Raspberry Pi, créer son propre système embarqué sous Linux

Dr. Eng. Chiheb Ameur ABID

/in/chiheb-ameur-abid
chiheb.abid@gmail.com

Janvier 2026

Plan

1. Programmation concurrentielle
2. Scripts Shell
3. Ré-compilation du noyau
4. Dèveloppement des modules noyau
5. Configuration "from scratch"

Programmation Parallèle

Les threads

- Programmation concurrente
  - Travailler sur plusieurs tâches à la fois
- Un thread (tâche) représente (généralement) une fonction (une méthode) en cours d'exécution
- Un processus peut contenir un ou plusieurs threads. Ces threads partagent alors la mémoire ainsi que diverses autres ressources*
  - Deux threads qui veulent collaborer peuvent le faire par l'intermédiaire de variables partagées.
  - Le contexte d'un thread est léger par rapport à un processus

Programmation concurrentielle

Création d'un thread

```
1  #include <iostream>
2  #include <thread>
3  void hello()
4  {
5   std::cout<<"Hello Concurrent World\n";
6  }
7  int main()
8  {
9   std::thread t(hello);
10  t.join();
11 }
```

Programmation concurrentielle

Mutex : protection des variables partagées

```
#include <list>
#include <mutex>
#include <algorithm>
std::list<int> some_list;        ← 1
std::mutex some_mutex;           ← 2
void add_to_list(int new_value)
{
    std::lock_guard<std::mutex> guard(some_mutex);   ← 3
    some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{
    std::lock_guard<std::mutex> guard(some_mutex);   ← 4
    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

1. Variable globale, donc elle est une variable partagée
2. Instance de mutex
3. L'accès est protégé par un mutex
4. L'accès est protégé par un mutex

- Création d'un mutex en `C++17`

```
std::lock_guard guard(some_mutex);
```


Programmation concurrentielle

Mise en application

Créer deux threads :

1. Un premier thread flashe une diode LED à chaque seconde
2. Un deuxième thread active/désactive le flashage de la diode LED à chaque appuie d'un bouton poussoir

Plan

1 Programmation concurrentielle

2 Scripts Shell

3 Ré-compilation du noyau

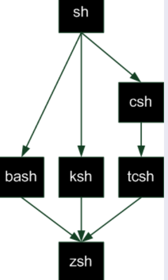
4 Dèveloppement des modules noyau

5 Configuration "from scratch"

Introduction aux scripts shell

Qu'est-ce qu'un script shell ?

- Un script shell est un fichier texte exécutable contenant des commandes shell. Ces commandes vont être interprétées par le shell les unes après les autres du haut vers le bas.
- Il existe plusieurs dialectes
  - Bourne Shell (`/bin/sh`),
  - le Korn Shell (`/bin/ksh`) pour lequel deux versions majeurs sont aujourd'hui couramment utilisées (ksh 88 et ksh 93)
  - le cShell (`/bin/csh`) pour les utilisateurs préférant un langage apparenté au C



Introduction aux scripts shell

1er script

```
pi@raspberrypi:~ $ cat script1.sh
#!/bin/bash
if test 5 = 5
then
echo "Condition vérifiée"
fi
pi@raspberrypi:~ $ chmod +x script1.sh
pi@raspberrypi:~ $ ./script1.sh
Condition vérifiée
```

- Il est possible de ne pas spécifier que le fichier est exécutable, dans ce cas le lancement d'un script s'effectue comme suit :
  `$ bash script1.sh`
- Exécution avec débogage

```
pi@raspberrypi:~ $ cat script2.sh
#!/bin/sh
set -x # Activation du débogage à partir de maintenant
echo "Bonjour!"
date # Cette ligne est la ligne qui affiche la date
set +x #Désactivation du débogage
pi@raspberrypi:~ $ ./script2.sh
+ echo Bonjour!
Bonjour!
+ date
mercredi 8 janvier 2020, 06:58:43 (UTC+0000)
+ set +x
```

Les scripts shell

Les variables

- Déclarer une variable
  `variable=chaine`
- L'accès en lecture à une variable s'effectue en précédant le nom de la variable du symbole $

```
1 nom="Pierre" # Définir la valeur de la variable
2 echo $nom
```

  - `${var}` : renvoie le contenu de `$var`. Il sert à isoler le nom de la variable par rapport au contexte de son utilisation. Ceci évite les confusions entre ce que l'utilisateur désire `${prix}F` (variable « prix » suivie du caractère « F ») et ce que le Shell comprendrait si on écrivait simplement `$prixF` (variable `prixF`).
  - `${var-texte}` : renvoie le contenu de la variable `var` si celle-ci est définie (existe en mémoire) ; sinon renvoie le texte « texte ».
- Exemples de variables prédéfinies
  `$HOME` : Répertoire personnel de l'utilisateur
  `$PWD` : Répertoire courant
  `$LOGNAME` : Nom de login
  `$PATH` : Chemins de recherche des commandes
  `$?` : Statut de la dernière commande lancée

Les scripts shell

Les arguments

- Dans un script, les paramètres ou arguments, positionnés par l'utilisateur exécutant le script, sont automatiquement et toujours stockés dans des « variables automatiques » (remplies automatiquement par le Shell).
  - `$0` : nom du script. Le contenu de cette variable est invariable. Il peut être considéré comme étant « à part » du reste des arguments
  - $1, $2, $3, ..., $9 : argument placé en première, seconde, troisième... neuvième position derrière le nom du script
  - `$#` : nombre d'arguments passés au script
  - `$*` : liste de tous les arguments (sauf `$0`) concaténés en une chaîne unique
  - `$@` : liste de tous les arguments (sauf `$0`) transformés individuellement en chaîne. Visuellement, il n'y pas de différence entre « `$*` » et « `$@` »

Exemple

```
pi@raspberrypi:~ $ cat script3.sh
#!/bin/sh
echo $0 # Affichage nom du script
echo $1 # Affichage argument n° 1
echo $2 # Affichage argument n° 2
echo $5 # Affichage argument n° 5
echo $# # Affichage du nombre d'arguments
echo $* # Affichage de tous les arguments
pi@raspberrypi:~ $ ./script3.sh a1 a2 a3 a4 a5
./script3.sh
a1
a2
a5
5
a1 a2 a3 a4 a5
```

Les scripts shell

Les tests

- La commande `test` a pour but de vérifier (tester) la validité de l'expression demandée, en fonction des options choisies. Elle permet ainsi de vérifier l'état des fichiers, comparer des variables, etc.
- `test` renvoie un statut vrai ou faux. Mais cette commande n'affiche rien à l'écran. Il faut donc, pour connaître le résultat d'un test, vérifier le contenu de la variable `$?`.
- Test simple sur les fichiers
  Syntaxe : `test option "fichier"`
  `-s` : fichier non vide, `-f` : fichier ordinaire, `-d` : un répertoire, etc.
- Tests sur les valeurs numériques
  Syntaxe : `test nb1 option nb2`
  `-eq` nb1 égal à nb2 (equal), `-ne` nb1 différent de nb2 (non equal), `-lt` nb1 inférieur à nb2 (less than), `-le` nb1 inférieur ou égal à nb2 (less or equal), `-gt` nb1 supérieur à nb2 (greater than), `-ge` nb1 supérieur ou égal à nb2 (greater or equal).

Les scripts shell

Les structures conditionnelles

- Les commandes `if - elif - else` sont utilisées pour exécuter des commandes selon la validité d'une ou de plusieurs conditions

```
if [ test ]
then
        echo "Le premier test a été vérifié"
elif [ autre_test ]
then
        echo "Le second test a été vérifié"
elif [ encore_autre_test ]
then
        echo "Le troisième test a été vérifié"
else
        echo "Aucun des tests précédents n'a été vérifié"
fi
```

Les scripts shell

Les structures conditionnelles

- Exemple d'utilisation de `if - elif - else`

```
#!/bin/bash

if [ $1 = "Bruno" ]
then
        echo "Salut Bruno !"
elif [ $1 = "Michel" ]
then
        echo "Bien le bonjour Michel"
elif [ $1 = "Jean" ]
then
        echo "Hé Jean, ça va ?"
else
        echo "J'te connais pas, ouste !"
fi
```

Les scripts shell

Les boucles

- La boucle `while` permet de répéter l'exécution d'une séquence de commandes tant que la condition en entrée est vérifiée

```
#!/bin/bash

while [ -z $reponse ] || [ $reponse != 'oui' ]
do
        read -p 'Dites oui : ' reponse
done
```

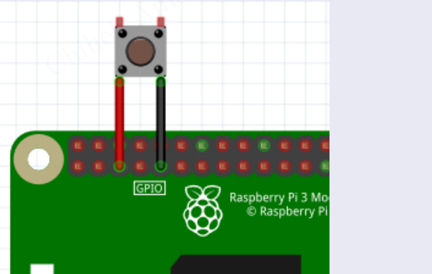
- La boucle `for` permet de parcourir une liste de valeurs et de boucler autant de fois qu'il y a de valeurs

```
#!/bin/bash

liste_fichiers=`ls`

for fichier in $liste_fichiers
do
        echo "Fichier trouvé : $fichier"
done
```

Les scripts shell

Mise en application

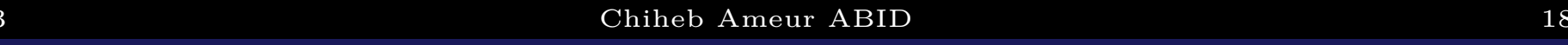
- On se propose d'écrire un script shell permettant d'arrêter la carte en appuyant sur un bouton connectée à travers les broches physiques 5 et 9
- On configure le script pour qu'il se lance automatiquement au démarrage du système afin de bénéficier de son service



Les scripts shell

Le script shell

```
#!/bin/bash
if [ ${UID} -ne 0 ]; then
   echo "Error: This script must be run as root" >&2
   exit 1
fi
while ( true )
do
    # check if the pin 5 is connected to GND
    if [ $(gpio -1 read 5) == 0 ]
    then
        shutdown -h now "System halted by a GPIO action"
    fi
    sleep 0.1 #0.1sec
done
```



Les scripts shell

Lancement du script shell au démarrage

- Pour lancer le script au démarrage, il faut éditer le fichier `rc.local` avec `sudo nano /etc/rc.local`

Amélioration du script

- Modifier le script pour prendre en considération l'utilisation de la broche 5 par le bus I2C. Dans ce cas, le script est désactivé au démarrage.
- Indications : utiliser les commandes `grep`, `cut`, `tail` et `i2cdetect`

Les scripts shell

Le script shell vérifiant si le bus I2C est utilisé

```
#!/bin/bash
if [ ${UID} -ne 0 ]; then
   echo "Error: This script must be run as root" >&2
   exit 1
fi
if [ $(i2cdetect -y 1 | tail -n 8 | cut -d':' -f2-
                | egrep "[0-9]|[a-f]" | wc -l) -ne 0 ]; then
   echo "Pin 5 is used by i2c"
   exit 1
fi
while ( true )
do
    # check if the pin is connected to GND
    if [ $(gpio -1 read 5) == 0 ]
    then
        shutdown -h now "System halted by a GPIO action"
    fi
    sleep 0.1 #0.1sec
done
```

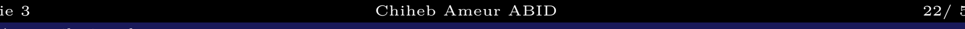
Plan

1. Programmation concurrentielle
2. Scripts Shell
3. Ré-compilation du noyau
4. Développement des modules noyau
5. Configuration "from scratch"

Ré-compilation du noyau

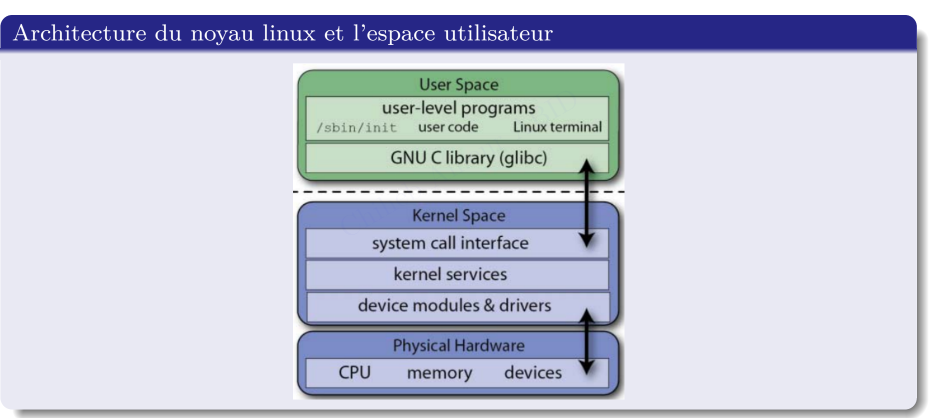
Intérêts de la ré-compilation, configuration ajustée

- Avoir une distribution sur-mesure
  - Une distribution qui répond à un besoin spécifique (serveur de partage, serveur multimédia, etc.)
  - Meilleure performance
  - Taille de distribution optimale
- Bénéficier de nouvelles fonctionnalités d'un noyau plus récent
- Développer des pilotes facilitant en garantissant un accès direct aux périphériques
  - Introduire l'aspect temps réel



Ré-compilation du noyau

Architecture du noyau linux et l'espace utilisateur



Plan

1 Programmation concurrentielle
2 Scripts Shell
3 Ré-compilation du noyau
4 Dèveloppement des modules noyau
5 Configuration "from scratch"

Dèveloppement des modules noyau

Caractéristiques d'un LKM

- Le noyau linux admet une structure modulaire : core kernel + LKMs
- LKM (Loadable Kernel Module) : un module chargé dynamiquement suite à la demande d'une fonction

Caractéristiques d'un LKM

- Exécution non séquentielle
  - Réaction à la suite de réception d'un évènement
- Absence du nettoyage automatique
- Pas d'accès aux bibliothèques dédiées à l'espace utilisateur (glibc)
  Pas de `printf`, mais on utilise `printk`
- Peut être interrompu
- Admet une priorité d'exécution plus élevée que les applications utilisateurs
- Les calculs en virgule flottante ne sont pas supportés

Développement des modules noyau

Structure d'un LKM (1/2)

```c
// The license type (affects behavior)
MODULE_LICENSE("GPL");
MODULE_AUTHOR("CA ABID"); // Author of the module
MODULE_DESCRIPTION("Exemple de module LKM sur RPi3."); // Descr.
MODULE_VERSION("0.1"); // Version of the module

// an LKM argumant example
static char *name = "monde";
// Argumant description charp = char pointer, defaults to "world"
module_param(name, charp, S_IRUGO); // S_IRUGO can be read
MODULE_PARM_DESC(name, "The name to display in /var/log/kern.log");
```

Développement des modules noyau

Structure d'un LKM (2/2)

```c
// Utilisée à l'initialisation, cette fonction
// écrit un message dans le fichier log du noyau
static int __init helloERPi_init(void) {
  printk(KERN_INFO "ERPi: Bonjour %s de RPi LKM!\n", name);
  return 0;
}
// Fonction de sortie ou de nettoyage
static void __exit helloERPi_exit(void) {
  printk(KERN_INFO "ERPi: Bye %s de RPi LKM!\n", name);
}
// Les macros spécifient quelles fonctions à invoquer
module_init(helloERPi_init);
module_exit(helloERPi_exit);
```

Développement des modules noyau

Compilation native d'un LKM

- La compilation s'effectue sur la RPi3
- Installer et préparer les fichiers headers du noyau avec la même version du noyau de la cible
  - ☞ La compilation d'un LKM nécessite de retrouver des informations à partir des fichiers headers du noyau
- Installation et préparation des fichiers headers du noyau sur la RPi3

1. Se connecter en tant qu'utilisateur `root` : plusieurs opérations nécessitent les privilèges du super-utilisateur

```
sudo -i
```

2. Récupérer la version du noyau

```
uname -r
```

3. Se placer dans le répertoire `/usr/src` et télécharger le code source du noyau

```
wget https://github.com/raspberrypi/linux/tarball/rpi-4.19.y
```

Installation et préparation des fichiers headers sur la RPi3

4. Extraire les fichier de l'archive téléchargé

```
tar zxf rpi-4.19.y
```

```
pi@raspberrypi:/usr/src $ ls
linux  linux-4.19.75-v7+  raspberrypi-linux-988cc7b  rpi-4.19.y  sense-hat
```

5. Écraser le fichier de configuration

```
modprobe configs
cd raspberrypi-linux-988cc7b
zcat /proc/config.gz > .config
```

6. Vérifier que toutes les bibliothèques nécessaires pour compiler un LKM sont présents. Il est peut être nécessaire d'installer de nouveaux paquets (bison, flex, libssl-dev)

```
make oldconfig
make modules_prepare
```

7. Télécharger le fichier `Module.symvers` contenant les symboles non définis dans le noyau

```
wget https://github.com/raspberrypi/firmware/raw/master /extra/Module7.symvers
cp Module7.symvers Module.symvers
```

Installation et préparation des fichiers headers sur la RPi3

8. On crée des liens symboliques permettant à `Makefile` de retrouver les fichiers headers du noyau
   On garde le chemin du répertoire contenant les fichiers headers dans la variable `KHEADER`

```
KHEADER=`pwd`
```

Les modules compilés des noyaux sont localisés dans `/lib/modules`

```
pi@raspberrypi:/usr/src/raspberrypi-linux-988cc7b $ ls /lib/modules
4.19.75+  4.19.75-v7+  4.19.75-v7l+  4.19.75-v8+  4.19.86+  4.19.86-v7+  4.19.86-v7l+  4.19.86-v8+
```

Les modules du noyau compilés destinés à RPi3 sont localisés dans le dossier `4.x.x.v7+`

```
cd /lib/modules/4.19.86-v7+
```

On crée les liens suivants :

```
1 ln -s $KHEADER build
2 ln -s $KHEADER source
```

Dans le répertoire `/usr/src`, on crée les liens suivants :

```
1 cd /usr/src/
2 ln -s \$KHEADER linux-`uname -r`
3 ln -s \$KHEADER linux
```

Développement des modules noyau

Compilation d'un LKM

- La compilation s'effectue par un fichier `kbuild Makefile`

```
obj-m+=hello.o
all:
  make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
  make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

  - Chaque appel de `make` doit être précédé d'une tabulation
  - `obj-m` indique à `make` que le binaire à générer est un LKM
  - L'option `-C` indique au `make` d'accèder au répertoire du noyau avant d'effectuer n'importe quelle tâche
  - L'affectation `M=$(PWD)` spécifie à la commande `make` l'emplacement du projet actuel

Développement des modules noyau

Compilation d'un LKM

- Pour procéder à la compilation, mettre les fichiers `Makefile` et le fichier source du LKM (dans notre cas `hello.c`) dans le même répertoire, puis taper :
  `$ make`
  - Lancer la compilation en tant qu'utilisateur `root` entraînera la ré-compilation de tout le noyau
  - Le module compilé portera l'extension `.ko`

Développement des modules noyau

Test du module LKM

- Charger le module

```
1 insmod hello.ko
2 insmod hello.ko name=Alain # Modifier la valeur du paramètre par défaut
```

  - Observer les messages du noyau à l'aide de la commande `dmesg`
- Afficher les informations relatives au LKM chargé

```
modinfo hello.ko
```

- Afficher la liste des modules chargés

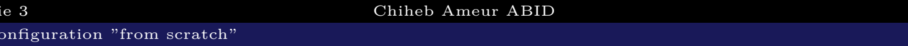
```
lsmod
```

- Décharger le module

```
rmmod hello.ko
```

Plan

1. Programmation concurrentielle
2. Scripts Shell
3. Ré-compilation du noyau
4. Dèveloppement des modules noyau
5. Configuration "from scratch"
   - Préparer le toolchain
   - Compilation from scratch



Plan

1. Programmation concurrentielle
2. Scripts Shell
3. Ré-compilation du noyau
4. Dèveloppement des modules noyau
5. Configuration "from scratch"
   - Préparer le toolchain
   - Compilation from scratch

Mise en oeuvre d'un toolchain avec Buildroot

Préparation

- Créer l'arborescence suivante
- Accéder au répertoire `br-tree`

Téléchargement de buildroot

- Télécharger la dernière version de Buildroot

```
wget http://www.buildroot.org/downloads/buildroot-2019.02.7.tar.bz2
```

- Décompresser le fichier téléchargé

```
tar xf buildroot-2019.02.7.tar.bz2
```

- Accéder au répertoire créé

```
cd buildroot-2019.02.7
```

Mise en oeuvre d'un toolchain avec BUILDROOT

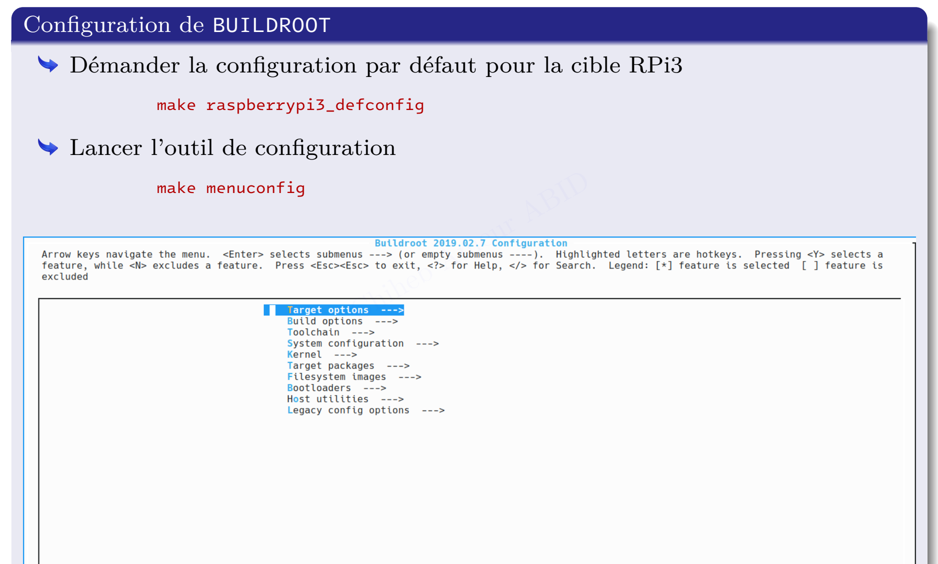
Configuration de BUILDROOT

- Démander la configuration par défaut pour la cible RPi3

```
make raspberrypi3_defconfig
```

- Lancer l'outil de configuration

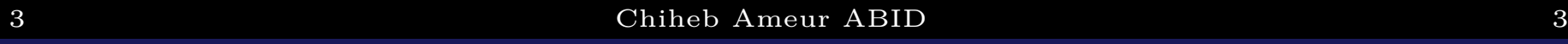
```
make menuconfig
```



Mise en oeuvre d'un toolchain avec BUILDROOT

Configuration de BUILDROOT

1. Menu `Target options` (Rien à changer) : options relatives à la cible
2. Menu `Build options`
   - `Download dir` : Pour éviter le re-téléchargement des mêmes fichiers pour des re-compilations successives
     Nouvelle valeur : `$(TOPDIR)/../dl`
   - `Host dir` : Spécifier l'emplacement où se trouvera la toolchain compilée
     Nouvelle valeur : `$(TOPDIR)/../board/rpi-3/cross`



Mise en oeuvre d'un toolchain avec BUILDROOT

Configuration de BUILDROOT

3. Menu `Toolchain`
   - `C library` : Spécifier quelle bibliothèque à utiliser pour bénéficier des services du système (appels système). La bibliothèque la plus complète est la Gnu C library
     Nouvelle valeur : `glibc`
   - `Kernel Headers` : Choisir les fichiers headers du noyau à compiler
     Nouvelle valeur : `Linux 4.19.x kernel headers`.
   - `GCC compiler Version` : Choisir `gcc 8.x` pour bénéficier des fonctionnalités d'une version plus récente du compilateur
     Nouvelle valeur : `gcc 8.x`
   - `Enable C++ support` : activer le compilateur C++
     Nouvelle valeur : `[*]`
   - `Build cross gdb for the host` : Avoir un débogueur sur le PC pouvant interprété le code ARM
     Nouvelle valeur : `[*]`

Mise en oeuvre d'un toolchain avec BUILDROOT

Configuration de BUILDROOT

4. Menu `System configuration` : options relatives à la génération du système
   - `Init system` : Nous désactivons BusyBox
     Nouvelle valeur : `None`
   - `Custom scripts to run before creating filesystem images` : Pas de script à exécuter,
     Nouvelle valeur : ()
   - `Custom scripts to run after creating filesystem images` : Pas de script à exécuter,
     Nouvelle valeur : ()
5. Menu `Kernel`
   - `Download dir` : L'objectif est de générer la toolchain, donc à désactiver
     Nouvelle valeur : ()
6. Menu `Target packages`
   - `BusyBox` : C'est le seul package initialement présent. Nous le désactivons.
     Nouvelle valeur : ()

Mise en oeuvre d'un toolchain avec BUILDROOT

Configuration de BUILDROOT

7. Menu `Filesystem images`
   - `ext2/3/4 root filesystem` : Inutile, nous ne voulons pas un système de fichiers
     Nouvelle valeur : ()
8. Menu `Legacy config options` : ce menu n'est utile que lorsque nous utilisons une ancienne configuration avec une nouvelle version de `Builtroot` pour déterminer les fonctions incompatibles.

Générer la toolchain avec BUILDROOT

- Sauvegarder la configuration et quitter `menuconfig`
  La configuration est sauvegardée dans un fichier nommé `.config`
- Il est utile pour des utilisations ultérieures de sauvegarder le fichier de configuration
  `$ cp .config ../board/rpi-3/buildroot-2019.02.7-toolchain.config`
- Générer la toolchain
  `$ make toolchain`

Plan

1. Programmation concurrentielle
2. Scripts Shell
3. Ré-compilation du noyau
4. Dèveloppement des modules noyau
5. Configuration "from scratch"
   - Préparer le toolchain
   - Compilation from scratch

Préparation de BUILDROOT

- Accèder au répertoire `br-tree` créé pour la compilation de la toolchain

```
cd $HOME/br-tree
```

- Supprimer le répertoire de la compilation `/buildroot-2019.02.7`

```
rm -rf buildroot-2019.02.7
```

- Décompresser de nouveau l'archive de `Buildroot`

```
tar xf buildroot-2019.02.7.tar.bz2
```

- Accéder au répertoire créé

```
cd buildroot-2019.02.7
```

- Demander la configuration par défaut pour la cible RPi3

```
make raspberrypi3_defconfig
```

- Lancer l'outil de configuration

```
make menuconfig
```


Compilation from scratch

Configuration de BUILDROOT

1. Menu `Target options` : Rien à changer
2. Menu `Build options`
   - `Download dir` : Les fichiers téléchargés seront sauvegardés dans le même répertoire `br-tree/dl`
     Nouvelle valeur : `$(TOPDIR)/../dl`

Compilation from scratch

Configuration de BUILDROOT

3. Menu `Toolchain` : La configuration permettra de retrouver la toolchain créée
   - `Toolchain type` : Indiquer à `Buildroot` que la toolchain est préexistante
     Nouvelle valeur : `External toolchain`
   - `Toolchain` : elle a été compilée spécifiquement
     Nouvelle valeur : `Custom toolchain`
   - `Toolchain path` : le répertoire dans lequel se trouve le sous-répertoire `bin` de la toolchain
     Nouvelle valeur : `$(TOPDIR)/../board/rpi-3/cross/usr`
   - `External toolchain gcc version` : Nous spécifions la version du compilateur de notre toolchain créée
     Nouvelle valeur : `8.x`
   - `External toolchain kernel headers serie` : Spécifier la version du noyau identique à celle utilisée avec la toolchain
     Nouvelle valeur : `Linux 4.19.x`
   - `External toolchain C library` : Le même choix que la toolchain, la bibliothèque `GlibC`
     Nouvelle valeur : `glibc/eglibc`
   - `Toolchain has C++ support` : Oui, notre toolchain supporte C++
     Nouvelle valeur : `[*]`

Compilation from scratch

Configuration de BUILDROOT

4. Menu `System configuration` : Activer la gestion des périphériques à travers `/dev`
   - Dans `/dev management`, sélectionner `Dynamic using devtmps+mdev`
   - Activer `Enable root login with password`
   - Sélectionner `Root password` et taper un mot de passe pour l'utilisateur `root`
5. Menu `Kernel` : Rien à changer
6. Menu `Target packages`
   - Dans `Hardware handling -> Firmware ----->`, sélectionner `rpi-wifi-firmware`
   - Dans `Networking applications`, sélectionner les packages suivants : `crda, ifupdown scripts, iw, openssh, wireless-regdb wpa_supplicant` avec `Enable EAP, Enable WPS, Install wpa_cli binary, Install wpa_client shared library, Install wpa_passphrase binary`
7. Menu `Filesystem images` : rien à changer.
8. Menu `Bootloaders` : rien à changer.
9. Menu `Host utilities` : rien à changer.
10. Menu `Legacy config options` : rien à changer.

Compilation from scratch

Configuration post-compilation

- Dans le répertoire `./board/raspberrypi3`, créer les fichiers de configuration réseau
  - Le fichier `interfaces`

```
auto lo
iface lo inet loopback
auto wlan0
iface wlan0 inet dhcp
pre-up wpa_supplicant -B -Dwext -iwlan0
-c/etc/wpa_supplicant.conf
post-down killall -q wpa_supplicant
wait-delay 15
iface default inet dhcp
```

  - Le fichier `wpa_supplicant.conf`

```
network={
ssid="SSID"
psk="PASSWORD"
}
```


Compilation from scratch

Configuration post-compilation

- Ajouter au fichier `post-build.sh` les lignes suivantes :

```
1 # Activer la gestion des périphériques à travers /dev
2 cp package/busybox/S10mdev ${TARGET_DIR}/etc/init.d/S10mdev
3 chmod 755 ${TARGET_DIR}/etc/init.d/S10mdev
4 cp package/busybox/mdev.conf ${TARGET_DIR}/etc/mdev.conf
5 # Mettre les fichiers de configuration réseau
6 cp board/raspberrypi3/interfaces ${TARGET_DIR}/etc/network/
7 cp board/raspberrypi3/wpa_supplicant.conf ${TARGET_DIR}/etc/
8 # Activer la connexion pour l'utilisateur root avec SSH
9 echo "PermitRootLogin yes" >> ${TARGET_DIR}/etc/ssh/sshd\_config
```

Compilation from scratch

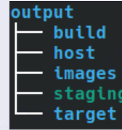
Générer la distribution

- Sauvegarder la configuration et quitter `menuconfig`
  La configuration est sauvegardée dans un fichier nommé `.config`
- Il est utile pour des utilisations ultérieures de sauvegarder le fichier de configuration
  `$ cp .config ../board/rpi-3/buildroot-2019.02.7-kernel.config`
- Lancer la compilation
  `$ make`

Compilation from scratch

Kernel compilé

- Tous les fichiers produits lors de la compilation se trouvent dans la sous arborescence `output`
  `$ tree output -L 1`



- `build` : tous les composants sont construits (ce qui inclut les outils nécessaires par Buildroot sur l'hôte et des paquets compilés pour la cible)
- `host` : contient l'installation des outils compilés pour l'hôte qui sont nécessaires pour la bonne exécution de `Buildroot`
- `images` : toutes les images (image du noyau, bootloader et système de fichiers racine) sont stockés. Ce sont les fichiers qui seront copiés sur le système cible
- `staging` : les en-têtes et les bibliothèques de la chaîne d'outils de compilation croisée et tous les paquets de l'espace utilisateur sélectionnés pour la cible
- `target` : contient presque le système de fichiers `rootfs` complet pour la cible : tout le nécessaire est présent, sauf les fichiers de périphérique dans `/dev`

Compilation from scratch

Tester la distribution

- Accéder au sous-répertoire `output/images`
- Le fichier `sdcard.img` est le fichier image à écrire sur la carte SD
- Écrire le fichier image sur la carte SD avec la commande `dd`
- Tester la distribution en se connectant via SSH, ou bien à travers le débogage UART

Merci pour votre attention

Questions ?