“On ne compile jamais sur la cible embarquée” : Buildroot propose GNU Radio sur Raspberry Pi (et autres)

G. Goavec-Merou, J.-M Friedt, 6 février 2021

Le développement de systèmes embarqués se doit d'optimiser l'utilisation des ressources de stockage, de calcul et énergétiques. En aucun cas compiler sur la plateforme embarquée cible ne respecte ces contraintes. Nous présentons Buildroot pour cross-compiler un système GNU/Linux efficacement, et le bénéfice en terme de performances qu'on en tirera.

Les plateformes qualifiées actuellement d'embarquées – Raspberry Pi, Beagle Bone, Redpitaya, PlutoSDR, STM32MP157 et autres – fournissent plus de puissance de calcul que bien des ordinateurs personnels d'il y a quelques années. Plutôt que promouvoir l'utilisation de ces dispositifs comme des ordinateurs généralistes exécutant des distributions binaires aux performances réduites par la plateforme aux ressources les plus faibles, nous allons promouvoir l'optimisation des performances par la génération d'une chaîne de compilation, noyau, bootloader et applicatifs par Buildroot [1, 2]. Afin de démontrer la puissance de l'approche, nous exécuterons GNU Radio – bibliothèque de traitement du signal compatible avec l'analyse en flux tendu de signaux radiofréquences impliquant une multitude de dépendances – sur diverses plateformes cibles embarquant des cœurs ARM. Ce document accompagne la présentation proposée au FOSDEM 2021 [3].

1 Introduction

GNU Radio fait partie des infrastructures considérées comme très pénibles à compiler à la main compte tenu de la multitude de dépendances, et en particulier vers une cible dont l'architecture de processeur diffère de celle de l'hôte (généralement un ordinateur personnel équipé d'un processeur compatible Intel x86). Diverses solutions sont proposées, du script qui tente d'automatiser la séquence de compilation (obsolète avec le passage à GNU Radio 3.8 en seconde moitié 2019) à PyBOMBS (bonne chance pour cross-compiler). En désespoir de cause, nombre d'utilisateurs s'abaisseront aux deux solutions antagonistes avec les préceptes du développement sur système embarqué (Fig. 1) : passer par une distribution binaire type Raspbian [4] (maintenant Raspberry Pi OS qui semble une aberration puisque simplement un port de Debian pour ARM dédié aux Raspberry et aucunement un “OS” à part entière) lorsqu'elle existe, avec les défauts de performance d'une unique distribution ciblée vers une multitude de plateformes, ou compiler sur la cible, supposant un espace de stockage suffisant pour contenir le compilateur et ses bibliothèques (essayez de compiler GNU Radio sur les 32 MB d'une PlutoSDR).

FIGURE 1 – Raspberry Pi 4 équipée de ses interfaces de communication (Ethernet, USB) alimentée en USB-C par un transformateur mural, recevant un flux de données radiofréquences d'un récepteur DVB-T muni d'un R820T2.

Pourquoi affirmer qu'une distribution binaire généraliste n'est pas appropriée pour un système embarqué? La réponse est évidente avec les anciennes versions de Raspbian qui abordaient toutes les Raspberry Pi (RPi) et donc devaient se contenter du plus petit dénominateur commun, à savoir le cœur 32 bits de pré-RPi 3. La situation change un peu avec une version de test supportant les instructions 64 bits, mais reste toujours sous-optimale, notamment sur l'utilisation des extensions d'instruction SIMD (*Single Instruction Multiple Data*) supportées par NEON sur ARM. Prenons le cas de VOLK (*Vector-Optimized Library of Kernels*) qui est intensivement utilisé par GNU Radio pour optimiser les calculs vectoriels en utilisant les instructions SIMD (pour ceux à qui calcul vectoriel et SIMD ne disent rien : penser CRAY[5]). Lors de l'installation de VOLK, une étape d'optimisation sélectionnant la méthode la plus rapide d'atteindre chaque résultat est tabulée par `volk_config`. Évidemment cet outil ne peut tester que les instructions supportées par le compilateur. Comparons quelques résultats de `volk_config` entre Raspbian (ver-

TABLE 1 – Temps d'exécution de diverses fonctions de VOLK selon les configuration lors de l'exécution de `volk_config`. De gauche à droite une image Buildroot et son libvolk 2.0.0 avec le processeur en économie d'énergie (600 MHz), avec le processeur en mode performance (1500 MHz), une distribution Raspbian avec noyau 64 bits et Ubuntu 20.10 avec libvolk 2.3.0-3.

Buildroot, powersave	Buildroot, performance	Raspbian, ondemand	Ubuntu 20.10, ondemand
volk_64u_popcntpuppet_64uu	**volk_64u_popcntpuppet_64u**	**volk_64u_popcntpuppet_64u**	**volk_64u_popcntpuppet_64u**
generic completed in 7103.62 ms	generic completed in 3089.73 ms	no architectures to test	generic completed in 1256.07 ms
neon completed in 4038.24 ms	neon completed in 1897.77 ms		neon completed in 1329.41 ms
Best aligned arch : neon	Best aligned arch : neon		Best aligned arch : generic
Best unaligned arch : neon	Best unaligned arch : neon		Best unaligned arch : generic
volk_64u_popcntpuppet_64u	**volk_64u_popcntpuppet_64u**	**volk_64u_popcntpuppet_64u**	**volk_64u_popcntpuppet_64u**
generic completed in 7154.26 ms	redgeneric completed in 3157.41 ms	no architectures to test	generic completed in 1271.43 ms
neon completed in 4106.08 ms	neon completed in 2081.84 ms		neon completed in 1594.87 ms
Best aligned arch : neon	Best aligned arch : neon		Best aligned arch : generic
Best unaligned arch : neon	Best unaligned arch : neon		Best unaligned arch : generic
volk_16ic_deinterleave_real_8i	**volk_16ic_deinterleave_real_8i**	**volk_16ic_deinterleave_real_8i**	**volk_16ic_deinterleave_real_8i**
generic completed in 1745.19 ms	generic completed in 697.845 ms	generic completed in 420.678ms	generic completed in 390.322 ms
neon completed in 254.155 ms	neon completed in 105.462 ms	u_orc completed in 391.035ms	neon completed in 121.945 ms
Best aligned arch : neon	Best aligned arch : neon	Best aligned arch : u_orc	Best aligned arch : neon
Best unaligned arch : neon	Best unaligned arch : neon	Best unaligned arch : u_orc	Best unaligned arch : neon
volk_16ic_s32f_deinterleave_32f_x2	**volk_16ic_s32f_deinterleave_32f_x2**	**volk_16ic_s32f_deinterleave_32f_x2**	**volk_16ic_s32f_deinterleave_32f_x2**
generic completed in 2258.27 ms	generic completed in 2185.24 ms	generic completed in 2211.99ms	generic completed in 2125.54 ms
neon completed in 1274.83 ms	neon completed in 728.173 ms	u_orc completed in 4766.13ms	neon completed in 687.01 ms
Best aligned arch : neon	Best aligned arch : neon	Best aligned arch : generic	Best aligned arch : neon
Best unaligned arch : neon	Best unaligned arch : neon	Best unaligned arch : generic	Best unaligned arch : neon
volk_16i_s32f_convert_32f	**volk_16i_s32f_convert_32f**	**volk_16i_s32f_convert_32f**	**volk_16i_s32f_convert_32f**
generic completed in 2181 ms	generic completed in 870.3 ms	generic completed in 749.928ms	generic completed in 530.426 ms
neon completed in 697.446 ms	neon completed in 310.137 ms	a_generic completed in 750.233ms	neon completed in 298.812 ms
a_generic completed in 2181.02 ms	a_generic completed in 870.304 ms		a_generic completed in 531.097 ms
Best aligned arch : neon	Best aligned arch : neon	Best aligned arch : generic	Best aligned arch : neon
Best unaligned arch : neon	Best unaligned arch : neon	Best unaligned arch : generic	Best unaligned arch : neon
volk_16i_convert_8i	**volk_16i_convert_8i**	**volk_16i_convert_8i**	**volk_16i_convert_8i**
generic completed in 1745.56 ms	generic completed in 696.289 ms	generic completed in 457.922ms	generic completed in 462.959 ms
neon completed in 134.038 ms	neon completed in 75.7975 ms	a_generic completed in 458.445ms	neon completed in 66.5504 ms
a_generic completed in 1745.59 ms	a_generic completed in 696.28 ms	Best aligned arch : generic	Best aligned arch : neon
Best aligned arch : neon	Best aligned arch : neon	Best unaligned arch : generic	Best unaligned arch : neon
volk_32f_cos_32f	**volk_32f_cos_32f**	**volk_32f_cos_32f**	**volk_32f_cos_32f**
generic_fast completed in 51036.2 ms	generic_fast completed in 19325.9 ms	generic_fast completed in 22240.9ms	generic_fast completed in 18609.7 ms
generic completed in 13673.1 ms	generic completed in 4678.62 ms	generic completed in 5470.72ms	generic completed in 4150.04 ms
			neon completed in 2637.33 ms
Best aligned arch : generic	Best aligned arch : generic	Best aligned arch : generic	Best aligned arch : neon
Best unaligned arch : generic	Best unaligned arch : generic	Best unaligned arch : generic	Best unaligned arch : neon

sion 64 bits du 27 mai 2020), Buildroot (version git d'aout 2020 avec volk 2.0.0) [6] et Ubuntu 20.10 avec volk 2.3.0-3. On notera que Raspbian et Ubuntu sélectionnent par défaut le mode `ondemand` du processeur qui le laisse traîner à 600 MHz (tel que l'annonce `/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq`) en veille pour l'accélérer à 1500 MHz lors des calculs, tandis que Buildroot laisse le processeur en `powersave` qui le maintient à 600 MHz. Pour rendre la comparaison réaliste, nous avons aussi passé le processeur (version Buildroot) en `performance` qui le maintient à 1500 MHz. Bien sûr Ubuntu s'en sort bien dans ce test de vitesse et pourrait fournir une solution crédible, mais essayez de faire tenir une telle distribution généraliste dans les 32 MB de stockage non-volatil d'une PlutoSDR!.

Le tableau 1, issu de l'analyse de VOLK par `volk_config` sur un système généré par Buildroot avec la configuration par défaut du processeur en `powersave` (cadencé alors à 600 MHz), puis en performance (cadencé alors à 1500 MHz) et avec un système Raspbian 64 bits (processeur en mode ondemand qui passe à 1500 MHz lors de la configuration de VOLK, image nommée `2020-05-27-raspios-buster-arm64.img`) résume les performances des principales fonctions extraites sous forme de leur temps d'exécution. La mise en forme s'est contentée de mettre en gras le nom de la fonction, en vert les appels à l'accélérateur SIMD des instructions NEON, et en rouge la méthode générique exploitant les instructions ARM du processeur. Même en mode d'économie d'énergie ralentissant la vitesse du processeur, l'exploitation des instructions NEON est plus rapide que la même fonction exécutée par la méthode générique de Raspbian poussant le processeur à sa vitesse maximale.

Nous constatons sur ce tableau que

1. Raspbian ne connaît pas le support de NEON et ne peut donc évidemment pas en profiter (2 instructions du haut),
2. que dans tous les cas où la méthode générique se compare à la méthode NEON, le gain en temps de calcul est incontestable (4 instructions du milieu) avec des facteurs allant de 3 à 9 sur la réduction de temps d'exécution,

3. que lorsque la méthode générique est sélectionnée (instruction du bas), Buildroot se comporte à l'identique de Raspbian dans la barre d'incertitude de la mesure.
4. Ubuntu 20.10 dans laquelle nous avons installé `sudo apt-get install libvolk2-dev` gagne haut la main tous les tests de vitesse lors de `volk_profile`! Cependant, cette distribution occupant plus de 3 GB sur une carte SD pour un environnement *Desktop* ne répond aucunement aux exigences d'un système embarqué (e.g. 32 MB de mémoire non-volatile sur PlutoSDR).

Pour toutes ces raisons, nous allons développer ici l'utilisation de Buildroot pour générer les outils de développement sur cible embarquée, et illustrerons sur le cas particulier de GNU Radio qui a été incorporé dans la liste des paquets supportés.

L'obsession sur le support des instructions SIMD traduit simplement le fait que nombre de problèmes s'expriment naturellement sous forme d'**algèbre linéaire** avec des opérations matricielles sur des vecteurs, et que manipuler des vecteurs correspond à effectuer la même opération (*Single Instruction*) sur chaque élément du vecteur (*Multiple Data*). L'implémentation matérielle de ces opérations, en pointant sur une zone mémoire et en effectuant le même calcul sur les éléments adjacents du tableau, réduit considérablement le temps d'exécution comme nous venons de le voir.

Buildroot v.s OpenEmbedded/Yocto
Un des membres actifs de la liste de diffusion des développeurs GNU Radio – Philip Balister – est développeur OpenEmbedded/Yocto (`https://github.com/balister/meta-sdr`). La question se pose donc de la promotion de Buildroot alors que GNU Radio est fourni comme paquet de OpenEmbedded/Yocto. La différence philosophique fondamentale entre OpenEmbedded/Yocto et Buildroot tient en la notion de distribution. OpenEmbedded/Yocto compile tout, encapsule les binaires résultants sous forme de paquet binaire, occupant un espace disque significatif (80 GB) et nécessitant un temps de compilation certain. Le bénéfice à terme est un ensemble de paquets que les utilisateurs ignorants du développement sur système embarqué pourront déployer. Au contraire, Buildroot génère une image binaire ne contenant que les applicatifs nécessaires, pour un volume d'espace disque considérablement réduit (typiquement 8 GB), mais nécessitant de régénérer une nouvelle image à chaque nouvel ajout. Les deux philosophies diffèrent donc dans leur objectif, Buildroot fournissant une solution dédiée à une application bien définie, OpenEmbedded/Yocto générant une distribution applicable à nombre de projets différents. Dans un contexte éducatif de diffusion des outils à une classe d'étudiants, l'utilisation de Buildroot peut être discutable compte tenu de l'hétérogénité des objectifs de chacun, mais pour une cible bien définie le bénéfice de Buildroot est incontestable.

Nous insistons sur la généralité de l'approche proposée : nous ne développerons pas ici la PlutoSDR qui est aussi supportée, mais démontrerons la chaîne de cross-compilation sur RPi (3 et 4 avec leurs cœurs 64 bits et instructions NEON) et le STM32MP157 de ST Microelectronics qui, malgré des ressources de calcul bien plus faibles, est fourni avec un écran qui permet d'illustrer une interface graphique sur système embarqué. Dans le cas des RPi, la démonstration portera sur une acquisition sur plateforme embarquée (e.g. `gr-acars` pour recevoir les messages de avions), voir un transfert vers l'ordinateur personnel (PC) hôte après un pré-traitement sur la plateforme embarquée que nous illustrerons par l'envoi du signal audiofréquence issu de la démodulation de stations dans la bande FM.

2 Concepts et prise en main de Buildroot

Avant de se lancer dans la prise en main de Buildroot, il est utile de comprendre la philosophie sous-jacente. Buildroot est une infrastructure de cross-compilation – donc travailler sur un environnement de calcul puissant, généralement un PC compatible Intel x86, pour générer du code à destination d'une cible embarquée aux ressources contraintes, généralement un processeur ARM mais potentiellement PowerPC ou RISC-V – qui fournit un ensemble cohérent de compilateur pour l'hôte, ainsi que pour la cible un code de démarrage (*bootloader*, généralement `uboot`), un noyau (Linux), des bibliothèques et des exécutables en espace utilisateur (*rootfs*).

Nous utilisons ici la nomenclature qui nous suivra tout au long de cet exposé : hôte pour l'ordinateur puissant sur lequel nous développons, cible pour la carte embarquée aux ressources contraintes. Contrairement aux distributions binaires (Debian ou Raspbian sur RPi, Suse, Redhat ...) ou aux infrastructures de cross-compilation OpenEmbedded et Yocto qui génèrent une bibliothèque de paquets binaires à installer, Buildroot compile uniquement les applications sélectionnées et conclut sa compilation par une unique image contenant tous les outils pour démarrer et exécuter la plateforme embarquée. Tout ajout de fonctionnalité se traduit par la génération complète d'une nouvelle image.

Cette cohérence évite bien des déboires lors de la compilation sur l'hôte d'applications ou module noyau pour la cible, alors qu'on ne compile **jamais** sur la cible de ressources réduites et dont la vocation est de contrôler un système embarqué et non d'être muni d'un compilateur aussi lourd et complexe que `gcc`.

L'arborescence de Buildroot peut apparaître impressionnante au premier abord. Contentons nous de mentionner que

— tout ce qui est produit par Buildroot est placé dans `output`. Ainsi, effacer `output` remet Buildroot dans on état initial (au fichier de configuration `.config` près),

— tout ce qui est produit pour l'hôte (PC généralement) se trouve dans `output/host` – inutile donc de tenter d'exécuter un binaire de ce répertoire sur la cible,
— tout ce qui est produit pour la cible se trouve dans `output/target` – inutile donc de tenter d'exécuter un binaire de ce répertoire sur l'hôte,
— le répertoire `configs` contient les configuration par défaut pour nombre de plateformes, par exemple `raspberrypi4_64_defconfig` pour la RPi 4 en mode 64 bits, `raspberrypi3_64_defconfig` pour la RPi 3 en mode 64 bits, ou `stm32mp157c_dk2_defconfig` pour le STM32MP1. Ces configurations sont appliquées par `make raspberrypi4_64_defconfig` dans le répertoire racine de Buildroot (adapter évidemment le `defconfig` sélectionné à sa cible),
— le répertoire `package` contient la description des paquets reconnus par Buildroot.

Buildroot est configurable intuitivement par une séquence de menus en mode ligne de commande accessible par `make menuconfig`. À l'issue de la configuration, `make` génère dans `output/images` le résultat de sa compilation, à savoir l'image prête à flasher sur carte SD (`sdcard.img`) qui contient le noyau `Image`), le système de fichiers (`rootfs.ext4`), l'image de la partition contenant le bootloader `boot.fat`) et le devicetree décrivant la configuration matérielle (`*.dtb`), tous ces fichiers étant accessibles indépendamment, fort utile pour une exécution dans `qemu` tel que nous le verrons en section 10.

L'ajout de paquets dans Buildroot est la partie la plus intéressante mais étant très bien documentée à `https://buildroot.org/downloads/manual/manual.html#adding-packages`, nous nous contentons de mentionner ici que nous avons rencontré bien des difficultés liées à la gestion (ou l'absence de ...) des dépendances. Nous repoussons à la section 12 cette description détaillée pour mettre en garde ici contre l'ajout incrémental de fonctionnalités qui induisent des dépendances dans des paquets déjà compilés qui ne seront pas mis à jour sans la commande `make paquet-reconfigure` avec `paquet` le nom, trouvé dans `output/build`, de la dépendance à reconfigurer après l'ajout d'une fonctionnalité par `make menuconfig`. Ce point est discuté dans `https://buildroot.org/downloads/manual/manual.html#full-rebuild` : "*if this package is a library that can optionally be used by packages that have already been built, Buildroot will not automatically rebuild those*" ...

Cas particulier : le noyau Linux
Appliquer `make linux-reconfigure` va écraser la configuration sélectionnée par `make linux-menuconfig` et replacer la configuration par défaut sélectionnée par Buildroot. Dans ce cas, on utilisera `make linux-rebuild`. Ceci est vrai pour tous les meta-paquets configurés, notamment Busybox et uBoot.

Lors de sa première compilation, Buildroot télécharge toutes les archives et les stocke dans le répertoire `dl`. Les compilations ultérieures sont ainsi accélérées, voir possibles sans connexion internet. Si plusieurs Buildroot cohabitent sur le même hôte, la variable d'environnement `BR2_DL_DIR` permet d'informer toutes les copies de Buildroot de stocker et chercher leurs archives dans ce répertoire : gain de temps et de place.

Pour installer Buildroot sur RPi 4, tel que décrit à `https://github.com/buildroot/buildroot/tree/master/board/raspberrypi` mais la documentation de cette page n'est pas à jour (?!) :

1. `git clone https://github.com/buildroot/buildroot`
2. `cd buildroot`
3. `make raspberrypi4_64_defconfig`

qui récupère l'arborescence de Buildroot, et la configure pour la RPi 4 en mode 64 bits (pour la RPi 3, sélectionner `raspberrypi3_64_defconfig`). Le cas du STM32MP157 est traité en détail à `https://bootlin.com/blog/building-a-linux-system-for-the-stm32mp1-basic-system/` et sera repris plus loin.

L'approche pédagogique idéale aurait été de progressivement ajouter les fonctionnalités à Buildroot en partant d'un système basique et en ajoutant petit à petit les fonctionnalités. Cependant, il est apparu à l'usage que *Buildroot ne sait pas détecter des incohérences entre dépendances de paquets* si la configuration est modifiée dynamiquement lors de la génération d'une image. Cette déficience est introduite par l'utilisation de `Kconfig` comme gestionnaire de configuration, au même titre que Linux, NuttX ou Zephyr. Ainsi, si GNU Radio a été activé mais qu'un support de matériel (UHD, OsmoSDR) ou de l'interface graphique (Qt5) a été oubliée, il n'est pas simplement possible d'activer ces fonctions et régénérer l'image. Il faut absolument reconfigurer GNU Radio (`make gnuradio-reconfigure`) après avoir activé les nouvelles fonctions pour que Buildroot décide de recompiler GNU Radio avec ces nouvelles dépendances.

Nous serons donc obligés à chaque ajout d'une nouvelle fonctionnalité de `make clean` avant de `make` qui refabriquera tout le contenu de `output`, opération longue et pénible mais nécessaire pour ne pas se heurter à des erreurs de dépendances inextricables. Pour les impatients, on pourra placer le fichier de configuration vérifiant toutes les dépendances de GNU Radio avec support des récepteurs DVB-T RTL-SDR fourni à `https://github.com/oscimp/PlutoSDR/tree/master/configs/raspberrypi4_64_gnuradio_defconfig` (RPi 4) ou `https://github.com/oscimp/PlutoSDR/tree/master/configs/raspberrypi3_64_gnuradio_defconfig` (RPi 3) dans le répertoire `configs` de Buildroot et appliquer la règle idoine pour configurer Buildroot. Dans ce cas, `make` suffira après cette configuration à générer une image fonctionnelle sans passer par toutes les configurations décrites ci-dessous puisque déjà prises en charge par les fichiers proposés.

La suite de cette section décrit les étapes pour retrouver la configuration proposée ci-dessus. Puisque le support Python de GNU Radio nécessitera la bibliothèque `glibc` au lieu de `uClibc` sélectionnée par défaut, nous ajustons la configuration initiale par

4. `make menuconfig`
5. Toolchain → C library (uClibc-ng) → glibc
6. Activer *Enable C++ support*
7. Exit

Une fois la bonne bibliothèque sélectionnée

7. `make`

compile l'ensemble des outils. Cette opération prend environ 40 minutes sur un processeur Xeon à 8 cœurs cadencés à 2,33 GHz avec une connexion internet rapide, et occupe environ 7,4 GB sur le disque dur.

À l'issue de la compilation, nous trouvons dans `output/images` l'image `sdcard.img` à transférer sur la carte uSD en vue d'en exécuter le contenu sur RPi. Ici, "transférer" ne signifie par copier car nous devons écrire octet par octet le contenu du fichier .img sur la carte. Cette opération est prise en charge sous GNU/Linux par `dd`.

La ligne qui va suivre peut **corrompre le disque dur** si le mauvais périphérique est sélectionné. Toujours **vérifier** le nom du périphérique associé à la carte SD (`dmesg & tail`) avant de lancer la commande `dd`.

L'image résultant de la compilation est transférée sur la carte SD par

```
sudo dd if=output/images/sdcard.img of=/dev/sdc
```

où nous avons volontairement choisi le périphérique `/dev/sdc` dans cet exemple car rarement utilisé : en pratique, la carte SD sera souvent nommée `/dev/sdb` (second disque dur compatible avec le pilote SCSI de Linux) ou `/dev/mmcblk0` (cas du lecteur SD interne).

En cas d'utilisation d'un gestionnaire de fichiers ou de bureau, bien vérifier que la carte SD n'est pas prise en charge par ces outils (*eject*) qui risquent d'interférer avec `dd`.

 **Attention** : nous répétons que le contenu de la carte SD, ou de tout support pointé par le dernier argument de cette commande, sera **irrémédiablement** perdu. Vérifier à deux (non, trois) fois le nom du périphérique cible de l'image issue de buildroot.

Une fois l'image flashée sur la carte SD, nous constaterons deux partitions : une première en VFAT (format compatible Microsoft Windows) avec le devicetree, le noyau Linux et le bootloader, et une seconde partition contenant le système GNU/Linux (`rootfs`).

On notera que la configuration de la plateforme ainsi générée, qui est stockée dans `.config`, peut être mémorisée par `make savedefconfig` qui écrase le fichier `defconfig` original dans `configs` (qui peut évidemment toujours être récupéré par `git checkout`). Cette méthode a été utilisée pour générer les `*gnuradio_defconfigs` proposés ci-dessus. Par ailleurs, toutes les commandes que nous avons énoncées pour le moment impactent Buildroot uniquement, sa chaîne de compilation et ses paquets. Pour configurer le noyau Linux et notamment ses pilotes, la commande `make linux-menuconfig` ouvrira le menu de configuration du noyau qui se trouve dans `output/build/linux-*`, utile pour localiser les sources du noyau lors de la compilation de modules personnalisés.

3 GNU Radio avec Python3 et OsmoSDR

La longue description du concept de paquets en début de ce texte visait à introduire le fait que nous ne pouvons pas ajouter progressivement les fonctionnalités à Buildroot. L'ajout de GNU Radio implique de nombreuses étapes, que nous devons toutes effectuer lors de cette étape de configuration si nous ne voulons pas corrompre les dépendances entre paquets.

Nous ferons l'hypothèse dans un premier temps d'un utilisateur désireux de recevoir des signaux radiofréquences avec un récepteur de télévision numérique terrestre (DVB-T) comportant un récepteur R820T2 couplé à un RTL2832U comme convertisseur-analogique numérique communiquant par USB, mais pas d'afficher un résultat sur interface graphique afin d'alléger la séquence de compilation et la taille de l'image résultante (mais capable uniquement d'exécuter des flux de traitement générés par GNU Radio Companion en mode No GUI dans les Options → Generate Options). Il nous faut maintenant activer les paquets nécessaires. Pour ce faire, exécuter dans le répertoire de Buildroot la commande `make menuconfig` et sélectionner `Target packages`. On rappelle que la recherche ("`/`" comme dans `vi`) permet de facilement trouver l'emplacement d'un paquet, par exemple GNU Radio :

1. `make menuconfig`
2. `/eudev`
3. Sélectionner la dernière option indiquée par `BR2_ROOTFS_DEVICE_CREATION_DYNAMIC_EUDEV` et remplacer `/dev management` par `Dynamic using devtmpfs + eudev`

4. /python3
5. Sélectionner l'option (4) indiquée par BR2_PACKAGE_PYTHON3
6. /gnuradio
7. Sélectionner l'option (1) indiquée par BR2_PACKAGE_GNURADIO
8. Sélectionner les options additionnelles de GNU Radio selon les besoins (nous aurons besoin de gr-zeromq support et python support)
9. /osmosdr
10. Sélectionner BR2_PACKAGE_GR_OSMOSDR (avec support Python et support Osmocom RTLSDR)

La première option sert à gérer l'insertion dynamique du récepteur radiofréquence sur le bus USB, la deuxième active le support Python3, la troisième GNU Radio, et la quatrième le support du récepteur DVB-T. Une fois cette configuration achevée, make se charge de fabriquer une image fonctionnelle. Le fichier résultant fera environ 500 MB, nécessitant d'augmenter la taille disponible dans .config par BR2_TARGET_ROOTFS_EXT2_SIZE="520M" ou, plus élégant, de définir dans make menuconfig l'option Filesystem Images → exact size.

À l'issue de la génération de l'image, le bon fonctionnement de GNU Radio sera validé en se connectant (ssh) sur la RPi 4 et, au prompt de Python3, en proposant la commande import gnuradio qui soit se conclure par un retour au prompt sans message d'erreur.

4 Support de l'Ettus Research B210

Le récepteur DVB-T est une solution faible coût pour aborder la radio logicielle mais ne propose qu'une bande passante réduite et ne peut travailler que sur une porteuse allant jusqu'à 1,6 GHz. Les plateformes matérielles de Ettus Research sont bien plus performantes (et plus chères), capables de travailler jusqu'à 6 GHz en couvrant notamment la bande Industrielle, Scientifique et Médicale (ISM) des 2,4 GHz (donc tous les modes de communication numériques tels que Wi-Fi, Bluetooth, Zigbee ...) et surtout d'exploiter pleinement le bus USB3 de la RPi 4 pour augmenter la bande passante d'acquisition limitée par le débit de communication. Les diverses plateformes USRP (*Universal Software Radio Peripheral*) commercialisées par Ettus Research sont controlées par une bibliothèque unique *USRP Hardware Driver* ou libuhd. Cette bibliothèque s'active désormais dans Buildroot, mais nécessite encore de manuellement charger les images (*firmware*) pour configurer le FPGA embarqué. Si la plateforme embarquée est connectée à Internet, le script Python uhd_images_downloader, fourni par uhd-utils, se chargera de télécharger les fichiers binaires correspondant à la version de libuhd et les placer dans /usr/share/uhd/images. Ici encore si libuhd est ajouté *a posteriori*, toute erreur de dépendance sera résolue par make dépendance-reconfigure, par exemple pour pallier aux déficiences de la dépendance boost. Si la même version de libuhd est installée sur l'hôte, nous pourrons exécuter uhd_images_downloader sur le PC et copier les fichiers binaires vers la cible. Sinon, le dernier recours est d'aller chercher manuellement les images binaires à https://files.ettus.com/binaries/cache/ et placer les images dans le répertoire adéquat.

5 Ajouter des *packages* externes : BR2_EXTERNAL

Analog Devices propose la plateforme PlutoSDR pour une centaine d'euros avec des performances proches de celles proposées par Ettus Research (mais une communication sur bus USB2 qui limite la bande passante de transfert). Le support de cette plateforme n'est pas intégré dans la version officielle de Buildroot, Analog Devices ayant fait le choix de dupliquer (*fork*) le dépôt de Buildroot pour y intégrer ses propres modifications à https://github.com/analogdevicesinc/buildroot. Cette solution rend, d'une part, la maintenance sur le long terme complexe car Analog Devices doit régulièrement synchroniser son dépôt avec la version officielle et gérer manuellement les divergences entre les deux dépôts, et d'autre part empêche les utilisateurs de profiter rapidement des évolutions de Buildroot entre deux re-synchronisations. Il semblent donc plus pertinent d'ajouter des fonctionalités externes à une version donnée de Buildroot par le mécanisme décrit ci-dessous.

Jusqu'ici nous n'avons travaillé qu'avec les paquets Buildroot "officiels" maintenus par la communauté des développeurs de Buildroot. Certains paquets ne sont pas encore intégrés sur le site officiel mais peuvent néanmoins compléter l'installation en cours grâce au mécanisme de BR2_EXTERNAL. Un exemple est le support de la PlutoSDR grâce à gr-iio, qui est fourni dans le dépôt BR2_EXTERNAL disponible à https://github.com/oscimp/PlutoSDR et plus spécifiquement dans la branche for_next. Ainsi, après être sorti du répertoire Buildroot pour créer une nouvelle arborescence :

1. git clone https://github.com/oscimp/PlutoSDR
2. cd PlutoSDR
3. git checkout for_next

4. `source sourceme.ggm`

L'approche `BR2_EXTERNAL` est une solution intéressante dès lors que l'utilisateur souhaite stocker des configurations personnalisées, que ce soit dans le cas d'un projet particulier ou pour éviter, comme au début de cet article, d'avoir à copier le fichier de configuration manuellement, ainsi que des `package` en cours de validation. Dans le dernier cas, il ne faut pas voir cette solution comme une fin en soi. Si un `package` est validé, il est important de le proposer en vue d'une intégration dans le dépôt officiel. En effet, si l'utilisateur a éprouvé le besoin de ce paquet, il n'est pas impossible que ce soit également le cas pour d'autres personnes. Cette démarche est en cours pour `libuhd` et `gnss-sdr`.

Maintenant que le dépôt `BR2_EXTERNAL` a été téléchargé, la branche appropriée sélectionnée, et les variables d'environnement définies (dernière commande), retourner dans le répertoire de Buildroot et `make menuconfig`. L'exécution de `make menuconfig` donne maintenant accès à un nouveau menu `External options` qui inclut `gr-iio`, `libad9361-iio` ou `gnss-sdr`.

Afin d'ajouter le transfert de données radiofréquences de la PlutoSDR par GNU Radio, sélectionner dans `External options` l'option `gr-iio`. Afin d'ajouter le support des plateformes USRP de Ettus Research, sélectionner dans `External options` le paquet `uhd`, et pour le cas particulier de la B210, cocher `b200 support` et `python API support`.

Nous pouvons par ailleurs ajuster la configuration avant de `dd` l'image sur la carte SD en ajoutant des fichiers dans `output/target`, par exemple une configuration statique du réseau dans `etc/network/interfaces`, ou copier les *firmware* des USRPs depuis le PC hôte dans le sous répertoire `usr/share/uhd/images` de `output/target` afin que ces fichiers soient ultérieurement disponibles sur la cible embarquée. Une fois le contenu de `output/target` convenablement ajusté, retourner dans le répertoire racine de Buildroot et exécuter `make` pour régénérer le fichier `output/images/sdcard.img`.

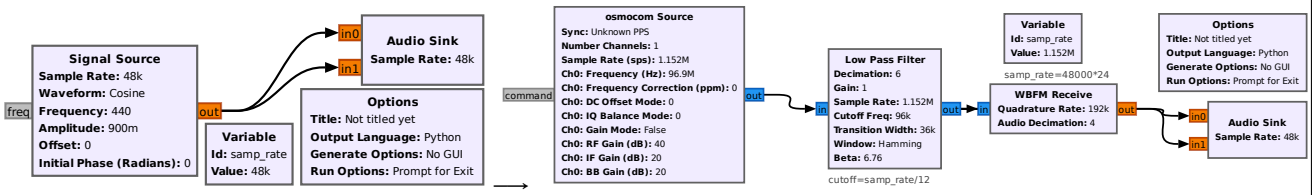
6 GNU Radio sur Raspberry Pi 3/4

À titre d'illustration de la façon que nous abordons d'utiliser GNU Radio sur plateforme embarquée, nous générons en utilisant GNU Radio Companion sur le PC une application en ligne de commande ("No GUI") puisque évidemment aucune interface graphique ne devrait être disponible sur la cible embarquée. Le script Python3 résultant sera exécuté sur la RPi. Le flux audio-fréquence résultant de la démodulation du signal de la bande FM commerciale sera transmis au PC pour être joué sur la carte son.

Son sur la Raspberry Pi 4

Nous sommes mitigés sur la solution la plus simple à mettre on œuvre, entre utiliser la carte son du PC après communication du flux audio issu de la démodulation par la RPi 4 qui suppose une configuration TCP/IP correcte pour transmettre les données (de toute facon nécessaire pour envoyer le fichier Python du PC vers la RPi 4), ou émettre le son sur la sortie jack audio 3,5 mm de la RPi 4 avant de s'attaquer au réseau. Le lecteur désireux de tester la seconde solution peut

1. activer le son sur l'image Buildroot en éditant dans la première partition de la carte SD le fichier `config.txt` pour y ajouter l'option `dtparam=audio=on`. Au prochain boot, le message `bcm2835_audio bcm2835_audio: card created with 8 channels` s'affichera,
2. valider que le son fonctionne en activant le paquet `speaker-test` de `alsa-utils` de `Target packages` → `Audio and video applications` et en exécutant `speaker-test -t sine -f 440` pour jouer un La sur la sortie écouteurs
3. valider que la carte son est bien vue par GNU Radio avec la chaîne de traitement triviale ci-dessous à gauche, en notant que les *deux* entrées audio doivent être connectées,



La génération du script Python et son transfert sur RPi 4 pour exécution sont décrits dans le texte,

4. étendre la chaîne de traitement de gauche avec la source Osmocom et la démodulation FM après avoir sélectionnés par un filtre passe-bas une unique station pour jouer le son sur la sortie audio.

Sur le PC, lancer GNU Radio Companion (fourni par GNU Radio 3.8) et générer la chaîne de traitement de la Fig. 2

Le script Python ainsi généré est transféré à la RPi. Bien prendre soin d'adapter l'adresse IP de la liaison TCP sur 0-MQ à l'adresse de la RPi : le serveur est exécuté sur la plateforme embarquée et en utilisant une liaison de type Publish-Subscribe (s'apparentant à une liaison UDP), tout client se connectant au serveur exécuté sur la cible embarquée peut recevoir le flux de données. L'adresse IP sera incluse idéalement dans le sous-réseau du PC pour simplifier la configuration du routage, tandis que le port peut être toute valeur au dessus de 1024. La seule contrainte sur cette chaîne de

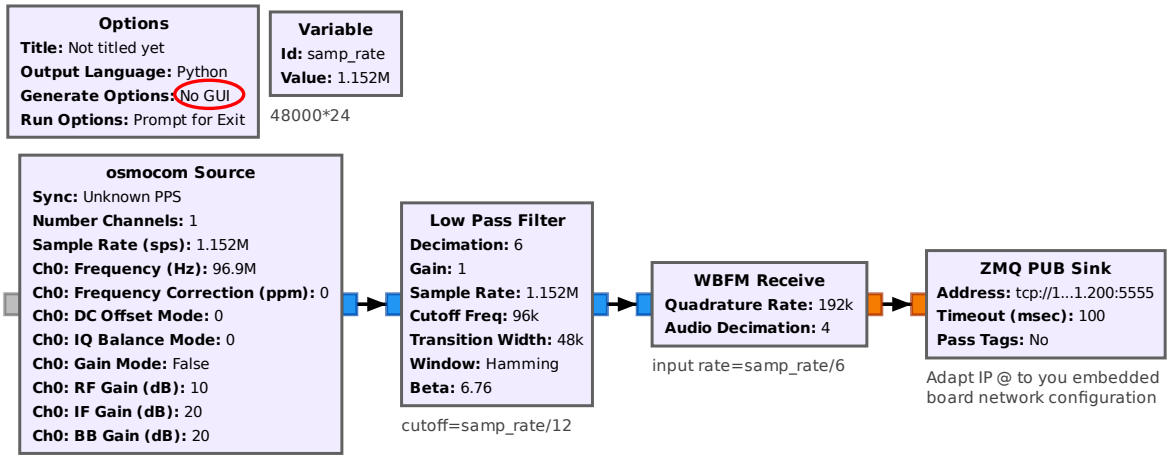


FIGURE 2 – Flux de traitement en ligne de commande pour acquérir et démoduler une station de la bande FM commerciale en vue de transmettre le son vers un PC.

traitement est d'aboutir à la fin à une fréquence d'échantillonnage égale à une valeur compatible avec la cartes son du PC (ici 48 kHz) après une séquence de décimations par des facteurs entiers, objectif atteint en sélectionnant ici une fréquence d'échantillonnage initiale de 48 kS/s. Le premier filtre passe-bas sélectionne une unique station FM tout en gardant assez de bande passante (≥200 kHz) pour la démodulation FM à bande large, et le démodulateur FM ajoute un second étage de décimation.

7 Communication Raspberry Pi vers PC

Après traitement des données radiofréquences brutes (I/Q) acquises sur la bande FM commerciale par la RPi, et avoir pré-traité le signal FM sur la plateforme embarquée, le flux audio-fréquence est transmis vers le PC par 0-MQ. Cette sur-couche à TCP/IP (`https://rfc.zeromq.org/spec/13/`) propose deux modes de communication, connecté tout comme TCP et non-connecté tout comme UDP. Dans le premier cas le transfert de données est garanti et acquitté par l'interlocuteur, dans le second cas le serveur jette les données sur le réseau et si un client les capture, elles seront traitées, sinon elles sont simplement perdues et l'acquisition se poursuit sur le serveur indépendamment de la connexion d'un client. C'est ce second mode qui nous intéresse ici, nommé Publish (serveur RPi) - Subscribe (client PC).

La RPi a vu son interface réseau configurée sur le même sous-réseau que le PC, et nous prendrons soin de faire pointer l'adresse du serveur vers un port associé à l'adresse IP *de la carte Ethernet* de la RPi (et non pas 127.0.0.1) pour écouter toute requête de connexion venant du PC, par exemple par `tcp://192.168.0.42:5555`. Ainsi, le PC connectera son client Subscribe sur cette même adresse pour en récupérer le flux de données (Fig. 3).

Ce résultat peut paraître trivial mais démontre un concept fondamental de la réception de signaux radiofréquences : les étapes successives de traitement ne peuvent que **perdre de l'information**, et comme nous l'a expliqué Shannon [7] le débit d'information est directement lié à la bande passante du signal. Réduire le contenu d'information du signal lors des démodulations successives ne peut que retirer de l'information (la redondance introduite par la modulation FM par exemple) et donc réduire l'encombrement spectral du signal, donc le débit de communication nécessaire à le communiquer. Cela explique que dans un récepteur de radio logicielle nous nous efforçons de pré-traiter le signal acquis au maximum dans le FPGA qui acquiert les flux de données radiofréquences après transposition en bande de base pour atteindre le débit de communication acceptable avec le processeur généraliste. Ce processeur généraliste effectue le maximum de pré-traitements possibles – ici filtrage passe-bas pour ne sélectionner qu'une station FM et démodulation du signal – avant de communiquer vers son interlocuteur, ici le PC qui servira de carte son et connecté par Ethernet au travers de Zero-MQ. Le subtil compromis tiendra toujours à traiter autant que possible en amont afin de réduire la bande passante nécessaire à transmettre à la prochaine unité de traitement, et la complexité de mise en œuvre de ce traitement ou les ressources disponibles. Au plus près du récepteur, nous nous contentons de traitements simples (car complexes à implémenter dans le FPGA tel que le PL du Zynq) mais excessivement rapides, dont le résultat est transféré au processeur local (par exemple le PS du Zynq) pour un prétraitement limité par la puissance de calcul relativement modeste de son double Cortex-A9, avant transfert vers le PC au travers du bus de communication relativement lent (par les standards de la radio logicielle) USB2, PC dont le processeur surpuissant se chargera de finaliser le traitement. Dans cet exemple, nous partons d'un signal échantilloné par le récepteur RTL-SDR à un multiple de 48 kHz, et ici les 24×48 kHz échantillons complexes échantillonés sur 8 bits nécessiteraient $48000 \times 24 \times 2 \simeq 2{,}3$ MB/s de bande passante pour être transmis vers la plateforme de traitement suivante. En démdulant la FM sur la Raspberry Pi, nous réduisons le flux de données à des

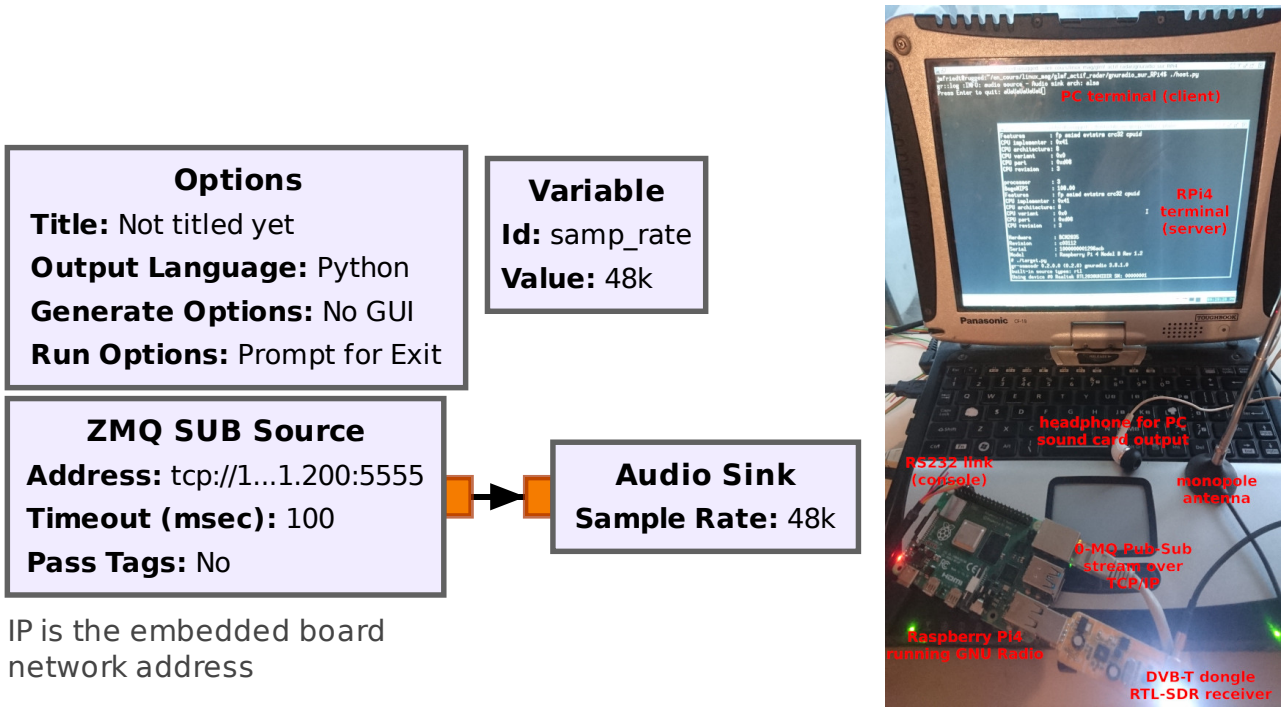


FIGURE 3 – Gauche : chaîne de traitement du client, qui récupère un flux au format *subscribe* de 0-MQ et alimente la carte son du PC. Droite : montage expérimental, avec une RPi 4 connectée par un port série virtuel et par Ethernet au PC portable. La RPi 4 échantillonne le flux de coefficients I/Q du récepteur de télévision numérique terrestre utilisé comme source de radio logicielle en ajustant sa fréquence dans la bande FM, et transmet le flux audio-fréquence après démodulation au PC, permettant d'écouter le programme grâce à un casque connecté à la sortie de la carte son. Cette figure ne permet pas d'illustrer l'excellente qualité audio entendue à la sortie de la carte son, démontrant le bon fonctionnement de ce montage.

nombres flottants réels codés sur 32 bits au débit de 48 kéchantillons/s soit 192 kB/s. Le bénéfice en terme de bande passante est donc évident, et encourage à traiter au maximum de ce que la puissance du sytème embarqué permet en amont de la communication.

8 Interface graphique sur Raspberry Pi 3/4 : Qt5

Il peut parfois être intéressant de proposer une application autonome ne nécessitant pas d'ordinateur associé à la RPi. Étant donné que la RPi 4 propose un port micro-HDMI, une sortie graphique est envisageable. Afin de ne pas s'imposer la lourdeur de X11, il semble judicieux de communiquer directement avec la mémoire vidéo au travers du *framebuffer*.

Le support de Qt5 est extrêmement lourd et sa pertinence est à valider avant de se lancer dans cette compilation. En vue d'activer le support Qt5, nous devons sélectionner

1. `BR2_PACKAGE_QT5`
2. `BR2_PACKAGE_PYTHON3`
3. `BR2_PACKAGE_GNURADIO_PYTHON`
4. `BR2_PACKAGE_GNURADIO_QTGUI`
5. `BR2_PACKAGE_LIBERATION`

— le package qt5 devrait, normalement, être affiché car dès le départ `glibc` a été sélectionnée à la place de `uClibc`.
— compte tenu de la modification des options de `gnuradio` (activation de python et qtgui), il est primordial de relancer sa compilation avec `make gnuradio-reconfigure`
— le package `liberation` doit être manuellement activé car, par défaut, aucune police de caractères n'est présente.

Une fois l'image résultante flashée sur carte SD et exécutée sur RPi 3 ou 4, une application Qt5 GNU Radio sera exécutée (ici sur RPi 4) par `python3 ./rpi.py -platform linuxfb:fb=/dev/fb0` pour envoyer sur un écran la sortie graphique (Fig. 4).

Les outils de capture d'écran proposés par Buildroot ne fonctionnent pas avec le *framebuffer* de la RPi 4. Les diverses illustrations de cet article sont obtenues en copiant directement le contenu de la mémoire du *framebuffer* `cat /dev/fb0 > screen.raw` et en convertissant la séquence de pixels bleu, vert, rouge et transparence en format plus commun compressé par `convert -size 1280x800 -depth 8 bgra:screen.raw outfile.png` en prenant bien entendu soin d'adapter la résolution de l'image à celle de l'écran connecté au port micro-HDMI de la RPi 4.

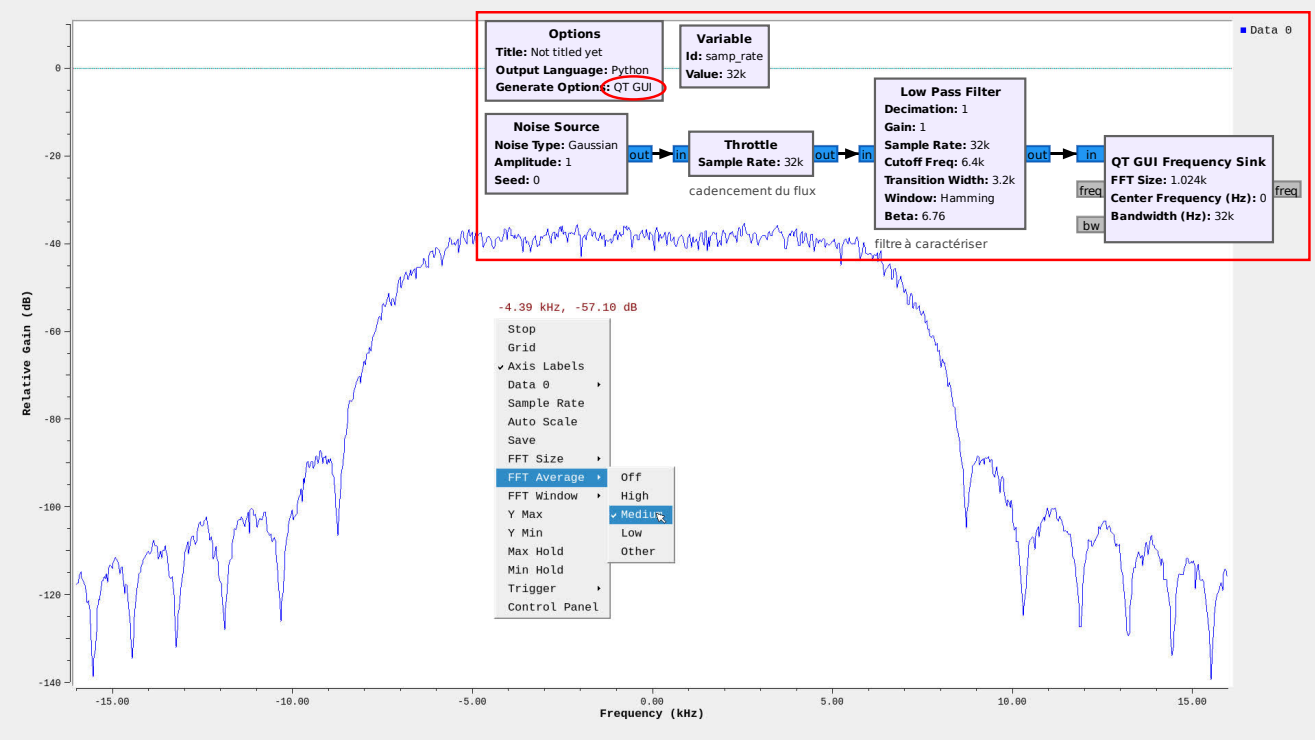


FIGURE 4 – Application Qt5 exécutée sur RPi 4 illustrant la sortie graphique de la chaîne de traitement GNU Radio permettant de caractériser la fonction de transfert spectrale d'un filtre passe-bas.

9 Développements logiciels

Grâce à la chaîne de compilation croisée fournie par Buildroot, la génération d'un exécutable pour notre cible se réduit trivialement à garantir que le compilateur `aarch64-buildroot-linux-gnu-gcc` de `output/host/usr/bin` de Buildroot se trouve dans le `PATH`, et à utiliser ce compilateur comme variable de `CC` dans le `Makefile`. À titre d'exemple avec l'utilisation de PiFM pour émettre un signal radiofréquence sur GPIO4 d'une RPi 4, nous téléchargeons `https://github.com/ChristopheJacquet/PiFmRds`, nous modifions le `Makefile` dans `src` qui fait l'hypothèse d'être exécuté sur la cible (**jamais!**) pour définir `CC = aarch64-buildroot-linux-gnu-gcc` et `RPI_VERSION := 4` puis `make` se charge de générer `pi_fm_rds` que nous copions (`scp`) sur la cible pour l'y exécuter. Il est très amusant d'utiliser le récepteur DVB-T avec GNU Radio démodulant la FM pour recevoir le signal émis depuis la plateforme cible par PiFM : les deux applications cohabitent parfaitement tel que démontré à `http://jmfriedt.free.fr/201229_rpitx.mp4`.

Nombre de logiciels sont cependant désormais proposé avec le générateur de Makefile nommé `cmake` qui facilite l'adaptation aux différentes distributions et systèmes d'exploitation pour compiler un logiciel libre vers une multitude de plateformes. Que ce soit pour modifier `gnss-sdr` ou notre propre décodeur de messages ACARS transmis par les avions vers le sol (Fig. 5), la compréhension de l'intégration des scripts de `cmake` dans Buildroot est indispensable. Afin de générer les Makefiles permettant de cross-compiler un programme vers la cible, nous utiliserons

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=$BR_RPI/output/target/usr \
   -DCMAKE_TOOLCHAIN_FILE=$BR_RPI/output/host/share/buildroot/toolchainfile.cmake ../
```

pour utiliser la toolchain de Buildroot et placer le résultat de la compilation dans l'image qui sera prête à être flashée sur carte microSD. Alternativement, on pourra compiler vers un répertoire cible quelconque sur le PC dont le contenu sera copié (`scp`) vers le répertoire `/usr` du système embarqué si on ne veut pas reflasher une carte microSD avec une nouvelle image.

Nous nous intéressons à modifier les fonctionnalités de `gnss-sdr`. Le code source de ce logiciel, qui s'appuie sur GNU Radio, a été téléchargé et placé dans le répertoire `output/build` lors de sa sélection et installation. Le répertoire

de compilation pour la cible se trouve à `output/build/gnss-sdr-0.0.13/buildroot-build/` tandis qu'un répertoire séparé `output/build/gnss-sdr-0.0.13/build` permet de simultanément tester les modifications aux codes sources sur le PC hôte. Le résultat de la compilation (`make`), soit dans `buildroot-build` (cible ARM) ou `build` (cible x86), se trouve dans `src/main/gnss-sdr`.

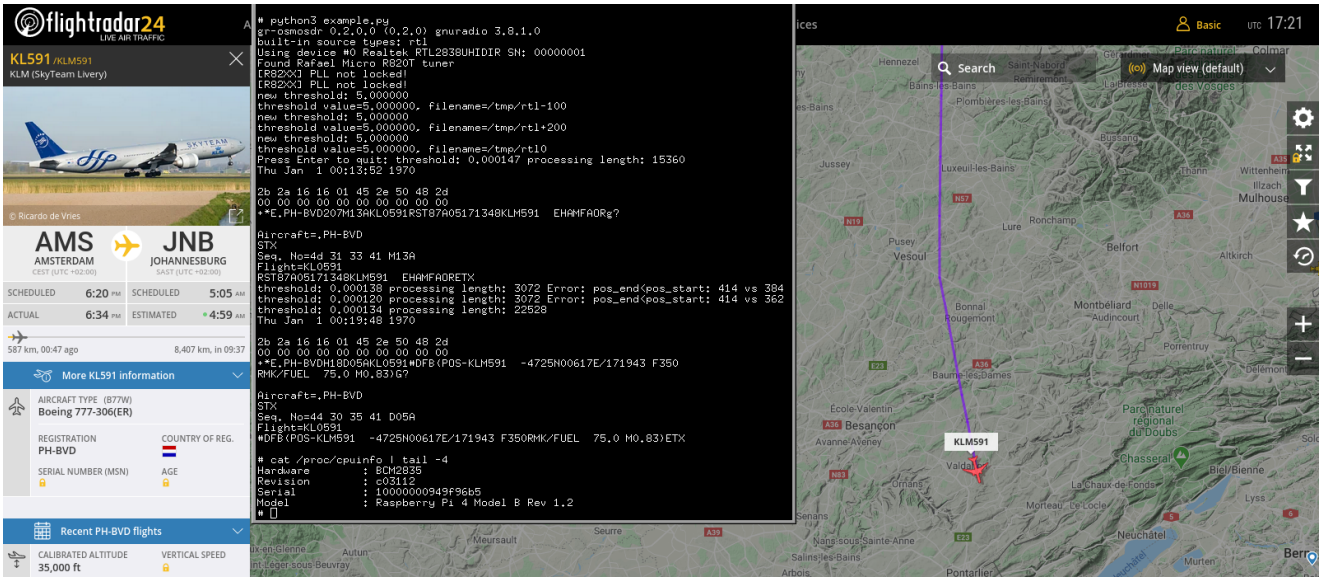


FIGURE 5 – Exécution de `gr-acars`, décodeur de messages ACARS, sur RPi 4, et affichage en arrière plan de la carte des avions à proximité du récepteur au moment de l'acquisition.

10 Exécution dans `qemu`

Supposons que nous n'ayons par reçu le matériel adéquat mais que nous désirions néanmoins nous entraîner avec Buildroot et les binaires générés. La RPi 4 est trop récente pour être intégrée dans QEmu, mais la RPi 3 est annoncée comme fonctionnelle. Nous installons donc QEmu émulant un processeur ARM 64 bits (paquets `qemu-system-arm` sous Debian/GNU Linux qu'on complète tout de suite de `qemu-system-gui` pour le support de l'émulation de l'interface graphique) disponible avec l'exécutable `qemu-system-aarch64`. Ainsi, par

```
qemu-system-aarch64 -kernel Image -dtb ./bcm2710-rpi-3-b.dtb \
 -drive file=./sdcard.img,format=raw,if=sd,id=hd-root \
 -append "rw earlycon=pl011,0x3f201000 console=ttyAMA0 loglevel=8 \
 root=/dev/mmcblk0p2 fsck.repair=yes net.ifnames=0 rootwait memtest=1" \
 -M raspi3 -m 1024 -serial mon:stdio -no-reboot
```

nous exécutons le noyau `Image` qui charge la configuration matérielle `bcm2710-rpi-3-b` par son devicetree, lit le système de fichier accessible sur la seconde partition de `sdcard.img`, le tout sur une machine émulant la RPi 3 (option `-M raspi3`) avec 1 GB de RAM.

Afin de démontrer la capacité à même émuler le *framebuffer*, la Fig. 6 illustre une capture d'écran de la simulation d'un filtre passe-bas par GNU Radio.

Presque incroyable, `qemu` peut même accéder au matériel physiquement connecté à l'hôte. À titre d'illustration, un récepteur DVB-T est connecté au PC qui exécute `qemu` émulant la RPi 3 (Fig. 7). À droite `lsusb` sur l'hôte (PC) indique que le périphérique est reconnu sur le bus `X=1`, adresse `Y=14` avec le Vendor ID de 0x0BDA et le Product ID de 0x2838 associé au Realtek 2832U. En passant ces paramètres comme argument de l'option `-usb -device usbhost=X,hostaddr=Y`, nous constatons (écran de gauche) que `lsusb` dans `qemu` trouve bien le périphérique. L'exécution d'une chaîne de traitement faisant appel au DVB-T par la source Osmocom Source dans GNU Radio (en bas à droite) sur `qemu` se traduit par l'appel au périphérique et l'acquisition de données. L'honnêté nous impose de mentionner que cette capture d'écran est *l'unique* cas, sur une dizaine d'essais, où `qemu` n'a pas crashé en capturant le flux de données, que ce soit en limitant l'acquisition à quelques échantillons (bloc `head` utilisé ici) ou en éliminant l'interface graphique. Le transfert USB au travers de l'émulation du bus USB est donc fonctionnelle mais incapable de supporter un débit de données important (quelques Méchantillons/s), ce qui ne retire rien à la prouesse technique de l'émulation de la RPi 3.

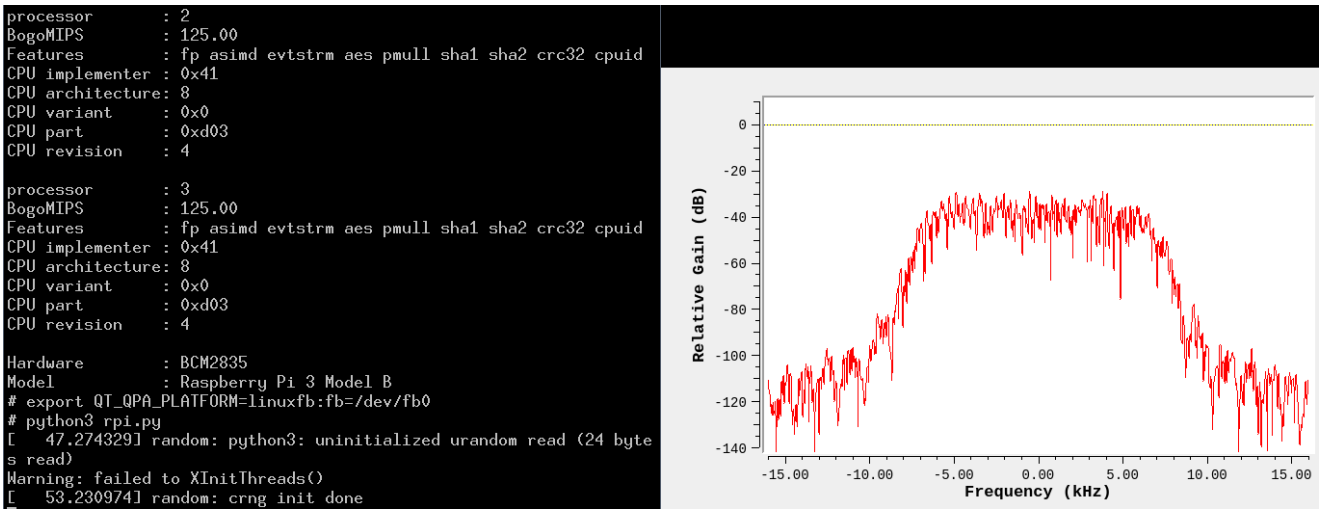


FIGURE 6 – Exécution dans QEmu d'une chaîne de traitement GNU Radio comportant une sortie Qt5.

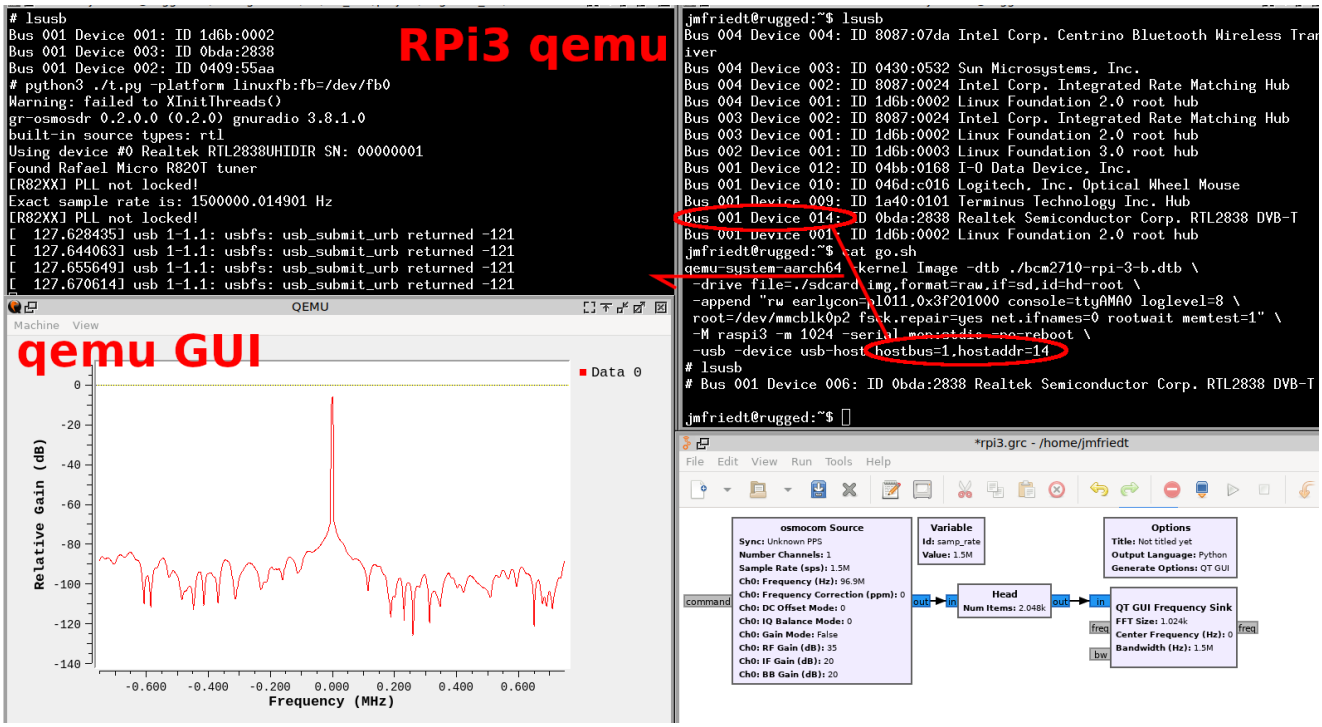


FIGURE 7 – L'émulateur `qemu` accédant au flux de données d'un récepteur DVB-T connecté au port USB de l'hôte.

11 Interface graphique sur STM32MP157

Le STM32MP1 est un processeur proposé par ST Microelectronics couplant un processeur ARM A7 capable d'exécuter GNU/Linux, et des petits Cortex M4 dédiés à des tâches plus simples sans s'handicaper d'un système d'exploitation. Notre intérêt pour cette plateforme hétérogène tient en ses promesses de décharger le processeur principal en déportant certaines fonctions de traitement sur les co-processeurs, sous hypothèse que le temps de communication ne dépasse pas le gain en temps d'éxécution. Sans prétention de puissance de calcul, il s'agit d'une opportunité d'explorer les HMP (*Heterogeneous Multi Processing*) sans s'encombrer des difficultés de développement sur FPGA rencontrés sur Zynq. La carte d'évaluation proposée par ST Microelectronics nommée STM32MP157-DK2 possède un écran et est donc idéale pour démontrer comment les concepts exposés jusqu'ici sont transposables à toute plateforme supportée par Buildroot.

Ici encore la capture d'écran se heurte à quelques soucis de compatibilité du *framebuffer* puisque `fbdump` fonctionne (Fig. 8), mais `fbgrab` ne fonctionne pas.

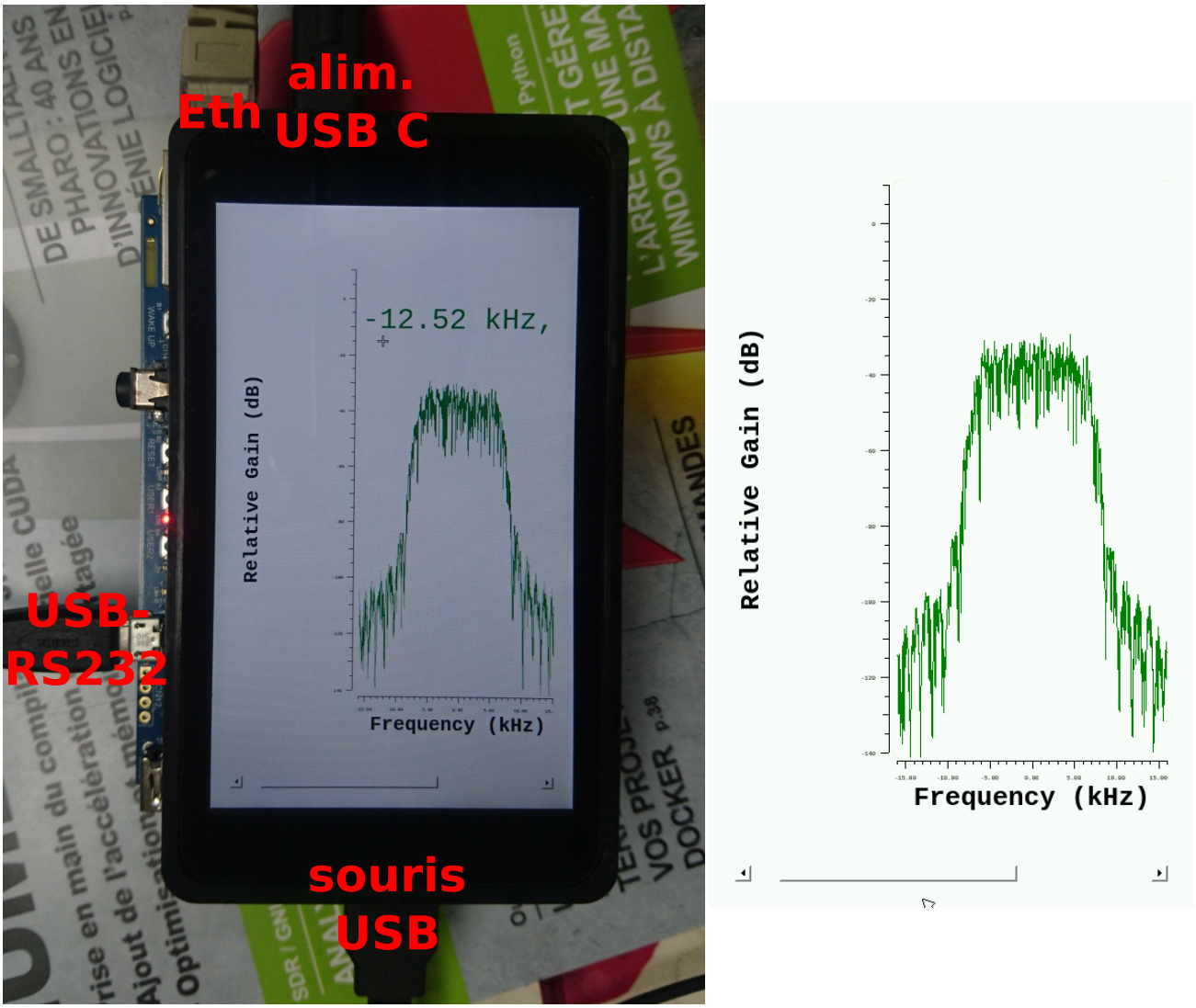


FIGURE 8 – Gauche : photographie du STM32MP157-DK2 exécutant GNU Radio pour simuler la caractérisation d'un filtre passe bande. Droite : capture d'écran de la sortie graphique du STM32MP157-DK2 exécutant cette chaîne de traitement qui nécessite Qt5.

12 Ajout de nouveau paquets

Cette description des paquets est la partie la plus intéressante et celle sur laquelle nous travaillons pour ajouter le support de nouvelles applications, ici GNU Radio et ses bibliothèques associées.

Chaque paquet, qu'il s'agisse de bibliothèques ou d'applications, peut se voir comme une recette pour

— décrire le paquet et ses relations avec les autres paquets (`Config.in`) ;
— décrire quels paquets doivent être construit avant pour générer l'arbre des dépendances et les options de configuration appliquées lors de la compilation selon les choix du développeur (fichier `.mk`).

Les relations entre paquets sont fondamentales lors de la génération de l'arbre des relations. Il existe deux types de relations :

— `depends on` : lorsque `pkg1` dépend de `pkg2`, signifiant que `pkg2` doit être manuellement activé avant de permettre la sélection de `pkg1`. Pour cette raison, un utilisateur qui lance `make menuconfig` ne verra pas apparaître certains paquets tant que les dépendances ne sont pas activées,
— `select` : cette relation est la plus simple à utiliser puisque l'activation de `pkg1` sélectionne automatiquement `pkg3` sans intervention explicite pour activer ce dernier.

Pourquoi utiliser `depends on` au lieu de `select` ? Ce choix dépend de la situation : le support Python de GNU Radio est compatible soit avec Python2, soit avec Python3. Cependant, Buildroot ne peut pas déterminer quelle version activer : c'est donc à l'utilisateur de choisir la version activée au moment de la configuration. Une seconde raison est que si `pkg2` présente beaucoup de dépendances, l'utilisation de `select` impose que l'ensemble de ces dépendances apparaisse dans la configuration de `pkg1`. Ceci ne limite donc pas la présence des `depends on` mais complique la maintenance, car toute

modification au niveau de pkg2 devra être appliquée à pkg1.

Compte tenu de l'absence des paquets qui ne sont pas encore visibles du fait des dépendances invalides, il est important de maîtriser le mécanisme de *recherche* qui proposera toutes les solutions répondant à un mot clé et donnera éventuellement la solution pour faire apparaître ce paquet dans les menus. La recherche s'active, comme sous vi, par "/" suivi du mot clé.

Ce mécanisme de dépendances s'active lors de la première compilation mais pose problème lors de l'évolution des configurations. En effet, en partant du fichier _defconfig, nous obtenons une image fonctionnelle, mais minimaliste ne proposant que quelques outils de busybox. Imaginons que nous voulions maintenant ajouter GNU Radio : make menuconfig et gnuradio nous permet d'atteindre ce résultat. Nous avons cependant oublié d'activer le support Python3 et ne pouvons exécuter les scripts issus de GNU Radio Companion. Il faudra, lors de l'activation de Python3 et de son support par GNU Radio, réactiver la configuration de ce dernier par make gnuradio-reconfigure pour que le support Python soit pris en compte. De la même façon, cette option est incompatible de uClibc – une implémentation minimaliste des appels systèmes de libc – mais nécessite d'utiliser la version plus complète qu'est glibc. Cette modifiction implique de recompiler toute la *toolchain* et tout ce qui a été compilé avec : inutile de tenter de sauver les meubles, autant tout recommencer du début en nettoyant le contenu de output par make clean et recommencer tout le processus de compilation de Buildroot.

Pour conclure cette discussion :

— si une option de la chaîne de compilation est modifiée, il faut recompiler tout Buildroot en repartant d'un répertoire output propre,

— lors de la modification d'une option d'un paquet pkg impactant sur ses dépendances, ce paquet doit être reconfiguré manuellement par make pkg-reconfigure. En fonction de l'impact de cette modification, recompiler tout Buildroot est parfois plus simple.

Nous illustrons cette procédure d'ajout de paquets, tel que décrit en détail à http://buildroot.uclibc.org/downloads/manual/manual.html#adding-packages [8], sur l'exemple cité auparavant de gr-acars qui dépend de GNU Radio. Les fichiers que nous devons modifier pour ajouter ce support sont :

— le fichier package/gr-acars/gr-acars.mk qui contient

```
GR_ACARS_VERSION=fd70b325906e02b758b89786a6470d05bfee0a67
GR_ACARS_SITE = https://git.code.sf.net/p/gr-acars/code
GR_ACARS_SITE_METHOD = git
GR_ACARS_SUPPORTS_IN_SOURCE_BUILD = NO
GR_ACARS_SUBDIR=3.8ng
GR_ACARS_LICENSE = GPL-3.0+
GR_ACARS_DEPENDENCIES = gnuradio \
        $(if $(BR2_PACKAGE_PYTHON3),host-python3,host-python) \
$(eval $(cmake-package))
```

fournissant le hash du dépôt Sourceforge de notre code, la méthode de téléchargement, la license et les dépendances. Subtilité ici, la dépendance s'adapte à la version de Python (2 ou 3) qui a été utilisée lors de la configuration de GNU Radio,

— le fichier package/gr-acars/Config.in qui contient la description du paquet tel qu'il apparaîtra dans l'interface utilisateur, avec

```
config BR2_PACKAGE_GR_ACARS
  bool "gr-acars"
  depends on BR2_PACKAGE_GNURADIO
  select BR2_PACKAGE_GNURADIO_BLOCKS
  #select BR2_PACKAGE_GNURADIO_FFT
  select BR2_PACKAGE_GNURADIO_FILTER
  select BR2_PACKAGE_GR_OSMOSDR
  help
    GNU Radio block for ACARS decoding

    https://sourceforge.net/projects/gr-acars/
```

On notera deux points dans ce fichier :

1. un select est utilisé pour gr-osmosdr car sa liste de dépendances se limite à gnuradio. Comme gr-acars dépend également de gnuradio, il n'est pas nécessaire de faire usage de depends on
2. la ligne select BR2_PACKAGE_GNURADIO_FFT est commentée (présente à titre pédagogique) car cette option est activée par BR2_PACKAGE_GNURADIO_FILTER : il n'est pas donc pas nécessaire de l'activer une seconde fois.

— le fichier package/gr-acars/gr-acars.hash qui contient

```
# Locally calculated:
```

`sha256  d3e8... gr-acars-fd70b325906e02b758b89786a6470d05bfee0a67.tar.gz`
résultat de la commande
`sha256sum dl/gr-acars/gr-acars-fd70b325906e02b758b89786a6470d05bfee0a67.tar.gz` puisque Buildroot clone le dépôt `git` avant d'assembler une archive compressée, les points de suspension étant la longue signature issue de la commande. Cette signature valide donc l'intégrité du téléchargement.

— Finalement, le fichier `package/Config.in` est complété de

```
menu "Miscellaneous"
  source "package/gr-acars/Config.in"
```

pour faire apparaître le paquet nouvellement ajouté.

Suite à ces modifications, la recherche sur le terme ACARS dans `make menuconfig` fait apparaître `gr-acars`, qui sera compilé et inclus dans l'image s'il est sélectionné.

13 Conclusion

Nous avons porté GNU Radio et ses dépendances à Buildroot en vue de l'exécuter sur la multitude de plateformes embarquées supportées par cet environnement de cross-compilation, dont les Rasperry Pi 3/4. Nous avons ainsi les bases pour ensuite y exécuter des infrastructures aussi complexes que `gnss-sdr`. Ce faisant, nous avons découvert les limites de Buildroot (absence de détection de modification de configuration d'un paquet) mais aussi la souplesse avec laquelle de nouveaux paquets peuvent être ajoutés. Une infrastructure aussi complexe que GNU Radio, avec ses multiples dépendances pour supporter une large gamme d'interfaces d'acquisition et de communication allant jusqu'à l'interface graphique en Qt5, est désormais supportée par Buildroot. Le gain en performance d'une compilation dédiée à une cible précise est incontestable face à une distribution binaire généraliste. Les précepts du développement de systèmes embarqués de tirer le meilleur parti des ressources disponibles et de minimiser l'empreinte mémoire et la consommation énergétique pour atteindre les performances requises sont donc atteints.

Références

[1] P. Ficheux, *Introduction à Buildroot*, GNU/Linux Magazine Hors-Série **47** (Avril 2010)

[2] P. Ficheux, *Linux embarqué – Mise en place et développement*, Eyrolles (2017)

[3] G. Goavec-Merou, J.-M Friedt, *Never compile on the target! GNU Radio on embedded systems using Buildroot – feedback on a graduate course on developing an embedded network analyzer*, FOSDEM 2021 (Free Software Radio devroom)

[4] L. Delmas, *Facilitez le déploiement de vos Raspberry Pi en créant vos propres images préconfigurées*, Hackable **35**, pp.66–83 (2020)

[5] C.J. Murray, *The Supermen : The Story of Seymour Cray and the Technical Wizards Behind the Supercomputer*, John Wiley & Sons (1997) et plus récemment D. Patterson & A. Waterman, *SIMD Instructions Considered Harmful* (2017) à `https://www.sigarch.org/simd-instructions-considered-harmful/`

[6] J.-M. Friedt, D. Rabus, G. Goavec-Merou, *Software defined radio based Global Navigation Satellite System real time spoofing detection and cancellation*, Proc. GNU Radio Conference 2020 à `https://pubs.gnuradio.org/index.php/grcon/article/view/73`

[7] C.E. Shannon, *Communication in the presence of noise*, Proc. IRE **37** (1) 10–21 (1949)

[8] Bootlin, *Buildroot training* (2020) à `https://bootlin.com/doc/training/buildroot/buildroot-slides.pdf`