# Formation à Linux Embarqué

Jérôme Pouiller <j.pouiller@sysmic.org>

# Sommaire

- Administration d'un Linux embarqué
- Création d'un Linux
- Le noyau
- Debug et instrumentation

# Première partie I

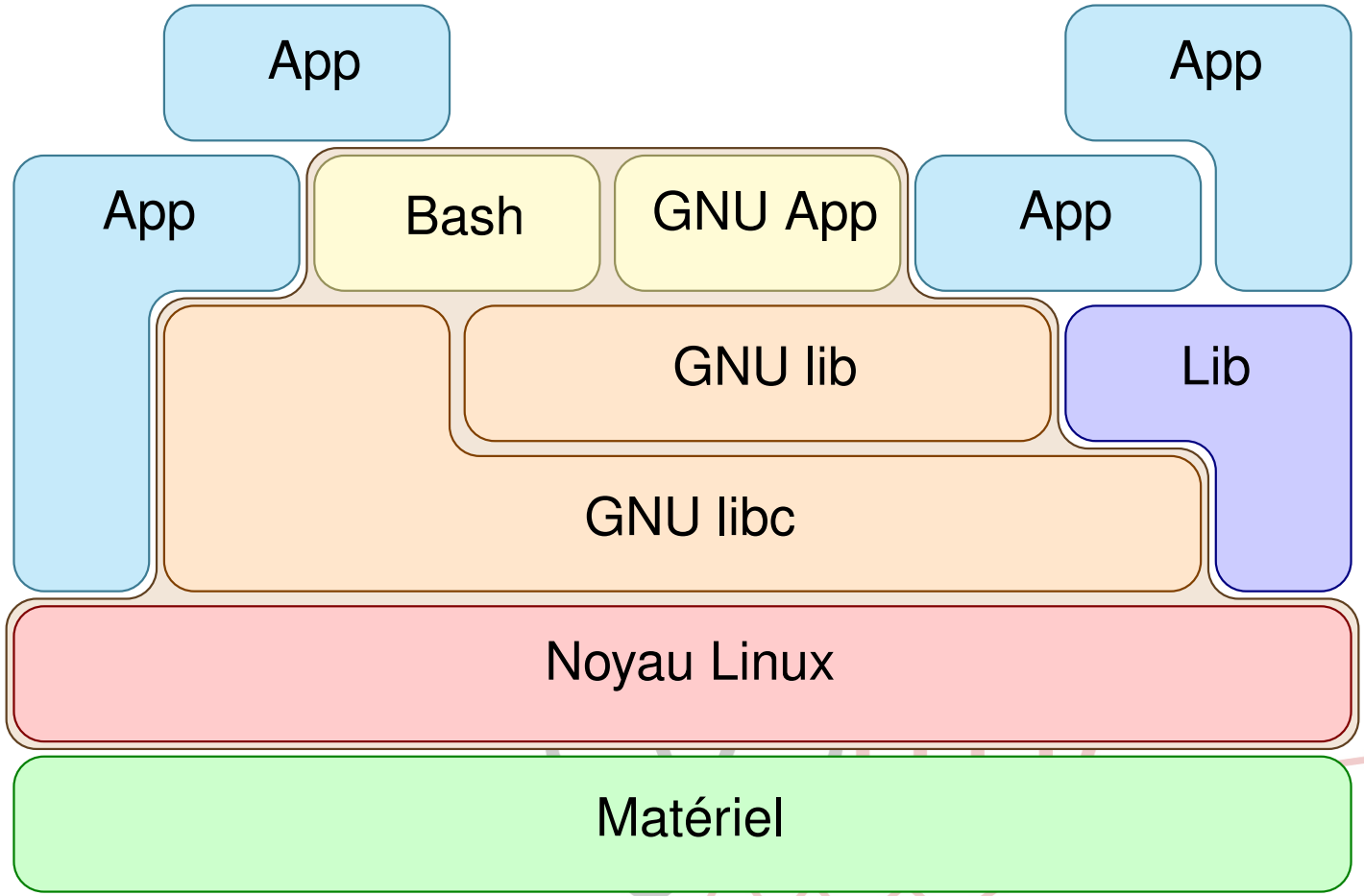
## Administrer

# Qu'est-ce que Linux ?

- Linux ne désigne que le noyau
- Linux est souvent associé aux outils GNU d'où le nom de GNU/Linux
- Systèmes avec les outils GNU mais un noyau différent : GNU/Hurd, Solaris, etc...
- Systèmes Linux sans GNU : Android
- Le nombre de systèmes Linux installés est difficile à évaluer (en partie à cause des système Linux embarqués)

# Composants de Linux

GNU/Linux est finalement un aggloméra :



# La Norme Posix

- *Portable Operating System Interface [for Unix]*
- Uniformise les OS
- Première version publiée en 1988
- Souvent implémentée 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
- Système communautaire
- 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 responsbales d'une partie du noyau)
- Domaine d'application très large, du DSP au super-calculateurs en passant pas les grille de calcul
- 24 architectures (= jeux d'instructions)
- Des centaines de plateformes
- Environ 1000 drivers
- Une centaine de versions publiées
- Environ 10000 contributions sur chaque version
- Enormément de "forks" et de version non-officielles

# Qu'est-ce qu'une distribution ?

- Debian, Ubuntu, Meego, Red Hat, Suse, ...
- Compilations de programmes disponibles pour GNU/Linux
- Ensemble de normes et de procédure
- Permet de garantir le fonctionnement des programmes distribué
- Notre distribution "Hôte" : Ubuntu
- Notre distribution "Cible" : Aucune

# Qu'est-ce que l'embarqué ?

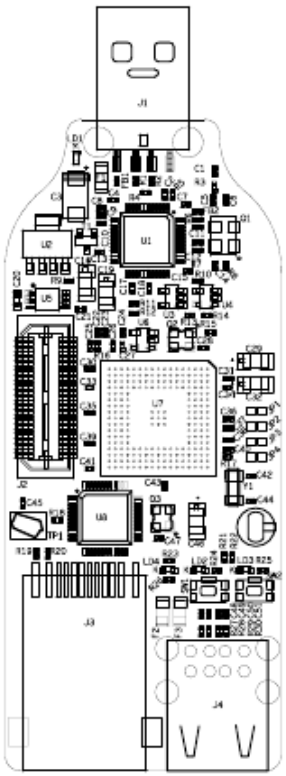
D'après Wikipedia :

> *Un système embarqué peut être défini comme un système électronique et informatique autonome, qui est dédié à une tâche bien précise. Ses ressources disponibles sont généralement limitées. Cette limitation est généralement d'ordre spatial (taille limitée) et énergétique (consommation restreinte).*
>
> *Les systèmes embarqués font très souvent appel à l'informatique, et notamment aux systèmes temps réel.*
>
> *Le terme de système embarqué désigne aussi bien le matériel que le logiciel utilisé.*

# Cible et Hôte

- Nous parlerons de système cible (target, la board) et de système hôte (host, votre machine)
- Le host va nous permettre de programmer, debugger, contrôler le système cible durant la période de développement
- Le système cible sera ensuite autonome.
- Nous utilisons un Linux sur les deux systèmes. Cela n'est pas une obligation (même, si cela facilite certains automatismes).

# 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
- 256Mo de flash
- 64Mo de RAM
- 64Ko d'EEPROM

Choisie car compacte, bien documentée, ouverte et très bien supportée par Linux

# Quelques notions de shell

## Bases

- Sous Linux, on peut faire beaucoup de choses avec la ligne de commande
- Très souvent, ce sera le seul langage de script disponible sur la cible
- Lancer une commande

```
$ ls
```

- Séparation des arguments par des espaces

```
$ mkdir dir1 dir2
```

- Souvent les arguments optionnels commence par '-' pour les options courtes et '--' pour les options longues (attention aux exceptions)

```
$ ls -l -a
$ ls -la
$ ls --sort=time
```

# Quelques notions de shell

## Redirections

On utilise beaucoup les redirections des entrées/sorties sous Linux (< | >) :

- Commande standard

```
$ echo foo
```

- Sortie standard vers un fichier

```
$ echo foo > file1
```

- Un fichier vers l'entrée standard

```
$ cat -n < file1
```

# Quelques notions de shell

## Redirections

- Sortie standard d'une commande vers l'entrée d'une autre

```
$ echo bar foo | wc
$ ls | wc -l
```

- Couplage des redirections

```
$ cat -n < file1 | wc > file3
```

- L'espace n'est pas obligatoire et les redirections ne sont pas forcement à la fin de la ligne

```
$ >file2 cat<file1 -n
```

# Quelques notions de shell

## Les chemins

Il est possible d'utiliser des chemins :

- absolus

```
$ mkdir /tmp/tete
```

- relatifs

```
$ mkdir ../../tmp/titi
```

`mkdir foo` signifie `mkdir ./foo`

# Quelques notions de shell

- Documentation : `man` (`man -l` pour une page “locale”). `</>` et `<?>` permettent de rechercher dans une page de man
- Globbing (à ne pas confondre avec les expressions régulières)

```
$ rm *.o
```

- Alias

```
$ alias ll="ls -l --color=auto"
$ alias cp="cp -i"
$ alias mv="mv -i"
$ alias rm="rm --one-file-system"
```

- Complétion (permet d’aller plus vite et se protèger des fautes de frappe)

```
$ cd /h<TAB>/j<TAB>/c<TAB>
```

# Quelques notions de shell

- Quelques séquences de touches :
  - `<Home>`/`<End>` : début/fin de ligne
  - `<Ctrl+Left>`/`<Ctrl+Right>` : se déplacer d'un mot à l'autre
  - `<Up>`/`<Down>` : se déplacer dans l'historique des commandes
  - `<Ctrl+R>` : rechercher dans l'historique. `<Ctrl+R>` de nouveau pour itérer
- Utilisez le copier-coller à la souris
  - selection : copie
  - click-milieu : colle
  - double-click : selectionne le mot
  - triple-click : selectionne la ligne

# Ecrire un script shell

Les fonctionnalités d'un script shell sont abolument identiques à celles de la ligne de commande.

- Ouvrir un nouveau fichier. On suffixe généralement les shell par +.sh+

```
$ vim script.sh
```

- Indiquer le shell utilisé sur la première ligne, préfixé de `#!`

```
#!/bin/sh
```

- La suite s'ecrit comme sur la ligne de commande

```
echo foo bar
```

- Il est nécéssaire de donner les droits en exécution au script

```
$ bash script.sh
$ chmod +x script.sh
$ ./script.sh
```

# Convention

Par convention, nous préfixons dans ces slides les commandes shell par :

- $ pour les commandes à exécuter par l'utilisateur normal
- % pour les commandes à exécuter par root
- > pour les commandes non-shell

Nous préfixons parfois les commandes shell par le système ou la commande doit être executée

# Se connecter par RS232

- RS232 est très utilisé. Vous trouverez peu de systèmes sans RS232
- D'un point de vue électrique : Gnd, Rx, Tx (+ RTS et CTS, mais inutile)
- Il est relativement simple de communiquer en RS232 avec une pin de sortie du processeur, mais de nos jours, on utilise des contrôleurs RS232 qui simplifient énormément le travail
- On pourra trouver des bus similaires et convertible en RS232 : RS422, RS485

# Se connecter par RS232

- Il faut un câble nul modem (= câble croisé)
- Il faut que le Linux sur la cible soit configuré pour utiliser le port RS232 comme console
  - Possible en ajoutant `console=ttyS0,115200n8` dans la ligne de commande du noyau (sur PC, il suffit de passer par *grub*)
  - Possible de le mettre dans la configuration du noyau
- Il faut sur le Linux un programme pour se logguer sur le port RS232 (comme sur PC) : `login` ou `getty`
- Il faut configurer le port RS232 : bitrate, protocole...

# Pourquoi n'y a-t-il pas de port série sur notre cible ?

- Vous savez qu'il existe des convertisseurs USB/RS232
- Sur notre cible, un convertisseur USB/RS232 intégré à la carte
- Le port USB que vous voyez n'est pas vu par le micro-contrôleur.
- Il est relié à une puce qui effectue la conversion USB/RS232 et relie la connexion série à la sortie console du micro-contrôleur.
- Cela permet :
  - d'économiser une alimentation (le PC alimente)
  - de gagner la place d'un port série et d'un port Jtag

# Communiqer par RS232

## Avec les outils de haut niveau

- Permettre d'accéder au port série sans être root :

```
host% adduser user dialout
```

- Il est nécessaire de se relogguer pour que les modifications sur les groupes soient prises en compte
- Les outils de haut niveau :
  - minicom : La référence
  - picocom : Fonctionne sans *ncurses* (mieux pour l'automatisation)
  - gtkterm : Fonctionne en graphique
  - putty : Fonctionne sous Linux et sous Windows
- Attention aux conflits avec certains services comme ModemManager

# Communiqer par RS232

## Minicom

```
host% apt-get install minicom
host$ minicom -D /dev/ttyUSB0
host$ minicom
```

La configuration se fait par <RET><Ctrl+A><O>

```
  Serial Device  : /dev/ttyUSB0
    ...
  Bps/Par/Bits   : 115200 8N1
Hardware Flow Control : No <- Important
Software Flow Control : No
```

# Communiquer par RS232

## Avec les outils de bas niveau

Outils de bas niveau : `stty`, `cat`, `echo`

- Permet de scripter (Tests automatiques, etc...)
- Dans le cas ou vous devriez coder votre propre client

Fonctionnement :

- Configuration :

```
host$ stty -F/dev/ttyUSB0 115200
```

- Lecture

```
host$ cat /dev/ttyUSB0 > file
```

- Ecriture

```
host$ echo root > /dev/ttyUSB0
host$ cat file > /dev/ttyUSB0
```

# Rebooter la clef

Pour le fun :

```
host$ lsusb -t
host$ echo 3-1 | sudo tee /sys/bus/usb/drivers/usb/
    unbind
host$ echo 3-1 | sudo tee /sys/bus/usb/drivers/usb/
    bind
host$ dmesg | tail
```

# Communiquer par réseau ethernet

Plus difficile. Il faut :

- Une interface réseau (plus complexe qu'un controleur RS232)
- Une pile IP (plus complexe qu'une communication RS232, donc ne permet de se connecter que plus tardivement dans le boot)
- Une configuration IP
- Un programme pour recevoir la demande de connexion

# Communiquer par réseau ethernet

Protocoles les plus utilisés :

- Telnet
  - `telnetd` et `telnet`

```
target% telnetd
host$ telnet -l root target
target%
```

  - Pas sécurisé, attention à votre mot de passe (`tshark -i lo -o out`, puis `chaosreader out`)
  - `<CTRL+]>` permet d'accéder à l'interface de commande
- Ssh
  - `sshd` et `ssh`

```
host$ ssh root@target
target%
```

  - Sécurisé
  - Plein de bonus de sécurisés
  - Il est possible de forcer la déconnexion avec `<RET><~><.>` et de suspendre une connexion avec `<RET><~><CTRL+Z>`

# Transférer des fichiers

Par réseau

- rcp/scp

```
host$ scp -r ~/dir root@target:dir-target
target$ scp -r user@host:dir-host .
```

- tftp. La syntaxe dépend de l'implémentation. udpsvd -vE 0.0.0.0 69 tftpd -c /files/to/serve

```
target$ tftp host -g -r file
target$ tftp host -p -l file
host$ tftp target -c get file
host$ tftp target -c put file
```

- wget

```
host$ wget http://host/file
host$ wget ftp://host/file
```

- Beaucoup d'autres méthodes plus ou moins standards

# Transférer des fichiers

Par liaison série

- Plus lent, moins malléable, mais peut vous sauver
- Protocoles `{X,Y,Z}MODEM` ou `Kermit`

```
host% apt-get install lrzsz
target$ sz file
target$ rz
<Ctrl-A><S>
```

# Utiliser des clef ssh

- Possibilité de créer des clefs pour `ssh`

```
host$ ssh-keygen -t dsa
```

- Toujours mettre un mot de passe sur votre clef
- Recopiez votre clef dans `~/.ssh/authorized_keys`

```
host$ ssh-copy-id root@target
```

# Utiliser des clefs ssh

- Utiliser `ssh-agent` (inutile de nos jours car déjà lancé avec la session)

```
host$ ssh-agent
host$ SSH_AUTH_SOCK=/tmp/agent.3391; export
    SSH_AUTH_SOCK;
host$ SSH_AGENT_PID=3392; export SSH_AGENT_PID;
host$ echo Agent pid 3392;
```

- Enregistrer votre passphrase auprès de l'agent

```
host$ ssh-add
```

- Forwarder votre agent

```
host$ ssh -A root@target
target%
```

# Notre premier programme

## hello.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "hello.h"

int main(int argc, char **argv) {
    static const char *str = "Hello World";
    long num = -1;
    char *end = NULL;

    if (argc > 1) {
        num = strtol(argv[1], &end, 10);
        if (!*argv[1] || *end || num < 0) {
            fprintf(stderr, "Invalid iteration number\
                n");
            num = -1;
```

# Notre premier programme

hello.c

```
        }
    }
    print(str, num);
    return 0;
}

void print(const char *str, long num) {
    int i;
    for (i = 0; i != num; i++) {
        printf("%s\n", str);
        sleep(1);
    }
}
```

# Notre premier programme

## hello.h

```
#ifndef HELLO_H
#define HELLO_H

void print(const char *str, long num);

#endif /* HELLO_H */
```

# Compilation

Installation du compilateur :

```
host% apt-get install gcc libc-dev
```

Compilation normale :

```
host$ mkdir build-x86
host$ gcc hello.c -o build-x86/hello
host$ build-x86/hello 1
Hello World
```

Remarque : On préfère effectuer des compilation *out-of-source* ou les objets se trouvent séparés des sources. Il est ainsi possible d'avoir un répertoire pour la production et un pour le debug ou bien un répertoire par cible.

# Compilation

Décompression de la toolchain :

```
host% cd /
host% tar xvzf .../arm-sysmic-linux-
    uclibcgnueabi_i386.tgz
```

- On installe très souvent les chaines de cross-compilation dans `/opt`.
- Beaucoup de toolchain ne sont pas *relocable*. C'est-à-dire qu'elle doivent être placée à l'endroit où elles ont été compilée (car il contiennent certains chemins en "dur").

# Compilation

## Compilation pour la cible :

```
host$ mkdir build-arm
host$ /opt/arm-sysmic-linux-uclibc/bin/arm-linux-
    gcc hello.c -o build-arm/hello
```

## ou bien

```
host$ PATH+=:/opt/arm-sysmic-linux-uclibc/bin
host$ arm-linux-gcc hello.c -o build-arm/hello
```

## Test :

```
target% ./hello 1
Hello World
```

# Identifier le résultat

Un bon moyen de reconnaître les binaires est d'utiliser la commande `file` :

```
host$ file */hello
arm/hello:    ELF 32-bit LSB executable, ARM,
    version 1 (SYSV), dynamically linked (uses
    shared libs), not stripped
arm-static/hello: ELF 32-bit LSB executable, ARM,
    version 1 (SYSV), statically linked, not
    stripped
x86/hello:    ELF 64-bit LSB executable, x86-64,
    version 1 (SYSV), dynamically linked (uses
    shared libs), for GNU/Linux 2.6.15, not
    stripped
x86-static/hello: ELF 64-bit LSB executable, x86
    -64, version 1 (GNU/Linux), statically linked,
    for GNU/Linux 2.6.15, not stripped
```

# Compiler et exécuter

- La force de Linux, c'est sa polyvalence
- Programmer pour un Linux embarqué n'est pas très différent que sur un Linux PC.
- Principales différences à garder à l'esprit :
  - Différence de vitesse de CPU et de quantité de mémoire
  - Différence de périphérique. Conséquences :
    - Drivers différents
    - Bibliothèques différentes (exemple : Qt dépend de l'accélération graphique)
  - Les différence d'architecture CPU peuvent empêcher l'utilisation de certains outils de debug (ex : Perf, Valgrind, SystemTap...)
- Compiler un programme pour une autre cible s'appelle cross-compiler

# Règle d'or

Jamais d'espaces dans les chemins de compilation

# Compiler avec un Makefile classique

- Pas très normalisé
- Utilisé pour les petits projets ou les projets non standard
- Ressemble souvent à :

```
host$ make CC=arm-linux-gcc LD=arm-linux-ld
```

- Il peut être nécessaire d'avoir la chaîne de cross-compilation dans son `PATH`.

```
host$ PATH+=:/opt/arm-sysmic-linux-
    uclibcgnueabi/bin
```

- Ne pas hésiter à lire le `Makefile`

# Compiler avec un Makefile classique

Exemple avec `memstat` :

- Récupération des sources

```
host$ wget http://ftp.de.debian.org/debian/pool
    /main/m/memstat/memstat_0.9.tar.gz
```

- Décompression des sources

```
host$ tar xvzf memstat_0.9.tar.gz
```

- Compilation

```
host$ cd memstat-0.9
host$ make CC=arm-linux-gcc LD=arm-linux-ld
    memstat
```

# Compiler avec Makefile classique

Il est aussi souvent possible de compiler ces programmes sans passer par Makefile ou de réécrire le Makefile pour une utilisation plus récurrente.

Exemple avec `memstat` :

```
host$ make clean
host$ arm-linux-gcc memstat.c -o memstat
```

# Compiler avec autotools

- C'est le cas le plus courant
- Pour une compilation classique :

```
host$ ./configure
host$ make
host% make install
```

- Compilation *out-of-source*. il est nécessaire d'appeller le `configure` à partir du répertoire de build.

```
host$ mkdir build
host$ cd build
host$ ../configure
host$ make
host% make install
```

- L'installation peut nécessiter les droits *root*
  - Utiliser `sudo -E` (attention à l'option `secure_path` dans `/etc/sudoers`)
  - Utiliser `fakeroot`

# Compiler avec autotools

- Compilation out-of-source :

```
host$ cd build
host$ ../configure --prefix=~/rootfs
host$ make
host$ make install
```

- Cross-compilation.

```
host$ PATH+=/opt/arm-sysmic-linux-uclibcgnueabi
    /usr/bin
host$ mkdir build
host$ ../configure --host=arm-linux --build=
    i386 --prefix=~/rootfs
host$ make
host$ make install
```

- Il est aussi possible (parfois préférable) d'utiliser `DESTDIR=` lors de `make install` au lieu de `--prefix=`

# Compiler avec autotools

Exemple avec `lzma` :

- Récupération et décompression des sources

```
host$ wget http://tukaani.org/lzma/lzma-4.32.7.
    tar.gz
host$ tar xvzf lzma-4.32.7.tar.gz
host$ cd lzma-4.32.7
```

- Configuration

```
host$ mkdir build && cd build
host$ PATH+=/opt/arm-sysmic-linux-uclibcgnueabi
    /usr/bin
host$ ../configure --host=arm-linux --build=
    i386 --prefix=~/rootfs
```

- Compilation

```
host$ make
```

- Installation

# Compiler avec autotools

Obtenir de l'aide :

```
host$ ./configure --help
```

Parmis les fichiers générés :

- `config.log` contient la sortie des opérations effectuées lors de l'appel de `./configure`. En particulier, il contient la ligne de commande utilisée. Il est ainsi possible de facilement dupliquer la configuration.

  ```
  host$ head config.log
  ```

- `config.status` permet de regénérer les Makefile. `config.status` est automatiquement appellé si un Makefile.am est modifié.

# Compiler un programme tiers

## Kconfig

- Système de compilation du noyau
- Très bien adapté à la cross-compilation
- Adapté aux environnements embarqués
- Adapté aux environnements avec beaucoup de configuration
- Pas un système de compilation réel. Composé de :
  - Kconfig, Système de gestion de configuration
  - Kmake, règles Makefile bien étudiées. Chaque projet les adapte à ces besoins
- Application de la règle : "Pas générique mais simple à hacker"
- Dépend principalement de `gmake`
- Exemple avec busybox :

```
host$ wget http://busybox.net/downloads/busybox
    -1.19.4.tar.bz2
host$ tar xvjf busybox-1.19.4.tar.bz2
host$ cd busybox-1.19.4
host$ make help
```

# Compiler un programme tiers

## Kconfig

- Pour configurer les options :
  - En ncurses (2 versions)

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

  - En Qt4

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

- Permettent d'effectuer des recherches
- Ne pas oublier d'installer les headers des bibliothèques

# Compiler un programme tiers

## Kconfig

- Pour cross-compiler

```
host$ make
```

- Pour compiler *out-of-source*

```
host$ mkdir build
host$ make O=build
```

- Mode verbose : `V=1`
- Forcer la toolchain : `CROSS_COMPILE=arm-linux-`
- Indispensable pour le noyau : `ARCH=arm`

# Compiler un programme tiers

## Kconfig

Test avec busybox :

- Récupération d'une configuration par défaut

```
host$ make CROSS_COMPILE=arm-linux- O=build
    defconfig
```

- Personnalisation de la configuration

```
host% apt-get install libncurses5-dev
host$ make CROSS_COMPILE=arm-linux- O=build
    menuconfig
```

- Compilation

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

- Installation

```
host$ make CROSS_COMPILE=arm-linux-
    CONFIG_PREFIX=~/rootfs install
```

# Compiler un programme tiers

## Placement des bibliothèques

- Pour que les autres programmes puissent en profiter le plus simple est d'installer les bibliothèques dans `TOOLCHAIN/TRIPLET/sysroot`. Elle seront ainsi trouvées automatiquement par le compilateur.
- Il est aussi possible de modifier les variables CFLAGS et LDFLAGS (plus complexe)
- Nous verrons que les bibliothèques doivent aussi se trouver sur la cible.

# Compiler un programme tiers

## Et si ca ne marche pas ?

- Cas où des programmes doivent être compilés puis exécutés sur la cible lors de la compilation
- Classiquement, des cas de bootstraping
- Cas notable de Python [1]
- Deux solutions :
  - Résoudre le problème et envoyer un patch
  - Compiler sur la target.
    - Vous avez alors besoin d'un gcc natif sur la target
    - Un émulateur type qemu peut vous aider

1. compile avec des patchs, mais pas évident

# Historique des Autotools

1. Makefile
2. Makefile + hacks pour effectuer de la configuration
3. Makefile.in + configure
4. Makefile.in + configure.ac
5. Makefile.am + configure.ac

# Créer un projet avec autotools

Fonctionnement des autotools :

- Préparation

```
% apt-get install automake autoconf
```

- Déclaration de notre programme et de nos sources pour `automake`

```
$ vim Makefile.am
```

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c hello.h
```

# Créer un projet avec autotools

- Création d'un template pour `autoconf` contenant les macros utiles pour notre projet

```
$ autoscan
$ mv configure.scan configure.ac
$ rm autoscan.log
$ vim configure.ac
```

- Personnalisation du résultat

```
...
AC_INIT([hello], [1.0], [bug@sysmic.org])
AM_INIT_AUTOMAKE([foreign])
...
```

- Génération du `configure` et des `Makefile.in`. C'est cette version qui devrait être livée aux packageurs.

```
$ autoreconf -iv
```

# Créer un projet avec autotools

- Compilation

```
$ ./configure --help
$ mkdir build
$ cd build
$ ../configure --host=arm-linux --build=i386 --
    prefix=~/rootfs
$ make
$ make install
```

# Créer un projet avec autotools

La cible `distcheck` :

1. Recopie les fichiers référencés dans Autotools
2. Retire les droits en écriture sur les sources
3. Lance une compilation *out-of-source*
4. Installe le projet
5. Lance la suite de test
6. Lance un distclean
7. Vérifie que tous les fichiers créés sont effectivement supprimés
8. Crée une *tarball* correctement nommée contenant les sources

# Créer un projet avec autotools

Si `automake` est appelé avec `-gnits`, `distcheck` effectue des vérifications supplémentaires sur la documentation, etc...

La fonctionnalité `distcheck` est le point fort souvent énoncé des autotools.

```
$ make distcheck
$ tar tvzf hello-1.0.tar.gz
```

# Créer un projet sans autotools

Utiliser les règles implicites facilite votre travail

```
hello: hello.o
```

Testons :

```
host$ make CC=arm-linux-gcc CFLAGS=-Wall
```

# Créer un projet sans autotools

`VPATH` vous permet de gérer la compilation *out-of-source*.
Remarquez que, pour que `VPATH` fonctionne correctement, il faut avoir correctement utilisé le quoting pour les directives d'inclusion (< pour les entêtes systèmes et " pour les entêtes du projet).

Testons :

```
host$ cd build
host$ make -f ../Makefile VPATH=.. CC=arm-linux-gcc
```

# Créer un projet sans autotools

`gcc` peut générer les dépendances de vos fichiers. On génère ainsi des morceaux de Makefile que l'on inclue. Il ne faut pas oublier d'ajouter la dépendance entre `hello.d` et les dépendances de `hello.c`

```
%.o: %.c
	$(COMPILE.c) -MMD -o $@ $<

-include hello.d
hello: hello.o
```

# Créer un projet sans autotools

Les Makefile permettent d'utiliser des fonctions de substitutions qui peuvent nous aider à rendre notre système plus générique.

```
%.o: %.c
	$(COMPILE.c) -MMD -o $@ $<

HELLO_SRC = hello.c
-include $(HELLO_SRC:%.c=%.d)
hello: $(HELLO_SRC:%.c=%.o)
```

# Créer un projet sans autotools

Nous pouvons ajouter des alias pour nous aider dans les commandes complexes

```
clean:
	rm -f hello $(HELLO_SRC:%.c=%.o) $(HELLO_SRC
	    :%.c=%.d)

.PHONY: debug-x86 debug-arm clean

debug-arm/Makefile:
	mkdir -p debug-arm
	echo 'all %:' >> debug-arm/Makefile
	echo '	make -f ../Makefile VPATH=.. CFLAGS=-g
	    CC=arm-linux-gcc $$@' >> debug-arm/
	    Makefile

debug-arm: debug-arm/Makefile
	make -C debug-arm
```

# Créer un projet sans autotools

En poussant ce fonctionnement un peu plus loin, on obtient le Kmake. Un système génère un fichier contenant toutes les options. Kmake, utilise des variables pour rendre la compilation conditionnelle :

```
HELLO_$(CONFIG1) += hello.o
```

Néanmoins, le résultat n'est pas portable entre les différentes implémentations de `make`.

# Bibliothèques courantes

Il existe des milliers de bibliothèques et d'API disponibles pour Linux. Parmi elles :

- Les appels systèmes. Inclus avec le noyau Linux. Documentés par les pages de man. Liste sur *syscalls(2)*
- L'interface Posix. Fournis par la libc. Documentés par les pages de man des sections 3 ou 3posix ou sur http://www.unix.org/version3/
- Les bibliothèques très utilisées : libapr, glibc, ømq, ... La documentation se trouve avec les source ou sur leur site web.
- Qt. Bibliothèque très complète pouvant être utilisé pour des usage très divers : embarqué, multimédia, etc... En C++. Documentation sur http://qt-project.org/doc/qt-5.0
- Regarder les packets inclus dans Buildroot ou sur votre distribution permet d'avoir une idée des bibliothèques les plus courantes