Python Radio 13: Software Defined Radio

Simon Quellen Field

Simon Quellen Field

Follow

6 min read

·

Aug 31, 2024

Listen

Share

More

Python listens to the radio.

Press enter or click to view image in full size

Photo by the author

If you do an Internet search for “SDR receiver price” you will find a large number of items for sale, from about $15 to $300. For our purposes (since we want to receive low frequencies as well as those above 30 MHz) we want receivers that can reach at least as low as 100 kilohertz. I have seen one for $18 that gets as low as 10 kilohertz.

What makes these devices special is the software on your computer that controls them and decodes the signals.

Another search, “SDR radio software” gets you a large number of software packages, most of them free, with names like SDR#, HDSDR, SDR-RADIO.COM, SDR++, Linrad, GQRX, CubicSDR, and many more.

There is also a Python library called pyrtlsdr, installed by the command pip install — upgrade pyrtlsdr[lib].

Using this library, the following program will pop up a spectrum display centered around the 7032000-hertz signal from our RP2040 CW transmitter:

From pylab import \*

From rtlsdr import RtlSdr

Import numpy as np

Import scipy.signal as signal

Import peakdetect

Real\_center\_freq = 7.032e6

Offset = 200e3

Margin = 10e3

Sdr = RtlSdr()

Sdr.set\_direct\_sampling(1)

Sdr.sample\_rate = 225001

Sdr.center\_freq = real\_center\_freq – offset

Sdr.gain = ‘auto’

Num\_samples = sdr.sample\_rate

Samples = sdr.read\_samples(num\_samples)

Power, psd\_freq = psd(samples, NFFT=1024, Fs=sdr.sample\_rate, Fc=real\_center\_freq)

Power\_db = 10\*np.log10(power)

Maxima, minima = peakdetect.peakdetect(power\_db, psd\_freq, delta=1)

For mx in maxima:

F = mx[0]

dBm = mx[1]

print(”Peak at”, f, “of”, dBm, “dB”)

# Was this peak anywhere near our target frequency?

If f > real\_center\_freq-margin and f < real\_center\_freq+margin:

Print(”We see a peak at”, str(f))

Sdr.close()

Show()

We set direct sampling to 1 so that we can reach the low frequencies. Then we read a second of samples from the receiver and plot them:

Press enter or click to view image in full size

Image by the author

The output of our program on the console looks like this:

Found Rafael Micro R820T/2 tuner

Enabled direct sampling mode, input 1

Exact sample rate is: 225001.000804 Hz

Peak at 6983879.6687734295 of -77.32257246972893 dB

Peak at 7032000.0 of -70.93046957472468 dB

We see a peak at 7032000.0

We chose the smallest sample rate that the RTL-SDR is capable of. It can sample in two ranges: 225 kHz to 300 kHz and 900 kHz to 2032 kHz (although rates higher than about 2.4 MHz drift unreliably). In the RTL-SDR, the sample rate is also the bandwidth, and for CW we want a narrow bandwidth (usually less than a kilohertz, but 225 kHz will do for now).

We can zoom in on the spectrum to see more detail:

Press enter or click to view image in full size

Image by the author

Let’s look at some of the aspects of the code that are not immediately obvious.

We set direct sampling because we are looking at the low frequencies (below 30 MHz). Without that, we can’t reach those frequencies.

We have an offset of 200,000 hertz. The RTL-SDR samples at the center frequency we give it, and returns us an array of samples centered at zero. This would give us what is called an “artifact”, something that looks like a signal but is not actually there in the real world. By offsetting the center frequency by a couple of hundred kilohertz, we can get rid of this artifact.

The actual collecting of the samples is done by read\_samples(). We could plot the array it returns to get an oscilloscope trace of our signal (we say that data is in the time domain). But we want to see the data in the frequency domain, so we use the “power spectral density” method psd() to get a Fast Fourier Transform (FFT) of the data.

Again, we could simply plot that (in fact, we do plot that on the last line of the program). But first, we want to find the signal peaks in the data, to see if the signal from our CW transmitter is there. We use the peakdetect() method to do that. It returns arrays of tuples that contain the frequency and the power of each peak.

We scan through that data to locate our target signal and print it out.

Another way to view the signals is with a waterfall display. Here is our RP2040 transmitter sending out Morse code CW signals:

Press enter or click to view image in full size

Image by the author

The code to do this is mostly involved with the graph itself, but the radio portion is fairly simple. As we did previously, we get the samples and convert them to a power spectral density plot with the psd() method. But then instead of drawing a line graph, we use the data to paint a bar of colors, with the lighter colors indicating higher power. We continue painting these bars of color one after another and when we get to the bottom of the image, it scrolls.

The code is shown below. We use a bandwidth of a megahertz centered around 28 MHz:

Import matplotlib.animation as animation

From matplotlib.mlab import psd

Import pylab as pyl

Import numpy as np

Import sys

From rtlsdr import RtlSdr

NFFT = 1024\*4

NUM\_SAMPLES\_PER\_SCAN = NFFT\*4

NUM\_BUFFERED\_SWEEPS = 100

Class Waterfall(object):

Keyboard\_buffer = []

Shift\_key\_down = False

Image\_buffer = -100\*np.ones((NUM\_BUFFERED\_SWEEPS, NFFT))

Def \_\_init\_\_(self, sdr):

Self.fig = pyl.figure()

Self.sdr = sdr

Self.init\_plot()

Def init\_plot(self):

Self.ax = self.fig.add\_subplot(1,1,1)

Self.image = self.ax.imshow(self.image\_buffer,

Aspect=’auto’, interpolation=’nearest’, vmin=-50, vmax=10)

Self.ax.set\_xlabel(’Current frequency (MHz)’)

Self.ax.get\_yaxis().set\_visible(False)

Def update\_plot\_labels(self):

Fc = self.sdr.fc

Rs = self.sdr.rs

Freq\_range = (fc – rs/2)/1e6, (fc + rs\*(0.5))/1e6

Self.image.set\_extent(freq\_range + (0, 1))

Self.fig.canvas.draw\_idle()

Def update(self, \*args):

Start\_fc = self.sdr.fc

Self.image\_buffer = np.roll(self.image\_buffer, 1, axis=0)

For scan\_num, start\_ind in enumerate(range(0, NFFT, NFFT)):

Self.sdr.fc += self.sdr.rs\*scan\_num

Samples = self.sdr.read\_samples(NUM\_SAMPLES\_PER\_SCAN)

Psd\_scan, f = psd(samples, NFFT=NFFT)

Pwr = 10 \* (np.log2(psd\_scan)/np.log2(8))

Self.image\_buffer[0, start\_ind: start\_ind+NFFT] = pwr

Self.image.set\_array(self.image\_buffer)

Self.sdr.fc = start\_fc

Return self.image,

Def start(self):

Self.update\_plot\_labels()

Ani = animation.FuncAnimation(self.fig, self.update, interval=50, save\_count=64\*1024, blit=True)

Pyl.show()

Return

Def main():

Sdr = RtlSdr()

Wf = Waterfall(sdr)

Sdr.rs = 1.0e6

Sdr.fc = 28000000

Sdr.gain = ‘auto’

Wf.start()

Sdr.close()

If \_\_name\_\_ == ‘\_\_main\_\_’:

Main()

We turned off Farnsworth sending in our RP2040 code so that we could more easily see the signals in the waterfall. If we send the CW very slowly (something called QRSS mode), we can use a very narrow bandwidth and receive weak signals very far away, since the narrow bandwidth gives us a better signal-to-noise ratio. Instead of listening to the Morse code, we watch it scroll down the waterfall display:

From cwmorse import CWMorse

From time import sleep

Frequency = 28050000

Def main():

Cw = CWMorse(15, frequency)

Cw.farnsworth(False)

Cw.speed(0.1)

Print(”CW transmitter”)

Msg = “This is AB6NY testing RP2040 as a 40 meter transmitter sending on “ + str(frequency) + “ Hertz.”

While True:

Print(msg)

Cw.send(msg)

Sleep(5)

Main()

The cwmorse.py code is modified slightly to allow turning off the Farnsworth method:

From machine import Pin, PWM

Class RP\_CW:

Def \_\_init\_\_(self, carrier\_pin, freq):

From machine import Pin

From rp2 import PIO, StateMachine, asm\_pio

@asm\_pio(set\_init=PIO.OUT\_LOW)

Def square():

Wrap\_target()

Set(pins, 1)

Set(pins, 0)

Wrap()

Self.carrier\_pin = Pin(carrier\_pin, Pin.OUT)

Self.f = freq

Self.sm = StateMachine(0, square, freq=2\*self.f, set\_base=self.carrier\_pin)

Self.sm.active(1)

Def on(self):

Self.sm.active(1)

Print(”#”, end=”)

Def off(self):

Self.sm.active(0)

Print(” “, end=”)

Def frequency(self, frq):

Self.f = frq

Class CWMorse:

Character\_speed = 20

Def \_\_init\_\_(self, carrier\_pin, freq):

Self.cw = RP\_CW(carrier\_pin, freq)

Self.cw.frequency(freq)

Self.farns = True

Def farnsworth( self, on\_off):

Self.farns = on\_off

Def speed(self, overall\_speed):

Print(”Farnsworth is”, self.farns)

If overall\_speed >= 20 or self.farns == False:

Self.character\_speed = overall\_speed

Units\_per\_minute = int(self.character\_speed \* 50) # The word PARIS is 50 units of time

OVERHEAD = 2

Self.DOT = int(60000 / units\_per\_minute) – OVERHEAD

Self.DASH = 3 \* self.DOT

Self.CYPHER\_SPACE = self.DOT

If overall\_speed >= 20 or self.farns == False:

Self.LETTER\_SPACE = int(3 \* self.DOT) – self.CYPHER\_SPACE

Self.WORD\_SPACE = int(7 \* self.DOT) – self.CYPHER\_SPACE

Else:

# Farnsworth timing from <https://www.arrl.org/files/file/Technology/x9004008.pdf>

Farnsworth\_spacing = (60000 \* self.character\_speed – 37200 \* overall\_speed) / (overall\_speed \* self.character\_speed)

Farnsworth\_spacing \*= 60000/68500 # A fudge factor to get the ESP8266 timing closer to correct

Self.LETTER\_SPACE = int((3 \* farnsworth\_spacing) / 19) – self.CYPHER\_SPACE

Self.WORD\_SPACE = int((7 \* farnsworth\_spacing) / 19) – self.CYPHER\_SPACE

Def send(self, str):

From the\_code import code

From time import sleep\_ms

For c in str:

If c == ‘ ‘:

Self.cw.off()

Sleep\_ms(self.WORD\_SPACE)

Else:

Cyphers = code[c.upper()]

For x in cyphers:

If x == ‘.’:

Self.cw.on()

Sleep\_ms(self.DOT)

Else:

Self.cw.on()

Sleep\_ms(self.DOT)

Self.cw.on()

Sleep\_ms(self.DOT)

Self.cw.on()

Sleep\_ms(self.DOT)

Self.cw.off()

Sleep\_ms(self.CYPHER\_SPACE)

Self.cw.off()

Sleep\_ms(self.LETTER\_SPACE)

Software Defined Radio

Rp2040