“Never compile on the target” : Buildroot supports GNU Radio for the Raspberry Pi (and others)

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

Embedded system development aims at optimizing resource usage including storage, processing and energy. Compiling on the target platform meets none of these constraints. We introduce Buildroot for efficiently cross-compiling a GNU/Linux system, emphasizing on the performance benefits of the dedicated toolchain.

Platforms currently qualified as embedded – Raspberry Pi, Beagle Bone, Redpitaya, PlutoSDR, STM32MP157 and others – provide much more computational power than most personal computers from a few years ago. Rather than promoting the use of these devices as general purpose computers executing binary distributions whose performance are limited by the least powerful supported platform, we promote the optimization of performances by generating a dedicated cross-compilation toolchain, kerrnel, bootloader and userspace applications using the Buildroot [1, 2] framework. In order to demonstrate how powerful this approach is, we will execute GNU Radio – digital signal processing libraries compatible with real time processing of radiofrequency signals requiring compliance with a multitude of dependencies – on various embedded ARM-core platforms. This document supports the presentation shared during the 2021 FOSDEM edition [3].

1 Introduction

GNU Radio is part of the signal processing frameworks considered as very challenging to manually compile due to the many dependencies, and most significantly when targeting a board whose processor is a different architecture than the host (usually a personal computer fitted with an x86-compatible processor). Several solutions are available, ranging from a script attempting to automate the compilation process (which became obsolete when GNU Radio shifted to version 3.8 during mid-2019) to PyBOMBS (good luck with cross-compiling). Out of frustration, many users will end up with one or both of the two solutions conflicting with the principles of embedded system development (Fig. 1): using a binary distribution such as Raspbian [4] (now called Raspberry Pi OS which seems strange since it appears as a port of Debian to ARM dedicated to Raspberry boards and not a dedicated “OS”) when it exists, with the performance losses associated with a unique distribution targeted to a whole range of platforms, or compiling on the target, assuming a sufficient storage space to hold the compiler and the development libraries (try compiling GNU Radio on the 32 MB storage space of the PlutoSDR).

Figure 1: Raspberry Pi 4 fitted with communication interfaces (Ethernet, USB) powered through its USB-C connector from a wall power adapter, receiving the radiofrequency datastream from a R820T2 based DVB-T receiver.

Why claim that a general purpose binary distribution is unsuitable for an embedded system? The answer is obvious with older versions of Raspbian which were to run on all versions of the Raspberry Pi (RPi) single board computers and hence had to adapt to the least common denominator, namely the 32-bit cores of the models prior to RPi 3. The situation is somewhat improving with the test version supporting 64-bit instructions, but the solution remains far from optimal, most significantly with the lacking support for the SIMD (*Single Instruction Multiple Data*) instruction set provided by the NEON extension on ARM processors. Let us consider VOLK (*Vector-Optimized Library of Kernels*) which is intensively used by GNU Radio to optimize computations on vectors by using SIMD instructions (for those to whom vector calculation and SIMD do not mean much: think CRAY[5]). When installing VOLK, an optimization step aimed at selecting the fastest means of completing various processing sequences is saved with `volk_config`. Obviously this tool can only test instructions supported by the compiler. We compare a few results

Table 1: Execution duration of various VOLK functions depending on the configuration when executing `volk_config`. From left to right, a Buildroot image and its libvolk2.0.0 with the processor saving energy (600 MHz), with the processor in performance mode running at 1500 MHz, a 64-bit Raspbian distribution and Ubuntu 20.10 with 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

provided by `volk_config` runnong on Raspbian (64 bit version downloaded May 27, 2020), Buildroot (git version from Aug. 2020 with volk 2.0.0) [6] and Ubuntu 20.10 with volk 2.3.0-3. Notice that both Raspbian and Ubuntu default selection for the processor mode is `ondemand`, selecting the slower 600 MHz clock frequency (as stated in `/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq`) in sleep mode to speed up to 1500 MHz when loaded during calculation, whereas Buildroot keeps the processor in `powersave` mode which keeps the clock at 600 MHz. For a realistic comparison, we set the processor during the Buildroot benchmarking to `performance` mode keeping the clock frequency at 1500 MHz. Whereas Ubuntu is doing a fine job in the speed test and might provide a relevant solution, try fitting a general purpose distribution in the 32 MB non-volatile storage of a PlutoSDR !

The table 1, summarizing some of the analysis of the results of VOLK obtained with `volk_config` on a system generated with Buildroot with the default `powersave` energy saving configuration (processor clocked at 600 MHz), then `performance` (clocked at 1500 MHz) and with a 64 bit Raspbian system (processor clocked with `ondemand` rules reaching 1500 MHz when configuring VOLK, binary image named `2020-05-27-raspios-buster-arm64.img`) displays the execution durations of some of the main functions. The chart only emphasizes in bold letters the name of the function, in green when the SIMD instructions of the NEON processor are used, and in red when the generic ARM instructions of the processor are used. Even in a mode saving energy when the processor speed is reduced, using NEON instructions leads to shorter execution durations than the same function executed with generic instructions on a Raspbian system running the processor at its fastest speed.

We observe on this chart that

1. Raspbian is not aware of NEON instructions and hence cannot benefit obviously (2 top lines),
2. in all cases when the generic method compares with the NEON implementation, the execution duration gain is unquestionable (4 middle lines) with improvements ranging from 3 to 9

3. when the generic method is selected (bottom lines), Buildroot performance is comparable to that of Raspbian within measurement uncertainty

4. Ubuntu 20.10 in which we installed `sudo apt-get install libvolk2-dev` performs by far best in all speed test during `volk_profile` ! Nevertheless, this distribution is 3 GB-large on an SD card providing a *Desktop* environment does not meet the requirements of embedded system development (e.g. 32 MB of non-volatile memory of the PlutoSDR).

For all these reasons, we will address here the use of Buildroot to generate the tools needed to develop on the embedded target, and illustrate how powerful the framework is with the particular case of GNU Radio which was added as part of the supported packages.

The obsession with SIMD instruction support simply results from the fact that most problems can be expressed, possibly under some assumption, as **linear algebra** problems handling matrix and vector operations, and that handling vector means applying the same operation (*Single Instruction*) on each element of the vector (*Multiple Data*). Hardware implementation of these operations, by sharing a pointer to the memory area holding the data and applying the same computation on all adjacent elements of the array, considerably reduces execution time as we have just seen.

Buildroot v.s OpenEmbedded/Yocto
One of the very active members of the GNU Radio developer mailing list – Philip Balister – is also an OpenEmbedded/Yocto developer (`https://github.com/balister/meta-sdr`). The question is thus open as to why promote Buildroot when GNU Radio is supported by the OpenEmbedded/Yocto framework. The fundamental philosophical difference between OpenEmbedded/Yocto and Buildroot lies in the concept of distribution. OpenEmbedded/Yocto compiles everything, wraps the resulting executable as binary packages, holding a significant disk space (80 GB) and requiring a significant compilation time. The benefit at the end is a set of packages that users clueless of embedded system development methods can deploy. On the opposite, Buildroot generates a single binary image including only the needed applications, requiring for the whole process a much lower storage space of about 8 GB. However, a new image must be generated again every time the list of software is updated. The two philosophies hence diverge in their objectives, with Buildroot providing a solution tailored to a well defined application, whereas OpenEmbedded/Yocto generate a general purpose distribution applicable to a whole range of projects. In the context of teaching embedded system development to student class, using the Buildroot framework might be questionable considering the heterogeneous requirements of each student, but for a well defined target the benefit of Buildroot is unquestionable.

Let us emphasize on how general the proposed approach is: we will not develop here for the PlutoSDR which is also supported with this framework, but will demonstrate the cross-compilation toolchain method on the RPi (3 and 4 with their 64 bit core supporting the NEON instruction set) and the ST Microelectronics STM32MP157 which, despite much lower computational resources, is provided with a screen allowing to demonstrate some graphical user interface display on an embedded system. In the case of RPi, the demonstration will focus on acquiring signals on the embedded board (e.g. `gr-acars` to receiver messages transmitted by planes) either for onboard processing or transfer to the host PC following some pre-processing on the embedded platform as illustrated by sending the audiofrequency signal resulting from commercial broadcast FM demodulation.

2 Concepts and getting started with Buildroot

Before getting started with Buildroot, understanding the underlying philosophy might be useful. Buildroot is a cross-compilation framework – so that working on a powerful host, most usually and Intel x86 based PC, to generate binary files for an embedded target with limited resource, most usually an ARM processor but possibly also PowerPC or RISC-V – that provides a consistent set of compilers for the host, as well as for the target a bootloader, usually *uboot*, a kernel (Linux), libraries and userspace libraries (*rootfs*).

We use here the common naming convention that will be used throughout this presentation: the host is the powerful computer we work with, the target is the embedded board with limited resources. As opposed to binary distributions (Debian or Raspbian on RPi, Suse, Redhat ...) or to the cross-compilation frameworks OpenEmbedded and Yocto which generate a set of binary packages to be installed, Buildroot only compiles the selected applications and concludes its compilation with a unique image including all the tools needed to start and execute the embedded platform. Adding new functionalities is only possible by generating a complete new image.

This consistency avoids difficulties often met when compiling applications or kernel modules on the host for a target while **never** compiling on the target with limited resources whos aim is to control an embedded system and not to be running a compiler as greedy in computational resources and complex as `gcc`.

The Buildroot tree structure appears impressive at first sight. Let us just briefly describe the architecture by mentioning that directories:

- all files produced by Buildroot are located in `output`. Thus, deleting `output` brings Buildroot back to its initial state (except for the `.config` configuration file),

- all files to be executed on the host (PC usually) are located in `output/host` – no need hence to try and execute a file copied from this directory on the target,
- all files to be executed on the target are stored in `output/target` – no need hence to try and execute a file from this directory on the host computer,
- the `configs` directory holds the default configuration for many platforms, for example `raspberrypi4_64_defconfig` for the RPi 4 supporting its 64 bit instruction set, `raspberrypi3_64_defconfig` for the RPi 3 supporting its 64 bit instruction set, or `stm32mp157c_dk2_defconfig` for the STM32MP1. These configurations are used by running `make raspberrypi4_64_defconfig` in the root directory of Buildroot (obviously adapt the selected `defconfig` for the appropriate target),
- the `package` directory holds the description of all applications supported by Buildroot.

Buildroot is intuitively configurable using a sequence of command line menus launched with `make menuconfig`. Having selected the configuration options, `make` generates in `output/images` the result of the compilation, namely the image (`sdcard.img`) ready to be transfered to the SD card, the kernel `Image`), the filesystem (`rootfs.ext4`), the image of the partition holding the bootloader `boot.fat`) and the devicetree describing the hardware configuration (`*.dtb`), with all these files accessible independently, most useful when executing in `qemu` as will be discussed in section 10.

Adding packages to Buildroot is the most interesting part but being already very well documented at `https://buildroot.org/downloads/manual/manual.html#adding-packages`, let us only mention here that we have met many problems related to the handling (or lack of ...) dependencies. We postpone to section 12 a detailed description of these issues to only warn here about the risks of incrementally adding functionalities which induce dependencies in packages already compiled which will not be updated unless the command `make mypackage-reconfigure` with `mypackage` the name, found in `output/build`, of the dependance to be reconfigured following the addition of the functionality with `make menuconfig`. This issue is discussed in `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*" ...

The case of the Linux kernel
Running `make linux-reconfigure` will overwrite the configuration selected with `make linux-menuconfig` and replace with the default Buildroot configuration. In this case, `make linux-rebuild` should be used. This is true for all meta-packages that can be configured, including Busybox and uBoot.

When first compiled, Buildroot will download all archives and store them in the `dl` directory. Subsequent compilation will hence be accelerated by avoiding the data transfer step, or even allow for compiling Buildroot when Internet connection is lacking. If multiple Buildroot are running on the same host, the shell variable `BR2_DL_DIR` tells all the copies of Buildroot to store and fetch their archives in this directory: space and speed will benefit from sharing this resource.

Installing Buildroot on a RPi 4, as described at `https://github.com/buildroot/buildroot/tree/master/board/raspberrypi` but this web page documentation is not up to date (?!), requires:

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

which fetches the Buildroot archive, configures for a 64-bit RPi 4 (for the RPi 3, select `raspberrypi3_64_defconfig`). The case of the STM32MP157 is addressed in detail at à `https://bootlin.com/blog/building-a-linux-system-for-the-stm32mp1-basic-system/` and will be discussed later.

The ideal teaching approach would have been to progressively add functionalities to Buildroot by starting from a basic system and adding one by one features. However, when trying this approach we realised that *Buildroot is unable to detect inconsistencies between package dependencies* is the configuration is dynamically modified during the generation of the image. This limitation is introduced by the use of `Kconfig` for handling configurations, as do Linux, NuttX or Zephyr. Thus, if GNU Radio has been activated but hardware support (UHD, OsmoSDR) or graphical interfaces (Qt5) have been forgotten, it is just not possible to activate these functionalities and re-generate a new image. GNU Radio must be reconfigured (`make gnuradio-reconfigure`) after activating the new functionalities so that Buildroot decides to recompile GNU Radio with the new dependances.

We must hence when each new functionality is added remember to `make clean` prior to `make` which will recreate the content of `output`, lengthy and painful operation but mandatory to avoid reaching hardly understandable and inextricable errors due to missing dependencies. For the impatient, a configuration file activating all the GNU Radio dependencies for activating hardware support of RTL-SDR DVB-T receivers is provided at `https://github.com/oscimp/PlutoSDR/tree/master/configs/raspberrypi4_64_gnuradio_defconfig` (RPi 4) or `https://github.com/oscimp/`

`PlutoSDR/tree/master/configs/raspberrypi3_64_gnuradio_defconfig` (RPi 3) to be stored in the `configs` directory of Buildroot and running the appropriate rule to configure Buildroot. In this case, `make` should be enough following this configuration step to generate a functional image without manually activating all the options described below since already included in the configuration file.

The remainder of this section describes the manual steps to reach the configuration described above. Since Python support for GNU Radio is mandatory and requires the `glibc` library instead of the default `uClibc`, we modify the initial configuration with

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

Once the appropriate library implementation has been selected

7. `make`

compiles all the tools. This operations will last about 40 minutes on a 8-core Xeon processor clocked at 2.33 GHz with a fast Internet connexion, and will use about 7.4 GB storage space.

At the end of the compilation, we find in `output/images` the file `sdcard.img` to be transfered to the uSD card in order to execute its content on the RPi. Here, "transfering" does not mean copying since we must write each and every byte of the .img file on the mass storage medium. This operation is handled on GNU/Linux with the tool `dd`.

The following line might **definitely damage the hard disk content** if the wrong peripheral is selected. Alway **check** the name of the peripheral associated with the SD card (`dmesg & tail`) before running the `dd` command.

The image resulting from the compilation is transfered to the SD card with

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

where we have on purpose selected the peripheral name `/dev/sdc` in this example since it is hardly ever used: most commonly, the SD card will be called `/dev/sdb` (second SCSI compatible hard disk on Linux) or `/dev/mmcblk0` (if an internad SD card reader is used.

If a file manager or a desktop window manager is used, make sure that the SD card has not been automagically handled and mounted by such tools (*eject*) which might overwise interfere with `dd` and corrupt the transfer process.

 **Warning**: we repeat that the content of the SD card, or any other storage support pointed to by the last argument of this command, will be **definitely** lost. Check twice (no, three times) the name of the target peripheral on which the Buildroot image will be transfered.

Once the image has been transfered to the SD card, we can find there two partitions: the first one formated as VFAT (compatible with Microsoft Windows) holding the devicetree, the Linux kernel and the bootloader, while the second one holds the GNU/Linux (`rootfs`) filesystem.

Notice that the platform configuration thus generated is saved in the `.config` file which can be saved by using `make savedefconfig` which overwrites the original `defconfig` in `configs` (which can obviously be recovered with `git checkout`). This method was used to generate the `*gnuradio_defconfigs` mentioned above. Furthermore, all the commands we have discussed so far only impact Buildroot, its compilation toolchain and its packages. In order to configure the Linux kernel and its device drivers, the command `make linux-menuconfig` opens the kernel configuration menu, found in `output/build/linux-*`, useful to locate the kernel sources when compiling one's custom kernel modules.

3 GNU Radio with Python3 and OsmoSDR

This lengthy description of the concept of packages at the beginning of this text aimed at introducing the fact that will shall not be able to progressively add functionalities to Buildroot. Adding GNU Radio support implies many steps, which must all be completed durint this configuration step if we wish to avoid package dependency corruption.

We will at first assume that the user wants to collect radiofrequency samples using a R820T2 frontend and RTL2832 analog to digital converter and USB transceiver digital video broadcast (DVB-T) receiver, but not display the graphical result in order to make the compilation duration and resulting image size lighter by only executing command line processing chains generated with GNU Radio Companion set in No GUI in its Options → Generate Options. We must now activate the necessary packages. To do so, we execute in the root directory of Buildroot the command `make menuconfig` and select `Target packages`. Remember that searching ("/" as would be done with `vi`) allows to easily locate the place where a package is stored, for example for GNU Radio:

1. make menuconfig
2. /eudev
3. Select the last option indicated with BR2_ROOTFS_DEVICE_CREATION_DYNAMIC_EUDEV and replace /dev management with Dynamic using devtmpfs + eudev
4. /python3
5. Select option (4) indicated with BR2_PACKAGE_PYTHON3
6. /gnuradio
7. Select option (1) indicated with BR2_PACKAGE_GNURADIO
8. Select additional GNU Radio options depending on the needs (we shall use gr-zeromq support and python support)
9. /osmosdr
10. Select BR2_PACKAGE_GR_OSMOSDR (with Python support and Osmocom RTLSDR support)

The first option is needed to dynamically generate the entry point to the radiofrequency receiver on the USB bus, the second activates Python3 support, the third GNU Radio, and the fourth the DVB-T receiver support. Once this configuration has been completed, make will generate a functional image. The resulting file will be about 500 MB large, requiring the increase of the available size in .config with BR2_TARGET_ROOTFS_EXT2_SIZE="520M" or, better, to define with make menuconfig the option Filesystem Images → exact size.

At the end of the image generation, proper operation of GNU Radio is validated by connecting to the RPi 4 (ssh) and, after getting the Python3 prompt, by running import gnuradio which must return the prompt with no error message.

4 Ettus Research B210 support

The DVB-T receiver is a low cost solution for getting started with software defined radio processing but only provides a restricted sampling bandwidth and working with carrier frequencies up to 1.6 GHz. Ettus Research hardware exhibits must better performance (but are far more expensive), able to operate up to 6 GHz including the 2.4 GHz Industrial, Scientific and Medical (ISM) band (and hence most digital communication modes such as Wi-Fi, Bluetooth, Zigbee ...) and most significantly to fully use the USB3 bandwidth as provided by the RPi 4 to increase the acquisition sampling rate limited by the communication speed. The various USRP (*Universal Software Radio Peripheral*) plateforms sold by Ettus Research are controled by a unique library called *USRP Hardware Driver* or libuhd. This library is now supported in Buildroot, but still requires manually downloading the (*firmware*) images to configure the embedded FPGA. If the embedded platform is connected to the Internet, then the uhd_images_downloader Python script, provided by uhd-utils, will download the binary files matching the libuhd version and store them in /usr/share/uhd/images. Here again if libuhd is added after an image had been generated, any dependency error on mypackage will be solved by running make mypackage-reconfigure, for example to solve the missing dependencies with boost. If the same version of libuhd is installed on the host computer, we can execute uhd_images_downloader on the PC connected to the Internet and copy the binary files to the target. Otherwise a last solution is to manually fetch the files at https://files.ettus.com/binaries/cache/ and store the images in the appropriate directory.

5 Adding external *packages*: BR2_EXTERNAL

Analog Devices is selling the PlutoSDR platform for about one hundred euros with performances close to those found in the hardware developed by Ettus Research (but communicating over a USB2 bus limiting the data transfer bandwidth). Support for this platform is not integrated in the official Buildroot version, Analog Devices having selected to fork the Buildroot repository to integrate its own updates at https://github.com/analogdevicesinc/buildroot. This solution makes long term maintenance challenging since Analog Devices must regularly synchronize its repository with the official version and manually handle divergences between the two archives, and on the other hand prevents users from quickly benefiting from Buildroot upgrades between two re-synchronizations. Adding external functionalities to a given Buildroot version appears as a better solution as will be described below.

So far we have only workded with "official" Buildroot packages maintained by the Buildroot developer community. Some of these pacakages are not yet integrated in the official repository but can still be included in the on-going installation thanks to the BR2_EXTERNAL mechanism. One such example is the PlutoSDR support thanks to gr-iio, which is provided in the BR2_EXTERNAL repository at https://github.com/oscimp/PlutoSDR and more specifically in the for_next branch. Hence, after leaving the Buildroot directory to create a new tree structure:

1. `git clone https://github.com/oscimp/PlutoSDR`
2. `cd PlutoSDR`
3. `git checkout for_next`
4. `source sourceme.ggm`

The `BR2_EXTERNAL` approach is an interesting solution as long as the users wish to store their own custom configuration, whether for a dedicated project or to avoid, as was the case in the beginning of this article, manually copying configuration files, as well as `package` under development and integration. In the latter case, this solution is not the end of the story. If a `package` integration is validated, it is important to share it with the developer community in order to try and include in the official repository. Indeed, if some users have met the need for this package, it might not be impossible that this would be the case for others as well. This process is in progress for `libuhd` and `gnss-sdr`.

Now that the `BR2_EXTERNAL` repository has been downloaded, the appropriate branch selected and the environment variables set (last command), return to the Buildroot root directory and `make menuconfig`. Executing `make menuconfig` now opens the access to a new menu `External options` including `gr-iio`, `libad9361-iio` or `gnss-sdr`.

In order to add the support for transfering data from the PlutoSDR to GNU Radio, select in `External options` the `gr-iio` option. In order to add support for the Ettus Research USRP platforms, select in `External options` the `uhd` package, and for the particular case of the B210, set the `b200 support` option as well as `python API support`.

We will furthermore tune the configuration prior to `dd` the image on the SD card by adding files in the `output/target` tree structure, for example the static network configuration in `etc/network/interfaces`, or copying the USRP *firmware* files from the host PC to the `usr/share/uhd/images` subdirectory of `output/target` so that these files are later available on the embedded target. Once the `output/target` content has been satisfactorily tuned, return to the Buildroot root directory and run `make` to regenerate the file `output/images/sdcard.img`.

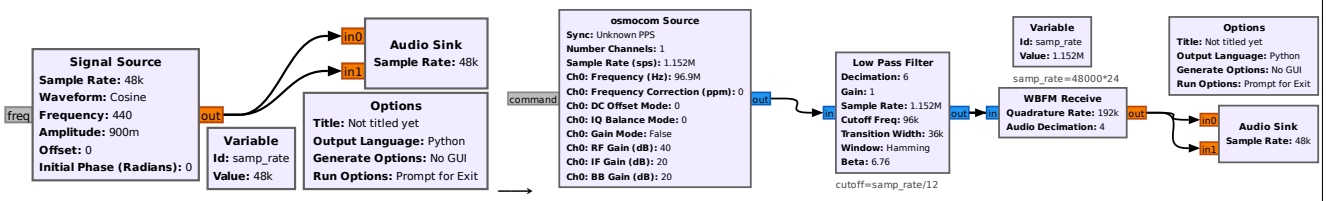
6 GNU Radio on Raspberry Pi 3/4

In order to demonstrate how we shall address using GNU Radio on embedded boards, we use GNU Radio Companion running on the host PC to generate a command line application ("No GUI") since obviously no graphical interface should be available on an embedded target. The resulting Python3 script will be executed on the RPi. The audiofrequency stream resulting from demodulating commercial broadcast FM band signals will be transmitted to the PC to be played on its sound card.

Sound on the Raspberry Pi 4

We hesistate as to what the simplest demonstration is, between using the PC sound card after transfering the audiofrequency stream resulting from the demodulation on the RPi 4 which assumes a properly configured TCP/IP interface for transfering data (which is anyway needed to transfer the Python script from the PC to the RPi 4), or emitting the sound from the 3.5 mm audio jack of the RPi 4 before addressing the network configuration. The reader willing to test the latter solution can

1. activate the sound on the Buildroot image by editing in the first partition of the SD card the `config.txt` file to add the option `dtparam=audio=on`. During the next boot sequence, the message `bcm2835_audio bcm2835_audio: card created with 8 channels` will display,
2. validate proper sound card operation by activating `speaker-test` of the `alsa-utils` package in `Target packages` → `Audio and video applications` and executing `speaker-test -t sine -f 440` to play a 440 Hz tone on the speaker output
3. validate that the sound card is properly selected by GNU Radio with the following trivial processing chain on the left, noticing that *two* audio inputs must be connected,



Generating the Python script and transfering to the RPi 4 for executing are described in the text,

4. extend the processung chain on the left with an Osmocom source and FM demodulation after selecting, thanks to a low pass filter, a single broadcast station and play the resulting sound output.

On the PC, launch GNU Radio Companion (provided by GNU Radio 3.8) and generate the processing chain of Fig. 2

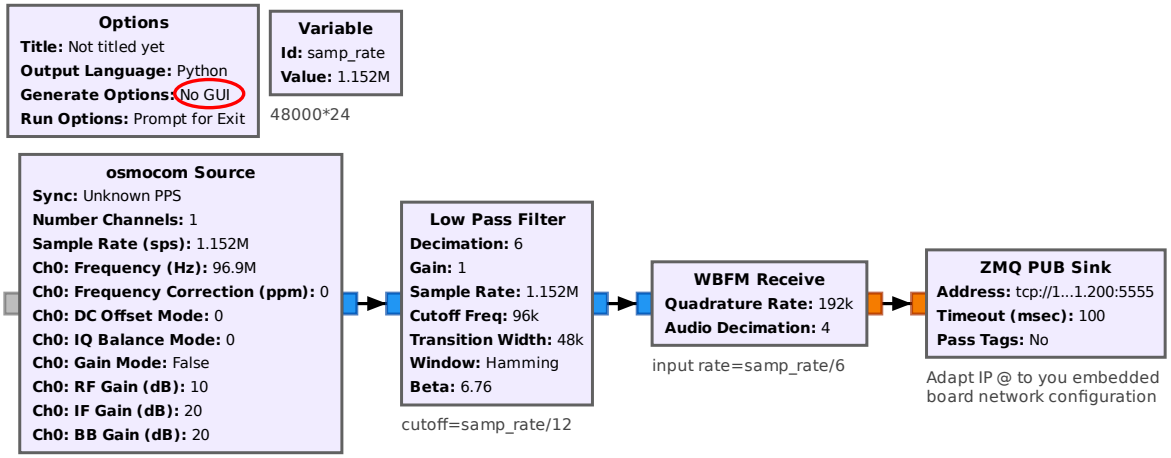


Figure 2: Command line processing chain to acquire and demodulate a broadcast commercial FM station in order to send the audio stream to a PC.

The resulting Python script is tranfered to the RPi. Make sure to adapt the IP address of the TCP link on the 0-MQ interface to the RPi address: the server is running on the embedded platform and using a Publish-Subscribe communication link (similar to a UDP broadcast), so that any client connecting to the server running on the embedded target can fetch the datastream. The IP address is ideally included in the PC subnet address to ease routing configuration, whereas the port index can be any number above 1024. The only constrain on this processing chain is at the end to reach a sampling rate equal to that of the PC sound card (here 48 kHz) following the successive integer decimation factors of the inital sampling rate set to 48 kS/s. The first low pass filter selects a single broadcast FM station while still keeping enough bandwidth (≥200 kHz) for properly demodulating the Wide Band FM signal, and the FM demodulator adds the second decimation step.

7 Communicating from the Raspberry Pi to the PC

After processing the raw radiofrequency data (I/Q) acquired in the commercial broadcast FM band by the RPi, and having pre-processed the FM signal on the embedded platform, the audio stream is transfered to the PC through a 0-MQ interface. This layer on top of TCP/IP (`https://rfc.zeromq.org/spec/13/`) offers two communication modes, connected as would TCP and non-connected as would UDP. In the former case, data transfer is guaranteed and acknowledged by the receiver, while in the latter case the server only broadcasts data over the network and a client might grab them for processing, otherwise they will be lost and acquisition continues on the server independently of the communication with a client. This second mode is the one used here, called Publish (RPi server) - Subscribe (PC client).

The RPi network interface is configured on the same sub-network than the PC, and we fill the server address to the IP address of the *Ethernet interface* of the RPi (and not 127.0.0.1) so as to listen to any incoming connection from the PC, for example with `tcp://192.168.0.42:5555`. Thus, the PC will connect its Subscribe client to this same address and collect the datastream (Fig. 3).

This result may seem trivial, but it shows a fundamental concept of the reception of radiofrequency signals: the successive processing steps **can only lose information**, and as Shannon explained [7] the information content is directly related to the signal bandwidth. Reducing the information content of the signal as successive demodulations steps are applied can only remove information (the redundancy introduced by FM modulation for example) and thus reducing the spectral bandwidth of the signal, thus the datarate needed to transfer. This explains why in a Software Defined Radio (SDR) receiver we strive to pre-process the acquired signal as much as possible in the FPGA that acquires the radiofrequency signal stream following the frequency transposition to baseband to reach a datarate compatible with communication with the general purpose processor. This general purpose processor performs as many pre-processing steps as possible – here low-pass filtering to only select a single FM station and signal demodulation – before communicating to the next destination, here the PC which will be used as sound card and connected by Ethernet through of Zero-MQ protocol. The subtle tradeoff will always be to process as much as possible upstream in order to reduce the bandwidth required to transmit to the next unit, and the complexity of implementing this processing or the available resources. Close to the receiver, we are satisfied with simple processing steps (because challenging to implement in the FPGA such as the PL of a Zynq) but excessively fast, whose output is transfered to the local processor (for example the Zynq PS) for further processing limited by the relatively modest computational power of its dual Cortex-A9, before transfer to the PC

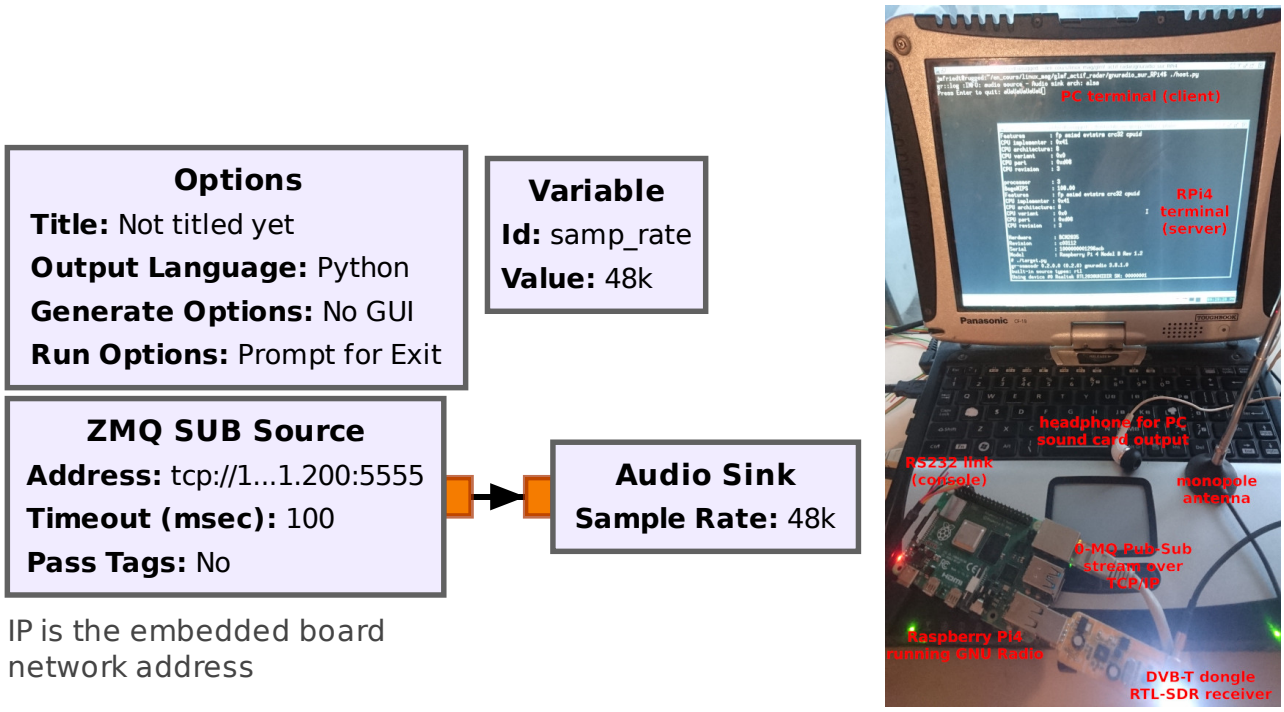


Figure 3: Left: processing chain on the client, which receives a stream in the 0-MQ *subscribe* format and feeds the PC sound card. Right: experimental setup, with an RPi 4 connected via a virtual serial port and Ethernet to the laptop. The RPi 4 samples the I/Q coefficient flow of the DVB-T receiver as a software defined radio source by tuning its frequency to the broadcast commercial FM band, and transmits the audio-frequency stream after demodulation to the PC, allowing to listen to the station output using a headset connected to the sound card output. This figure does not illustrate the excellent audio quality heard at the sound card output, demonstrating how well this setup works.

through the relatively slow communication bus (by the standards of software defined radio) USB2, a PC with a powerful processor that will finalize the processing. In this example, we start from a signal sampled by the RTL-SDR receiver at a multiple of 48 kHz, here the 24×48 kHz complex 8 bit samples would require $48000 \times 24 \times 2 \simeq 2.3$ MB/s bandwidth to be transmitted to the next processing platform. By demodulating the FM signal on the Raspberry Pi, we reduce the data flow to a few real floating point numbers encoded on 32 bits at a datarate of 48 ksamples/s or 192 kB/s. The benefit in terms of bandwidth is therefore obvious, and promotes handling as much pre-processing as possible on the embedded board, depending on the available computing power, prior to communicating.

8 Graphical interface on the Raspberry Pi 3/4: Qt5

It might sometimes be interesting to develop a fully autonomous application that does not require a computer associated with the RPi. Since the RPi 4 is fitted with a micro-HDMI port, a graphical output can be considered. In order not to impose the burden of X11, it seems relevant to directly communicate with the video memory through the framebuffer.

Qt5 support requires very large resources and its relevance should be carefuly considered before starting this compilation. Activating Qt5 support is achieved by selecting

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

- the package qt5 should be displayed from the beginning since `glibc` has been selected instead of `uClibc`.
- considering the modifications to the `gnuradio` options (activation of python et qtgui), restarting the compilation procedure with `make gnuradio-reconfigure` is mandatory

- the `liberation` package must be manually activated since no font is present in the default configuration.

Once the resulting image has been transfered to the SD card and executed on the RPi 3 or 4, a Qt5 GNU Radio application can be executed (here on an RPi 4) with `python3 ./rpi.py -platform linuxfb:fb=/dev/fb0` to output on a screen the graphical display (Fig. 4).

The screenshot tools provided by Buildroot do not work with the framebuffer of the RPi 4. The various figures of this article were acquired by copying directly the content from the framebuffer memory to a file `cat /dev/fb0 > screen.raw` and converting the pixel sequence blue, green, red and transparency to a more common compressed format with `convert -size 1280x800 -depth 8 bgra:screen.raw outfile.png` by being careful to tune the image resolution to that of the screen connected to the micro-HDMI port of the RPi 4.

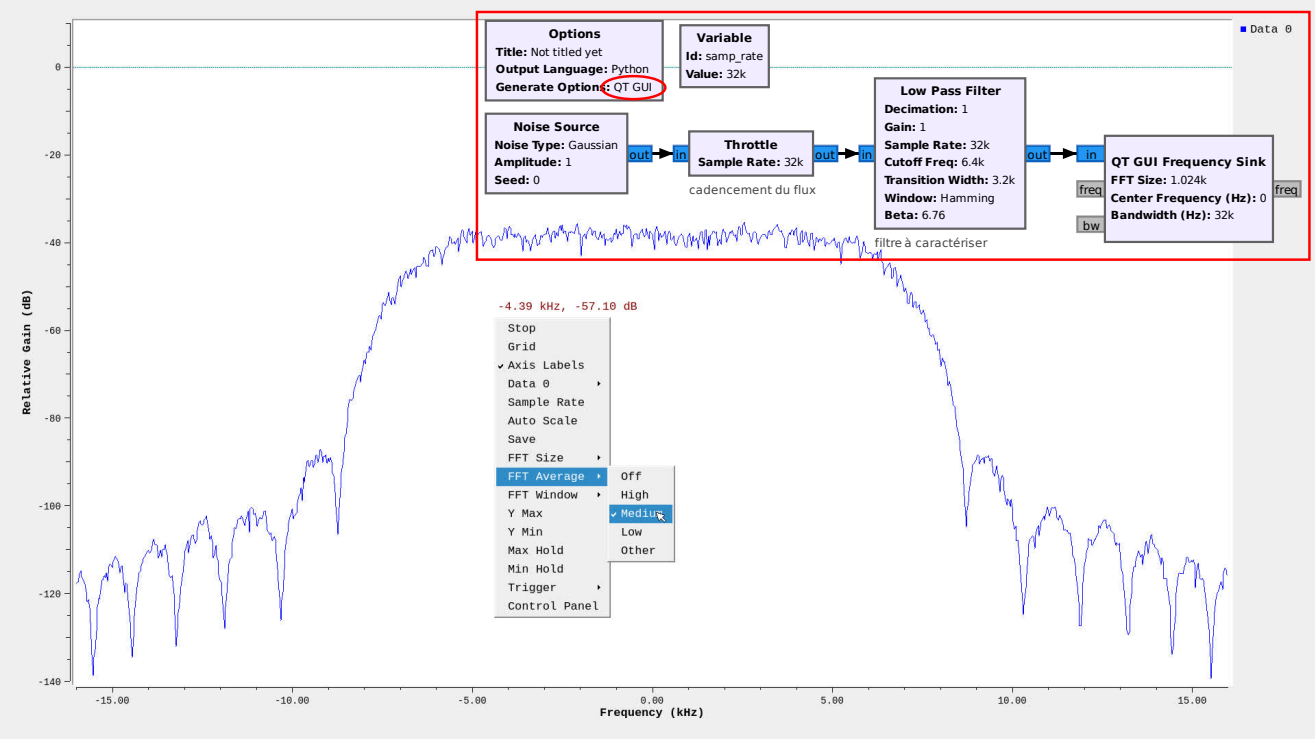


Figure 4: Qt5 application executed on a RPi 4 illustrating the graphical output resulting from the GNU Radio processing chain allowing to characterize the transfer function of a low pass filter.

9 Software developments

Thanks to the cross-compilation toolchain provided by Buildroot, generating an executable for our embedded target trivialy requires adding the `aarch64-buildroot-linux-gnu-gcc` compiler found in `output/host/usr/bin` of the Buildroot directory to the `PATH`, and using this compiler as variable of `CC` in the `Makefile`. As an example of using PiFM to emit a radiofrequency signal from the GPIO4 output of an RPi 4, we download `https://github.com/ChristopheJacquet/PiFmRds`, update the `Makefile` in `src` which assumes being executed on the embedded target (**never !**) by defining `CC = aarch64-buildroot-linux-gnu-gcc` and `RPI_VERSION := 4` followed by `make` to generate `pi_fm_rds` which is copied (`scp`) to the target board in order to execute. It is most entertaining to use a DVB-T receiver running GNU Radio demodulating the FM signal to receive the signal broadcast from the embedded target running PiFM: both applications get along perfectly.

Many software is however now provided by a Makefile generator named `cmake` which makes porting to various distributions and operating systems easier when compiling free opensource software to many platforms. Whether for modifying `gnss-sdr` with custom features or our own ACARS message decoder as transmitted from planes to ground (Fig. 5), understanding how to integrate `cmake` scripts in Buildroot is mandatory. In order to generate Makefiles able to cross-compile a program to the target, we use

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

to use the Buildroot toolchain and store the compilation result in the image that will be ready to be transfered to the microSD card. Alternately, we can compile to any output directory on the PC whose content will be copied (`scp`) to the `/usr` directory of the embedded board if flashing the microSD card with a new image is not desirable.

We aim at updating the capabilities of `gnss-sdr`. The source code of this software, relying on GNU Radio, has been downloaded and stored in the `output/build` directory when selected and installed. The source code in which the software will be compiled for the embedded target is `output/build/gnss-sdr-0.0.13/buildroot-build/` while a separate directory `output/build/gnss-sdr-0.0.13/build` allows for simultaneously testing modifications to the source code on the host PC. The result of compiling (`make`), either in `buildroot-build` (ARM target) or `build` (x86 target), is stored in `src/main/gnss-sdr`.

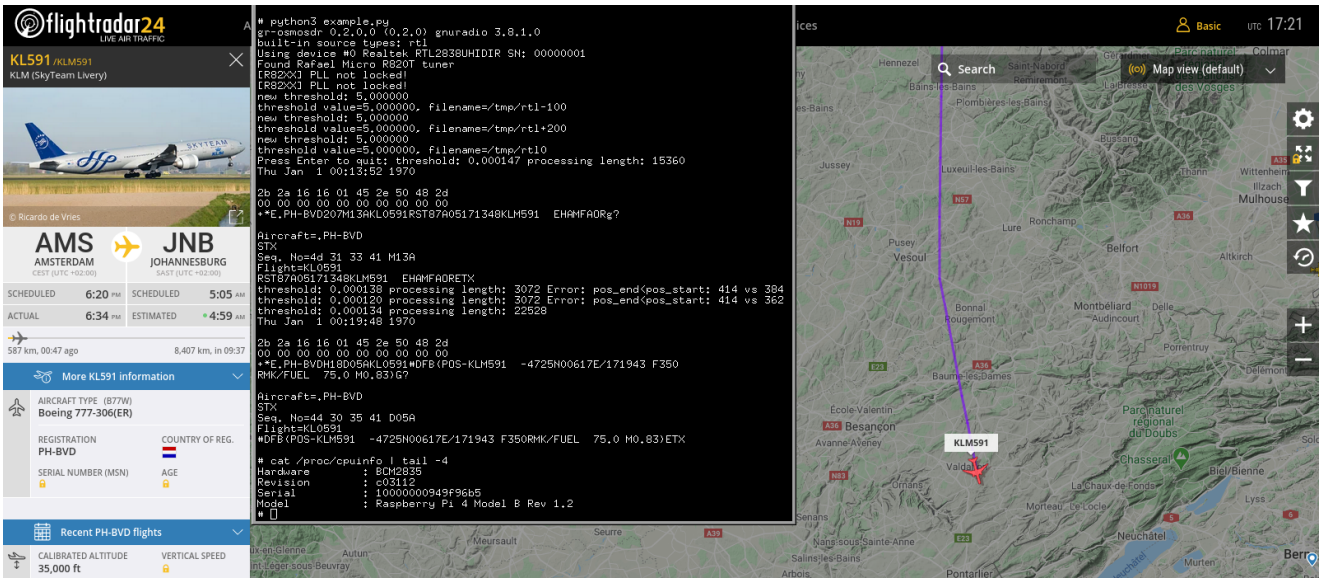


Figure 5: Executing `gr-acars`, ACARS message decoder, on RPi 4, and display as a background image the map of the planes close to the receiver during acquisition.

10 Executing in `qemu`

Assume that we have not received the appropriate hardware but we still wish to train using Buildroot and the generated binary files. The RPi 4 is too new to be integrated in QEmu, but the RPi 3 is claimed as functional. We hence install QEmu emulating a 64 bit ARM processor (`qemu-system-arm` package for Debian/GNU Linux which is immediately complemented with `qemu-system-gui` for adding the graphical interface emulation support) available through the executable `qemu-system-aarch64`. Thus, with

```
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
```

we execute the `Image` kernel which loads the hardware configuration `bcm2710-rpi-3-b` described in the devicetree, read the filesystem available on the second partition of `sdcard.img`, running on an emulation of the RPi 3 (option `-M raspi3`) fitted with 1 GB RAM.

In order to demonstrate that even the *framebuffer* can be emulated, Fig. 6 displays a screenshot of the simulation of a low-pass filter characterization by GNU Radio.

Incredibly, `qemu` can even access hardware connected to the host. This aspect is illustrated with a DVB-T receiver connected to the PC executing `qemu` emulating the RPi 3 (Fig. 7). Right, `lsusb` on the host (PC) indicates that the dongle is detected on the bus `X=1`, address `Y=14` with Vendor ID 0x0BDA and Product ID 0x2838 associated with the Realtek 2832U. Providing these parameters as arguments to the option `-usb -device usbhost=X,hostaddr=Y`, we observe (left screen) that `lsusb` in `qemu` indeed detects the dongle. Executing a signal processing chain fed from the DVB-T thanks to the Osmocom Source in GNU Radio (bottom right) in `qemu` leads to the peripheral being accessed and acquiring some samples. Honnesty requires that we mention that this screenshot is the *unique* case, on a few tens of trials, when `qemu` did not crash when gathering the datastream, whether attempting to limit the data acquisition to a few samples (`head`

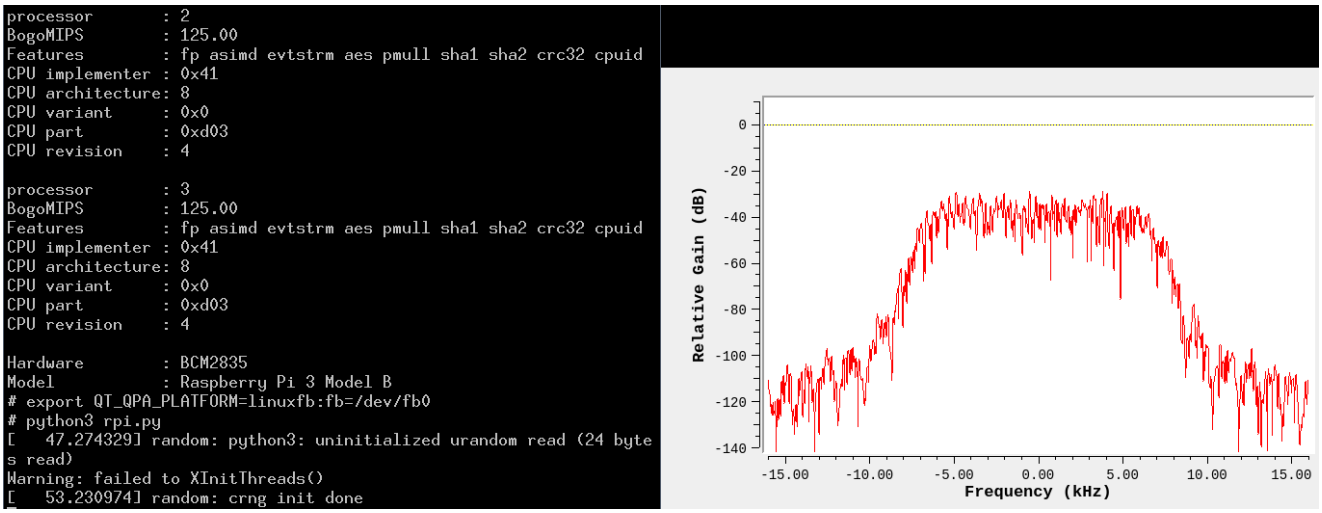


Figure 6: Executing in QEmu a GNU Radio signal processing chain with a Qt5 graphical output.

block used here) or by removing the graphical interface output. Transfering a USB stream from the USB bus emulation is thus functional but unable to handle large datarates (a few Msamples/s), which does not take anything away from the technical prowess of the RPi 3 emulation.

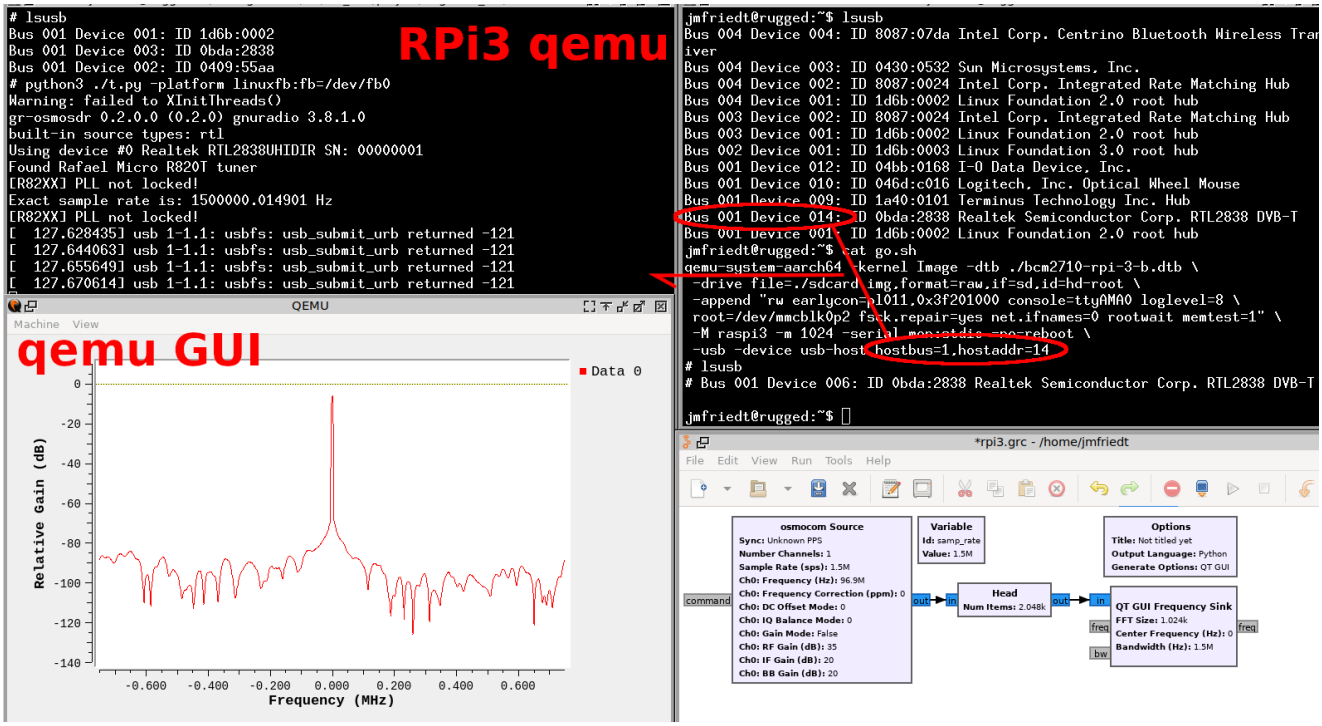


Figure 7: The qemu emulator collecting the datastream from a DVB-T dongle connected to the host USB port.

11 STM32MP157 graphical interface

The STM32MP1 is a processor designed and manufactured by ST Microelectronics fitted with a powerful ARM A7 processor able to run GNU/Linux, and small Cortex M4s dedicated to simpler tasks without the burden of running an operating system. Our interest for this heterogeneous multiprocessor platform (HMP) lies in its capability of off-loading the main processor by deporting some processing functions to the co-processors, assuming that communication time is not too long and does not exceed the time saving in execution. Without focusing on raw computing power, this is an opportunity to explore HMP without the development challenges of working with FPGA as met on a Zynq for example. The evaluation board provided by ST Microelectronics named STM32MP157-DK2 has a screen and is therefore ideal for showing how

the concepts presented so far are transposable to any platform supported by Buildroot.

Here again screenshots will be hindered by compatibility issues between the framebuffer implementation since `fbdump` works properly (Fig. 8), but `fbgrab` does not.

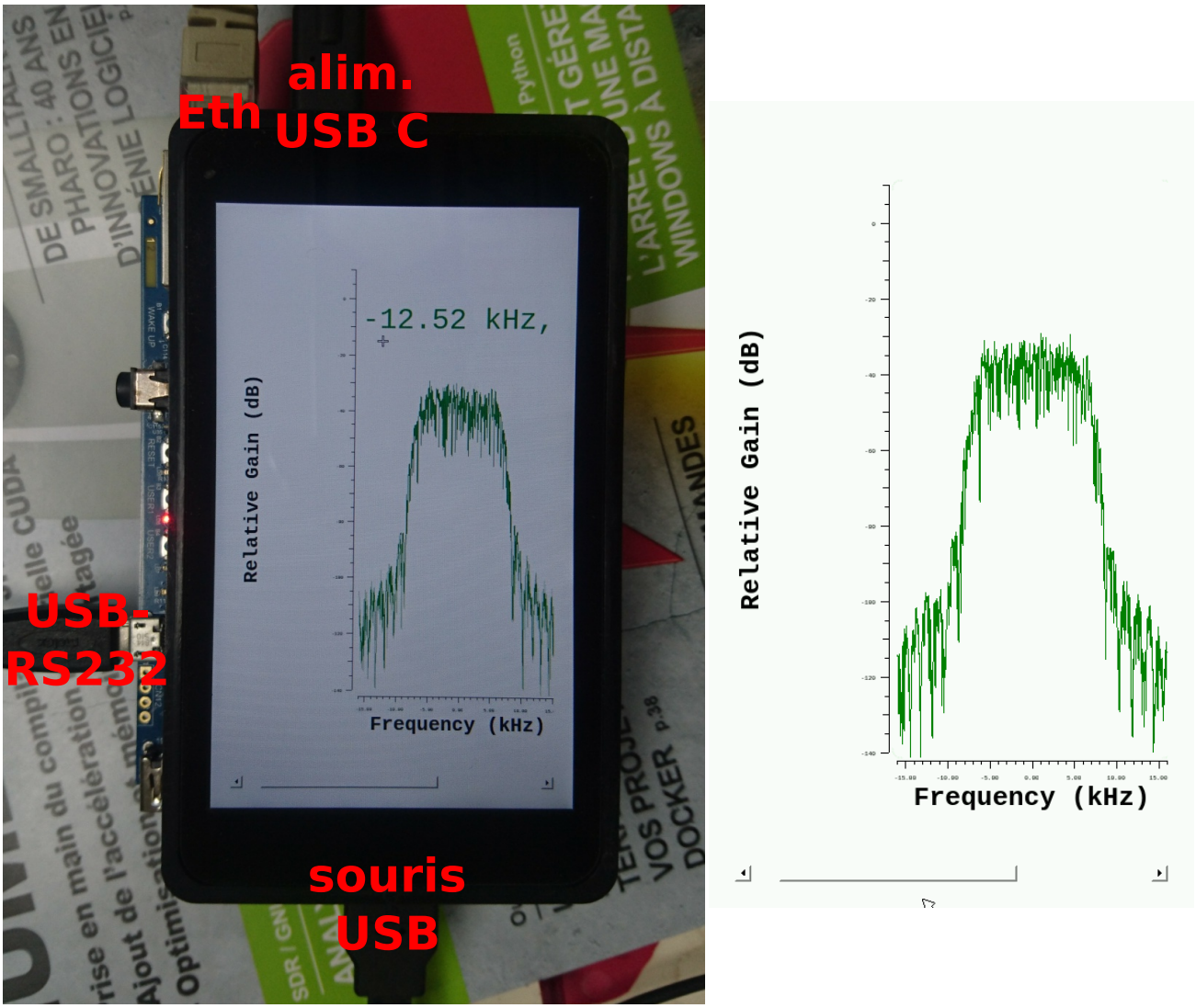


Figure 8: Left: picture of the STM32MP157-DK2 executing GNU Radio to simulate the transfer function of a low-pass filter. Right: screenshot of the graphical display of the STM32MP157-DK2 executing this processing chain with a Qt5 output.

12 Adding new packages

This description of packages is the most interesting and the one we focus on when adding support for new applications, here GNU Radio and associated libraries.

Each package, whether a library or an application, can be considered as a recipe for

- describing the package and its relation to other packages (`Config.in`);
- describing which packages must be built before for generating a tree of dependencies and configuration options applied during the compilation depending on the developer selections (`.mk` file).

The relations between packages are a core topic when generating the dependency tree. Two kinds of relations exist:

- `depends on`: when `pkg1` depends on `pkg2`, meaning that `pkg2` must be manually activated prior to allowing for the selection of `pkg1`. For this reason, a user running `make menuconfig` might not see some of the packages appear as long as the dependencies are not activated,

- `select`: this relation is the easiest to use since activating `pkg1` automatically selects `pkg3` with no explicit action to activate the latter.

Why use `depends on` instead of `select`? This choice depends on the situation: Python support in GNU Radio is compatible with either Python2, or with Python3. However, Buildroot cannot decide which version to activate: the user must select the active version during the configuration step. A second reason is that if `pkg2` requires many dependencies to be met, using `select` means that the whole set of these dependencies appear in the configuration of `pkg1`. This fact hence does not restrict the presence of `depends on` but makes maintenance harder, since all modifications in `pkg2` must be brought to `pkg1`.

Given the absence of the packages that are not yet visible due to of invalid dependencies, it is important to understand and master the mechanism for *searching* that will display all the solutions matching a given keyword and will possibly provide the solution to make this package appear in the menus. The search is activated, as under `vi`, by "`/`" followed by the keyword.

This mechanism of dependencies is activated during the first compilation, but leads to issues during the evolution of the configurations. Indeed, starting from the file `_defconfig`, we get a functional image, but with minimalist functionalities offering only a few tools from `busybox`. Let us imagine that we now want to add GNU Radio: `make menuconfig` and `gnuradio` allows to reach this reslut. However, we have forgotten to activate Python3 support and cannot execute scripts that were generated by GNU Radio Companion. When activating Python3 and its support by GNU Radio, we must reconfigure the latter through `make gnuradio-reconfigure` so that Python support is taken into account. In the same way, this option is incompatible with the `uClibc` – a minimalist implementation of `libc` system calls – but need to use the more complete version provided with `glibc`. This modification implies to recompile the whole toolchain and everything that has been compiled with it: to avoid disaster, we might as well start all over again from the beginning by deleting the content of the `output` directory with `make clean` and restart the entire Buildroot build process.

To conclude this discussion:

- if an option from the compilation toolchain is modified, the whole of Buildroot must be compiled again, starting from a clean `output` directory,
- when modifying an option of package `mypkg` impacting its dependencies, this package must be reconfigured manuall using `make mypkg-reconfigure`. Depending on the impact of this update, recompiling the whole of Buildroot is sometimes easier.

We illustrate this procedure by adding a package, following the detailed description at `http://buildroot.uclibc.org/downloads/manual/manual.html#adding-packages` [8], on the example cited earlier of `gr-acars` which depends on GNU Radio. The files we must update to add this support are:

- the `package/gr-acars/gr-acars.mk` file which holds

```
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))
```

  providing the hash key of the Sourceforge repository of our code, the method used to download source codes, the licence and dependencies. The subtelty lies here in adapting the dependence to the version of Python (2 or 3) which was selected when GNU Radio was configured,

- the `package/gr-acars/Config.in` file which holds the description of the package as will be shown to the user, with

```
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/
```

We notice two points on this file description:

1. a `select` is used for `gr-osmosdr` since the list of dependencies is restricted to `gnuradio`. Since `gr-acars` depends on `gnuradio` as well, it is not necessary to use `depends on`
2. the line `select BR2_PACKAGE_GNURADIO_FFT` is commented (shown for teaching purpose) since this option is activated by `BR2_PACKAGE_GNURADIO_FILTER`: activating this option a second time would not be wise here.

- the `package/gr-acars/gr-acars.hash` file which holds

```
# Locally calculated:
sha256  d3e8... gr-acars-fd70b325906e02b758b89786a6470d05bfee0a67.tar.gz
```

resulting from the command
`sha256sum dl/gr-acars/gr-acars-fd70b325906e02b758b89786a6470d05bfee0a67.tar.gz` since Buildroot clones the `git` repository prior to assembling a compressed archive, the ellipsis representing the long signature string generated by the command. This signature is used to validate that the downloaded archive is not corrupt and matches expectations.

- Finally, the `package/Config.in` file is appended with

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

so that the newly added package appears in the associated menu.

Following these modifications, searching the string ACARS in `make menuconfig` leads to `gr-acars` appearing, which will be compiled in the resulting image of selected.

13 Conclusion

We have ported GNU Radio and its dependencies to Buildroot so it runs on the multitude of embedded platforms supported by this cross-compilation environment, including the Rasperry Pi 3/4. We thus have the basics to then run complex infrastructures such as `gnss-sdr`. In doing so, we have discovered the limits of Buildroot (lack of detection of configuration modification of a package) but also the flexibility with which new packages can be added. An infrastructure as complex as GNU Radio, with multiple dependencies to support a wide range of acquisition and communication interfaces up to the Qt5 graphical user interface, is now supported by Buildroot. The performance gain of a tailored compilation targeted to a given board is unquestionable with respect to a general purpose binary distribution. The precepts of embedded system development of getting the most out of available resources and minimizing energy consumption and memory footprint to achieve the targeted performance are therefore achieved.

References

[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)

[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) at `https://bootlin.com/doc/training/buildroot/buildroot-slides.pdf`