A look into SDR - Part I

Y.Ramgulam

February 2019

Introduction

As an RF engineer I always try to stay up to date with the latest trends and technologies, and for a certain number of years now we've been hearing more and more about Software Defined radio (SDR). But what is SDR?

Usually radio systems are built and optimized to achieve a specific purpose. A commercial transceiver chip like TI's CC1120 is mainly built for communications using OOK/FSK modulations in the ISM bands, for applications such as metering or telemetry for example. It's amazingly good at doing that, but it can't do much else (it's simply not designed to). The concept of SDR, in a nutshell, is to have a generic hardware, not optimized for anything but instead completely open and configurable in software. The price for that flexibility is that RF performance might be less good than with a specialized hardware, and the required processing power can be heavy. But then we get a system that can be adapted to pretty much anything, only using software updates. SDRs are used nowadays in cellular networks, where complexity is ever-increasing; and we can find countless implementation examples on the web.

This technology has now developed to the point where devices and tools are available for a very reasonable price. Table 1 shows a few examples for reference:

	RTL-SDR	**HackRF One**	**USRP(-2900)**
Frequency Range	24MHz - 1.7GHz	1MHz - 6GHz	70MHz - 6GHz
TR/RX Capability	RX Only	Half-Duplex	Full Duplex
Bandwidth	<3MHz	<20MHz	<56MHz
Price Range (Euro)	30	300	900

Table 1: *A few examples of SDRs*

Since we can now have access to very cheap hardware, I thought I should take a peek sometime and see what the fuss is all about. I'm a hardware guy; and SDR is more about software and signal processing. Considering I haven't done any signal processing since Engineering school, I thought it could be a good excuse to refresh my memory and learn new things.

In my constant technology watch, I also kept hearing about more and more Electronics engineers using the programming language Python to help them in their work. Python has gained significant traction over the past few years, to the point where it became the most popular language in early 2019 according to the PYPL index [1].

While digging I realised that Python has an amazing community, especially if we're talking about signal processing and numerical analysis. So I figured I could learn Python and have my first steps in the SDR world at the same time... sort of killing two birds with one stone (no bird was hurt in the process of writing this article though).

In this series of articles I'll be sharing an experimental view with minimal theory. Theory is very well covered in books and it's always nice to see it being put in action. We will focus on the cheapest and most famous device: the RTL-SDR. We will try to decode a signal captured from a simple keyfob (like garage door openers) using Python and its signal processing packages. Once the articles are complete, I will post the python code on Github so anyone can play with it.

1 The transmitter

In this series of articles we will try to decode data captured from a keyfob. We will postulate that we already know the protocol we are trying to decode; we are not going for a blind approach (although we could). For the purpose of this article, the payload is known in advance. To give an idea of real-world application, it could be for example that we have a set of sensors transmitting data wirelessly and we want to build ourselves a gateway using an SDR (because why not?) to collect all that data.

Our transmitter has the following characteristics:

- Carrier Frequency : 869.525 MHz
- Modulation : OOK
- Datarate : 5000 bps
- Synchronisation word (Hex) : DB 6A
- Expected data (Hex) : B2 6D 49 9A A5 B3 64 C9 2A

2 The RTL-SDR

The RTL- SDR is a pretty cool gadget. It originates from the cheap DVB-T dongles that have been massively produced in Asia a few years ago. These dongles, made to decode Digital Television, were using a receiver chip from Realtek that has been diverted from its original intended use to output a raw stream of I/Q samples. Since then a few companies have redesigned around that chip to give a better signal quality and better frequency stability. Figure 1 shows an example of SDR dongle built by NooElec around the Realtek chip.

Figure 1: NooElec NESDR

Figure 2 shows a simple diagram of th RTL-SDR. I won't go into details as there is plenty of material available online[6]. The important part for us is the RTL2832. This chip is the DVB-T demodulator. In its intended mode of operation, it takes in the downconverted signal from the RFIC, digitizes it, converts it to IQ samples and then decodes the DVB-T frames. In the case of RTL-SDR, a manufacturer "debug mode" is exploited, bypassing the DVB-T decoder and allowing us to output a raw stream of IQ samples on the USB bus. We're then essentially using the RTL2832 as an ADC. Unfortunately the RTL2832's datasheet isn't available but the R820T's can be found[7].

After this little presentation of the RTL-SDR, let's dive into the interesting stuff.

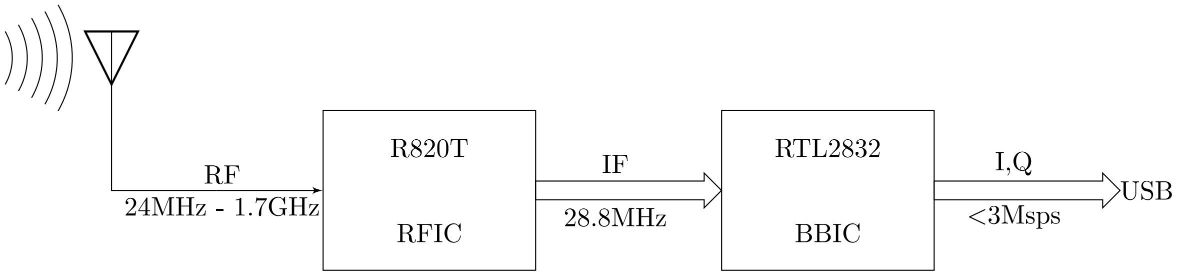


Figure 2: Simplified RTL-SDR internal diagram

3 Demodulating and decoding an OOK signal

In order to demodulate and decode an OOK signal, we will perform a number of operations which are depicted in Figure 3.

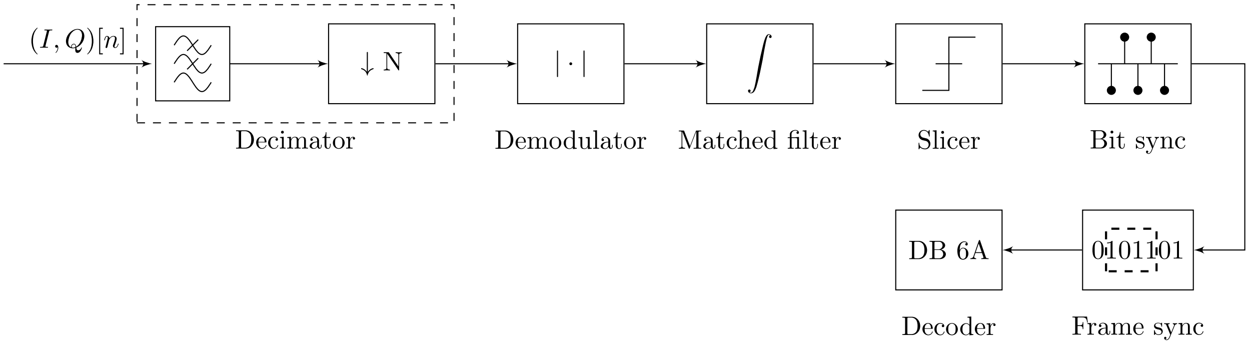


Figure 3: Process flow block diagram

Each of these steps will be detailed in their own section.

3.1 Acquiring the IQ samples

There are multiple ways to acquire the IQ samples. You can either write your own python script for this using the pyrtlsdr library[2], or you can use 3rd party software that have a recording function. Personally I use both methods, depending on what I'm doing at that moment. The software I sometimes use is called "SDR Console V3"[3]. It's quite popular, but there are others such as SDR Sharp[4] which is among the most populars.

Using SDR Console V3, I recorded a radio transmission, and plotted it with matplotlib. By default, SDR Console records with a sampling frequency Fs = 2Msps, at full LNA Gain.

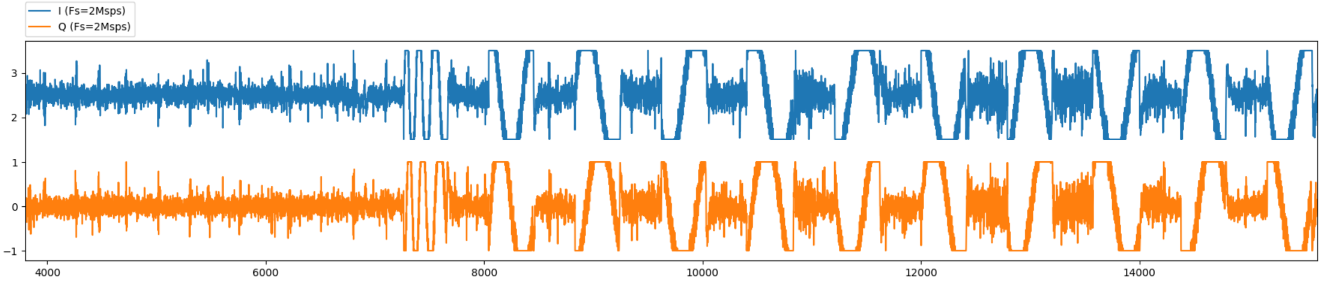


Figure 4: IQ samples

Figure 4 shows the recorded IQ signal. The result is interesting: IQ signals are supposed to be portions of sinusoid in quadrature with each others, which is exactly what we see here. Furthermore, we can see "blanks" between our portions of sinusoid. This result is expected: our transmitter uses the OOK modulation. OOK stands for "ON-OFF Shift Keying". In other words, when the transmitter wants to send a '1', it turns on the RF output and when it wants to send a '0', it turns it off. Without any kind of encoding, a blank means a '0' and a portion of sinusoid means a '1'.

In the following sections we won't use the complete recording: the keyfob I used transmits the same message several times when I press a button. I've selected only the equivalent of one message for the sake of graphic readability.

3.2 Decimation

So far our IQ signals have been recorded with a sample rate $Fs = 2Msps$. But we don't need such a high sampling rate: our transmitter uses an OOK modulation at 5kbps so theoretically we could do with only 10kHz of bandwidth. In practice, we need a bit more than that to ensure a sufficient signal quality. For our case study, 200kHz would be more than enough.

Decimation is the process of reducing the bandwidth by dropping samples. Decimating a signal by a factor D is equivalent to dropping one out of every D samples in the original signal, or equivalent to reducing the sampling rate to Fs/D.

In reality, it's a bit trickier than that. The process of reducing the sample rate is actually called downsampling. But if we stop there we might have a problem, which we can illustrate in Figure 5.

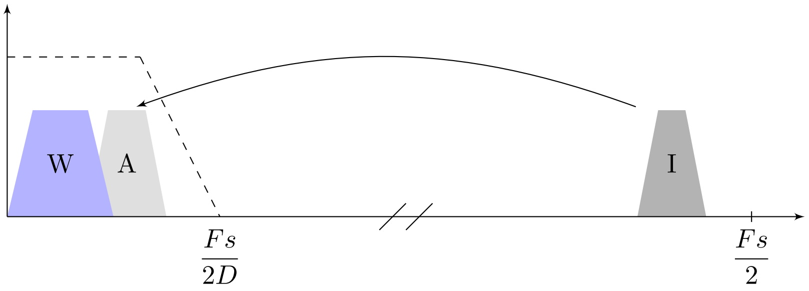


Figure 5: Aliasing

Say we are sampling at Frequency Fs: the Nyquist bandwidth will extend from DC to $Fs/2$, this is known as the Nyquist-Shannon sampling theorem. This theorem states that in order to sample a signal correctly, the sampling frequency must be superior to twice that signal's bandwidth[8].

In our example of Figure 5, there are two signals in the Nyquist bandwidth: our wanted W and an interferer I. If we reduce our sampling rate to Fs/D, our new Nyquist bandwidth now extends from DC to $Fs/(2D)$. W is in that bandwidth, so it's fine. I, however, is now *undersampled*. Because of that, its spectrum could be folded back inside our new Nyquist bandwidth. This is a famous phenomenon known as *Aliasing*. Surely most electronics engineers who have played with digital oscilloscope have encountered this foe: you're measuring a sinewave that reads at a certain frequency when in fact you're using the wrong timebase and the sinewave actually has a completely different frequency. But the scope is still showing you a beautiful sinewave either way.

In order to get rid of potential Aliases, we must first implement a low-pass filter (otherwise known as anti-aliasing filter) prior to the downsampling operation. This filter is represented by the dashed lines on Figure 5. The combination of low-pass filtering + downsampling is what we ultimately call *Decimation*.

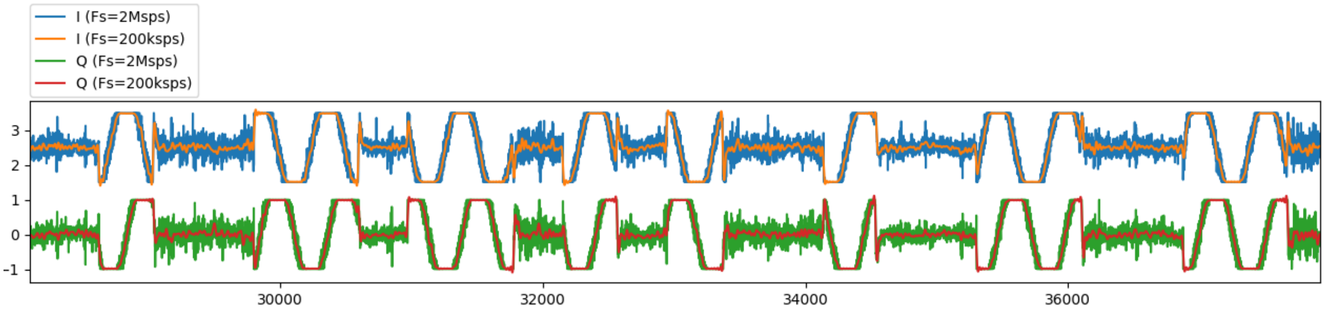


Figure 6: Decimated IQ

Figure 6 shows us the effect of decimating our signal by a factor of 10. Since we've reduced our bandwidth, we've gotten rid of a lot of noise. Thus we've increased our Signal-to-Noise Ratio, or SNR. This type of gain is called a *processing gain*, and its value in dB can be calculated by the formula:

$$G_{dB} = 10log_{10}\frac{BW_{old}}{BW_{new}} \tag{1}$$

which in our example is $10log_{10}(2MHz/200kHz) = 10dB$. So just like that, we've reduced our noise floor by 10dB, neat. Additionally, since we now have 10x less samples, it will result in a decreased processing load for the next steps.

But then we might ask ourselves: why go through all this trouble? Since we're using an SDR, why don't we just set the SDR to sample at 200ksps directly instead of going through all that nonsense? We could indeed do that, but then we'd be missing out on a couple of things. The trivial one is that we wouldn't be using the anti-aliasing filter to get rid of potential close-range interferers. The other one is less obvious and concerns the ADC inside the RTL2832 that samples our incoming signal.

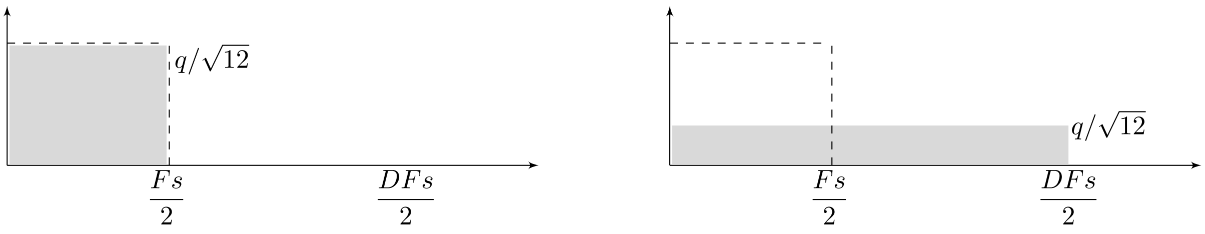
Theory tells us that for an N-bit ADC, the maximum achievable SNR will be[5]:

$$SNR = 6.02N + 1.76dB \tag{2}$$

The RMS quantization noise for a perfect classical N-bit ADC is equal to $q/\sqrt{12}$ (q being an LSB), and that noise is spread uniformly over the Nyquist bandwidth. If the ADC is not perfect and the noise is higher than the theoretical minimum level, then the ADC's effective resolution will be less than N bits. For example, an 8-bit ADC could only have 7 "useful" bits due to 1 bit lost to noise. This leads us to define a very important ADC figure of merit: the ENOB, for Effective Number Of Bits. ENOB can be defined as:

$$ENOB = \frac{SNR + 1.76dB}{6.02} \tag{3}$$

If the SNR is ideal (equal to Eq.2), then $ENOB = N$. The interesting thing about quantization noise is that it is independant of the sampling rate. If the sampling rate is higher the quantization noise is still $q/\sqrt{12}$, it's just spread over a wider bandwidth. The advantage of oversampling now becomes clearer: oversampling means we are spreading the quantization noise, and if we follow by decimation (i.e bandwidth reduction), we're effectively reducing the ADC's quantization noise. We are then able to do high-resolution conversions with a low-resolution ADC. This is illustrated in Figure 7.



(a) Quantization noise before spreading

(b) Quantization noise after spreading

Figure 7: Effect of oversampling on quantization noise

Now that've got our IQ signals recorded and decimated, we're going to be able to demodulate our message. Stay tuned (pun intended), as this will be covered in the next article.

References

[1] Pypl ranking. http://pypl.github.io/PYPL.html.

[2] pyrtlsdr library. https://github.com/roger-/pyrtlsdr.

[3] sdr console v3. https://www.sdr-radio.com/Software/Version3.

[4] sdr sharp. https://airspy.com/download/.

[5] Travis F. Collins, Robin Getz, Di Pu, and Alexander M. Wyglinski. *Software Defined Radio for Engineers*. Artech House, 2018.

[6] Joao Ferreira. Intro to rtl-sdr part 1. http://ajoo.blog/intro-to-rtl-sdr-part-i-principles-and-hardware.html, january 2017.

[7] Realtek. R820t datasheet. https://www.rtl-sdr.com/wp-content/uploads/2013/04/R820T_datasheet-Non_R-20111130_unlocked1.pdf.

[8] Claude E. Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37, no.1:10–21, 1949.