# Rapport de TX52

## "*Mise en place d'une chaîne de compilation et réalisation d'un «SlideShow» pour une carte embarquée Embest (Sbc9261)*"

**Etudiants :**
*Benoit Mauduit - GI04 - Filière* ***LEIM***
*Julien Moine - GI04 - Filière* ***LIBRE***
**Enseignant Suiveur :**
*Nicolas Lacaille*

*Printemps 2010*

# Table des Matières

**1 - Introduction ........ 4**
1.1 - Matériel ........ 4
1.1.1 - Carte Embest9261 ........ 4
1.1.2 - Accessoires ........ 5
1.2 - Informations Fournies ........ 6
1.2.1 - Documentation ........ 6
1.2.2 - Sources ........ 7
1.3 - Définitions ........ 7
1.3.1 - La procédure d’amorçage ........ 7
1.3.2 - Le Noyau Linux ........ 8
1.3.3 - Le système de base ........ 8
**2 - Utilisation de la carte ........ 9**
2.1 - Pré-requis ........ 9
2.1.1 - PuTTY ........ 9
2.1.2 - Configurer le réseau ........ 10
2.1.3 - Serveur Tftpd ........ 10
2.2- Charger le système en Ram ........ 10
2.2.1 - Définition du Ramdisk ........ 10
2.2.2 - Configurer U-boot ........ 10
2.3 - Démarrer sur la NAND Flash ........ 11
2.3.1 - Définitions ( Nand Flash ) ........ 11
2.3.2 - Configurer U-boot ........ 11
2.4 - Démarrer via le réseau ........ 12
2.4.1 - Définitions ( NFS ) ........ 12
2.4.2 - Configurer U-boot ........ 12
**3 - Développement ........ 13**
3.1 - Mise en place d'une chaîne de compilation ........ 13
3.1.1 - Théorie ........ 13
3.1.2 - Obtention ........ 13
3.1.3 - Création ........ 14
3.2 - Gestionnaire de Démarrage (U-boot) ........ 16
3.3 - Compilation du Kernel Linux ........ 17
3.3.1 - Problématiques ........ 17
3.3.2 - Configuration du noyau Linux (2.6.24) ........ 18
3.3.3 - Développement Linux 2.6.32 ........ 19
3.4 - Création du Rootfs ........ 23
3.4.1 - Introduction ........ 23
3.4.2 - L'essentiel au système de base ........ 23
3.4.3 - Les besoins pour le sujet ........ 25
3.4.4 - Quelques modifications ........ 27
3.5 - Mise en place d'une machine virtuelle ARM (avec qemu) ........ 28
3.5.1 - Introduction ........ 28
3.5.2 - Création de l'image du disque ........ 29
3.5.3 - Utilisation ........ 0
3.6 - L’Application SlideShow ........ 31

3.6.1 - Choix de Qt et cross-compilation ..................................... 31
3.6.2 - Le développement........................................................... 32
3.6.3 - Auto-lancement à la connection d'un périphérique .......... 33
**4 - Résultats .......................................................................... 35**
4.1 - Avec le 2.6.24 ........................................................................ 35
4.2 - Avec le 2.6.32 ........................................................................ 36
**5 - Conclusions .......................................................................... 38**
5.1 - Benoit...................................................................................... 38
5.2 - Julien ...................................................................................... 38
5.3 - Améliorations possibles ......................................................... 38
**ANNEXES .................................................................................. 40**
A - Compilation du Noyau Linux ..................................................... 40

# 1 - Introduction

Nous avons choisi ce sujet de **Tx** car nous sommes intéressés par le développement sur une plate-forme embarquée et nous manquions de compétences dans ce domaine.

Nous espérions réaliser des recherches personnelles dans ce domaine afin de découvrir le monde particulier de l'open source orienté vers les systèmes embarqués. Ce sujet de **TX** fut pour nous une belle opportunité de réalisé cela.
La carte fournie pour l'unité de valeur est une carte avec un processeur **ARM** de chez **ATMEL** fourni par «*Embest Info&Tech, Co.*» correspondant aux caractéristiques que l'on peut trouver sur ce site : http://www.armkits.com/Product/SBC9261-I.asp.

Le but de la **Tx** est de découvrir les procédés de compilation en mettant en place une chaîne de compilation croisée (*cross-compilation*) spécifique pour la carte, et ainsi pouvoir compiler et démarrer un noyau Linux. Nous découvrirons aussi les différents procédés pour démarrer un système sur la carte, et en finalité, développer une petite application permettant de faire un *slideshow* sur l'écran **LCD** allant chercher des images sur un périphérique connecté à la carte (clef **USB** par exemple).

## 1.1 - Matériel

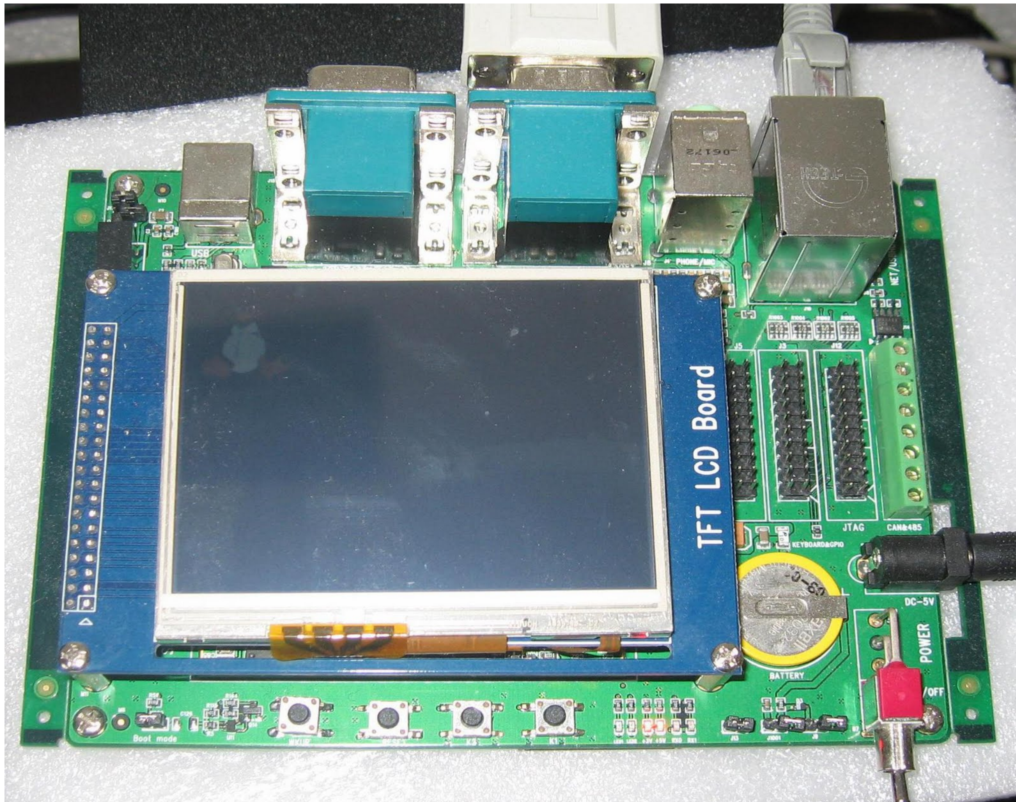
### 1.1.1 - Carte Embest9261

**Caractéristiques techniques de la carte:**

La carte Embest9261 qui constitue notre sujet d'étude à la particularité de posséder un écran LCD tactile ainsi que des ports servant à brancher des périphériques externes. Voici la liste des principaux composants de la carte :

- Processeur 'Atmel AT91SAM9261S' (Cadencé à ***200MHz***)
- **64** Mbyte **SDRAM**
- **128** Mbyte **Nand Flash**
- Port **Ethernet** 10M/100M
- 4 ports **Séries**
- 2 ports Hôtes **USB2.0** (12 Mbits/s)
- 1 port device **USB2.0** (12 Mbits/s)
- 1 écran **LCD** (Avec Touchscreen)
- 1 slot pour carte **SD/MMC**
- etc...

**Aperçu de la carte:**




## 1.1.2 - Accessoires

**Câble Série:**

Nous allons accéder à la console de la carte par l'intermédiaire d'une connexion RS232, pour cela nous utilisons un câble série. C’est l’accessoire indispensable pour obtenir un retour de la console de la carte sur un écran d’ordinateur et ainsi intéragir avec la carte.

**Câble Croisé Ethernet :**

Il nous permet d'avoir une liaison Ethernet entre un ordinateur hôte et la carte. Indispensable également pour charger les nouvelles images de démarages. Il est important d'utiliser un câble croisé car la carte est connectée directement à l'ordinateur.

**Support de cartes MMC :**

La carte qui est entre nos mains possède un port permettant d'y insérer une carte de type MMC (MultiMedia Card). C'est une unité amovible de stockage de données numériques très répandue pour les appareils photos numériques. C'est donc un périphérique important pour nous puisque nous afficherons les photos de la carte MMC sur l'écran.

# 1.2 - Informations Fournies

Avec la carte, il nous a été fourni un CD-Rom avec tout un ensemble de documentations et schémas techniques. Nous disposons aussi d'un ensemble de logiciels permettant de flasher le boot par exemple. Ces logiciels sont surtout destinés aux utilisateurs de Microsoft Windows, délivrant ainsi des interfaces graphiques.

## 1.2.1 - Documentation

La documentation fournie sur le CD-Rom nous a été indispensable pour les opérations de bases sur la carte. Nous avons appris beaucoup de choses grâce aux différents document fournis. Comment utiliser le gestionnaire de démarrage par exemple?; Comment démarrer le noyau en Ram ?; etc... Ces questions seront traitées dans la suite de ce rapport.

On trouve aussi une **FAQ** intéressante sur les pilotes de périphériques qui sont à compiler dans le noyau **Linux** afin de les faire fonctionner avec le matériel de la carte.

### 1.2.2 - Sources

Il y a également tout un ensemble de sources logiciels qui peuvent s'avérer utile, aussi bien pour l'inspiration, que pour s'en resservir.

On trouve sur ce Cd :

- Des **Toolchains**[1] (mais que nous n'utiliserons pas car nous avons fait les nôtres),
- Les sources d'un noyau **Linux** (2.6.24) modifiées pour fonctionner sur la carte,
- Les sources de **DataFlashboot**,
- Les sources de **U-boot** (voir plus bas),
- Différents systèmes de fichiers racines (**Rootfs**).

## 1.3 - Définitions

Avant de poursuivre, il faut définir quelques termes et logiciels utilisés et cités dans la suite de ce rapport.

### 1.3.1 - La procédure d'amorçage

Un gestionnaire d'amorçage est un petit logiciel qui s'exécute au démarrage du système et qui permet d'amorcer le système. La théorie est plus compliquée car plusieurs mécanismes rentrent en jeu. Sur un ordinateur PC par exemple, le BIOS va charger un certain nombre d'octet sur le disque dur principal (512 octets qui correspond au **M**aster **B**oot **R**ecord) qui lui donne l'information sur l'emplacement du chargeur d'amorçage (Windows ou Grub par exemple) et le lance. C'est ce gestionnaire d'amorçage qui va ensuite lancer la procédure de démarrage du système d'exploitation.

Dans notre cas, nous n'avons pas de Bios, mais un emplacement qui est lu automatiquement au démarrage : la NOR flash. Sur celle-ci se trouve un petit logiciel d'amorçage : **DataFlashBoot** (dont les sources sont disponibles sur le cd) à l'adresse physique *0x8000.* **DataFlashBoot** initialise un certain nombre de paramètres et charge ensuite **u-boot**.

«**Das U-Boot**» (*Universal Bootloader*) est le gestionnaire d'amorçage qui pourrait être comparé au célèbre **Grub**. Cependant, il est beaucoup plus léger et est destiné à être exécuté sur plateforme embarquée. **U-boot** doit être configuré (compilé) avec les spécifications de la carte sur lequel il se trouve.

L'avantage est qu'il offre un prompt au démarrage pour pouvoir régler quelques paramètres d'amorçages (comme la commande de démarrage passée au noyau) et

---

1. Les toolchains (Chaîne de compilation) sont utilisées pour compiler des programmes pour une plateforme particulière depuis une plateforme hôte (d'architecture généralement différente de la cible).

faire des requêtes **tftp** pour récupérer un système de fichier ou des images noyaux. Cela est très pratique et nous verrons par la suite comment se servir de ce prompt.

### 1.3.2 - Le Noyau Linux

Le noyau est le coeur du système, il fournit aux logiciels qui s'exécutent dessus une interface pour utiliser le matériel. Le noyau **Linux** est un noyau de système d'exploitation de type **UNIX**, développé initialement par Linus Torvald au début des années 1990.

L'énorme avantage du noyau **Linux** est que chacun peut accéder et contribuer à son développement. Ainsi, le noyau était initialement destiné aux architectures de type **x86**, mais des contributeurs ont porté le noyau sur différentes architectures comme l'**ARM.**

**Linux,** combiné aux outils **GNU** donne **GNU/Linux,** le duo que l'on retrouve aujourd'hui dans presque toutes les distributions. C'est un acteur incontournable pour les serveurs ou les systèmes embarquées.

Nous verrons plus tard dans ce rapport les questions qu'il faut se poser avant d'utiliser un noyau **Linux** à destination des plateformes embarquées.

### 1.3.3 - Le système de base

Le système de base peut être vu comme une distribution **GNU/Linux**. Le noyau, une fois chargé, va lancer le script de démarrage (appelé *script d'init*), et celui-ci va charger le système. C'est l'agencement et la configuration du système qui fait la différence entre les distributions **GNU/Linux** (exemple : Debian, Ubuntu, Fedora, etc...).

Le système de base est généralement stocké dans un espace non temporaire comme la Flash. Nous l'appellerons par la suite le **rootfs**, pour «*root file system*» (Système de fichier racine).

Contrairement à une distribution **GNU/Linux** classique, nous verrons par la suite que nous construirons nous-même la nôtre, avec Buildroot (expliqué dans la partie 3). Nous nous rapprochons ainsi d'un **Gnu/Linux** de type **LFS** (*Linux From Scratch*), mais en simplifié grâce à Buildroot.

# 2 - Utilisation de la carte

## 2.1 - Pré-requis

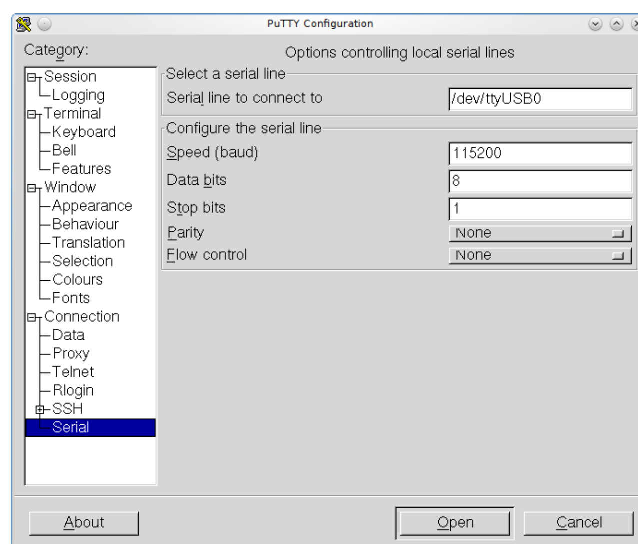
Pour pouvoir utiliser la carte embarquée il est nécessaire de disposer de certains outils indispensables. Tout d'abord, il nous faut un Ordinateur Hôte, qui permettra de recevoir la sortie console, d'établir également une liaison réseau via le câble ethernet croisé pour envoyer des images du système ou du noyau sur lequel nous souhaitons démarrer. Il peut aussi être utile de disposer d'un câble **USB** entre la carte et l'ordinateur lorsque cette fonctionnalité sera activée.

Sur l'ordinateur Hôte, il faut installer :

- **PuTTY** (ou *minicom*, qui est un logiciel similaire) pour la sortie console de la carte,
- Configurer le réseau Ethernet entre la carte et l'hôte,
- Un serveur **Tftp** (*Trivial File Transfer Protocol*) pour l'envoie des images.

### 2.1.1 - PuTTY

PuTTY est un logiciel libre très utile pour établir une connexion avec notre carte et ainsi avoir un retour de la console. Nous nous connectons sur le port série de la carte par l'intermédiaire d'un câble série/USB afin de le brancher sur un port usb de notre ordinateur. A la connexion du câble, une interface **ttyUSB0** (le nom peut différer selon les systèmes...) se crée sur le PC hôte, c'est là que nous obtenons le retour console de la carte. Il faut donc régler PuTTY pour qu'il se connecte sur cette interface (**ttyUSB0**) comme sur cette copie d'écran :



### 2.1.2 - Configurer le réseau

Pour configurer le réseau il faut s'assurer qu'aucun démon ne s'occupe déjà de l'interface câblée, il est nécessaire de désactiver le **network-manager** ou **wicd** par exemple. Ensuite, il faut trouver l'interface câblée sur l'hôte (par exemple eth0) et lui assigner une adresse ip.

*(en root)*

```
# ifconfig eth0 192.168.1.1
```

Même commande sur la carte avec une adresse ip de la même classe (exemple *192.168.1.2*). Un simple ping de la carte vers le serveur hôte (ou inversement) nous permet de tester la connectivité.

### 2.1.3 - Serveur Tftpd

Le protocole **Tftp** *(Trivial File Transfer Protocol)* est un protocole simplifié de transfert de fichier. Par rapport à **FTP** (*File Transfert Protocol*), celui-ci ne gère pas l'authentification, le listage de fichier et le chiffrement. Il reste donc utile à un usage local et correspond parfaitement aux besoins de l'embarquée.
Nous utiliserons ici le serveur **tftp-hpa** qui se configure très simplement.

Depuis la ligne de commande du gestionnaire de démarrage, il est possible de faire des requêtes clientes **tftp**. Nous utiliserons cela pour charger les images Noyaux ainsi que les images du système de fichier dans le cas d'utilisation des **Ramdisk** (expliqué ci-dessous).

## 2.2- Charger le système en Ram

### 2.2.1 - Définition du Ramdisk

Un **Ramdisk** est un disque dur virtuel qui utilise une partie de la mémoire vive (**RAM**) de la carte. Il y a plusieurs avantages à utiliser un tel système : D'une part, les accès à la mémoire vive sont beaucoup plus rapides que sur un support disque classique. D'autre part, il n'y a pas besoin de monter le système de fichier maître de la NAND Flash.

Cependant, le désavantage majeur de ce mécanisme est que à l'extinction du système, toutes les données en **RAM** sont perdues. Il faut donc garder à l'esprit que c'est un moyen temporaire de démarrage. Nous l'utiliserons d'ailleurs exclusivement pour tester les différents noyaux Linux que nous compilerons.

### 2.2.2 - Configurer U-boot

Pour démarrer sur un Ramdisk, il faut accéder au menu d'**u-boot** et le charger par tftp. Pour notre carte **Embest**, il faut le charger à l'adresse *0x21100000*

(emplacement défini dans la documentation constructeur). Il faut ensuite changer la commande de démarrage pour préciser au noyau qu'il va charger un ramdisk et que la procédure d'initialisation est dessus (le deuxième paramètre renseigne la taille du fichier à charger, ici approximativement 4 Mo).

```
Mini9261> tftp 0x21100000 ramdisk.gz
Mini9261> setenv bootargs initrd=0x21100000,0x400000 root=/dev/ram0
console=ttySAC0,115200
```

## 2.3 - Démarrer sur la NAND Flash

### 2.3.1 - Définitions ( Nand Flash )

La Nand Flash sur la carte **Embest** est en quelque sorte le "disque dur" du système : Contrairement à la mémoire vive, la flash conserve les données dans le temps.

Elle contient en fait tout le système de fichier (**rootfs**) avec aussi bien les scripts d'initialisations, que les modules du noyau et les fichiers utilisateurs.

Pour la flash, il est nécessaire, comme pour un disque dur, d'être formatée pour un système de fichier cible (*ex: ext2, NTFS, etc...*), nous utiliserons ici celui qui était par défaut, à savoir **YAFFS2**[2] .

### 2.3.2 - Configurer U-boot

Nous supposons ici que le **rootfs** sur la Flash est valide, nous verrons plus loin dans le rapport comment le changer. Normalement, **u-boot** est configuré pour démarrer automatiquement sur la Flash. La commande de démarrage doit préciser la partition système de la flash (ici: */dev/mtdblock1*) et éventuellement le type du système de fichier (**yaffs2**).

```
Mini9261> setenv bootargs noinitrd root=/dev/mtdblock1 rootfstype=yaffs2
console=ttySAC0,115200

Mini9261> setenv bootcmd run flashboot
```
[3]

2. YAFFS2 : Yet Another Flash File System, c'est un système de fichier spécialement dédié aux disques à base de mémoire NAND flash.

3. flashboot est un alias qui va d'abord initialiser la Flash, puis la Ram et lance ensuite la séquence de démarrage à l'adresse 0x10050000

## 2.4 - Démarrer via le réseau

### 2.4.1 - Définitions ( NFS )

Démarrer via le réseau est un procédé que l'on retrouve souvent dans le développement embarqué. Il a l'avantage d'éviter d'une part de solliciter la mémoire flash continuellement et donc d'augmenter sa durée de vie matériel, et d'autre part, il devient très facile d'apporter des modifications au **Rootfs**, car celui-ci se retrouve directement sur l'ordinateur de développement.

Nous avons donc mis en place un partage NFS (*Network File System*) sur une machine de développement avec un **Rootfs** complet (Nous ne parlerons pas ici de la configuration du serveur **NFS**).

### 2.4.2 - Configurer U-boot

Il suffit encore une fois de modifier la commande de démarrage, qui aura besoin de connaître l'adresse du partage **NFS** :

```
Mini9261> setenv bootargs noinitrd root=/dev/nfs rw nfsroot=192.168.1.1:/srv/nfs
nfsaddrs=192.168.1.2:192.168.1.1:::: console=ttySAC0,115200
```

L'adresse du partage étant : *192.168.1.1:/srv/nfs*
et l'adresse **ip** de la carte est *192.168.1.2* et celle du serveur *192.168.1.1*. (les autres commandes telles que définition de la passerelle, du masque, etc. ne sont pas nécessaires).

# 3 - Développement

## 3.1 - Mise en place d'une chaîne de compilation

### 3.1.1 - Théorie

La compilation consiste à transformer du code initialement en langage source (le C par exemple) en un langage cible (langage machine). Le problème est que le langage cible est spécifique pour une architecture de machine. (x86, x86_64, ARM, MIPS, etc..).

On se rend vite compte que dans notre cas, l'architecture visée est l'**ARM,** il nous faudrait donc logiquement un compilateur sur la carte et compiler nos programmes directement sur la carte. Cela est en fait maladroit car la puissance de calcul de ce processeur est très faible comparée à celui d'une machine que nous possédons pour travailler par exemple, compiler le noyau **Linux** serait beaucoup trop long.

Pour palier à ce problème, nous effectuons ce que l'on appelle de la *cross-compilation* : le compilateur va produire du code objet spécifique à une machine cible, généralement d'architecture différente de la machine hôte. Dans la théorie, pour compiler un compilateur croisé, il faut au préalable un compilateur fonctionnel pour la machine hôte. Ce compilateur va servir à compiler un autre compilateur qui sera quant à lui exécutable sur la machine hôte mais produira du code pour la machine cible (ici du code pour plateforme **ARM**).

Cela est dans la pratique assez difficile car il faut préciser au compilateur un nombre conséquent de paramètres correspondant aux spécifications de la machine cible. Divers moyens existent pour générer son *cross-compiler* que nous allons détailler ci-dessous.

### 3.1.2 - Obtention

**Cd**

Sur le cd fourni avec la carte, on retrouve des Toolchains, qui ont d'ailleurs été générées à partir de crosstool, un outil permettant d'automatiser le processus de cross compilation.
Celles-ci contiennent un compilateur (**Gcc** version **3.4.5**) avec **glibc**[4]**.** Plusieurs objectifs nous ont poussé à ne pas retenir cette solution. D'une part, cela n'aurait pas été très pédagogique et cette partie nous semblait très intéressante à approfondir. D'autre part, si on regarde plus en détail, le compilateur généré est un binaire 32 bit et la version est assez ancienne (dernière sortie en date de **Gcc** : version 4.5.0).

---

4. glibc est la bibliothèque standard du language C.

Possédant des machines en 64bit, il serait plus intéressant d'avoir un binaire qui exploite les capacités de l'architecture, surtout pour un compilateur. Aussi, la **glibc** est une librairie lourde et peut être remplacée par **uclibc** sur les plateformes embarquées.

**CodeSourcery**

Il est possible aussi de se procurer des Toolchains spécifiques via CodeSourcery. Mais pour les mêmes raisons nous avons décider de compiler les nôtres.

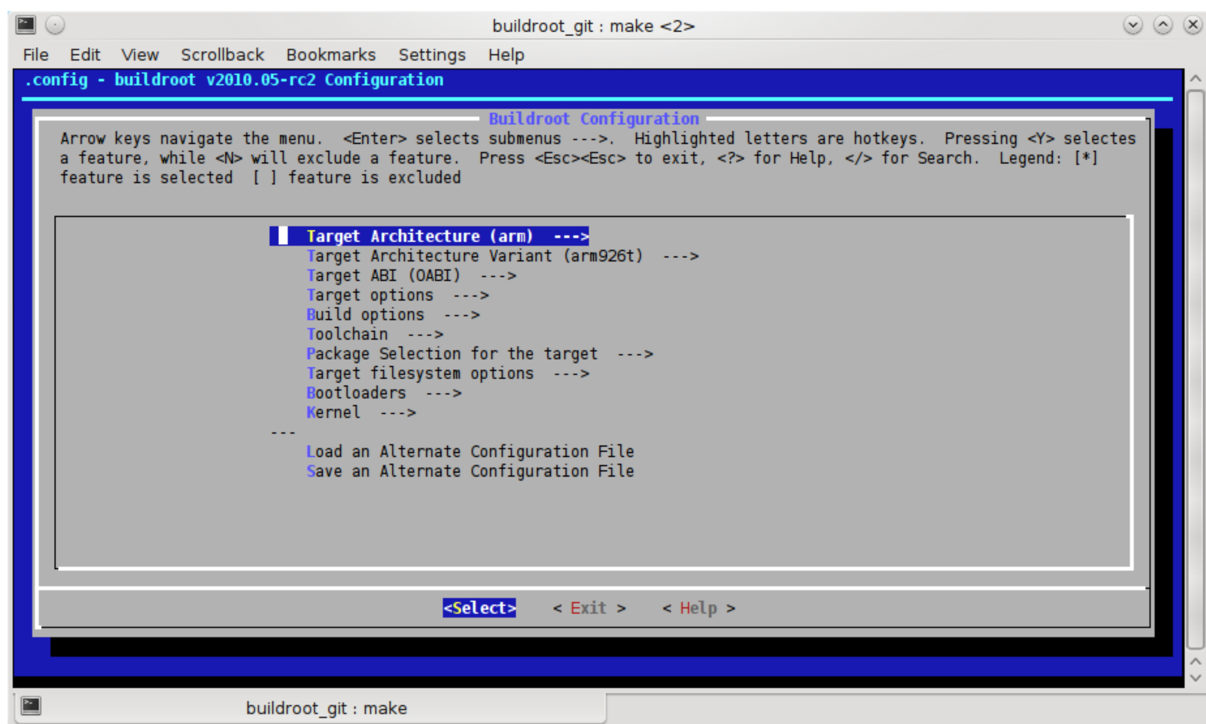
### 3.1.3 - Création

**Crosstool**

Crosstool est un outil pour automatiser la création de son cross compilateur. Cependant, au vu des capacités de **Buildroot** nous ne l'avons pas étudié.

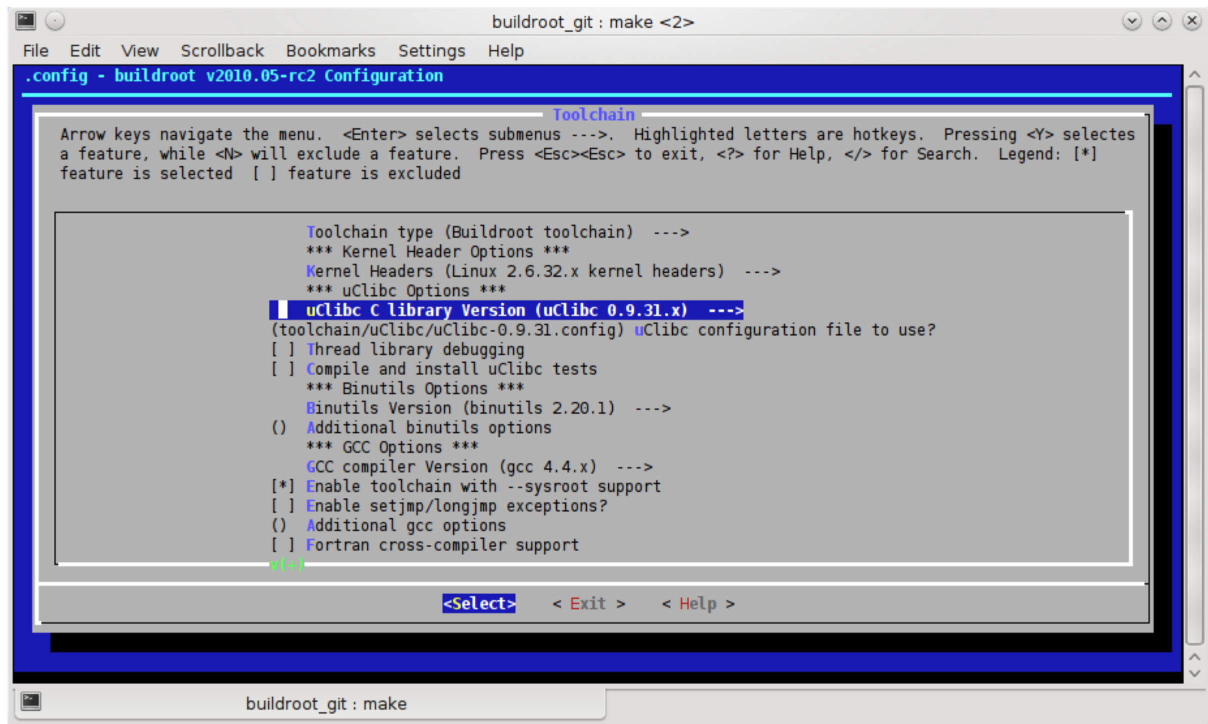
**Buildroot**

**Buildroot** est un ensemble de fichiers de règles de compilation (**Makefile**) qui facilite grandement la génération des **toolchains** mais aussi du système de fichier final. C'est donc un outil presque tout en un pour la génération de systèmes cibles.
Il est relativement facile à utiliser car il intègre le même outil de configuration que le noyau **Linux**. Il faut d'abord sélectionner les caractéristiques de la machine cible (Type de processeur, jeu d'instruction, etc...) utiles pour la génération des **Toolchains**, le reste sera expliqué plus tard, pour la génération du **rootfs**.



Voici l'interface de configuration de buildroot, dans notre cas il faut impérativement régler la *Target Architecture* sur *arm, Target Architecture Variant* sur *arm926t* et *Target ABI* sur *OABI.*

Nous allons maintenant dans la section **Toolchain** et nous allons choisir notre version de **Gcc** (ici 4.4.x, la 4.5.0 n'étant pas ajoutée à **buildroot**) et nous choisissons aussi de compiler la dernière version stable de uClibc (*0.9.31*).




Il faut aussi sélectionner quelques options en plus qui seront obligatoires pour le **rootfs** (Comme le support des WCHAR, des fichiers larges, et un compilateur *c++* avec la *libstdc++* nécessaire pour *Qt* par exemple)

Une petite astuce pour accélérer la compilation est de spécifier dans le menu «*Build Option*» le nombre maximum de jobs de compilation. Il faut en général spécifier un nombre équivalent aux nombres de processeur ou de coeur sur la machine (Par exemple, notre machine hôte étant un core i7 avec 8 coeurs nous pouvons spécifier 8 jobs). Cela aura pour conséquence de paralléliser la compilation entre les différents coeurs.

La compilation peut donc être lancée avec la commande make (en se plaçant dans le répertoire de buildroot). Les toolchains peuvent ensuite être récupérés dans le répertoire :
*«output/staging/»*.

## 3.2 - Gestionnaire de Démarrage (U-boot)

Nous n'avions pas un besoin spécifique de recompiler u-boot car c'est une zone critique qui peut perturber le démarrage de la carte, et il faut donc une bonne raison de remplacer le gestionnaire existant. Cependant, pour des raisons pédagogiques, nous avons regardé les sources fournies sur le Cd puis nous l'avons compilé avec nos **Toolchains**.

Dans le répertoire *include/configs*, nous retrouvons un fichier de configuration spécifique pour beaucoup de cartes embarquées. Le notre ne correspond pas totalement à notre carte, mais à une similaire, ou modifiée par **Embest** pour correspondre. Le fichier est :
*at91sam9261ek* (correspondant à cette carte).

Ce fichier définit via des constantes :

- le type du **CPU**,
- les périphériques et leurs configurations,
- le mappage de la mémoire,
- les composants d'**U-boot** qui doivent être compilés.

Par exemple, dans ce fichier, nous retrouvons la configuration du réseau par défaut (elle peut être changée par la suite dans le prompt d'**U-boot**) :

```
#define CONFIG_IPADDR          192.192.192.200
#define CONFIG_GATEWAYIP       192.192.192.101
#define CONFIG_ETHADDR         DE:AD:BE:EF:01:01
#define CONFIG_NETMASK         255.255.255.0
#define CONFIG_SERVERIP        192.192.192.105
```

Ou encore la commande de boot par defaut :

```
#define CONFIG_BOOTARGS "noinitrd root=/dev/mtdblock1 rootfstype=yaffs2
console=ttySAC0,115200"
```

Enfin, pour compiler U-boot, il suffit de spécifier la configuration utilisée avec :

```
make  at91sam9261ek_config
```

Puis s'assurer que nos toolchains sont dans notre variable d'environnement PATH :

```
$ export PATH=/Path/to/toolchain:$PATH
```

Par exemple sur notre machine hôte, les toolchains se trouvent dans *«/opt/ArmToolchain»*, la variable PATH vaut :

```
$ echo $PATH
/opt/ArmToolchain/usr/bin:/bin:/usr/bin:/sbin:/usr/sbin
```

*(Cette manipulation du **PATH** ne sera pas redétaillée dans le rapport, mais sera similaire lorsque nécessaire)*

Pour finir, il nous faut compiler U-boot avec nos **toolchains** avec la commande suivante :

```
$ make CROSS_COMPILE=arm-linux-
```

Une fois fini, nous disposons d'un binaire : **u-boot.bin.** Pour flasher et installer ce binaire sur la carte, il faut suivre la procédure définit dans le guide utilisateur sur le CDRom fournit avec la carte, *Partie 1.4 - Download U-boot to Dataflash, page 6* (en utilisant le logiciel **SAM-BA**). Nous n'avons pas réalisé cette opération.

## 3.3 - Compilation du Kernel Linux

### 3.3.1 - Problématiques

Compiler un noyau n'est jamais une chose facile... De premier abord, on se retrouve avec un menu souvent bien documenté certes, mais très dense. Il ne faut pas hésiter à approfondir les termes que l'on ne comprends pas bien. Compiler un noyau à l'aveugle est le meilleur moyen d'en obtenir un qui ne démarrera pas (*kernel panic* ou autres erreurs). Il faut aussi bien se renseigner sur son matériel et avancer progressivement. Heureusement, les noyaux pour architecture **ARM** sont plus faciles à configurer qu'un noyau pour **x86** par exemple car ils disposent de beaucoup moins d'options.

Une autre problématique se pose pour le noyau **Linux**, c'est le choix de la version. En effet, celui-ci subit un développement intensif et une nouvelle version sort en moyenne tout les trois mois. En théorie, pour une machine de bureau (type intel x86) il est souvent intéressant de récupérer la dernière version stable. Pour un noyau **ARM**, cela dépend des besoins. Il y a en général des versions précises du noyau qui une fois testées et les bugs corrigés plaisent mieux que d'autres pour être choisies sur le matériel cible. Nous avons choisis la version 2.6.32 (La dernière version en date étant la 2.6.34).

Cependant, nous nous sommes rendus compte que les sources du noyau (**2.6.24**) disponible sur le cd n'était pas un noyau officiel et a été modifié (patché) pour rajouter quelques options indispensables pour le bon fonctionnement de la carte. Nous avons donc décidé de nous intéresser aux sources du noyau et de regarder ce que faisait ces modifications. Cette partie sera détaillée dans la section correspondant à la version du noyau dont nous nous sommes attaqués (le **2.6.32**).

### 3.3.2 - Configuration du noyau Linux (2.6.24)

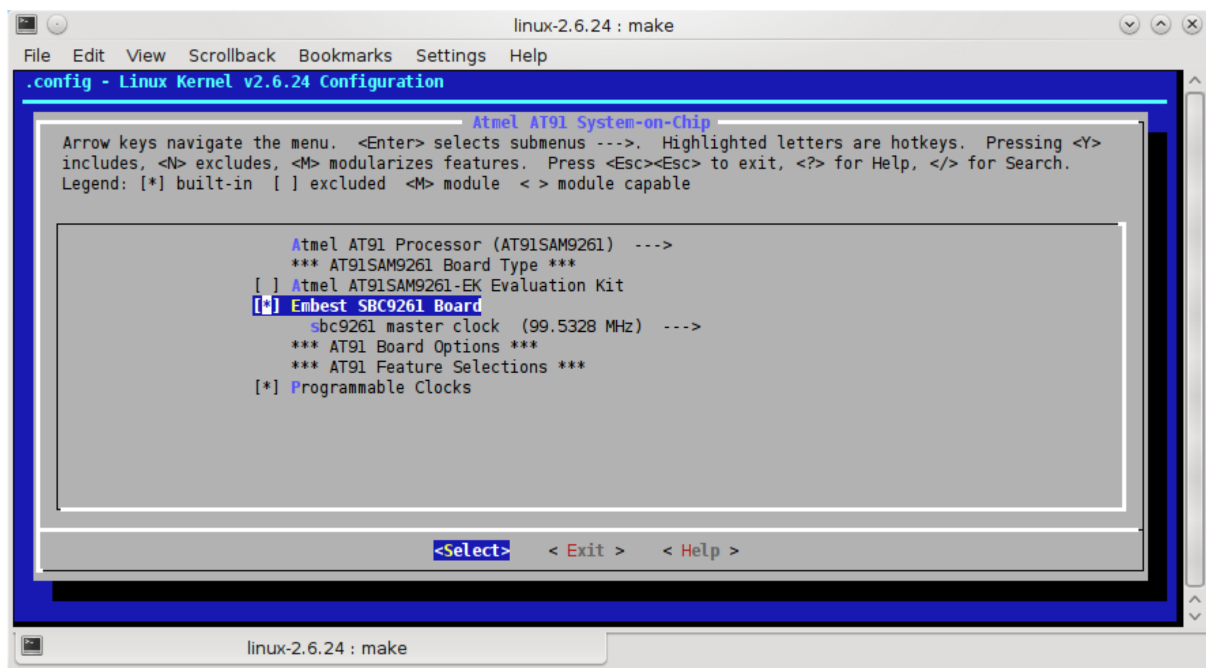
Cette version est fournie sur le cd avec un ensemble de modifications pour fonctionner sur la carte embarquée. Le fichier de configuration est correct et fonctionne directement cependant on peut améliorer la gestion de la compilation des drivers en module ou en dur dans le noyau. Sachant qu'il est préférable de ne pas charger du code qui ne sera pas utilisé ou très occasionnellement, les modules sont fait pour ça. Aussi, il est intéressant de voir la configuration d'un noyau **Arm** qui est sensiblement différente de la configuration d'un noyau x86 par exemple.

Dans les options indispensables, il faut tout d'abord choisir notre type de machine (**Atmel AT91**) et spécifier la famille de processeur **AT91SAM9261** dans le menu *AT91 System-on-Chip*. Cela donne ensuite accès à notre carte **Embest SBC9261 Board** (Cette option n'est pas présente dans un noyau de base, mais correspond bien à un patch d'Embest)**.** Nous verrons pour le noyaux **2.6.32** que cette partie compile des fichiers d'initialisation de la carte.

Ce menu est obtenu en tapant :

```
make menuconfig ARCH=arm
```

...dans le repertoire des sources du noyaux




Il faut ensuite sélectionner tous les drivers de la carte (MMC, LCD, Usb, Réseau, bouton, led, etc..). Pour nous aider, une documentation sous forme de FAQ est disponible dans la documentation du cd.

**<u>En dur ou en Module ?</u>**

Le noyau permet de compiler les options de deux manières différentes. Soit directement dans le fichier binaire final (dit «**en dur**») soit indépendamment (dit «**en module**»). L'avantage de cette distinction est, pour un noyau bien compilé, qu'il n'y aura pas de code inutile dans le code de base. Aussi, mettre seulement l'essentiel en dur permet de réduire l'image binaire finale, pratique pour de l'embarqué si la place d'exécution du noyau est petite. Les modules sont installés sur le système de fichier (dans */lib/modules/<versionduNoyau>/*) et chargés seulement au besoin (par exemple, à l'insertion d'une carte **MMC**, le module se charge et la carte sera reconnue).

Il y a cependant une chose à faire attention : Les modules peuvent être chargés seulement après que le système de fichier soit monté, ce qui signifie que si par exemple le driver du système de fichier *Root* ou celui de la gestion de la flash sont compilés en module, le noyau sera incapable de démarrer (bien que des mécanismes existent pour palier ce problème : **initrd** pour **INITial RamDisk** utilisé par *Debian*, *Ubuntu* et bien d'autres). C'est pour cela qu'il faut bien choisir et définir "qui fait quoi" au moment de la configuration.

Nous compilerons donc en module le support des systèmes de fichiers externes à la carte (comme le **FAT**[5], le **NTFS, ext2** etc...), le driver de la carte **MMC** et les pilotes **USB**. Nous compilerons en dur tout le reste et surtout le driver **YAFFS2** (système de fichier du **rootfs**), le pilote de la sortie série (Pour avoir un retour direct en cas de problème), et les pilotes principaux du matèriel de la carte (l'**Ethernet** par exemple, l'écran **LCD**).

L'annexe **A** décrit les commandes de bases pour compiler notre noyau et les modules.

### 3.3.3 - Développement Linux 2.6.32

Nous nous sommes ici lancé un gros défi que nous avons malheureusement pas pu aboutir jusqu'au bout, par manque de temps mais aussi d'expériences.

Le problème majeure ici, est que les drivers spécifiques de la carte n'existent pas dans le noyau de base (surement par manque d'utilisation de la carte par la communauté libre). Comme nous le disions au dessus, les sources du noyaux **2.6.24** récupérées sur le cd ont été modifiées.

Nous avons donc regardé les différences, et il manque :

- La définition spécifique de la carte (dans *./arch/arm/mach-at91/board-sbc9261.c*)
- Le «*Machine Id*» envoyé par **U-boot** ne correspond pas (le noyau ne se lance donc pas)
- Le système de fichier **YAFFS2** n'a jamais été inclus dans la branche principale du noyau.

5. FAT est un système de fichier Microsoft répandu sur la plupart des périphériques externes

- Le pilote du **SPI** (*Serial Peripheral Interface Bus*) (**Tsc2301**)
- Le pilote du Touchscreen (**Tsc2301**)
- Le pilote du Clavier (Les 4 boutons de la carte)
- Le pilote Audio.

## Avant de commencer le Développement

Il va nous être très pratique d'utiliser le gestionnaire de version utilisée pour le noyau Linux, en l'occurence **Git** (nous ne détaillerons pas dans ce rapport son utilisation). Grâce à celui-ci, nous allons récupérer la branche principale du noyau mais en spécifiant le Tag **2.6.32**, qui nous mettra à l'état des sources comme elles l'étaient pour la sortie de cette version, avec en plus les derniers ajouts (Patch de sécurités, corrections de bug, etc...).

Le travail qui va être realisé est juste un portage des sources spécifiques de la carte que nous retrouvons dans la version **2.6.24** présentes sur le **Cd** vers le **2.6.32** de la branche officielle du noyau Linux.

Ces deux versions sont espacées de presque 2 ans (le **2.6.24** est sorti le *25 janvier 2008*, le **2.6.32** est sorti le *03 décembre 2009*) et la difficulté va être que la plupart des API (*Application Programming Interface*) ont changé.

En reprenant le fichier *arch/arm/mach-at91/board-sbc9261.c* et en le mettant dans les sources du **2.6.32**, nous nous rendons compte que certains *#includes* n'existent plus. C'est en fait dû à une reorganisation des chemins du noyaux entre les deux versions. Il nous suffit donc de comparer avec d'autres fichiers de configurations du **2.6.32** et retrouver les bons chemins. Ensuite, il a fallu désactiver certaines initialisations obsolètes, et reprendre l'initialisation des sorties séries.

## Board-sbc9261.c

Pour que notre fichier puisse être disponible dans le menu-config (Menu comme dans les captures d'écran précèdentes) du noyau compilé, il faut modifier le fichier *Kconfig*, celui-ci est trivial et cette opération sera renouvelée plusieurs fois. Une entrée ajoute un menu et correspond à la définition d'une constante (utilisée ensuite dans le *Makefile*). On peut aussi ajouter des sous-menu qui seront activables suivant la dépendance définie. Par exemple, la modification apportée pour rajouter le menu pour notre carte est :

```
config MACH_SBC9261
	bool "Embest SBC9261 Board"
	depends on ARCH_AT91SAM9261
	help
	  Select this if you are using Embest sbc9261 board.

choice
	prompt "sbc9261 master clock "
	depends on MACH_SBC9261
```

```
	default SBC9261_100MHz

config SBC9261_100MHz
	bool "99.5328 MHz"

config SBC9261_120MHz
	bool "119.8080 MHz"

endchoice
```

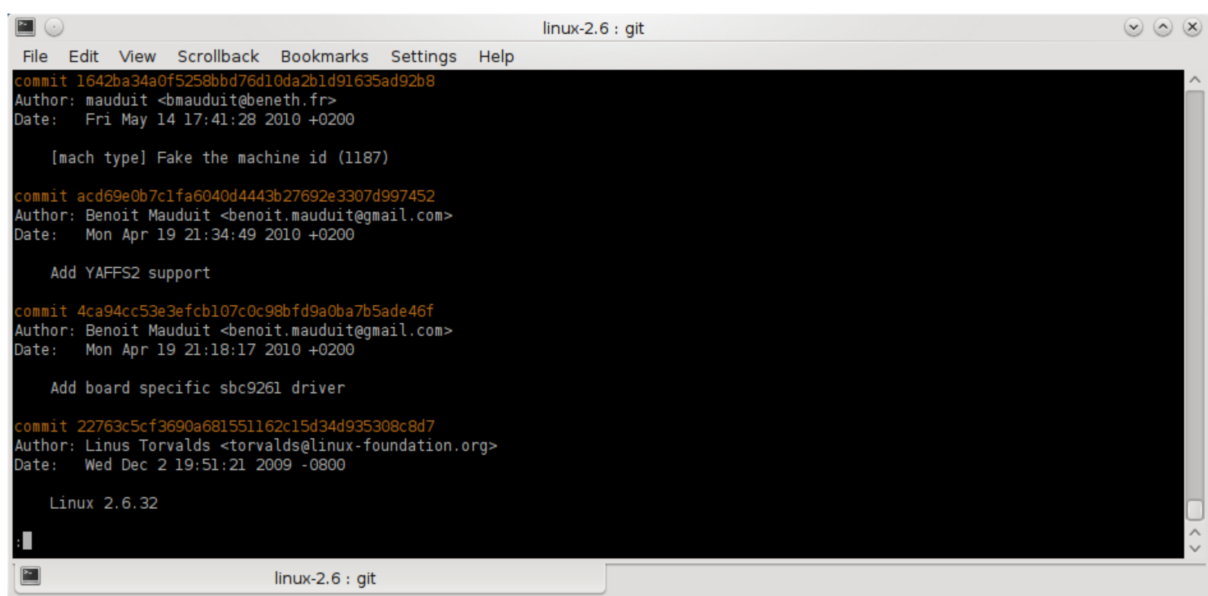
La constante *MACH_SBC9261* est ensuite utilisée dans le Makefile que nous allons modifier en ajoutant :

```
# AT91SAM961 Embest
obj-$(CONFIG_MACH_SBC9261)    += board-sbc9261.o
```

Pour ajouter une parenthèse pour justifier l'utilisation de **GIT :** à ce stade nous avons effectué une modification conséquente et précise (ie: «*ajouter aux sources du noyau la définition de la carte sbc9261*»). Il est donc nécessaire d'effectuer un commit. Avec **Git**, un commit se passe en local (pour envoyer un ou plusieurs commit sur le serveur, il faut effectuer un *push*), de toute façon on ne risque pas d'avoir le droit d'envoyer nos commit sur le développement principal du noyau sans validations préalables.

Cette action nous permet de retrouver clairement nos modifications sur le noyau et par la suite de générer un fichier de différence (un *diff*) qui pourra être utilisé ensuite pour «patcher» (ie: Appliquer les différences du *diff* dans les sources).

La capture suivante montre la sortie de git log, avec quelques commits effectués sur le noyau :



## Machine ID

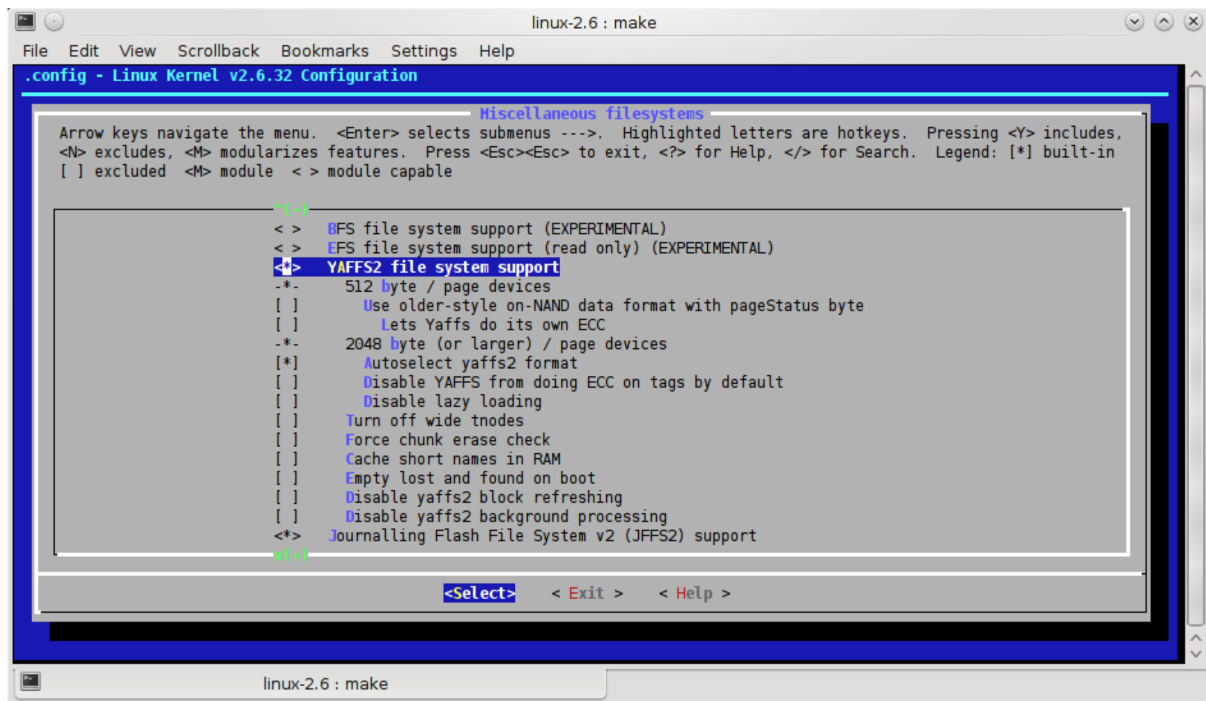
L'identifiant de machine doit être unique par spécification de carte, on le définit au niveau du noyau dans *include/asm-arm/mach-types.h.* Au lancement de la carte, le numéro de machine est envoyé par **U-Boot** dans le registre r1. Le noyau teste ensuite si les numéros correspondent et démarre seulement dans ce cas.

Dans les sources présentes sur le cd, nous nous sommes rendus compte que le numéro de machine de la carte a remplacé un numéro de machine existant (nous avons vérifié en comparant avec les sources officielles du **2.6.24**). Cela n'est pas très propre, mais après quelques recherches, nous n'avons pas trouvé comment changer l'identifiant venant d'**U-Boot.** Nous avons donc procédé de la même manière pour le **2.6.32**...

## Ajout du système de fichier Yaffs2

Le système de fichier **Yaffs2** n'a jamais été ajouté aux sources officielles du noyau pour des raisons diverses et variées (principalement parce que le système **Jffs2,** de performances similaires**,** est préféré par la communauté). Cela est problématique car c'est notre système de fichier par défaut sur la **NAND Flash**.

Cependant, il suffit de se rendre sur le site officiel de **Yaffs2** ([http://www.yaffs.net/](http://www.yaffs.net/)) pour obtenir un patch que l'on peut appliquer au noyau Linux pour en ajouter son support. Une fois le patch appliqué, il suffit de se rendre dans la catégorie FileSystem et selectionner le menu **Yaffs2** :



**Le driver Tsc2301**

Le travail ici n'a pas été terminé par faute de temps et de connaissance en développement Kernel. Cependant, nous disposons de quelques références, et un aboutissement de ce travail pourrait être intéressant. Une recherche sur internet de **Tsc2301** nous envoie sur ce document de spécification : http://focus.ti.com/lit/ds/symlink/tsc2301.pdf. C'est un controlleur fabriqué par Texas Instrument qui gère le Touchscreen, l'audio, etc...

Le portage des sources s'est révélé complexe et fastidieux car les interfaces ont beaucoup changé entre les deux versions. Nous avons réussi à faire compiler le driver SPI, le clavier et l'écran tactile. Cependant, cela doit être vérifié, car par exemple, lors de l'insertion d'une carte dans l'emplacement **MMC**, nous obtenions un message d'erreur («Can't access to SPI bus»)...

Pour conclure cette partie sur le développement du noyau, nous avons réussi à démarrer avec un noyau 2.6.32 et des fonctionnalités de base de la carte fonctionnent. Nous avons donc atteint partiellement notre objectif.
Il manque cependant:

- L'**Audio**
- Le **Lcd**
- Le lecteur de carte **MMC**

# 3.4 - Création du Rootfs

## 3.4.1 - Introduction

Le principe de compilation est le même que pour le noyau. Il faut récupérer les sources du logiciel que l'on desire avoir sur la carte, puis les compiler avec nos Toolchains. Cela peut s'avérer très long et fastidieux.. Heureusement, **Buildroot** nous permet d'automatiser cette tâche.

Il suffit, via le menu, de choisir les logiciels dans la liste, et **buildroot**, lors de la compilation, va chercher automatiquement les sources sur le site officiel, et les compiler avec les toolchains obtenues précédemment.

Nous allons développer la création du **rootfs** en plusieurs étapes : d'une part nous préciserons les paquets essentiels au bon fonctionnement du système après le lancement du noyau, nous présenterons ensuite les choix des logiciels pour l'objectif de l'unité de valeur (les composants utiles au *slideshow*).

(Le menu pour choisir les logiciels est : «*Package Selection for the target*». )

## 3.4.2 - L'essentiel au système de base

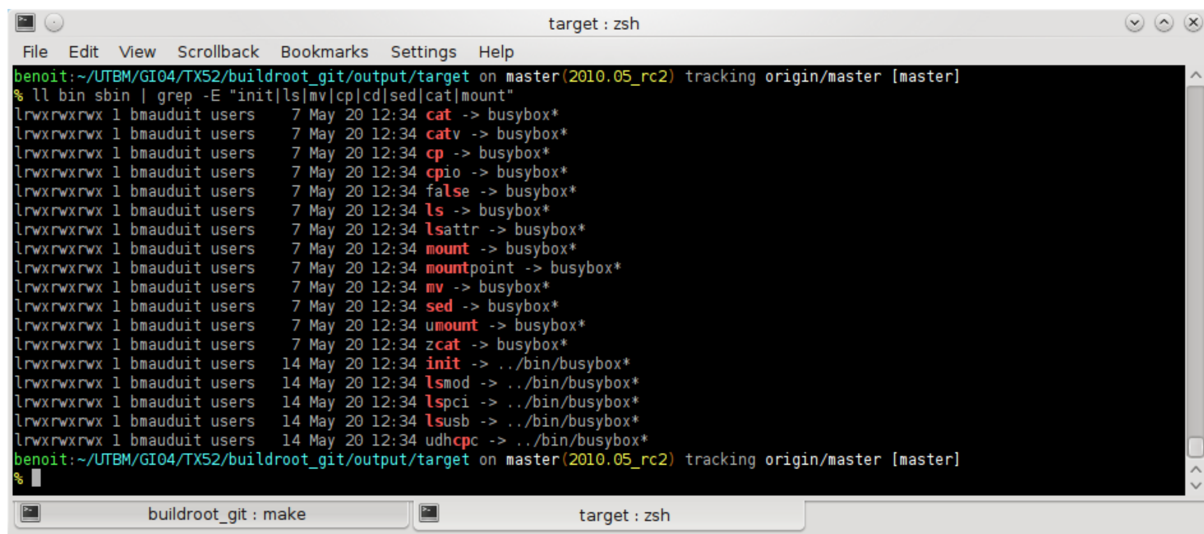
La dernière action du noyau **Linux** après son chargement est d'appeler un script d'initialisation (le *script* d'init, possédant l'identifiant 1, le plus petit). Il est responsable

de la dernière phase du processus de démarrage : il vérifie le système de fichier racine et monte les autres, il active les interfaces reseaux et charge les scripts de démarrage qui sont configurés pour se lancer automatiquement (par exemple, si on veut qu'un serveur **ssh** soit lancé directement au démarrage) ceux-ci étant très souvent présent dans le repertoire /etc/init.d du système.

Le script de démarage se trouve dans presque toute les distributions dans */sbin/init,* mais peut être spécifié dans la ligne de chargement du noyau en ajoutant le chemin à la variable init.

Ensuite, il est nécessaire d'avoir un certain nombre de commandes de base comme *cp* (copie un fichier), *mv* (Déplace un fichier ou un répertoire), *ls* (Liste les fichiers), ou *cd* (Change de répertoire), et bien d'autres.

Dans le monde de l'embarqué, nous retrouvons un outil tout en un pour le système de base, appelé **BusyBox**. Il contient une version épurée de presque toutes les commandes de base d'un système **Unix** développées par **GNU** (Comme celles citées ci-dessus).
Son avantage pour l'embarqué est qu'il contient toutes ces commandes, dont le script *d'init*, dans un seul binaire, résultant un gain de volume conséquent.
La capture d'ecran suivante, confirme cela avec un affichage de quelques commandes de base :




*Note: La sortie de ll (alias ls -l) montre que toute les commandes sont des liens vers /bin/busybox.*

**Busybox** est un outil incontournable dans le développement de système embarquée de type Unix. Celui-ci contient les commandes essentiel pour pouvoir démarrer sur notre carte !

### 3.4.3 - Les besoins pour le sujet

Pour notre sujet, à savoir développer un *slideshow*, il va falloir afficher sur l'ecran **LCD** de la carte des images (pas nécessairement de même type) provenant de medias externes (clés usb, cartes Sd, etc...).

Pour la partie ne concernant pas l'affichage, nous avons définis les besoins suivant :
*A l'insertion d'un media externe, faire :*

- *Creation d'un point de montage,*
- *Montage du périphérique (en lecture seule),*
- *Lancement du slideshow dans le repertoire du périphérique,*
- *Fin du slideshow,*
- *Demontage du périphérique,*
- *Suppression du point de montage.*

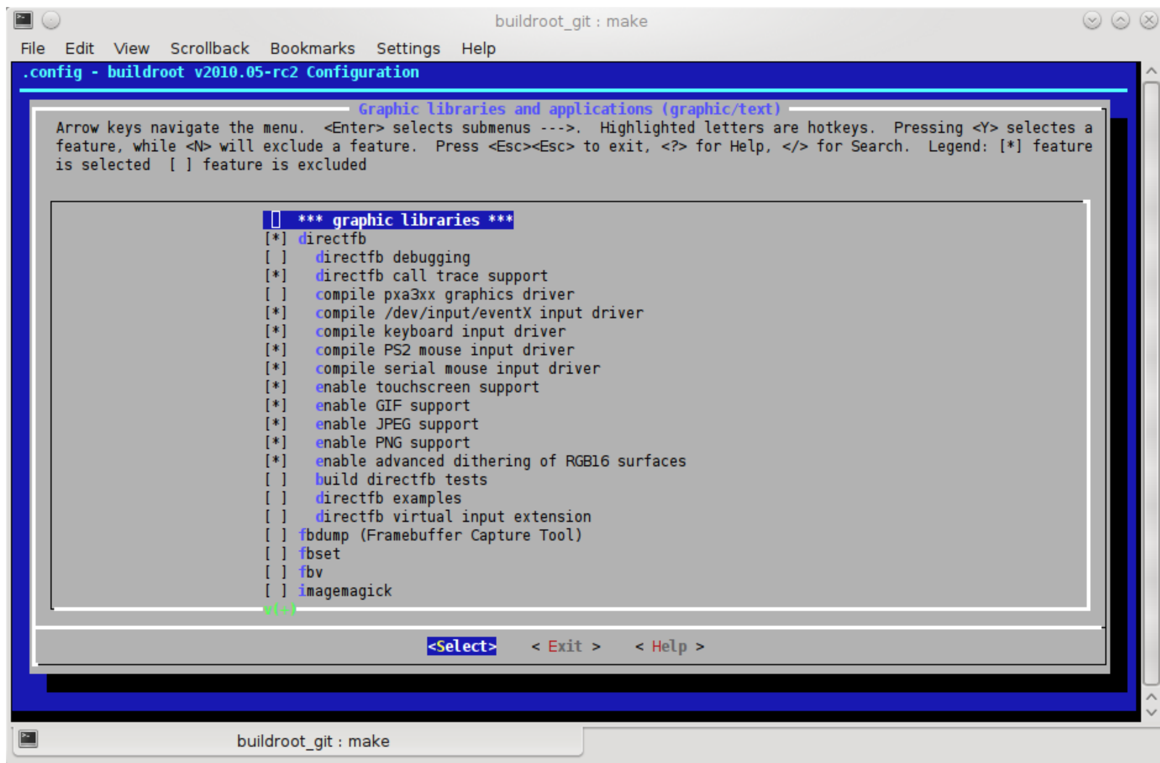
Après quelques recherches, cela peut se faire via des règles **udev.** Ce dernier est un gestionnaire de périphériques sous **GNU/Linux** (compatible à partir des versions **2.6.x** du noyau), qui permet de gèrer automatiquement la création des noeuds dans */dev (*Chaque périphérique branché sur le système nécessite un noeud pour y accéder physiquement, par exemple, pour la première partition du disque dur principal, il est courant que son noeud associé soit : */dev/sda1).*

Les règles **udev** permettent de rajouter des options lors de la détection d'un périphérique suivant des conditions et aussi d'exécuter des scripts au montage du périphérique. Il remplit donc parfaitement nos besoins (nous verrons plus loin dans le rapport l'écriture de la règle). Pour ajouter **udev** dans notre **rootfs**, il faut activer l'option en dessous de BusyBox : «*Show Package that are also provided by busybox*»[6] puis aller dans le menu «*Hardware Handling*» et cocher le menu «*udev*».

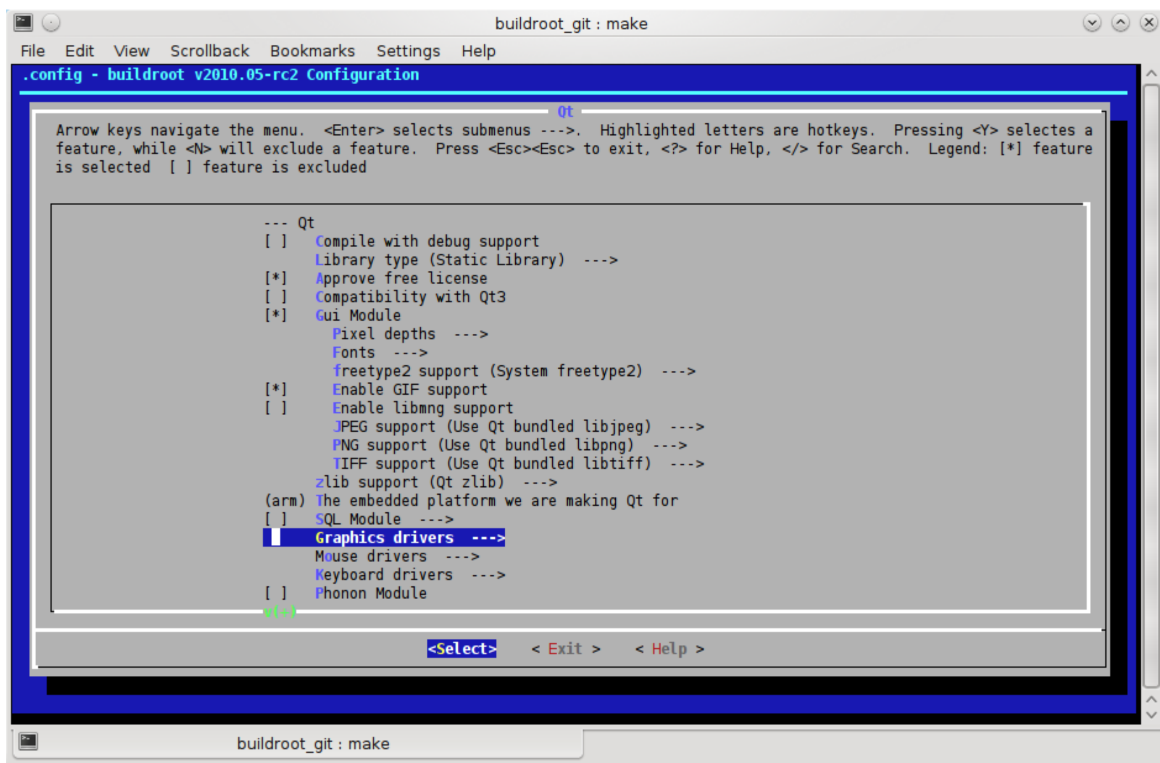
Concernant l'affichage, nous voulions une solution simple, qui nous permette de développer rapidement notre propre application effectuant le *slideshow*, mais en gardant à l'idée que la plateforme de destination dispose de peu de puissance. Sous linux, pour l'affichage, il est souvent nécessaire d'avoir un serveur d'affichage (appelé serveur X), le plus connu et présent dans nos machines de travail est Xorg. Cependant celui-ci est très complet et même trop complet pour nos besoins (et donc risque de prendre trop de ressources). Nous avons donc cherché une alternative correspondant au besoin d'un simple affichage pour le *slideshow*. Nous avons trouvé **DirectFB** (pour *Direct Frame Buffer*), qui nous fournira tout ce qu'il faut pour afficher des images. Cependant, pour faciliter le développement de l'application, et connaissant assez bien la librairie graphique **Qt**, nous avons également trouvé que celle-ci peut être utilisée avec comme pilote graphique **DirectFB** (Nous n'aurons donc pas besoin de serveur graphique).
De plus, **DirectFb** est réputé pour avoir une empreinte mémoire faible et est de plus en plus utilisé dans les systèmes embarquées. Il est également supporté dans

6. Busybox fournit aussi un outil similaire plus léger appelé mdev, mais nous n'avons pas choisis cette solution.

**buildroot** et pour le rajouter au **rootfs**, il faut le sélectionner dans «*Graphic Librairies and applications*» :




Dans le même menu nous ajoutons également **Qt**, en spécifiant bien **DirectFb** pour les pilotes graphiques utilisés. (Menu : «*Qt/Graphic drivers*»)



**Qt** est une librairie graphique, connue grâce au gestionnaire de fenêtre **Kde**, qui propose beaucoup de fonctionnalités et qui a l'avantage d'être très complète et portable sur différentes architectures, comme l'**ARM**. **Qt** a par ailleur été rachetée par la société *nokia* qui fabrique essentiellement des appareils mobiles.

Nous l'avons donc choisi car premièrement nous sommes familiarisé avec le développement de telle application, puis surtout pour son coté multi-plateformes (Cela sera détaillé dans la section 3.5.1).

Enfin, il faut choisir la forme de stockage du **rootfs** après la compilation. Cela se passe dans le menu «*Target Filesystem Options*» et il faut choisir «*tar the root filesystem*». Pour le compiler, un *make* suffit (et beaucoup de patience suivant la machine !).
L'archive obtenue se trouve dans le repertoire *output/images* de **Buildroot** il suffit de la copier sur un media lisible par le noyau de la carte (clé usb par exemple ou via le réseau) puis de démarrer en ramdisk (section 2.2.2).
Il faut ensuite écraser toutes les données présentent sur la flash puis lui définir un point de montage. Il faut également monter le média ou se trouve l'archive, et la décompresser à la racine de la flash.

Les commandes associées à ces actions sont les suivantes:

```
# flash_eraseall /dev/mtd1[7]
# mkdir /mnt/flash
# mount /dev/mtdblock1 /mnt/flash
# mkdir /mnt/usb
# mount /dev/[NoeudCréeParUdev[8]] /mnt/usb
# tar xvf rootfs.tar -C /mnt/flash
```

Nous pouvons maintenant, avant de démarrer notre nouveau système, ajouter quelques modifications...

### 3.4.4 - Quelques modifications

Le script d'init va lire le fichier */etc/inittab.* Dans celui-ci, il faut vérifier que la sortie série est bien crée avec une ligne qui ressemble à :

```
# Put a getty on the serial port
ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt100 # GENERIC_SERIAL
```

Il faut vérifier aussi les points de montages des périphériques par défaut dans */etc/fstab*, puis la configuration du réseau dans */etc/network/interfaces.* Pour définir une adresse ip statique par exemple, le fichier précedent doit ressembler à :

7. La partition du système étant la 2ème sur notre carte.
8. Pour une clé usb, souvent sda1, si aucun autre périphérique n'a été monté.

```
auto lo
iface lo inet loopback

iface eth0 inet static
	address 192.168.1.2
	netmask 255.255.255.0
	network 192.168.1.0
	broadcast 192.168.1.255
	gateway 192.168.1.1
```

On peut personaliser le premier message affiché à l'écran (avant le login) dans */etc/issue*, et configurer le nom d'hôte dans */etc/hostname*.

Une mise en garde particulière avec notre support de stockage du **rootfs** (ie: la Nand Flash) est qu'il doit subir le moins d'écriture possible. Cependant, sous linux, des journaux s'écrivent très souvent (le syslog par exemple, situé dans */var/log*) c'est pourquoi ces répertoires de destination seront stockés en *tmpfs*, c'est à dire dans un système de fichier temporaire situé dans la mémoire **RAM**.
Pour cela, deux solutions sont applicables : soit monter directement dans le */etc/fstab* le répertoire */var/log* en tmpfs, soit monter un répertoire (*/tmp* par exemple) en **tmpfs** et créer un lien de */var/log* vers */tmp* (c'est la méthode utilisée par **buildroot**).

```
/dev/ttyUSB0 - PuTTY
# mount
rootfs on / type rootfs (rw)
/dev/root on / type yaffs2 (rw)
proc on /proc type proc (rw)
devpts on /dev/pts type devpts (rw)
tmpfs on /tmp type tmpfs (rw)
sysfs on /sys type sysfs (rw)
udev on /dev type ramfs (rw)
/dev/pts on /dev/pts type devpts (rw)
# ls -l
total 6
lrwxrwxrwx    1 root     root            6 Dec 31  1969 cache -> ../tmp
drwxr-xr-x    1 root     root         2048 May 20  2010 lib
lrwxrwxrwx    1 root     root            6 Dec 31  1969 lock -> ../tmp
lrwxrwxrwx    1 root     root            6 Dec 31  1969 log -> ../tmp
lrwxrwxrwx    1 root     root            6 Dec 31  1969 pcmcia -> ../tmp
lrwxrwxrwx    1 root     root            6 Dec 31  1969 run -> ../tmp
lrwxrwxrwx    1 root     root            6 Dec 31  1969 spool -> ../tmp
lrwxrwxrwx    1 root     root            6 Dec 31  1969 tmp -> ../tmp
#
```

La partie sur le **rootfs** est maintenant terminée, un redémarrage du système nous permet de vérifier si celui-ci fonctionne.

## 3.5 - Mise en place d'une machine virtuelle ARM (avec qemu)

### 3.5.1 - Introduction

Pour accélérer le développement d'applications embarquées, il est souvent coutume de configurer et de mettre en place une machine virtuelle correspondant au mieux à l'architecture de destination avec le même système de base.

Nous allons détailler ci-après l'installation et la configuration de notre machine virtuelle. Il est nécessaire de disposer de qemu (disponible dans beaucoup de distribution **Linux**) ; celui-ci est un émulateur de système, libre, qui permet d'exécuter un ou plusieurs systèmes d'exploitations sur une machine hôte.

**Qemu** supporte la virtualisation de machine **ARM**, il est nécessaire par contre de récupérer, ou de compiler une image noyau compatible avec l'émulateur. Nous avons donc trouvé un noyau **2.6.24** (pour s'approcher le plus de celui sur la carte) sur internet (Nous supposerons que l'image noyau s'appelle *vmlinuz*).

### 3.5.2 - Création de l'image du disque

Une fois le noyau récupéré, il est nécessaire de créer une image du disque dur principal. Nous allons en créer une qui fait la même taille que notre **Nand Flash** (ie: 128 Mo). Pour cela il faut initialiser un espace avec la commande :

```
dd if=/dev/zero of=tx52-arm.img bs=1MB[9] count=0 seek=128[10]
```

Il faut ensuite créer un système de fichier sur cet espace :

```
mkfs.ext4[11] -F tx52-arm.img
```

Nous pouvons maintenant monter notre image sur notre système hôte afin de lui copier notre **rootfs** :

```
mkdir tmp
mount -o loop tx52-arm.img tmp
```

Puis :

```
tar xvf rootfs.tar -C tmp/
```

Nous avons donc entre nos mains tous les outils nécessaires pour démarrer notre machine virtuelle.

---

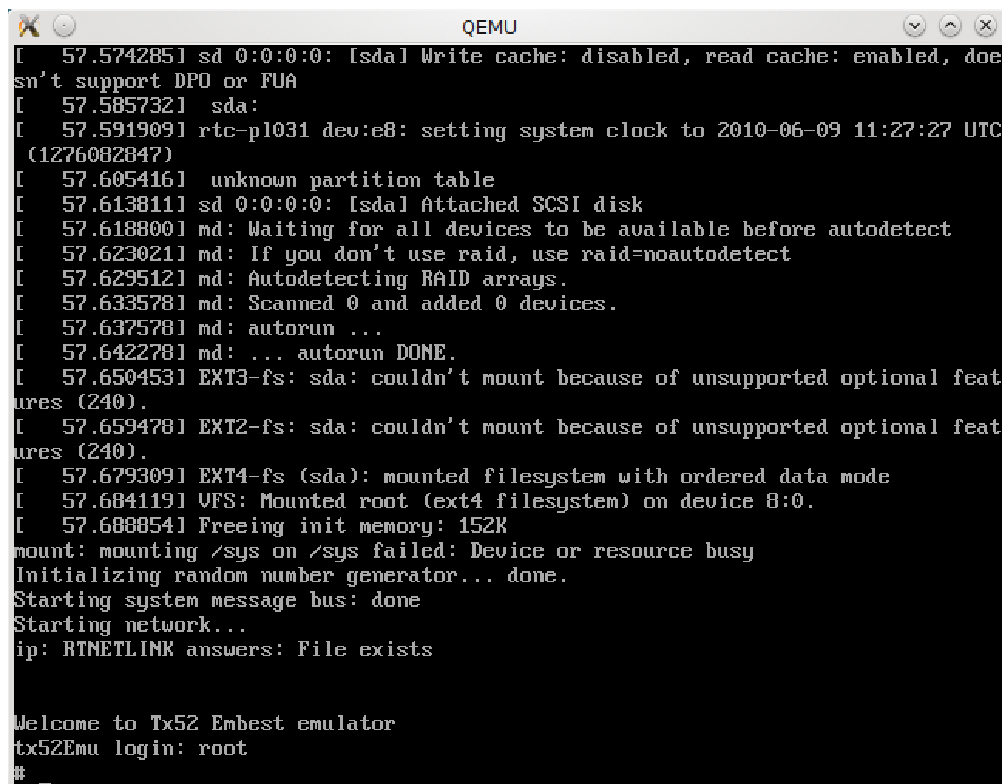
9. bs = blocksize (taille des blocs)
10. 128 blocs de 1Mo = 128 Mo
11. Nous avons choisi ici le système de fichier **Ext4**, car performant, mais le choix est libre, tant qu'il est supporté par le noyau.

### 3.5.3 - Utilisation

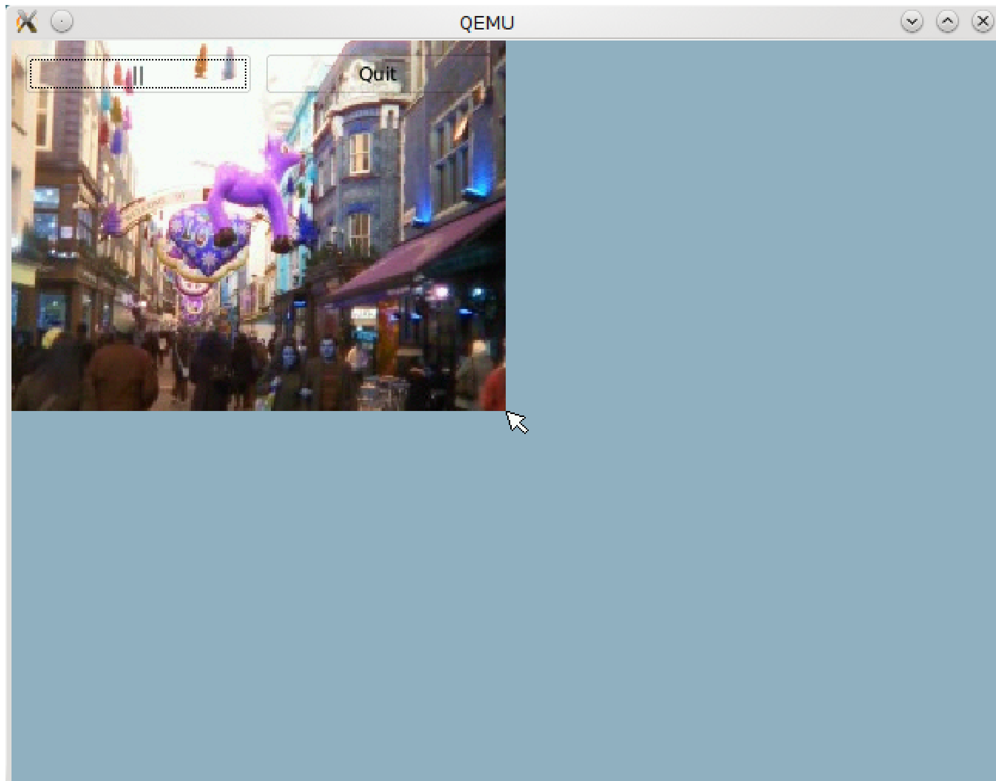
Pour lancer la machine virtuelle, il faut spécifier le chemin vers le noyau, l'image du disque dur principal précédemment créée et éventuellement la taille mémoire allouée au système virtuel (nous allons allouer autant de Ram que disponible sur la carte, soit 64 Mo)

```
qemu-system-arm -M versatilepb -kernel ./vmlinuz -cpu cortex-a8 -hda tx52-arm.img -m 64 -append "root=/dev/sda mem=64M devtmpfs.mount=0 rw"
```

Notre système se lance, voici quelques captures d'écran :



Le *slideshow*, en lançant exactement le même exécutable que sur la carte embarquée :




# 3.6 - L'Application SlideShow

## 3.6.1 - Choix de Qt et cross-compilation

Nous avons choisis d'appuyer le développement du slideshow sur la librairie graphique Qt pour son coté multiplateformes. Cela est pour nous un grand avantage, car de plus, la cross-compilation d'application **Qt** se passe très simplement (avec comme simple pré-requis d'avoir compilé le framework avec les toolchains, dans notre cas, avec **Buildroot**).

**Qt** intègre un générateur de Makefile, appelé qmake, qui permet de configurer l'architecture cible (Par default, l'architecture est celle de l'hôte). Voici un exemple de compilation du slideshow en ne configurant pas qmake avant la génération du Makefile :
*Notre variable d'environnement ne contient pas le chemin vers les toolchains, il est donc impossible d'effectuer de la cross compilation. Il est aussi supposé que la machine hôte possède la librairie Qt installée et qmake.*

```
# qmake
# make
# file <binaireObtenu>
```

```
>> <binaireObtenu>: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
```

Nous pouvons observer que le binaire obtenu est à destination d'une architecture x86-64, et donc ne pourrait pas être exécuté sur la carte **ARM**.

Pour spécifier à qmake de cross-compiler, il faut tout d'abord régler la variable d'environnement PATH pour faire référence aux toolchains de la carte, puis préciser les spécifications de l'architecture cible qui se trouvent dans le répertoire de compilation de **Qt**[12]**.**
Ce qui donne dans notre cas :

```
# make distclean
<export du Path comme indiqué a la section x.x.x>
# qmake -spec /path/to/buildroot/ouput/build/qt-everywhere-opensource-src-4.6.2/
mkspecs/default
# make
# file <binaireObtenu>
>> <binaireObtenu>: ELF 32-bit LSB executable, ARM, version 1, dynamically linked
(uses shared libs), not stripped
```

Nous obtenons bien un binaire exécutable sur plateforme ARM, donc sur notre carte. (On peut par exemple l'envoyer et le tester sur la machine virtuelle avant de l'envoyer sur la carte)

### 3.6.2 - Le développement

Comme nous venons de le voir, nous arrivons à compiler des applications QT pour la plateforme ARM de notre carte assez simplement. C'est la raison pour laquelle nous nous sommes tournés plutôt vers QT qu'autre chose. Au début, nous nous étions orientés vers DirectFB qui permet d'afficher simplement une image. Cependant, QT nous intéresse plus car la documentation est bien plus importante et c'est une plateforme plus répandue.

Notre fonction main prend en argument le chemin de destination des images. La classe principale ImageViewer est exécutée dans la fonction main. Celle-ci liste les fichiers images qui se trouvent dans le répertoire et va les afficher les unes après les autres. Une fois finit, l'application s'arrête. Les images sont affichées en arrière plan dans un objet Qlabel. Afin de changer d'image au bout d'un certain temps, nous utilisons un thread de temporisation chargée de réveiller la classe ImageViewer et charger l'image suivante. Le thread boucle donc sur :

```
emit changePicture();
sleep(sleepTime);
```

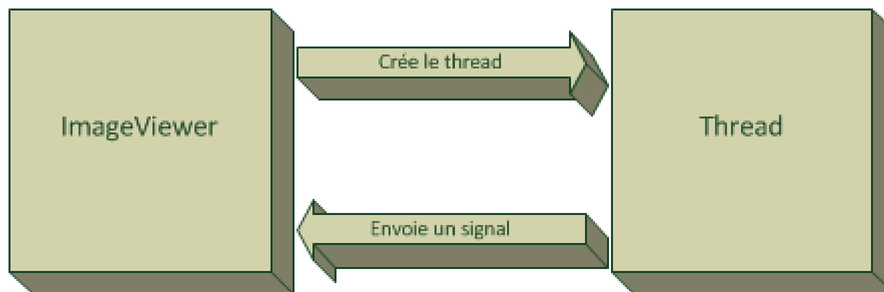
---

12. Si Qt à été compilé avec buildroot, le repertoire est : *CheminBuildroot/output/build/qt-everywhere-opensource-src-4.6.2/mkspecs/default*

L'objet ImageViewer est quant à lui connecté au signal grâce à une fonction statique de la class QObject, Connect() :

```
QObject::connect(ssThread, SIGNAL(changePicture()), this,
SLOT(ChangePic()));
```

Ce qui lui permet de changer d'image à partir d'un ordre du thread.




### 3.6.3 - Auto-lancement à la connection d'un périphérique

Les règles udev ont une syntaxe particulière, mais elles permettent d'effectuer beaucoup d'actions suivant un évènement précis. Notre règle doit s'activer à la détection d'un périphérique usb de stockage (périphérique */dev/sd[a-z][0-9]*) ou à l'insertion d'une carte **SD** dans le slot **MMC** (périphérique */dev/mmcblk[0-9]p[0-9]*). Elle doit monter le périphérique dans un répertoire spécifique et lancer le slideshow avec ce répertoire en argument.
La règle correspondante est :

```
# mmc card :
KERNEL=="mmcblk[0-9]p[0-9]"
ACTION=="add", KERNEL=="mmcblk[0-9]p[0-9]", RUN+="/bin/mkdir -p /mnt/
mmc%n"
ACTION=="add", KERNEL=="mmcblk[0-9]p[0-9]", RUN+="/bin/mount -t auto -o
ro,noauto,noexec,nodev,noatime /dev/%k /mnt/mmc%n"
ACTION=="add", KERNEL=="mmcblk[0-9]p[0-9]", RUN+="/usr/bin/runslideshow
/mnt/mmc%n/",OPTIONS="last_rule"
ACTION=="remove", KERNEL=="mmcblk[0-9]p[0-9]", RUN+="/bin/umount -l /mnt/
mmc%n"
ACTION=="remove", KERNEL=="mmcblk[0-9]p[0-9]", RUN+="/bin/rmdir /mnt/
mmc%n" , OPTIONS="last_rule"
# Any other usb device :
KERNEL=="sd[a-z]", NAME="%k", SYMLINK+="usb%m", GROUP="users",
OPTIONS="last_rule"
```

```
ACTION=="add", KERNEL=="sd[a-z][0-9]", SYMLINK+="usb%n" , GROUP="users",
NAME="%k"
ACTION=="add", KERNEL=="sd[a-z][0-9]", RUN+="/bin/mkdir -p /mnt/usb%n"
ACTION=="add", KERNEL=="sd[a-z][0-9]", RUN+="/bin/mount -t auto -o
ro,noauto,noexec,nodev,noatime /dev/%k /mnt/usb%n"
ACTION=="add", KERNEL=="sd[a-z][0-9]", RUN+="/usr/bin/runslideshow /mnt/
usb%n/",OPTIONS="last_rule"
ACTION=="remove", KERNEL=="sd[a-z][0-9]", RUN+="/bin/umount -l /mnt/usb%n"
ACTION=="remove", KERNEL=="sd[a-z][0-9]", RUN+="/bin/rmdir /mnt/usb%n" ,
OPTIONS="last_rule"
```

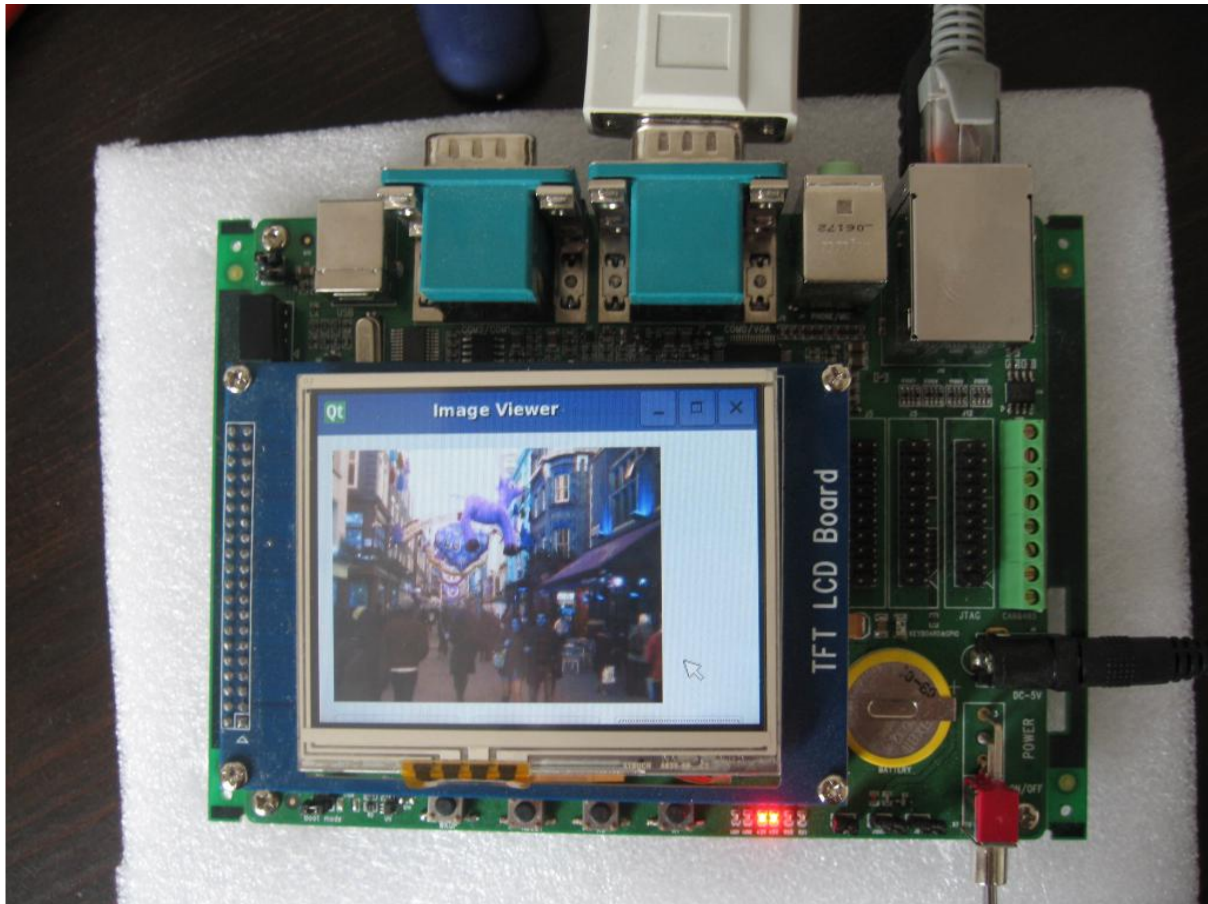
Celle-ci doit se situer dans */etc/udev/rules.d/* et peut se nommer par exemple : *70-automount.rules.*

# 4 - Résultats

## 4.1 - Avec le 2.6.24

Au démarrage, le système de base se lance, et nous obtenons un shell, à la fois sur l'écran LCD et sur la sortie série :

A la connections d'un périphérique usb, le «slideshow» se lance automatiquement.




## 4.2 - Avec le 2.6.32

Malgré le fait que certain composant de la carte ne fonctionnent pas avec cette version du noyau, nous avons quand même pu tirer profit de certaine avancé. Par exemple, la carte dispose d'un port usb de type device, nous permettant d'utiliser des fonctionnalités de type «Gadget Usb». Nous pouvons par exemple, utiliser le cable usb pour simuler une interface réseau (*Usb Gadget ethernet*) ou faire que la carte soit reconnue comme un périphérique de stockage de masse (*Usb Gadget file storage*).

**Ethernet sur Usb**

Sur la Carte, une fois le cable usb branché, nous chargeons le module :

```
# modprobe g-ether
```

Et sur l'ordinateur hôte, le journal système nous indique une nouvelle interface réseau :

```
usb 2-2: new full speed USB device using uhci_hcd and address 2
usb0: register 'cdc_eem' at usb-0000:00:1d.0-2, CDC EEM Device, 96:5c:f5:f9:a9:9f
usbcore: registered new interface driver cdc_eem
```

Nous pouvons donc établir une connection réseau sur l'interface usb0 entre la carte et l'ordinateur hôte, via le câble usb.

## Stockage de Masse via Usb

Il faut charger le module sur la carte avec comme argument un chemin vers le périphérique à partager. Pour l'exemple, nous allons brancher une clé usb sur la carte (le noeud associé sera */dev/sda1*) et la rendre accessible grâce à ce module :

```
# modprobe g-file-storage file=/dev/sda1
```

Sur l'ordinateur hôte, le journal système nous indique un nouveau périphérique :

```
usb 2-2: new full speed USB device using uhci_hcd and address 3
scsi3 : usb-storage 2-2:1.0
scsi 3:0:0:0: Direct-Access     Linux    File-Stor Gadget 0313 PQ: 0 ANSI: 2
sd 3:0:0:0: Attached scsi generic sg2 type 0
sd 3:0:0:0: [sdb] 995967 512-byte logical blocks: (509 MB/486 MiB)
sd 3:0:0:0: [sdb] Write Protect is off
sd 3:0:0:0: [sdb] Mode Sense: 0f 00 00 00
sd 3:0:0:0: [sdb] Assuming drive cache: write through
sd 3:0:0:0: [sdb] Assuming drive cache: write through
 sdb:
sd 3:0:0:0: [sdb] Assuming drive cache: write through
sd 3:0:0:0: [sdb] Attached SCSI disk
```

Nous pouvons accéder à la clé usb directement depuis l'ordinateur hôte, comme si celle-ci était branchée sur ce dernier.

*Note : Les gadgets usb étaient expérimentaux avec la version 2.6.24 du noyau, et ne fonctionnaient pas avec notre carte.*

# 5 - Conclusions

## 5.1 - Benoit

Cette unité de valeur m'a permis de découvrir et d'apprendre beaucoup de chose concernant l'avancé de Linux dans les systèmes embarqués. J'étais déjà à l'aise avec les systèmes du même type, mais la TX m'a permis de me confirmer et de comprendre certains concepts que je n'avais pas assimilé.

J'ai aussi pu me faire une idée de l'état d'avancement des technologies ouvertes dans le domaine de l'embarqué et du besoin actuel et futur de personnes compétentes dans la matière. Par exemple, nous avons utilisé Buildroot, qui commence à être de plus en plus utilisé et de plus en plus stable. Beaucoup de mise à jour ont été effectuée durant la période ou nous l'avons utilisé pour la première fois et jusqu'à la fin de la Tx, montrant bien le développement actif et l'intéressement que subit Linux à destination de l'embarqué.

Je garde donc un excellent souvenir de cette Unité de Valeur et de tout ce dont j'y ai appris.

## 5.2 - Julien

Ce sujet de TX m'a apporté un bon aperçu au niveau du développement pour l'embarqué. C'est avec très peu de connaissance que j'ai entamé ce travail et cela m'a fait découvrir plus en profondeur l'environement unix. Beaucoup de vocabulaire étaient flou avant ce projet, j'ai pu éclaircir certains points et découvrir de nouveaux mots qui me serviront sans doute dans d'autres contextes.

Aussi, le codage de l'application m'a permis de faire de la programmation avec QT mais cette fois ci, avec les contraintes de l'embarqué (mémoire réduite, petit écran, etc). L'utilisation des threads était aussi une chose nouvelle même si finalement son emploi est similaire aux autres langages déjà appris.

## 5.3 - Améliorations possibles

Il reste un certain nombre d'améliorations qui peuvent être ajoutées au développement de la carte. Le travail de portage des sources du **Tsc2301** peut-être terminé et pourquoi pas être soumis aux sources officielles du noyau Linux. Le pilote du lecteur MMC reste très expérimental (surtout le module mmc-spi) et peut necessiter du développement pour le stabiliser.

Le Touchscreen, même avec une calibration est très instable, et mérite aussi de s'y intéresser.

Concernant l'utilisation de la carte, le slideshow peut être améliorer pour utiliser des effets graphiques comme un fondu avant de passer à l'image suivante, ou après la stabilisation du touchscreen, l'utiliser pour changer d'image.

# ANNEXES

## A - Compilation du Noyau Linux

Il faut au préalable récupérer les sources de la version que l'on désire compiler sur www.kernel.org.

Il faut ensuite les décompresser : (suivant le type de compression, la commande diffère)

```
# tar xjvf linux-version.tar.bz2
ou
# tar xzvf linux-version.tar.gz
```

Il faut ensuite lancer le menu de configuration et préciser l'architecture cible :

```
# make menuconfig ARCH=arm
```

*(Le choix des options à compiler ne sera pas détaillé.)*
Nous pouvons maintenant cross-compiler notre noyau, pour cela il faut régler sa variable d'environnement «PATH» avec le chemin vers le cross-compilateur puis:

```
# make ARCH=arm CROSS_COMPILE=arm-linux- bzImage -jxx
```
[13]

Une fois la compilation terminée, il faut transformer l'image binaire obtenue en image compréhensible par u-boot. Celui-ci fournit un outil, lors de sa compilation, appelé «mkimage» qui s'occupe de cette tâche.

Par exemple :

```
./mkimage -A arm -O linux -C none -a 0x20008000 -e 0x20008000 -n linux-<version>-Tx52sbc9261 -T kernel -d ./arch/arm/boot/zImage ./uImage
```

Il faut maintenant compiler les modules :

```
# make ARCH=arm CROSS_COMPILE=arm-linux- modules
```

puis les installer dans un repertoire que l'on copiera sur la carte :

```
# make ARCH=arm CROSS_COMPILE=arm-linux- modules_install
INSTALL_MOD_PATH=./modules/
```

---

13. xx doit être remplacé par le nombre de processeur ou de coeurs disponible sur la machine de compilation.