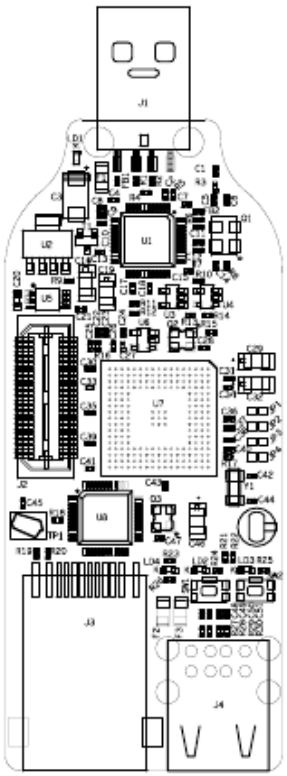
Formation au Noyau Linux

Jérôme Pouiller <j.pouiller@sysmic.org>

Sommaire

- Présentation générale
- Compiler
- Les concepts de développement
- Debugguer
- L'API
- Contribuer

La cible : Calao USB-A9260



La cible : Calao USB-A9260

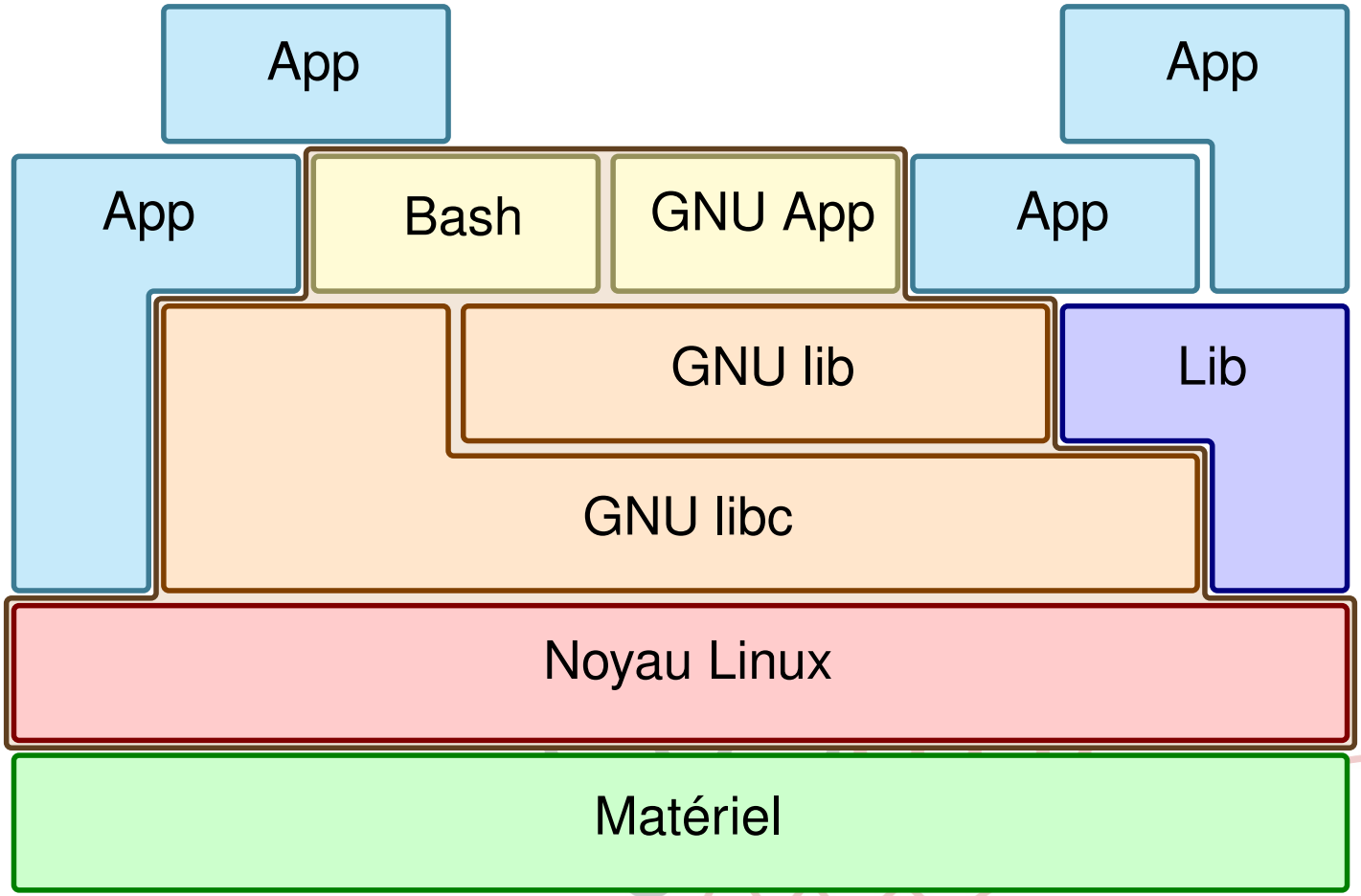
Architecture très classique dans le milieu de Linux embarqué :

- Microcontrolleur Atmel AT91SAM9260
- Core ARM926EJ-S 180MHz
- 64Mo de RAM
- 256Mo de flash
- 64Ko d'EEPROM

Choisie car compacte et très bien supportée par Linux

Composants de Linux

GNU/Linux est finalement un aggloméra :



Elements

Il y a trois éléments important de GNU/Linux :

- Le noyau : Linux
- Le projet GNU et plus particulièrement la glibc, bash et les coreutils
- Les normes : SystemV, Posix, etc...

La Norme Posix

- *Portable Operating System Interface [for Unix]*
- Uniformise les OS
- Première version publiée en 1988
- Souvent implémenté en partie
- ... et parfois s'en inspire simplement
- Posix $\not\rightarrow$ Linux
- Linux $\not\rightarrow$ Posix

Le Projet GNU

- Créé en 1983 par Richard Stallman
- Pose les bases politiques de GNU/Linux
  - GPL publiée en 1989
  - GPLv2 en 1991
  - GPLv3 en 2006
- `gcc` apparait en 1985
- `bash` et les Coreutils apparaissent en 1988 (inspirés de `sh` 1971/1977)
- Nombre d'architectures supportées incalculable

Le noyau Linux

- Créé en 1991 par Linus Torvalds :
  - http://groups.google.com/group/comp.os.minix/browse_thread/thread/76536d1fb451ac60
  - Inspiré de Minix
  - *"just a hobby, won't be big and professional like gnu"*
- Noyau monolithique
  - Débat Tanenbaum–Torvalds
  - http://groups.google.com/group/comp.os.minix/browse_thread/thread/c25870d7a41696d2
  - Modulaire depuis la version 2.6
- Système communautaire
  - La licence GPL à été choisie par pragmatisme et non par conviction
  - Eternels débats d'opinions entre Stallman et Torvalds

Le noyau Linux
Quelques chiffres

- 15 millions de lignes de code dans 30000 fichiers (+15%/an)
- Environ 1200 développeurs dans 600 entreprises (+35%/an)
- Environ 5000 contributeurs depuis la première version de Linux
- Environ 650 mainteneurs (c'est-à-dire responsables d'une partie du noyau)
- 26 architecures (= jeux d'instructions)
- Des centaines de plateformes
- Plus d'un millier de drivers
- Une centaine de versions publiées
- Environ 10000 contributions sur chaque version
- Enormément de "forks" et de version non-officielles
- Domaine d'application très large, du DSP au super-calculateurs en passant pas les grille de calcul

Deuxième partie II

Compiler

Qu'est-ce qu'un BSP ?

- *Board Support Package (BSP)*
- Normalement fourni par l'intégrateur.
- Contient au minimum la toolchain (compilateur, linker, debuggueur) pour la cible, au minimum les sources, et souvent des versions pré-compilées
- Cette toolchain est souvent compilée avec une libc (glibc, μclibc, newlib, bionic, eglibc, dietlibc, klibc, etc...) et une version des binutils. Si ça n'est pas le cas, elle pourra compiler le noyau Linux, mais aucune binaire utilisateur.
- Contient souvent le bootloader (sources et/ou binaire) le noyau Linux (au minimum les sources, et parfois une version pré-compilée).
- Si la cible possède des drivers spécifique externes à Linux, ils doivent (devraient) être fournis avec la toolchain.
- De même, si la cible doit utiliser certaine bibliothèques spécifiques, elle sont normalement fournies
- Contient la documentation (parfois incomplète...)
- Assez souvent, un *rootfs* est fourni

Récupération des sources

Ou récupérer les sources du originales noyau ?

- Utiliser les sources souvent fournies avec le BSP. Il arrive souvent qu'elles contiennent des drivers particuliers et qu'elles soient déjà configurées
- Télécharger sur `kernel.org`

```
host$ wget http://www.kernel.org/pub/linux/
    kernel/v3.x/linux-3.3.tar.bz2
host$ tar xvjf linux-3.3.tar.bz2
```

- Utiliser `git clone`

Versionning

- Au début, le noyau s'incrémentait de deux en deux : 2.0, 2.2, 2.4, etc... Les version impaires indiquait les noyau en développement.
- Chaque version du noyau apportait des ruptures importantes avec la version précédente
- Avec le noyau 2.5 puis 2.6, le noyau est arrivé à une certaine maturité. Les gros changements sont devenus rares et les développements sont devenus de plus en plus itératifs
- Finalement dans la version 2.6, toutes les versions mineures sont stables
- Les versions stables de la 2.6 peuvent recevoir des correctifs et sont alors numérotés sur 4 chiffres : (exemple : 2.6.32.59)
- Le développement d'une nouvelle version du noyau 2.6 en intégrant les patchs provenant des sous-systèmes. Les noyaux produit lors de l'intégration de ses patchs est suffixés par `rcX` (*release candidate*).
- Le développement noyau alterne les fenêtres de *merge* pendant lesquels, les mainteneurs des sous-systèmes envoient leurs développements à Linus Torvalds et fenêtres de stabilisation.

Versionning

- La version 3.0 correspond en fait la version 2.6.40 renommée :
  - Il n'y a eu aucune refonte de l'architecture entre les version 2.6 et 3
  - Pour fêter les 20ans du kernel
  - Pour marquer l'intégration de la branche RT-Preempt dans le mainstream
  - Parce qu'avec le cycle de développement itératif, la version 2.6 ne s'incrémentera jamais. Les version stable quant à elles peuvent recevoir des correctif et se retrouver sur 4 chiffres. Il y avait par conséquent un chiffres en trop. Le passage en 3.Y.Z permettait de revenir sur un modèle classique à 3 chiffres.
  - Le passage en 3.X marque ainssi la stabilisation du cycle de développement du noyau
- Référence : `Documentation/development-process`

Git

- `git` est l'outil de gestion de sources du noyau
- Il est fortement recommander de l'utiliser dans le cadre du développement du noyau
- Il s'agit d'un système de gestion décentralisé.
- Pour expliquer la décentralisation, imaginez que :
  - Un utilisateur duplique un dépôt svn
  - Des modifications sont apportées sur les deux dépôts
  - On essaye de resynchronisez les deux dépôts...
- Considérez git comme un svn capable d'effectuer cette opération très simplement.
- Lorsqu'un utilisateur récupère le code d'un dépôt, il devient lui-même dépôt
- Si l'utilisateur laisse un moyen quelconque d'accès en lecture à son dépôt, d'autres personnes pourrons à leur tour le cloner ou tirer les modification qu'il a effectué
- `http://git.kernel.org` liste les dépôts publiques des principaux développeurs du noyau

Git

- Pour récupérer un dépôt :

```
git clone <depôt>
```

- Quelques dépôts notables :
  - git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git Le dépôt de Linus Torvalds. Il contient les derniers patchs du noyau en développement
  - git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git Le dépôt stable. Contient la dernière version stable du noyau, ainsi les mises à jours des versions stables
  - git://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git Les dépôt une dizaine de sous-projets importants sont mergés automatiquement dans ce dépôt. Utilisé principalement par des robots afin d'identifier les problèmes de merge en amont des phases d'intégrations
  - git://git.kernel.org/pub/scm/linux/kernel/git/tglx/history.git L'historique des versions entre 2.5.0 et 2.6.12.

Git

Récupérons la version stable

```
host$ git clone git://git.kernel.org/pub/scm/linux/
    kernel/git/stable/linux-stable.git
```

Chaque version du noyau est marquée avec un tag

```
host$ git tag
```

Puis il est possible de récupérer une version avec

```
host$ git checkout v3.3
```

Votre dépôt est alors non-modifiable. Vous devez créer une branche de travail :

```
host$ git checkout -b mybranch v3.3
```

Organisation des sources

On remarque qu'il existent d'énorme différences de tailles entre les répertoires :

```
$ du -s */ | sort -n | column
40    usr            4092  scripts
156   samples        5248  kernel
160   init           6436  firmware
180   virt           19524 Documentation
232   ipc            21332 net
884   block          22728 include
1904  crypto         24016 sound
1932  lib            32436 fs
2076  security       121584 arch
2400  mm             252688 drivers
3536  tools
```

Organisation des sources

- Les services indispensables à un OS :
  - `kernel` Le scheduler de tâches, les frameworks de gestion des IRQ, le loader de binaires, diverses autre fonctionnalités ne rentrant dans aucune autre catégories.
  - `mm` Le gestionnaire de mémoire. Considéré comme la partie la plus complexe du noyau
  - `init` La fonction de démarrage du noyau (`start_kernel`).
  - `arch` Le code spécifique à chaque architecture. En particulier, le code nécessaire au boot, à la génération des images, les routine assembleur pour la gestion des interruptions, etc...
  - On peut estimer que l'intelligence réside principalement dans `kernel` et `mm`. On remarque cette partie est finalement relativement petite dans le noyau.

Organisation des sources

- Les services supplémentaires du noyau :
  - `fs` Les systèmes de fichiers : ext3, NTFS, NFS, etc...
  - `net` La gestion du réseau et plus particulièrement la stack IP
  - `ipc` La gestion des communication inter-processus (shmem, mq, etc..)
  - `security` Les frameworks de gestion de la sécurité (selinux, apparmor, etc..)

Organisation des sources

- Les drivers
  - `drivers/*` Tous les autres drivers et frameworks de développements. De loin la plus grosse partie du code du noyau.
  - `sound` Les drivers de cartes son.
  - `firmware` Certains périphériques nécessitent l'upload d'un firmware pour s'initialiser. `firmware` contient les binaires de ces firmwares (sans les sources !). La politique de Linux au sujet du code de ces firmwares est qu'il s'agit de données d'initialisation de ces périphériques et que par conséquent, ils ont leurs place dans l'arborescence du noyau.
  - `block` Le framework des périphériques de *block*.
  - `virt/kvm` Le framework de virtualisation kvm.

Organisation des sources

- Des bibliothèques utilitaires :
  - `lib` Divers utilitaires
  - `crypto` Fonctions utilitaires relatives à la cryptographie
- Le code annexe :
  - `include` Les headers exposés du noyau.
  - `scripts` Les scripts ou les programmes utilitaires nécessaires à la compilation ou à l'exploitation du noyau
  - `usr` Script nécessaire à la génération des *initramfs*
  - `tools` Les outils permettant la communication de certains frameworks avec le noyau (particulièrement `perf`)
- La documentation
  - `Documentation` La documentation
  - `sample` Des exemples de code pour certains frameworks

Les architectures

Regardons `arch/` de plus près :

- Les PC : `x86`
- Les workstations et les serveurs : `alpha`, `sparc` (Sun), `ia64` (Intel), `powerpc`, `s390` (Mainframes d'IBM) `parisc` (Workstation HP)
- L'embarqué : `arm`, `mips`, `sh` (STMicroelectronics), `avr32`, `m68k`, `score`, `mn10300`, `m32r` `h8300`
- Les architectures dédiées : `cris` (Embedded Network), `frv` (Futjisu, Traitement d'image)
- Les DSP : `c6x` (Texas Instrument), `heaxgon` (Qualcomm), `blackfin`
- Les softcores : `microbaze`, `xtensa`
- Les expérimentaux `tile` (Architecture distribuée), `unicore32` (Université de Pekin), `openrisc`
- User Mode Linux : `um`

On retrouve nos 26 (+1) architectures supportées

Le code spécifiques aux architectures

- On retrouve dans les sous-répertoire de `arch/` certains répertoires de la racine.
- `mach-`, `plat-`, `plateforms` contiennent du code spécifique à un type de plateformes : Ti Omap, Atmel AT91, PowerPC 85xx
- On pourra trouver des fichiers spécifique aux *board*. Particulièrement vrai pour les architectures non plug-and-play qui nécessite que les périphériques soient déclarés manuellement
- `include/` contient des headers spécifiques à l'architecture. Lors de la compilation, un lien symbolique sera créé entre `arch/<ARCH>/include/asm` et `include/asm`. Ce lien permet au noyau de s'abstraire de la plateforme
- `boot/` contient le code nécessaire au démarrage de la cible :
  - Le code du *bootloader* et les scripts associés permettant la décompression du noyau en mémoire
  - Les scripts nécessaire à la génération d'une image au format du bootloader, du flasher ou de la sonde JTAG

Fonctionnement de Kconfig

- Système de compilation du noyau
- Application de la règle : “Pas générique mais simple à hacker”
- Dépend principalement de `gmake`
- Pas un système de compilation réel. Composé de :
  - Kconfig : Système de gestion de configuration
  - Kbuild : Ensemble de règles de Makefile bien pensées
- Adapté aux environnements proposant beaucoup d’options de configuration
- Très bien adapté à la cross-compilation
- Utilisé dans d’autre projets : μclibc, busybox, buildroot (tous dans le milieu de l’embarqué)

Le système de compilation

- Pour obtenir de l'aide sur les différentes cibles :

```
host$ make help
```

- La variable `ARCH=` spécifie l'architecture cible à utiliser. Elle impacte les options du noyau. Comparez `make help` avec `make ARCH=arm help`.
- Si `ARCH` n'est pas spécifiée, Kconfig utilise l'architecture *host*.
- Lorsque vous avez commencé à spécifier `ARCH`, vous devez toujours la spécifier.
- Il est toutefois possible de placer cette variable dans le Makefile racine ou dans l'environnement pour éviter de l'oublier.

```
host$ export ARCH=arm
```

Travailler avec les configurations

- `make help` propose des configurations préétablies. Il est possible d'importer une de ces configuration :

```
host$ make ARCH=arm usb-a9260_defconfig
```

- Kconfig sauvegarde la configuration dans `.config`.
- Le fichier `.config` sera ensuite :
  - Sourcé dans les systèmes de Makefile
  - Transformé en `include/generated/autoconf.h` et inclut dans les headers de compilations
- Certains constructeur vous fournirons un patch ajoutant une cible `_defconfig`
- ... d'autres vous fournirons un `.config`

Travailler avec les configurations

- Lorsque votre fichier `.config` n'est parfaitement compatible avec vos sources (import, mise à jours des sources, édition manuelle, ...), il recommandé (nécessaire ?) de lancer

```
host$ make oldconfig
```

  `oldconfig` vous indique d'éventuelles incompatibilité entre votre configuration et vos sources et vous demande votre avis sur les nouvelles options
- Pour répondre systématiquement avec la réponse par défaut :

```
host$ yes "" | make oldconfig
```

- Obtenir la liste des nouvelles options par rapport à votre configuration :

```
host$ make listnewconfig
```

- Vous pouvez normalement trouver la configuration du noyau de votre *host* dans `/boot/config-`\`uname -r\`
- Sauver votre configuration en effectuant un *sanity check*

Configurer le noyau

Pour configurer les options :

- En ligne de commande (inutilisable pour un humain)

```
host$ make config
```

- En ncurses

```
host% apt-get install libncurses5-dev
host$ make menuconfig
```

- Avec la nouvelle version de ncurses

```
host% apt-get install libncurses5-dev
host$ make nconfig
```

Configurer le noyau

- En Qt4

```
host% apt-get install libqt4-dev
host$ make xconfig
```

- En Gtk

```
host% apt-get install libglade2-dev
host$ make gconfig
```

- Dans toutes les interfaces, il est possible d'obtenir de la description sur l'élément sélectionné (<h> ou <?>)
- Il est possible de rechercher dans les descriptions des éléments (</>)
- Dans la recherche et dans l'aide, vous trouverez des informations sur les dépendances entre les options
- Les script `scripts/config` permet de changer les configuration à la main

Les cibles de compilation

- La compilation du noyau se lance juste avec

```
host$ make
```

- Le système choisi les cible appropriée en fonction de votre architecture (principalement, une image et les modules)
- Il est souvent préférable (nécessaire ?) de spécifier le type d'image voulue avec

```
host$ make XXImage
```

- `XX` fait référence au format de la binaire produite :
  - Le code commence=t=il au premier octet ?
  - Respecte-t-il le format ELF ?
  - Y a-t-il un format particulier d'entête à respecter ?
- Dans le doute, il faut consulter la documentation de votre bootloader

Options de Kbuild

Certaines options peuvent être passées sur la ligne de commande afin de modifier le comportement général du système.

- `ARCH=` spécifie l'architecture à utiliser. (Nous l'avons déjà vu)
- `CROSS_COMPILE=` spécifie le préfixe de la toolchain. Ainsi, si vous compilez avec `/opt/arm/usr/bin/arm-linux-ulibc-gcc`, vous devez spécifier : `CROSS_COMPILE=/opt/arm/usr/bin/arm-linux-ulibc-`. Par commodité, on préférera ajouter `/opt/arm/usr/bin` à la variable d'environnement `PATH`. Il est aussi possible de configurer `CROSS_COMPILE` par Kconfig
- `V=1` permet d'afficher les commandes lancées par le système de compilation plutôt que la version abrégée. Pas très lisible lors des compilations parallèles mais indispensable pour comprendre certaines erreurs de compilation
- De base, le noyau n'active que les warnings utiles (ainsi, les warnings produits sont rarement à ignorer). `W=[123]` permet d'activer des warnings supplémentaires

Options de Kmake

- `C={1,2}` lance l'outil sparse sur les sources. Nous y reviendrons.
- `-jX` est une option de `make` qui permet de lancer `X` compilation simultanées. Grosso modo, `X` devrait être plus ou moins votre nombre de coeurs CPU.
- `O=` permet de compiler *out-of-source* :

```
host$ mkdir build
host$ make ARCH=arm CROSS_COMPILE=arm-linux- O=
    build menuconfig
```

Tous les fichier issus de la génération seront placés dans `build`. Une fois que votre configuration est crée, vous pouvez lancer `make` directement à partir de `build`. La compilation *out-of-source* permet une grande souplesse de développement et est fortement recommandée.

- Référence : `Documentation/kbuild/kbuild.txt`

Résultats de la compilation

Fichiers produits (ou productibles) par la compilation :

- `vmlinux` : L'image ELF du noyau. Lisible par les debugueurs, certains flasheurs, certain bootloaders
- `vmlinuz` : parfois équivalent du `bzImage`, mais normalement, il s'agit de`vmlinux` compressé et strippé des informations inutiles au démarrage. Inutilisable dans l'état, il est nécessaire de lui adjoindre un bootloader pour le décompresser et l'exécuter.
- `Image` : `vmlinux` strippé et préfixé par un mini-bootloader permettant de sauter sur la fonction `start_kernel` de `vmlinux`.
- `bzImage` et `zImage` : `vmlinuz` avec le bootloader `bz2` ou `gz`.
- `xipImage` : Idem `Image` mais destiné à être exécuté directement sur un *eeprom* sans être copier en mémoire au préalable.
- `uImage` : `Image` avec une entête spéciale pour *u-boot*.

Le format S3

Il est possible de générer des image au format SRecord en utilisant `objcopy`

```
host$ objcopy -O srec vmlinux vmlinux.srec
```

Les modules

Une grosse partie du noyau peut être compilé directement dans le noyau ou sous forme de modules.

- Ils sont marqués par < > (non-compilé), <*> (linké en statique) ou <M> (compilé en module).
- Les modules peuvent être apparentés à des plugins pour le noyau. Il peuvent être chargé et déchargés dynamiquement.
- Les modules doivent être présent sur la cible
- Les modules permettent d'alléger la taille (et améliorer les performances) du noyau et évitent de redémarrer la cible lors du développement
- Les modules ne peuvent être chargé qu'après le démarrage du noyau. Par conséquent, certaines fonctionnalités ne peuvent pas être sous forme de modules
- Les drivers nécessaire au chargements des modules ne peuvent pas être des modules. Ainsi, si vos modules sont sur une flash, tous les drivers nécessaires à l'accès à cette flash doivent être statiques
- Il est possible de développer des modules en dehors de

Les modules et l'installation

- `make modules` permet de compiler les modules
- `make INSTALL_MOD_PATH=$(pwd)/../target modules_install` copie les modules dans `$INSTALL_MOD_PATH` (= dans le rootfs de la cible)
- `make modules_prepare` prepare les sources pour que les modules extérieur puissent compiler
- `make INSTALL_PATH=$(pwd)/../target install` appelle `arch/$ARCH/boot/install.sh` qui appelle `~/bin/installkernel` ou copie le noyau dans `$INSTALL_PATH`
- `make *-pkg` crée des packages (naïfs) pour diverses distributions. Ces packages contiennent le noyau et les modules.
- `make INSTALL_HDR_PATH=$(pwd)/../BSP/include headers_install` copie les headers dans `$INSTALL_HDR_PATH`. Ces headers sont suffisants pour compiler les programmes de l'espace utilisateur. Si votre toolchain est correctement compilée, vous ne devriez pas en avoir besoin.

Clean

- `make clean` Supprime les fichier objets (qui ne sont plus utiles une fois le noyau compilé)
- `make mrproper` Supprime tous les résultats de la compilation ainsi que les fichiers de configuration
- `make distclean` Supprime les résultats de compilation, les configurations et fichiers originaires de l'intégration de patchs ou de l'édition de fichiers (`*~`, `*.orig`, `*.rej`, etc...)

Configuration globale

General setup :

- *Prompt for development and/or incomplete code/drivers* : Débloque les options de compilation pour les drivers/option instables (staging, etc...)
- *Cross-compiler tool prefix* : Affecte la variable `CROSS_COMPILE`
- *Local version* : Ajoute un identifiant à la version. Indispensable dans les phases d'intégration. La version peut être lue dans `/proc/version`. Il est aussi possible de faire `make kernelrelease` dans un répertoire de compilation du noyau.
- *Automatically append version information* : Ajoute l'identifiant git à la version. Indispensable dans les phases de développement
- *Kernel compression mode* : Permet de choisir le type de compression. Chaque algorithme a ces inconvénients et ses intérêts.
- `SWAP` : Permet de gérer un espace d'échange dur un disque

Configuration globale

- `SYSVIPC` et `MQUEUE` : Communication inter-processus définis par Posix
- `IKCONFIG` : Embarque le `.config` dans le noyau
- `EXPERT` et `EMBEDDED` Débloque les options permettant principalement de réduire la taille du noyau en supprimant des modules importants
- `CC_OPTIMIZE_FOR_SIZE` : Compile avec `-Os`
- `KPROBES`, `PERF_EVENTS`, `PROFILING`, `GCOV_KERNEL` : Active les différentes instrumentations du noyau

Les périphériques de block

`MODULES` : Active la gestion des modules

`BLOCK` : Il est possible de désactiver la gestion des périphérique de block si votre système n'utilise que de la mémoire flash.

- *IO Schedulers* : Permet de choisir un ordonnanceur d'E/S différent de celui proposé en standard

System type :

- Permet de choisir le type d'architecture et de chipset
- Il est possible de désactiver certains cache lors des phases de développement
- Vous trouverez aussi dans ce menu les options relative au jeu d'instructions accepté

Options de l'horloges

Kernel features

- `HZ` (pas sur ARM) : Définit l'intervalle de réordonnancement de l'ordonnanceur. Plus cette valeur est forte, plus l'overhead introduit par le changement de contexte est important et plus les temps de réponses des tâches sont courts
- `NO_HZ` : Permet de rendre la période de réordonnancement des tâches dynamique. Devrait permettre un léger gain de CPU (finalement négligeable avec l'ordonnanceur en $o(1)$). Permet surtout de gagner en consommation électrique.
- `HIGH_RES_TIMER` : Gère les timers avec une horloge différente de l'ordonnanceur (l'horloge est alors géré comme un périphérique à part). Permet d'obtenir une bien meilleure précision sur les mesure de temps, à condition que votre matériel possède une horloge *HighRes*.

Options de l'ordonnanceur

- *Preemption Model* : Permet d'activer la préemption du noyau. Le pire temps réponse sont améliorés, mais le temps moyen est généralement moins bon. Un noyau préemptif stresse beaucoup plus de code. Ne pas activer si vous utilisez des drivers extérieur non garanti pour cette option.
- `RT_PREEMPT` (sur certaines architectures seulement) : Permet de threader les IRQ et ainsi de remplacer les spinlock par des mutex. Ajoute un protocole d'héritage de priorité aux mutex. Le kernel devient alors totalement préemptif. A n'utilisez que lors d'application temps réelle. Etudiez des solutions à base d'hyperviseurs.
- Ne confondez pas la préemption du noyau avec la préemption des tâches utilisateur.

Option de gestion de la mémoire

- `EABI`, `OABI`, etc... : Différentes format d'appel des fonctions. Spécifique à ARM (mais très important)
- *Memory Model* : Permet de gérer les futurs systèmes à mémoire asymétriques entre les CPU
- `COMPACTION` : Permet de compresser les page de mémoire plutôt que les mettre en swap. Particulièrement utile dans les systèmes sans swap !
- `KSM` : Permet de fusionner les page mémoire identiques. Uniquement utile avec des machines virtuelles ou des chroot. Sinon, les noyau sait que le fichier est déjà en mémoire et ne duplique pas la page

Configuration du boot et du FPE

Boot options :

- *Flattened Device Tree* : Utilise *OpenFirmware*, le nouveau format de description matériel appelé aussi *Flatten Device Tree*
- *Default kernel command string* : Permet de passer des paramètres par défaut au noyau (nous verrons cela un peu plus loin)
- *boot loader address* : Permettent de démarrer le noyau à partir d'une ROM, d'une MMC, etc...
- *Kernel Execute-In-Place from ROM* : Permet d'exécuter un noyau non compressé à partir d'une ROM

Floating point emulation : Si une instruction sur des nombres à virgule flottante est rencontrée et ne peut pas être exécutée, le noyau peut alors émuler l'instruction (voir aussi `-msoft-float`)

Configuration réseau

Networking :

- Possibilité d'activer les innombrables protocoles réseaux de niveaux 1, 2 et 3
- *Network options* : Beaucoup de fonctionnalité réseau : client dhcp, bootp, rarp, ipv6, ipsec, les protocole de routage, gestion de QoS, support des VLAN, du multicast,
- *Unix domain sockets* : Les sockets *UNIX* (cf. sortie de `netstat`)
- *TCP/IP networking* : Les sockets bien connue *TCP/IP*
- *Netfilter* : Le firewall de Linux. D'innombrable options. Permet l'utilisation d'iptables si l'option `IPTABLES` est active.

Configuration des systèmes de fichiers

File systems :

- *Second extended*, *Ext3 journalling file*, *The Extended 4 filesystem* : Le file system standard de Linux
- *FUSE* : Permet de développer des systèmes de fichiers en espace utilisateur
- *Pseudo filesystems* Systèmes de fichiers sans supports physiques
  - `TMPFS` : File system volatile en RAM. Très utilisé avec des système en flash vu que l'accès à la Flash est coûteux en temps et destructeur pour la flash
  - `SYSFS` et `PROC_FS` : Permettent au noyau d'exporter un certain nombre de donnée interne vers le userland. Beaucoup d'outils système tirent lors informations de ces systèmes de fichiers. Ils doivent être montés dans `/sys` et `/proc`. `/proc` est plutôt orienté processus alors que `/sys` est orienté modules et paramétrage du noyau.

Configuration des systèmes de fichiers

- *Miscellaneous filesystems* Contient des systèmes de fichiers spécifiques
  - *eCrypt filesystem layer* : Gestion transparent d'un file system codé
  - *Journalling Flash File System v2* : Spécialisé pour les Flash avec gestion de l'écriture uniforme, des *bad blocks* et des *erase blocks*.
  - *Compressed ROM file system* : Spécialisé pour ROM sans accès en écriture.
  - *Squashed file system* : Idem *cramfs* mais fortement compressé
- *Network File Systems*
  - *NFS client support* : File system sur ethernet. Très utilisé dans l'embarqué durant les phases de développement
  - *Root file system on NFS* : Permet de démarrer le noyau sur une partition NFS

Configuration des Drivers

Device Drivers Des centaines de drivers. Notons :

- *path to uevent helper* : Le programme apellé lorsqu'un nouveau périphérique est détecté (cf. `/proc/sys/kernel/hotplug` et `/sys/kernel/uevent_helper`)
- *Maintain a devtmpfs filesystem to mount at /dev* : Un tmpfs spécifique pour les devices automatiquement monté sur `/dev`. Les fichiers devices sont alors automatiquement créés sans l'aide d'un programme extérieur.
- *Memory Technology Device* : Les flashs
- *Staging drivers* : Des drivers en cours de bêta

Configuration du noyau

Mais aussi :

- *Kernel Hacking* : Options concernant le débugging du noyau.
- *Security Options* : Plusieurs framework permettant de gérer des droits plus fin sur les programmes exécutés et/ou de garantir l'intégrité des donnée à l'aide de TPM.
- *Cryptographic API* : Fonctions de cryptographies sélectionnées automatiquement par d'autres modules (particulièrement les protocoles réseaux)
- *Library routines* : Idem *Cryptographic API* mais avec principalement des calculs de checksum.

Boot par tftp/nfs

Pour le développement du noyau, il est commun d'utiliser les technologies :

- *tftp* : Il s'agit d'un protocole de transfert de fichier très simple. Beaucoup de bootloaders l'implémentent. Il permet un démarrage rapide d'un nouveau noyau lors du développement. On pourra aussi trouver des protocole sur RS232 ou sur USB permettant la même fonctionnalité. Comme pour un démarrage normal, on indique au bootloader à quelle adresse le noyau doit être chargé et on *jump* à cette adresse
- *nfsroot* : On demande au noyau de monter une partition réseau. On dispose ainsi d'un espace de stockage illimité et il est simple et rapide de mettre à jour le rootfs.

Notre cas

Dans notre cas, nous utilisons U-Boot (standard)

- Compilation

```
host% apt-get install uboot-mkimage
host$ make O=build ARCH=arm usb-a9260_defconfig
host$ make O=build ARCH=arm CROSS_COMPILE=arm-
    linux- -j3 uImage
```

- Partage de l'image par TFTP

```
host% cp build/arch/arm/boot/uImage /srv/tftp/
    uImage-2.6.33.7
host% ln -s uImage-2.6.33.7 /srv/tftp/uImage
```

- Au redémarrage, le bootloader passe par un registre l'identifiant de la carte. Cet identifiant (spécifique à l'architecture ARM) est erroné. A ce stade, il est plus facile de corriger ce problème dans le noyau dans le fichier `arch/arm/tools/mach-types`.

Passage d'options au noyau

- Il est possible de passer des options au démarrage du noyau
- C'est normalement le bootloader qui se charge de passer la ligne de commande au noyau
- Le bootloader utilise un protocole prédéfini (lorsqu'il donne la main au noyau, un des registre contient un pointeur sur la ligne de commande)
- Il est possible de surcharger la ligne de commande lors de la compilation avec l'option `CMDLINE`
- Les diverses options acceptées sont décrites dans `Documentation/kernel-parameters.txt`
- Il est possible d'accéder à la ligne de commande après le démarrage dans `/proc/cmdline`
- Il existe des paramètres globaux au kernel et des paramètres spécifiques à un module. Lorsque le module est compilé dans en statique, il est possible de lui passer des paramètres avec la syntaxe `<MODULE_NAME>.<PARAM>=<VALUE>`
- Beaucoup d'options sont modifiable à posteriori par `/sys`

Le rootfs

- `root=` indique le disque à monter sur `/`
- Sûrement l'option la plus utilisée
- Il est possible de spécifier le nom d'une partition. Par exemple `root=/dev/sda1` (PC) ou `root=/dev/mtd0` (partition flash)
- `root=/dev/nfs` demande au noyau de démarrer sur NFS
- Remarque : la partition `/` n'étant pas encore montée, ces nom de partition ne correspondent pas à des fichiers existants dans `/dev`. Le kernel utilise simplement la même syntaxe.

La configuration réseau

- Il est possible d'initialiser le réseau avant de montage du rootfs
- Indispensable pour le démarrage par NFS
- Syntaxe : `ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>`
- Exemple : `ip=192.168.1.72:::::eth0:`
- Pour le démarrage par nfs, il est aussi nécessaire de spécifier le répertoire partagé par le serveur : `nfsroot=192.168.1.10:/srv/nfs`
- Il aussi possible de démarrer en utilisant un DHCP (ou un autre protocole d'auto-négociation) : `ip=on`

La configuration réseau

- Il est alors possible de spécifier le `nfsroot` dans la configuration du serveur DHCP :

```
host target {
    option root-path "192.168.1.10:/srv/nfs";
    next-server 192.168.1.10;
    hardware ethernet 00:26:24:3a:14:5c;
    fixed-address 192.168.1.72;
}
```

- Lors du démarrage NFS, attention aux modifications de la configuration réseau postérieure au montage du rootfs
- Référence :
  `Documentation/filesystems/nfs/nfsroot.txt`

Démarrage du noyau

- A la fin du démarrage du noyau, celui donne la main à l'exécutable déclaré avec `init=`. Par défaut, il s'agit de `/sbin/init`
- `init` ne se termine jamais
- Les arguments non-utilisés par le noyau sont passé à `init`
- On peut estimer que notre système démarre à partir du moment ou nous obtenons un shell (c'est en tous cas la que la plupart des intégrateur Linux embarqué s'arrêteront)
- Du moins complexe au plus complexe à démarrer :
  - `init=/hello-arm-static`
  - `init=/hello-arm`
  - `init=/bin/sh`
  - `init=/sbin/init`

Effectuons ces tests avec le Rootfs original et un Rootfs vierge.

La console

- Sûrement la deuxième option la plus utilisée
- `console=` permet de demander au noyau d'afficher la sortie de `printk` sur un périphérique spécifique.
- Sur PC, souvent limité à `console=ttyS0,115200n8`.
- Sur un système embarqué, il existe beaucoup de driver de ports séries différents. `console` peut alors prendre des valeurs exotiques. A voir au cas par cas pour chaque driver.

Panic

- Un *kernel panic* est une erreur détectée mais irrécupérable
- `panic=X` permet demande au noyau de redémarrer après *X* secondes en cas de *kernel panic*
- Par défaut, le noyau ne redémarre pas en cas de *kernel panic*

La mémoire

- Il est parfois nécessaire de donner des instructions au kernel sur l'utilisation qu'il peut faire des espaces mémoires
- `mem=nn` Force la taille de la mémoire que le noyau peut utiliser. Sur beaucoup de plateformes, le noyau n'est pas capable de détecter la capacité de la mémoire.
- `memmap=nn@ss` et `memmap=nn$ss` Force le noyau à n'utiliser que la mémoire entre `ss` et `ss+nn`
- `memmap=nn@ss` et `memmap=nn$ss` Force le noyau à ne pas utiliser la mémoire entre `ss` et `ss+nn`
- Il est possible d'utiliser plusieurs fois ces options
- Utiles pour rapidement réservés des espace d'adresse d'E/S pas encore défini dans la configuration ou se réserver des blocs de mémoire particuliers pour la communication avec d'autres périphériques

Etape de fabriquation d'un BSP

- Commencez par compiler une toolchain si celle-ci n'est pas fournie.
- Si le bootloader est fourni, travaillez avec celui-ci. Il arrive souvent que les bootloader initialisent certains paramètres du CPU sans lesquels le noyau ne peut démarrer
- Si vous n'avez pas de bootloader, vous devrez commencez par paramétrer une sonde JTAG
- Sinon, vous pouvez développer le noyau et le bootloader en parallèle. Sans bootloader, vous devrez utiliser une sonde JTAG pour démarrer votre cible
- Configurer le noyau jusqu'à pouvoir démarrer l'espace utilisateur
- Compiler un busybox pour l'espace utilisateur
- Faites fonctionner les différents périphériques
- Intégrer votre toolchain, votre bootloader, les fichiers de configuration de votre noyau et de votre busybox, vos outils et drivers spécifique dans un outil tel que BuildRoot
- Zipper l'ensemble
- Compilez l'ensemble et zipper le résultat