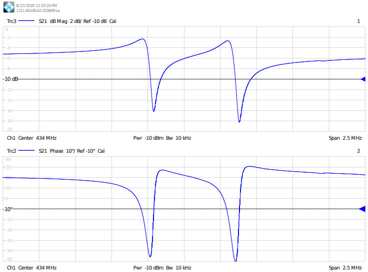
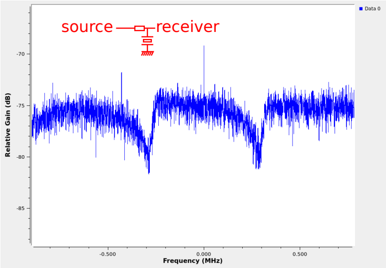
Porting GNU Radio to Buildroot: application to an embedded digital network analyzer

feedback on a graduate course on developing an embedded network analyzer

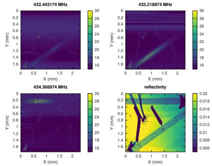


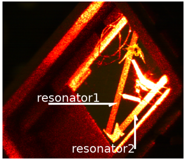
G. Goavec-Merou, J.-M Friedt

FEMTO-ST Time & Frequency, Besançon, France

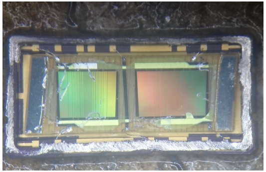


References at `http://jmfriedt.free.fr`





January 14, 2021

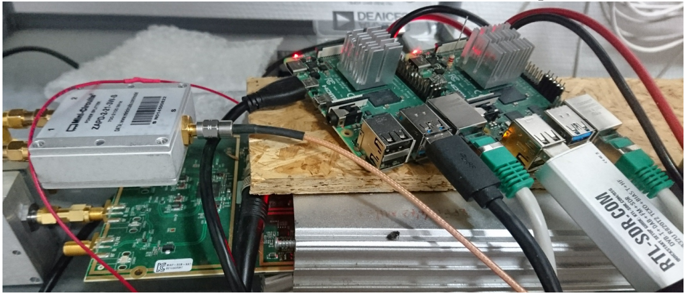


Outline

1. The Buildroot framework (kernel + library + userspace application + toolchain)
2. GNU Radio running on the target system (Raspberry Pi4) – demonstration with FM broadcast radio demodulation and sound transfer to the host used as sound card [1].
3. RPiTX as flexible signal source to probe the Device Under Test: embedded network analyzer

Buildroot on host (PC) —make→ sdcard.img on PC —dd→ sdcard.img on SD card —SD card on RPi4→ filesystem with GNU Radio on RPi4 → ...

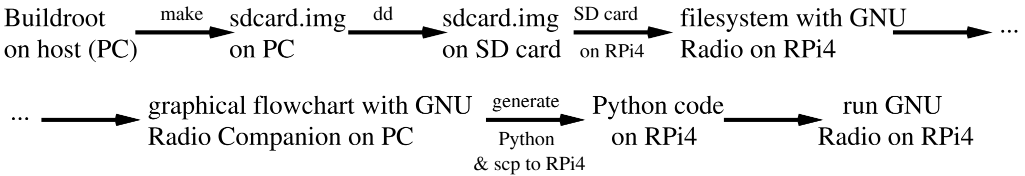
... → graphical flowchart with GNU Radio Companion on PC —generate Python & scp to RPi4→ Python code on RPi4 → run GNU Radio on RPi4

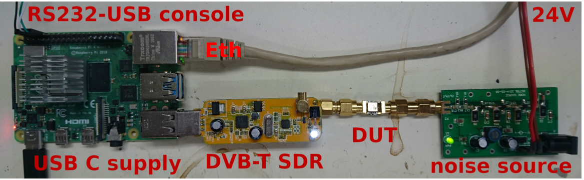


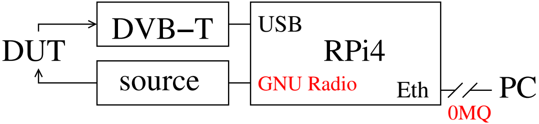
https://github.com/oscimp/gnss-sdr

Outline

1. The Buildroot framework (kernel + library + userspace application + toolchain)
2. GNU Radio running on the target system (Raspberry Pi4) – demonstration with FM broadcast radio demodulation and sound transfer to the host used as sound card [1].
3. RPiTX as flexible signal source to probe the Device Under Test: embedded network analyzer







[1] G. Goavec-Merou, J.-M Friedt, *"On ne compile jamais sur la cible embarquée" : Buildroot propose GNURadio sur Raspberry Pi (et autres)*, Hackable, to be published, at `http://jmfriedt.free.fr/hackable_buildroot.pdf`

Performamce: Buildroot v.s Raspbian v.s Ubuntu (RPi4)

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

C.J. Murray, *The Supermen: The Story of Seymour Cray and the Technical Wizards Behind the Supercomputer* , John Wiley & Sons (1997)

Embedded system development under GNU/Linux

Embedded systems development is about **optimizing resources** (lower power consumption for maximum computational power) ⇒ **don't compile on the target !**
Functional GNU/Linux (**host** = Intel x86) environment:

- develop for the **target** ARM board by cross-compiling: need for a consistent toolchain (compiler and binary handling utilities), kernel (Linux), libraries and userspace applications
- several frameworks provide such consistent functionaliy (Yocto, OpenEmbedded, Buildroot) – the latter being arguably the easiest to grasp and requiring fewer resources (*8 GB hard disk space*)
- fetch the latest stable release of Buildroot:
  wget https://buildroot.org/downloads/buildroot-2020.11.1.tar.gz
  (or check `https://buildroot.org/download.html`)
- *do not attempt* moving the Buildroot directory to some different location after configuring: some hard-coded directory structure will be broken

Embedded system development: initial compilation of Buildroot

- **ls configs/raspberrypi***: check available configurations (**raspberrypi4_64_defconfig**)
- **make raspberrypi4_64_defconfig** to configure with the default configuration
- **make** to compile Buildroot: many archives will be downloaded ⇒ about 8 GB
- Buildroot (BR) should be self-contained and independent of the host operating system assuming basic developer functions are available (**gcc**, **g++**, **make**, **git**, **cmake** ...)
- at the end: **output/images/sdcard.img** is the image to be transferred to the SD card
- bitwise copy from a file to a storage medium: **dd** (Disk Dump)
- ⚠ **WARNING:** the following command will **definitely delete** all data on the target medium. Make sure how the SD-card is called. It is usually **/dev/sdb** but in case a mobile hard disk/USB stick is inserted, it could be that the SD-card is called something else. Check many times before running **dd**
- identify the block name [2] using **dmesg** | **tail** after inserting the SD card reader, or **lsblk**

```
[514523.735373] scsi 6:0:0:0: Direct-Access     Mass     Storage Device   1.00 PQ: 0 ANSI: 0 CCS
[514523.735669] sd 6:0:0:0: Attached scsi generic sg1 type 0
[514523.994885] sd 6:0:0:0: [sdb] 31422464 512-byte logical blocks: (16.1 GB/15.0 GiB)
[514523.995006] sd 6:0:0:0: [sdb] Write Protect is off
[514523.995008] sd 6:0:0:0: [sdb] Mode Sense: 03 00 00 00
[514523.995129] sd 6:0:0:0: [sdb] No Caching mode page found
[514523.995133] sd 6:0:0:0: [sdb] Assuming drive cache: write through
[514524.024807]  sdb: sdb1 sdb2
[514524.025712] sd 6:0:0:0: [sdb] Attached SCSI removable disk
```

- **sudo dd if=output/images/sdcard.img of=/dev/sdd** (repeat for every BR modification)

[2] also make sure a file manager has not automagically mounted the filesystems stored on the SD

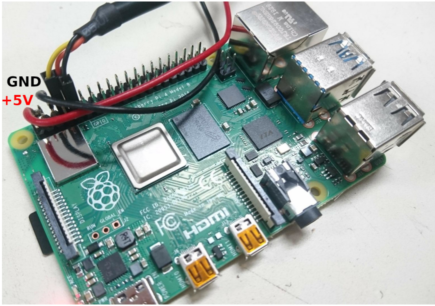
Network configuration

We need to connect the Raspberry Pi4 to the host computer

- point to point Ethernet connection easily established if host and target on the same sub-network
- On the SD-card: network configuration is handled by **/etc/network/interfaces**

```
iface eth0 inet static
  address 192.168.2.2
  netmask 255.255.255.0
```

- No Ethernet ? serial-USB cable to setup the configuration ⟶



- No Ethernet ? virtual Ethernet over USB-C[3] (routing table !)
  - first SD card partition: add `dtoverlay=dwc2` to **config.txt**
  - second SD card partition: add in **/etc/init.d/** an executable script **S01-module** with

    ```
    modprobe dwc2
    modprobe g_ether
    ```
- Secure SHell (**ssh**) server on target provided by **dropbear**
- in the Buildroot directory on the host computer: **make menuconfig** to configure BR with new packages
- search ("/") the keywork **dropbear** and select this package
- ssh server requires a root password: `System Configuration` → `Enable root login with password` → provide a password → **make** generates **sdcard.img** → **dd**

[3] `https://dev.webonomic.nl/4-ways-to-connect-your-raspberry-pi-4-to-the-internet`

Buildroot with GNU Radio support

GNU Radio requires multiple additional options not selected with the default Buildroot:

- **glibc** C library (instead of uClibc)
- **eudev** device handling
- Python3 support
- some additional GNU Radio options (Python support, 0-MQ ...)

Buildroot **cannot handle dependency changes** (Kconfig) ⇒ `make clean` for major upgrades

To avoid iterative selection of the Buildroot packages, a new **defconfig** file is available from

`https://github.com/oscimp/PlutoSDR/tree/master/configs`

Download **raspberrypi4_64_gnuradio_defconfig**, put the file in the local Buildroot **configs**, and restart the whole compilation

```
make clean && make raspberrypi4_64_gnuradio_defconfig && make
```

(should be faster since the downloaded archives are still in **dl/**): total disk space about 12 GB

- Check that GNU Radio is properly installed: on the RPi4,

  ```
  # python3
  import gnuradio
  ```

  must return with a prompt and no warning/error

Adding audio support

Audio is not active in the default Buildroot configuration.

To activate audio, add in the `config.txt` of the first partition of the SD card:

```
dtparam=audio=on
```

After booting, **dmesg** will now display

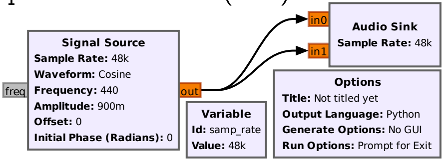
```
[    3.438439] bcm2835_audio bcm2835_audio: card created with 8 channels
```

ALSA[4] utilities have been installed on the custom Buildroot configuration supporting GNU Radio: test sound with

```
# speaker-test -t sine -f 440
```

My first GNU Radio flowchart running on RPi4

- Host: use PyBOMBS (Python Build Overlay Managed Bundle System) as described at `https://github.com/gnuradio/pybombs` to install GNU Radio 3.8 on your system
- no graphical output on the target: launch `gnuradio-companion` on the PC (host) and select **Options** → **Generate Options** → **No GUI**
- the **Id** defines the name of the output Python script
- **Run** → **Generate** converts flowgraph to Python script
- copy (**scp**) Python script from host to target
- target (Raspberry Pi4): execute **python3 my_script.py**



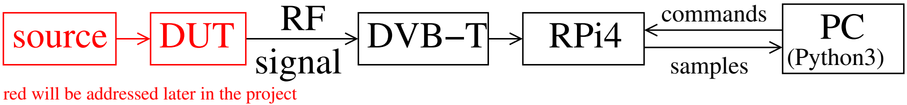
48 kS/s, stereo sink ⇒ tone on audio jack

[4] Advanced Linux Sound Architecture

Outline

We are now sure GNU Radio is properly installed and GNU Radio can access the sound card

General context: we wish to design an instrument in which the data are collected by the Raspberry Pi 4, under control of the PC, to be transferred to the PC for processing and display.



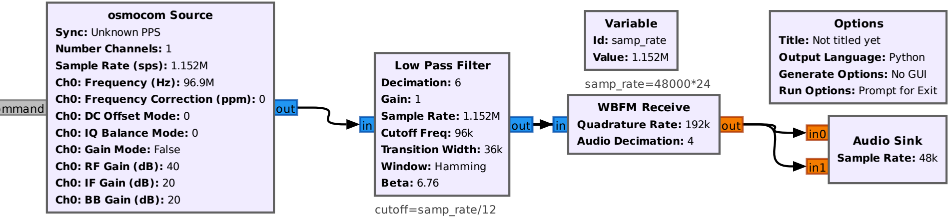
GNU Radio on Raspberry Pi 4

1. first demonstration with RTL-SDR dongle: FM receiver
2. from RPi4 to PC used as sound card: Zero-MQ publish/subscribe
3. from PC to RPi4: TCP/IP server running as a Python thread

Objective: a FM radio receiver running on the RPi4, streaming sound from the RPi4 to the PC, whose carrier frequency is controlled from the PC

GNU Radio on Raspberry Pi4: streaming from RPi4 to PC

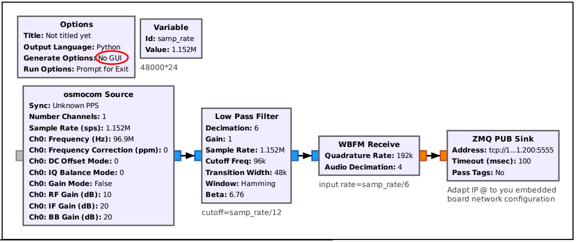
1. FM radio receiver to check proper operation of DVB-T dongle using the sound card

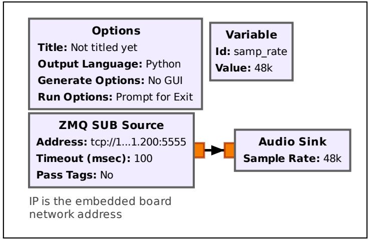


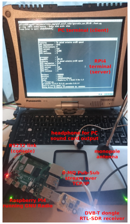
2. Streaming from RPi4 to PC[5]

target

host







[5] UDP-like Zero-MQ stream: Publish-Subscribe mechanism, **tcp://192.168.x.y:5555** is the *RPi4* Ethernet @ (listen)

Commands from PC to RPi4

Multithreaded Python script approach

- GNU Radio Companion is a Python script generator
- GNU Radio Companion 3.8 allows for inserting additional Python commands in its initialization code: **Python Snippets**
- GNU Radio Companion 3.8 allows for adding Python functions: **Python Module**
- Launch a separate thread running a TCP (connected mode) server
- Receive commands from the PC running a TCP client (**telnet**)
- Tune the GNU Radio flowgraph variables by calling the callback function associated with the modified variable

What is a thread ?

- function run in parallel to the main program but sharing the same memory space

```
import threading
import time

def jmf1(argument):
        while True:
                print(argument)
                time.sleep(1)

threading.Thread(target=jmf1,args=(1,)).start()
threading.Thread(target=jmf1,args=(2,)).start()
threading.Thread(target=jmf1,args=(3,)).start()
```

What is a server ?

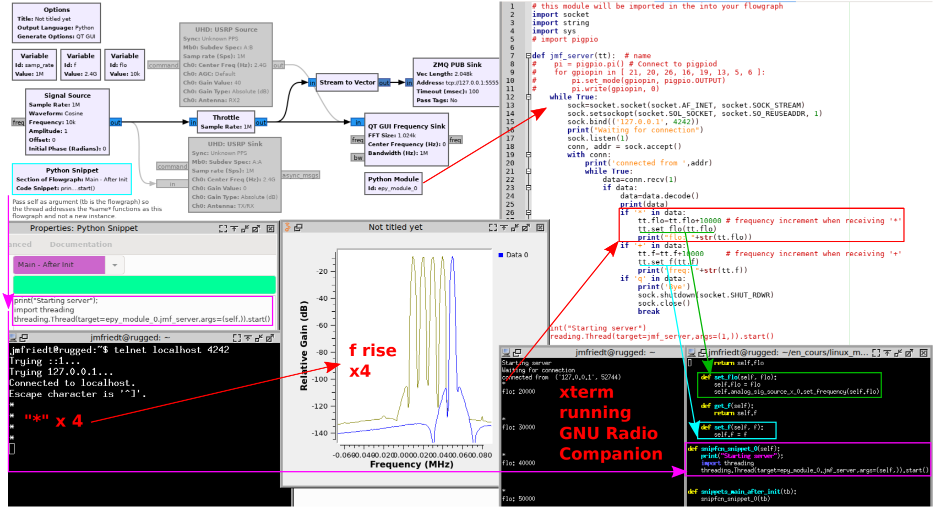
Definition: a *server* waits for a connection, a *client* connects to the server when it needs information

```
import socket
import string
while True:
    sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1', 4242))
    print("Waiting for connection")
    sock.listen(1)
    conn, addr = sock.accept()
    with conn:
        print('connected from ',addr)
        while True:
            data=conn.recv(1)
            if data:
                data=data.decode()
                print(data)
                if 'q' in data:
                    sock.shutdown(socket.SHUT_RDWR)
                    sock.close()
                    break
```

- Run `python3 my_server` in one terminal
- Run `telnet localhost 4242` in another terminal
- Enjoy ... quit by sending 'q'

Putting it all together ...

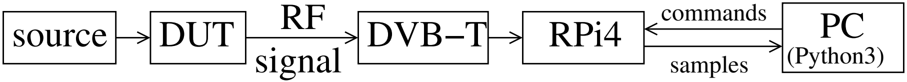
Python Snippet executes the thread including the Python Module running the TCP server controlling the GNU Radio execution by tuning parameters with the associated callback function



Modify the previous flowchart, streaming the output of the FM demodulator to the PC, to tune the broadcast station

Outline

General context: embedded network analyzer architectured around the Raspberry Pi 4 and using an RTL-SDR DVB-T dongle as radiofrequency receiver.



Emitting a radiofrequency signal from the Raspberry Pi 4 clock

1. Investigating radiofrequency emission sources
2. Using the RPi4 internal PLL feeding a GPIO as radiofrequency source
3. Making sure the radiofrequency is controlled and understood by receiving with the DVB-T dongle

Objective: emitting an FM radio signal from the Raspberry Pi4 and listening to the resulting sound [6]

[6] sample video of expected result: `http://jmfriedt.free.fr/201229_rpitx.mp4`

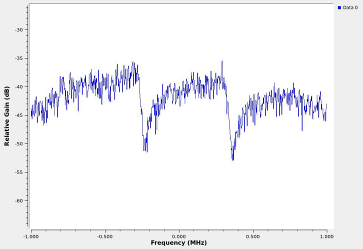
Radiofrequency sources

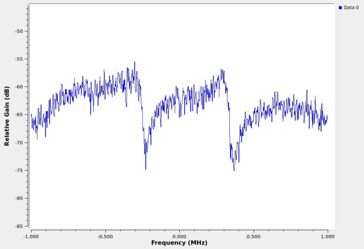
Characterize the transfer function of a passive Device Under Test

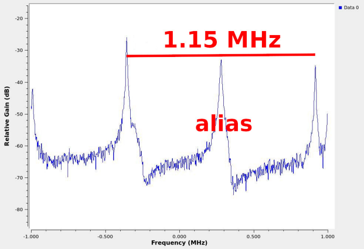
$\Rightarrow$ radiofrequency driving signal

- broadband = noise: Zener diode, but requires high (24 V) voltage for broadband signal + radiofrequency amplifiers
- pulse: must be short and sharp edges. Test with ADCMP fast comparators (e.g. ADCMP573 [7] for single supply operation): functional but requires an external trigger, e.g. RPi PWM

These solutions require additional, external hardware and are prone to artefacts ...







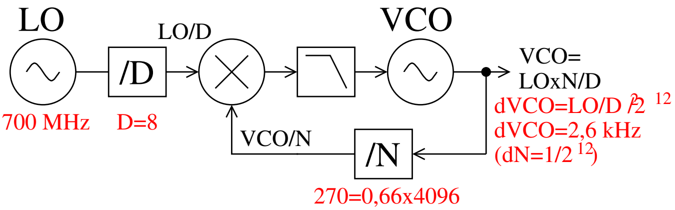
Broadband noise source | 40 ns pulse every 160 ns | 40 ns pulse every 800 ns

... but the RPi GPIO can be driven from a radiofrequency clock source ! See the PiFM project [8].

[7] https://www.analog.com/en/products/adcmp573.html

[8] http://www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter

Fractional PLL



- Raspberry Pi single board computers provide a reference clock LO (700 MHz for RPi4, 500 MHz for others)
- this clock feeds a fractional Phase Locked Loop (PLL [9]) with a pre-scaler of D
- the PLL Voltage Controlled Oscillator (VCO) is divided by N
- the phase comparator compares LO/D with VCO/N: $VCO = LO \times N/D$
- output frequency < 200 MHz (GPIO limitation) $\Rightarrow$ use overtone (5th overtone of FM band to reach 434 MHz ISM band)
- output frequency resolution: considering that $\boldsymbol{VCO = LO \times N/D}$ and that the resolution $\boldsymbol{dD}$ on $\boldsymbol{D}$ is $\mathbf{2^{-12}}$, frequency resolution at 434 MHz is $dVCO = LO \times dD/D^2$ by tuning the fractional part of the PLL
- since $dD = 2^{-12}$ and $D \simeq 8$ for a $433.92/5 = 86.8$ MHz $\Rightarrow dVCO = 2.7$ kHz $\ll$ DDS resolution but usable for $Q = 10^4$ @ 434 MHz (width $\simeq 43$ kHz).

[9] `https://elinux.org/The_Undocumented_Pi`

Many implementations derived from the original PiFM demonstration [10]:

- https://github.com/ChristopheJacquet/PiFmRds is easiest [11] to understand
- GPIO clock sourced from a fractional PLL is described pp.104–105 of https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0.pdf
- a much more general (and complex [12]) implementation is available at github.com/F5OEO/rpitx relying on github.com/F5OEO/librpitx
- interfacing the latter with GNU Radio {I,Q} stream is explained at https://github.com/ha7ilm/rpitx-app-note
- http://abyz.me.uk/rpi/pigpio/pigs.html explains that "Access to clock 1 is protected by a password as its use will likely crash the Pi. The password is given by or'ing 0x5A000000 with the GPIO number."

Our application only requires a single continuous-wave (CW) tone for a Frequency Swept CW analyzer

- Makefile based software: replace gcc with arm-linux-gcc from Buildroot output/host/usr/bin
- cmake based software:

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

[10] O. Mattos & O. Weigl, https://github.com/rm-hull/pifm described at http://www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter

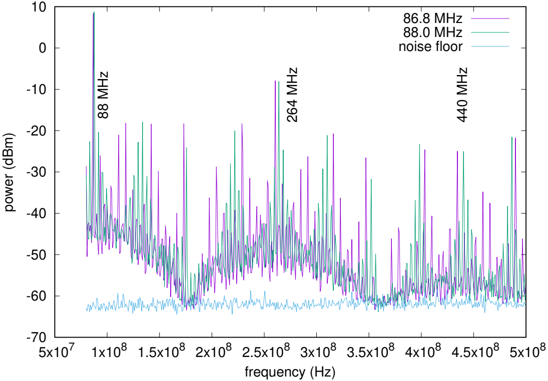
[11] github.com/ChristopheJacquet/PiFmRds/blob/master/src/pi_fm_rds.c#L534

[12] E. Courjaud *Rpitx: Raspberry Pi SDR transmitter for the masses*, SDRA (2017) at https://www.youtube.com/watch?v=Jku4i8t_nPc

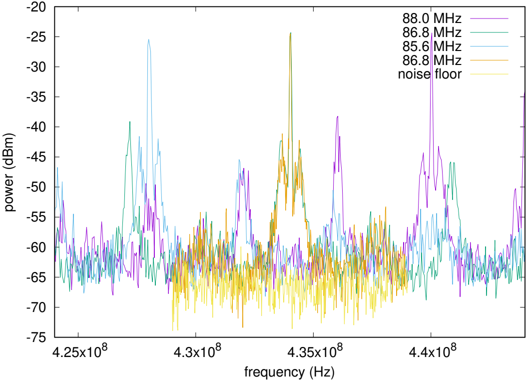
Overtone

The RPi GPIO has been observed to generate a strong signal up to 250 MHz.

We aim for the 434 MHz band ⇒ use overtones



Broadband spectrum

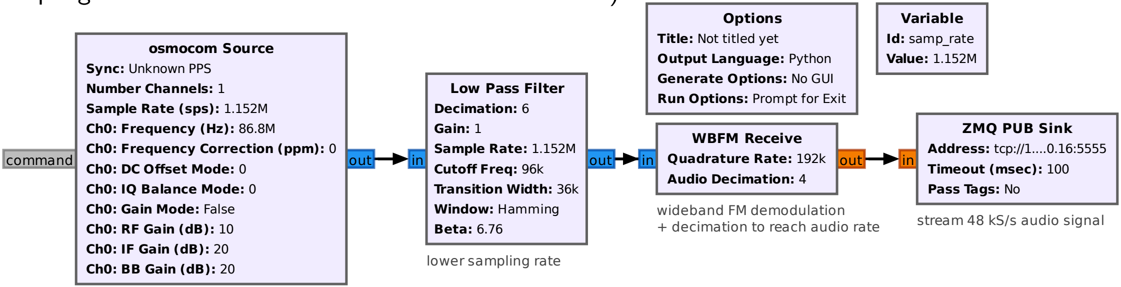


Zoom on the (FM modulated) overtone signal

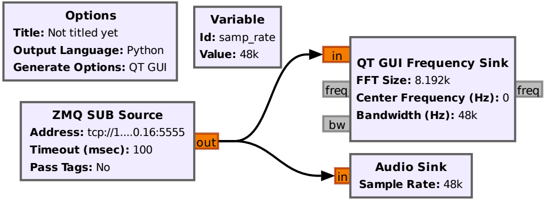
Square wave output ⇒ overtone N scales as $1/N$. Emit at $434/5 = 86.8$ MHz

FM emission/reception from the RPi4

On the **embedded board**: CLI flowchart for acquisition, demodulation and streaming (lowering the sampling rate and hence the communication bandwidth)



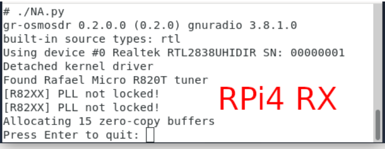
On the **host PC**: GUI for displaying the spectrum and playing audio on the sound card from the signal generated by PiFM(-RDS) ⇒ video @ `http://jmfriedt.free.fr/201229_rpitx.mp4`

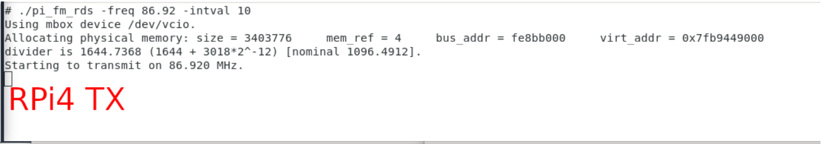


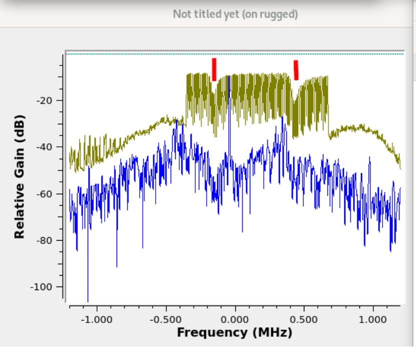
Conclusion: characterize the SAW resonator transfer function

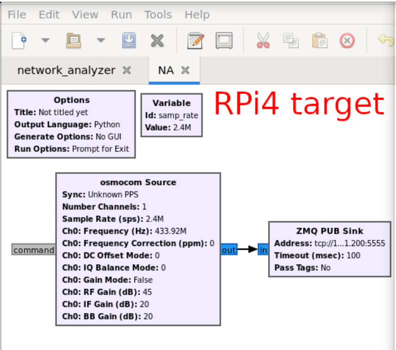
PiFM as BR2_EXTERNAL external package at https://github.com/oscimp/PlutoSDR/ in the for_next branch

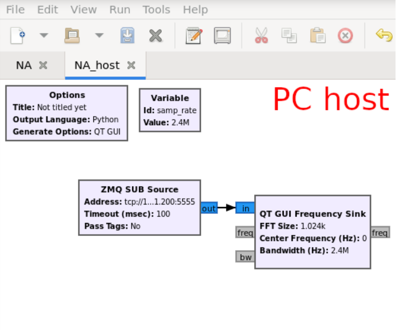
1. Modify PiFM or use https://github.com/F5OEO/rpitx/ → chirp to sweep a frequency (FSCW signal)
2. Generate signals and check that their spectra are consistent with expectations/frequency range
3. Control the emitted signal, in addition to the received signal, from the Python server





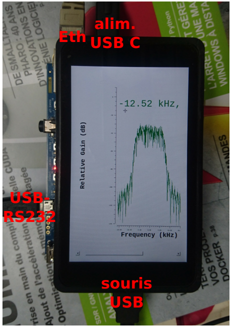


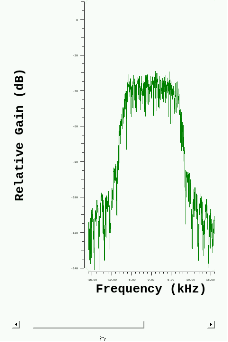




General conclusion

- GNU Radio port to Buildroot provides access to all boards supported by BR (PlutoSDR [13], RPi*, Beaglebone*, Redpitaya [14]/Zynq*, STM32MP157-DK2 ...)
- opportunity to become familiar with embedded development tools
- single board computer computational power has reached the level needed by SDR.





TODO (article on the FOSDEM web site)

- Using non-officially supported packages (e.g. `gnss-sdr`) with `BR2_EXTERNAL`

Further reading

- K. Yaghmour, J. Masters, G. Ben-Yossef, P. Gerum, *Building Embedded Linux Systems, 2nd Ed.*, O'Reilly (2008)
- J. Madieu, *Linux Device Drivers Development*, Packt (2017)
- C. Hallinan, *Embedded Linux Primer: A Practical, Real-World Approach, 2nd Ed.*, Prentice Hall (2010)
- M. Corbin, *Buildroot for RISC-V*, FOSDEM 2019 [15]
- P. Ficheux & É. Bénard, *Linux embarqué*, Eyrolles (2012) [French]
- P. Ficheux, *Linux embarqué – Mise en place et développement*, Eyrolles (2017) [French]

[13] https://github.com/oscimp/PlutoSDR
[14] https://github.com/trabucayre/redpitaya.git
[15] https://archive.fosdem.org/2019/schedule/event/riscvbuildroot/