Créer un système complet avec Buildroot

Table of Contents

Le projet **Buildroot** permet de construire facilement un système embarqué et d'en ajuster finement le contenu. Pour produire des systèmes autonomes robustes, résilients et reproductibles, on préfère cette approche à l'utilisation d'une distribution pré-compilée dont le contenu est assez mal maîtrisé. On va utiliser **Buildroot** pour construire un système personnalisé pour Raspberry Pi.

Les développeurs de **Buildroot** fournissent une nouvelle version chaque trimestre. Celle de février est une version maintenue sur le long terme pendant toute l'année, alors que celles de mai, août et

novembre sont des versions intermédiaires marquant les évolutions du projet.

Environnement de travail

Commencer par créer une arborescence de travail qui contiendra tous les fichiers personnalisés, les répertoires de construction, les archives des projets compilés par **Buildroot**, etc.

```
mkdir br-lab
cd br-lab
```

Espace d'échange

Pour les configurations aux ressources limitées, il faut allouer un swapfile conséquent (32Go pour un sbc raspberry disposant de 4Go de mémoire)

Au sein de cet environnement, commencer par créer et initialiser le fichier pour contenir le swap. Par exemple, pour créer un fichier d'échange de 32 Go, on peut utiliser la commande :

```
sudo dd if=/dev/zero of=swapfile bs=1M count=32768 status=progress
```

Fichiers d'échange sur BTRFS: À partir du noyau 5.0 et supérieur, les fichiers d'échange sont pris en charge sur btfrs. Ils nécessitent encore une manipulation spéciale en plus des étapes ci-dessus.

```
sudo truncate -s 0 /swapfile
sudo chattr +C /swapfile
sudo btrfs property set /swapfile compression none
```

\\Ces commandes créent un fichier d'échange vide, désactivent COW pour ce fichier et garantissent que la compression est désactivée.

Définir les autorisations appropriées sur le fichier. Il doit être accessible en lecture et en écriture uniquement par root. Cela peut être fait avec la commande :

```
sudo chmod 600 swapfile
```

Ensuite, formater le fichier d'échange :

```
sudo mkswap swapfile
```

Configuration par défaut

Télécharger l'archive de la dernière version de Buildroot, et la décompresser:

```
wget http://www.buildroot.org/downloads/buildroot-2020.02.tar.bz2
tar xf buildroot-2020.02.tar.bz2
```

On va travailler dans un répertoire de compilation indépendant des sources de Buildroot. Ainsi on peut enchaîner plusieurs builds successifs pour différentes architectures ou différents projets en conservant les sources originales intactes. Par acquit de conscience on peut s'assurer qu'aucune modification n'interviendra dans les sources de Buildroot d'y interdire l'accès en écriture :

```
chmod -R -w buildroot-2020.02
```

Cela garantit qu'aucun fichier de configuration ne pourra être sauvegardé par erreur dans cette arborescence dans le cas d'une faute de frappe dans le nom d'une variable d'environnement sur une ligne de commande.

On va demander à Buildroot de préparer une configuration par défaut pour Raspberry Pi 4. Puis on va la modifier très légèrement dans un premier temps, et l'affinerons de manière plus importante ensuite.

La liste des configurations disponibles est à la racine du répertoire `configs/`

Buildroot intègre un nombre assez conséquent de configurations toutes prêtes pour des plateformes variées. Même si on ne trouve pas celle qui correspond exactement à la cible, il est souvent possible de choisir une configuration pour un system-on-chip proche et de l'ajuster assez facilement.

On veut construire un système pour Raspberry Pi 4, aussi on demande à Buildroot de préparer une configuration pour cette cible :

```
[buildroot-2020.02]$ make O=../build-pi4 raspberrypi4_defconfig
  [...]
#
# configuration written to /home/cpb/br-lab/build-pi4/.config
#
```

- **make**: le projet Buildroot utilise comme moteur de compilation l'outil «make». Les recettes servant à configurer ou produire une image seront ainsi des «Makefile» ou des fragments de «Makefile». Ceci n'est pas sans rappeler la commande «bitbake» employée pour remplir ce rôle lorsqu'on utilise Yocto Project.
- **O=../build-pi4**: pour éviter de polluer les sources de Buildroot, on fera tout le travail dans un répertoire indépendant. On mentionne ici le nom de ce répertoire, en l'indiquant dans la variable «O» (la lettre «O» majuscule, comme Output, pas le chiffre zéro). Comme le répertoire n'existe pas encore, Buildroot va le créer.

- **raspberrypi4_defconfig**: le fichier de configuration pour la plateforme, tel qu'il se trouve dans le répertoire «configs/» vu plus haut. Lorsque on aura personnalisé cette configuration, on verra comment l'intégrer dans une arborescence external personnelle.

Maintenant que la configuration a été préparée dans le répertoire de compilation, on peut s'y rendre pour continuer le travail, on n'aura plus besoin de revenir dans les sources de Buildroot.

```
[buildroot-2020.02]$ cd ../build-pi4/
ls
build  Makefile
ls -a
.                       .br2-external.in.openssl     build
..                      .br2-external.in.paths       .config
.br2-external.in.jpeg   .br2-external.in.toolchains  Makefile
.br2-external.in.menus  .br2-external.mk
```

La configuration ainsi préparée est enregistrée dans le fichier «.config» (dont le nom commence par un point ce qui le rend invisible à la commande «ls» sauf si elle est suivie de l'option «-a»).

Le fichier `.config` contient les définitions de constantes pour «make» :

```
#
# Automatically generated file; DO NOT EDIT.
# Buildroot 2020.02 Configuration
#
BR2_HAVE_DOT_CONFIG=y
BR2_HOST_GCC_AT_LEAST_4_9=y
BR2_HOST_GCC_AT_LEAST_5=y
BR2_HOST_GCC_AT_LEAST_6=y
BR2_HOST_GCC_AT_LEAST_7=y
 [...]
# BR2_ELF2FLT is not set
# BR2_VFP_FLOAT is not set
# BR2_PACKAGE_GCC_TARGET is not set
# BR2_HAVE_DEVFILES is not set
[build-pi4]$
```

Modification de la configuration

Dans un premier temps on va faire quelques personnalisations rapides, on y reviendra plus en détail ultérieurement.

Pour cela on demande le menu de configuration de Buildroot. Comme on l'a indiqué, on fait appel à «make» pour interagir avec Buildroot :

```
make menuconfig
  [...]
```

Un menu de configuration s'ouvre.

Le menu est affiché en utilisant la bibliothèque Ncurses. Il existe des commandes équivalentes pour afficher le menu avec les bibliothèques Qt («make xconfig») ou Gtk («make gconfig»).

Menu «Target options»

Le premier sous-menu «Target options» contient essentiellement des paramètres correspondant à certaines options du compilateur.

Dans le cas, il n'y a pas de raison de modifier le contenu de ce sous-menu mais on peut en profiter pour jeter un œil à la variété des processeurs supportés directement par Buildroot.

Menu «Build options»

Le sous-menu «Build options» contient de nombreux paramètres permettant de configurer la manière de compiler le système. Une modification est conseillée : celle de l'option «Download dir». Il s'agit de l'emplacement où les packages téléchargés sont stockés avant d'être extraits et compilés. Par défaut il s'agit du sous-répertoire «dl» dans le répertoire de Buildroot. Mais on a supprimé les droits d'écriture sur ce dernier. Il faut donc extraire le répertoire «dl» en le remontant d'un cran. Ceci permet en outre de mutualiser les téléchargements entre les différents builds que l'on est amené à réaliser.

Editer le champ «Download dir» pour insérer un «../» avant «dl».

Menu «Toolchain»

Le sous-menu «Toolchain» permet de configurer des paramètres de la chaîne de cross-compilation. Tout d'abord on peut choisir d'utiliser une chaîne de compilation produite par Buildroot (le cas par défaut) ou une toolchain externe, précompilée et téléchargée depuis certains sites de références (Linaro par exemple). Bien que cette dernière solution soit plus rapide, on déconseille pour des raisons de pérennité et de reproductibilité du système de compilation.

Un second choix important concerne la bibliothèque C à employer pendant la production du compilateur et à installer sur la cible. C’est une décision qui dépend beaucoup du code métier. La bibliothèque C est un point-clé du système ; c’est elle qui permet d’entrer dans le noyau pour bénéficier de ses services (les appels-système). Le choix par défaut est celui de la «uclibc-ng», une bibliothèque spécialement dédiée aux systèmes embarqués restreints. C'est la solution qu'on va conserver ici, mais les alternatives «glibc» et «musl» sont tout aussi acceptables.

On va modifier deux options:

- activer **«Enable WCHAR support»** afin d'intégrer dans la bibliothèque C le suport des caractères larges (multi-octets) qui est nécessaire pour de nombreux packages proposés par Buildroot.
- activer également **«Thread library debug»** pour pouvoir utiliser le débogueur «gdbserver» lors de la mise au point du code applicatif.

Enfin, activer l'option «Build cross gdb for the host» comme on le voit sur la figure 6, pour produire tout de suite le débogueur qui fonctionnera sur la machine hote (le PC).

Pour le moment, on se limite à ces modifications, on examinera le contenu des autres menus ultérieurement. on peut quitter en sauvegardant la configuration.

Compilation et test

Compilation de l'image standard

Pour lancer la production de cette image, rien de plus simple :

```
make
 [...]
```

La première compilation est plus longue que les suivantes, du fait de la génération de la toolchain et de la compilation du noyau. La durée exacte dépend beaucoup de la machine hôte et de la connexion Internet, mais on peut considérer environ une heure sur une machine d'entrée de gamme. La compilation se termine ainsi :

```
INFO: vfat(boot.vfat): cmd: "MTOOLS_SKIP_CHECK=1 mcopy -bsp -i
'/home/cpb/br-lab/build-pi4/images/boot.vfat' '/home/cpb/br-lab/build-
pi4/images/rpi-firmware/overlays' '::'" (stderr):
INFO: vfat(boot.vfat): adding file 'zImage' as 'zImage' ...
INFO: vfat(boot.vfat): cmd: "MTOOLS_SKIP_CHECK=1 mcopy -bsp -i
'/home/cpb/br-lab/build-pi4/images/boot.vfat' '/home/cpb/br-lab/build-
pi4/images/zImage' '::'" (stderr):
INFO: hdimage(sdcard.img): adding partition 'boot' (in MBR) from 'boot.vfat'
...
INFO: hdimage(sdcard.img): adding partition 'rootfs' (in MBR) from
'rootfs.ext4' ...
INFO: hdimage(sdcard.img): writing MBR
[build-pi4]$
```

De nouveaux sous-répertoires sont apparus dans le répertoire de travail.

```
ls
build  host  images  Makefile  staging  target
```

- C'est dans `build/` que Buildroot travaille pour compiler les différents packages qu'il produit.

On y trouve un sous-répertoire pour chaque package, y compris le kernel, la toolchain et les outils qu'il utilise sur la machine de compilation (préfixés par «host-»).
- Le répertoire `host` contient la chaîne de cross-compilation. On pourra l'appeler directement ou demander à Buildroot de l'exporter dans une archive pour la partager avec d'autres développeurs.
- Le répertoire `images` contient comme son nom l'indique les images produites : celle du noyau, du root filesystem, du device tree, du bootloader, etc. Pour simplifier l'installation sur le Raspberry Pi, on y trouvera même une image de l'ensemble de la carte SD.
- Le répertoire `stagging` est en réalité un lien symbolique vers le «sysroot» de la toolchain. Son rôle reste un peu mystérieux pour moi, on n'a jamais eu besoin de m'en préoccuper.
- On trouve dans `target` une copie de l'arborescence du root filesystem de la cible. Elle est utilisée comme stockage avant la production de l'image finale. Toutefois les propriétaires et groupes des fichiers ne sont pas représentatifs de ceux qu'on retrouvera sur la cible. C'est pour cela qu'on y trouve un fichier d'avertissement avec un nom surprenant de prime abord :

```
ls target/
bin  lib      media  proc  sbin                                tmp
dev  lib32    mnt    root  sys                                 usr
etc  linuxrc  opt    run   THIS_IS_NOT_YOUR_ROOT_FILESYSTEM  var
```

Pour le moment on s'intéresse au répertoire «images»

```
ls images/
bcm2711-rpi-4-b.dtb  rootfs.ext2  rpi-firmware  zImage
boot.vfat            rootfs.ext4  sdcard.img
[build-pi4]$
```

Les fichiers qu'il contient sont les suivants :

- **«zImage»** le noyau Linux et «bcm2711-rpi-4-b.dtb le device tree décrivant le matériel,
- **«rpi-firmware/»** les fichiers précompilés du bootloader spécifique au Raspberry Pi,
- **«boot.vfat»** une copie binaire du contenu de la première partition contenant les éléments ci-dessus, formatée en VFAT (Fat 32),
- **«rootfs.ext2»** et le lien symbolique «rootfs.ext4» représentent une image binaire du contenu de la seconde partition (regroupant l'arborescence que on a vue dans le répertoire «target/» plus haut),
- **«sdcard.img»** une image binaire contenant une table des partitions et les deux images de partitions précédentes. C'est ce dernier fichier que on va copier sur une carte micro-SD.

Préparation de la carte micro-SD

Exécuter la commande suivante:

```
lsblk
NAME          MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
sda             8:0    0 465,8G  0 disk
├─sda1          8:1    0   400G  0 part  /home/testing
└─sda2          8:2    0     8G  0 part  [SWAP]
```

```
sdb               8:16   0 931,5G  0 disk
└─sdb1            8:17   0   512G  0 part  /media/cpb/USB-EXT
nvme0n1         259:0    0 232,9G  0 disk
├─nvme0n1p1     259:1    0   512M  0 part  /boot/efi
├─nvme0n1p2     259:2    0 224,6G  0 part  /
└─nvme0n1p3     259:3    0   7,8G  0 part
  └─cryptswap1 253:0    0   7,8G  0 crypt
```

On voit tous les périphériques «blocks» présents : «nvme0n1» le SSD principal du système, «sda» un disque dur SATA interne de travail, «sdb» un disque externe USB. Connecter ensuite une carte micro-SD insérée dans un adpateur USB, puis relancer la même commande après quelques secondes d'attente pour que l'auto-monteur ait terminé son travail :

```
lsblk
NAME            MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
sda               8:0    0 465,8G  0 disk
├─sda1            8:1    0   400G  0 part  /home/testing
└─sda2            8:2    0     8G  0 part  [SWAP]
sdb               8:16   0 931,5G  0 disk
└─sdb1            8:17   0   512G  0 part  /media/cpb/USB-EXT
sdc               8:32   1   3,8G  0 disk
├─sdc1            8:33   1    32M  0 part  /media/cpb/0B09-D1D7
└─sdc2            8:34   1   256M  0 part  /media/cpb/0af348dc-34fe-4482-
b341-a87
nvme0n1         259:0    0 232,9G  0 disk
├─nvme0n1p1     259:1    0   512M  0 part  /boot/efi
├─nvme0n1p2     259:2    0 224,6G  0 part  /
└─nvme0n1p3     259:3    0   7,8G  0 part
  └─cryptswap1 253:0    0   7,8G  0 crypt
```

Le périphérique apparu est «/dev/sdc/». Démonter les partitions montées automatiquement et copier l'image produite par Buildroot sur ce périphérique. Cette étape est potentiellement dangereuse (comme tout ce qu'on préfixe par «sudo»), il faut être attentif pour ne pas écraser le disque système par mégarde.

```
umount  /dev/sdc?
sudo  cp  images/sdcard.img  /dev/sdc
```

Une fois la copie terminée, insérer la carte SD dans le Raspberry Pi 4 et le démarrer.

Test de l'image standard

Connecter un écran sur l'une des sorties mini-HDMI, pour voir les traces de boot avec les fameuses framboises emblèmatiques du Raspberry Pi.

On peut connecter un terminal («minicom») sur le port série du Raspberry Pi avec un

câble USB-Série (ne pas connecter le fil rouge).

On peut se connecter avec l'identité «root» (pas de mot de passe pour le moment), et passer quelques commandes.

le clavier est configuré en «Qwerty» ce qui nécessite une petite gymnastique si on ne dispose que d'un clavier «Azerty».

```
Welcome to Buildroot
buildroot login: root
# uname -a
Linux buildroot 4.19.97-v7l #1 SMP Sun Mar 8 23:43:48 CET 2020 armv7l
GNU/Linux
# cat /proc/device-tree/model
Raspberry Pi 4 Model B Rev 1.1#
# free
              total        used        free      shared  buff/cache
available
Mem:        1962164       20664     1938580          48        2920
1929232
Swap:             0           0           0
#
```

On voie ici Buildroot fonctionner sur un Raspberry Pi 4 avec 2 Gb de RAM. La commande «ps» affiche la liste des processus présents :

```
# ps
PID   USER     COMMAND
    1 root     init
    2 root     [kthreadd]
    3 root     [rcu_gp]
    4 root     [rcu_par_gp]
    6 root     [kworker/0:0H-kb]
    7 root     [kworker/u8:0-ev]
    8 root     [mm_percpu_wq]
    9 root     [ksoftirqd/0]
   10 root     [rcu_sched]
   11 root     [rcu_bh]
   12 root     [migration/0]
   13 root     [cpuhp/0]
   14 root     [cpuhp/1]
   15 root     [migration/1]
   16 root     [ksoftirqd/1]
   17 root     [kworker/1:0-eve]
   18 root     [kworker/1:0H-kb]
```

```
   19 root     [cpuhp/2]
   20 root     [migration/2]
   21 root     [ksoftirqd/2]
   22 root     [kworker/2:0-eve]
   24 root     [cpuhp/3]
   25 root     [migration/3]
   26 root     [ksoftirqd/3]
   29 root     [kdevtmpfs]
   30 root     [netns]
   32 root     [khungtaskd]
   33 root     [oom_reaper]
   34 root     [writeback]
   35 root     [kcompactd0]
   36 root     [crypto]
   37 root     [kblockd]
   38 root     [watchdogd]
   39 root     [kworker/2:1-ipv]
   40 root     [rpciod]
   41 root     [kworker/u9:0]
   42 root     [xprtiod]
   43 root     [kswapd0]
   44 root     [nfsiod]
   55 root     [kthrotld]
   56 root     [iscsi_eh]
   57 root     [kworker/u8:1]
   59 root     [kworker/1:1-mm_]
   60 root     [DWC Notificatio]
   61 root     [vchiq-slot/0]
   62 root     [vchiq-recy/0]
   63 root     [vchiq-sync/0]
   64 root     [vchiq-keep/0]
   65 root     [SMIO]
   66 root     [kworker/0:2-eve]
   67 root     [irq/37-brcmstb_]
   68 root     [irq/38-mmc1]
   69 root     [irq/38-mmc0]
   70 root     [kworker/0:3-eve]
   72 root     [mmc_complete]
   73 root     [kworker/0:1H-mm]
   74 root     [kworker/3:1H-kb]
   75 root     [kworker/3:2H]
   76 root     [jbd2/mmcblk0p2-]
   77 root     [ext4-rsv-conver]
   79 root     [kworker/2:1H-kb]
   80 root     [kworker/1:1H-kb]
   94 root     /sbin/syslogd -n
   98 root     /sbin/klogd -n
  123 root     [kworker/2:2H]
  124 root     [ipv6_addrconf]
  145 root     -sh
  164 root     /sbin/getty -L tty1 0 vt100
```

```
  169 root     [kworker/3:0-eve]
  170 root     [kworker/3:1-mm_]
  171 root     [kworker/3:2]
  172 root     ps
#
```

Hormis les threads internes du noyau (toutes les tâches avec des noms entre crochets), on observe la présence de seulement six processus :

- **«init»**: le premier processus qui est chargé d'abord de l'initialisation du système depuis l'espace utilisateur;
- **«syslogd»** et **«klogd»** sont deux démons chargés de l'enregistrement des messages du système;
- **«getty»**: le service qui attend les connexions sur le terminal tty1 (écran HDMI + clavier USB). On est connecté sur la console série et le «getty» correspondant a laissé sa place au shell;
- **«sh»** le shell sur lequel on est connectés;
- la commande **ps** elle-même.

Voilà un système dont le contenu est bien sous contrôle !

Cette première image est très toutefois limitée, elle ne comprend guère que Busybox, et le compte «root» est même accessible sans mot de passe. on va améliorer sa configuration.

Affinement de la configuration

on peut faire une toute une série d'améliorations, afin d'obtenir un système un peu plus convivial, accueillant un autre utilisateur que «root» par exemple ou renforçant la partition principale contre les risques de coupures d'alimentation intempestives.

Menu «System Configuration»

On relance :

```
make  menuconfig
```

Et on s'intéresse au menu «System configuration».

Intervenons sur quelques options de ce menu :

- **«System hostname»**: choisir un nom plus représentatif pour la carte. Il apparaîtra dans l'invite de connexion, et il est possible de l'afficher dans le prompt du shell.
  - **Nouvelle valeur** : «R-Pi».
- **«System banner»** : cette petite phrase s'affichera au démarrage avant la proposition de connexion ; on peut la personnaliser à volonté. Sur la plupart des systèmes qu'on configure, on affiche ainsi le nom du projet et celui de mon client.
  - **Nouvelle valeur**: «Welcome on board!».
- **«Enable root login with password»**: si le système a la moindre chance de se retrouver connecté à Internet, il est préférable de désactiver cette option. En effet le compte «root» sera

le premier visé par les attaques automatiques par force brute. Si cette option est désactivée, il faudra intégrer la commande «sudo» afin de pouvoir réaliser les opérations d'administration.
    - Sur un système expérimental, on laisse la valeur originale : [*].
- **«Root password»**: de même, il est conseillé de choisir pour tous les comptes des mots de passe solides (longs, assez faciles à retenir mais difficiles à deviner). Pour cette démonstration prenons un mot de passe ridiculement simple.
    - **Nouvelle valeur** : «root».
- **«Remount root filesystem read-write during boot»** : sur un système embarqué où l'alimentation peut être coupée à tout moment, il est conseillé de conserver le système de fichiers principal en lecture-seule. On le basculera en lecture-écriture temporairement pour des modifications de configuration par exemple.
    - **Nouvelle valeur**: [ ].
- **«Network interface to configure through DHCP»**: suivant la situation, on utilisera ou non une configuration réseau par DHCP. Si tel est le cas, on indique ici le nom de l'interface Ethernet.
    - **Valeur conservée**: «eth0».
- **«Path to the users tables»**: on indique ici le chemin d'accès pour un fichier contenant la liste des utilisateurs.

On va devoir fournir un fichier pour les utilisateurs supplémentaires (autres que «root»). Ce sera le premier fichier de configuration personnalisé mais on en ajoutera d'autres par la suite. La méthode la plus adéquate à mon sens consiste à ajouter une arborescence externe (external tree) regroupant tous les fichiers personnels.

On va donc quitter quelques instants le menu de configuration (en sauvegardant les modifications) pour créer le fichier des utilisateurs dans un external tree.

External Tree

Une arborescence externe est un répertoire placé en-dehors des sources de Buildroot et en-dehors du répertoire de compilation «build-pi4» dans lequel on peut ajouter des recettes supplémentaires (pour intégrer des packages absents de ceux connus par Buildroot, ou pour intégrer du code métier applicatif), des paramètres de configuration pour Busybox ou pour le noyau, ou encore des fichiers personnalisés à ajouter dans l'arborescence de la cible. on va commencer par un external tree très simple.

Remonter d'un cran dans le système de fichiers, puis créer un répertoire. Par exemple «pi4-config».

```
cd  ..
mkdir  pi4-config
ls
build-pi4          buildroot-2020.02.tar.bz2  pi4-config
buildroot-2020.02  dl
cd  pi4-config/
```

Pour que cette arborescence soit valide pour Buildroot, on doit y créer trois fichiers :

- **«external.desc»** qui contient le nom et la description de l'arborescence externe ;
- **«external.mk»** un fragment de Makefile — initialement vide — indiquant comment compiler

les packages contenus dans cet external tree ;
- **«Config.in»** un fichier décrivant les sous-menus à ajouter au menu de configuration (initialement vide également).

```
echo  "name: PI4_CONFIG"  >  external.desc
echo  "desc: Custom configuration for Pi 4"  >>  external.desc
cat external.desc
name: PI4_CONFIG
desc: Custom configuration for Pi 4

touch  external.mk
touch  Config.in
```

Ajouter un répertoire dans lequel on stockera les éléments de configuration, à commencer par la table des utilisateurs.

```
mkdir configs
```

Table des utilisateurs

on va remplir «Path to the users tables» avec le nom d'un fichier qui contient la liste des utilisateurs. Il doit y avoir un compte par ligne. Les champs, séparés par une espace, sont les suivants :

- **«login**: identifiant de connexion du compte, une chaîne de lettres minuscules, de chiffres et de tirets hauts ou bas. Le login «root» ne doit pas être indiqué dans ce fichier.
- **«user identifier» (UID)**: numéro d'identification de l'utilisateur. On peut indiquer -1 pour que l'attribution d'un numéro libre soit automatique. on aura besoin un peu plus loin d'un UID connu, on précise donc «1000».
- **«group»**: groupe principal de l'utilisateur. Généralement le même nom que le login, ou alors un groupe global pour tous les comptes, comme «users».
- **«group identifier» (GID)** : numéro du groupe. Comme pour le champ UID, on précise «1000» pour connaître à l'avance le numéro de groupe.
- **«password»** : le mot de passe, en clair si précédé d'un «=», chiffré sinon. Si le mot de passe est «!», pas de connexion possible (compte utilisé pour un démon système par exemple).
- **«home»** : répertoire personnel. On peut indiquer «-» si on ne veut pas disposer de répertoire personnel. C'est le cas lorsqu'un service doit s'exécuter avec une identité autre que «root» sans permettre de connexion.
- **«shell»**: le shell de connexion. Sur le système minimal, «/bin/sh» est le shell «ash» inclus dans Busybox.
- **«groups»**: ce champ contient la liste des groupes supplémentaires auxquels appartient l'utilisateur («-» si aucun).
- **«gecos»**: des informations sur le compte, comme le nom en clair de l'utilisateur. Ce dernier champ peut contenir éventuellement des espaces.

On crée donc un fichier «configs/users.tbl» :

```
echo  "rpi 1000 rpi 1000 =rpi /home/rpi /bin/sh - Raspberry Pi User"  >
```

```
configs/users.tbl
```

Comme on le voit, on a ajouté un utilisateur «rpi» appartenant à son propre groupe, et dont le mot de passe est... «rpi» !

Le stockage du mot de passe en clair dans ce fichier de configuration est acceptable pour une expérimentation comme celle-ci mais doit être proscrit en environnement de production. En effet si la sécurité de la machine utilisée pour la compilation est compromise, c'est l'ensemble des systèmes déployés qui sont vulnérables

Intégration de l'arborescence externe et test

Pour que l'arborescence externe soit prise en considération, il va falloir l'indiquer une fois sur la ligne de commande de «make menuconfig» puis elle sera mémorisée automatiquement.

```
cd ../build-pi4/
export BR2_EXTERNAL=../pi4-config/
make menuconfig
```

Une nouvelle ligne est apparue au bas du menu de configuration: sous-menu «External options»

Comme on n'a pas indiqué de menu ou de packages supplémentaires, le sous-menu concerné indique seulement la description de cette option.

Pour le moment, il faut renseigner la table des utilisateurs. on peut donc donner le chemin dans le menu «System configuration». On utilise une variable qui va être automatiquement créée par Buildroot : «BR2_EXTERNAL_PI4_CONFIG_PATH». Elle représente la racine de l'arborescence externe. Le nom de la variable a été composé à partir du champ «name» que on a inscrit dans le fichier «external.desc». Le nom de l'external tree était «PI4_CONFIG», celui de la variable est donc «BR2_EXTERNAL_PI4_CONFIG_PATH».

Cette approche donne de la souplesse, il sera possible d'utiliser le même external tree dans d'autres builds. on verra même comment extraire la configuration de Buildroot qu'on réalise pour la réutiliser avec une version ultérieure par exemple.

On a donc modifié l'option «Path to the users tables» du menu «System configuration» pour y inscrire la chaîne : «$(BR2_EXTERNAL_PI4_CONFIG_PATH)/configs/users.tbl».

Après sauvegarde, on relance la compilation du système :

```
make
```

La durée est très courte, car il n'y a que quelques fichiers de paramétrage système à modifier.

Après réécriture de la carte micro-SD comme on l'a fait plus haut, on peut démarrer le Raspberry Pi :

```
Welcome on board!
R-Pi login:
```

On peut déjà remarquer la bonne prise en compte du message de bienvenue et du nom d'hôte («R-Pi» au lieu de «Buildroot» précédemment). On se connecte sous la nouvelle identité ajoutée :

```
R-Pi login: rpi
Password: (rpi)
$ pwd
/home/rpi
$ exit
```

La connexion sur le nouveau compte a fonctionné, vérifier si «root» est bien muni d'un mot de passe cette fois :

```
Welcome on board!
R-Pi login: root
Password: (root)
# pwd
/root
#
```

on a placé le système de fichiers en lecture seulement, vérifions cela :

```
# echo  HELLO  >  file.txt
-sh: can't create file.txt: Read-only file system
#
```

L'écriture est bien interdite. on peut toutefois remonter le système de fichiers en lecture-écriture :

```
# mount  /  -o  remount,rw
[   73.376221] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
# echo  HELLO  >  file.txt
# ls
file.txt
#
```

Le message du kernel qui apparaît sur cette console après la commande «mount» est normal, il annonce que la partition a été remontée.

on peut aussi remettre le système de fichiers en lecture seule :

```
# mount  /  -o  remount,ro
[   89.070566] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
# rm  -f  file.txt
rm: can't remove 'file.txt': Read-only file system
#
```

Le système est bien en lecture seule à nouveau. Le système n'est plus vulnérable aux coupures d'alimentation électrique.

Ajout de contenu

Ajout de packages supportés par Buildroot

L'ajout de logiciels proposés par Buildroot est très simple, il suffit de se rendre dans le menu «Target packages» visible sur la figure 14, d'explorer ses sous-menus et de piocher dans les applications proposées pour composer le contenu de le système.

Par exemple, dans le menu «Networking applications» on ajoute le serveur SSH Dropbear :

Cocher également l'option «disable reverse DNS lookups» pour accélérer les connexions.

Ajouter l'éditeur Gnu Nano se trouvant dans le sous-menu «Text editors and viewers».

Ajout de scripts

On a précédemment vu comment remonter le système de fichiers, initialement en lecture seule, pour y accéder en lecture-écriture puis le replacer en lecture seule à nouveau. Ces commandes ne sont pas compliquées, mais durant la phase de mise au point on peut être amené à les taper plusieurs dizaines de fois par jour. On a donc l'habitude de créer deux petits scripts, qu'on appelle simplement «rw» et «ro», qui sont installés sur la cible et permettent très rapidement de changer le mode d'accès à l'arborescence.

On va placer les deux scripts dans une arborescence spécifique au sein de l'external tree créé plus haut. Ensuite on indiquera à Buildroot de venir ajouter le contenu de cette arborescence dans le système de fichiers de la cible.

Créer un premier répertoire dans l'external tree, correspondant à la racine de mon arborescence :

```
cd  ../pi4-config/
mkdir  custom-rootfs
```

On souhaite placer mes scripts dans le répertoire «/usr/bin» de ma cible. On crée donc ce repertoire dans mon arborescence :

```
mkdir  -p  custom-rootfs/usr/bin/
```

Puis y créer le script »rw»

```
nano  custom-rootfs/usr/bin/rw
```

Avec le contenu suivant :

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

```
mount / -o remount,rw
```

Et le script «ro» :

```
nano  custom-rootfs/usr/bin/ro
```

contenant :

```
#!/bin/sh

mount / -o remount,ro
```

Il faut rendre les scripts exécutables :

```
chmod  +x  custom-rootfs/usr/bin/*
```

Il faut indiquer à Buildroot la présence de cette nouvelle arborescence à venir ajouter, superposer à celle d'origine. Il s'agit de l'entrée «Root filesystem overlay directories» du menu «System configuration». Comme précédemment, on fait référence au répertoire de l'external tree.

```
cd  ../build-pi4/
make  menuconfig
```

On remplis donc l'entrée «Root filesystem overlay directories» ainsi :

```
$(BR2_EXTERNAL_PI4_CONFIG_PATH)/custom-rootfs
```

Lancer la nouvelle production d’image (qui ne dure que quelques dizaines de secondes, le temps de compiler Dropbear, Nano et de reconstruire l'image).

```
Welcome on board!
R-Pi login: root
Password: (root)
```

Vérifier si le serveur «dropbear» est bien démarré et si la commande «nano» est présente :

```
# ps |  grep  drop
  144 root     /usr/sbin/dropbear -R
  156 root     grep drop
# nano  --version
 GNU nano, version 4.7
 (C) 1999-2011, 2013-2019 Free Software Foundation, Inc.
 (C) 2014-2019 the contributors to nano
 Email: nano@nano-editor.org    Web: https://nano-editor.org/
```

```
 Compiled options: --enable-tiny --disable-nls --disable-utf8
```

Les packages qu'on a demandés sont bien présents. Le système de fichiers doit toujours être en lecture seulement, vérifions si le script de passage en lecture-écriture fonctionne :

```
# touch  my-file
touch: my-file: Read-only file system
# rw
[  244.951926] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
# touch  my-file
# ls
my-file
#
```

Parfait, on peut revenir en lecture seule :

```
# ro
[  266.057973] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
# rm  -f  my-file
rm: can't remove 'my-file': Read-only file system
#
```

Le système est bien sécurisé contre les coupures d'alimentation et il offre suffisament de souplesse pour intervenir dans le système de fichiers si besoin.

Surcharge de fichiers de configuration

On n'a pas encore utilisé le réseau. Si on connecte un câble Ethernet avant de démarrer le Raspberry Pi, celui-ci peut obtenir directement une adresse IP si un serveur DHCP est présent sur le réseau local :

```
Welcome on board!
R-Pi login: root
Password: (root)
# ip  addr
1: lo: >LOOPBACK,UP,LOWER_UP< mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: >BROADCAST,MULTICAST,UP,LOWER_UP< mtu 1500 qdisc mq qlen 1000
    link/ether dc:a6:32:02:57:3d brd ff:ff:ff:ff:ff:ff
    inet 192.168.3.11/24 brd 192.168.3.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 2a01:e0a:22:ea90:dea6:32ff:fe02:573d/64 scope global dynamic
       valid_lft 86366sec preferred_lft 86366sec
```

```
    inet6 fe80::dea6:32ff:fe02:573d/64 scope link
       valid_lft forever preferred_lft forever
#
```

Il est toutefois des situations où l'on préfère utiliser une adresse IP fixe. Pour cela, il faut modifier le fichier `/etc/network/interfaces`. De même, le serveur DHCP fournit ici l'adresse du serveur DNS local. on peut en fixer un statiquement dans `/etc/resolv.conf`.

Le plus simple consiste à surcharger ces deux fichiers dans l'arborescence external tree :

```
cd  ../pi4-config/
mkdir  -p  custom-rootfs/etc/network/
nano  custom-rootfs/etc/network/interfaces
```

Contenu du fichier «interfaces» (bien entendu il faut l'ajuster selon la situation) :

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
  address 192.168.3.200
  netmask 255.255.255.0
  gateway 192.168.3.254
```

```
nano  custom-rootfs/etc/resolv.conf
```

Contenu du fichier «resolv.conf» :

```
nameserver 1.1.1.1
```

Après avoir regénéré l'image, l'avoir copiée sur carte micro-SD et démarré le Raspberry Pi, se connecter en SSH depuis mon PC :

```
ssh  root@192.168.3.200
The authenticity of host '192.168.3.200 (192.168.3.200)' can't be
established.
ECDSA key fingerprint is SHA256:sUFGwSyZCG/gWkEO4r9OmGgMFnZgs1LuT9fQgESIdTs.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.3.200' (ECDSA) to the list of known
hosts.
root@192.168.3.200's password: (root)
# uname  -a
Linux R-Pi 4.19.97-v7l #1 SMP Sun Mar 8 23:43:48 CET 2020 armv7l GNU/Linux
#
```

On peut vérifier que le Raspberry Pi accède également à Internet :

```
# ping  www.kernel.org
PING www.kernel.org (136.144.49.103): 56 data bytes
64 bytes from 136.144.49.103: seq=0 ttl=53 time=17.657 ms
64 bytes from 136.144.49.103: seq=1 ttl=53 time=17.413 ms
64 bytes from 136.144.49.103: seq=2 ttl=53 time=17.844 ms
^C
--- www.kernel.org ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 17.413/17.638/17.844 ms
# wget  https://www.kernel.org
wget: not an http or ftp url: https://www.kernel.org
# exit
Connection to 192.168.3.200 closed.
```

Le cible accède bien à Internet, mais on ne peut pas télécharger une page en utilisant le protocole HTTPS, la commande «wget» n'acceptant que HTTP ou FTP.

Utilitaires implémentés par Busybox

La commande «wget» est implémentée en utilisant l'utilitaire Busybox. on peut modifier la configuration de ce dernier avec une commande particulière de Buildroot :

```
make  busybox-menuconfig
```

Un écran de configuration ressemblant à celui de Buildroot apparaît.

Se rendre dans le menu «Networking Utilities», et valider l'option «Support HTTPS using internal TLS code» pour l'application «wget».

Bien entendu, on en profitera pour regarder les commandes implémentées dans Busybox, et en ajouter — ou en retirer — quelques unes suivant la destination du système embarqué.

Après compilation, installation et reboot du Raspberry Pi, on observe:

```
# wget  https://www.kernel.org
Connecting to www.kernel.org (136.144.49.103:443)
wget: note: TLS certificate validation not implemented
wget: can't open 'index.html': Read-only file system
#
```

«wget» ne peux pas écrire dans le répertoire courant puisque le système est en lecture seulement. Se déplacer alors dans «/tmp» sur lequel est monté un «tmpfs» (une sorte de ramdisk) utilisable en écriture :

```
# cd  /tmp
# wget  https://www.kernel.org
Connecting to www.kernel.org (136.144.49.103:443)
```

```
wget: note: TLS certificate validation not implemented
saving to 'index.html'
index.html           100% |*******************************| 16749  0:00:00
ETA
'index.html' saved
#
```

Tout s'est bien passé, «wget» a pu télécharger une page en utilisant le protocole HTTPS.

Ajout de code métier

Il est rare qu'un système embarqué soit composé uniquement d'applicatifs standards et de fichiers de configuration. Cela m'est arrivé lorsqu'il s'agissait de faire de petits serveurs pour des tâches simples (référence NTP, serveur DHCP, DNS local, etc.) mais ce n'est pas le cas le plus courant d'utilisation de Buildroot.

Si on doit ajouter du code métier, il y a plusieurs approches possibles. Tout d'abord, pendant une phase de développement, on va devoir coder et tester le projet indépendamment du BSP. Pour cela on doit disposer de la toolchain produite par Buildroot.

Extraction de la toolchain

Il est possible d'appeler directement le cross-compiler fourni par Buildroot en appelant les exécutables «arm-linux-gcc» ou «arm-linux-g++» qui se trouvent dans le sous-répertoire «host/bin/» du répertoire de travail.

```
host/bin/arm-linux-gcc  --version
arm-linux-gcc.br_real (Buildroot 2020.02) 8.3.0
Copyright © 2018 Free Software Foundation, Inc.
Ce logiciel est un logiciel libre; voir les sources pour les conditions de
copie.  Il n'y a
AUCUNE GARANTIE, pas même pour la COMMERCIALISATION ni L'ADÉQUATION À UNE
TÂCHE PARTICULIÈRE.
```

Toutefois ceci peut s'avérer gênant si le développeur applicatif est dans une équipe différente de celle du concepteur du BSP. Pour cela, Buildroot permet d'extraire la toolchain et de l'installer sur une autre machine.

On appelle :

```
make  sdk
```

qui fournit une archive :

```
ls  images/
```

```
arm-buildroot-linux-uclibcgnueabihf_sdk-buildroot.tar.gz  rootfs.ext4
bcm2711-rpi-4-b.dtb                                       rpi-firmware
boot.vfat                                                 sdcard.img
rootfs.ext2                                               zImage
```

on peut copier cette archive sur une autre machine ou à un endroit plus approprié de la même machine, et l'extraire. Par exemple on va la placer dans «/opt/» sur mon poste.

```
sudo  tar  xf  images/arm-buildroot-linux-uclibcgnueabihf_sdk-
buildroot.tar.gz  -C  /opt/
ls  /opt/
arm-buildroot-linux-uclibcgnueabihf_sdk-buildroot
picomono
picoscope
poky
sublime_text
[build-pi4]$
```

On peut raccourcir le nom du répertoire :

```
sudo  mv  /opt/arm-buildroot-linux-uclibcgnueabihf_sdk-buildroot/
/opt/cross-br-2020.02
ls  /opt/cross-br-2020.02/
arm-buildroot-linux-uclibcgnueabihf  include  libexec          share
bin                                  lib      relocate-sdk.sh  usr
etc                                  lib64    sbin
```

Puis il faut appeler le script «relocate-sdk» se trouvant dans ce répertoire pour qu'il enregistre le nouvel emplacement. Il faut l'invoquer avec les droits suffisants pour pouvoir écrire dans les fichiers concernés.

```
sudo  /opt/cross-br-2020.02/relocate-sdk.sh
Relocating the buildroot SDK from /home/cpb/br-lab/build-pi4/host to
/opt/cross-br-2020.02 ...
[build-pi4]$
```

Compilation par appel direct du cross-compiler

N'importe quel utilisateur peut alors appeler le cross-compiler.

```
/opt/cross-br-2020.02/bin/arm-linux-g++  --version
arm-linux-g++.br_real (Buildroot 2020.02) 8.3.0
Copyright © 2018 Free Software Foundation, Inc.
Ce logiciel est un logiciel libre; voir les sources pour les conditions de
copie.  Il n'y a
AUCUNE GARANTIE, pas même pour la COMMERCIALISATION ni L'ADÉQUATION À UNE
```

```
TÂCHE PARTICULIÈRE.
```

Supposons par exemple qu'on souhaite compiler un exécutable comme celui-ci :

```
// hello-direct.cpp

#include <iostream>

int main()
{
        std::cout << "Hello World! (direct call to the cross-compiler)";
        std::cout << std::endl;

        return 0;
}
```

On peut appeler le cross-compiler «arm-linux-g++» directement en ajoutant son emplacement dans la variable «PATH» de la session shell.

```
PATH=$PATH:/opt/cross-br-2020.02/bin/
arm-linux-g++  hello-direct.cpp  -o  hello-direct  -Wall
```

Ce fichier peut être transféré avec «scp» sur la cible ou placé dans l'arborescence de external tree. On choisi cette seconde solution. Elle nécessite bien sûr un rebuild de l'image.

```
cp  hello-direct  ../pi4-config/custom-rootfs/usr/bin/
make
```

```
Welcome on board!
R-Pi login: root
Password: (root)
# hello-direct
Hello World! (direct call to the cross-compiler)
```

Naturellement, il est possible d'invoquer le compilateur au travers d'un «Makefile» en remplissant les variables d'environnement adéquante (souvent «CROSS_COMPILE»).

Écriture de recettes pour des projets métiers

La meilleure manière d'intégrer un projet disposant d'une structure de compilation comme «automake», «Cmake» ou même un simple «Makefile» est d'écrire une recette que on ajoutera dans external tree et que Buildroot se chargera de produire avec le reste de l'image.

Il y a de très nombreuses recettes fournies avec Buildroot dans le répertoire «package/» de ses sources, elles peuvent servir d'exemples pour compiler des applications supplémentaires.

Par exemple, on souhaite installer un package qui se compile avec les Autools.

On crée dans external tree un répertoire qui accueillera les différentes recettes que nous souhaitons ajouter. Pour être cohérent avec l'organisation de Buildroot, on le nomme «package».

```
cd  ../pi4-config/
mkdir  package
[pi4-config]$
```

Dans ce répertoire on crée un sous-répertoire dédié au projet concerné. On va prendre le package «hello-autotools» qu'on a écrit en exemple de mon cours en ligne sur Yocto Project.

```
mkdir  package/hello-autotools
```

Dans ce répertoire on doit créer trois fichiers :

- **«Config.in»**: l'entrée de menu pour sélectionner le package lors de la compilation de l'image.
- **«hello-autotools.mk»**: la recette de compilation du package.
- **«hello-autotools.hash»**: un fichier contenant des sommes de contrôles pour les éléments téléchargés.

Voici le contenu du premier fichier, «package/hello-autotools/Config.in» :

```
config BR2_PACKAGE_HELLO_AUTOTOOLS
    bool "hello-tools"
    help
        This is an example of a package built with the autotools.
```

A présent, voici la recette de compilation «package/hello-autotools/hello-autotools.mk». Du fait qu'on emploie le mécanisme des autotools elle est très simple et se borne à indiquer la provenance du package.

```
################################################################################
#
# hello-autotools
#
################################################################################

HELLO_AUTOTOOLS_VERSION = 1.0
HELLO_AUTOTOOLS_SOURCE = hello-autotools-$(HELLO_AUTOTOOLS_VERSION).tar.bz2
HELLO_AUTOTOOLS_SITE = https://www.blaess.fr/christophe/yocto-lab/files
HELLO_AUTOTOOLS_AUTORECONF = YES
HELLO_AUTOTOOLS_LICENSE = GPL-2.0

$(eval $(autotools-package))
```

Dès qu'il y a un téléchargement de package, il est conseillé de vérifier une somme de contrôle. Pour cela, on emploie l'utilitaire «sha256sum» et l'on inscrit la checksum dans le fichier «package/hello-autotools/hello-autotools.hash».

```
sha256  ad06e44345fc85e061932da4bdda964c65f1dc56fbc07ea03ea8b93c68065cfe
hello-autotools-1.0.tar.bz2
```

La recette de compilation pour le package est prête. Mais elle ne sera intégrée par Buildroot que si on lui indique où elle se trouve. Pour cela éditer le fichier «external.mk» à la racine de external tree (fichier précédemment vide) et ajoutons la ligne suivante :

```
include $(sort $(wildcard $(BR2_EXTERNAL_PI4_CONFIG_PATH)/package/*/*.mk))
```

Cette ligne demande à Buildroot d'explorer toutes les recettes se trouvant dans le sous-répertoire «package» de external tree.

Enfin on ajoute dans le fichier «Config.in» de external tree (précédemment vide également) les lignes suivantes qui y définissent un menu incluant le «Config.in» de le package.

```
menu "Custom packages"

source "$BR2_EXTERNAL_PI4_CONFIG_PATH/package/hello-autotools/Config.in"

endmenu
```

Lorsqu'on relance la commande «make menuconfig» de Buildroot, le menu «External options» s'est rempli.

```
cd  ../build-pi4/
make  menuconfig
```

Dans ce nouveau menu, on voie l'option «hello-tools». Bien sûr, on active cette option avant de relancer le build.

Pendant la construction du système, on voit passer brièvement les messages indiquant le téléchargement, l'extraction et la compilation du package.

Vérifier sa présence sur la cible :

```
R-Pi login: root
Password: (root)
# hello-autotools
Hello from R-Pi (built with autotools)
#
```

De la même façon, on a ajouté des recettes pour compiler un package utilisant CMake ou avec un simple «Makefile» classique. On ne les détaille pas ici, ils se trouvent dans l'archive que l'on trouvera

plus loin.

Lancement automatique au démarrage

Buildroot utilise de préférence un mécanisme de lancement des services au démarrage basé sur le processus «init» à la mode System V fourni par «Busybox».

Il est toujours possible, dans le sous-menu «Init system» du menu «System configuration», de choisir un démarrage avec «systemd» mais on l'évite généralement car la maîtrise de la séquence d'initialisation des services est plus simple avec les scripts System V.

Si on veut lancer une application au démarrage, il suffit d'ajouter un script dans le répertoire «etc/init.d» de la cible dont le nom commence par un «S» suivi d'un numéro d'ordre de démarrage. Ce script sera appelé au boot du système avec l'argument «start» et à l'arrêt du système avec l'argument «stop».

Voici ce que contient le répertoire «etc/init.d» initial :

```
# ls  /etc/init.d/
S01syslogd   S02sysctl    S40network   rcK
S02klogd     S20urandom   S50dropbear  rcS
#
```

On ajoute un script de lacement dans l'arborescence de l'external tree :

```
mkdir  ../pi4-config/custom-rootfs/etc/init.d
nano  ../pi4-config/custom-rootfs/etc/init.d/S60hello
```

Le contenu de ce fichier est le suivant :

```
#!/bin/sh

if [ "$1" = "start" ]
then
        /usr/bin/hello-autotools &
fi
```

Ce script est très simple car il ne fait que lancer une commande. Il y a des situations où il est plus complexe car il faut arrêter le service lors du shutdown du système. On a placé un «&» à la fin de la commande, cela n'a aucune utilité ici puisqu'elle se termine immédiatement, mais pourrait servir dans le cas d'une application qui reste en fonction de manière prolongée car le script doit rendre la main une fois le programme démarré.

Il ne faut pas oublier de le rendre exécutable :

```
chmod  +x  ../pi4-config/custom-rootfs/etc/init.d/S60hello
```

Une fois le système recompilé et installé, on vérifie les traces de boot du Raspberry Pi :

```
random: dropbear: uninitialized urandom read (32 bytes read)
OK
Hello from R-Pi (built with autotools)
Welcome on board!
R-Pi login:
```

Le processus a donc bien été lancé automatiquement au boot.

Ajout d'une partition utilisateur

Le système fonctionne de manière plutôt satisfaisante :

```
Welcome on board!
R-Pi login: rpi
Password: (rpi)
$ hello-autotools
Hello from R-Pi (built with autotools)
$ hello-cmake
Hello from R-Pi (built with CMake)
$ hello-makefile
Hello from R-Pi (build with standard Makefile)
```

Un petit problème survient si on souhaite écrire dans un fichier :

```
$ hello-makefile > my-file.txt
-sh: can't create my-file.txt: Read-only file system
```

C'est normal, la partition système est en lecture seulement, on a écrit un script pour y remédier :

```
$ rw
mount: you must be root
$
```

Voilà un véritable problème. on a protégé la partition système en la plaçant en lecture seule. Pour pouvoir y faire quand même des modifications, on a prévu des scripts «rw» et «ro» (en étant conscient que pendant la période où le système est en lecture-écriture, une coupure d'alimentation peut lui être fatale). Mais ces scripts ne peuvent être appelés que par «root», pas par un utilisateur (ou une application métier) s'exécutant sous une identité quelconque.

D'autre part, on sent bien que le fait de mettre temporairement la partition système en lecture-écriture est un risque. Si une coupure d'alimentation survient juste à ce moment le système de fichiers peut être corrompu. Et la probabilité que l'utilisateur débranche et rebranche le système juste après avoir fait une modification de configuration pour voir si elle a été prise en compte est assez élevée.

Si on souhaite pouvoir sauvegarder des données d'application ou des éléments de paramétrage, il va falloir envisager l'ajout d'une partition supplémentaire montée en lecture-écriture à côté de la partition système.

Cette partition peut être formatée en «vfat», système de fichiers simple et robuste qui résiste plutôt bien à des coupures d'alimentations pendant une écriture.

Il faut indiquer la présence de cette partition dans le fichier `/etc/fstab`. on va récupérer le fichier original produit par Buildroot (qui se trouve dans le sous-répertoire «target/etc/» de le dossier de compilation), le copier dans external tree et lui ajouter une ligne pour cette nouvelle partition.

```
cp  target/etc/fstab  ../pi4-config/custom-rootfs/etc/
nano  ../pi4-config/custom-rootfs/etc/fstab
```

Le fichier est modifié ainsi (dernière ligne ajoutée) :

```
# <file system> <mount pt>      <type>  <options>       <dump>  <pass>
/dev/root       /               ext2    ro,noauto       0       1
proc            /proc           proc    defaults        0       0
devpts          /dev/pts        devpts
defaults,gid=5,mode=620,ptmxmode=0666   0       0
tmpfs           /dev/shm        tmpfs   mode=0777       0       0
tmpfs           /tmp            tmpfs   mode=1777       0       0
tmpfs           /run            tmpfs   mode=0755,nosuid,nodev  0       0
sysfs           /sys            sysfs   defaults        0       0
/dev/mmcblk0p3  /home/rpi       vfat    defaults,uid=1000,gid=1000   0   0
```

Cette nouvelle partition sera montée directement sur «/home/rpi», elle appartiendra toute entière à l'utilisateur «rpi» créé précédemment, et sera accessible en lecture et écriture.

Pour créer les partitions et l'image «sdcard.img» finale, Buildroot appelle le script «board/raspberrypi4/post-image.sh» qui est indiqué dans l'option «Custom scripts to run after creating filesystem images» du menu «System configuration». Ce script fait appel à un utilitaire nommé «genimage» en lui passant le fichier de configuration «board/raspberrypi4/genimage-raspberrypi4.cfg».

Pour personnaliser cette étape, on va copier ces deux fichiers dans l'external path en respectant la même structure de sous-dossiers, et les personnaliser ensuite.

```
cd  ../pi4-config/
mkdir  -p  board/raspberrypi4/
cp  ../buildroot-2020.02/board/raspberrypi4/post-image.sh
board/raspberrypi4/
cp  ../buildroot-2020.02/board/raspberrypi4/genimage-raspberrypi4.cfg
board/raspberrypi4/
```

Le script «post-image.sh» fait référence à son propre emplacement dans l'arborescence pour chercher le fichier de description des partitions. Il n'est donc pas nécessaire de le modifier, juste de le placer dans l'external tree.

Editer le fichier de configuration «genimage-raspberrypi4.cfg» et ajouter les lignes en surbrillance :

```
nano  board/raspberrypi4/genimage-raspberrypi4.cfg

image boot.vfat {
  vfat {
    files = {
      "bcm2711-rpi-4-b.dtb",
      "rpi-firmware/cmdline.txt",
      "rpi-firmware/config.txt",
      "rpi-firmware/fixup4.dat",
      "rpi-firmware/start4.elf",
      "rpi-firmware/overlays",
      "zImage"
    }
  }
  size = 32M
}

image pi-home.vfat {
  name = "pi-home.vfat"
  vfat {
    files = { }
  }
  empty = true
  size = 128M
}

image sdcard.img {

  hdimage {
  }

  partition boot {
    partition-type = 0xC
    bootable = "true"
    image = "boot.vfat"
  }

  partition rootfs {
    partition-type = 0x83
    image = "rootfs.ext4"
  }

  partition pi-home.vfat {
    partition-type = 0xC
    image  = "pi-home.vfat"
  }
}
```

Avant de relancer la génération d'une image, il faut indiquer à Buildroot de venir chercher le script dans external tree plutôt que dans le répertoire initial. Ceci se configure dans l'option «Custom scripts to run after creating filesystem image» (avant-dernière ligne) du menu «System configuration».

```
cd  ../build-pi4/
make menuconfig
```

Après installation, on vérifie si l'utilisateur «rpi» a bien accès à son répertoire personnel en lecture et écriture :

```
Welcome on board!
R-Pi login: rpi
Password: (rpi)
$ pwd
/home/rpi
$ echo  hello  >  my-file
$ cat  my-file
hello
$exit
```

Vérifier que la partition système est seulement accessible en lecture, même pour l'utilisateur «root» :

```
Welcome on board!
R-Pi login: root
Password: (root)
# pwd
/root
# echo  hello  >  my-file
-sh: can't create my-file: Read-only file system
#
```

L'image est ainsi robuste et insensible aux coupures d'alimentation en ce qui concerne son arborescence système. La partition utilisateur permet d'enregistrer des données, et en cas d'arrêt brutal seules les données en cours d'écriture seront perdues, le système «vfat» étant assez résilient.

Sauvegarde des paramétrages réalisées

Sauvegarde de la configuration de Busybox

Afin de pouvoir aisément réutiliser la configuration dans un build pour une autre cible par exemple — ou pour la prochaine version de Buildroot — on va sauvegarder cette configuration dans external tree.

La configuration de Busybox est sauvegardée dans un fichier «.config» ressemblant à celui de Buildroot ou du kernel. Ce fichier se trouve dans le sous-répertoire dans lequel Buildroot a fait la compilation de Busybox :

```
ls  -a  build/busybox-1.31.1/
.                       editors                 networking
..                      examples                NOFORK_NOEXEC.lst
applets                 .files-list.txt         NOFORK_NOEXEC.sh
applets_sh              findutils               printutils
.applied_patches_list   include                 procps
arch                    .indent.pro             qemu_multiarch_testing
archival                init                    README
AUTHORS                 INSTALL                 runit
busybox                 .kconfig.d              scripts
busybox.links           .kernelrelease          selinux
busybox_unstripped      klibc-utils             shell
.busybox_unstripped.cmd libbb                   size_single_applets.sh
busybox_unstripped.map  libpwdgrp               .stamp_dotconfig
busybox_unstripped.out  LICENSE                 .stamp_downloaded
.config                 loginutils              .stamp_extracted
Config.in               mailutils               .stamp_kconfig_fixup_done
.config.old             Makefile                .stamp_patched
configs                 Makefile.custom         sysklogd
console-tools           Makefile.flags          testsuite
coreutils               Makefile.help           .tmp_versions
debianutils             make_single_applets.sh  TODO
docs                    miscutils               TODO_unicode
e2fsprogs               modutils                util-linux
```

On copie ce fichier dans external tree en lui donnant un nom un peu plus explicite :

```
cp  build/busybox-1.31.1/.config  ../pi4-config/configs/busybox.config
```

Et on indique à Buildroot où trouver ce fichier de configuration. Pour cela, on se rendons en haut de son menu «Target packages» et remplissons l'option «Busybox configuration file to use?» (la deuxième ligne) avec la chaîne «$(BR2_EXTERNAL_PI4_CONFIG_PATH)/configs/busybox.config».

La même méthode pourrait s'appliquer à la configuration du kernel, à laquelle on peut accéder avec la commande «make linux-menuconfig» et dont la configuration est enregistrée dans le fichier «build/linux-custom/.config».

Sauvegarde de la configuration de Buildroot

Il est possible de sauver la configuration de Buildroot lui-même dans external tree afin de pouvoir la réutiliser ultérieurement sous forme de defconfig et regénérer le système sur une autre machine par exemple.

Tout d'abord on va indiquer à Buildroot l'emplacement de la sauvegarde en utilisant l'option «Location to save buildroot config» du menu «Build options» (deuxième ligne). On donne le chemin vers le répertoire «configs» de external tree. Le nom du fichier se termine par «_defconfig» pour pouvoir l'utiliser comme configuration par défaut, par exemple «custom_pi4_defconfig

Une fois cette opération réalisée, il suffit de demander à Buildroot de sauver les élements essentiels

de sa configuration ainsi :

```
make  savedefconfig
  GEN     /home/cpb/br-lab/build-pi4/Makefile
ls  ../pi4-config/configs/
busybox.config  custom_pi4_defconfig  users.tbl
```

Restauration d'une configuration de Buildroot

Maintenant que la configuration par défaut est enregistrée dans external tree, celui-ci contient :

- la configuration de Buildroot, de Busybox, et la table des utilisateurs
- les élements (scripts, fichiers de configuration, etc) que l'on souhaite ajouter sur la cible
- les recettes pour compiler des packages supplémentaires (code métier).

Il s'agit uniquement de données textuelles (configuration, recettes, etc) que l'on peut versionner avec Git par exemple.

On peut surtout transmettre cet external tree à un autre développeur — c'est ce qu'on fait pour les clients — pour qu'il puisse regénérer le système à l'identique. Il lui faudra juste lancer une ligne un petit peu complexe détaillé plus bas.

On va tester que tout se passe bien. Commençons par effacer toute la compilation faite sur cette machine et télécharger external tree que l'on peut trouver ici :

https://www.blaess.fr/christophe/buildroot-lab/pi4-config.tar.bz2

```
br-lab]$ ls
build-pi4  buildroot-2020.02  buildroot-2020.02.tar.bz2  dl  pi4-config
chmod  -R  +w  buildroot-2020.02
rm  -rf  build-pi4/ pi4-config/ buildroot-2020.02/
ls
buildroot-2020.02.tar.bz2  dl
```

Puis recommencer à zéro le build en utilisant la configuration par défaut personnalisée :

```
tar  xf  buildroot-2020.02.tar.bz2
chmod  -R  -w  buildroot-2020.02
ls
buildroot-2020.02  buildroot-2020.02.tar.bz2  dl
wget  https://www.blaess.fr/christophe/buildroot-lab/pi4-config.tar.bz2
  [...]
tar  xf  pi4-config.tar.bz2
ls
buildroot-2020.02  buildroot-2020.02.tar.bz2  dl  pi4-config  pi4-
config.tar.bz2
cd  buildroot-2020.02/
```

Voici la commande un peu complexe, qui joue trois rôles : demander à Buildroot d'utiliser un external tree, de compiler le code dans un répertoire de travail indépendant de ses sources, et d'utiliser la configuration que on a sauvegardée précédemment :

```
make  O=../build-pi4  BR2_EXTERNAL=../pi4-config/  custom_pi4_defconfig
  [...]
#
# configuration written to /home/cpb/br-lab/build-pi4/.config
#
```

Il ne reste plus qu'à relancer la compilation, et attendre quelques dizaines de minutes :

```
 cd  ../build-pi4/
[buildroot-2020.02]$ cd  ../build-pi4/
make
```