Final Project
Degree in Telecommunications Engineering

Python modelling of the SX1257 transceiver

Author: Roberto Lama Rodríguez
Advisor: Hipólito Guzmán Miranda

Electronic Engineering Department
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Final Project
Degree in Telecommunication Engineering

Python modelling of the SX1257 transceiver

Autor:

Roberto Lama Rodríguez

Advisor:

Hipólito Guzmán Miranda

Profesor titular

Electronic Engineering Department

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2022

Final Project: Python modelling of the SX1257 transceiver

Autor: Roberto Lama Rodríguez

Tutor: Hipólito Guzmán Miranda

The comitee judging the Project indicated above is compromised of the following members:

President:

Chair:

Secretary:

They agree to grant the project a grade of:

Sevilla, 2022

Tribunal Secretary.

To my family

To my teachers

Acknownledgements

I wanted to thank my family and friends for the support they have given me throughout my studies. I would also like to thank the professors I have had, especially Hipólito, the tutor of this project who has guided and taught me in the process of the work.

Roberto Lama Rodríguez

Sevilla, 2022

Abstract

The main purpose of this project is to create a python model of the SX1257 transceiver by Semtech and everything has been prepared in order to be able to compare the model with the real component in the near future. The ultimate goal is to have a simulation model that can be used to develop FPGA SDR designs, for example by cosimulating the HDL code with the python model using the cocotb cosimulation framework.

Table of contents

Acknownledgements **viii**

Abstract **x**

Table of contents **xi**

List of tables **xiii**

List of figures **xv**

List of code snippets **xviii**

Acronyms and definitions **xx**

1 Introduction **22**
1.1 Software Defined Radio *22*

2 Document Structure **24**
2.1 Project scope *24*
2.2 Description of the SX1257 transceiver *24*
2.3 Python model *24*
2.4 RF Loopback *24*
2.5 How to use the python model *24*
Bibliography *24*
Annex *25*

3 Description of the SX1257 transceiver **26**
3.1 General description *26*
Description of the main subsystems *27*
3.2.1 Transmitter 27
3.2.2 Receiver 30

4 Python model **32**
4.1 Python model project structure *32*
4.2 Transmitter *34*
4.2.1 Digital-to-Analog Converters 34
4.2.2 TX I / Q Channel Filters 40
4.2.3 TX I / Q Up-Conversion Mixers 41
4.2.4 RF Amplifier 46
4.2 Receiver *49*
4.2.1 Low Noise Amplifier 49
4.4.2 I/Q Downconversion Quadrature Mixer 51
4.4.3 Baseband Analog Filters and Amplifiers 52
4.4.4 Downsampling 54
4.4.5 Analog-to-Digital Converters 54
4.5 Control registers and interface *56*
4.5.1 How does registerMapSerializer work? 56
4.5.2 Register map used for memory examples 57

5 RF Loopback **63**

6 How to use the python model **67**
6.1 Transmit *67*

6.2 Receive *68*
6.3 Examples *68*

7 Conclusions and future work **69**

References **70**

Annex **72**
1. Dealing with the analog world from a digital point of view *72*
2. Transmitting longer signals *73*
2.1 Reducing memory consumption 73
2.2 Splitting signals into pieces 73

List of tables

Table 1. TxDacBw register 8

Table 2. TxAnaBw register 9

Table 3. RegFrfTx register 10

Table 4. RegRxAnaGain register 11

Table 5. RegRxBw register 12

List of figures

Figure 1. Typical software-defined radio architecture 1
Figure 2. SX1257 Block Diagram 7
Figure 3. SX1257 Transmitter Analog Front-End Block Diagram 8
Figure 4. SX1257 Receiver Analog Front-End Block Diagram 11
Figure 5. Main class diagram 14
Figure 6. Fourier transform of kaiser window 16
Figure 7. FIR-DAC Normalized Magnitude Response with fs=32MHz and N=32 (datasheet) 17
Figure 8. FIR-DAC Normalized Magnitude Response with fs=32MHz and N=32 (generated in python) 19
Figure 9. Transmitter input 20
Figure 10. Delta-sigma DAC x[n]: input and x(t) output 21
Figure 11. Delta-sigma DAC x[n]: input and x(t) output (zoom) 21
Figure 12. Lowpass Filter Frequency Response 22
Figure 13. Input signal and filtered signa lof the lowpass channel filter 23
Figure 14. Butterworth filter output 23
Figure 15. I-Mixer Oscillator (zoom) 24
Figure 16. Q-Mixer Oscillator (zoom) 25
Figure 17. Mixer Input 25
Figure 18. Mixer Input resampled (Linear method) 26
Figure 19. Mixer Input resampled (Fourier method) 27
Figure 20. Conversion Mixer Output 28
Figure 21. Conversion Mixer Output Frequency domain 28
Figure 22. RF Amplifier Module and Phase 30
Figure 23. RF Amplifier output 30
Figure 24. LNA Module and pase 32
Figure 25. LNA Output 32
Figure 26. I-Downconversion Mixer Output 33
Figure 27. Q-Downconversion Mixer Output 34
Figure 28. Lowpass filter frequency response 34
Figure 29. Butterworth filter output 35
Figure 30. I and Q filter output 35
Figure 31. Delta-Sigma ADC Output 37
Figure 32. Digital and RF Loop-Back paths 45
Figure 33. Transmitter input (and DAC output) 46

Figure 34. Transmitter output 46
Figure 35. Receiver output 47
Figure 36. Transmitter input and DAC output 47
Figure 37. Transmitter output 48
Figure 38. DAC output resampled 54
Figure 39. Measuring response of the Butterworth filter 56
Figure 40. UML Diagram of `Chunk_Processor` 56

List of CODE SNIPPETS

Code 1: Plot Fourier transform of kaiser window 16
Code 2: Freqz function 17
Code 3: Create the DAC filter 17
Code 4. Represent response to the impulse filter 18
Code 5: Plot FIR-DAC Normalized Magnitude Response 19
Code 6. Creating the transmitter signal input 20
Code 7. Filtering the input using the DAC filter 20
Code 8. Create the Butterworth Filter 22
Code 9. Linear resampler 26
Code 10. Configure the amplifier 29
Code 11. Calculate output gain og the amplifier 29
Code 12. Power output of the amplifier 29
Code 13. Set LNA gain impedance 31
Code 14. Calculate the Frf 33
Code 15. Selecting the Baseband gain of the Butterworth filter 34
Code 16. Set the Baseband gaing of the Butterworth filter 35
Code 17. Downsampling method 36
Code 18. Delta-sigma ADC 36
Code 19. Create the map of variables 38
Code 20. Reducing the memory consumption 55
Code 21. Measuring response of the Butterworth filter 56

Acronyms and definitions

FPGA	Field-Programmable Gate Arrays
ISM	Industrial, scientific and medical
VHSIC	Very High Speed Integrated Circuit
HDL	Hardware Description Language
GHDL	GHDL is an open source VHDL simulator
VHDL	VHSIC Hardware Description Language
FOSS	Free and Open Source Software
SDR	Software Defined Radio
DSP	Digital Signal Processor
UML	Unified Modeling Language
IOBs	In/Out Blocks
CLBs	Configurable Logic Blocks

1 INTRODUCTION

The SX1257 is a low-cost transceiver for the 862 - 960 MHz band. This transceiver is gaining popularity in the free and open source software community for FPGAs design, as it operates in the ISM band, which does not require a license. The project objective is to make a model of the transceiver it in python so it can be used in python-VHDL cosimulations by means of the cocotb tool (coroutine cosimulation testbench) and the GHDL simulator, which are both open-source tools. In this way, the community of developers will be able to have a model of the transceiver, to simulate signal processing HDL designs and to be able to debug them with much greater visibility with respect to directly performing tests on a hardware prototype.

1.1 Software Defined Radio

In their beginnings (around 1900), radio systems were fully analog systems. This caused many problems because two different radio systems could not communicate with each other if they were not designed for it due to them being using different modulations and changing the mixers and filters in an analog system is not an easy task. One way of solving this problem is using Software defined radio (SDR). This means that the transmitter modulation is generated or defined by a computer and the receiver also uses a computer to recover the signal.

The main goal of SDR is to replace as many analog components as possible. There are many benefits of using SDR, including:

- For subscribers – easier international roaming
- For mobile network operators – provide added-value services
- For handset and base-station manufacturers – increased production flexibility

The advantages of SDR are flexibility and ease of adaptation. Any aspect of a program that implements radio functions can be easily changed.

The new digital and software radios do almost all the work of signal processing, such as channel selection, tuning and demodulation in the digital domain.

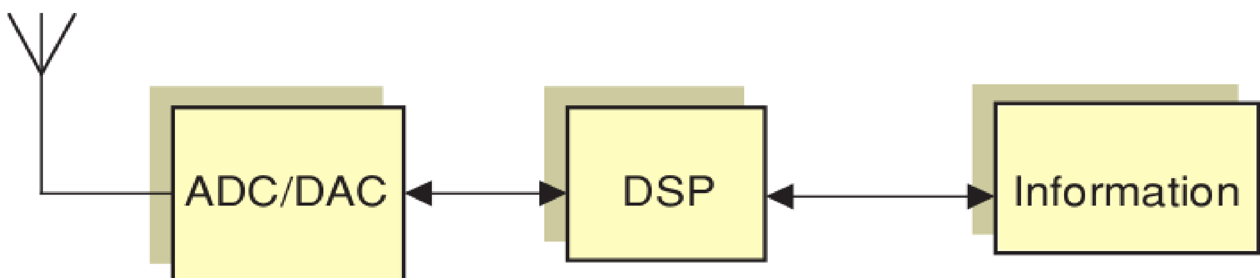

Figure 1. Typical software-defined radio architecture. Source: [6]

Digital signal processors (DSPs) are increasingly being used to implement SDR because they are used for detection, equalization, demodulation, frequency synthesis and channel filtering. The main function that DSPs apply is the fourier transform, which allows working in the frequency domain.

The radio community has realized that the more basic functions are managed by software, the more flexibility is achieved. Furthermore, industrial leaders such as Nokia, Toshiba and Motorola are committed strongly to this technology.

2 DOCUMENT STRUCTURE

2.1 Project scope

Firstly, the document begins with an introduction discussing the scope of the project and continues with the structure of the document and a brief description of SDR, FPGA and co-processing.

2.2 Description of the SX1257 transceiver

Secondly, the transceiver blocks are described, starting with the transmitter and continuing with the receiver.

Afterwards, it is explained how the model has been implemented in python, detailing each of the blocks and comparing them with the information extracted from the datasheet.

2.3 Python model

In this section all the blocks of the python model are explained, divided in the two main subsystems: transmitter and receiver.

A number of samples are then passed to the transmitter to check the outputs of each block. The same is then done with the receiver, using the transmitter output as input.

2.4 RF Loopback

The output of the receiver is given as input to the transmitter and viceversa, to show that the model works properly as expected.

2.5 How to use the python model

Finally instructions are given on how to use the python model.

Bibliography

[1] Semtech Corporation. SX1257 Low Power Digital I and Q RF Multi-PHY Mode Transceiver Datasheet [online]. Rev 1.2. [March 2018] Available at: https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/44000000MDmO/OfVC_rbxi4JjkT4hLzU1kq4gOXb4POLNRprWlqxRIZs

[2] NumPy community. NumPy User Guide [online]. Release 1.22.0.: January 14, 2022. Available at: https://numpy.org/doc/stable/

[3] Scipy community. Scipy Documentation [online] [consulted: 02 2022]. Release 1.22.0. Available at: http://scipy.github.io/devdocs/dev/

[4] Matplotlib development team. Pyplot tutorial [online] [consulted: 02 2022]. Available at: https://matplotlib.org/stable/tutorials/introductory/pyplot.html

[5] Harris, Fredric J. (Jan 1978). "On the use of Windows for Harmonic Analysis with the Discrete Fourier Transform" (PDF). Proceedings of the IEEE.

[6] Sadiku, M., & Akujuobi, C. (2004). Software-defined radio: a brief overview. IEEE Potentials, 23(4), 14–15. https://doi.org/10.1109/mp.2004.1343223

[7] Cai, X., Zhou, M., & Huang, X. (2017). Model-Based Design for Software Defined Radio on an FPGA. IEEE Access, 5, 8276–8283. https://doi.org/10.1109/access.2017.2692764

[8] Christie, W. M. (1986). Towards a unified model of language description. Language Sciences, 8(2), 177–191. https://doi.org/10.1016/s0388-0001(86)80015-8

Annex

1. Dealing with the analog world from a digital point of view
2. Transmitting longer signals

3 DESCRIPTION OF THE SX1257 TRANSCEIVER

3.1 General description

The SX1257 transceiver can be divided into two main subsystems, the transmitter and the receiver. It is also important to note that the register map has also been modelled. That is, the transmitter and receiver operation are affected by the values stored in this register map.

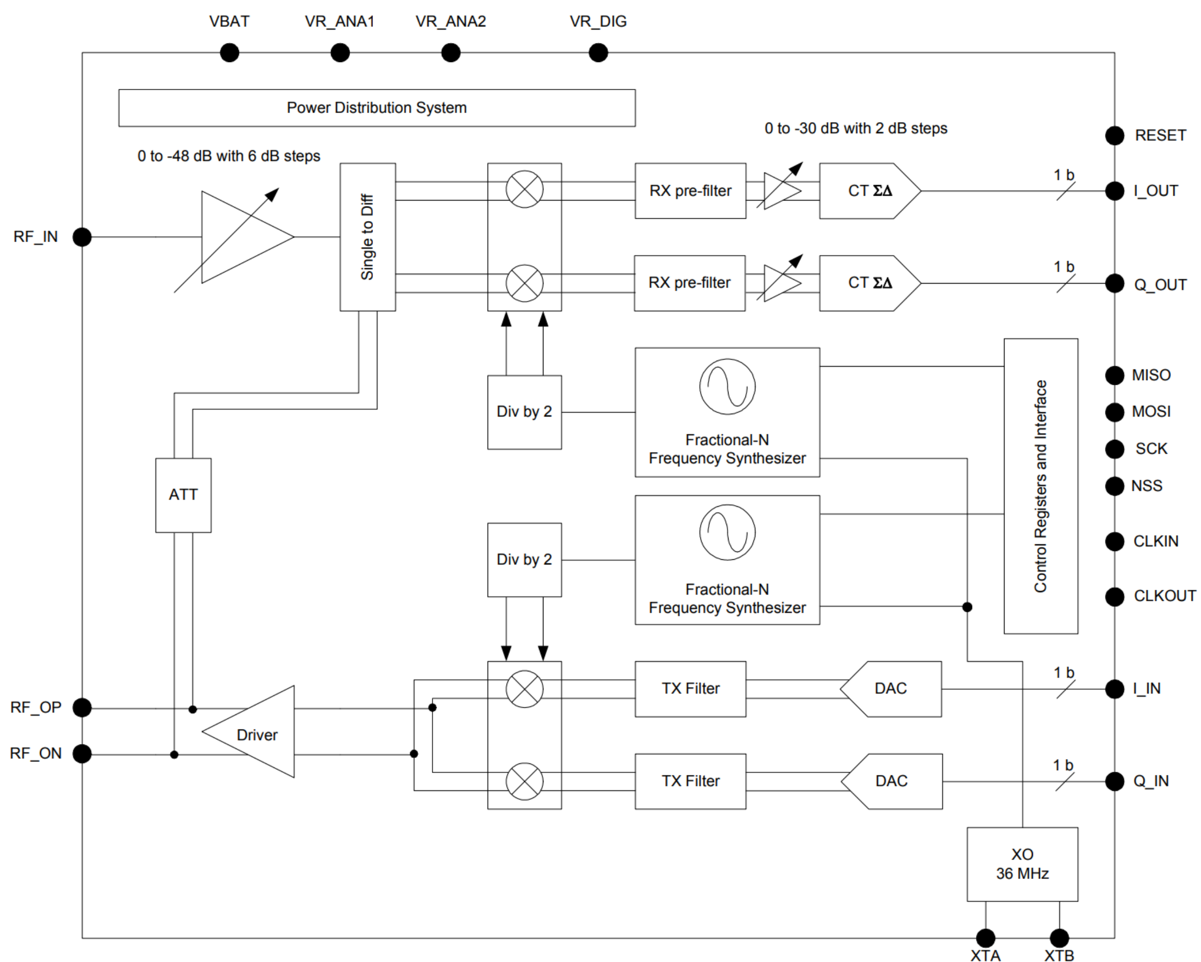

Figure 2. SX1257 Block Diagram. Source : [Ref 1]

Description of the main subsystems

3.2.1 Transmitter

The transmitter is made up of 4 main blocks : The digital-analog converter, the TX channel filter, the Frequency mixer and the RF amplifier. All these blocks have two inputs and two outputs, one for the I channel and another one for the Q channel.

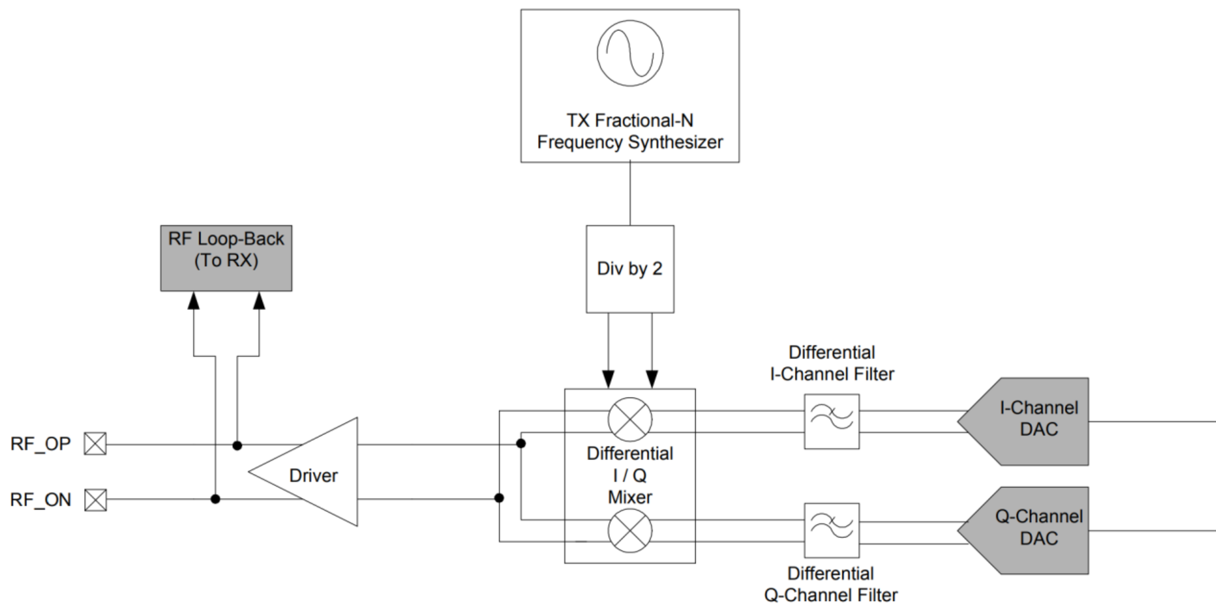

Figure 3.: SX1257 Transmitter Analog Front-End Block Diagram. Source : [Ref 1]

- Digital to analog converter

The DAC converter consists of an FIR low-pass filter that accepts 1 bit I and Q at 32Msample/s or 36Msample/s bit stream, depending on the frequency of the oscillator. The number of specific taps and the bandwidth can be changed via the `TxDacBw` register.

TxDacBw [Dec]	TxDacBw [Bin]	No. DAC-FIR Taps	SSB Filter BW [kHz]
0	000	24	
1	001	32	450
2	010	40	
3	011	48	
4	100	56	
5	101	64	290

Table 2. `TxDacBw` register. Source : [Ref 1]

The number of taps follows this formula:

$$DACnumtaps = 24 + 8 \cdot TxDacBw$$

The bandwidth chosen in the implementation is always 500kHz for each channel, except if `TxDacBw` is equal to 5, in this case it is 290kHz and if `TxDacBw` is equal to 1 the bandwidth is 450kHz.

- TX Channel filters

Analog I and Q signals are filtered by TX channel filters. They smooth the analog waveforms and remove quantization noise generated by the FIR DACs. The TX Channel filters are unity gain third-order low-pass Butterworth types with programmable bandwidth configured by changing the value in the `TxAnaBw` register. The 3 dB BW of the analog TX filter BW can be calculated using the equation:

$$BW(3dB) = \frac{17.15}{41 - RegTxBWAna(4,0)}$$

RegTxBw (0x0A)	7	-	r	0	unused
	6-5	TxPllBw	rw	11	TX PLL bandwidth, programmable: 00 = 75 kHz 01 = 150 kHz 10 = 225 kHz 11 = 300 kHz
	4-0	TxAnaBw	rw	0000	TX analog filter bandwidth, programmable: $Bandwidth = \frac{17.5}{2 \times (41 - Int(TxBw(4,0)))}$ in MHz

Table 1. `TxAnaBw` register. Source: [Ref 1]

- TX I/Q Up-Conversion mixer

The conversion mixer mixes the signal output I and Q of the TX channel filters with the mixer signal in order to convert it to the RF carrier frequency, and then sum the real and imaginary parts in the following way. The output signal of the I/Q conversion mixer is s(t):

$$s(t) = I(t) \cdot cos(2 \cdot \pi \cdot f_{RF} \cdot t) - j \cdot Q(t) \cdot sin(2 \cdot \pi \cdot f_{RF} \cdot t)$$

In addition this block includes a programmable gain that can be modified using the `RegTxGain` register.

In order to calculate the carrier frequency, the following formula is used:

$$F_{STEP} = \frac{FX_{OSC}}{2^{19}}$$

$$f_{RF} = F_{STEP} \cdot f_{RFXX}(23,0)$$

RegFrfTxMsb (0x04)	7-0	FrfTx(23:16)	rw	0xCB	MSB of the TX carrier frequency
RefFrfTxMid (0x05)	7-0	FrfTx(15:8)	rw	0x55	Middle byte of the TX carrier frequency
RefFrfTxLsb (0x06)	7-0	FrfTx(7:0)	rw	0x55	LSB of the TX carrier frequency $F_{RF} = F_{STEP} \times F_{rfxx}(23, 0)$ With a 36 MHz XO, value 0xCB5555 FrfTx = 915 MHz and frequency resolution = 68.66455 Hz The TX RF frequency is updated only under the following conditions: - 36 MHz XO is active - RefFrfTxLsb is written - when exiting SLEEP mode

Table 3. `RegFrfTx` register. Source: [Ref 1]

- RF amplifier:

  The RF amplifier takes, as its input, the output of the conversion mixer. It can provide a peak of +5dBm and its optimum load impedance is 100 ohms.

3.2.2 Receiver

The receiver is made up of 4 main blocks: Low Noise Amplifier (LNA), the downconversion mixer, the channel filters and the analog to digital converter. All these blocks have two inputs and two outputs, one for the I channel and another one for the Q channel.

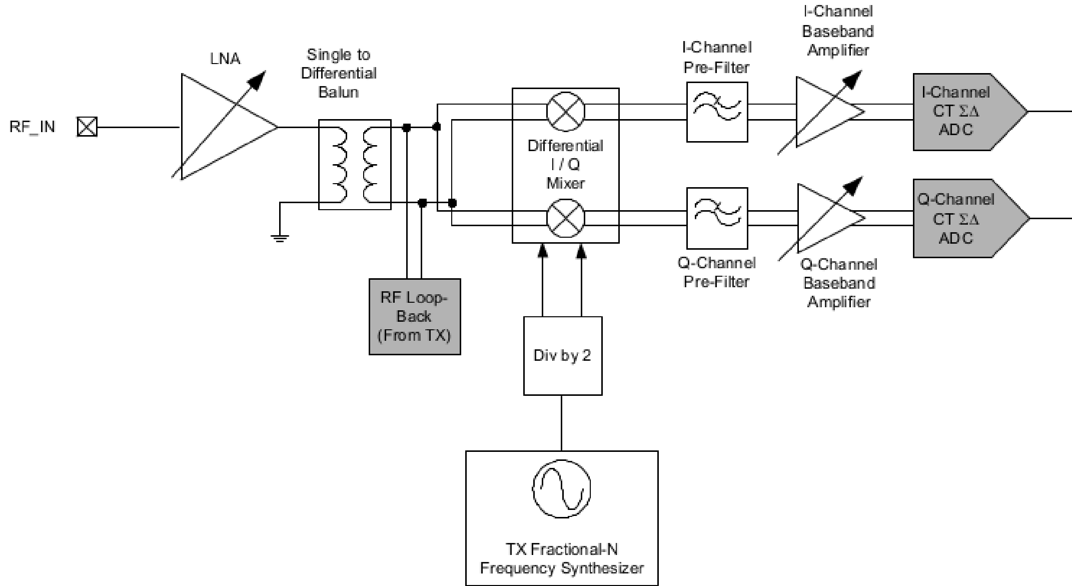

Figure 4. SX1257 Receiver Analog Front-End Block Diagram [Ref 1]

- Low Noise Amplifier (LNA)

The Low Noise Amplifier has a flat characteristic over the whole frequency range. Its expected input impedance is 50 ohms or 200 ohms, that can be selected by setting the `LnaZin` bit of the `RxAnaGain` register. Also at its output has a single to differential buffer to improve the second order linearity of the receiver.

The LNA gain can be programmed by modifying the `RxLnaGain` variable that is located in the `RegRxAnaGain` register.

Register	Bits	Variable	Mode	Default	Description
RegRxAnaGain (0x0C)	7-5	RxLnaGain	rw	001	RX LNA gain setting: 000 = not used 001 = G1 = highest gain power - 0 dB 010 = G2 = highest gain power - 6 dB 011 = G3 = highest gain power - 12 dB 100 = G4 = highest gain power - 24 dB 101 = G5 = highest gain power - 36 dB 110 = G6 = highest gain power - 48 dB 111 = not used
	4-1	RxBaseband-Gain	rw	1111	RX Baseband amplifier gain, programmable: $Gain = Gain + 2 \times Int(RxBaseBandGain(4, 1))$
	1	LnaZin	rw	1	LNA input impedance 0 = 50 Ω 1 = 200 Ω

Table 4. `RegRxAnaGain` register. Source: [Ref 1]

- RX I/Q Down-Conversion Quadrature mixer

The downconversion mixer takes the LNA output and downconverts it to the base-band. The downconversion mixer offers high IIP2 and IIP3 responses. IIP2 is the input second order intercept point and IIP3 is the input third order intercept point.

The output signals of the I/Q downconversion mixer are defined by the following formula:

$$S_I(t) = I(t) \cdot cos(2 \cdot \pi \cdot w \cdot f_c \cdot t)$$

$$S_Q(t) = -j \cdot Q(t) \cdot \sin(2 \cdot \pi \cdot w \cdot f_c \cdot t)$$

- Baseband Analog Filters and Amplifiers

The I and Q baseband mixer signal outputs are filtered by a programmable 1st order low-pass filter and then the output is sent to the input of programmable linear baseband amplifiers. The bandwidth of the filters can be programmed between 250 kHz and 750 kHz. This filtering improves the selectivity of the receiver for complex modulation schemes, including OFDM (Orthogonal frequency-division multiplexing).

The amplifier stage gain offers 32 dB gain range, that can be programmed from -24 dB to +6 dB via configuration register `RegRxAnaGain` (previously showed in the memory) while the analog filter bandwidth is programmed via the configuration register `RegRxBw` (the table below).

RegRxBw (0x0D)	7-5	RxAdcBw	rw	111	RX ADC BW, programmable: 010 = 100 kHz < RxAdcBw < 200 kHz 101 = 200 kHz < RxAdcBw < 400 kHz 111 = 400kHz < RxAdcBw
	4-2	RxAdcTrim	rw	111	RX ADC trim 32 MHz reference crystal: RxAdcTrim = 110 36 MHz reference crystal: RxAdcTrim = 101
	1-0	RxBase-bandBw	rw	01	Bandwidth of RX analog roofing filter, programmable: 00 = 750 kHz 01 = 500 kHz 10 = 375 kHz 11 = 250 kHz

Table 5. `RegRxBw` register. Source: [Ref 1]

- Analog to digital converter

The Analog to Digital Converter consists of a 5th order sigma-delta modulator ADC that samples and digitizes the signal from the baseband analog filter and amplifier output. When the signal is processed by the ADC, its output allows a resolution of 13 bits after decimation and filtering, with a 1MHz bandwidth. The bitstream speed is 32/36 MSamples/s, depending on the clock frequency.

4 PYTHON MODEL

Python has been chosen as a programming language to model the transceiver due to its versatility and modularity. It is a language that allows object-oriented programming and has a large number of modules that are useful to develop the model. Among them:

- `numpy`: A module that allows working with multidimensional information arrays (discretized signals are arrays, and time vectors are arrays).
- `scipy`: Provides algorithms for optimization, integration, interpolation...
- `matplotlib.pyplot` is a collection of functions in python that make the library work like MATLAB. It is mainly oriented to the graphic representation of figures.

4.1 Python model project structure

The main folder contains two scripts that are essential:

`main.py`: Contains the main project class that configures the SX1257 from the arguments passed through the command line and the register map found in the `data` folder.

`SX1257.py`: Contains the SX1257 class, that is responsible for managing the transmitter and receiver blocks

The `data` folder contains the register map in json format: `registerMap.json`.

The `input` folder contains some text files that stores signals, created to test the transmitter and the receiver.

The `tools` folder contains the different modules of the simulator and some useful scripts:

- `amplifier/amp.py:` The model of the amplifier block.
- `butterworth_filter/bw_filter.py`: The Butterworth filter block model.
- `conversion_mixers/conv_mixer.py`: The model of the conversion mixer block.
- deltasigma_block
    - `DAC/delta_sigma_DAC.py`: The DAC block model.
    - `ADC/delta_sigma_ADC.py`: The ADC block model.
- `Resampler/resampler.py`: The resampler model.

An Unified Modeling Language (UML) diagram is created in order to explain the interaction between the classes of our model. The UML is used to create the project model. Is the most well-known and currently used software system modeling language; it is supported by the Object Management Group. It is a graphical language for visualizing, specifying, building and documenting a system. Source [8].

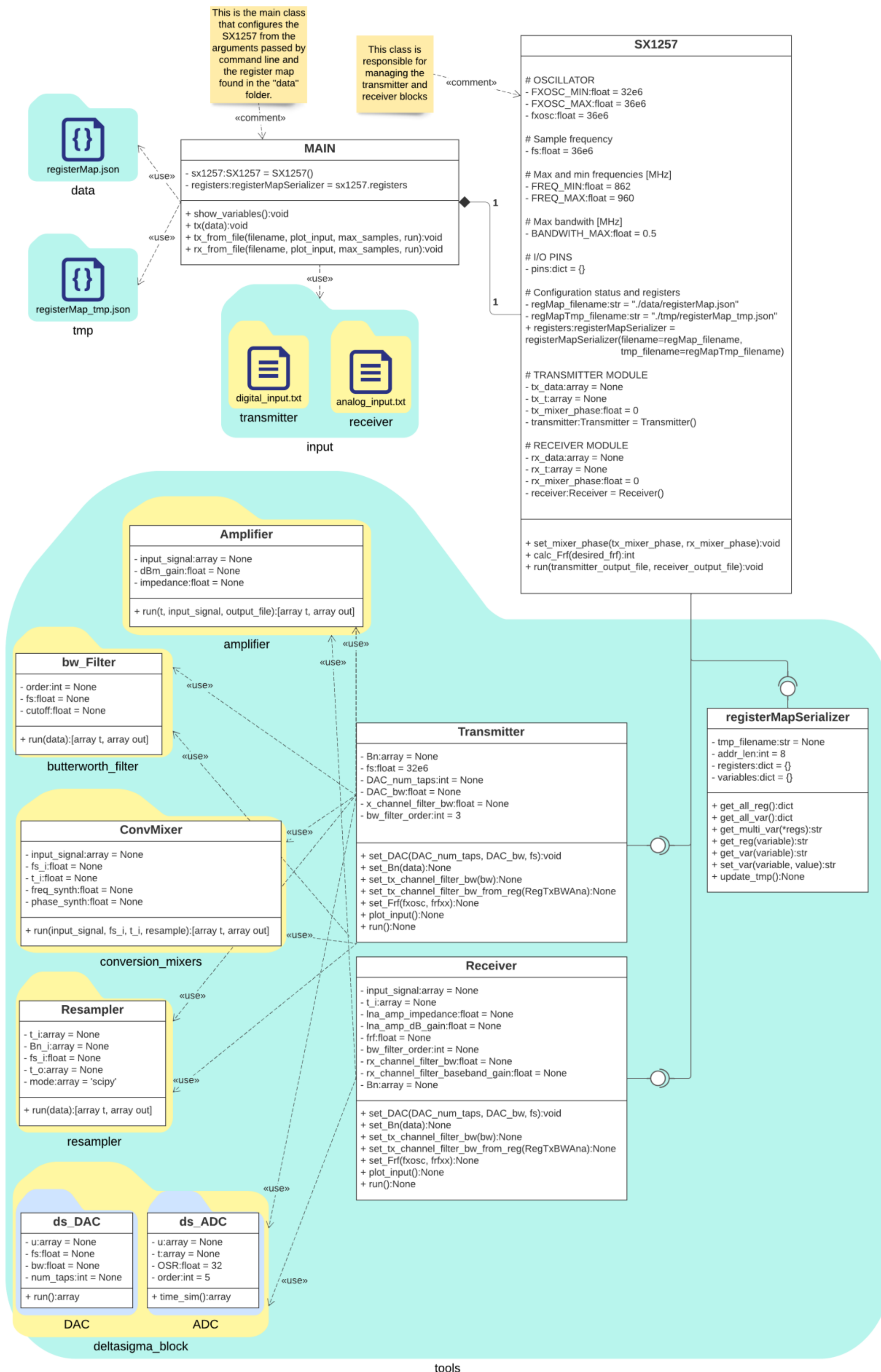

Figure 5. Main class diagram

4.2 Transmitter

4.2.1 Digital-to-Analog Converters

To implement the DAC the `firwin` function of the `scipy` library is used. This library allows the design of FIR filters using the window method. The parameters that are passed to the function are:

- *numtaps*: Number of filter coefficients, that is, the filter order plus one.
- *cutoff*: Filter cut-off frequency expressed in the same order as the sampling frequency.
- *window*: Computes the coefficients of a finite impulse response filter. The filter will have linear phase.

The window selected is the kaiser window, since, from the available windows in `scipy`, it is the one whose frequency response is the most similar to the one that appears in the datasheet. The following code can be executed in order to plot the Kaiser Window FFT:

```
win = windows.kaiser(M=51, beta= 0.001, sym=True)
bins = np.arange(0, len(win))

win_fft = sc.fft.fft(win, 2048) / (len(win2)/2.0)
freq = np.linspace(-0.5, 0.5, len(win_fft))

plt.figure()
plt.title('Kaiser window FFT')
plt.ylabel('Normalized Magnitude (dB)')
plt.xlabel('DFT bins')
plt.plot(freq2, 20*np.log10(np.abs(sc.fft.fftshift(win2_fft /
abs(win2_fft).max()))), 'r', label='M=%d, beta=%f'%(M_show, beta_show))
plt.legend(loc='best')
plt.grid()
```

Code 1. Plot Fourier transform of kaiser window *tools/deltasigma_block/DAC/delta_sigma_DAC.py*

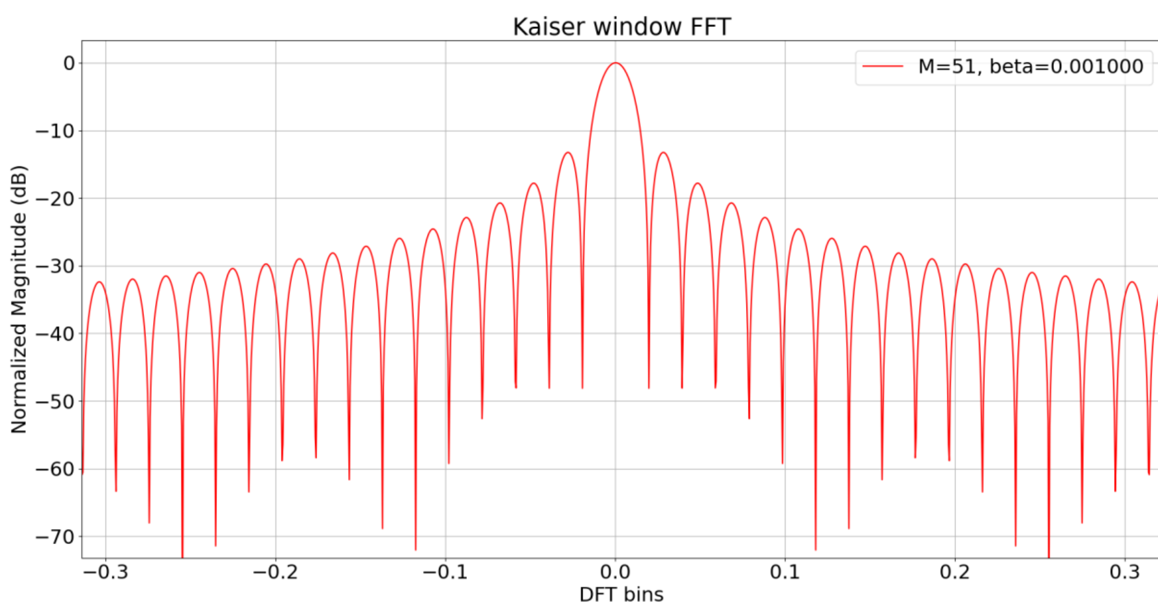

Figure 6. Fourier transform of kaiser window

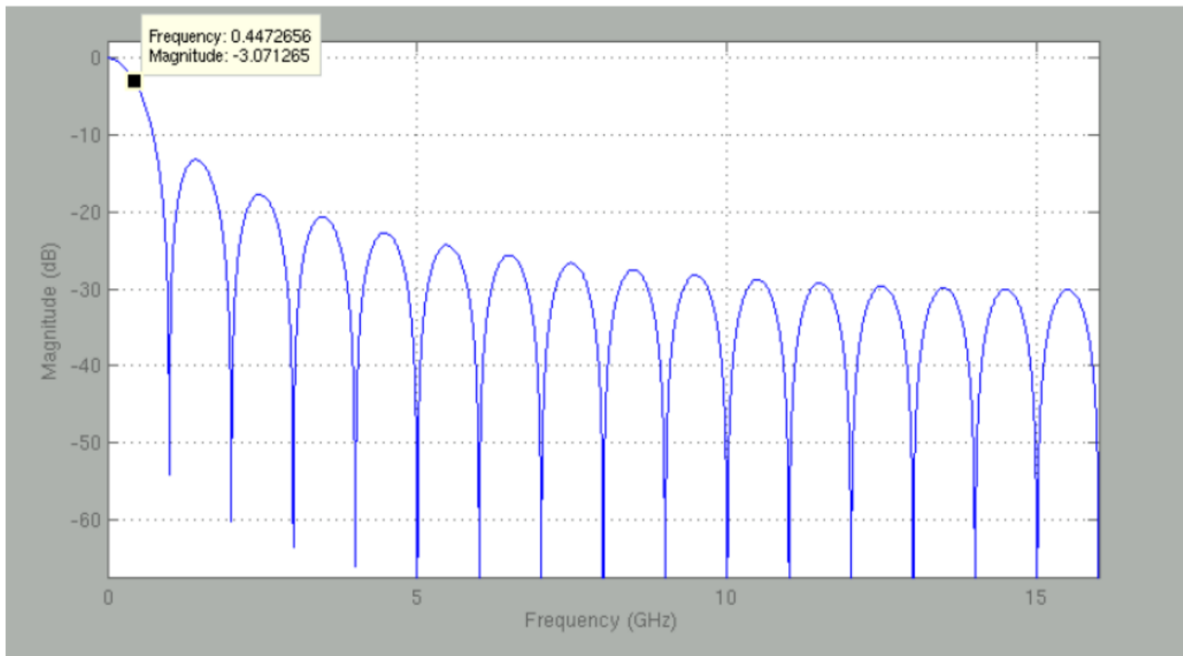

Figure 7. FIR-DAC Normalized Magnitude Response with fs = 32 MHz and N = 32. Source: [Ref 1]

The length of the chosen window is 1 and the `beta` parameter is also 1. The `sym` parameter specifies that the window is symmetric. With those parameters the response to the FIR-DAC Normalized Magnitude Response with fs=32MHz and N=32 is as similar as possible to that shown on the datasheet.

```
win = sc.signal.windows.kaiser(M=1, beta=1, sym=True)
self.fir_filter = sc.signal.firwin(numtaps = self.num_taps,
                                   cutoff = self.bw,
                                   window=win)
```

Code 2. Create the DAC filter *tools/deltasigma_block/DAC/delta_sigma_DAC.py*

To represent the response to the impulse response is used the freqz function of scipy:

```
[w, h] = sc.signal.freqz(self.fir_filter, a=1, fs=self.fs)
```

Code 3. Represent response to the impulse filter *tools/deltasigma_block/DAC/delta_sigma_DAC.py*

Details of its implementation can be seen in the "signal/_filter_design.py" file in the github repository: https://github.com/scipy/scipy/blob/main/scipy/signal/_filter_design.py

Here is an excerpt from the freqz function code:

```
if _is_int_type(worN):
        N = operator.index(worN)
        del worN
        if N < 0:
            raise ValueError('worN must be nonnegative, got %s' % (N,))
        lastpoint = 2 * pi if whole else pi
        # if include_nyquist is true and whole is false, w should include
end point
        w = np.linspace(0, lastpoint, N, endpoint=include_nyquist and not
whole)
        if (a.size == 1 and N >= b.shape[0] and
                sp_fft.next_fast_len(N) == N and
                (b.ndim == 1 or (b.shape[-1] == 1))):
```

```
            # if N is fast, 2 * N will be fast, too, so no need to check
            n_fft = N if whole else N * 2
            if np.isrealobj(b) and np.isrealobj(a):
                fft_func = sp_fft.rfft
            else:
                fft_func = sp_fft.fft
            h = fft_func(b, n=n_fft, axis=0)[:N]
            h /= a
            if fft_func is sp_fft.rfft and whole:
                # exclude DC and maybe Nyquist (no need to use axis_re-
verse
                # here because we can build reversal with the truncation)
                stop = -1 if n_fft % 2 == 1 else -2
                h_flip = slice(stop, 0, -1)
                h = np.concatenate((h, h[h_flip].conj()))
            if b.ndim > 1:
                # Last axis of h has length 1, so drop it.
                h = h[..., 0]
                # Move the first axis of h to the end.
                h = np.moveaxis(h, 0, -1)
```

Code 4. Freqz function *scipy/_filter_design.py*. Source: [Ref 3.1]

Given a system defined by the following equation, this python function computes the frequency response of a digital filter. The equation is composed by the M-order numerator « b » and N-order denominator « a » of a digital filter:

$$H(e^{jw}) = \frac{B(e^{jw})}{A(e^{jw})} = \frac{b_0 + b_1 \cdot e^{-jw} + \cdots + b_M \cdot e^{-jwM}}{a_0 + a_1 \cdot e^{-jw} + \cdots + a_N \cdot e^{-jwN}}$$

Therefore, a FFT (Fast Fourier Transform) is carried out for its calculation, because in our case the following conditions are met:

1. An integer value is given for `worN` (by default 512).
2. `worN` is fast to compute via FFT (i.e., `next_fast_len(worN)` equals `worN`).
3. The denominator coefficients are a single value (`a.shape[0] == 1`).
4. `worN` is at least as long as the numerator coefficients (`worN >= b.shape[0]`).
5. If `b.ndim > 1`, then `b.shape[-1] == 1`.

Now we can visualize the normalized magnitude using pyplot given as input the output of the freqz function.

```
print("Calculate Filter Response...")
plt.figure()
plt.title('FIR DAC normalized magnitude response with fs=%dMHz, N=%d,
BW=%dkHz'%(self.fs/1e6, self.num_taps, self.bw/1e3))
plt.ylabel('Magnitude (dB)')
plt.xlabel('Frequency (GHz)')
plt.plot(w, 20*np.log10(np.abs(h)), 'b')
plt.grid()
```

Code 5. Plot FIR DAC normalized magnitude response *tools/deltasigma_block/DAC/delta_sigma_DAC.py*

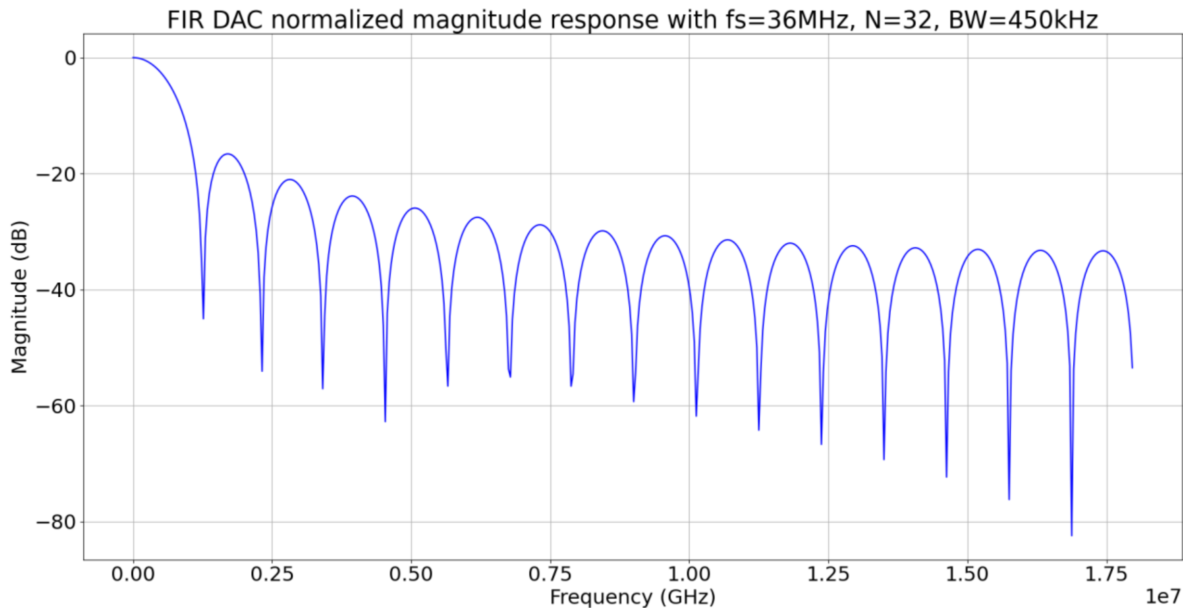

Figure 8. FIR DAC normalized magnitude response with fs=32MHz and N=32

The impulse response obtained is the same as that represented in the datasheet. The input signal is passed through the filter using the `lfilter` function. The filter is a direct form II transposed implementation of the standard difference equation.

- `b`: specifies the the numerator coefficient vector of the filter.
- `a`: the denominator coefficient vector
- `x`: is the input signal of the filter.

To display the behavior of the different blocks, the following entry is used, that has been done using a third order delta-sigma modulator (that is going to be explained when describing the receiver block in chapter Analog-to-Digital Converters, section 4.4.5).

```
# I/Q COMPONENT
OSR = 32 # Over Sampling Ratio
order = 3 # ADC order

# Input Signal
freq = 500e3
periods_to_show = 10
fs = 32e6

t = np.arange(0, periods_to_show/freq, 1/fs) # Time array
I = 0.5*np.sin(2*np.pi*freq*t)
Q = 0.5*np.cos(2*np.pi*freq*t)

samples_per_sec = 32e6
cte = len(t)/samples_per_sec

# Time array
t = np.arange(len(I))/len(I)*cte

myds_ADC_I = ds_ADC(OSR, order, t, I)
I_s = myds_ADC_I.time_sim()
```

```
myds_ADC_Q = ds_ADC(OSR, order, t, Q)
Q_s = myds_ADC_Q.time_sim()

output_file = '../../../input/transmitter/digital_input_or-
der3_IQ_fs_32M_bw_%dk_%dT_with_time.txt'%(int(freq/1e3), periods_to_show)
print("OUTPUT FILE: ",output_file)
np.savetxt(output_file,
           np.column_stack([
               T, np.real(I_s), np.real(Q_s),]))
```

Code 6. Creating the transmitter signal input *tools/deltasigma_block/DAC/delta_sigma_DAC.py*

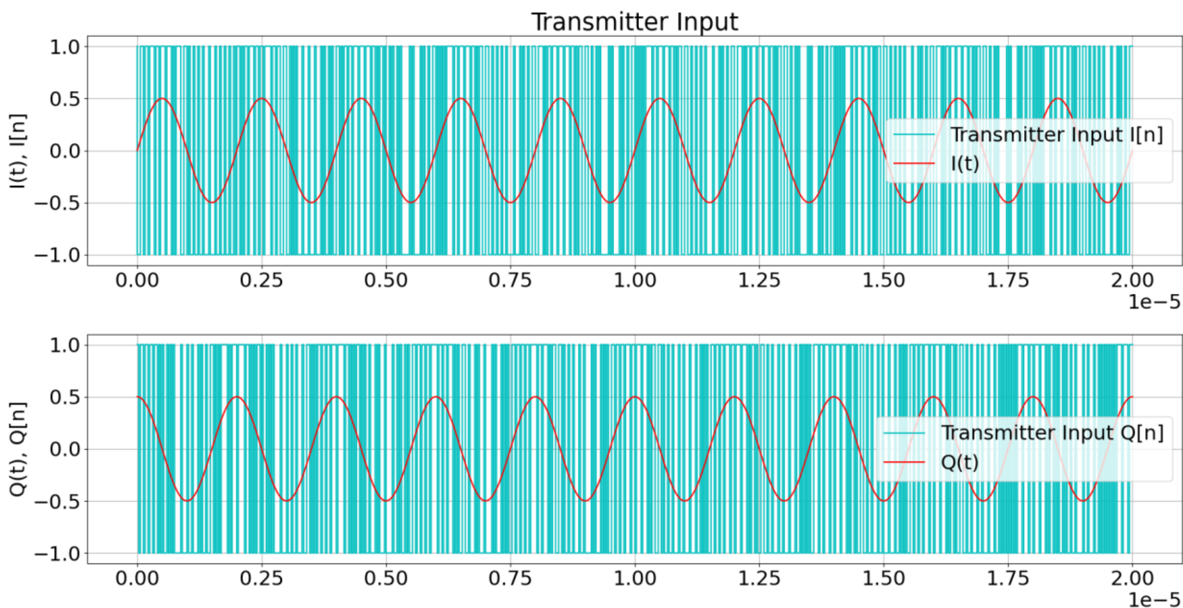

Figure 9. Transmitter input

Then using the previously designed filter, we filter the signal designed before:

```
self.xdigital = sc.signal.lfilter(b=self.fir_filter, a=1, x=self.u)
```

Code 7. Filtering the input using the DAC filter */tools/deltasigma_bock/delta_sigma_DAC.py*

In the following representation, the input of the signal x(n) and the output of the filter x(t) are displayed.

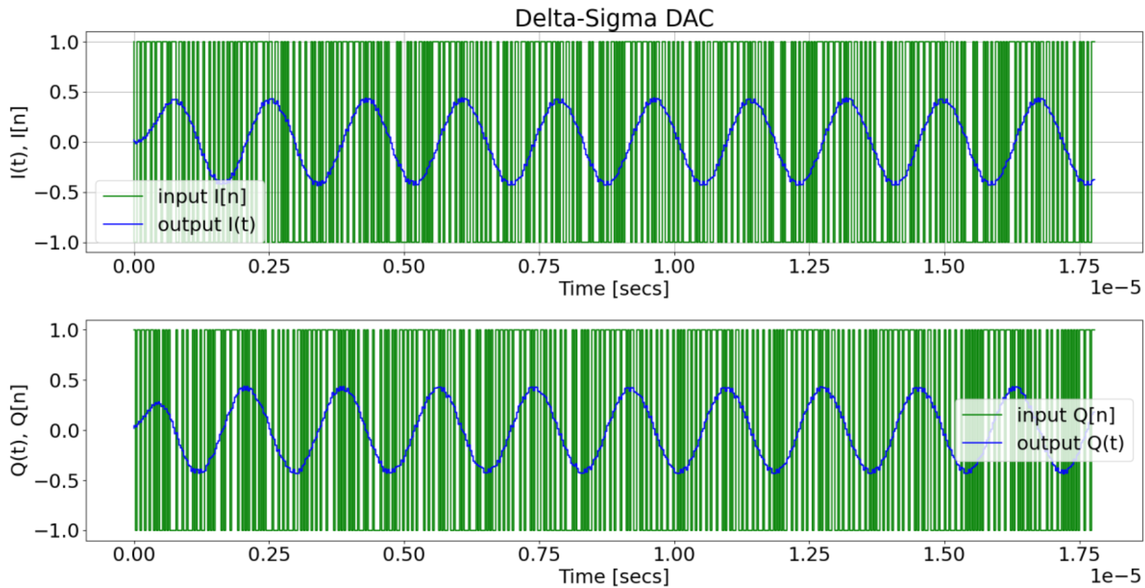

Figure 10. Delta-sigma DAC x[n]: input and x(t) output

It might seem that the model is not transmitting the signals that we initially specified: channel I a sine and channel Q a cosine, but it this happens because the first period is still on a transient phase. If we move and extend the signal to start at 2us it is observed that the desired signals are being transmitted.

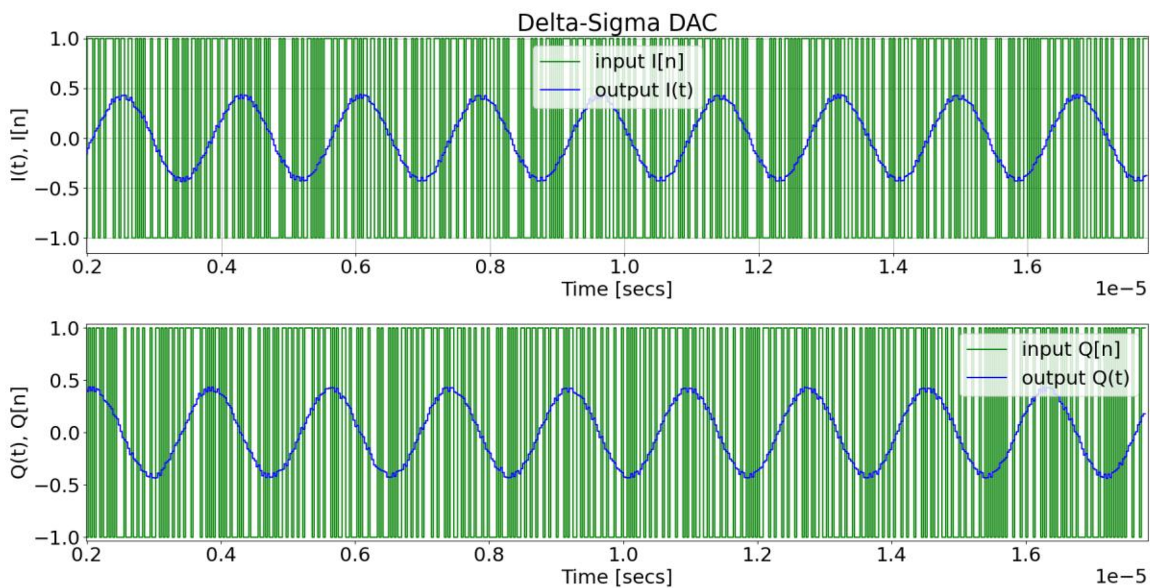

Figure 11. Delta-sigma DAC x[n]: input and x(t) output (zoom)

The actual circuit, the filter output will be differential, which makes sense to maintain low the signal-to-noise ratio (SNR). Modelling this it is not necessary, as the reverse polarity signal would have the same magnitude as y(t) but with the polarity reversed.

The code of the implementation of the delta-sigma DAC is available in the following path: *tools/deltasigma_block/DAC/delta_sigma_DAC.py*.

4.2.2 TX I / Q Channel Filters

To implement the Butterworth filter is used the butter function of the aforementioned scipy library. The parameters of the function are:

- Nint: The order of the filter.
- W: The critical frequency or frequencies (normalized).
- btype: {'lowpass', 'highpass', 'bandpass', 'bandstop'}. The type of filter. Default is 'lowpass'.
- analog: When True, return an analog filter, otherwise a digital filter is returned.

It returns numerator (b) and denominator (a) polynomials of the filter.

```
nyq = 0.5 * fs
normal_cutoff = cutoff / nyq
b, a = sc.signal.butter(order,
        normal_cutoff,
        btype='low',
        analog=False)
```

Code 8. Create the Butterworth Filter *tools/butterworth_filter/bw_filter.py*

The output of the DAC converter is given as input to TX I / Q Channel Filters. The frequency response of the filter is also represented in the next figure:

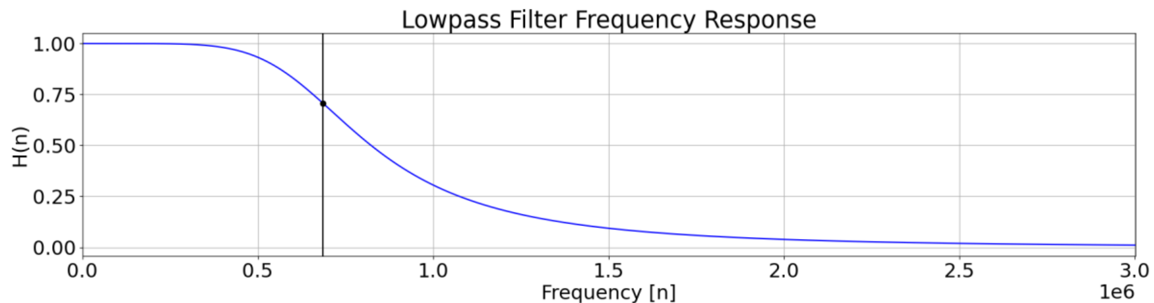

Figure 12. Lowpass Filter Frequency Response

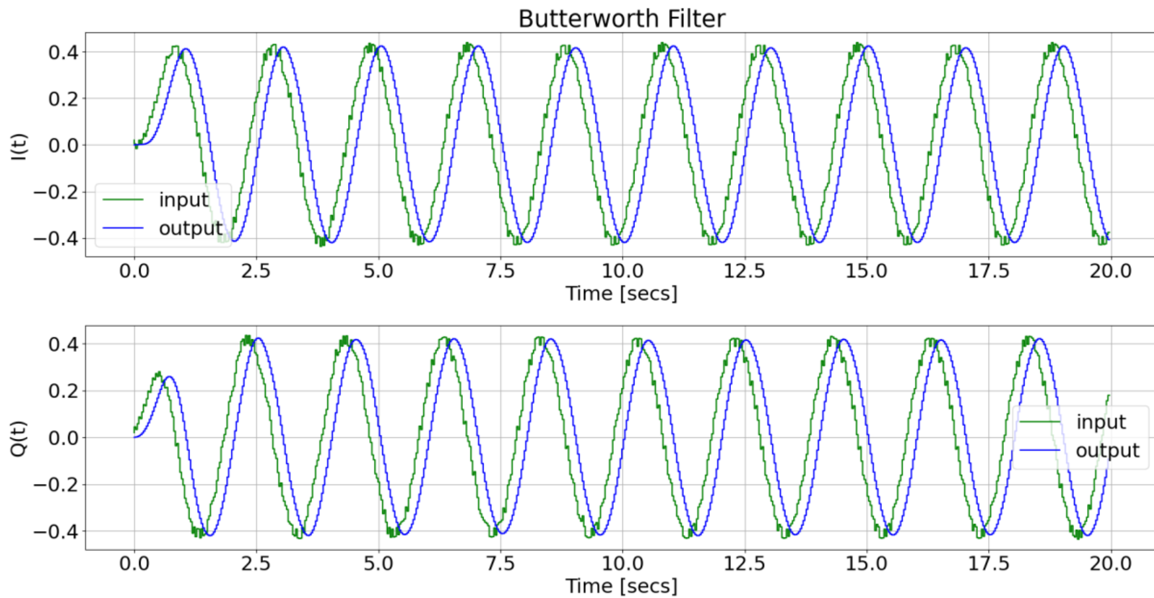

Figure 13. Input signal and filtered signal of the lowpass channel filter

To visualize better that the phase difference between both signals is $\frac{\pi}{2} rad$, the following plot is represented:

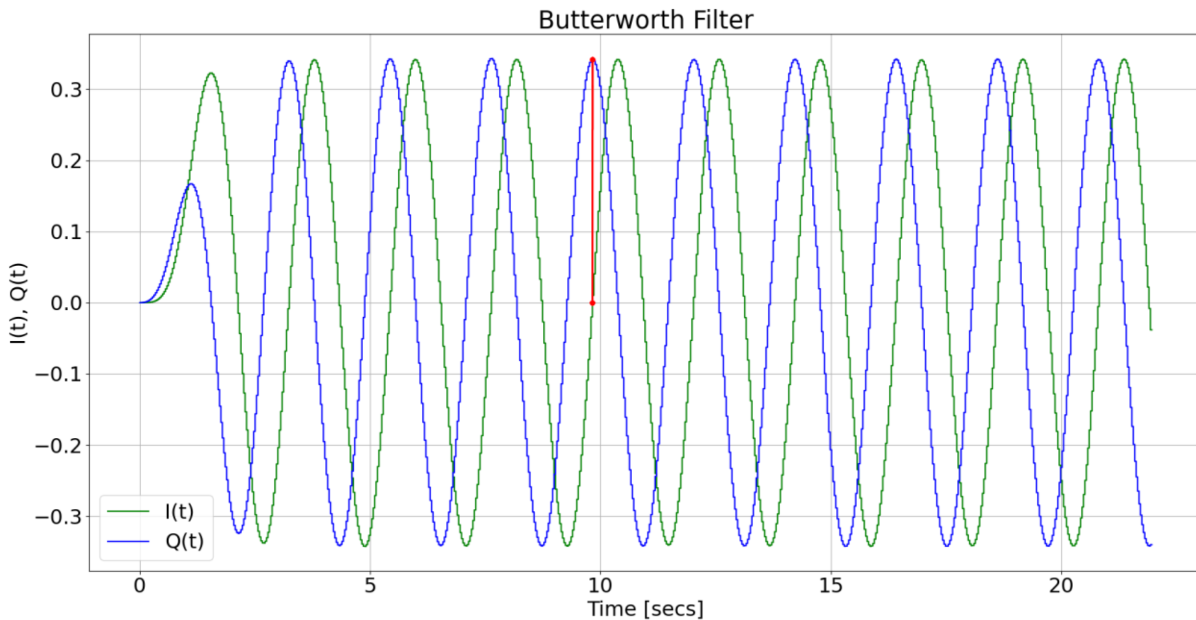

Figure 14. Butterworth filter output

The code of the implementation of the butterworth filter is available in the following path: */tools/butterworth_filter/bw_filter.py*

4.2.3 TX I / Q Up-Conversion Mixers

To implement the mixer first we need to create the carrier signal. In this case, we configured the registers to get a carrier frequency of 915MHz. As can be seen in Table 2-3: Power consumption specifications, the value of `fxosc` can vary between 32MHz and 36MHz, but it is typically worth 36MHz, so we select this value.

$$FX_{OSC} = 36MHz$$

$$F_{STEP} = \frac{FX_{OSC}}{2^{19}} = 68.6645507813$$

$$F_{RFTX}(23,0) = 13325653 \text{ (en base 10)}$$

$$F_{RF} = F_{STEP} \cdot F_{RFTX}(23,0) = 914999999.9999999Hz$$

$$F_{RF} \approx 915MHz$$

The mixer signal is very high frequency compared to the time we are representing. Obviously, it has to have the same size as the input signal. In the figure we couldn't appreciate the mixer signal, so in the following image can be seen more clearly after zooming in:

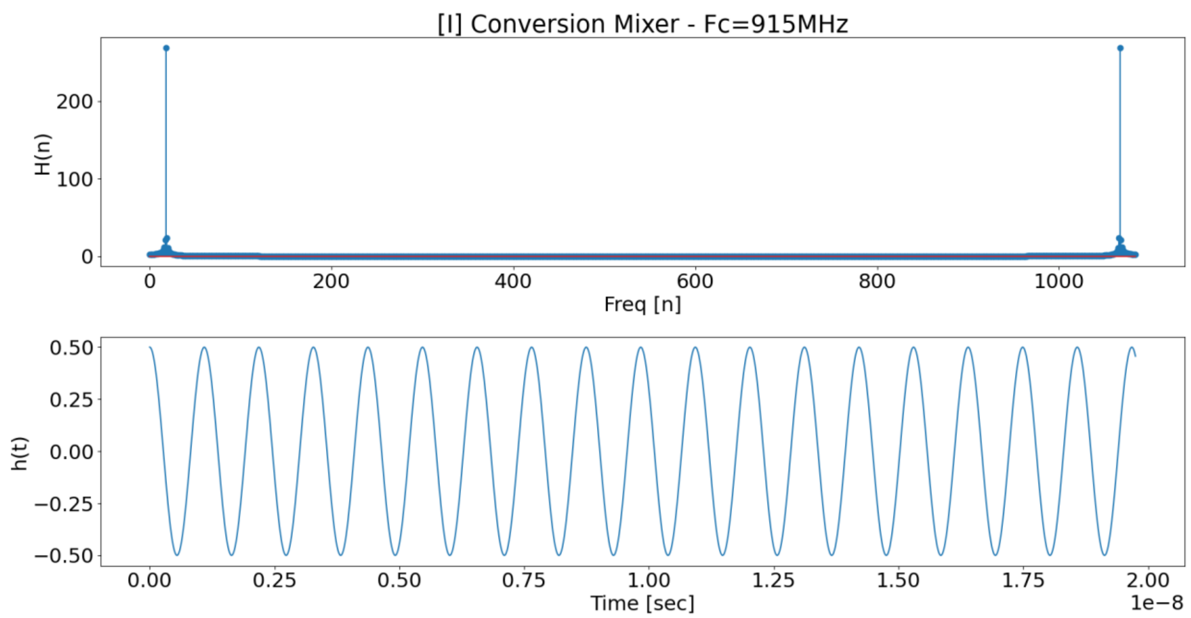

Figure 15. I-Mixer Oscillator (zoom)

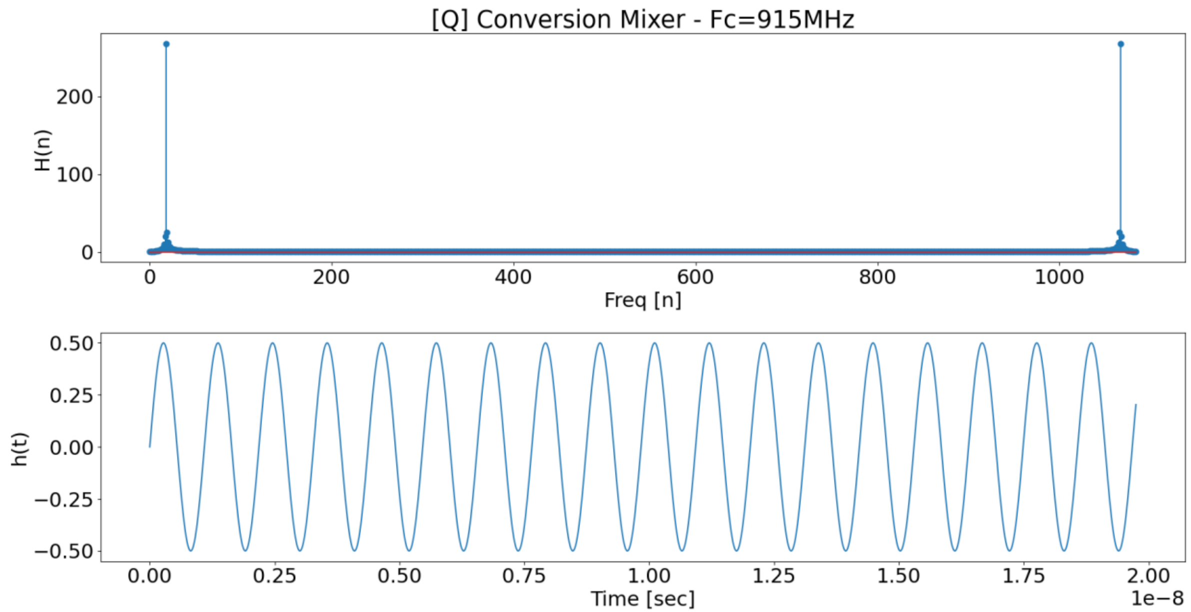

Figure 16. Q-Mixer Oscillator (zoom)

The input signal is sampled at 32 MHz and the mixer signal is sampled at 58.56 GHz (64 times the carrier frequency $> 2 \cdot \text{nyquist frequency}$). Therefore we proceed to perform a resample. This can be done in several ways.

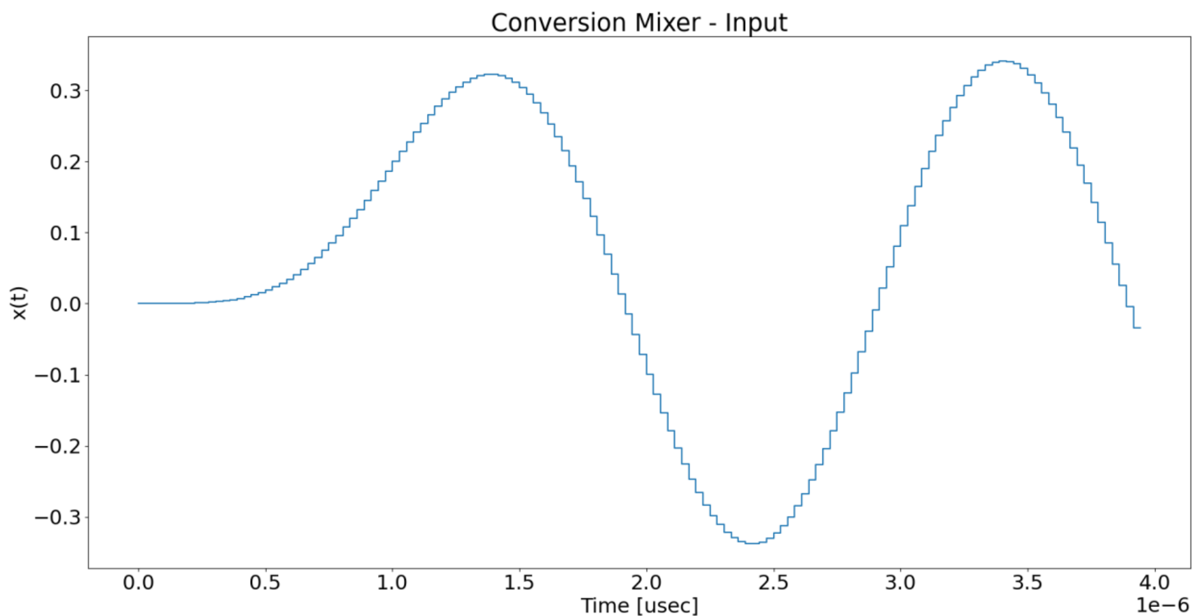

Figure 17. Mixer Input

The simplest resampling method is the linear, as shown in the following code, available in */tools/resampler/resampler.py*. The input signal array is `self.Bn_I`, and `self.Bn_O` is the output signal array. It is observed that the code takes the current value and the next value, and adds as many samples between them as the relationship between the output frequency and the input frequency. These added samples are linearly interpolated from the two samples mentioned before (current ad next values).

```
Bn_prev = 0
```

```
Bn_next = 0
i=0
counter=0
for x in range(len(self.Bn_i)-1):
    Bn_prev = self.Bn_i[x]
    Bn_next = self.Bn_i[x+1]
    diff = float(Bn_next-Bn_prev)
    Bn_tmp = Bn_prev
    for i in range(int(self.relation)+1):
        Bn_tmp = Bn_tmp + diff/float(self.relation)
        self.Bn_o[counter] = Bn_tmp
        counter+=1
```

Code 9. Linear resampler */tools/resampler/resampler.py*

The output of the above resampler given as input the Butterworth filter output is the following one:

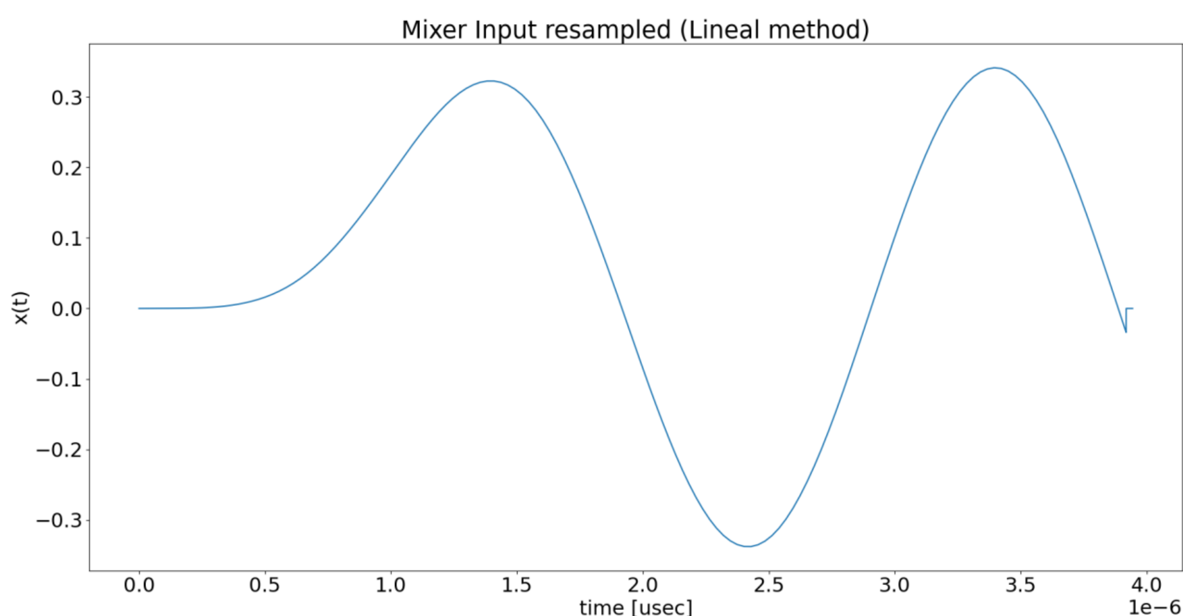

Figure 18. Mixer Input resampled (Lineal method)

As we can see the method of linear resample is good enough for this type of sinusoidal signals.

Nevertheless, we have tested resampling using the fourier method. The scipy library implements a function called resample that uses that method:

```
scipy.signal.resample(x, num, t=None, axis=0, window=None, domain='time')
```

- `x`: the data to be resampled
- `num`: The number of samples in the resampled signal.
- `t`: If t is given, it is assumed to be the equally spaced sample positions associated with the signal data in `x`.
- `axis`: The axis of x that is resampled. Default is 0.
- `window`: Specifies the window applied to the signal in the Fourier domain. See below for details.

- `domain`: A string indicating the domain of the input `x`: « `time` » Consider the input `x` as time-domain (Default), « `freq` » Consider the input `x` as frequency-domain.

It returns the resampled array, or, if t was given, a tuple containing the resampled array and the corresponding resampled positions.

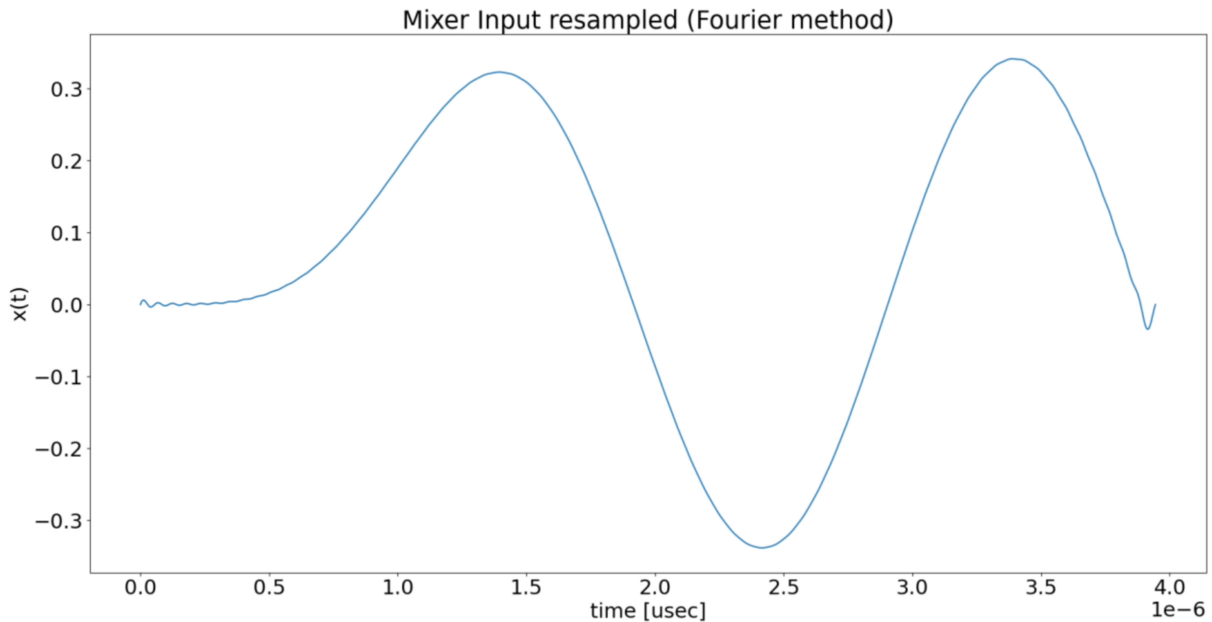

Figure 19. Mixer Input resampled (Fourier method)

It is observed that this method is not so good, at the beginning of the signal is more distorted. More information about the resampling in [Annex.1].

Therefore, we choose the linear resample method. To get the mixer output, we multiply the signal we just resampled with the mixer signal. The output signal can be seen in the following graph:

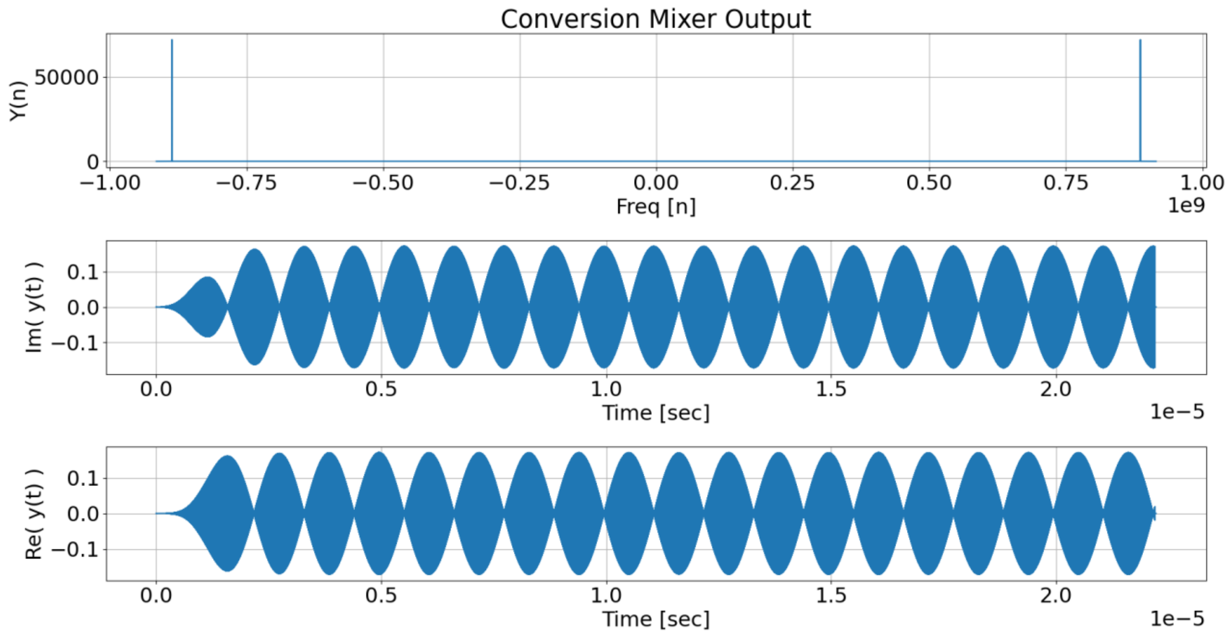

Figure 20. Conversion Mixer Output

As it is known, when a sinusoidal signal is mixed with the carrier, the expected output in frequency domain is two dirac deltas in the frequency of the carrier (positive and negative). To see it properly, we are going to zoom in:

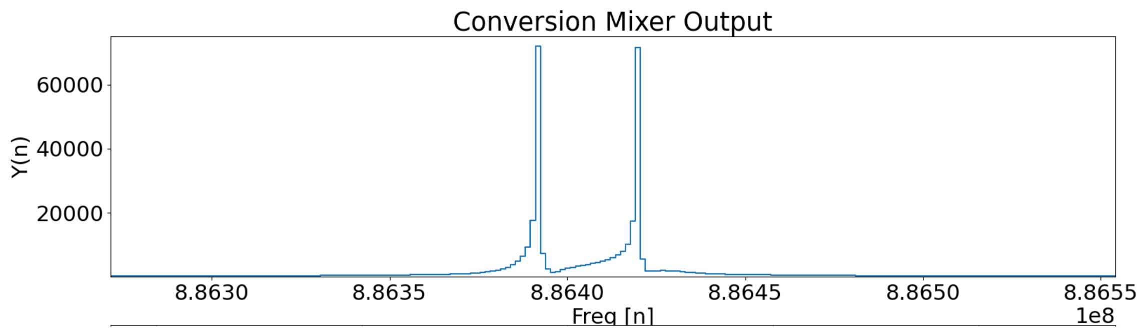

Figure 21. Conversion Mixer Output Frequency domain

It is important to note that the represented frequency domain is discretized, so the x axis is not in Hertzs, it is in "number of samples".

4.2.4 RF Amplifier

The peak output power of the radio frequency amplifier is +5dBm and the optimal load impedance present in the amplifier is 100 ohms.

With this information provided by the datasheet, we assume that the amplifier is linear and has no delay. In addition, as the maximum bandwidth of the signal is 500 kHz (for I and other 500 kHz for Q), it is chosen that the bandwidth of the amplifier is much higher, as it usually happens in reality.

To calculate the gain of the signal:

$$A = 10^{\frac{A(dBm)}{20}}$$

Then we multiply this gain the input signal of the amplifier. We proceed to make the fourier transform of the filter to represent the module and phase of the amplifier

```
# Calculate the gain of the amplifier
self.gain = 10**(self.dBm_gain/20)

# The frequency domain of the filter is used to represent the module and
phase of the amplifier
self.myfilter_f = np.ones(Np)*self.gain
self.myfilter_t = np.abs(np.fft.ifft(self.myfilter_f))
self.input_signal_f=np.fft.fft(self.input_signal)

# Calculate the output of the amplifier block in time and frequency do-
mains
self.out_t = self.input_signal*float(self.gain)
self.out_f = np.fft.fft(self. out_t)
```

Code 10. Configure the amplifier *tools/amplifier/amp.py*

Then the output power of the signal is calculated using the following formula:

$$Z = 100\Omega$$

$$V_{RMS} = \sqrt{\overline{y}^2}$$

$$Po(dBm) = 20 \cdot log\left(\left|\sqrt{\frac{V_{RMS}}{0.001 \cdot Z}}\right|\right)$$

In python, the above formula can be expressed as follows:

```
def get_Vrms(self, y):
    return np.sqrt(np.mean(y**2))

def get_dbm(self, y):
    vrms = self.get_Vrms(y)
    return 20*np.log10(np.abs(vrms/(0.001 * self.impedance)**0.5) )

def get_gain(self):
    relation = self.get_Vrms(self.out_t)/self.get_Vrms(self.input_signal)
    return 20*np.log10(np.abs(relation))
```

Code 11. Calculate output gain of the *amplifier tools/amplifier/amp.py*

```
print("P(dbm) input: ",self.get_dbm(self.input_signal_f))
print("P(dbm) output: ",self.get_dbm(self.out_f))))
print("GAIN (dBm): ", self.get_gain())
```

```
P(dbm) input:  32.735191686472305
P(dBm) output:  37.735191686472305
GAIN (dBm):  5.0
```

Code 12. Power output of the amplifier */tools/amplifier/amp.py*

Then using pyplot we represent both the module and the phase of the amplifier, as well as its output.

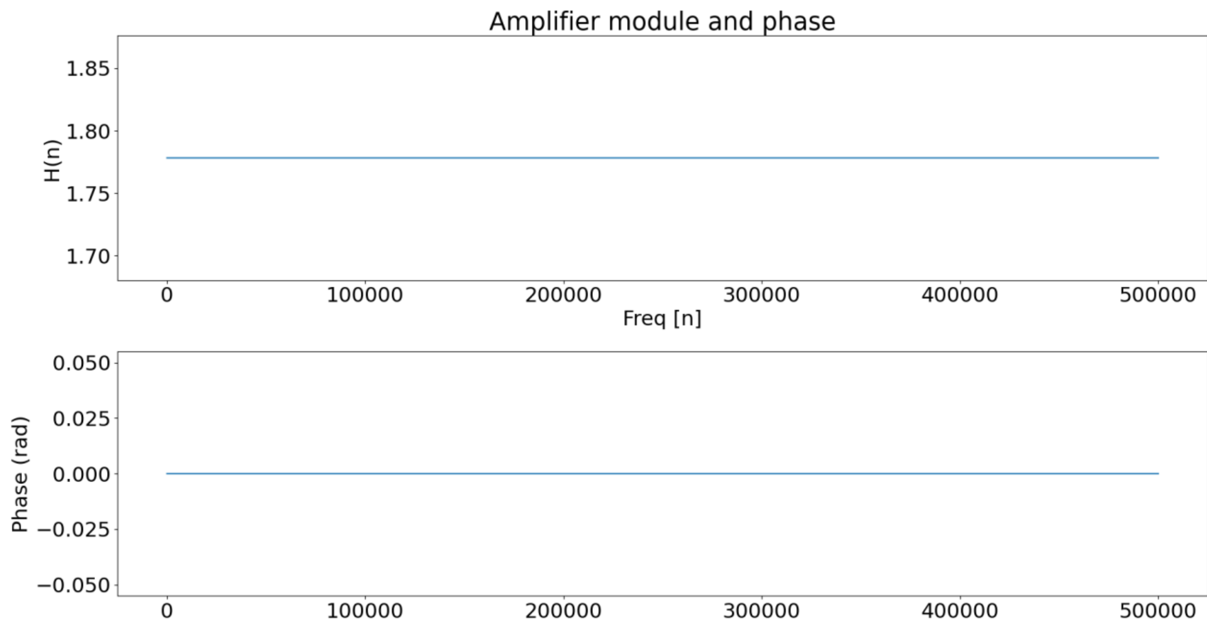

Figure 22. RF Amplifier Module and Phase

As discussed above, a linear amplifier has been chosen because there is no further information in the datasheet.

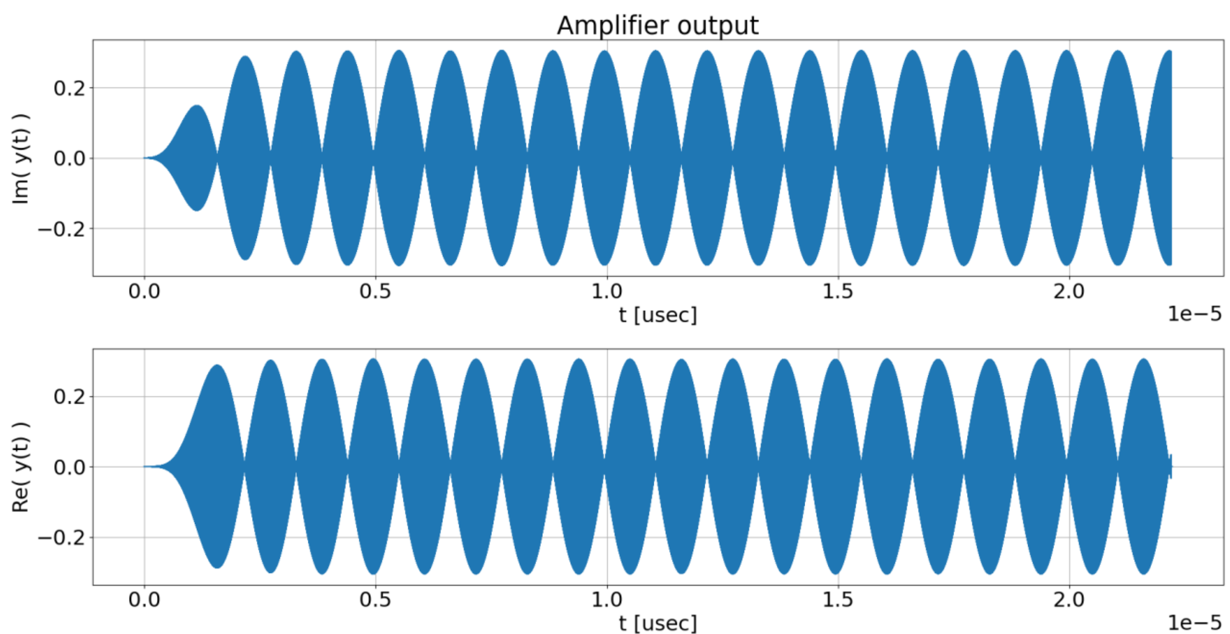

Figure 23. RF Amplifier output

The amplifier is configured to give a gain power of 5 dBm. The power of the input signal is 46.682 dBm, and the power of the output signal is of 51.682 dBm. Both powers had been calculated using the previously defined function `get_dbm()` in section 3.3.4 RF Amplifier. So the gain of the amplifier is 5 dBm as we specified.

4.2 Receiver

4.2.1 Low Noise Amplifier

A single to differential buffer is implemented to improve the second order linearity of the receiver. As explained earlier in the transmitter, the transition from single to differential is not modeled as it makes no sense in a python model.

The LNA gain is programmable over a 48 dB dynamic range, and gain control can be enabled via an external AGC function.

The model used for this block is the same as the one described in section 3.1.4 RF Amplifier. The parameters of gain and impedance are established by `RegRxAnaGain` register.

Implementation in python to determine the impedance and the gain of the LNA:

```
# Impedance
if self.registers.get_var("LnaZin"):
    amp_impedance = 200
else:
    amp_impedance = 50
    self.receiver.set_lna_amp_impedance(amp_impedance)

 # LNA Gain
RxLnaGain = self.registers.get_var("RxLnaGain")
if int(self.registers.get_var("RxLnaGain"), 16)==7 or
         int(self.registers.get_var("RxLnaGain"), 16) == 0 or
         int(self.registers.get_var("RxLnaGain"), 16)==1:
    self.receiver.set_lna_amp_gain=0
elif int(self.registers.get_var("RxLnaGain"), 16)==2:
    self.receiver.set_lna_amp_gain=12
elif int(self.registers.get_var("RxLnaGain"), 16)==3:
    self.receiver.set_lna_amp_gain=12
elif int(self.registers.get_var("RxLnaGain"), 16)==4:
    self.receiver.set_lna_amp_gain=24
elif int(self.registers.get_var("RxLnaGain"), 16)==5:
    self.receiver.set_lna_amp_gain=36
elif int(self.registers.get_var("RxLnaGain"), 16)==6:
    self.receiver.set_lna_amp_gain=48
```

Code 13. Set LNA gain and impedance *SX1257.py*

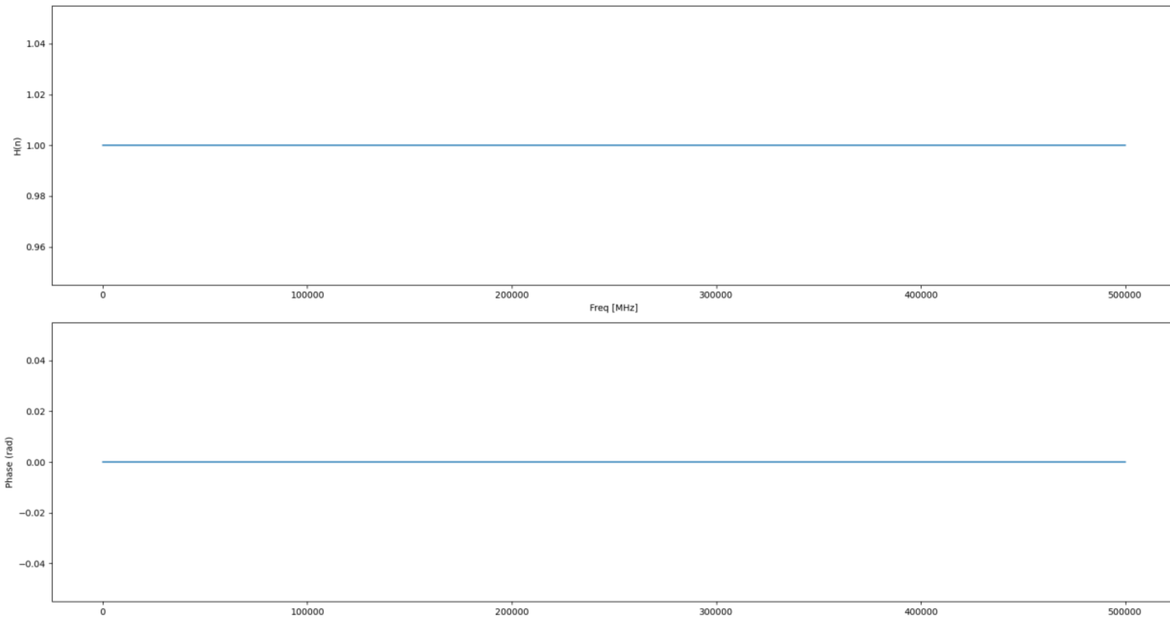

Figure 24. LNA Module and phase

To test the receiver, we will use as input the transmitter output. And the following picture represents the output of the receiver amplifier:

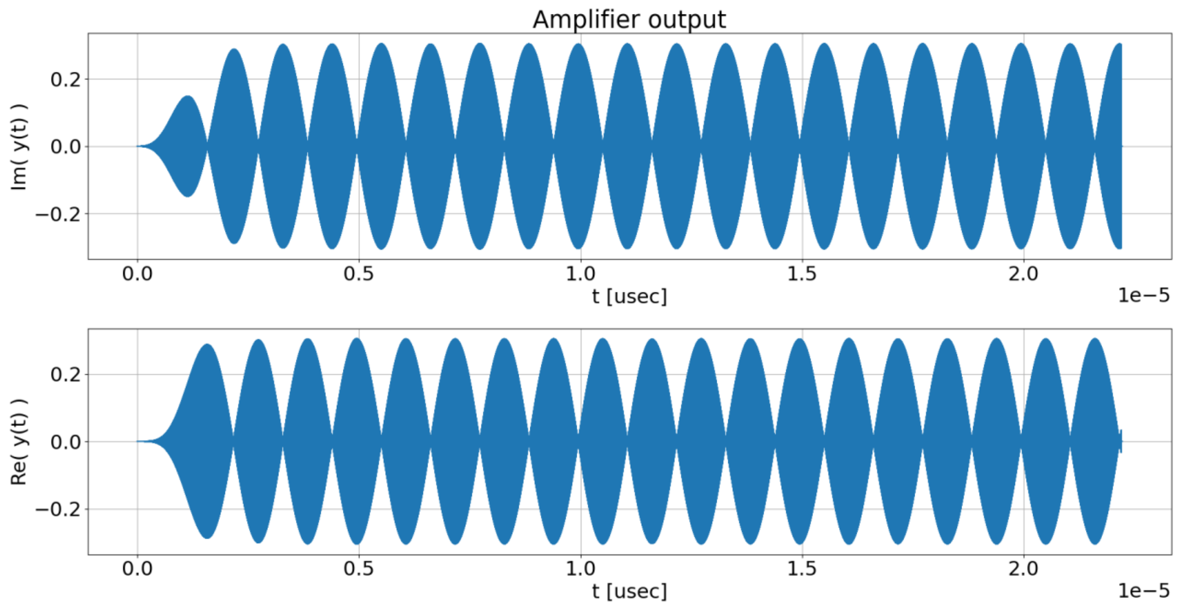

Figure 25. LNA Output

The LNA amplifier is configured to output a gain power of 6 dBm setting the `RegRxAnaGain` register to `0x2F`, so the `RxLNAGain` is setted to 0dB. The power of the input signal is 52.529 dBm, and the power of the output signal is of 58.529 dBm. Both powers had been calculated using the previously defined function `get_dbm()` in section 3.3.4 RF Amplifier. So the gain of the amplifier is 6 dBm as we specified.

4.4.2 I/Q Downconversion Quadrature Mixer

The block model used for this block is the same as 3.1.3 TX I / Q Up-Conversion Mixers, but is phase-shifted with the transmitter mixer signal so that the mixer output is not canceled.

The following code shows how to calculate the Frf (carrier frequency) given the value of `FrfRx` register.

```
RegFrfRx = self.registers.get_multi_var("FrfRx(23:16)", "FrfRx(15:8)",
"FrfRx(7:0)")
frfxx_k = 13325653
fstep_k = self.fxosc/int((2)^19)
frfxx = int(RegFrfRx, 16)/(fstep_k*frfxx_k)*915e6
self.receiver.set_Frf(fxosc=self.fxosc, frfxx = frfxx)
```

Code 14. Calculate the Frf, available at : *SX1257.py*

The I mixer signal is a cosine and the Q mixer signal is a sine, as in the case of the transmitter. But, what happens if we vary the sine and cosine phases?. The answer is simple. The output will be more time shifted or less as we vary it. But if we use the oposite signal (imagine we transmit a sine and the mixer is a cosine), the output will be cancelled. In real systems this is not usual, as we do not typically transmit a sine or cosine alone and it is improbable that the phases that the phases coincide.

The output of the downconversion mixer is shown in the following figure:

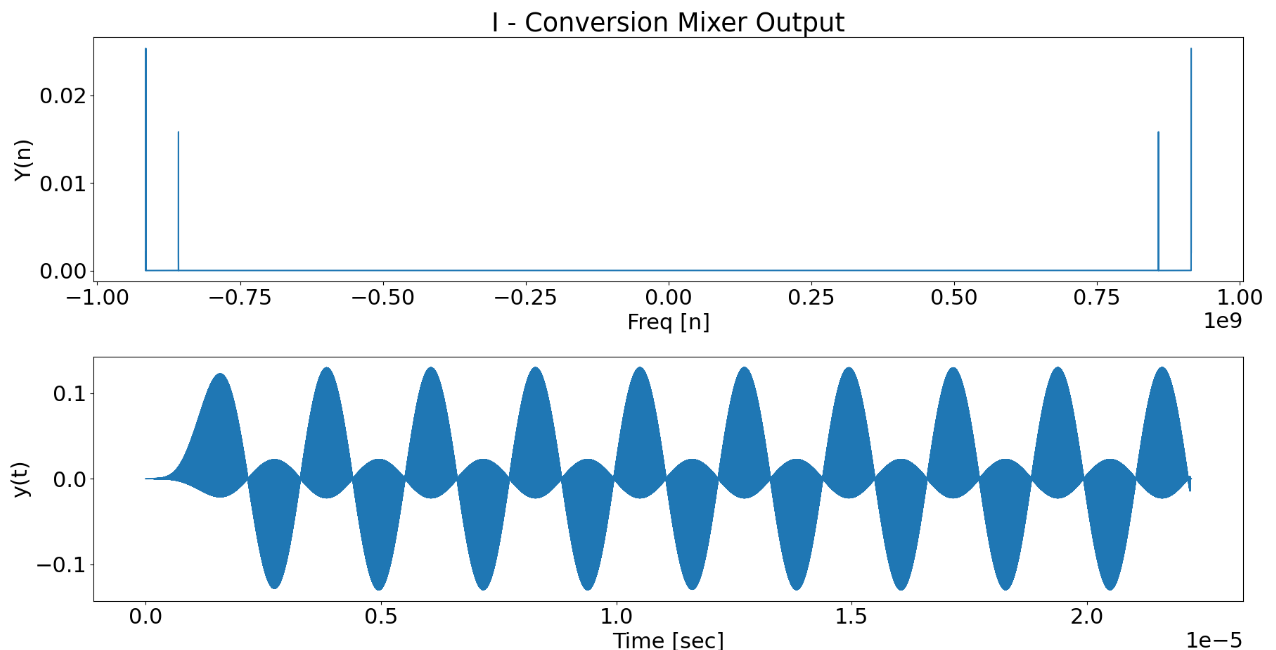

Figure 26. I-Downconversion mixer output

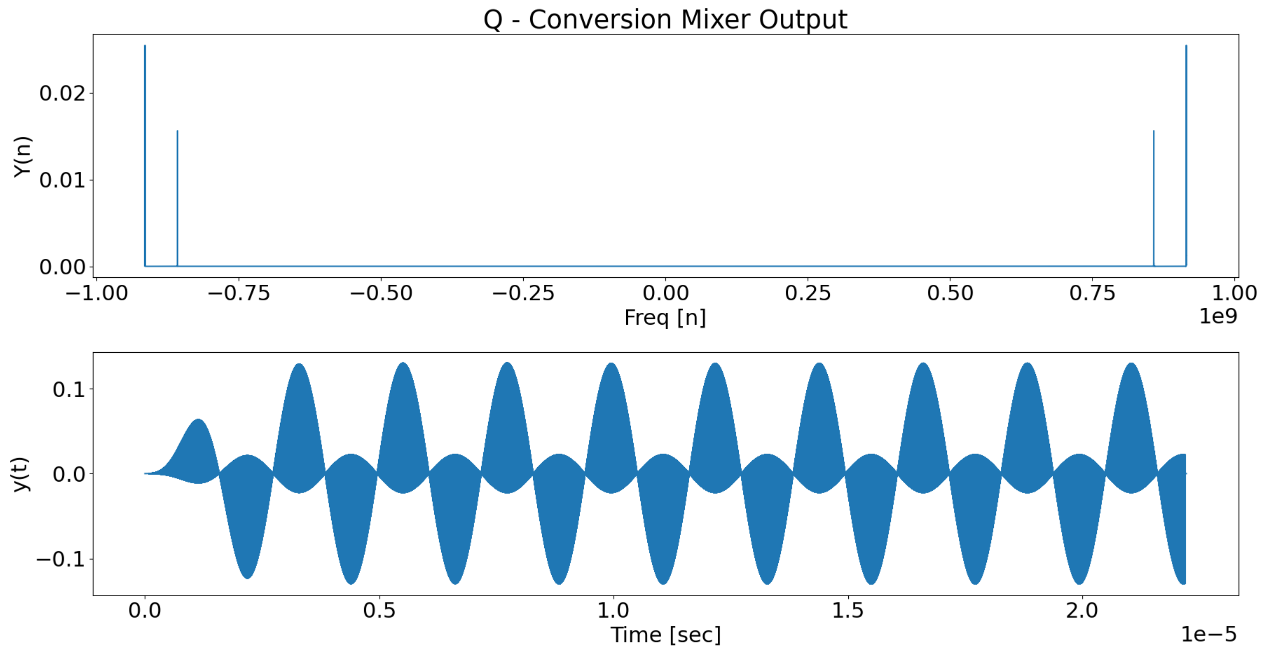

Figure 27. Q-Downconversion mixer output

4.4.3 Baseband Analog Filters and Amplifiers

In order to model this in python we have decided to divide into two stages: filtering and amplification. A first order butterworth filter is used for filtering. The following python code is used to determine the lowpass filter gain:

```
RxBasebandBw = int(self.registers.get_var("RxBasebandBw"), 16)
if RxBasebandBw == 0:
        rx_channel_filter_bw=750e3
elif RxBasebandBw == 1:
    rx_channel_filter_bw=500e3
elif RxBasebandBw == 2:
    rx_channel_filter_bw=375e3
else:
    rx_channel_filter_bw=250e3
self.receiver.set_rx_channel_filter_bw(rx_channel_filter_bw)
```

Code 15. Selecting the Baseband gain of the Butterworth filter available at : *SX1257.py*

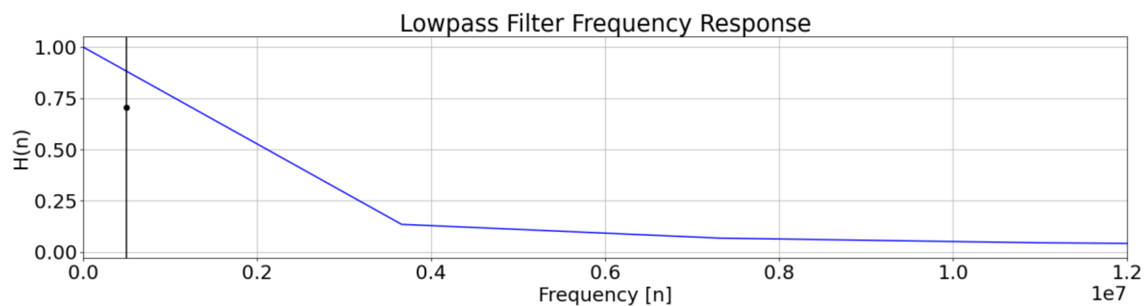

Figure 28. Lowpass filter frequency response

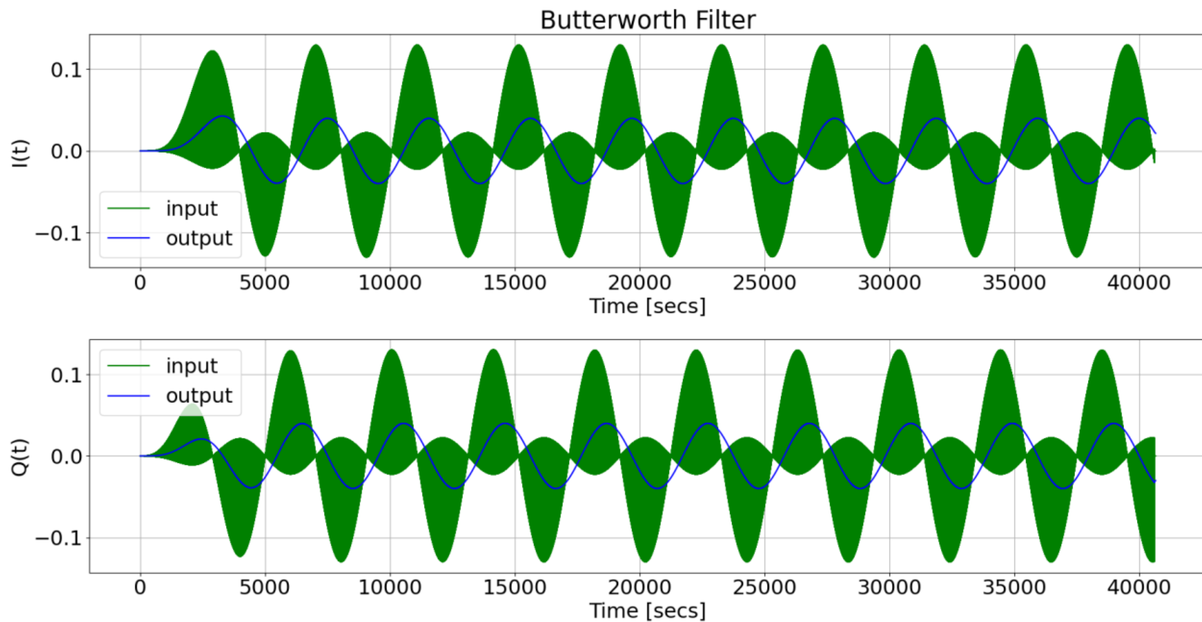

Figure 29. Butterworth filter output

As we did in the transmitter, we are going to plot the two Butterworth filter outputs in the same figure to show that the phase between them is correct.

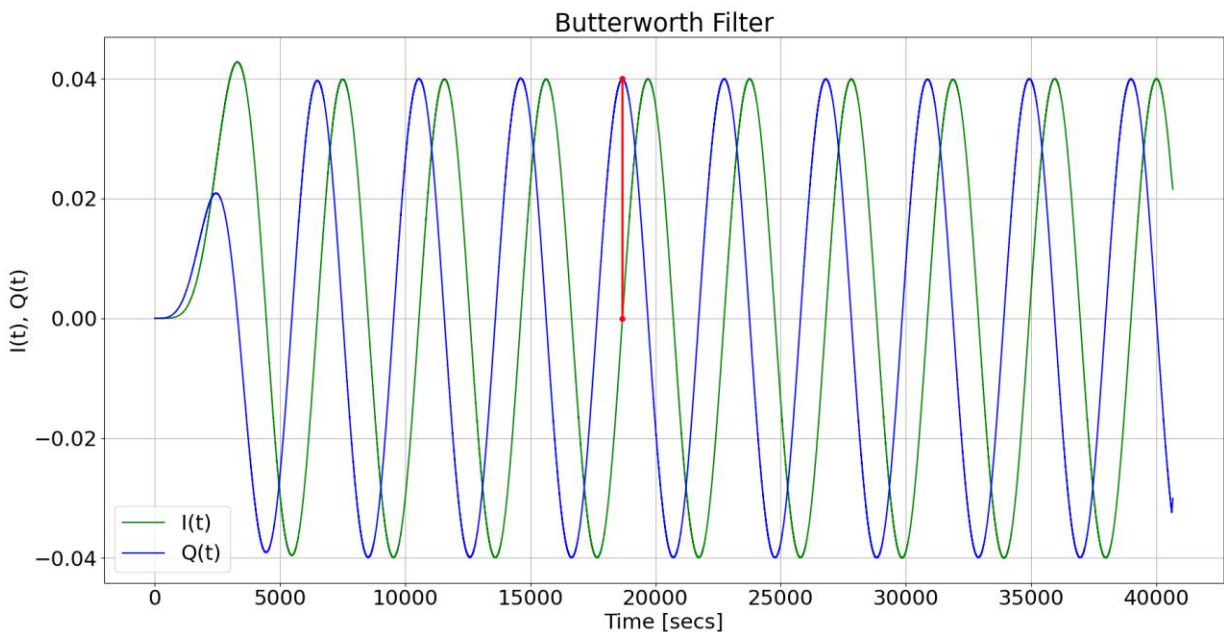

Figure 30. I and Q Butterworth filter output

And for amplification, we reuse the same ideal amplifier that has been modeled before. The amplifier gain is determined by `RxBasebandGain`. The following python code is used to determine the gain given the value of the mentioned variable:

```
baseband_gain = int(self.registers.get_var("RxBasebandGain"),
16)/int('0x7', 16)*30-24
self.receiver.set_rx_channel_filter_baseband_gain(baseband_gain)
```

Code 16. Set the baseband gain of the Butterworth filter available at : *SX1257.py*

4.4.4 Downsampling

The ADC output allows for 13 bits of resolution after decimation and filtering by the external baseband processor within a 500 kHz maximum bandwidth, corresponding to a maximum RF received double sideband bandwidth of 1 MHz. The ADC output is one bit per channel quadrature bit stream at 32 or 36 MSamples/s.

Consider that the length of the output signal of the previous block is 230987 samples, and the output that we are expecting has 144 samples, we need to take one sample every 1614 samples.

```
downsampling=int(len(self.input_signal)/original_length)
print("Downsampling ratio: ",downsampling)
downsampling_output_I= []
downsampling_output_Q= []
downsampling_time_output= []

counter=downsampling
for time_element,element_I,element_Q in zip(self.t_i, output_base-
band_amp_I, output_baseband_amp_Q):
    if counter == downsampling:
        downsampling_output_I.append(element_I)
        downsampling_output_Q.append(element_Q)
        downsampling_time_output.append(time_element)
        counter=0
    counter+=1
```

Code 17. Downsampling method */tools/receiver.py*

4.4.5 Analog-to-Digital Converters

The receiver digital baseband consists of separate I and Q channel 5th order continuous-time sigma-delta modulator analog -to-digital converters which sample and digitize the analog baseband I and Q signals output at the analog baseband amplifiers.

Intending to model the ADC in python, is the `deltasigma` python module is used. A matlab's deltasigma toolbox port which support NTF synthesis, modulator simulation and more.

The oversampling ratio choosen for the ADC it being 32, due to is the same as in the DAC, since the datasheet doesn't specify it.

```
H = synthesizeNTF(self.order, self.OSR, 1) # 1 -> optimized
v, xn, xmax, y = simulateDSM(u, H)
```

Code 18. Deltasigma ADC */tools/delta_sigma_ADC.py*

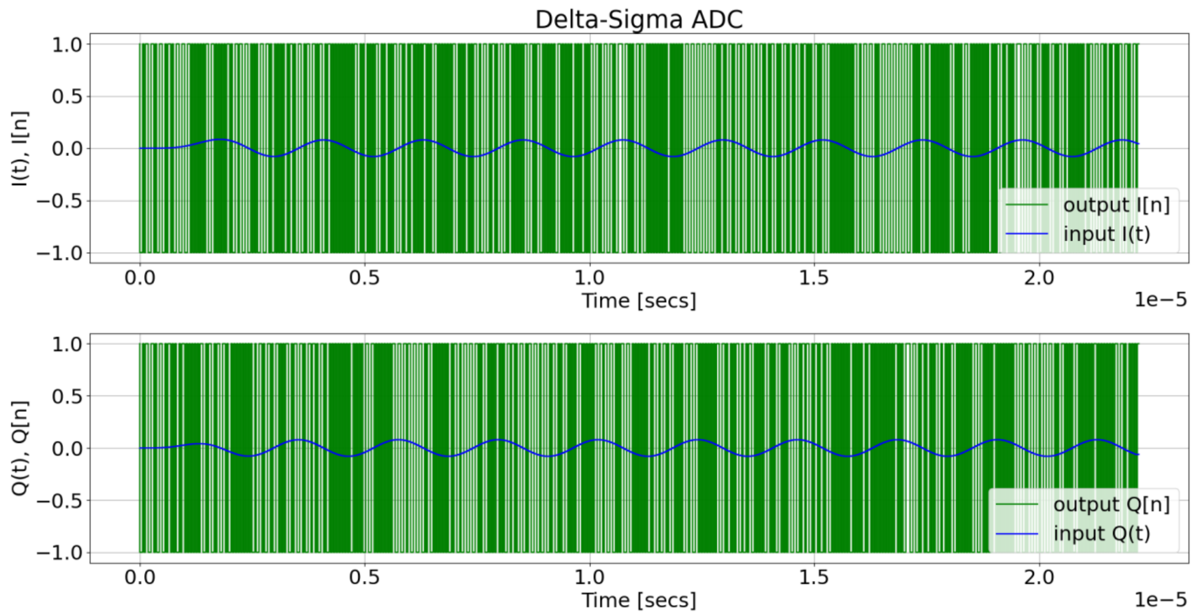

Figure 31. Delta Sigma ADC Output

The ADC output signal requires a bit of explanation. If you take a closer look it is ok: when the input signal increases,, the output is still at 1, and when it decreases, the output goes down to –1. And if we pass the output signal of the ADC through a DAC, we obtain the expected signal.

4.5 Control registers and interface

The register memory map has also been modelled. Given a python dictionary stored in the register map file stored at */data/registerMap.json* with the format shown in section "4.5.2 Register map used for memory examples" the following script */tools/registerMapSerializer.py* is in charge of associating the values of the registers to the corresponding variables of each block. Therefore, if we change any value of the register map file, it will be reflected in the behavior of the model.

Given a json with the structure of *registerMap.json* we can get and set variables of the memory map easily. There are multiple python modules that implement similar features, but it has been decided to use json files, which are more readable and comfortable to work with in Python. For example:

- registerMap https://pypi.org/project/registerMap/ uses yaml files to create the register map
- regdesc https://github.com/airwoodix/regdesc it modelates the register map with python classes

4.5.1 How does registerMapSerializer work?

The main class builder receives 4 parameters: the file path to the register map (in json format), another path to store a temporal copy of the memory map, the length of the addresses and the length of the registers, both in bytes.

```
def __init__(self, filename, tmp_filename, addr_len=8, reg_len=8):
```

Code X: Arguments for the registerMapSerializer builder */tools/registerMapSerializer.py*

The temporal copy of the register map is done because I wanted to preserve the original json. If the register map is changed, only the temporal copy will be updated.

4.5.1.1 Create the map of variables

Create the map of variables, so when you want to change the value to a variable you will know which register it is in and which positions.

```
for register_addr in self.registers:
    this_reg = self.registers[register_addr]
    # If the register have variables
    # we store their position in our variable's dict
    if "variables" in this_reg.keys():
        for variable in this_reg["variables"]:
            self.variables[variable] = {"register_addr":register_addr,
                             "pos":this_reg["variables"][variable]["pos"],
                             "mode":this_reg["variables"][variable]["mode"]}
```

Code 19. Create the map of variables */tools/registerMapSerializer.py*

4.5.1.2 Main functions

The main functions that allow us to make `registerMapSerializer` are:

- `get_all_reg():` To obtain the whole map of records.
- `get_all_var():` To obtain all the variables.

- `get_multi_var(*regs)`: To obtain variables that are distributed in several registers.
- `get_reg(variable)`: To obtain the register to which a given variable belongs.
- `get_var(variable)`: Get the value of a given variable.
- `set_var(variable)`: Set the value of a variable

4.5.2 Register map used for memory examples

The following register map is used for this memory examples.

```
{
  "0x00": {
    "name": "RegMode",
    "value": "0x00",
    "desc": "Operating modes of the SX1257",
    "variables": {
      "StandbyEnable": {
        "pos": "0:0",
        "mode": "rw",
        "default_value": "0x1"
      },
      "RxEnable": {
        "pos": "1:1",
        "mode": "rw",
        "default_value": "0x0"
      },
      "TxEnable": {
        "pos": "2:2",
        "mode": "rw",
        "default_value": "0x0"
      },
      "PADriverEnable": {
        "pos": "3:3",
        "mode": "rw",
        "default_value": "0x0"
      },
      "-.00": {
        "pos": "7:4",
        "mode": "r",
        "default_value": "0x0"
      }
    }
  },
  "0x01": {
    "name": "RegFrfRxMsb",
    "value": "0xcb",
    "desc": "RX carrier frequency MSB",
    "variables": {
      "FrfRx(23:16)": {
        "pos": "7:0",
        "mode": "rw",
        "default_value": "0xCB"
      }
    }
  },
  "0x02": {
```

```
    "name": "RegFrfRxMid",
    "value": "0x55",
    "desc": "RX carrier frequency intermediate bits",
    "variables": {
      "FrfRx(15:8)": {
        "pos": "7:0",
        "mode": "rw",
        "default_value": "0x55"
      }
    }
  },
  "0x03": {
    "name": "RegFrfRxLsb",
    "value": "0x55",
    "desc": "RX carrier frequency LSB",
    "variables": {
      "FrfRx(7:0)": {
        "pos": "7:0",
        "mode": "rw",
        "default_value": "0x55"
      }
    }
  },
  "0x04": {
    "name": "RegFrfTxMsb",
    "value": "0xcb",
    "desc": "TX carrier frequency MSB",
    "variables": {
      "FrfTx(23:16)": {
        "pos": "7:0",
        "mode": "rw",
        "default_value": "0xCB"
      }
    }
  },
  "0x05": {
    "name": "RegFrfTxxMid",
    "value": "0x55",
    "desc": "TX carrier frequency intermediate bits",
    "variables": {
      "FrfTx(15:8)": {
        "pos": "7:0",
        "mode": "rw",
        "default_value": "0x55"
      }
    }
  },
  "0x06": {
    "name": "RegFrfTxLsb",
    "value": "0x55",
    "desc": "TX carrier frequency LSB",
    "variables": {
      "FrfTx(7:0)": {
        "pos": "7:0",
        "mode": "rw",
        "default_value": "0x55"
      }
    }
  },
  "0x08": {
    "name": "RegTxGain",
    "value": "0x55",
```

```
    "desc": "TX DAC and mixer gain setting",
    "variables": {
      "TXMixerGain": {
        "pos": "3:0",
        "mode": "rw",
        "default_value": "0xE"
      },
      "TxDacGain": {
        "pos": "6:4",
        "mode": "rw",
        "default_value": "0x2"
      },
      "-.08": {
        "pos": "7:4",
        "mode": "r",
        "default_value": "0x2"
      }
    }
  },
  "0x0A": {
    "name": "RegTxBw",
    "value": "0x30",
    "desc": "TX FE PLL and analog filter bandwidths",
    "variables": {
      "TxAnaBw": {
        "pos": "4:0",
        "mode": "rw",
        "default_value": "0x1F"
      },
      "TxPllBw": {
        "pos": "6:5",
        "mode": "rw",
        "default_value": "0x3"
      },
      "-.0A": {
        "pos": "7:7",
        "mode": "r",
        "default_value": "0x0"
      }
    }
  },
  "0x0B": {
    "name": "RegTxDacBw",
    "value": "0x01",
    "desc": "TX DAC bandwidth",
    "variables": {
      "TxDacBw": {
        "pos": "2:0",
        "mode": "rw",
        "default_value": "0x2"
      }
    }
  },
  "0x0C": {
    "name": "RegRxAnaGain",
    "value": "0x03",
    "desc": "RX FE LNA and baseband amplifier gain",
    "variables": {
      "LnaZin": {
        "pos": "1:1",
        "mode": "rw",
        "default_value": "0x1"
```

```
      },
      "RxBasebandGain": {
        "pos": "4:1",
        "mode": "rw",
        "default_value": "0xF"
      },
      "RxLnaGain": {
        "pos": "7:5",
        "mode": "rw",
        "default_value": "0x1"
      }
    }
  },
  "0x0D": {
    "name": "RegRxBw",
    "value": "0xFD",
    "desc": "RX FE ADC and analog filter bandwidths",
    "variables": {
      "RxBasebandBw": {
        "pos": "1:0",
        "mode": "rw",
        "default_value": "0x1"
      },
      "RxAdcTrim": {
        "pos": "4:2",
        "mode": "rw",
        "default_value": "0x7"
      },
      "RxAdcBw": {
        "pos": "7:5",
        "mode": "rw",
        "default_value": "0x7"
      }
    }
  },
  "0x0E": {
    "name": " RegRxPLLBw",
    "value": "0x06",
    "desc": "RX FE PLL bandwidth",
    "variables": {
      "RxAdcTemp": {
        "pos": "0:0",
        "mode": "rw",
        "default_value": "0x0"
      },
      "RxPllBw": {
        "pos": "2:1",
        "mode": "rw",
        "default_value": "0x2"
      },
      "-.0E": {
        "pos": "7:3",
        "mode": "r",
        "default_value": "0x0"
      }
    }
  },
  "0x0F": {
    "name": "RegDioMapping",
    "value": "0x00",
    "desc": "Mapping of DIO pins",
    "variables": {
```

```
      "Dio3Mapping": {
        "pos": "1:0",
        "mode": "rw",
        "default_value": "0x0"
      },
      "Dio2Mapping": {
        "pos": "3:2",
        "mode": "rw",
        "default_value": "0x0"
      },
      "Dio1Mapping": {
        "pos": "5:4",
        "mode": "rw",
        "default_value": "0x0"
      },
      "Dio0Mapping": {
        "pos": "7:6",
        "mode": "rw",
        "default_value": "0x0"
      }
    }
  },
  "0x10": {
    "name": "RegClkSelect",
    "value": "0x02",
    "desc": "Sampling clock congfiguration",
    "variables": {
      "TxDacClkSelect": {
        "pos": "0:0",
        "mode": "rw",
        "default_value": "0x0"
      },
      "Clk_out": {
        "pos": "1:1",
        "mode": "rw",
        "default_value": "0x1"
      },
      "RfLoopBack": {
        "pos": "2:2",
        "mode": "rw",
        "default_value": "0x0"
      },
      "DigitalLoopBack": {
        "pos": "3:3",
        "mode": "rw",
        "default_value": "0x0"
      },
      "-.10": {
        "pos": "7:4",
        "mode": "r",
        "default_value": "0x0"
      }
    }
  },
  "0x11": {
    "name": "RegMode Status",
    "value": "0x00",
    "desc": "SX1257 mode status",
    "variables": {
      "PlllLockTx": {
        "pos": "0:0",
        "mode": "r",
```

```
          "default_value": "0x0"
        },
        "PllLockRx": {
          "pos": "1:1",
          "mode": "r",
          "default_value": "0x0"
        },
        "LowBatEnable": {
          "pos": "2:2",
          "mode": "rw",
          "default_value": "0x0"
        },
        "-.11": {
          "pos": "7:3",
          "mode": "r",
          "default_value": "0x0"
        }
      }
    },
    "0x1A": {
      "name": "RegLowBatThres$",
      "value": "0x02",
      "desc": "Low battery threshold",
      "variables": {
        "LowBatThres": {
          "pos": "2:0",
          "mode": "rw",
          "default_value": "0x2"
        },
        "-.1A": {
          "pos": "7:3",
          "mode": "r",
          "default_value": "0x0"
        }
      }
    }
}
```

Register map */data/registerMap.json*

5 RF LOOPBACK

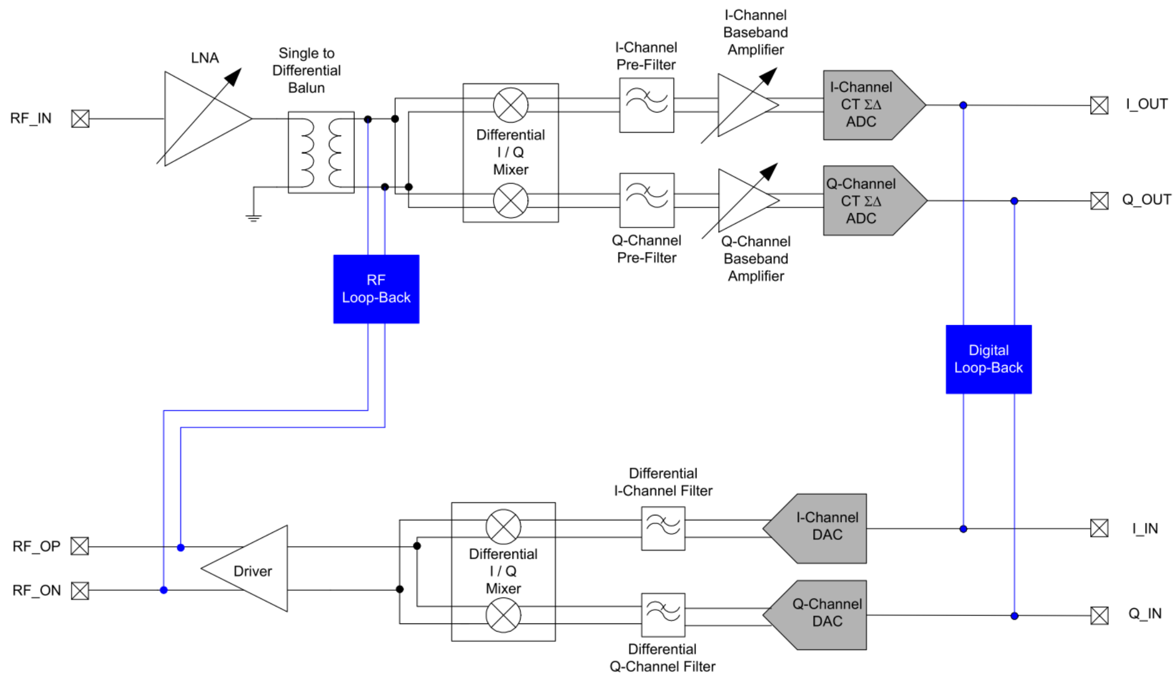

Figure 32. Digital and RF Loop-Back Paths. Source: [Ref 1]

The RF loop-back path connects the balanced RF output signal of the transmitter driver stage to the output of the mixer of the receiver. This path provides a mechanism for the external baseband processor to implement a calibration for the following:

- Receiver I, Q gain mismatch
- Receiver I and Q phase imbalance
- Transmitter I, Q gain mismatch
- Transmitter I and Q phase imbalance
- Transmitter DC offset

We can check that our complete model works correctly by inserting the transmitter output into the receiver input and the receiver output into the transmitter input.

If we do that, these are the transmitter input and output:

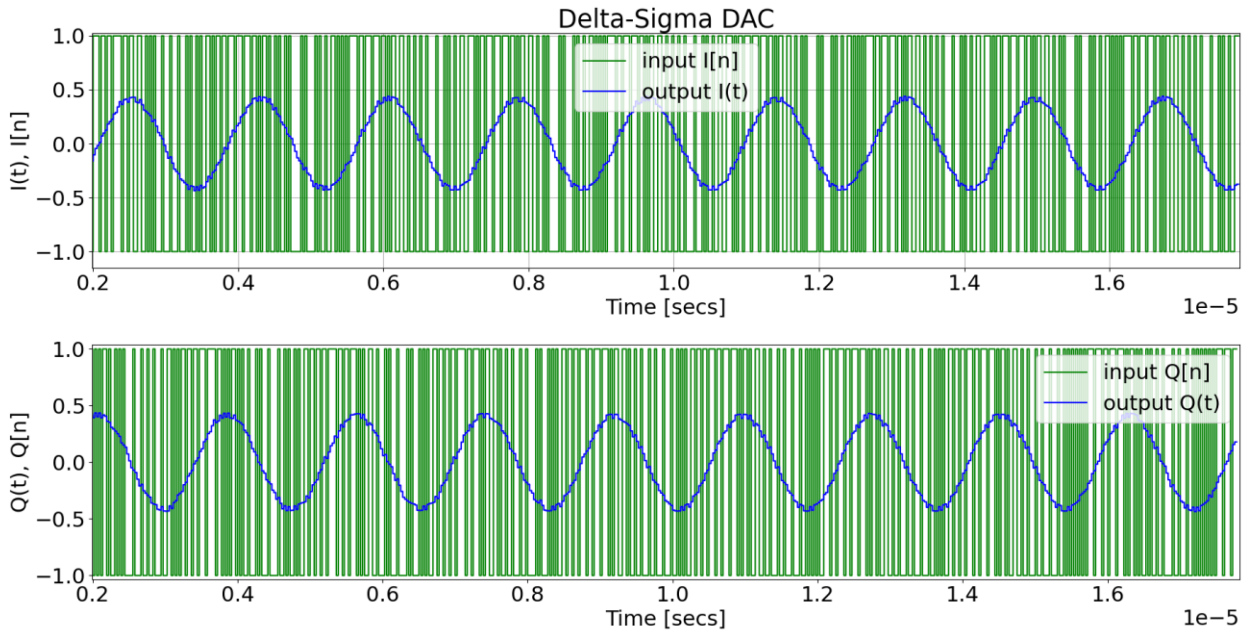

Figure 33. Transmitter input (and DAC output)

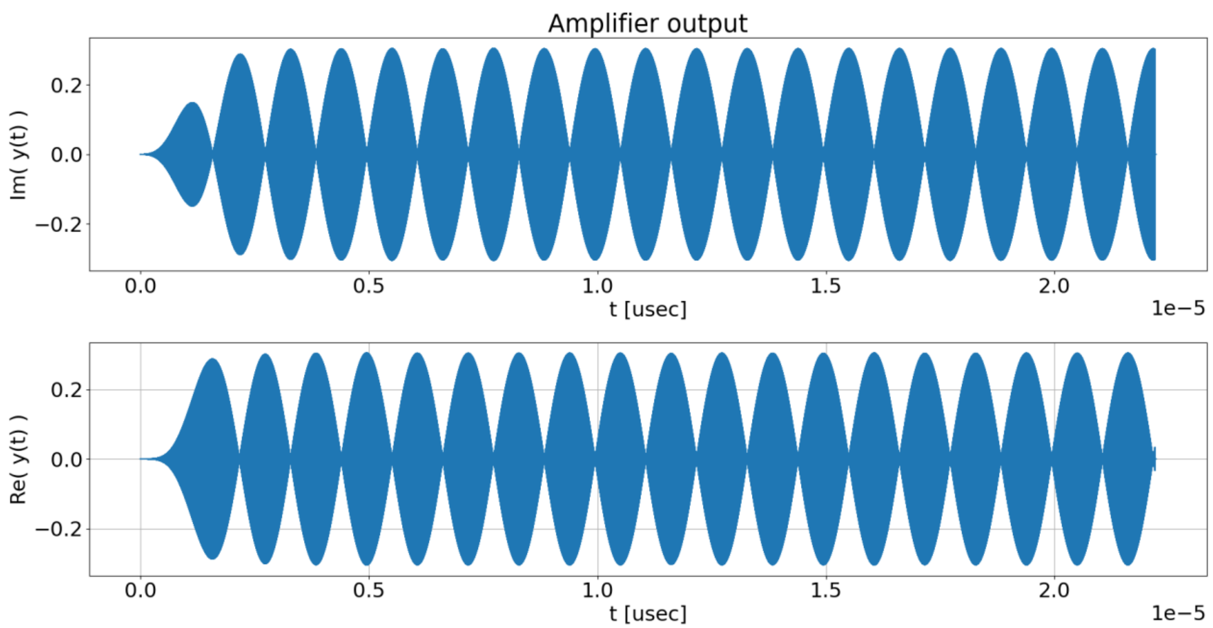

Figure 34. Transmitter output

And then, we introduce the transmitter output into the receiver input, which results in the following output signal:

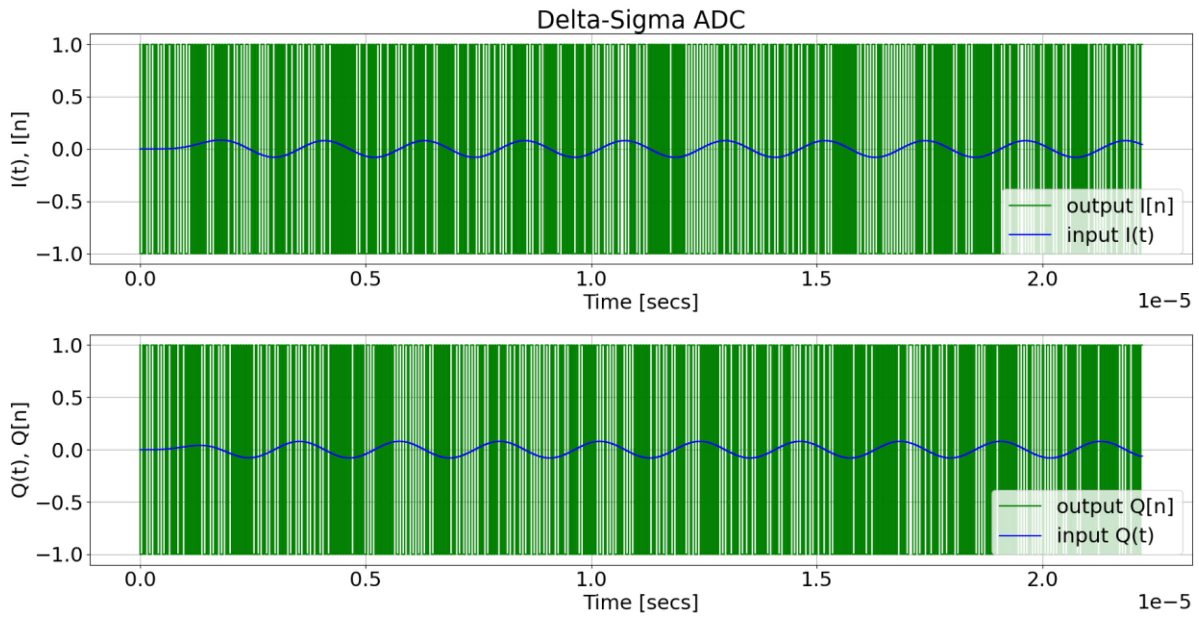

Figure 35. Receiver output

Finally we put the output of the receiver back into the transmitter to check that the signal is reconstructed correctly.

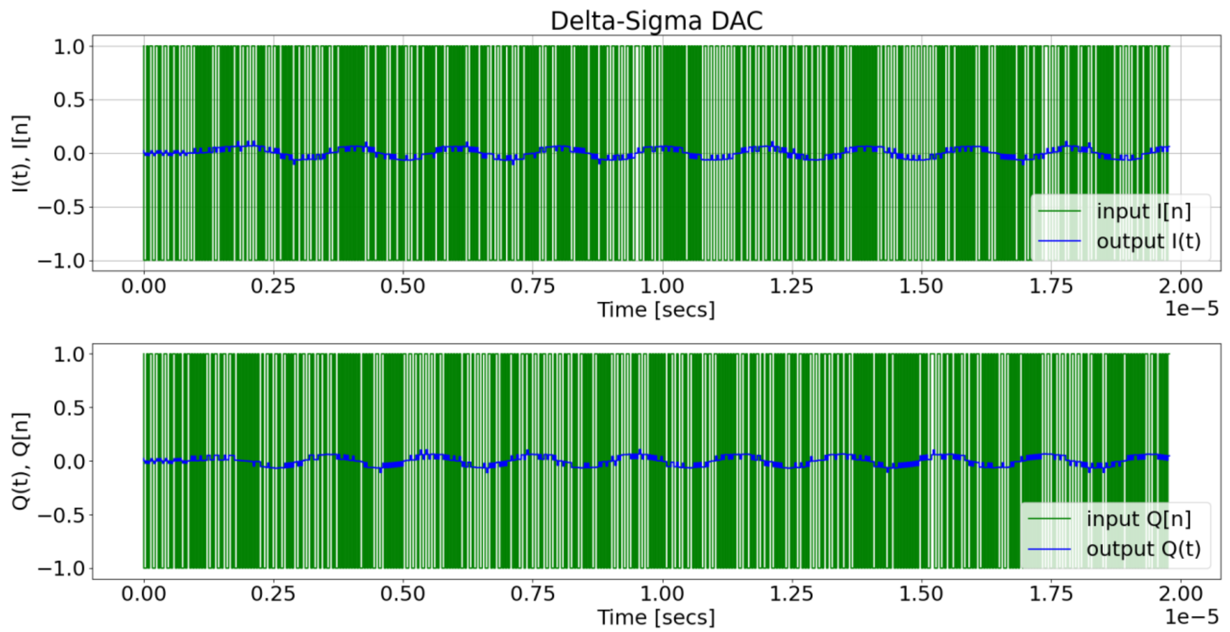

Figure 36. Transmitter input and DAC output

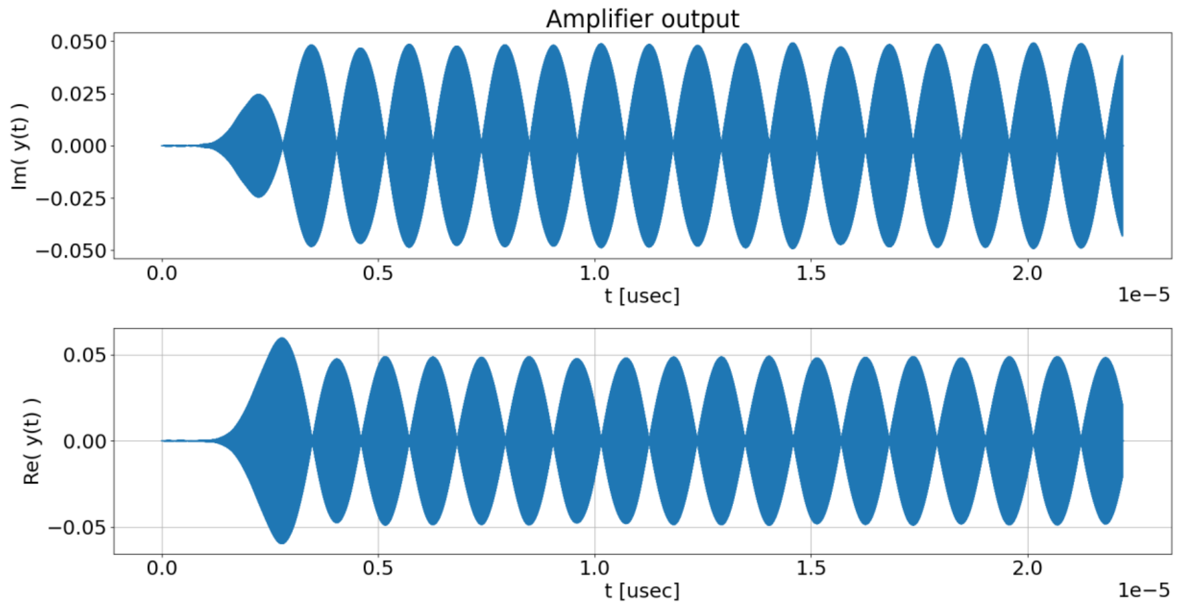

Figure 37. Transmitter output

It can be seen that the second time it passes through the transmitter the signal has more noise. This is because the signal has been filtered more times and has passed through more stages. Ideally, the signal should be reconstructed by the digital baseband processor before moving back from the receiver to the transmitter.

6 How to use the python model

If we run the program directly without specifying any option, an usage guide is shown :

```
roberto@gitt:~/sx1257-python-model$ python3 main.py
usage: main.py [-h] [--tx_input_data TX_INPUT_DATA] [--rx_input_data
RX_INPUT_DATA] [--tx_output TX_OUTPUT] [--rx_output RX_OUTPUT] [--tx] [--
rx] [--tx-example] [--rx-example] [--loopback-example]

sx1257 python model

optional arguments:
  -h, --help            show this help message and exit
  --tx_input_data TX_INPUT_DATA
                        Data to tx
  --rx_input_data RX_INPUT_DATA
                        Data to rx
  --tx_output TX_OUTPUT
                        Output file of the TX
  --rx_output RX_OUTPUT
                        Output file of the RX
  --tx                  Enables TX in the SX1257 memory map
  --rx                  Enables RX in the SX1257 memory map
  --tx-example          Shows an example of TX
  --rx-example          Shows an example of RX
  --loopback-example    Shows an example of TX and the output goes to the
RX input
```

If we run the script with the option –tx-example, –rx-example or –loopback-example i twill process the examples used in this document to show the behaviour of the model.

6.1 Transmit

If we want to specify a string of bytes to transmit, first of all we have to enable the transmission with the tx flag set, and then, with the `tx_input_data` argument, specify the transmitter input file:

```
roberto@gitt:~/sx1257-python-model$ python3 main.py --tx --tx_input_data
input/transmitter/digital_input_fs_32M_bw_450k.txt
```

The transmitter input data file shoud have two values in each line separated by one whitespace, the first one is the time and the second one the value of the signal in that moment, like this example:

```
0.000000000000000000e+00 1.000000000000000000e+00
2.758352758352758384e-08 -1.000000000000000000e+00
5.516705516705516768e-08 -1.000000000000000000e+00
8.275058275058276476e-08 1.000000000000000000e+00
1.103341103341103354e-07 1.000000000000000000e+00
1.379176379176379457e-07 -1.000000000000000000e+00
1.655011655011655295e-07 1.000000000000000000e+00
```

```
1.9308469308469311134e-07 1.000000000000000000e+00
2.2066822066822206707e-07 -1.000000000000000000e+00
2.4825174825174830750e-07 1.000000000000000000e+00
2.7583527583527589140e-07 1.000000000000000000e+00
```

/input/transmitter/digital_input_fs_32M_bw_450k_2T_with_time.txt

6.2 Receive

If we want to specify a signal to receive, first of all we have to enable the reception with the rx flag set, and then, with the `rx_input_data` argument, specify the receiver input file:

```
roberto@gitt:~/sx1257-python-model$ python3 main.py --rx --rx_input_data
input/receiver/ analog_input_IQ_fs_32M_bw_450k_10T_with_time.txt
```

The format of the input data file is the same as the transmitter input data file.

6.3 Examples

In order to make it easier to test and check the operation you can run the examples shown in the development of the document with the following commands:

```
roberto@gitt:~/sx1257-python-model$ python3 main.py --tx-example
roberto@gitt:~/sx1257-python-model$ python3 main.py --rx-example
roberto@gitt:~/sx1257-python-model$ python3 main.py --loopback-example
```

7 CONCLUSIONS AND FUTURE WORK

A powerful model has been created capable of transmitting and receiving signals just as the SX1257 transceiver would. Everything has been prepared so that it can be compared with the real chip.

Because python consumes a lot of RAM (Random Access Memory), no more than 40 microseconds of signal could be transmitted. Corrections have been made to fix this and it has been decided to modify the model which can be seen in section 2 of the Annex. With these modifications it can process as large signals as you want, as long as you have enough space on the hard disk, which is less limiting than RAM.

REFERENCES

[1] Semtech Corporation. SX1257 Low Power Digital I and Q RF Multi-PHY Mode Transceiver Datasheet [online]. Rev 1.2. [March 2018] Available at: https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/44000000MDmO/OfVC_rbxi4JjkT4hLzU1kq4gOXb4POLNRprWlqxRIZs

[2] NumPy community. NumPy User Guide [online]. Release 1.22.0.: January 14, 2022. Available at: https://numpy.org/doc/stable/

[3] Scipy community. Scipy Documentation [online] [consulted: 02 2022]. Release 1.22.0. Available at: http://scipy.github.io/devdocs/dev/

[4] Matplotlib development team. Pyplot tutorial [online] [consulted: 02 2022]. Available at: https://matplotlib.org/stable/tutorials/introductory/pyplot.html

[5] Harris, Fredric J. (Jan 1978). "On the use of Windows for Harmonic Analysis with the Discrete Fourier Transform" (PDF). Proceedings of the IEEE.

[6] Sadiku, M., & Akujuobi, C. (2004). Software-defined radio: a brief overview. *IEEE Potentials*, *23*(4), 14–15. https://doi.org/10.1109/mp.2004.1343223

[7] Cai, X., Zhou, M., & Huang, X. (2017). Model-Based Design for Software Defined Radio on an FPGA. *IEEE Access*, *5*, 8276–8283. https://doi.org/10.1109/access.2017.2692764

[8] Christie, W. M. (1986). Towards a unified model of language description. *Language Sciences*, *8*(2), 177–191. https://doi.org/10.1016/s0388-0001(86)80015-8

Annex

1. Dealing with the analog world from a digital point of view

The input signal is sampled at another sampling frequency, how should you proceed?. Well, it would seem logical to perform the resample of the signal in order to multiply it just at the DAC output, thus, the continuous domain would be replicated at that point. The problem is that the signal at that point has many jumps, and the resample is not very suitable. It can be seen in the following example what would occur with our signal if we had carried out the resample on the DAC output:

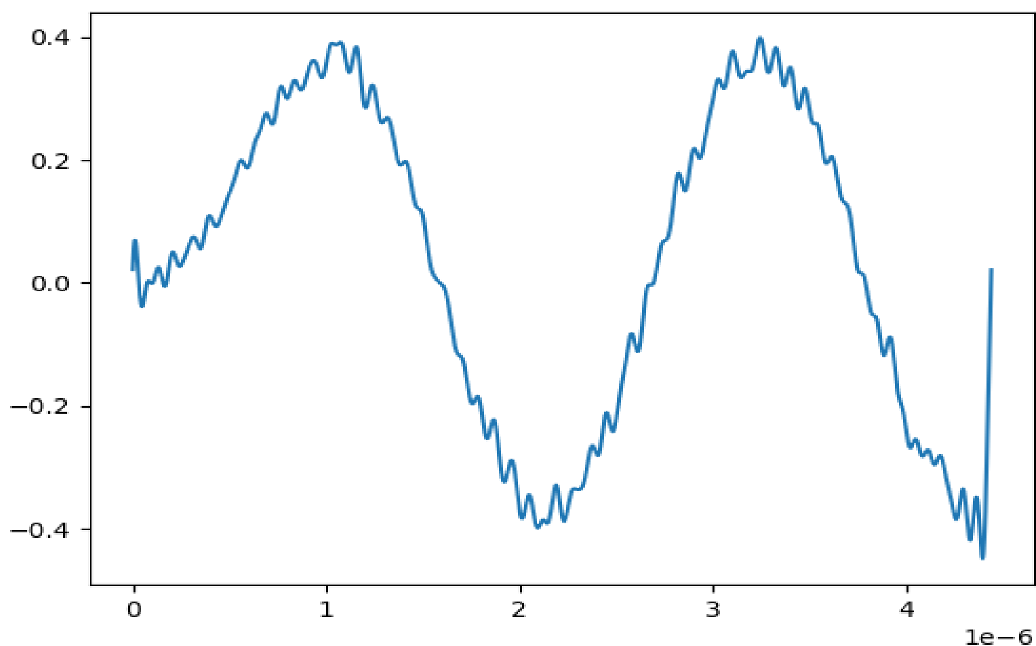

Figure 38. DAC output resampled

In conclusion, before performing resample, it is better to filter the DAC output signal using the butterworth filter that incorporates the SX1257 itself.

2. Transmitting longer signals

2.1 Reducing memory consumption

2.2.1 Generation of graphics

As for the memory consumption in the generation of graphics, `matplotlib` left in memory the figures that have been created even if a capture has already been saved on the hard disk. Therefore, if it was not necessary, it was decided to delete and close the figures after saving a snapshot in memory. To do so, the following matplotlib commands are executed, and each of them is explained below :

- `plt.cla()` : is used to delete the current axes in Matplotlib.
- `plt.clf()` : deletes the entire figure in Matplotlib.
- `plt.close()` : simply closes the figure window in Matplotlib, and we will see nothing when calling the `plt.show()` method.

2.2.2 Signal processing

By default python does not handle objects that have been discontinued and are kept in memory. Therefore, in order to reduce RAM consumption even further, it has been decided to remove objects that are no longer needed and delete them from memory using the python garbage collector module (`gc`) as can be seen in the following example:

```
myds_DAC = ds_DAC_IQ(self.fs, self.DAC_num_taps, self.DAC_bw)
output_DAC_I ,output_DAC_Q = myds_DAC.run(self.Bn_I, self.Bn_Q, plot_out-
put=self.kind_plot=="all")

del myds_DAC
gc.collect()
```

Code 20. Reducing memory consumption */tools/transmitter.py*

This treatment has been carried out after using each block and keeping the output variables we were interested in.

2.2 Splitting signals into pieces

The first approach of the model was capable of transmitting 4 microseconds of signal, which is not very suitable for complex modulations with long symbol lengths such as OFDM. This happened because the computerran out of RAM when processing longer signals. So, we try to improve the model, storing the signal in pieces so we can transmit longer signals.

For most blocks it is no problem: DAC and ADC are finite response filters, and for the mixer, being a simple multiplication, we can divide the input by pieces. The problem is in the Butterworth filter, because it is an infinite response filter and adds a transient to the beginning of the filter output. To solve this, we can divide the signal in pieces, overlapping with the end of the previous slot that is no longer in transient.

To know from which sample the output signal of the butterworth filter ceases to be in transient mode, it is measured empirically.

```
data_example= np.zeros(10000)
data_ example [0] = 10000
y_ example = self.butter_lowpass_filter(data_ example, self.cutoff,
self.fs, self.order)
plt.figure()
plt.plot(y_ example)
```

Code 21. Measuring response of the Butterworth filter

A dirac delta is inserted into the butterworth filter, and it is observed that from the sample number 150 the signal ceases to be in transient mode.

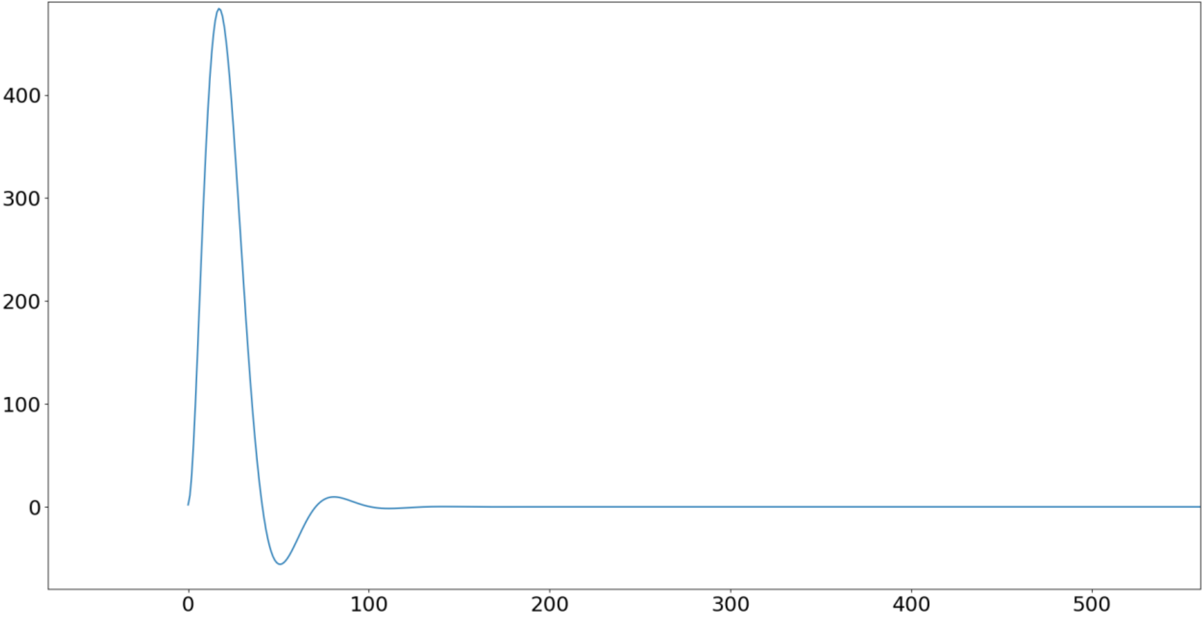

Figure 39. Measuring response of the Butterworth filter

To take advantage of the model we have already created, we have decided to create a separate class in charge of splitting and managing the signals in memory, which we have called `Chunk_Processor`, available at */tools/chunk_processor.py*.

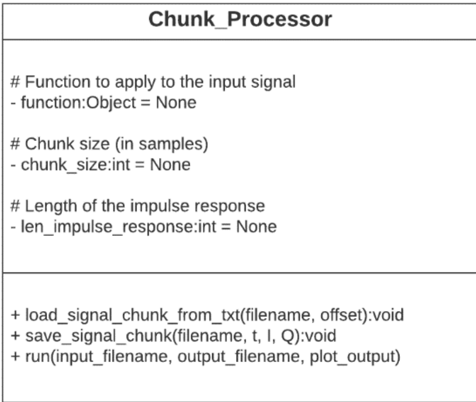

Figure 40. UML Diagram of `Chunk_Processor.`

Each of the methods of this class is described as follows:

- `__init__(function, chunk_size, len_impulse_response)`: It is the constructor of `Chunk_Processor`, it is in charge of setting the object variables passed by argument.
- `load_signal_chunk_from_txt(filename, offset)`: loads the signal from the file specified in the `filename` argument from `offset` to `offset+chunk_size`.
- `save_signal_chunk(filename, t, I, Q)`: Adds the signal chunk (composed of the time vector `t`, `I` and `Q` channel vectors) to the end of the file indicated in `filename`.
- `run(input_filename, output_filename, plot_output)`: We have taken advantage of the functionality offered by Python to pass a function as an argument. Executing the `run` method of `Chunk_Processor` calls the `run` method of the `function` that was passed to the constructor. The output chunk will be stored in `output_filename` using `save_signal_chunk` function. To process the next chunk, it Will start from `offset-len_impulse_response`, thus overwriting the signal transient with the previously processed part.

By applying `Chunk_Processor` and the methods mentioned above to minimize RAM usage it would be possible to transmit signals as large as you want as long as there is enough space on the hard disk, which is less limiting than RAM.