PROJECT REPORT

ON

Software-Defined Radio Hacking

Submitted by

- Yanis Alim

Directed by :

- Mr. Bruce Denby

SORBONNE UNIVERSITÉ

1- False Data Injection Attack on a weather station with URH & 433Utils

Input :

1) We are provided with 5 different files. Each one represents a captured frame of a temperature probe at different degrees.

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet$ tree FDIA/
FDIA/
├── New RTL-SDR-20190926_233403-433_900MHz-1MSps-1MHz_273C.complex
├── New RTL-SDR-20190927_013114-433_900MHz-1MSps-1MHz197.complex
├── New RTL-SDR-20191001_193432-433_920MHz-1MSps-1MHz_7_2.complex
├── New RTL-SDR-20191001_194432-433_920MHz-1MSps-1MHz_3_3.complex
└── New RTL-SDR-20191001_205026-433_920MHz-1MSps-1MHz_1_4.complex
```

2) In addition, we have a short description :

We are studying Low Power Devices (LPD), such as remote controls, weather stations, etc.

- Operating at 433 MHz or 868 MHz
- In uplink as well as in downlink
- In OOK and GFSK modulation

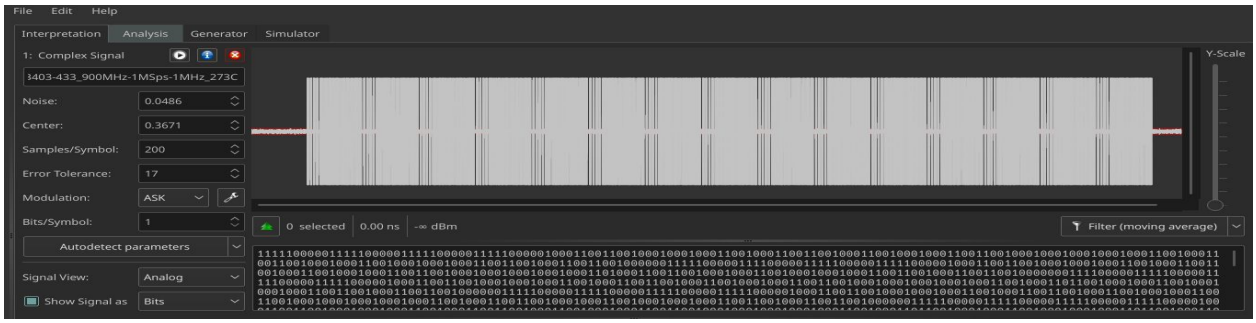
Output :

We are asked to :

- Reverse engineer the protocol of the temperature probe
- Demodulate the signal
- Send a false temperature

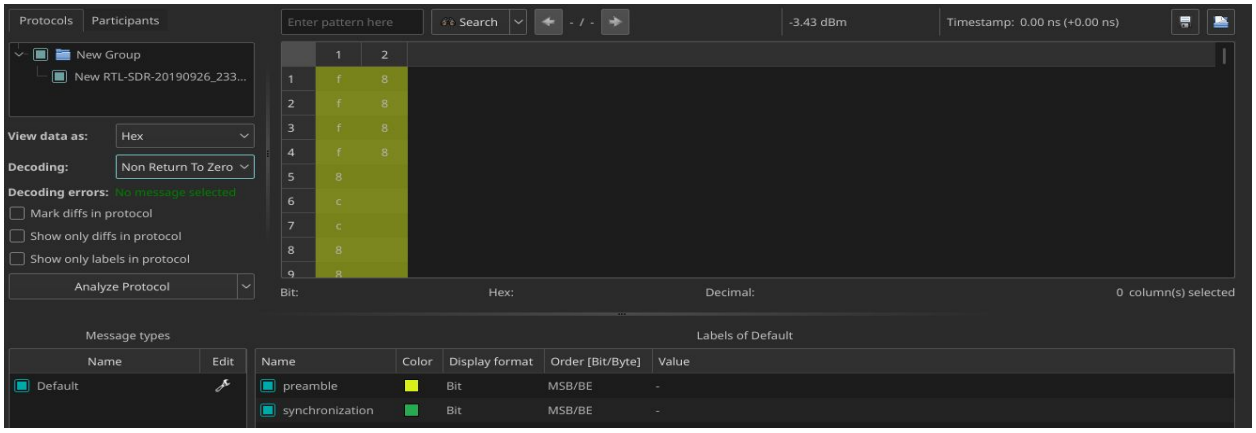
Reverse engineering the protocol

We start by loading the file in the *Universal Hacker Radio* (URH) tool that helps us see a big picture of the signal :



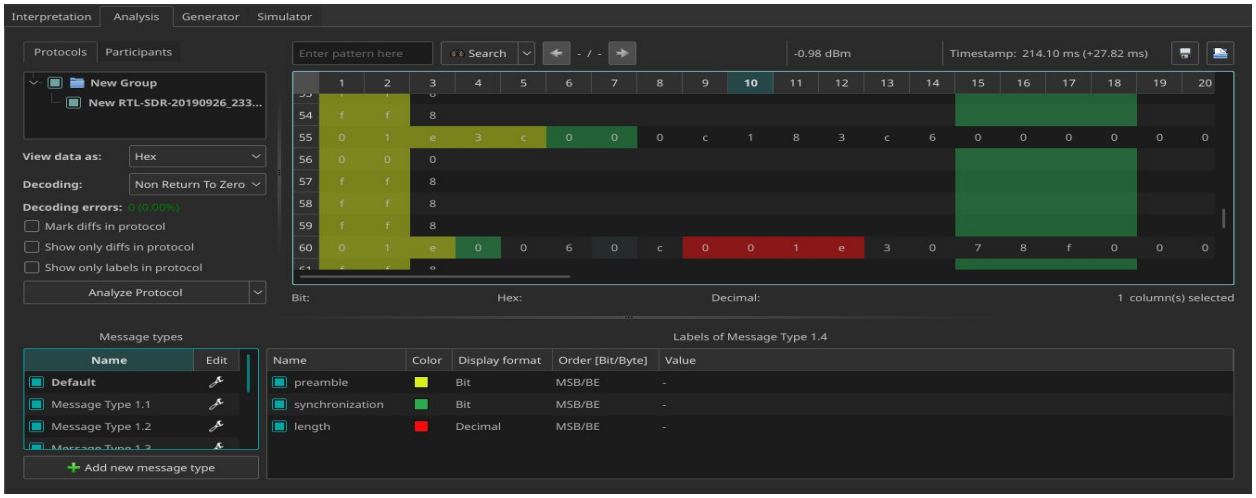
That doesn't give us much information about the signal...
First thing I tried is "autodetect parameters" option of urh with different modulations

1. **ASK :**



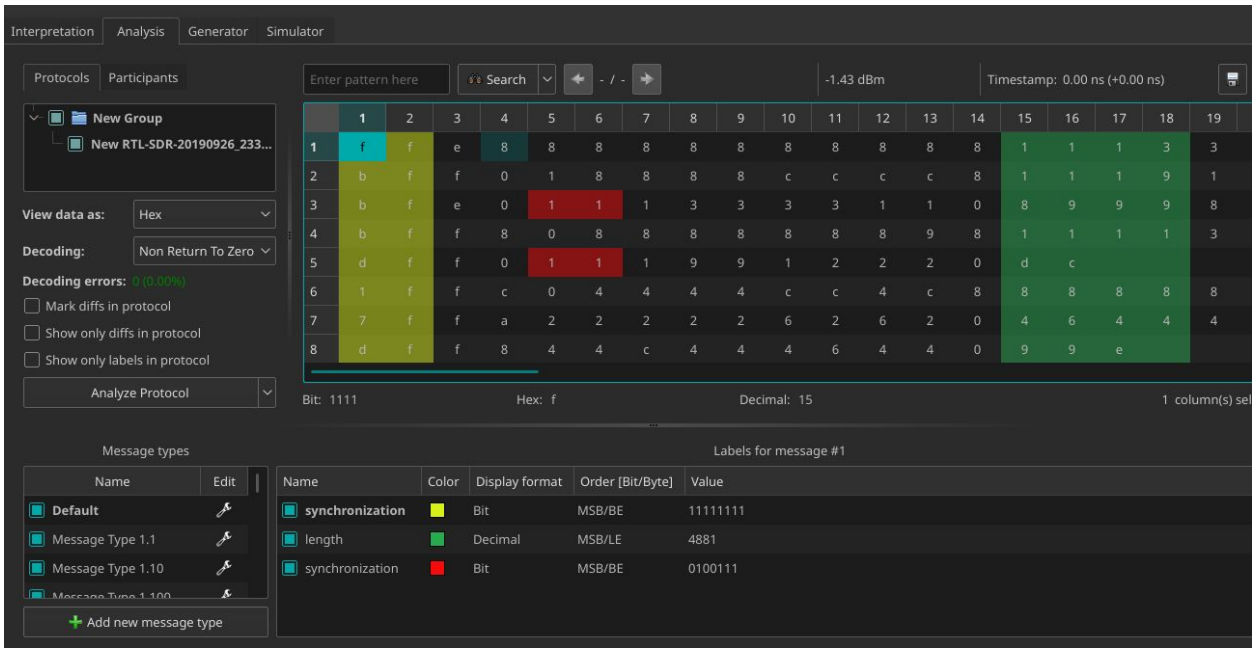
We can consider this as a fail because we can see that it detects that there is only a preamble and a synchronization bytes on the frame. We know that there is a temperature data.

2. **PSK :**



The result is different from the previous one: we have a length byte in addition but still no data bytes.

3. FSK :

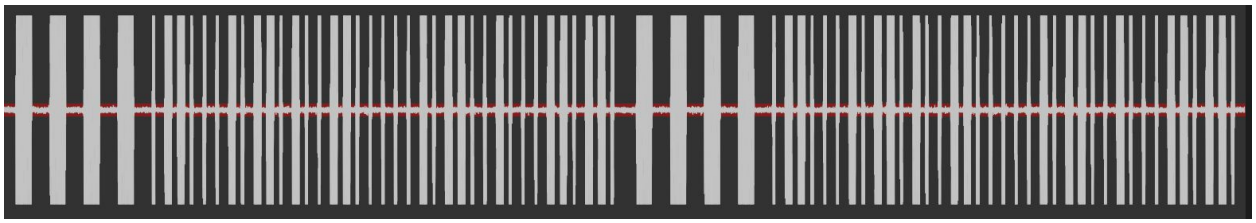


Just similar to the previous one. No data found.

I guess we have to manually reverse this protocol...

Manual analysis:

Let's take a step back and see if we can find something interesting :



Doesn't that look like a repeated pattern ?
I loaded the other files and checked if we have the same pattern and it was the case :
A pattern is repeated 15 times !
Reversing the protocol was the hard part of all the tasks.

Conclusion :

There was random looking data followed by a consistent pattern followed by a series of wide and short pulses.

Eventually I figured out the random pulses at the start of data must be for radio synchronization between the transmitter and receiver.

None of the “random” data at the start of the bit stream was consistent between any runs and I ended up simply chopping it off and ignoring it in the data stream.

After the random bits there is a low pulse of varying length followed by 4 data sync pulses. The data sync pulses are 0.95 msec.

Immediately after the 4 data sync pulses are 37 data bit pulses. Each data bit pulse is ~0.72 msec long.

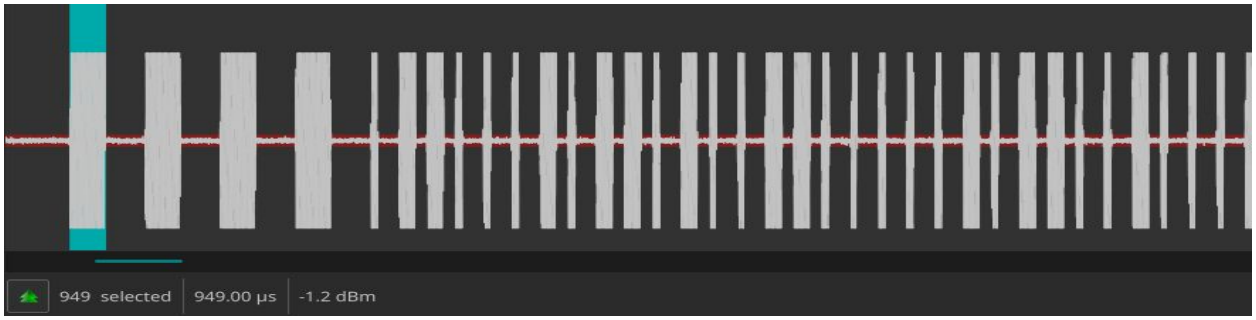
A logic high (1) bit is encoded as a 0.45 msec high pulse followed by a 0.27 msec low pulse.

A logic low (0) bit is encoded as a 0.21 msec high followed by a 0.51 msec low.

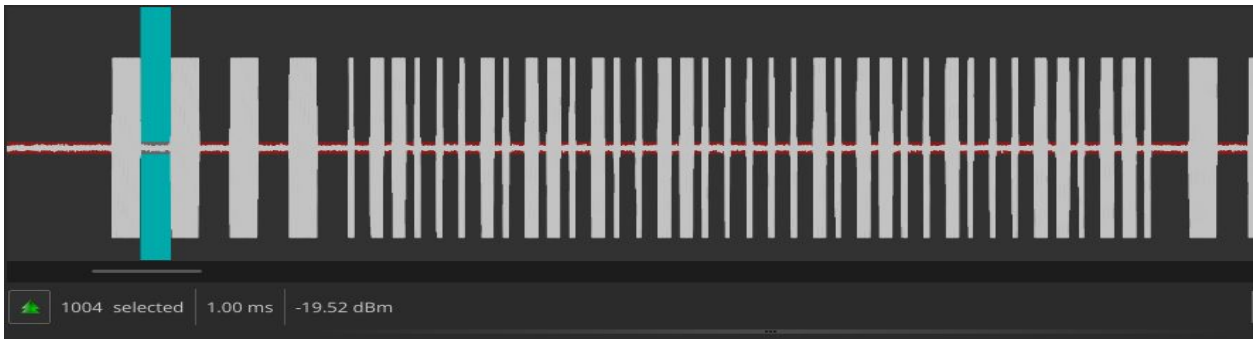
The modulation looks like **Pulse Width Modulation (PWM)**

URH :

The data synchronization pulses have a duration of 0.95 ms:

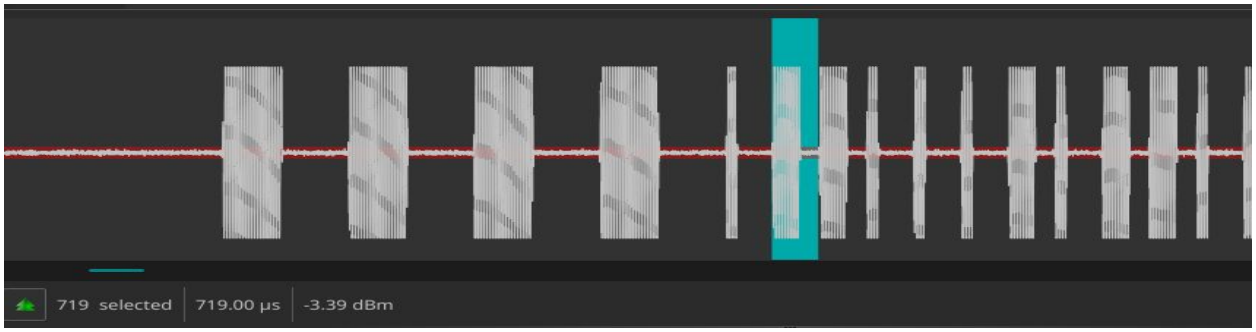


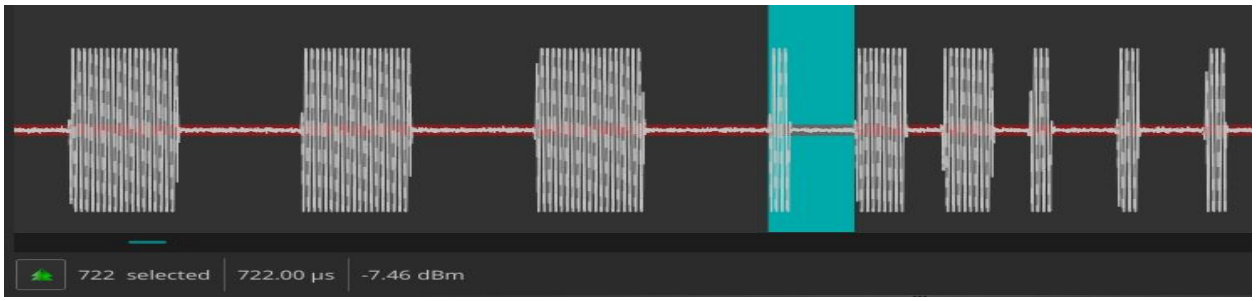
Then there is a 1ms of silence ;



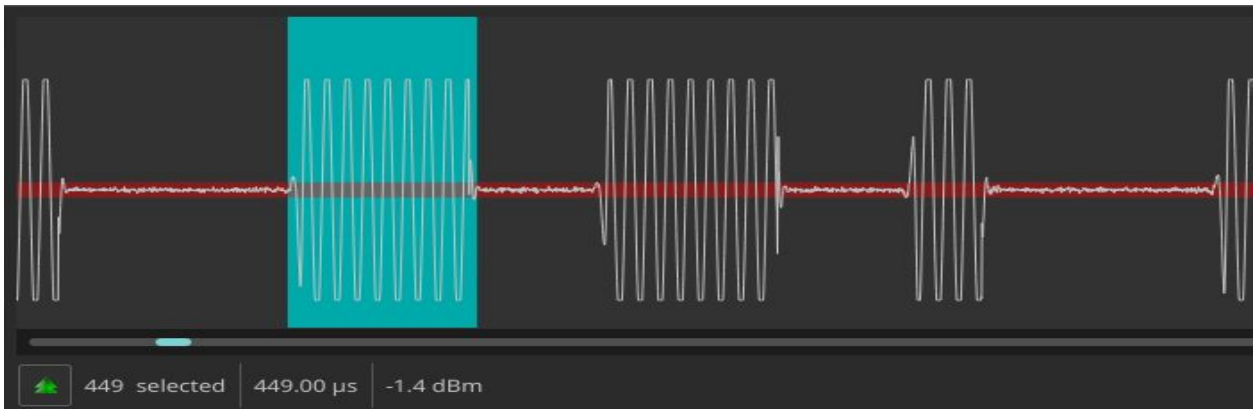
Immediately after the 4 data synchronization pulses, there are 37 data bit pulses :

Each data bit pulse lasts approximately 0.72 ms:

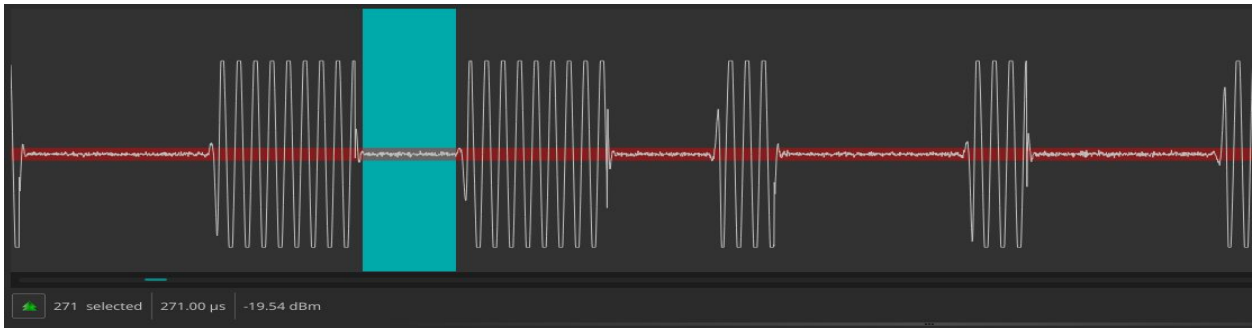




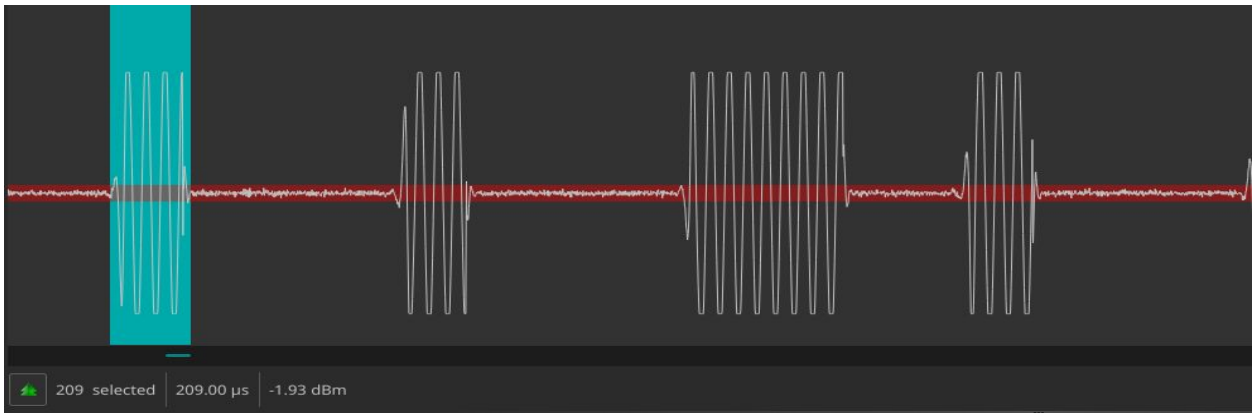
- A logic high (1) bit is encoded as a 0.45 msec high pulse :



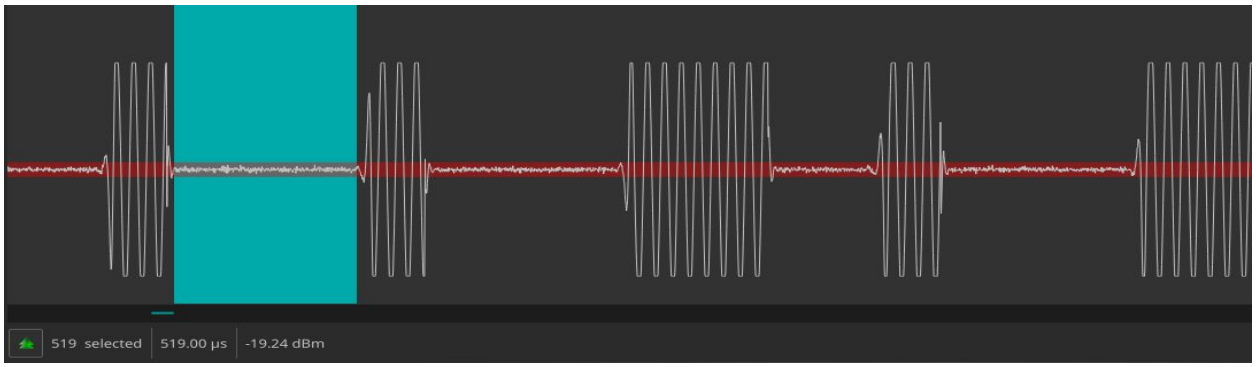
followed by a 0.27 msec low pulse :



- A logic low (0) bit is encoded as a 0.21 msec high:

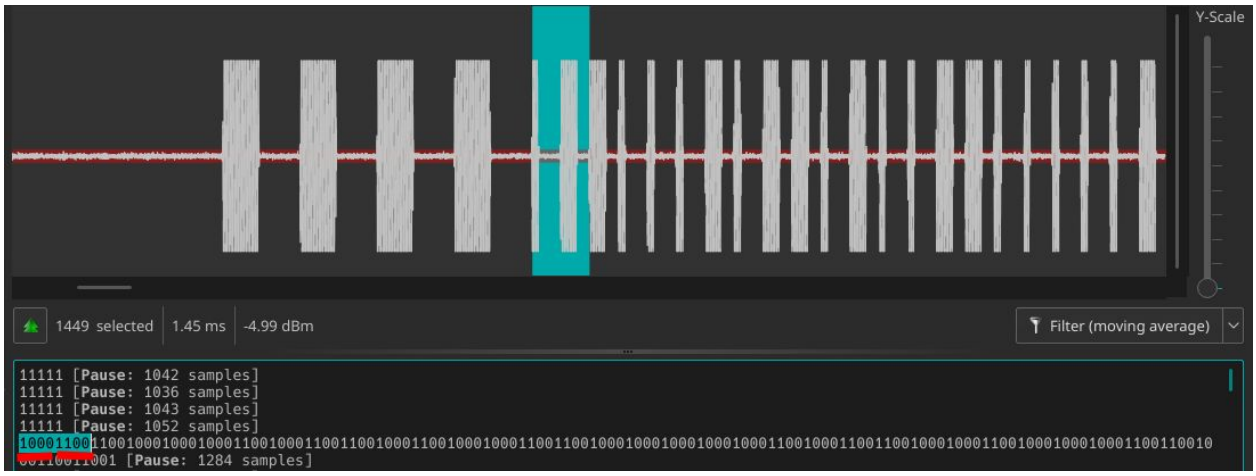


followed by a 0.51 msec low :



Now, it's time to feed URH a good input so it can decode it for us :

Let's select both type of pulses :

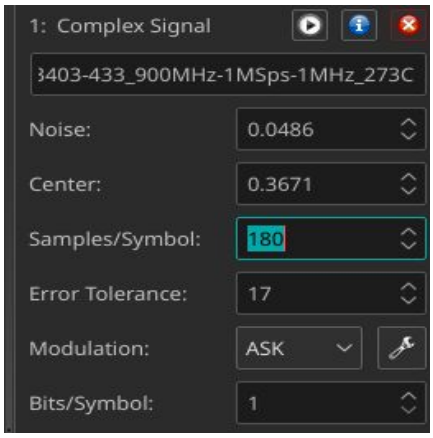


We notice that for the :

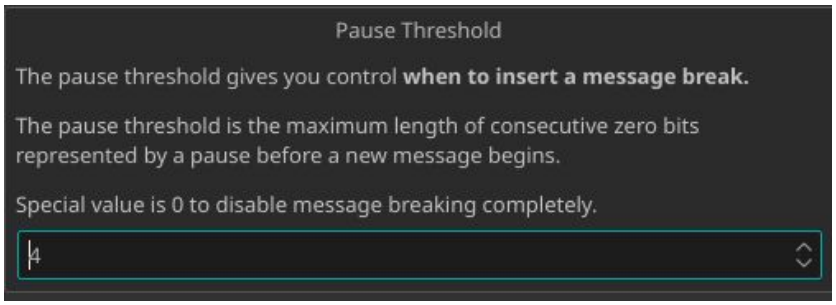
- ***Logic low (0) bit is : 1000***
- ***Logic high (1) bit is : 1100***

Each pulse is about 0.72ms, we have 4 bits for each one :

That makes it 0.18ms for a bit.
Let's add this to URH Sample/Symbol input :

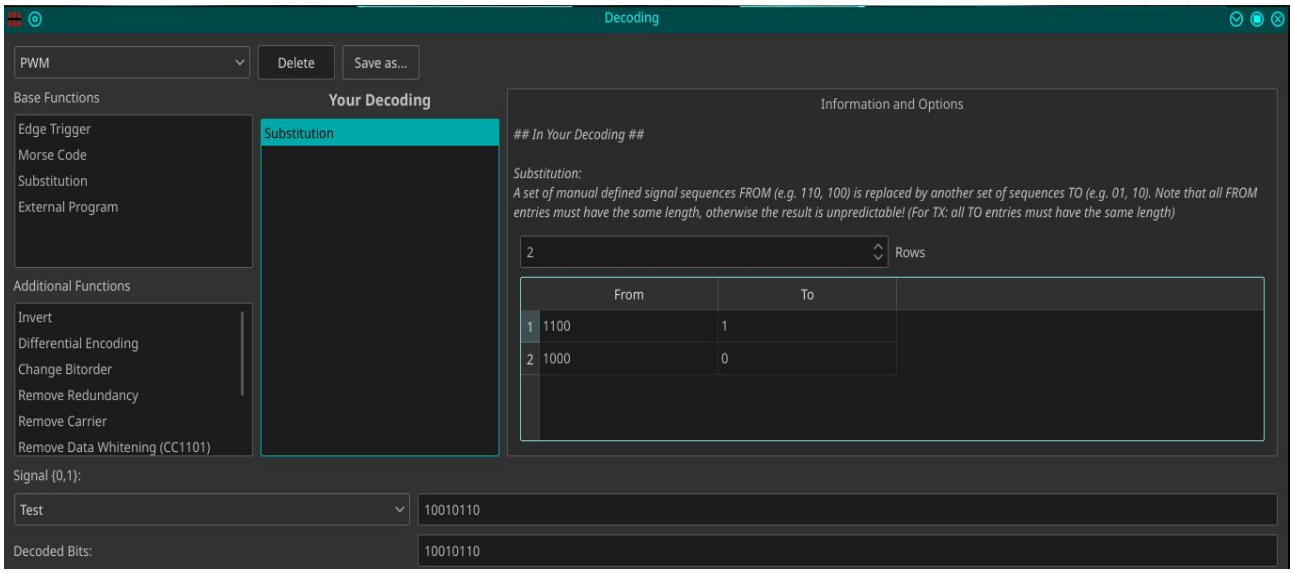


We can also add the value 0000 as silence in the advance parameters :

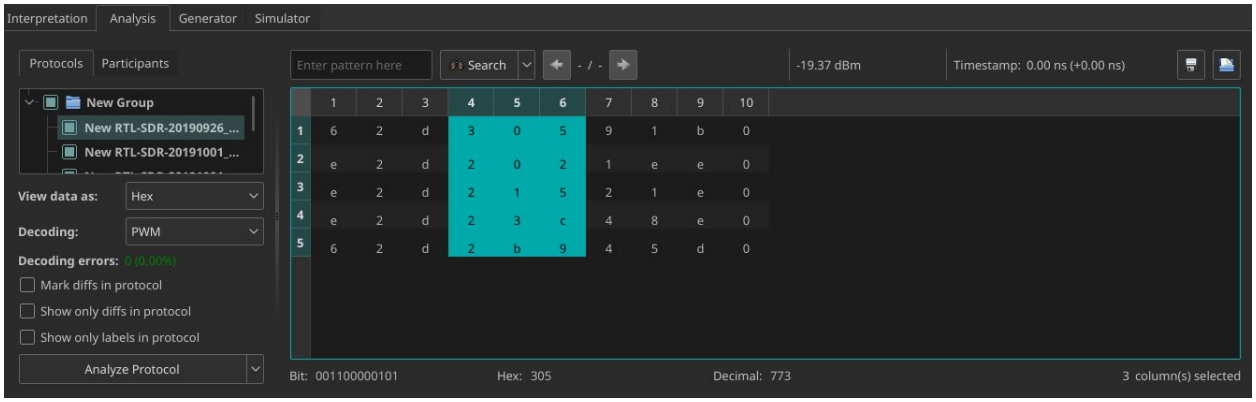


Last step is to substitute :

- **1100 -> 1**
- **1000 -> 0**



We add our Pulse Width Modulation, then we crop one data frame from 5 samples :



That looks promising.

The temperature is surely not represented in the floating representation since it takes more bytes than a normal representation.
I googled how temperature probes encode the data. And i found they do mathematical operations so they remove the float representation and one of the simplest things to do is to multiply it by a power of ten :

For example : 27.3 would be 27.3 * 10^1 = 273

To figure out what has been done in out case, we need to load more files and compare between them :

27.3 => 62d30591b

19.7 => 62d2b945d

7.2 => e2d23c48e

3.3 => e2d21521e

1.4 => e2d2021ee

We pick the 3 last temperatures because they have the same 3 first bytes.
If we consider the last 2 bytes as a checksum (it's the case in a lot of protocol),

We will be left with :

23C

215

202

The difference between the 1st and 2nd temperature is ***0x27 (39)***

And between the 2nd and 3rd is ***0x13 (19)***

Exactly as expected because:

72 - 33 = 39

33 - 14 = 19

So, they did multiply the temperature by 10 to avoid the floating representation but there is something else : ***0x23C is not 77, 0 x215 is not 33, 0x202 is not 14***

Quick math :

0x23C = 572

0x215 = 533

0x202 = 514

That's obvious... They added 500 to each temperature

Let's verify for the 1st and 2nd temperature :

27.3 * 10 + 500 = 773 = 0x305

19.7 * 10 + 500 = 697 = 0x2b9

Problem solved !

Encoding : Temperature * 10 + 500

Decoding : Temperature - 500 / 10

Sending false data to the station

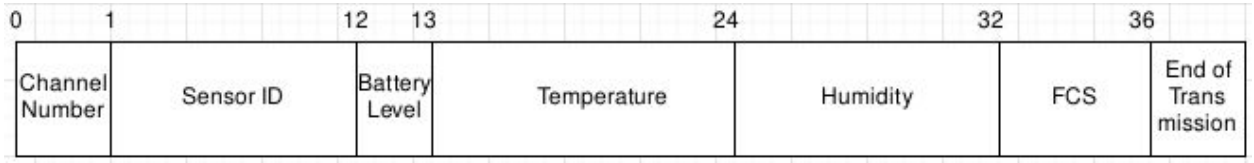
To be able to send temperature, we need to know each field of the packet.
To do that we need to have a binary representation of all the temperatures and compare between them :

27.3	0	1	1	0	0	0	1	0	1	1	0	1	0	0	1	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	0
19.7	0	1	1	0	0	0	1	0	1	1	0	1	0	0	1	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1	0	1	1	1	0	1	0
7.2	1	1	1	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	1	1	1	1	0	0	0	1	0	0	1	0	0	0	1	1	1	0	0
3.3	1	1	1	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	0	1	0	1	0	1	0	0	1	0	0	0	0	1	1	1	1	0	0
1.4	1	1	1	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	1	1	1	0	0

After some comparison between the frames and checking temperature sensor's protocol on the internet (http://wmrx00.sourceforge.net/Arduino/OregonScientific-RF-Protocols.pdf), i reconstructed the most probable frame structure : 110011111011

27.3	0	1	1	0	0	0	1	0	1	1	0	1	0	0	1	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	0
19.7	0	1	1	0	0	0	1	0	1	1	0	1	0	0	1	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1	0	1	1	1	0	1	0
7.2	1	1	1	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	1	1	1	1	0	0	0	1	0	0	1	0	0	0	1	1	1	0	0
3.3	1	1	1	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	0	1	0	1	0	1	0	0	1	0	0	0	0	1	1	1	1	0	0
1.4	1	1	1	0	0	0	1	0	1	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	1	1	1	0	0

The structure should be :



- Channel ID (1bit) : There are 2 channels :

  - 0 : for high temperatures

  - 1 : for low temperatures

- Sensor ID (11bits) : the sensor identifier which is always fixed at 11000101101 (0x62D)
- Battery indicator (1bit) : Indicates the level of the sensor's battery :

  - 0 : High

  - 1 : Low

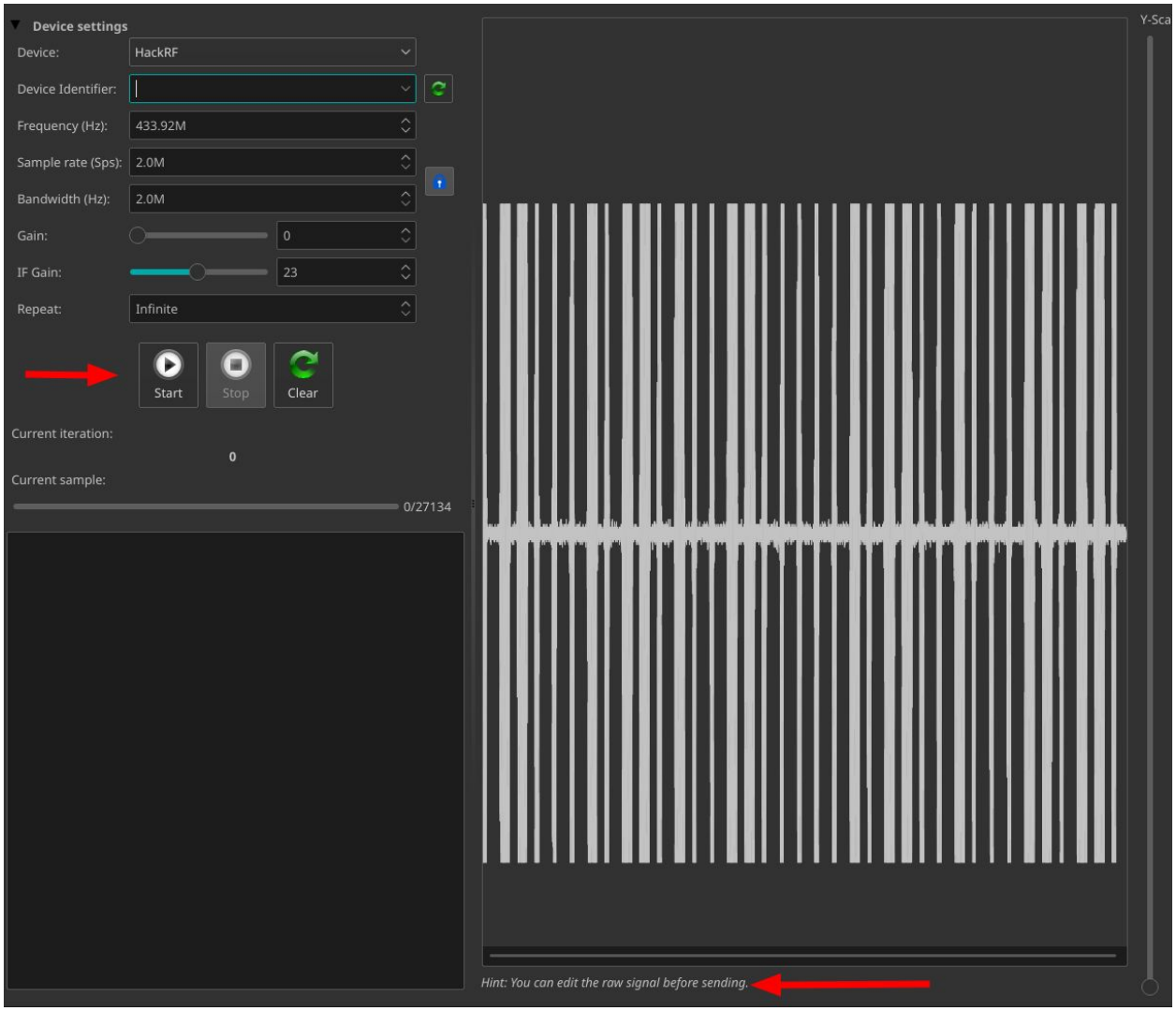
- Temperature (11bits): indicates the actual temperature encoded
- Humidity (8bits) : indicates the actual humidity percentage in BCD format :
  - For 27.3° C: 10010001 (0x91) : 91%
  - For 19.7° C: 01000101 (0x43) : 43%
  - For 7.2° C: 01001000 (0x48): 48%
  - For 3.3° C: 00100001 (0x21): 21%
  - For 1.4° C: 00011100 (0x19): 19%

- FCS (4bits) : Frame checksum, calculated using this formula :

$$\mathrm{FCS_H} = 0\mathrm{xf} - (\sum_{i=0}^{7} \mathrm{nibble}_i) \;\&\; 0\mathrm{xf}$$

- End Of Transmission (1bit) : a fixed bit to the value of 0, indicates the end of transmission

Now, to send any false data to the weather station, we can just edit a frame, recalculate the FCS and play it in URH :



2- Demodulation of an FM radio station with URH

Input :

1) We are provided with 3 different files that represent several seconds of music from 3 Parisian stations whose frequency appears in the file name.

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet$ tree RadioFM/
RadioFM/
├── RTL-SDR-20191001_174535-105_100MHz-1MSps-1MHz.complex
├── RTL-SDR-20191001_212358-106_700MHz-1MSps-1MHz.complex
└── RTL-SDR-20191001_212636-104_300MHz-1MSps-1MHz.complex
```

2) In addition, we have a short description :

- Operating in the band 88 - 108 MHz
- At downlink only
- in Analog Frequency Modulation

Output :

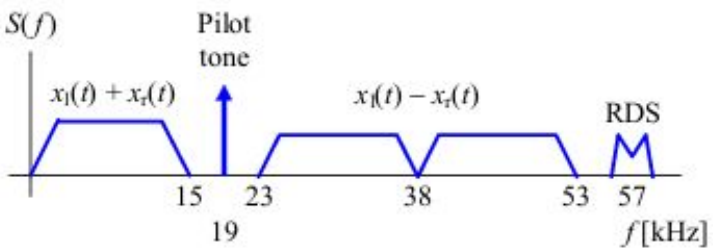
We are asked to :

- Demodulate the signal to recover the audio signal and listen to it.

FM Receiver

FM broadcast signals are transmitted in the frequency range of 87 MHz to 108 MHz. The baseband signal is the audio signal limited to a bandwidth of 15 kHz.

Stereo transmission uses a multiplex scheme, where the sum of the left and the right channel, ***xL(t) + xR(t)***, is transmitted in the lower frequency range and the difference of the stereo channels is modulated using DSB-SC (double-side band suppressed carrier) onto a 38-kHz carrier



This multiplex scheme is compatible with monaural receivers which use the ***xL(t) + xR(t)*** signal only.

A pilot tone with frequency 38 kHz/2 = 19 kHz indicates the stereo signal to the receiver and is used for coherent demodulation of the DSB signal. As shown above, the RDS signal is inserted above the stereo multiplex using a subcarrier of 57 kHz. The baseband signal is FM-modulated onto a carrier using a frequency deviation of fΔ = 75 kHz.

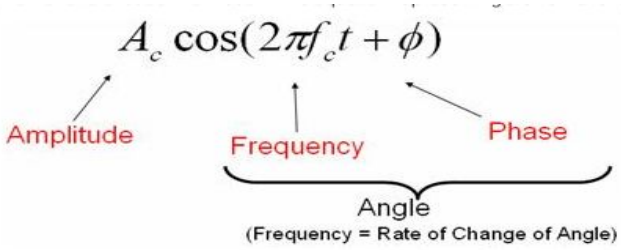
The bandwidth of the modulated signal is

$$B \approx 2(f_\Delta + 2f_g),$$

where Fg ≈ 60 kHz is the upper frequency limit of the baseband signal. Thus the FM signal has a bandwidth of 390 kHz.

Background

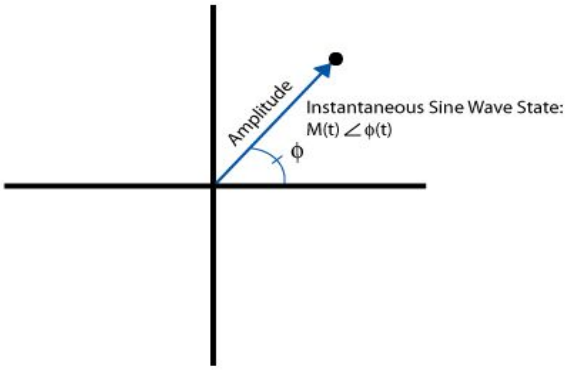
Signal modulation changes a sine wave to encode information. The equation representing a sine wave is as follows:



The equation above shows that we are limited to making changes to the amplitude, frequency, and phase of a sine wave to encode information.

Frequency is simply the rate of change of the phase of a sine wave (frequency is the first derivative of phase), so frequency and phase of the sine wave equation can be collectively referred to as the phase angle.

Therefore, we can represent the instantaneous state of a sine wave with a vector in the complex plane using amplitude (magnitude) and phase coordinates in a polar coordinate system.



In the graphic above, the distance from the origin to the black point represents the amplitude (magnitude) of the sine wave, and the angle from the horizontal axis to the line represents the phase.

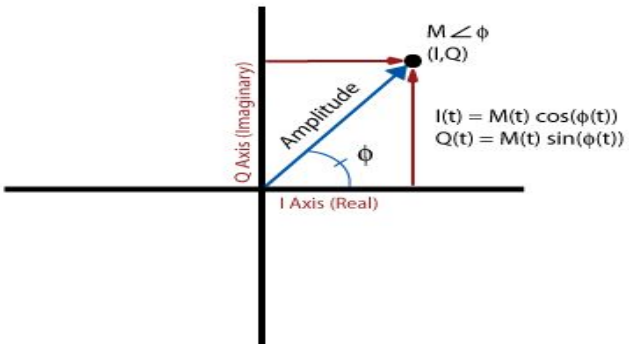
Thus, the distance from the origin to the point remains the same as long as the amplitude of the sine wave is not changing (modulating). The phase of the point changes according to the current state of the sine wave.

For example, a sine wave with a frequency of 1 Hz (2π radians/second) rotates counter-clockwise around the origin at a rate of one revolution per second. If the amplitude doesn't change during one revolution, the dot maps out a circle around the origin with radius equal to the amplitude along which the point travels at a rate of one cycle per second.

IQ Modulation

All the concepts discussed above apply to I/Q data. In fact, I/Q data is merely a translation of amplitude and phase data from a polar coordinate system to a Cartesian (X,Y) coordinate system.

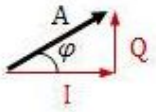
Using trigonometry, we can convert the polar coordinate sine wave information into Cartesian I/Q sine wave data. These two representations are equivalent and contain the same information, just in different forms :



I/Q Data consists of I and Q represented as two separate variables, a vector of length two, or more often, the complex number I + Q*i* .

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

$$A\cos(2\pi f_c t + \varphi) = A\cos(2\pi f_c t)\cos(\varphi) - A\sin(2\pi f_c t)\sin(\varphi)$$



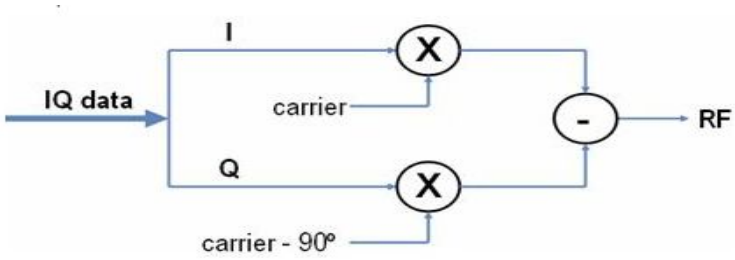
$$I = A\cos(\varphi)$$
$$Q = A\sin(\varphi)$$

$$A\cos(2\pi f_c t + \varphi) = I\cos(2\pi f_c t) - Q\sin(2\pi f_c t)$$

where I is the amplitude of the in-phase carrier
Q is the amplitude of the quadrature-phase carrier

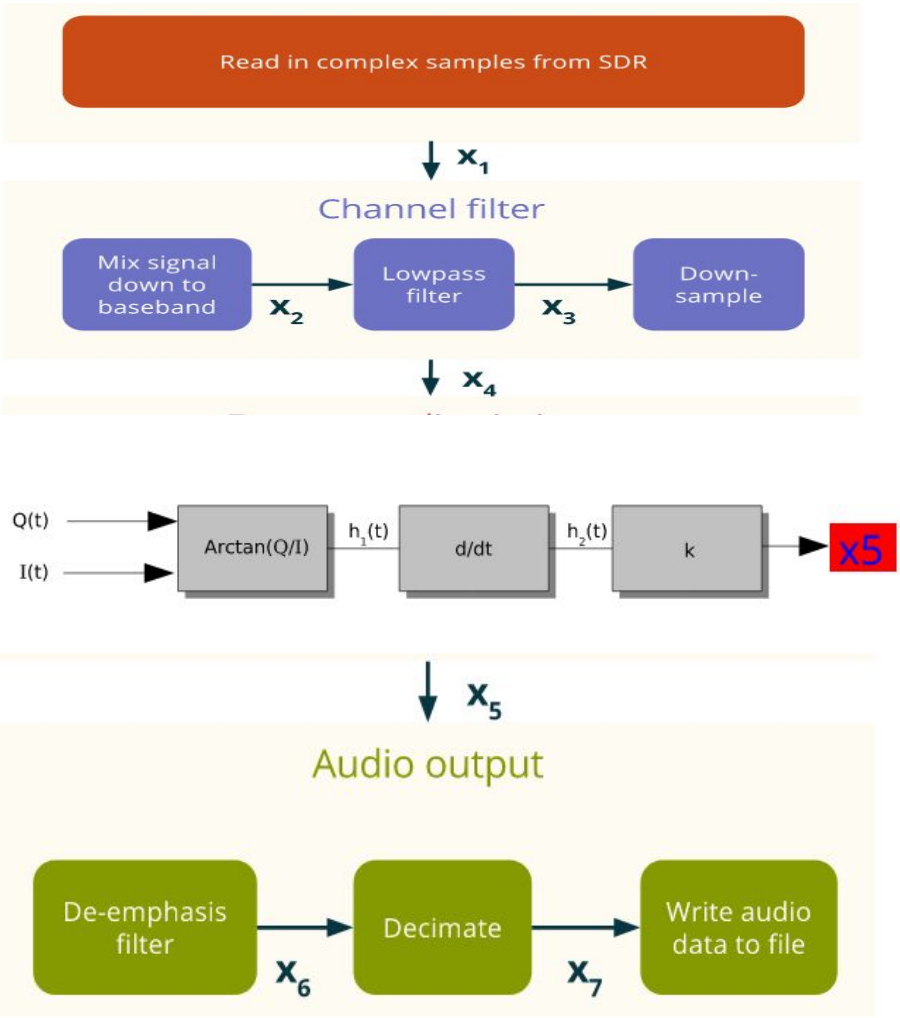
We know that the difference between a sine wave and a cosine wave of the same frequency is a 90-degree phase offset between them.

Essentially, what this fact means is that you can control the amplitude, frequency, and phase of a modulating carrier sine wave by simply manipulating the amplitudes of separate I and Q input signals. With this method, you do not need to directly vary the phase of an RF carrier sine wave.



Demodulation algorithm

The demodulation algorithm is accessed through the IO system which is marked on the figure below :



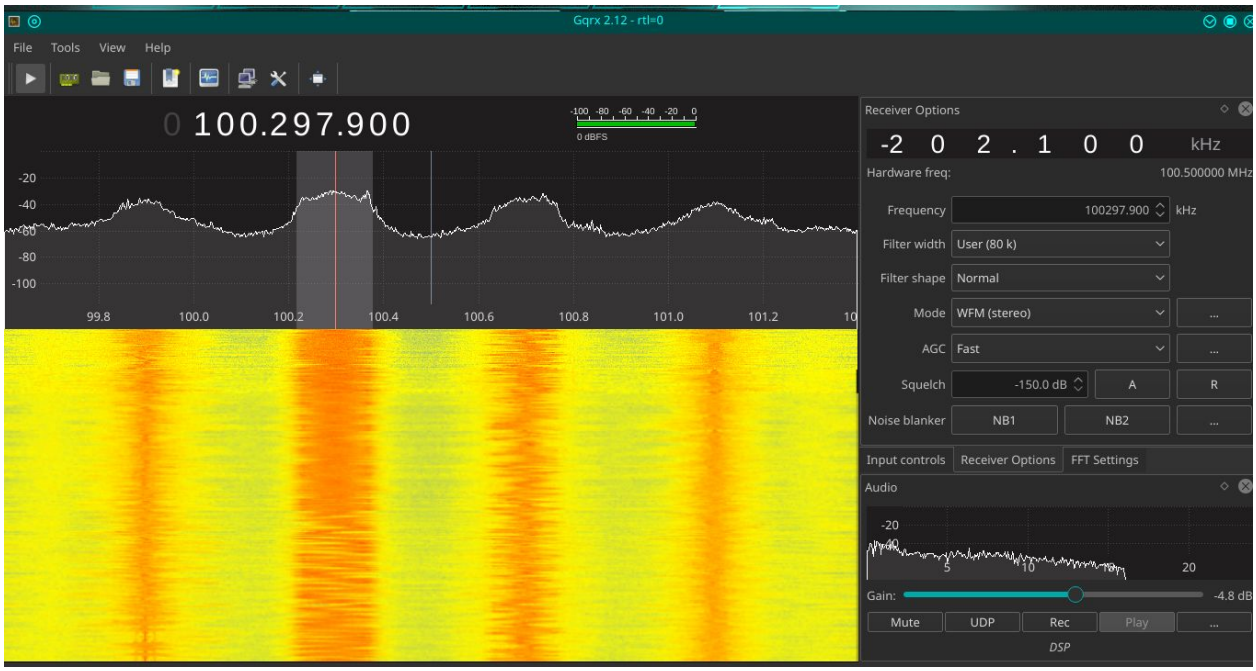
RF to IQ samples :

The first goal is to identify an interesting frequency band to capture. We 'll capture some of the FM broadcast bands. To do this I open GQRX by typing the following in the Terminal:

$ gqrx

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/RadioFM$ gqrx
Controlport disabled
No user supplied config file. Using "default.conf"
gr-osmosdr 0.2.0.0 (0.2.0) gnuradio 3.8.1.0
built-in source types: file osmosdr fcd rtl rtl_tcp uhd miri hackrf bladerf rfspace airspy airspyhf soapy redpitaya freesrp
gr::log :WARN: file_source0 - file size is not a multiple of item size
FM demod gain: 3.05577
Resampling audio 96000 -> 48000
IQ DCR alpha: 1.04166e-05
Using audio backend: auto
BookmarksFile is /home/Yan1x0s/.config/gqrx/bookmarks.csv
[INFO] [UHD] linux; GNU C++ version 9.2.1 20200224; Boost_106700; UHD_3.15.0.0-2+b1
```

We setup the receiver at 100Mhz in FM Mode :



Now I use the rtl-sdr IQ recorder utility to capture some data. First I close GQRX to free up the RTL dongle. Next, to see the options and default settings for the rtl-sdr utility, I type the following at the Terminal window:

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/RadioFM$ rtl_sdr
rtl_sdr, an I/Q recorder for RTL2832 based DVB-T receivers

Usage:   -f frequency_to_tune_to [Hz]
        [-s samplerate (default: 2048000 Hz)]
        [-d device_index (default: 0)]
        [-g gain (default: 0 for auto)]
        [-p ppm_error (default: 0)]
        [-b output_block_size (default: 16 * 16384)]
        [-n number of samples to read (default: 0, infinite)]
        [-S force sync output (default: async)]
        filename (a '-' dumps samples to stdout)
```

Now, suppose I want to capture 10 seconds worth of data, with a center frequency of 100.297 MHz and a gain of 40 dB. If I set the sample rate to 1000000 samples/sec (which, equates to 2.5 MHz bandwidth) then I will need to capture 10000000 samples (100000000 [samples]/ (1000000 [samples/sec]) = 10 [sec]), and I type the following command at the Terminal:

$ rtl_sdr -f 100297000 -g 40 -s 1000000 -n 10000000 FMcapture1.complex

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/RadioFM$ rtl_sdr -f 100297000 -g 40 -s 1000000 -n 10000000 FMcapture1.complex
Found 1 device(s):
  0:  Realtek, RTL2838UHIDIR, SN: 00000001

Using device 0: Generic RTL2832U OEM
Found Rafael Micro R820T tuner
Exact sample rate is: 1000000.026491 Hz
[R82XX] PLL not locked!
Sampling at 1000000 S/s.
Tuned to 100297000 Hz.
Tuner gain set to 40.20 dB.
Reading samples in async mode...

User cancel, exiting...
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/RadioFM$ ls
FMcapture1.complex                RTL-SDR-20191001_212358-106_700MHz-1MSps-1MHz_stradfilt.wav
```

The raw, captured IQ data is saved in a file called FMcapture1.complex, which is stored in my home directory by default.

Explaining the code

First thing first

We start by reading the file :

- As a 64bit complex type [I + iQ , I + iQ , ...]
- As a 32bit float type [I, Q, I, Q, ...]

Second one is easier because it's matter of position to get I and Q data separately :

I data has an even position and Q has an odd position

```
# Read input complex file
c = np.fromfile(fpath, np.float32)

#if input file more than about 190 Mb the program never finishes (memory problem?)
I = c[0:min(len(c),23752362):2]
Q = c[1:min(len(c),23752362):2]
```

The purpose of the downconverter is to downconvert the received FM signal to a frequency range that can be sampled using a sound card.

```
Idown = I[0::downer]
Qdown = Q[0::downer]
```

The signal at hand are :

$$\begin{aligned} I(t) &= A\cos(\phi(t)) \\ Q(t) &= A\sin(\phi(t)) \end{aligned}$$

The proof of this method to derive the message signal m(t), is carried out in three steps referring to the blocks of the next figure.

$$
\begin{aligned}
\phi(t) &= 2\pi k_f \int_0^t m(\tau)d\tau \\
\phi'(t) &= 2\pi k_f m(t) \\
h_1(t) &= \tan^{-1}\left(\frac{Q(t)}{I(t)}\right) = \phi(t) \\
h_2(t) &= \frac{d}{dt}\left[\phi(t)\right] \\
&= 2\pi k_f m(t) \\
h_3(t) &= \frac{1}{2\pi k_f} \cdot 2\pi k_f m(t) = m(t)
\end{aligned}
$$

We derive the instantaneous phase of the received FM signal by deriving the angle of the complex number ***I(t) + i Q(t)*** as depicted by the equation :

```
unwarp_atan2 = np.unwrap(atan2(Qdown, Idown))
```

Note: why ***unwrap*** is used ?

The arcus tangent is implemented using the function atan2() which is a four quadrant arcus tangent. The problem with using its two quadrant counterparts; the regular atan(),is that it only gives output in the range between −π/2 and π/2. As both I and Q can take on both positive and negative values, and they are divided with each other in order to use the arcus tangent, the two expressions atan(-1/-1) and atan(1/1) would of course yield the same result even though they are in opposite quadrants. The function atan2() can handle this situation, thus producing the correct output in the full unit circle range between−π and π.

The function unwrap() is used to correct jumps in angles that are over π

All of the above calculations can be carried out on samples instead of continuous time signals. The only calculation that may introduce approximation errors is the differ-entiator, which can not be accurate when operating in discrete time. ***A discrete time differentiator will use the approximation :***

$$
\frac{dx}{dt} \approx \frac{x(nT_s) - x((n-1)T_s)}{T_s}
$$

Where:

- x(t) is a continuous time signal.
- N is an integer sample number.
- Ts is the sample time.

```
strad = unwarp_atan2[1:] - unwarp_atan2[:-1]
```

Now, we redo the same process to filter some input (noise).

The purpose of the input filter is to match the signal from the downconverter to the sound card. The signal from the downconverter is compared with the specifications for the sound card; it is shown that some of the signals will be interpreted as noise. This is because the sound card filter has analog frequency response from 20 Hz - 20 kHz.

```
# create an order 6 Butterworth lowpass filter to remove unwanted stations, that
# starts cutting at 0.2392*fs/2 and has the coefficients:
b1 = [0.00073786 , 0.00442716, 0.01106791 , 0.01475721 , 0.01106791 ,  0.00442716, 0.00073786]
a1 = [1.000000 ,-3.183561 , 4.622162 , -3.779396 , 1.813557 , -0.479983 ,  0.054443]

# The coefficients are used like this:
# a(1)*y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
#           - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
# which is equivalent to the lfilter command below
Ifilt = lfilter(b1,a1,I)
Qfilt = lfilter(b1,a1,Q)

Ifiltdown = Ifilt[::downer]
Qfiltdown = Qfilt[::downer]
unwarp_atan2_down = np.unwrap(atan2(Qfiltdown,Ifiltdown))
stradfilt = unwarp_atan2_down[1:] - unwarp_atan2_down[:-1]
```

We apply the Discrete Fourier Transformation to achieve the best frequency.

The DFT will be implemented using the fft() function in python, this is a speed optimized version of the DFT. Passing the entire signal x[n] to the fft() function is the equivalent of using a rectangular window with the same size as the signal.

```
#SpecFM is the spectrum of the demodulated signal
specFM = np.abs(fft(stradfilt))
```

We remove the down conversion filter and rebuild the original frequencies in order to plot them later :

```
freq = np.array(range(len(stradfilt)))   * 1.0 / len(stradfilt) * ( fs * 1.0 / downer )
return specFM, strad, stradfilt, freq
```

We write an audio file using using the data we got before applying the input filter to remove noise and other stations data :

```
f = wave.open("strad.wav", "wb")
f.setnchannels(1)
f.setsampwidth(2)
f.setframerate(fs * 1.0 / downer)
f.setframerate(142857)
factor = 16383.0 / max(abs(strad.max()),abs(strad.min()))
print("strad audio to write",(strad * factor).astype(np.int16))
f.writeframes((strad * factor).astype(np.int16).tostring())
f.close()
```

Now, we write the final audio which is more pure (filtered) :

```
f = wave.open("stradfilt.wav", "wb")
f.setnchannels(1)
f.setsampwidth(2)
f.setframerate(fs * 1.0 / downer)
factor = 16383.0 / max(abs(stradfilt.max()),abs(stradfilt.min()))
print("stradfilt audio to write",(stradfilt * factor).astype(np.int16))
f.writeframes((stradfilt * factor).astype(np.int16).tostring())
f.close()
```

Finally, we plot the FM data :

```
#plt.plot([1,2,3,4], [1,4,9,16])
# Plot the spectrum of the demodulated signal. You should see a monaural bump at low
# frequency, a pilot carrier at 19 kHz, a stereo bump at 38 kHz, and, on some stations,
# an RDS bump at 57 kHz.

plt.plot(freq, specFM)
plt.show()
```

2- Demodulation of a sigfox uplink frame & GFSK signal

Input :

1) We are provided with a file that represents the raw data sent from SigFox and we also have the original data content

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/SigFox$ tree .
.
├── data_sent
└── RTL-SDR-20191001_210150-868_130MHz-500kSps-400kHz_w_GFSK8.complex

0 directories, 2 files
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/SigFox$ cat data_sent
085F140015CD5B0715CD5B07
```

2) In addition, we have a short description :

- Operating in the band 868.034 - 868.226 MHz
- In uplink only
- in DBPSK Modulation

Output :

We are asked to :

- Analyse the SigFox signal in URH and describe its characteristics (all 3 repetitions)
- Explain why URH can't auto demodulate the signal
- Write a program that demodulate the message
- Verify the program with other frames
- Find the GFSK signal
- Try to decode all three repeated frames
- Determine the reception threshold for RTL-SDR
- Estimate the maximum distance to which we should be able to capture and interpret a Sigfox frame. How could we validate this estimate empirically ?

What is Sigfox ?

Sigfox is a narrowband (or ultra-narrowband) technology. It uses a standard radio transmission method called binary phase-shift keying (BPSK), and it takes very narrow chunks of spectrum and changes the phase of the carrier radio wave to encode the data. This allows the receiver to only listen in a tiny slice of spectrum, which mitigates the effect of noise. It requires an inexpensive endpoint radio and a more sophisticated basestation to manage the network.

Sigfox communication tends to be better if it's headed up from the endpoint to the base station. It has bidirectional functionality, but its capacity going from the base station back to the endpoint is constrained, and you'll have less link budget going down than going up. This is because the receiver sensitivity on the endpoint is not as good as on the expensive basestation.

Uplink vs Downlink

Bidirectional communication sometimes is relevant on the Internet of Things field. In some LPWAN protocols it's complex to have bidirectional communication. With Sigfox and thethings.iO it's really easy to set up bidirectional communication, that is called downlink.

The uplink process is the usual way of data coming from the devices to the Sigfox backend, that can be redirected to thethings.iO with a callback. Find here how to connect Sigfox with thethings.iO.

The downlink process is the process that enables a device to get data coming from a cloud, a mobile application or another device.

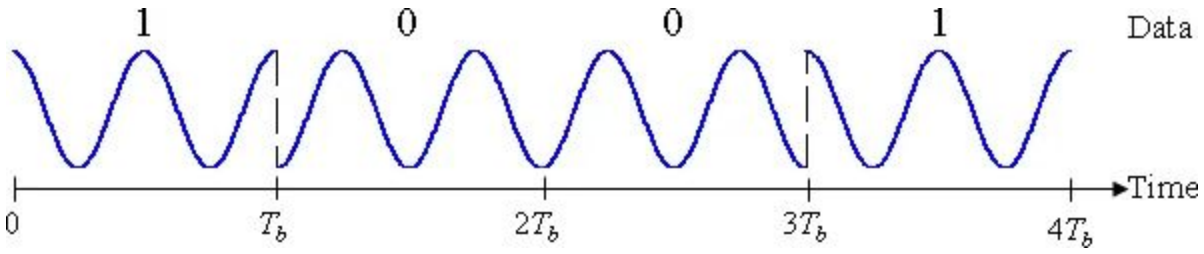
This downlink process is only available when the uplink process includes an ACK flag corresponding to a downlink request. At that point thethings.iO will be available to transfer the downlink message to the device through the Sigfox backend.

Background

Sigfox is a LPWAn (Low Power & Wide Area) communication technology and network dedicated to IoT. The specificity of this solution is related to the LPWAn characteristics : communicating Low Power (25mW) and Wide Area (60km).

This is possible thanks to particular radio characteristics and modulation.

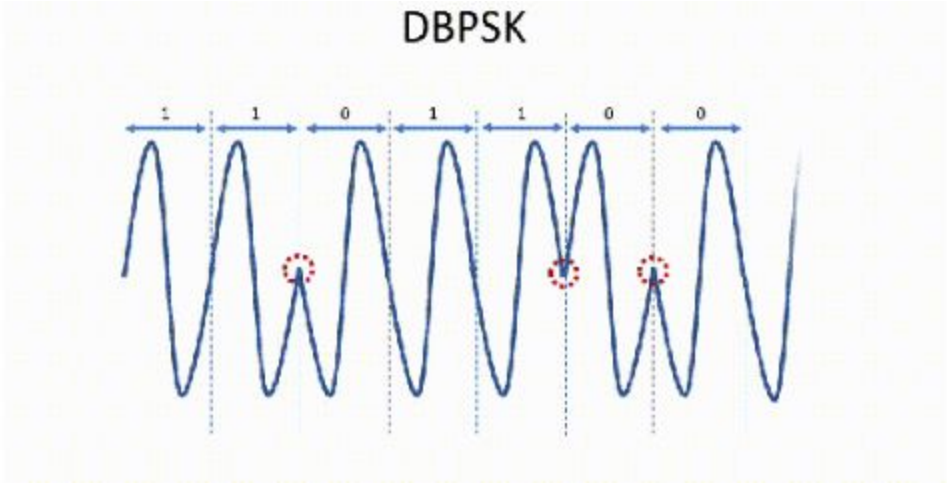
Understanding BPSK :



The receiver job is to determine a reference for the phase and identify the phase change to identify the bit switching. There is no way when listening to identify what is a 0 and what is a 1 at this point.

Understanding DBPSK :

According to different documentation the Sigfox modulation is **DBPSK** for **Differential Binary Phase Shift-Keying**. In this modulation we also use the phase to encode 0 and 1 but instead of using the phase for this we use the phase change. That way we do not need a reference of phase to identify a 1 vs a 0.



In the above example the 1 is identified by no change in the signal phase when the 0 is identified by a phase change during the transmission.

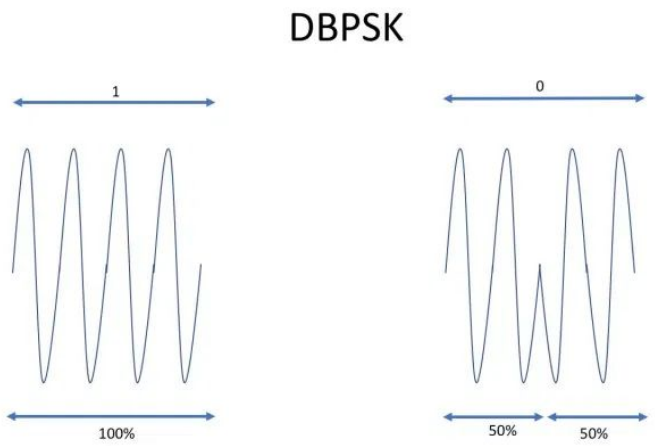
This is the way Sigfox is encoding the information on the radio medium.

That said there is a big difference between Sigfox encoding and this animation : In the animation is encoding 1 different bit per signal period. At 868MHz it means 868Mb/s

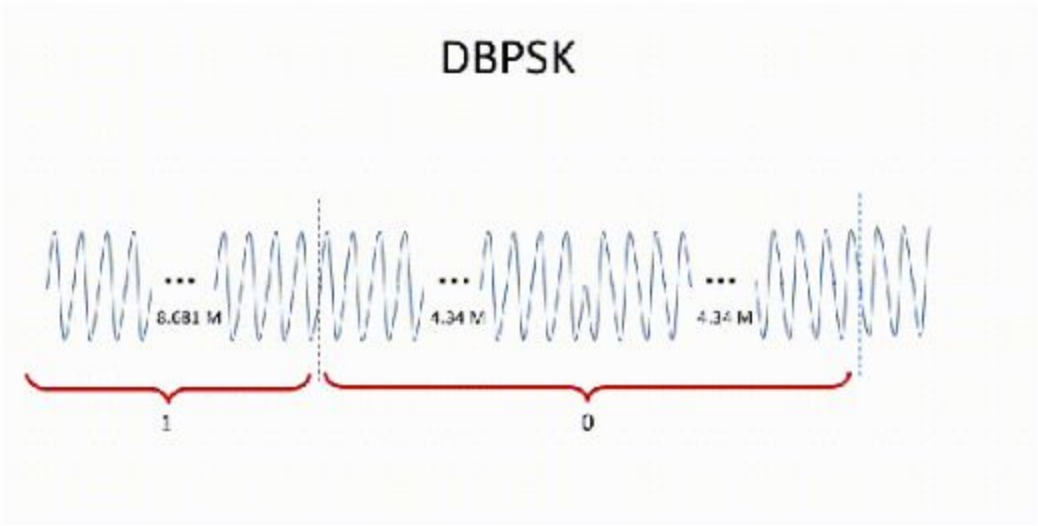
As you may know Sigfox is far away from this bitrate and basically such encoding would have been really difficult to decode even in a short range.

The Sigfox DBPSK encoding is based on a suite of signal periods : when all have the same phase the bit value is a 1, if the phase change in the middle of the time slot the bit value is a 0.

That way the receiver is less impacted by local phase variation.

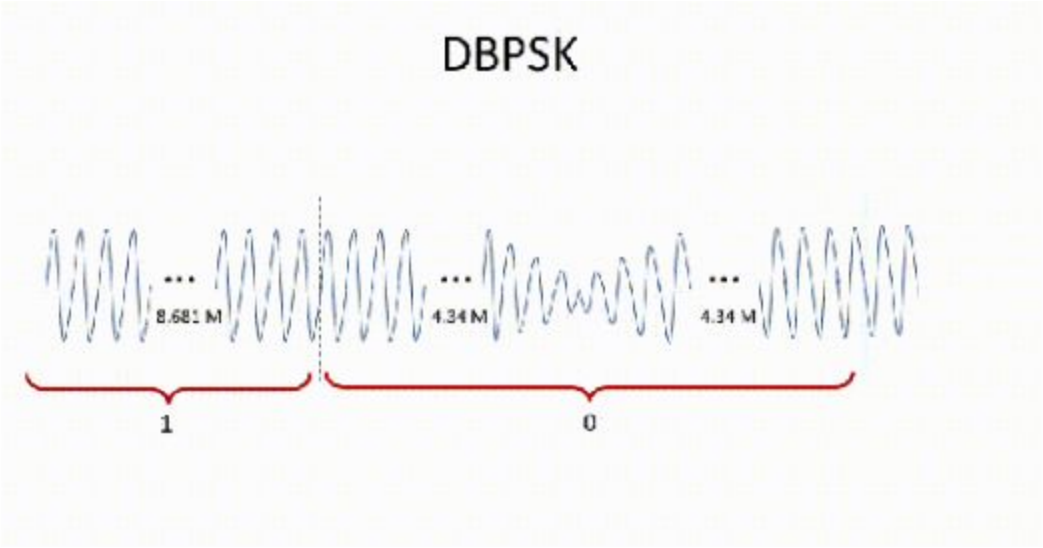


In the Sigfox case the bitrate is 100 bit/second and the radio signal is 868.1MHz. It means that each transmitted bit will correspond to 8 681 000 consecutive cycles. The receiver will use this large number of phases to statistically find the phase shift for the corresponding bit.



The above animation shows a 1 encoded with 8.68M periods encoded the same phase and a 4.34 + 4.34 periods encoded with a phase shift in the middle, representing a 0.

In this above example the phase shift is executed with a full signal amplitude (and power) this would impact the spectral occupation the wrong way. To avoid it the Sigfox signal is modulating the amplitude during the phase change like in the following animation.



Sigfox Protocol

Sigfox protocol is mainly based on low throughput network (LTN) protocol. The Message is 12 bytes long and it includes a sequence number for security purposes.

Frequency Bands (MHz)	Europe: 868, Others: 902 - 928	**Tx Output Power**	Europe: 14 dB, USA: 22 dB
Bandwidth	192 kHz	**Data Rate**	100 bps or 600 bps
Modulation	Uplink: DBPSK Downlink: GFSK	**Payload**	Uplink :12 bytes Downlink: 8 bytes
Power consumption	Active: 10 - 50 mA, Idle: 6 nA	**Number of Messages**	Per Hour: 6, Per Day: 140
Security	AES	**Roaming required**	No

Sigfox uses ultra narrowband (UNB) modulation to send and receive messages. To be specific, Sigfox sends 3 messages using multiple channels and the packet duration is 2 ms.

Sigfox messages are 0 to 12 bytes long. Each Sigfox Message frame includes preamble bits, frame synchronization bits, device identifier bits, payload bits, authentication codes and also the frame check sequence (FCS) bits.

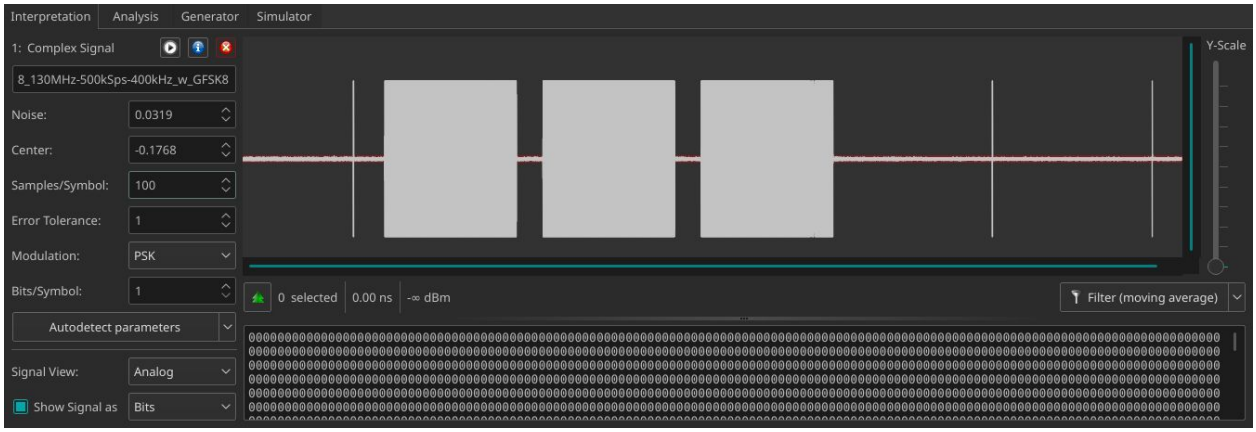
The ***generic*** structure of uplink and downlink frames are given in Table below.

Uplink Frame	Preamble (32 bits)	Frame Sync (16 bits)	Device ID (32 bits)	Payload (0 - 96 bits)	Message Authen-tication Code (16 - 40bits)	FCS (16 bits)
Down link Frame	Preamble (32 bits)	Frame Sync (13bits)	ECC (32bits)	Payload (0-64 bits)	Message Auth Code (16bits)	FCS (8 bits)

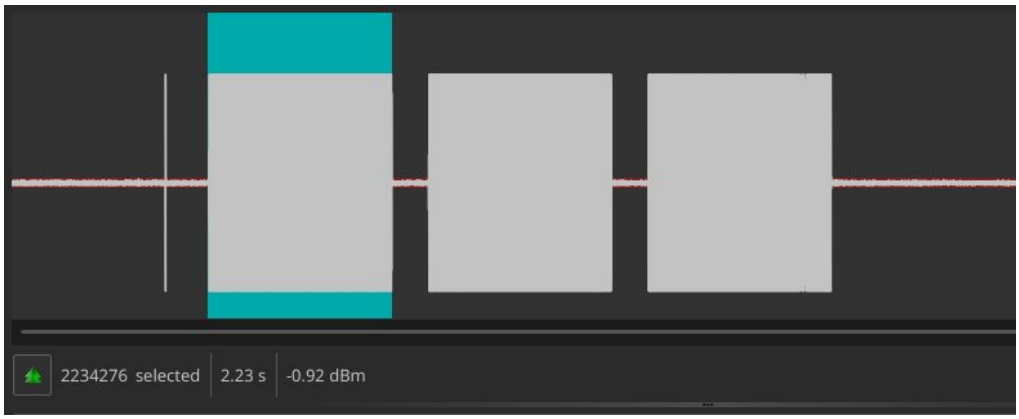
Static analysis

In our case, we are provided with a sigfox ***uplink*** signal that corresponds to the data "***085F140015CD5B0715CD5B07***"

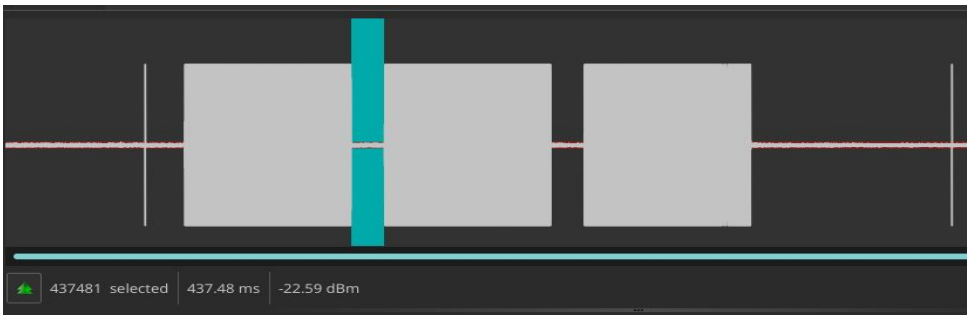
We want to take a look at it in urh :



We notice that there are 3 frames of approximative 2.23s :

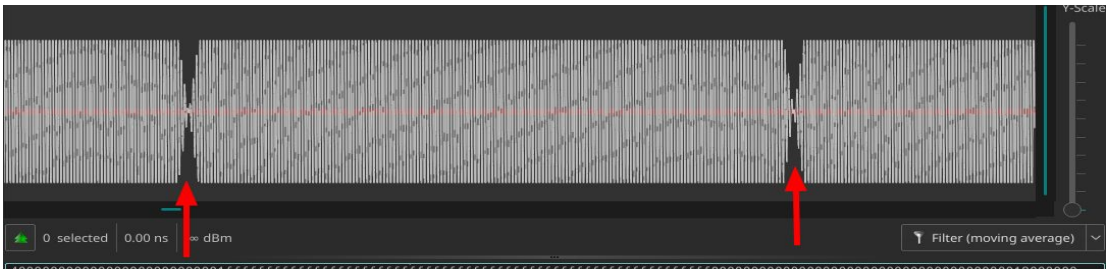


With a silence of 0.437s between each of them :

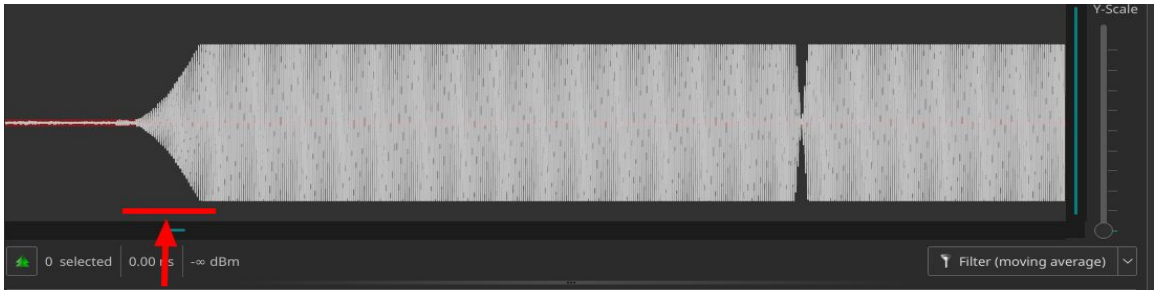


Signal characteristics :

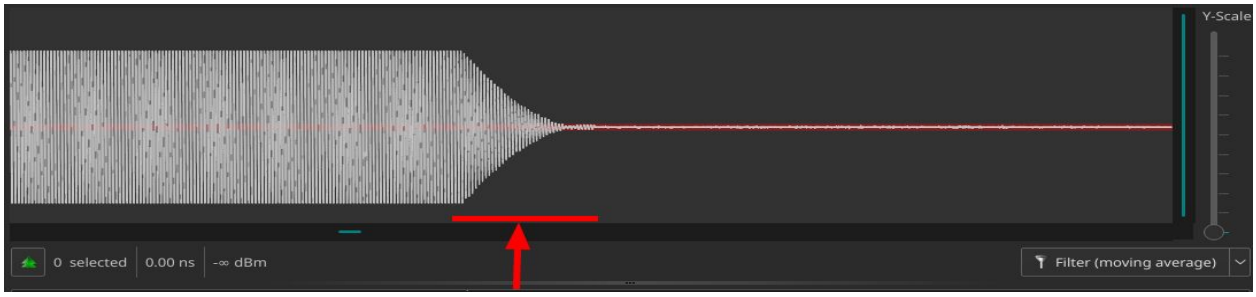
We notice that there is a symbol shaping is added during the phase rotation, to keep the emission spectrum within the limits :



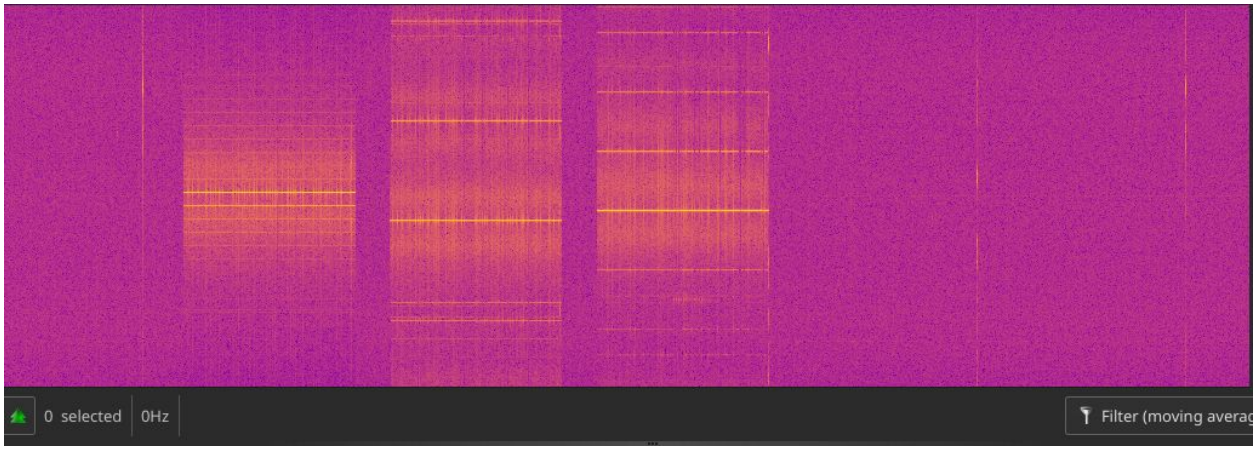
Before the first symbol corresponding to the first bit of a frame, a radio burst may start with a ramp-up and extra symbols. This extra transmission shall be less or equal to 2 x TSul and with no phase modulation in it.



In the same manner, a radio burst may end with extra symbols and a ramp down. This extra transmission shall be less or equal to 2 x TS ul and with no phase modulation in it.



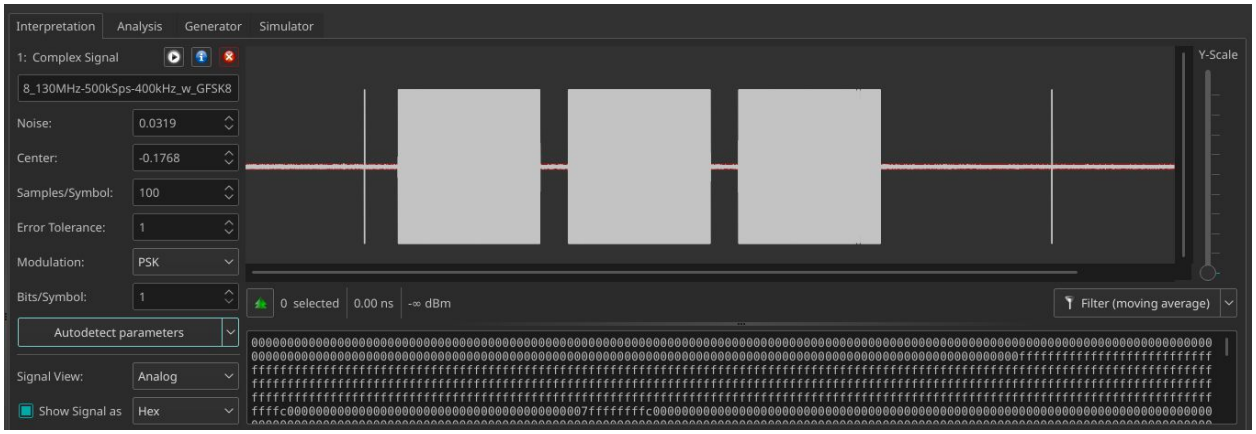
The 2 replicas frames are sent at different frequency :



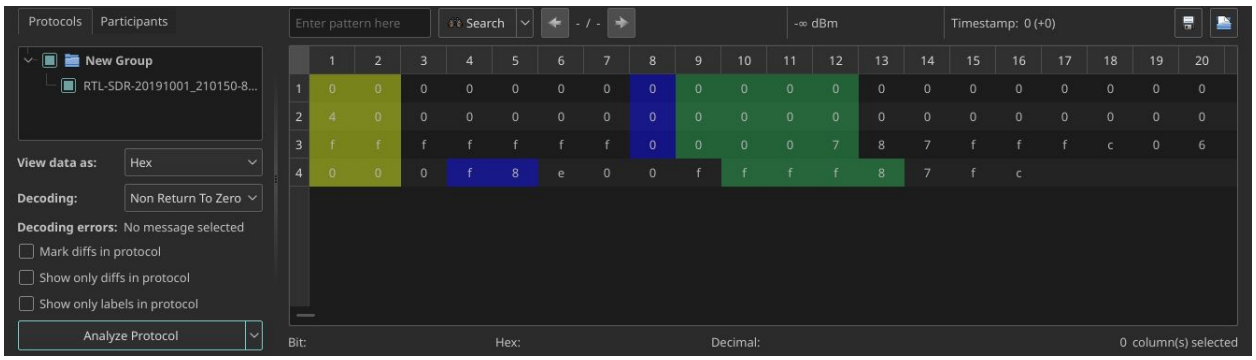
This signal respects all the characteristics of unplink sigfox signal from frequency (868MHZ), number of frames (3 at different frequency) and the modulation (DBPSK).

URH automatic demodulation :

If we try to autodetect the parameters in URH :



And then auto-analyse the protocol :



We notice that it fails immediately and that's not surprising !

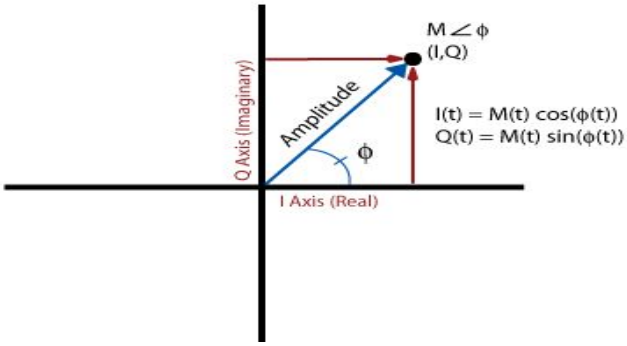
Because urh doesn't support DBPSK modulation and it doesn't have a plugin for decoding the sigfox protocol.

Uplink demodulation

We start by loading the complex file and read the IQ data separately :

```
# read I and Q data
Fs = 1000000
bit_rate = Fs / UPLINK_BAUDRATE
IQ = np.fromfile(file, dtype=np.float32)
I = IQ[0::2]
Q = IQ[1::2]
```

We calculate the signal power's envelope using the IQ data:



One common calculation that is performed on the IQ data is to determine the RF power vs. time. The formula used to compute RF power is remarkably simple. It's interesting to review how this formula is derived.

The I and Q values represent the peak value of the in-phase and quadrature components of the RF signal vector. The Pythagorean Theorem tells us that peak RF signal voltage is equal to:

$$V_{peak} = \sqrt{(I^2 + Q^2)}$$

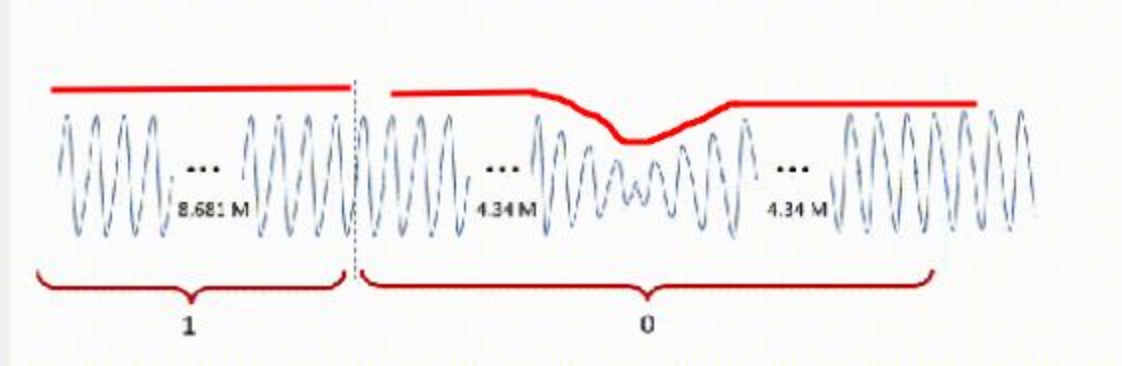
By definition, the I and Q values are the peak values of the sinusoidal quadrature components of the RF signal. Thus, the RMS value of the RF signal is given by:

$$V_{RMS} = \sqrt{\left(\frac{I^2 + Q^2}{2}\right)}$$

```
# get signal envelope r = \/ (I^2 + Q^2) / 2    https://www.tek.com/blog/calculating-rf-power-iq-samples
baseband_envelope = []
for i in range(len(I)):
    baseband_envelope.append( math.sqrt( (I[i]**2 + Q[i]**2) / 2 ) )
```

But, why ?

We are not going to do DBPSK demodulation because it's harder.
SigFox has a special characteristic : when changing the phase of the DBPSK modulation it lowers the signal's power :



We will exploit this to spot the phase shift.

Then, we calculate the average power to know when it's high and when it's going low :

```
# calculate the average  of the signal power to spot the amplitude change on the sigfox DBPSK modulation
average_power = max(baseband_envelope)/2

# get the binary data of the signal
binary_data   = exploit_envelope(baseband_envelope, average_power, bit_rate)
```

Here is the function that exploit the signal's envelope :

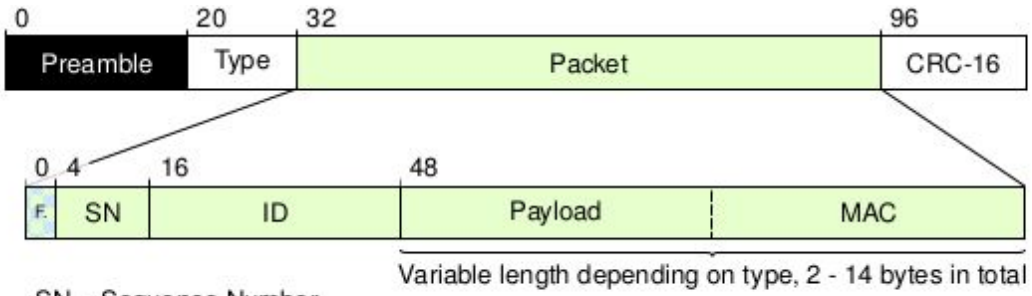
```python
def exploit_envelope(baseband_envelope, average_power, bit_rate) :
    binary_data = ""
    rate_count = 0
    for i in baseband_envelope :
        if i > average_power:                    # Signal power is stable at the high degree
            rate_count += 1                      # we increment the rate_counter and continue
        else:                                    # Signal power is going down :  either it's the end of a bit count or changing phase in the DBPSK
            bit_count = round(rate_count/bit_rate)  # calculate the actual bit count
            rate_count = 0                       # reset rate counter

            if bit_count > 0:                    # if we reach 1 bit count at least we decode it
                                                 # if the last amplitude changed from the last one we keep it 0, else we change it to 1
                binary_data += ( "0" + "1" * (bit_count - 1) )
    return binary_data
```

We loop through the signal's power envelope and we calculate the rate in parallel.

If the power is still above average we keep looping because the state is highly stable else we calculate the bit count from the rate counter and we reset the counter then, if we have a rate of 1 bit or more we switch the state from 0 to 1 and we keep adding 1 if the bit count is more than 1 bit count.

Once the data is returned we parse it using the sigfox uplink structure

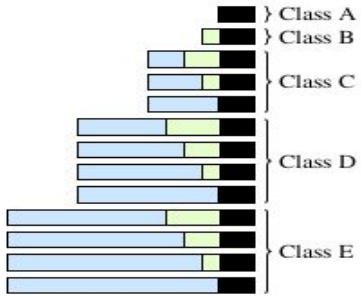


SN = Sequence Number
ID = Device ID
F. = Flags

Preamble	Predefined pattern, `0b1010` repeated 5 times
Type	Type of packet (class and number of repetition)
Flags	Various uses: Number of bytes allocated for MAC, down-link request, single-bit message value
Sequence Number	Number that is incremented after each transmission, so that it is unique until it overflows
Device ID	32-bit identifier that is preconfigured in every Sigfox object
Payload	Message to be transmitted, in plaintext
MAC	Message Authentication Code (variable length, 2 to 5 bytes)
CRC-16	16-bit CRC checksum

The Payload and MAC lengths are dynamic and determined through the type field using this table :

Lengths [bytes]			
Payload	**MAC**	**Total Packet Length, Class**	**MSBs of Flags**
0	2	8, A	0b1V*
1	2	9, B	0b00
2	4	12, C	0b10
3	3	12, C	0b01
4	2	12, C	0b00
5	5	16, D	0b11
6	4	16, D	0b10
7	3	16, D	0b01
8	2	16, D	0b00
9	5	20, E	0b11
10	4	20, E	0b10
11	3	20, E	0b01
12	2	20, E	0b00



Which i translated to a python dictionary :

```
SIGFOX_FRAME      = namedtuple("SIGFOX_FRAME", "PREAMBULE FTYPE FLAGS SEQUENCE_NUM DEVICE_ID PAYLOAD MAC CRC16")
SYNC              = "1010" * 5
UPLINK_BAUDRATE   = 100
CRC16_POLYNOMIAL = 0x1021
MIN_LEN_FRAME     = 86
MAX_LEN_FRAME     = 192
MAX_FRAME_NUM     = 3
PACKET_AND_PAYLOD_LEN_FROM_FTYPE = {
    0x06b : [8, 0 ], 0x6e0 : [8, 0 ], 0x034 : [8, 0 ],                  # class A
    0x08d : [9, 1 ], 0x0d2 : [9, 1 ], 0x302 : [9, 1 ],                  # class B
    0x35f : [12,2 ], 0x598 : [12,3 ], 0x5a3 : [12,4 ],                  # class C
    0x611 : [16,5 ], 0x6bf : [16,6 ], 0x72c : [16,7 ], 0xf67 : [16,8],  # class D
    0x94c : [20,12], 0x971 : [20,11], 0x997 : [20,10], 0x9bf : [20,9]   # class E
                                    }
```

```
def parse_frames(binary_data) :
    #             "01010"*5    "XXX"             "XXXX"
    # FRAME :    | PREAMBLE | FTYPE | PACKET | CRC16       X       XXX          XXXXXXXX    X....X   X..X
    #                                 PACKET :          | FLAG | SEQUENCE_NUM | DEVICE_ID | PAYLAOD | MAC
    binary      = binary_data
    CRC         = "NOT OK"
    frames      = []
    frame_count = 0
    end_frame   = 0
    while len(binary) > MIN_LEN_FRAME and frame_count < MAX_FRAME_NUM :
        start_frame  = binary.index(SYNC)
        binary       = binary[ start_frame : ]
        ftype        = int(binary[20 :32],2)
        len_packet   = PACKET_AND_PAYLOD_LEN_FROM_FTYPE[ftype][0] * 8
        len_payload  = PACKET_AND_PAYLOD_LEN_FROM_FTYPE[ftype][1] * 8
        len_mac      = len_packet - len_payload - ( 6 * 8 )
        len_frame    = 20 + 12 + len_packet + 16
        end_payload  = 80 + len_payload
        end_mac      = end_payload + len_mac
        end_frame   += start_frame + len_frame

        PREAMBULE    = "0x{:05x}".format(int(binary[0 :20],2))
        FTYPE        = "0x{:03x}".format(int(binary[20:32],2))
        FLAGS        = "0x{:01x}".format(int(binary[32:36],2))
        SEQUENCE_NUM = "0x{:03x}".format(int(binary[36:48],2))
        DEVICE_ID    = "0x{:08x}".format(int(binary[48:80],2))
        PAYLOAD      = "0x{:0{}x}".format(int(binary[80:end_payload],2),      len_payload // 4)
        MAC          = "0x{:0{}x}".format(int(binary[end_payload:end_mac],2), len_mac     // 4)
        CRC16        = "0x{:04x}".format( int(binary[end_mac:len_frame], 2))

        frame = SIGFOX_FRAME(PREAMBULE, FTYPE, FLAGS, SEQUENCE_NUM,  DEVICE_ID, PAYLOAD, MAC, CRC16)
        hex_frame = hex( int(binary[20:len_frame],2) )
        frames.append([frame,hex_frame])

        if frame_count == 0 and CRC16 == uplink_crc(binary[32:end_mac]) :
            CRC = "OK"
        binary = binary_data[end_frame:]
        frame_count += 1

    return frames, CRC
```

I have added the CRC function of sigfox which a CRC-16-CCITT binary value is inverted before transmission, in other words all bits in the CRC-16 checksum are flipped.

```
def uplink_crc(data) :
    data = bytes(int(data[i : i + 8], 2) for i in range(0, len(data), 8))
    remainder = np.uint16(0)

    for byte in data  :
        remainder ^= (byte << 8)
        for _ in range(8) :
            remainder = ((remainder << 1) ^ CRC16_POLYNOMIAL) if (remainder & (1 << 15)) else (remainder << 1)

    return hex( ~ np.uint16(remainder) )
```

Here is our program demodulating the first frame :

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/SigFox$ ./sigfox_demodulator.py -h
usage: sigfox_demodulator.py [-h] -f FILE -d {uplink,downlink}

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  Complex file containing the uplink recording to be demodulated
  -d {uplink,downlink}, --decode {uplink,downlink}
                        Decode content of uplink/downlink frame
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/SigFox$ ./sigfox_demodulator.py -f RTL-SDR-20191001_210150-868_130MHz-500kSps-400kHz_w_GFSK8.complex -d uplink
SIGFOX_FRAME(PREAMBULE='0xaaaaa', FTYPE='0x94c', FLAGS='0x0', SEQUENCE_NUM='0x040', DEVICE_ID='0xdb632e00', PAYLOAD='0x085f140015cd5b0715cd5b07', MAC='0x2d3e', CRC16='0
x71df')
SIGFOX_FRAME(PREAMBULE='0xaaaaa', FTYPE='0x971', FLAGS='0x0', SEQUENCE_NUM='0x070', DEVICE_ID='0x800a7280', PAYLOAD='0x0e675b001a58a0455a58a0', MAC='0x4570ee', CRC16='0
xd547')
SIGFOX_FRAME(PREAMBULE='0xaaaaa', FTYPE='0x997', FLAGS='0x0', SEQUENCE_NUM='0x050', DEVICE_ID='0xedbbe580', PAYLOAD='0x0a48d10010be0dc6d0be', MAC='0x0dc6e6e3', CRC16='0
xdb50')
CRC: OK
```

With the second frame also :

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/SigFox$ ./sigfox_demodulator.py -f RTL-SDR-20191001_210322-868_130MHz-500kSps-400kHz_trame_2.complex -d uplink
SIGFOX_FRAME(PREAMBULE='0xaaaaa', FTYPE='0x94c', FLAGS='0x0', SEQUENCE_NUM='0x041', DEVICE_ID='0xdb632e00', PAYLOAD='0x085f14001aaaaa0715cd5b07', MAC='0x1168', CRC16='0
xc06a')
SIGFOX_FRAME(PREAMBULE='0xaaaaa', FTYPE='0x971', FLAGS='0x0', SEQUENCE_NUM='0x071', DEVICE_ID='0x400a7280', PAYLOAD='0x0e675b00115555855a58a0', MAC='0x455d86', CRC16='0
x9045')
SIGFOX_FRAME(PREAMBULE='0xaaaaa', FTYPE='0x997', FLAGS='0x0', SEQUENCE_NUM='0x051', DEVICE_ID='0xadbbe580', PAYLOAD='0x0a48d1001c000086d0be', MAC='0x0dc6d532', CRC16='0
xf070')
CRC: OK
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/SigFox$ _
```

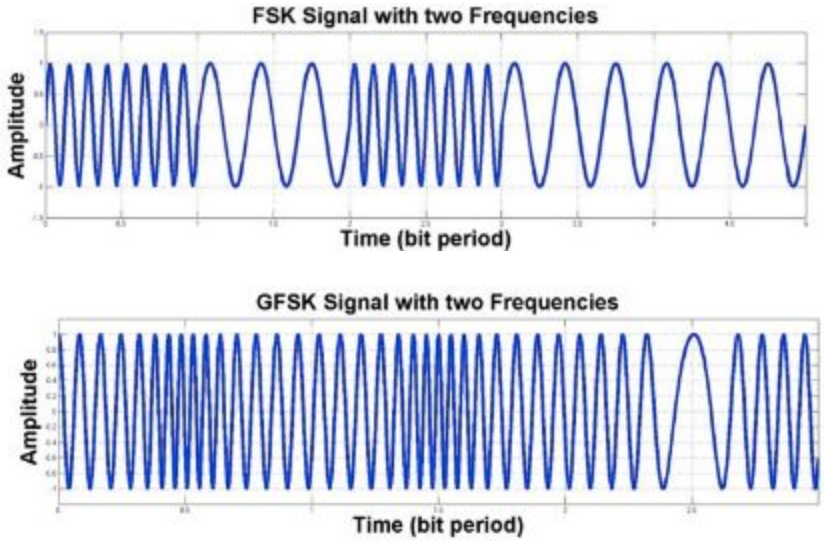
Et voila !

GFSK Detection

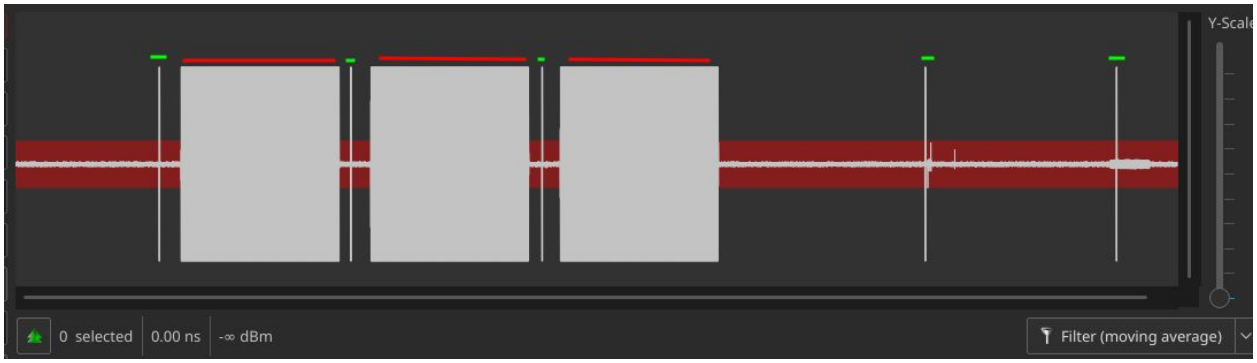
We are given information that our sigfox frames contains also some data from smoke detector sent every 2 seconds which is modulated in GFSK.

We have seen what FSK modulation is, but what is GFSK ?

GFSK stands for Gaussin Frequency Shift Keying modulation. In GFSK, baseband pulses (consists of -1 and 1) are first passed through the gaussian filter before modulation. This makes pulses smooth and hence limits the modulated spectrum width. This process is known as pulse shaping.



We start by loading the file in urh :

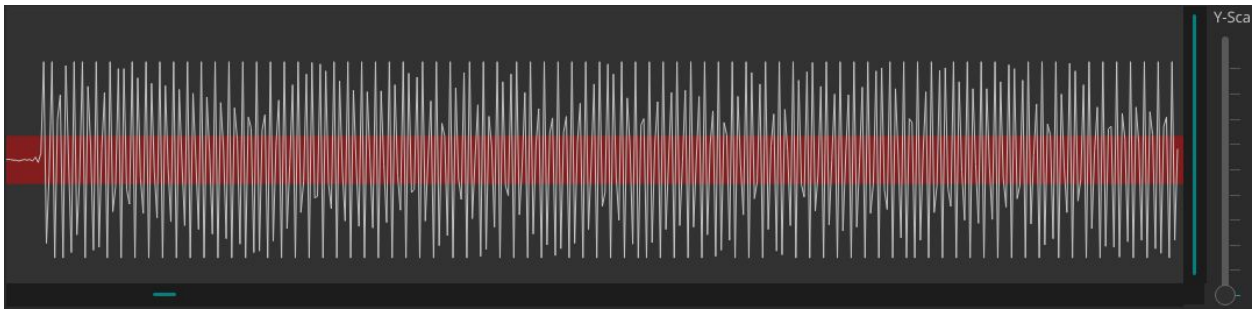


We have already studied the red signals which are sigfox uplink signals.

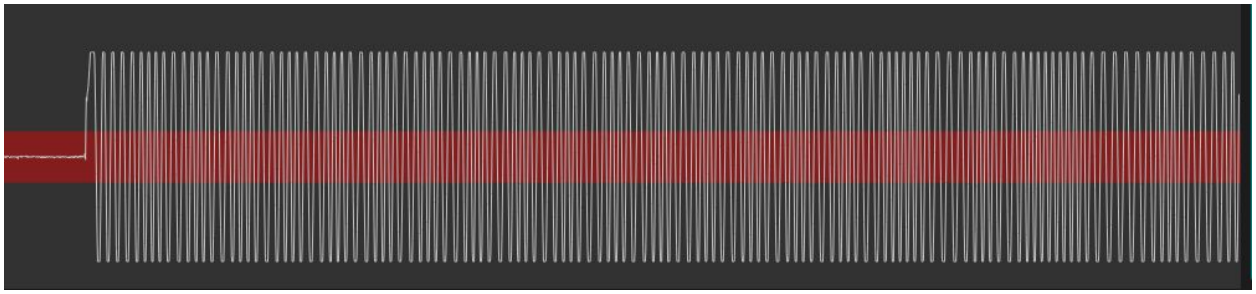
What are the green ones ?

Let's zoom at them :

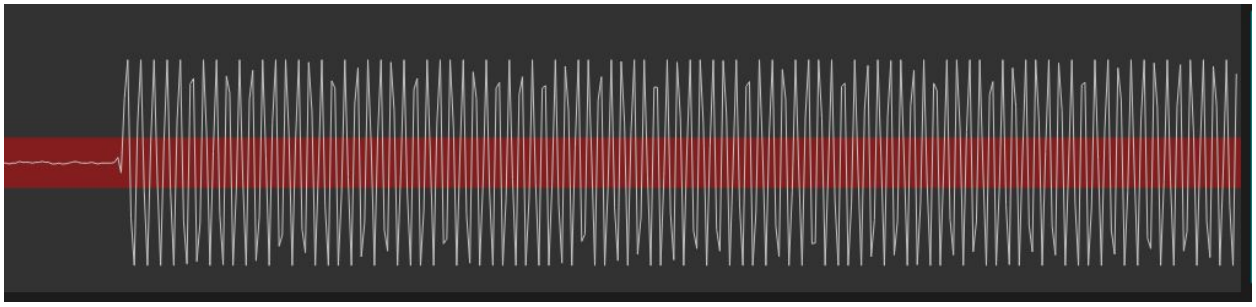
- **1st one:**



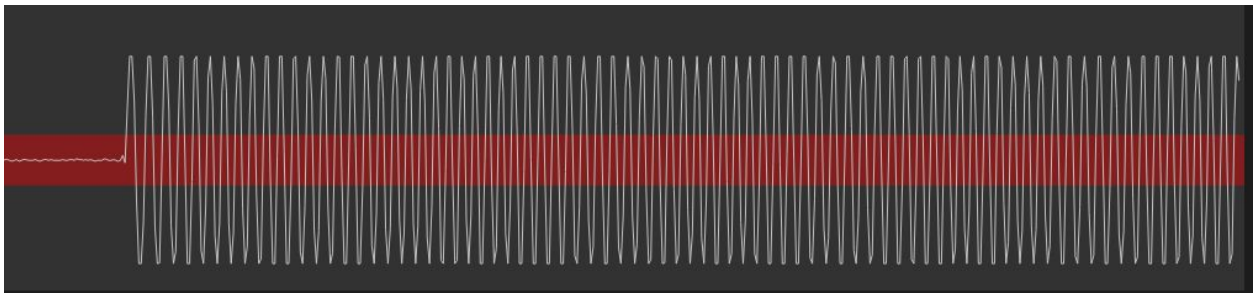
- **2nd one:**



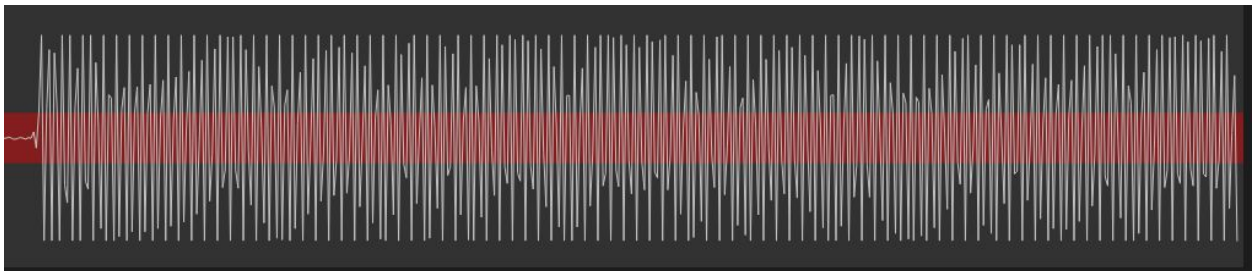
- **3rd one :**



- **4th one :**



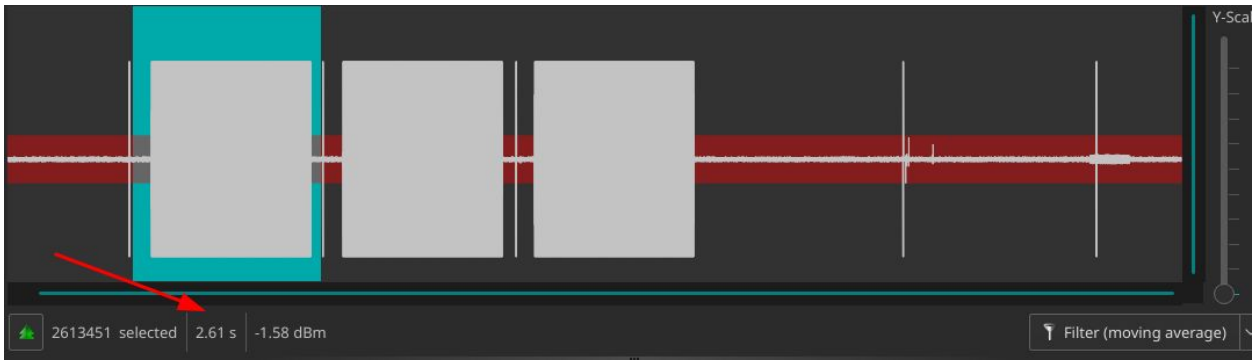
- **5th one :**



That looks like a pattern :

1st one contains data in GFSK modulation, 2nd one is a stable one (silence), 3rd one contains data and so on....

There is a break of approximately 2.6s between each one :



Except between 3rd and 4th one which has a double duration (4.6)

In URH choosing FSK modulation and pressing autodetect parameters give a value for the modulated signals :

1. **4000244804400f80**
2. **ffffffffffffffff**
3. **f7ff6ea68ab2019e**
4. **ffffffffffffffff**
5. **0000489017880d80**

- All the demodulated signals give a 8 byte data.

- Silence is represented by 8 bytes 0xff

Demodulating uplink replicas

During a transmission of data the frame is transmitted 3 times. The network does not transmit exactly the same information 3 times. The First Repeat (first transmission) is not encoded; the bytes are directly readable.

The second repeat is encoded in a specific way, thanks to this encoding the phase switching will be totally different from the first transmission. I assume this is helping the receiver to extract the information more easily in strong noise conditions.

The function is not to apply to the whole frame but only the F.TYPE, SEQ.ID, DEVICE.ID and PAYLOAD part separately.

Basically for the second repeat, the bit value depends on the previous two bit values :

when not identical the current bit is switched :
CONVOLUTIONAL CODE with a generator polynomial of $X^2 + 1$ (101 or 5)

```python
# convolutional "encoder/decoder" : realizes the register shifts (Polynomial Multiplication/Division) using a Polynomial Generator 101 : X² + 1
def encode_decode_r2_fields(field, Enc_Dec):
    field = bin(int('1'+field[2:], 16))[3:]
    w     = [int(field[i : i + 8], 2) for i in range(0, len(field), 8)]

    b2    = 0
    b1    = 0
    j     = 0
    result = []
    while j < len(w):
        n = w[j]
        v = 0
        i = 0
        while i < 8 :
            b0 = (n & 128) >> 7
            v += b0 if b2 == b1 else 1 - b0
            b2 = b1
            b1 = b0 if Enc_Dec else v & 1
            n  = n << 1
            v  = v << 1
            i +=1
        v  >>= 1
        w[j] = v
        result.append("0x{:02x}".format(w[j]))
        j   += 1

    return "0x" + "".join( x.replace("0x",'') for x in result )
```

The third repeat is encoded the same way but using a different algorithm. ***Basically depends on the previous two bits, each of them indicates the current bit and the next one has to be switched or not :***
CONVOLUTIONAL CODE with a generator polynomial of X² + X + 1 (111 or 7)

```python
# convolutional "encoder/decoder" : realizes the register shifts (Polynomial Multiplication/Division) using a Polynomial Generator 111 : X² + + X + 1
def encode_decode_r3_fields(field, Enc_Dec):
    field = bin(int('1'+field[2:], 16))[3:]
    w     = [int(field[i : i + 8], 2) for i in range(0, len(field), 8)]

    b2    = 0
    b1    = 0
    j     = 0
    result = []
    while j < len(w):
        n = w[j]
        v = 0
        i = 0
        while i < 8 :
            b00 = (n & 128) >> 7
            b01 = (n & 64) >> 6
            v  += b00 if b2 == 0 else 1 - b00
            v   = v << 1
            v  += b01 if b1 == 0 else 1 - b01
            b2  = b00 if Enc_Dec else (v & 2) >> 1
            b1  = b01 if Enc_Dec else v & 1
            v   = v << 1
            n   = n << 2
            i  += 2
        v  >>= 1
        w[j] = v
        result.append("0x{:02x}".format(w[j]))
        j   += 1

    return "0x" + "".join( x.replace("0x",'') for x in result )
```

Result :

```
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/SigFox$ ./sigfox_demodulator.py -r 0x9970050edbbe5800a48d10010be0dc6d0be0dc6e6e3db50
[*] Reversed replica:
SIGFOX_FRAME(PREAMBULE='0xaaaaa', FTYPE='0x94c', FLAGS='0x0', SEQUENCE_NUM='0x040', DEVICE_ID='0xdb632e00', PAYLOAD='0x085f140015cd5b0715cd5b07', MAC='0x2d83', CRC16='0
xe3bf')
Yan1x0s@1337:~/0x00/2019/SECCOM/Projet/SigFox$ _
```

Resources :

https://www.usenix.org/system/files/conference/woot18/woot18-paper-pohl.pdf

Reverse Engineer Wireless Temperature / Humidity / Rain Sensors — Part 1 « RAYSHOBBY.NET

Hacking the Acurite 0077XW / 00592TX Wireless Remote Temperature Probe – Part 1 – hacking the hardware – Tech Spin

elttam - Intro to SDR and RF Signal Analysis

Capture and decode FM radio

Reading Amateur Radio Frequencies with RTLSDR device and Python - How To Train Your Robot

reverse engineering static key remotes with gnuradio and rfcat ·

What is I/Q Data? - National Instruments

I/Q Data for Dummies

An FM/RDS (Radio Data System) Software Radio - 54_IWK_2009_1_0_08.pdf

Capture and decode FM radio

The Sigfox radio protocol - disk91.com - technology blogdisk91.com – technology blog

https://media.ccc.de/v/35c3-9491-hunting_the_sigfox_wireless_iot_network_security

https://github.com/Jeija/renard-phy