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

1sample video of one possible outcome: https://www.youtube.com/watch?v=JIiKZ3UVAIw
Device under test: dual resonator surface acoustic wave temperature sensor

Operating range; Industrial, Scientific and Medical (ISM) band, [433.05, 434.79] MHz

![Graphs showing device performance](image-url)
Radiofrequency sources

Characterize the transfer function of a passive Device Under Test
⇒ 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\(^2\) 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 ...

---


---

... but the RPi GPIO can be driven from a radiofrequency clock source! See the PiFM project\(^3\).

\(^3\)http://www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter
Raspberry Pi single board computers provide a reference clock $LO$ (700 MHz for RPi4, 500 MHz for others)

$LO$ feeds a fractional Phase Locked Loop (PLL $^4$) with a pre-scaler of $I \in \mathbb{N}$

the PLL Voltage Controlled Oscillator (VCO) is divided by $Q \in \mathbb{Q}$

the phase comparator compares $LO/I$ with $VCO/Q$: $VCO = LO \times Q/I = LO/I \times \left(1 + \frac{F}{I \times 4096}\right)^{-1}$ (see $^5$) = $LO/(I + F/4096)$

output frequency < 125 MHz (GPIO limitation $^6$) $\Rightarrow$ use overtone (5th overtone of FM band to reach 434 MHz ISM band)

output frequency resolution: considering that $VCO = LO \times Q/I$ and that the resolution $dF$ on $F \in [0 : 4095]$ is 1, what is the frequency resolution at 434 MHz? Is it enough for our application? How does it compare with a DDS?

$^4$https://elinux.org/The_Undocumented_Pi#Clocks

$^5$https://datasheets.raspberrypi.org/bcm2711/bcm2711-peripherals.pdf p.81 “5.4 General Purpose GPIO Clock”

$^6$https://datasheets.raspberrypi.org/bcm2711/bcm2711-peripherals.pdf p.82 “5.4.1 Operating Frequency”
Fractional PLL

Many implementations derived from the original PiFM demonstration\(^7\):

- [https://github.com/ChristopheJacquet/PiFmRds](https://github.com/ChristopheJacquet/PiFmRds) is easiest\(^8\) to understand
- GPIO4 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](https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0.pdf)
- A much more general (and complex\(^9\)) implementation is available at [github.com/F5OEO/rpitx](https://github.com/F5OEO/rpitx) relying on [github.com/F5OEO/librpitx](https://github.com/F5OEO/librpitx)
- Interfacing the latter with GNU Radio \{I,Q\} stream is explained at [https://github.com/ha7ilm/rpitx-app-note](https://github.com/ha7ilm/rpitx-app-note)
- [http://abyz.me.uk/rpi/pigpio/pigs.html](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 (FSCW)


\(^8\) [github.com/ChristopheJacquet/PiFmRds/blob/master/src/pi_fm_rds.c#L534](https://github.com/ChristopheJacquet/PiFmRds/blob/master/src/pi_fm_rds.c#L534)

\(^9\) E. Courjaud *Rpitx: Raspberry Pi SDR transmitter for the masses*, SDRA (2017) at [https://www.youtube.com/watch?v=Jku4i8t_nPc](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
SAW resonator characterization

The SAW device is connected in parallel to the antenna ⇒ make sure to disconnect the antenna (MCX connector) when measuring the SAW device to avoid radiating the probe signal and collecting environmental noise!

Example of radiated spectrum when seeping around the 434 MHz ISM band:
gr-rpitx general purpose sink

- Wrapping librpitch as a general purpose GNU Radio Sink
- Although targeted to the Raspberry Pi 4, must also be compiled on the host to be accessible from GNU Radio Companion
- datarate: 10000-250000 Hz so that fourth overtone will span 1 MHz
- compiles and tested on 3.8 and 3.9, compiles with 3.10:
  1. git clone https://github.com/jmfriedt/gr-rpitx/
  2. cd gr-rpitx
  3. mkdir build && cd build
  4. cmake ../ && make
  5. sudo make install
  6. reload processing block list in GNU Radio Companion

Using the Raspberry Pi PLL as radiofrequency source controlled from GNU Radio.

Compiling for the Raspberry Pi target

Easiest: gr-rpitx is available for GNU Radio 3.8 as a Buildroot BR2_EXTERNAL package at https://github.com/oscimp/oscimp_br2_external and can be selected from the External options. For GNU Radio 3.9, switch from main to gr39 branch and compile manually as described below, assuming librpitch has been compiled with Buildroot.

We assume librpitch to have been compiled and installed, most probably on the cross-compilation framework Buildroot as described at https://github.com/oscimp /oscimp_br2_external (tested with Buildroot 2020.11.1 and above). The benefits of using...
Consistency issue between GNU Radio version (host v.s target)

Consistency issues between flowgraph generated on PC and run on the Raspberry Pi:

- Debian/stretch (oldoldstable): 3.7.10.1; buster (oldstable): 3.7.13.4;
  bullseye (stable): 3.8.2.0; bookworm (testing) and sid (unstable): 3.10.1.1
- Ubuntu/bionic (18.04LTS): 3.7.11; focal (20.04LTS): 3.8.1.0;
  hirsute (21.04) and impish (21.10): 3.8.2.0; jammy: 3.9.4.0

For GNU Radio 3.8 as provided by the official Buildroot release:
  - PyBOMBS at https://github.com/gnuradio/pybombs produces with the gnuradio-default recipe generates a local copy of GNU Radio 3.8 (source setup.env.sh after completion to use this version)

For GNU Radio 3.9 as provided by most current binary distributions:
  - use the gnuradio39 recipe: pybombs prefix init pybombs-3.9 -R gnuradio39

after completing GNU Radio PyBOMBS compilation:
  - in the PyBOMBS directory: source setup.env.sh
  - pybombs install gr-osmosdr

- a temporary Buildroot is available from the BR2_EXTERNAL at https://github.com/oscimp/oscimp_br2_external/ (see package/gnuradio39 for the recipe)
  - GNU Radio 3.9 will never be part of Buildroot, shifting straight to GNU Radio 3.10

10 watch for conflicting libuhd function prototypes: https://github.com/gnuradio/pybombs/issues/612
11 in case of error with gr-fcdproplus, remove this entry from $HOME/.pybombs/recipes/gr-recipes/gr-osmosdr.lwr since it is not needed.
Initialize the ZeroMQ socket, receive a byte array and convert to the appropriate format:

```python
import numpy as np
import zmq
import array

Nt=1024
context=zmq.Context()
sock1=context.socket(zmq.SUB)
sock1.connect("tcp://127.0.0.1:5556"); # replace with RPi4 IP and port
sock1.setsockopt(zmq.SUBSCRIBE, b"")
vector1=[]
while (len(vector1)<Nt):
    raw_recv=sock1.recv()
    # recv=array.array('f',raw_recv) # float
    recv=array.array('l',raw_recv) # integer
    print(recv)
```
Conclusion

PiFM is provided as Buildroot external package (BR2_EXTERNAL) at https://github.com/oscimp/oscimp_br2_external/rpitz has been converted to a library compatible with a GNU Radio sink: https://github.com/jmfriedt/gr-rpitz

1. Compile PiFM and its dependencies for the RPi4
2. Generate signals and check that their spectra are consistent with expectations (tone, chirp, FM)
3. Listen to the generated signals
4. Control the emitted signal, in addition to the received signal, from the Python server

Do not add a wire to GPIO4: the pin will radiate a low enough power to be detected by the DVB-T receiver without bothering neighbours

Conclusion: characterize the SAW resonator transfer function, either using a noise generator or a frequency sweep, either using RPiTX (standalone program) or gr-rpitz

12 main branch for GNU Radio 3.8, gr39 branch for GNU Radio 3.9
Executing a processing flowchart when inserting the RTL-SDR dongle

1. flowchart runs to completion (*not* prompt)
2. remove dvb-usb-rtl28xxu.ko and dvb_usb_v2.ko
3. add in `/etc/udev/rules.d/rtl-sdr.rules`:

   ```
   SUBSYSTEMS=="usb", ATTRS{idVendor}=="0bda", ATTRS{idProduct}=="2838", \
   ENV{ID_SOFTWARE_RADIO}=="1", MODE="0660", GROUP="plugdev", "\n   RUN="/root/rpi_fm_auto.py"
   ```

   with `/root/rpi_fm_auto.py` the GNU Radio flowchart to be executed.

Take care that GNU Radio Companion flowcharts are *not* backward-compatible.