`, `c`, `print`…).

- Les commandes en ligne de GDB étant assez ésotériques, on peut utiliser un debugger comme DDD qui possède une interface graphique conviviale. Il faut voir DDD comme un « *front end* » à GDB. Comme précédemment, après avoir lancé `gdbserver` :
```
host% cd linux/user/mybug
host% ddd –-debugger bfin-uclinux-gdb mybug.gdb
```

- Dans la fenêtre de commandes GDB de DDD, se connecter au serveur `gdbserver` :
```
(gdb) target remote @IP_cible:3000
```

- Debugger maintenant de façon graphique avec DDD. Conclusion sur les 2 méthodes.

16 TP 12 : MISE EN MEMOIRE FLASH DU NOYAU ET DU SYSTEME DE FICHIERS ROOT

On désire maintenant mettre en mémoire Flash le noyau Linux et le système de fichiers *root*.
La mémoire Flash de 4 Mo a été partitionnée comme on a pu le voir au TP 10 et que voit le noyau Linux au boot avec les traces suivantes :

```
physmap platform flash device: 00400000 at 20000000
physmap-flash.0: Found 1 x16 devices at 0x0 in 16-bit bank
 Amd/Fujitsu Extended Query Table at 0x0040
number of CFI chips: 1
RedBoot partition parsing not available
Using physmap partition information
Creating 4 MTD partitions on "physmap-flash.0":
0x000000000000-0x000000040000 : "Bootloader PK"
0x000000040000-0x000000160000 : "Kernel PK"
0x000000160000-0x0000003f0000 : "RootFS PK"
0x0000003f0000-0x000000400000 : "MAC Address"
```

On utilisera la partition MTD 1 « `Kernel PK` » pour le noyau Linux et la partition MTD 2 « `RootFS PK` » pour le système de fichiers *root*.
Pour cela, on utilisera le *bootloader u-boot* pour programmer la mémoire Flash. On partira de 2 fichiers image de référence `vImage0-bf` pour le noyau Linux et `rootfs0.jffs2-bf` pour le système de fichiers *root* car on a une taille de mémoire Flash très petite dont moins de 3 Mo pour la partition MTD 2 qui accueillera le système de fichiers *root*.

- Prendre la main sur *u-boot* en appuyant sur n'importe quelle touche durant de compte à rebours.

- Programmer le noyau Linux :

```
bfin> print gok
bfin> run gok
```

- Programmer le système de fichiers *root* :

```
bfin> print gor
bfin> run gor
```

- Passer l'argument au noyau pour monter la partition MTD 2 comme partition JFFS2 contenant le système de fichiers *root* via la variable d'environnement `bootargs` du *bootloader u-boot* :

```
bfin> setenv bootargs root=/dev/mtdblock2 rw rootfstype=jffs2
```

- Booter sur le noyau Linux en mémoire Flash:

```
bfin> run flashboot
```

- On vérifiera enfin que la partition *root* est bien la partition MTD 2 de type JFFS2 :

```
target# mount
rootfs on / type rootfs (rw)
/dev/root on / type jffs2 (rw,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
mdev on /dev type tmpfs (rw,nosuid,relatime,mode=0755,size=10M)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,mode=600)
var on /var type ramfs (rw,relatime)
tmp on /tmp type tmpfs (rw,nosuid,nodev,relatime)
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
```

17 REFERENCES

- TCP/IP Application Layer Protocols for Embedded Systems. M. Tim Jones. Editions Charles River Media. CDROM inclus avec sources utilisé pour le TP sur SMTP.
- Linux embarqué. P. Ficheux. Editions Eyrolles.
- ADSP-BF537 EZ-KIT Lite Evaluation System Manual. http://www.analog.com/static/imported-files/eval_kit_manuals/ADSP-BF537_EZ-KIT_Lite_Manual_Rev_2_4.pdf
- ADSP-BF537 Blackfin® Processor Hardware Reference http://www.analog.com/static/imported-files/processor_manuals/bf537_hwr_Rev3.2.pdf

18 ANNEXE 1 : PRÉSENTATION D'U-BOOT

19 ANNEXE 2 : AJOUT D'UNE APPLICATION USER DANS µCLINUX

20 ANNEXE 3 : INTERFACE CGI POUR SERVEUR WEB

21 ANNEXE 4 : MISE EN ŒUVRE DE SMTP SOUS LINUX ET µCLINUX

22 ANNEXE 5 : CONFIGURATION RESEAU HOTES ET CIBLES

	POSTE PC01	
	NOM	**ADRESSE IP**
HOTE	savoir	10.7.2.116
CIBLE	blackfin001	10.7.2.118

	POSTE PC02	
	NOM	**ADRESSE IP**
HOTE	reduction	10.7.5.40
CIBLE	blackfin002	10.7.2.119

	POSTE PC03	
	NOM	**ADRESSE IP**
HOTE	infirmation	10.7.5.28
CIBLE	blackfin003	10.7.2.120

	POSTE PC04	
	NOM	**ADRESSE IP**
HOTE	holiste	10.7.5.251
CIBLE	blackfin004	10.7.2.121

	POSTE PC05	
	NOM	**ADRESSE IP**
HOTE	dissonance	10.7.5.36
CIBLE	blackfin005	10.7.2.122

	POSTE PC06	
	NOM	**ADRESSE IP**
HOTE	decision	10.7.5.30
CIBLE	blackfin006	10.7.2.123

Masque de sous réseau : **255.255.248.0**

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

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