Chapter 1

Utilisation basique de Buildroot

Objectifs:

- Télécharger Buildroot
- Configurer un système minimal avec Buildroot pour la Raspberry Pi 3
- Lancer la compilation
- Flasher et tester le système généré

1.1 Setup

Comme spécifié dans la documentation Buildroot[1], vous avez besoin de certains paquets/dépendances :

```
sudo apt install sed make binutils gcc g++ bash patch \
  gzip bzip2 perl tar cpio python unzip rsync wget libncurses-dev
```

Lisez les spécificités de votre carte RPi3 sur différents sites notamment le site de la Raspberry Fondation

1.2 Télécharger Buildroot

Comme nous allons réaliser du développement Buildroot, on va télécharger les sources depuis le répertoire Git :

```
git clone git://git.busybox.net/buildroot
```

Aller dans le nouveau dossier `buildroot` créé.
Nous allons utiliser la branche *2019.08* de cette release.

```
git checkout 2019.08
```

[1] https://buildroot.org/downloads/manual/manual.html#requirement-mandatory

1.3 Configuration de Buildroot

Si vous regardez le dossier `configs/`, vous verrez qu'il y a un fichier `raspberrypi3_defconfig`, qui est une configuration pour construire un système pour une Raspberry Pi 3. Faites donc un:

```
make raspberrypi3_defconfig
```

Mais comme on souhaite apprendre Buildroot, on va regarder la configuration un peu plus en détails. Démarrez l'utilitaire de configuration Buildroot :

```
make menuconfig
```

Il est aussi possible d'utiliser les autres outils comme `nconfig`, `xconfig` ou `gconfig`.
Maintenant, regardons les points importants Buildroot:

- Menu `Target Options`
  - `Target Architecture` : `ARM (little endian)`
  - `Target Architecture Variant` : D'après le site web de la RPi3, ils utilisent un CPU Broadcom BCM2837 qui est basé sur ARM `Cortex-A53`
- Menu `Build options` : Regardez notamment le lieu où sauver la configuration Buildroot
- Menu `Toolchain`
  - Par défaut, Buildroot compile sa propre toolchain. Ca prend un peu de temps même si pour `ARMv8` il y a une toolchain pré-buildée fournie par ARM.
- Menu `Kernel`
  - Par défaut, le kernel le plus récent dans la release Buildroot est utilisé. Ici, il s'agit d'un kernel spécifique, sur github d'où le `Custom Tarballs`
  - Faites attention à la configuration kernel utilisé `Using an in-tree defconfig file` avec `bcm2709` comme configuration. Correspond au ficher `bcm2709_defconfig` dans le kernel: https://github.com/raspberrypi/linux/blob/rpi-4.19.y/arch/arm/configs/bcm2709_defconfig
  - Sur ARM, beaucoup de plateformes utilisent un *Device Tree* pour décrire le hardware. Vous pouvez regarder son DT : https://github.com/raspberrypi/linux/blob/rpi-4.19.y/arch/arm/boot/dts/bcm2710-rpi-3-b.dts,
- Menu `Target packages`. Surement le plus important car c'est le menu qui permet d'installer les paquets souhaités dans le rootfs final. On verra dans un autre lab un exemple.
- Menu `Filesystem images`. Pour savoir quel type d'image on veut générer.
- Menu `Bootloaders`
  - Le plus populaire pour ARM est *U-Boot*. Pour la RPi3 n'a pas besoin d'un bootloader pour executer et charger le kernel donc pas besoin!

1.4 Compilation

Simplement lancez `make` ou redirigez la sortie avec la commande `tee`:

```
make 2>&1 | tee build.log
```


1.5 Flasher la carte SD

Une fois la compilation terminée, on va pouvoir flasher notre système sur notre carte SD. Mais avant, encore faut-il l'identifier ! Regardez la sortie de `cat /proc/partitions` et chercher votre carte SD. En géneral, un lecteur de carte SD donnera `mmcblk0`, alors qu'un lecteur externe USB, ça sera du type `sdX` (i.e`sdb`, `sdc`, etc.). **Attention : `/dev/sda` est généralement le disque dur de votre laptop!**

Si votre carte SD est `/dev/mmcblk0`, les partitions seront `/dev/mmcblk0p1`, `/dev/mmcblk0p2`, etc. Si c'est `/dev/sdc`, les partitions seront `/dev/sdc1`, `/dev/sdc2`, etc.

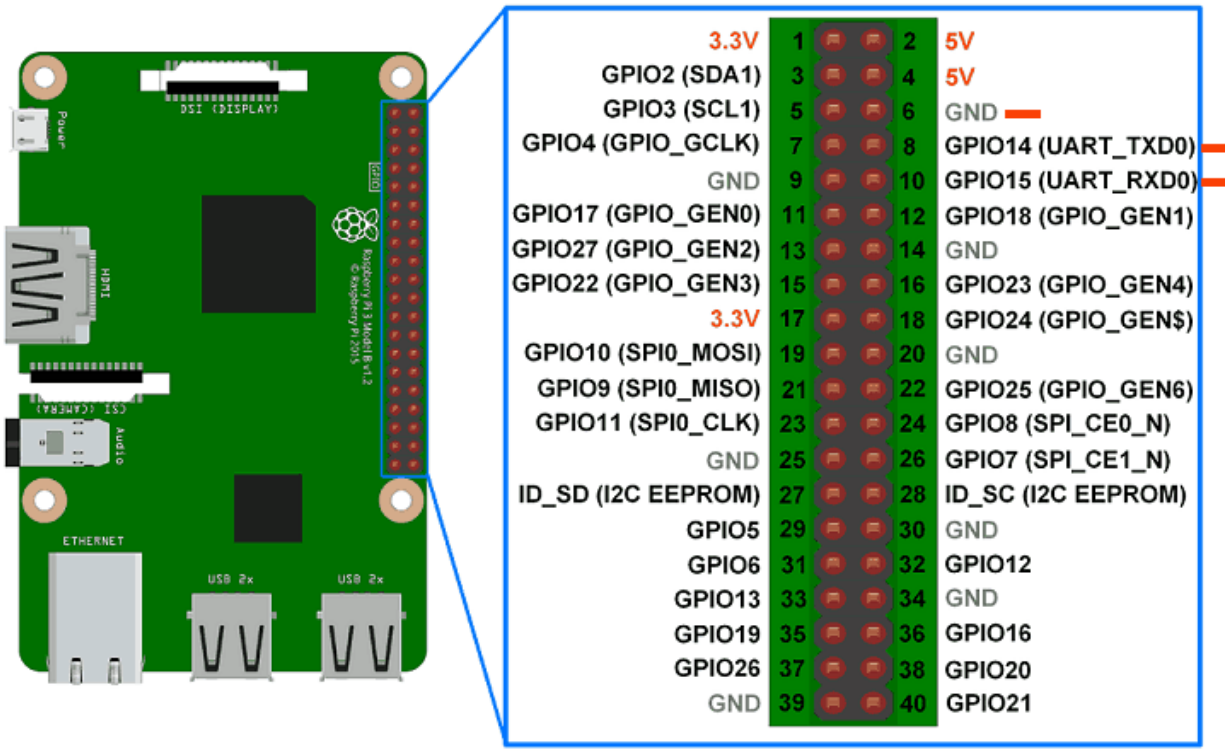
Pour flasher votre système, suivez la commande indiquée dans le `README` de la RPi3 : https://git.buildroot.net/buildroot/tree/board/raspberrypi/readme.txt?h=2019.08:

```
sudo dd if=output/images/sdcard.img of=/dev/mmcblk0 bs=1M
```

Ou `sdc` ou `sdb` au lieu de `mmcblk0` si besoin.

1.6 Démarrage du système

Insérez la carte SD dans la RPi3.
Branchez la série selon les pins de votre RPi3 :



Allumez via USB votre board et vous devriez voir le système booter !

Le login est `root` et profitez pour explorer le système. Lancer `ps` pour regarder combien de processus sont en cours d'exécution, qu'est-ce que Buildroot a généré dans `/bin`, `/lib`, `/usr` et `/etc`.

1.7 Explorer le log de build

De retour sur votre machine de build, regardez la sortie du log `build.log`. Buildroot est un peu verbose mais il affichera chaque message important d'un prefixe `>>>`. Pour avoir une idée générale de ce qui a été fait, vous pouvez lancer:

```
grep ">>>" build.log
```

Vous voyez les différents paquets être téléchargés, extraits, patchés, configurés compilés et installés.

Chapter 2

Ajout du support CAN

Objectifs:

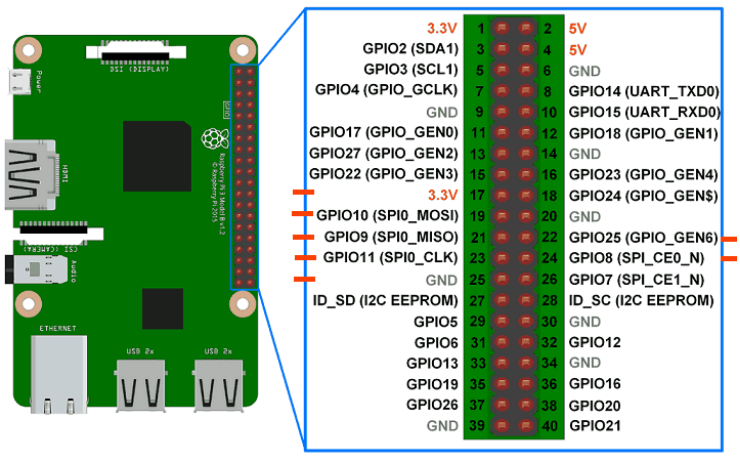
- Ajouter de nouveaux paquets à compiler
- Configurer un périphérique hardware (CAN) sous la RPi3
- Tester le nouveau système

2.1 Avant de commencer...

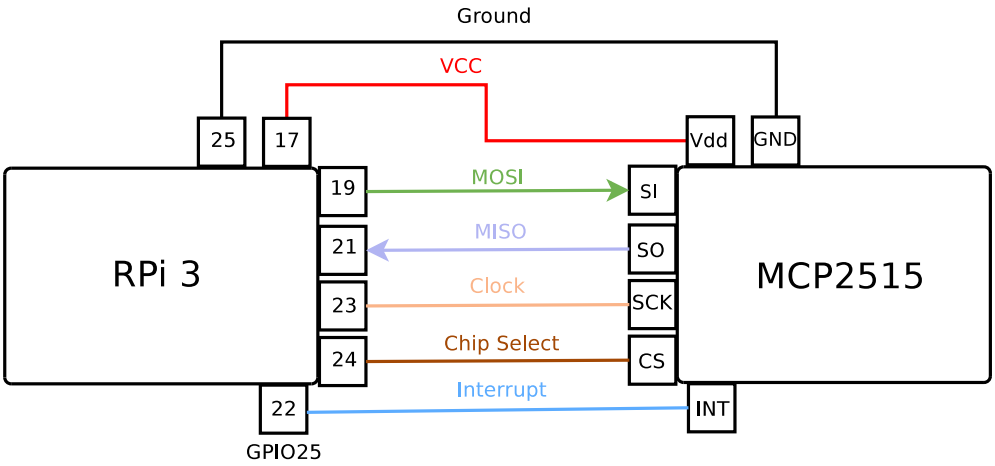
Le controlleur CAN que nous allons utiliser est un MCP2515 qui utilise un bus SPI.

Prendre un peu de temps pour lire la datasheet du MCP2515, regarder les différentes pins, le voltage en entrée, etc. Cela vous permettra de comprendre les pins suivantes utilisées.

Nous allons utiliser les pins SPI de la RPi :



Ce qui nous donnera une connexion SPI suivante :



2.2 Configuration

2.2.1 Configuration de la RPi3 : defconfig + device-tree

Pour ajouter le support d'un périphérique dans Linux, il faut:

- Ajouter un noeud dans le device tree selon le bus utilisé
- Activer le driver dans la configuration kernel

Device-tree overlay

La RPi a une particularité par rapport à certaines cartes de développpement, elle utilise un fichier de configuration de type texte pour faire des modifications hardware via un firmware.

Rechercher ce fichier :

```
$ find . -iname config.txt
./package/rpi-firmware/config.txt
```

Lire les fichiers contenus dans le dossier `board/raspberrypi3/`.

Regarder comment est-il possible de modifier le fichier `config.txt`, notamment avec ce qui est déjà effectué avec l'option `dtoverlay=pi3-miniuart-bt`. Les informations sur la syntaxe de ce fichier sont disponibles sur ce site http://elinux.org/RPiconfig. Les dtoverlay permettent d'activer des parties hardware de la raspberry pi en modifiant le Device Tree.

Pour activer le bus SPI, il faut utiliser le `dtparam` et pour activer le hardware CS, ça sera via `dtoverlay` (voir la documentation ici):

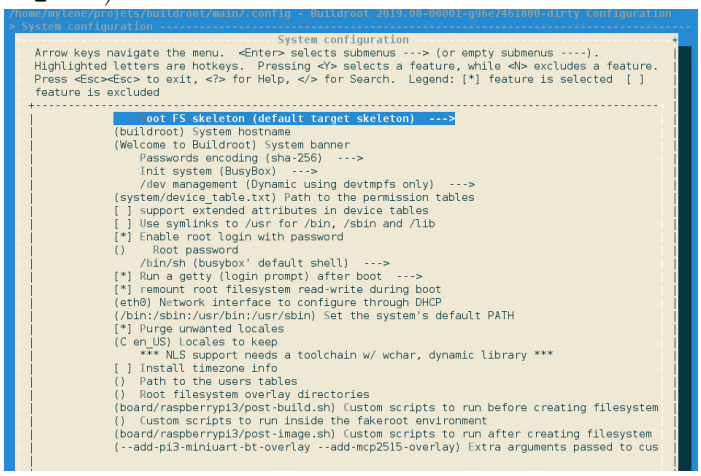
```
dtparam=spi=on
dtoverlay=spi0-hw-cs
```

En ce qui concerne le chip CAN, il faut utiliser un `dtoverlay`. Voici une documentation à lire concernant leur utilisation avec la node du controlleur CAN que nous utiliserons MCP2515.

```
dtoverlay=mcp2515-can0,oscillator=16000000,interrupt=25
```


Toutes ces nouvelles valeurs devront être ajoutées dans le fichier `config.txt` en ajoutant du code dans le script `post-image.sh` de la RPi3. Prendre exemple sur les autres arguments pour ajouter une option `--add-mcp2515-overlay`.

Une fois que le script a été modifié, il faudra ajouter l'option dans la configuration Buildroot (cf variable `BR2_ROOTFS_POST_SCRIPT_ARGS`).



Configuration kernel

Par défaut, le driver pour le controlleur CAN MCP2515 est déjà activé dans la configuration que l'on utilise du kernel.

Vérifier que la configuration `MCP251X` est activée (soit builtin `=y` soit en module `=m`) en utilisant la commande suivante :

```
make linux-menuconfig
```

Cela vous ouvrira le menuconfig spécifique au kernel (et non pas à Buildroot).

2.2.2 Configuration de Buildroot

Maintenant que la configuration coté Kernel et device-tree est réalisée, il faut activer des nouveaux paquets dans Buildroot pour pouvoir avoir des outils pour utiliser le CAN :

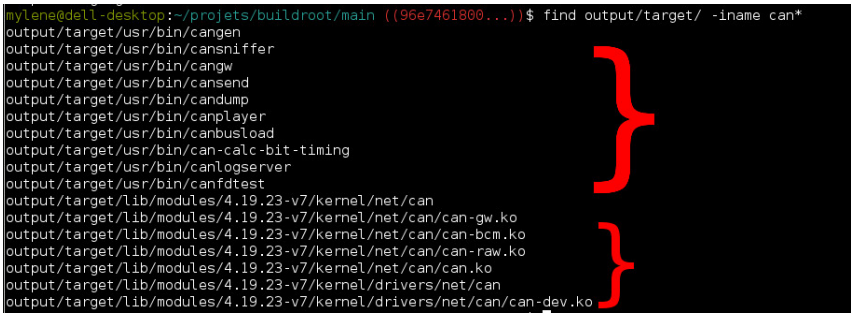
- `LIBSOCKETCAN` : Permet de controller un périphérique CAN via du code en C
- `CAN_UTILS` : Permet d'ajouter des outils en ligne de commande pour tester des périphériques CAN tel que `candump` et `cansend`.

Ouvrir le menuconfig de buildroot et chercher les paquets qui nous intéressent. Les activer, sauvegarder votre configuration et compiler une nouvelle image.

2.2.3 Compilation et test

Vérifier qu'ils sont bien compilés pour votre RPi3 avec :

```
$ find output/target/ -iname can*
```




```
$ find output/staging/ -iname can*
```

Maintenant, flasher votre carte SD avec votre nouvelle image, booter votre RPi3 avec le module CAN connecté comme il faut et tester votre interface `can0` avec :

```
$ ip link set can0 up type can bitrate 500000
$ ifconfig
$ cansend can0 01a#11223344AABBCCDD
$ candump can0
```

2.2.4 Un peu d'overlay

Pour activer automatiquement l'interface CAN au boot, on va utiliser une règle UDEV. Pour cela, on utilisera le système d'overlay dans Buildroot.

Lire la documentation de Buildroot sur les overlays et notamment la variable `BR2_ROOTFS_OVERLAY`, la documentation sur les règles udev.

Créer un overlay pour la RPi3 pour créer une règle UDEV `can.rules` suivante :

```
SUBSYSTEM=="net", DEVPATH=="/devices/platform/soc/3f204000.spi/spi_master/spi0/spi0.0/net/can0", \
RUN+="/sbin/ip link set can0 up type can bitrate 500000"
```

Configurer Buildroot pour prendre en compte votre overlay. Vérifier que votre règle est bien installée dans le dossier `target` :

```
$ find . -iname can.rules
./board/raspberrypi/overlay/etc/udev/rules.d/can.rules
./output/target/etc/udev/rules.d/can.rules
```

Et, une fois le rootfs installé sur votre carte SD, que l'interface CAN est automatiquement montée :

```
$ ifconfig
```


Chapter 3

Création d'un nouveau paquet dans Buildroot

Objectifs:

- Création d'un nouveau paquet pour *nInvaders*
- Comprendre comment ajouter des dépendances

3.1 Préparation

Après une recherche Google, trouver le site web pour *nInvaders* et télécharger le code source. Analyser son système de build et déduire quelle infrastructure de paquet est la plus appropriée.

3.2 Paquet minimal

Créer un répertoire pour ce paquet dans les sources de Buildroot : `package/ninvaders`. Créer un fichier `Config.in` avec une option pour activer ce paquet et un fichier `ninvaders.mk` minimal qui spécifiera ce qu'il faut pour juste *télécharger* le paquet.

Pour information, l'URL de téléchargement des sources *nInvaders* est `http://downloads.sourceforge.net/project/ninvaders/ninvaders/0.1.1/`.

Note: Pour y arriver, seulement deux variables sont nécessaires d'être définies dans le fichier `.mk` ainsi qu'un appel à la macro de l'infrastructure appropriée.

Maintenant, aller dans le `menuconfig`, activer *nInvaders*, et lancer une compilation `make`. Vous devriez voir l'archive *nInvaders* être téléchargée et extraite. Regarder dans `output/build/` pour voir si ça a été extrait comme attendu.

3.3 C'est parti pour la compilation !

Comme vous avez pu le voir, *nInvaders* utilise un simple `Makefile` pour son processus de build. Vous allez donc devoir définir les variables de build pour *nInvaders*. Pour faire cela, 4 variables sont fournies par Buildroot :

- `TARGET_MAKE_ENV`, qui devra être passé dans l'environnement utilisé lors de la commande `make`.
- `MAKE`, contient le vrai appel du `make` avec potentiellement quelques paramètres pour paralleliser le build.
- `TARGET_CONFIGURE_OPTS`, contient la définition de pleins de variables utiles dans les `Makefiles` comme : `CC`, `CFLAGS`, `LDFLAGS`, etc.
- `@D`, contient le chemin vers le répertoire où le code source de *nInvaders* a été extrait.

Quand vous créez des paquets Buildroot, c'est une bonne pratique de regarder comment les autres paquets sont définis. Regarder par exemple le paquet `jhead` qui sera très similaire à ce dont on a besoin pour notre paquet `ninvaders`.

Lorsque vous avez écrit l'étape de build de *nInvaders*, il est temps de tester! Mais si vous lancez seulement un `make` pour redémarrer un build, le paquet `ninvaders` ne sera pas rebuildé car il l'a été déjà été précemment.

On va donc forcer le build en supprimant complètement le répertoire des sources :

```
make ninvaders-dirclean
```

Ensuite, recommencer un build :

```
make
```

Cette fois, vous devriez voir l'étape `ninvaders 0.1.1 Building` mais elle échouera car le fichier `ncurses.h` est manquant.

3.4 Gérer les dépendances

Le fichier `ncurses.h` est manquant parce que *nInvaders* dépends de la bibliothèque `ncurses` pour l'interface sur un terminal en mode "texte". On va donc installer `ncurses` comme dépendance de *nInvaders*. Pour cela :

- Indiquer la dépendance dans le fichier `Config.in`. Utiliser le `select` pour être sûr que le paquet `ncurses` sera automatiquement sélectionné avec la sélection de `ninvaders`.
- Indiquer la dépendance dans le fichier `.mk`.

Recommencer un build du paquet avec : `make ninvaders-dirclean all` (qui est pareil que `make ninvaders-dirclean` suivi d'un `make`).

Maintenant, le paquet devrait être compilé avec succès ! Si vous jetez un oeil à `output/build/ninvaders-0.1.1/`, vous devriez voir le binaire `nInvaders`. Lancer le programme `file` sur `nInvaders` pour vérifier que c'est bien pour ARM.

`nInvaders` a bien été compilé mais il n'est pas pour autant installé dans le rootfs de notre cible !

3.5 Installation et test du programme

Si vous regardez le `Makefile` des sources de *nInvaders*, vous remarquez qu'il n'y a rien pour installer le programme. Il n'y a pas de règle `install:`.

Dans `ninvaders.mk`, on va devoir créer une *commande d'installation pour cible* qui va simplement installer le binaire `nInvaders`. Utiliser la variable `$(INSTALL)` pour cela. Regarder le paquet `jhead` comme exemple.

Rebuilder une nouvelle fois `ninvaders` et cette fois, vous devriez voir le binaire `nInvaders` dans `output/target/usr/bin/`!

Reflasher votre rootfs sur la carte SD et rebooter le système.

3.6 Ajout d'un fichier de hash

Pour finaliser notre paquet, ajouter un fichier manquant : *hash file* pour permettre de vérifier que les personnes builderont les mêmes sources que celles que vous avez utilisé pour votre paquet. Pour savoir le hash, SourceForge permet de directement connaitre cela via : aller dans la page *download* et à coté du nom du fichier, il y a une icône d'information qui vous fournit les hashes MD5 et SHA1.

Recompiler avec `make ninvaders-dirclean all`. Regarder l'output du build et avant le message `ninvaders 0.1.1 Extracting`, vous devriez voir :

```
ninvaders-0.1.1.tar.gz: OK (sha1: ....)
ninvaders-0.1.1.tar.gz: OK (md5: ....)
```

3.7 Tester la suppression d'un paquet

Pour jouer un peu avec Buildroot, réaliser les tests suivants : désactiver le paquet `ninvaders` dans le `menuconfig` et redémarrer le build via `make`. Quand le build est fini, regarder dans `output/target/`. Est-ce que *nInvaders* est toujours installé ? Si oui, pourquoi ?

