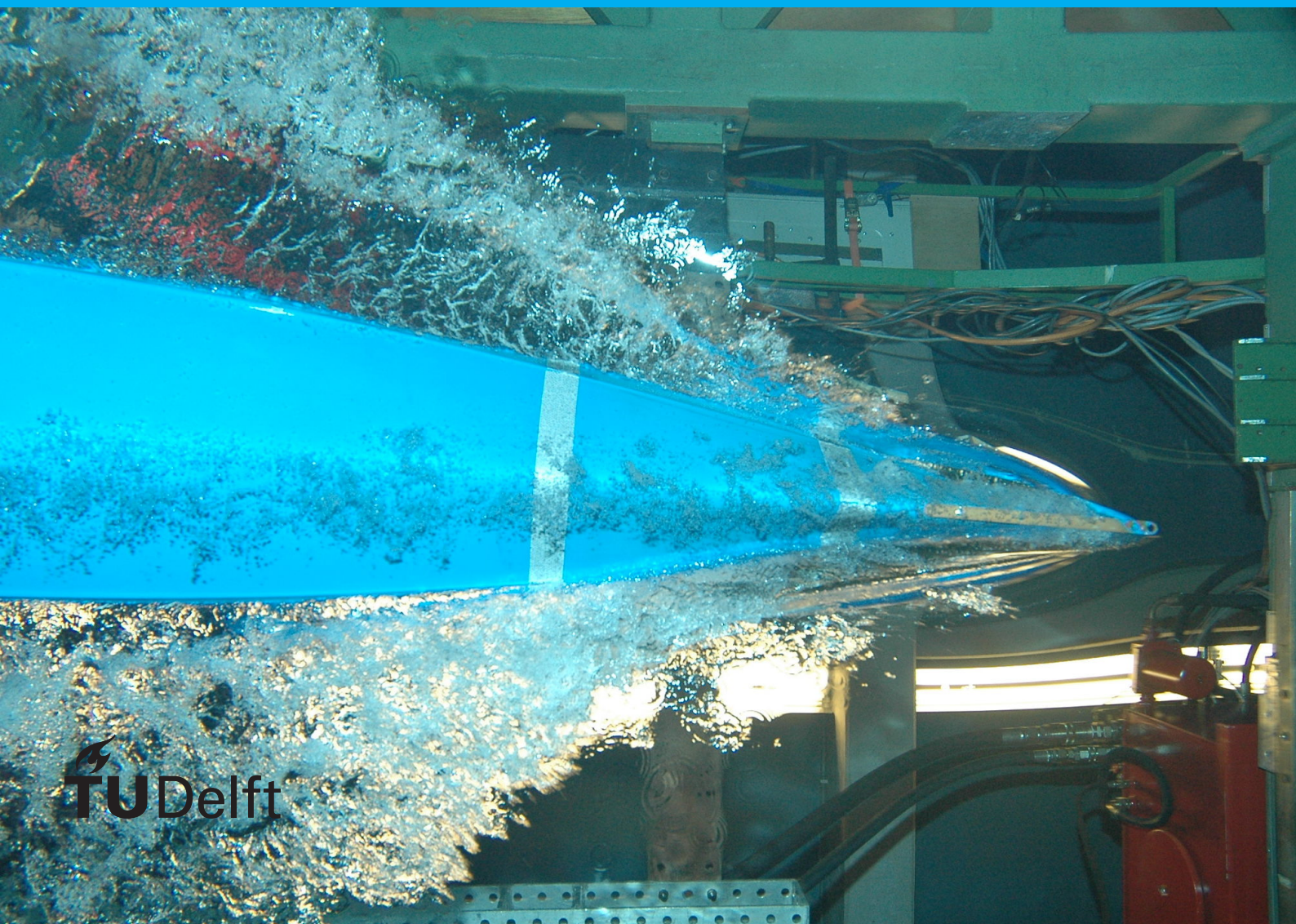
A FSK-decoder in GNU Radio

J.B Miog



A FSK-decoder in GNU Radio

by

J.B Miog

.

Student number: 4336313
Date: April 2016

An electronic version of this report and the project files are available at
`https://github.com/JBmiog/ET4394-Wireless-Networking`.

TU Delft

Contents

1 Introduction

1.1. Assignment objective

The objective of this project is to receive and decode two FSK-transmitters both transmitting in the receiving frequency bandwith of a Software Defined Radio dongle. The transmitter modules that are being used have been build by the author in order to fulfill the degree of bachelor of applied sciences in 2013. The transceiver module is designed at, and currently produced by, T-Minus engineering B.V. The PCB contains two chips, an Atmega Arduino compatible micro controller and a transceiver chip, the Murata TRC105. The module can be connected to a computer via USB, after which a Arduino sketch can be uploaded.

At the moment, this transceiver is used by ESA during the CanSat competition, in which multiple high-school teams create mini satellites the size of a soda can. The mini satellites are launched with a rocket, up to a height of 1,5 kilometer. Then, the satellites are decoupled and start descending using a parachute. During descending, measurements are being done by the mini satellite. In order to simulate a space mission as good as possible, the data of the measurements is to be wireless transmitted during the 'mission'. This wireless link has been the primary reason to develop the transmitters.

While the teams on the ground aim Yagi antenna's on their CanSat's to receive the signal, every once in a while, they don't succeed in receiving anything. The SDR is used, together with GNU-radio, to design a proof of concept, showing that multiple FSK signals can be received and decoded at one receiver.

The goal of this project is to build a receiver which can listen to the channels of each team simultaneously. After receiving, the data has to be decoded into the initial ASCII-data string.

2

Transmitter

The Transceiver PCB's contain a couple of indicator LED's, a reset switch, and some other trivial components. The schematic of the RF part is depicted in 2.1. The TRC105 is an UHF multi-channel controller, which is configured through an SPI interface by a Atmel 16u2 micro controller on thesame PCB. This micro controller, is used as a serial to SPI converter once the module is connected by USB to a computer. It is then possible to send and receive wireless data using the common serial port on the desktop.

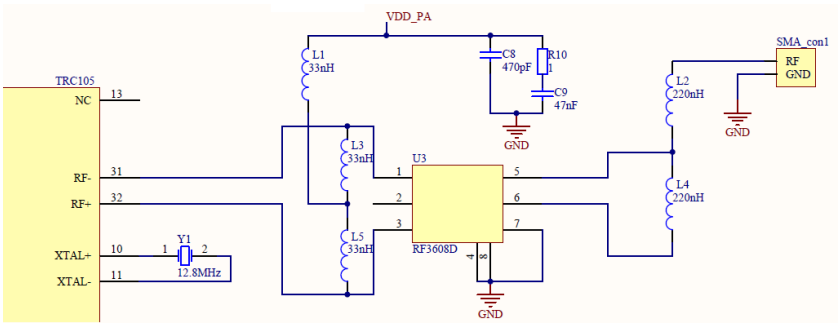


Figure 2.1

The atmega 16u2 can be connected to an Atmega2560 used as the main processing unit in the CanSat competition. Instead of printing serial data from the main processing unit to the computer through USB, this data can now be "printed" to the transceiver module, which will wireless forward the data to another receiving transceiver, and hence, the two-way wireless link is created.

Two transceivers will be used during this project, both are set to transmitting mode only. They are named 'Buenos aires' and 'Pijnacker' respectivily. The transmitter settings are show in the table below.

	Tx Freq	Deviation	Mode	Tx Data	baudrate	Tx power
Buenos Aires	432.9mHz	+/- 20kHz	FSK	'Pijnacker'	1200 symb/sec	-8dBm
Pijnacker	433.62	+/- 20kHz	FSK	'Buenos Aires'	1200 symb/sec	-8dBm

The signal that will be transmitted will contain a preamble of 4 bytes (0xAA). This is followed by the transmitter identifiers, which are the same for both modules. The identifier is "TMCS" (0x54, 0x4D, 0x43, 0x53), followed by a node address byte (0xXX).

3

Receiver

In order to receive both nodes simultaneously, an Ezcap DVB-T FM DAB receiver is used, which contains the RTL2832 chip. The SDR dongle is connected to the computer by USB2.0, and the I&Q data is imported using the GNU-Radio software.

3.1. Capturing the two singals

First, figure 3.1 is obtained by a direct recording in order to measure the frequency offset. The left signal is produced by Buenos Aires, and should be at center frequency 432.990 mHz, but as we can see, it is actually received at 432.964 mHz. The difference is caused by the initial offsets of the transmitters, and the frequency offset of the VCO in the SDR.

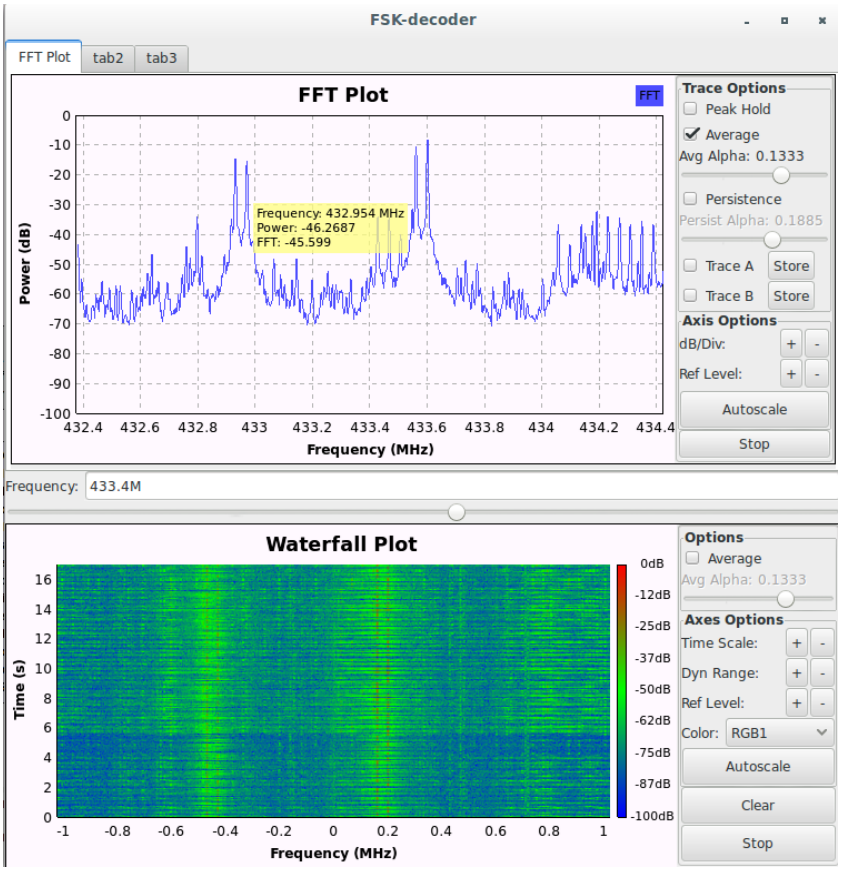


Figure 3.1: a gnu-radio capture of the frequency spectrum

3.2. Save as file

In order to build the decoder efficiently, the FSK_capture_to_file.grc is build, which contains a simple data sink block. The sampling frequency is set to 2.048 MSPS, which results in $2.048 MSPS/1200 baud \approx 1706 samples/symbol$. This is useful to know in order to perform a correct clock recovery later on.

The input data is measured with a center frequency of 433.4 MHz. We'll take a graphical look at the recorded signal of both transmitters using the GNU radio flowchart depicted in 3.2

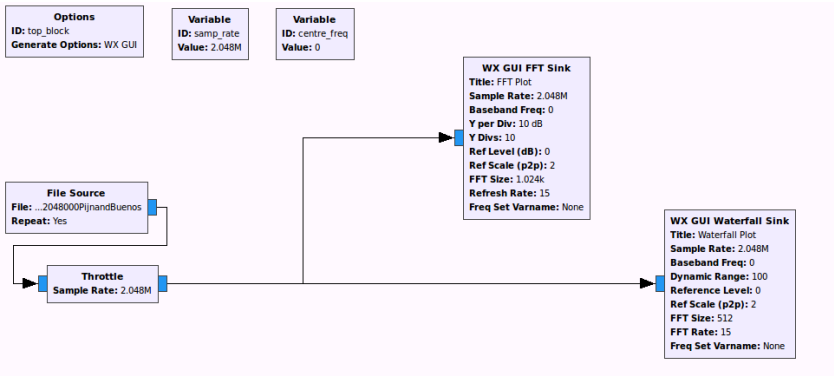


Figure 3.2

We expect that Pijnacker transmitting frequency is now 433.4 - 432.950 = 450 kHz below the current center frequency. Which is roughly shown in 3.3.

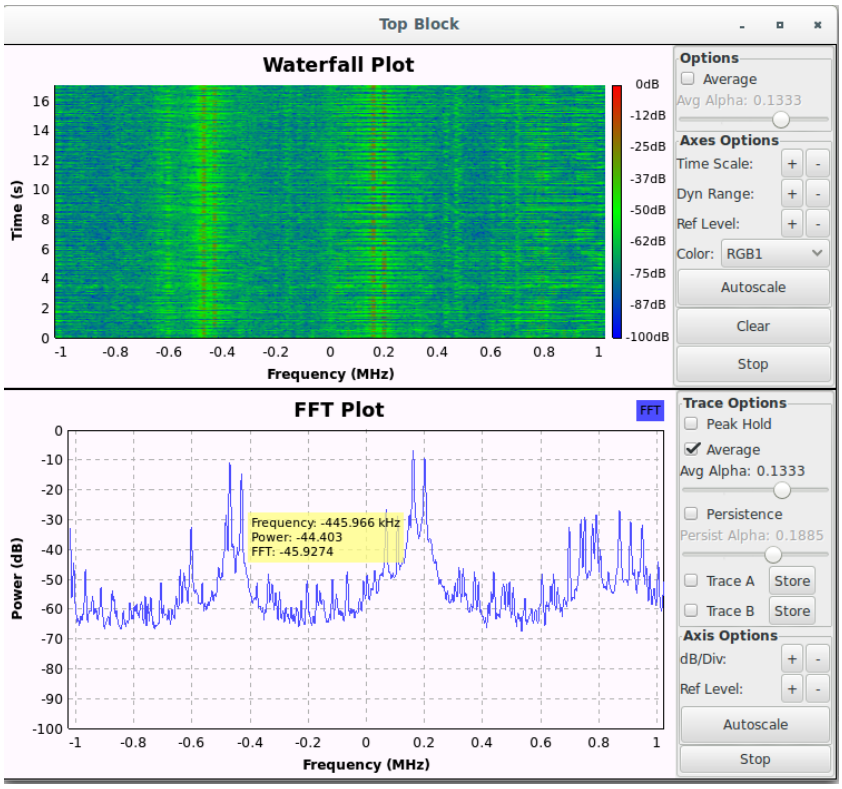


Figure 3.3: replay of signal

Once we replay the saved data using the file source block, the frequency of Pijnacker is at -450kHz, which can be seen in figure 3.3.

3.3. Filtering the signal

In order to decode the signal, the frequency is increased using a multiplier and an addition signal source block, with a frequency of 470 kHz. This sets the '0' at + 30kHz, and the '1' at 50 kHz. Then, the signal is passed through a low pass filter. The flowchart and the results are depicted in 3.4.

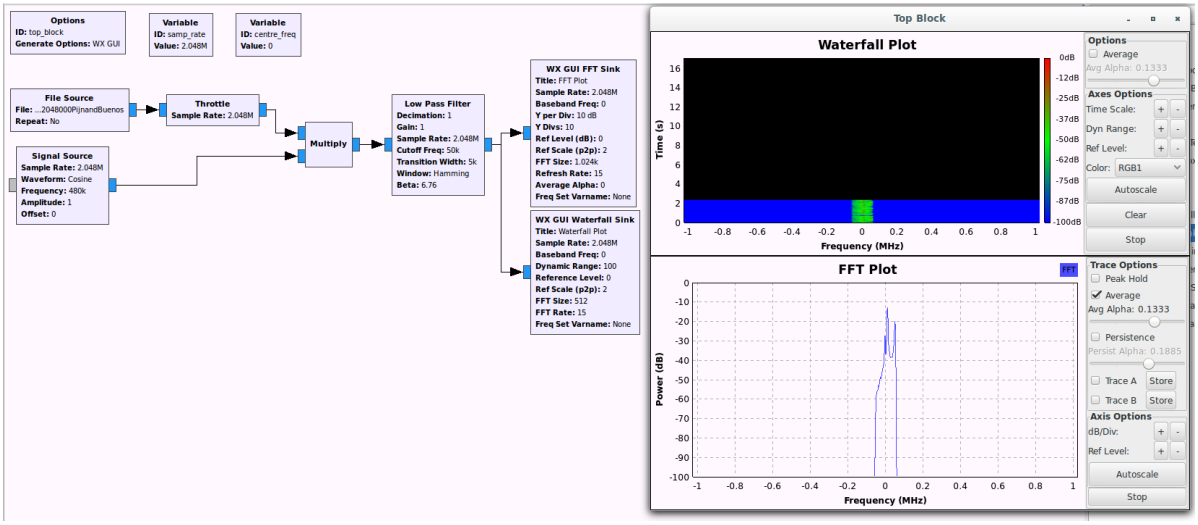


Figure 3.4: filtered signal of Pijnacker

3.4. quadrature demodulating

From this point, we could use the GFSK decode block already available in the GNU-radio toolbox. However, this block only supports binary data output which is unfortunately not easy readable. To gain a better understanding in the underlying principles, it is decided to use a quadrature demodulator, and decode the signal in a binary AM fashion. After adding the quadrature demodulator, another low pas filter is used. with a cutt-off frequency of 1200 hertz. As the baud rate is 1200 symbols a second, and each symbol represents a binary '1' or a binary '0', we know that a signal representing a bit in AM can never be at a frequency higher than 1200 hertz. However, the filter is not infinitively steep, and therefor the cutoff frequency is set to 1100 hertz with a 800 hertz transition width. Making the filter steeper takes more processing power and imposes a larger delay, but can be done if deemed necessary. The flowchart and the output up to this point are depicted in figure 3.5.

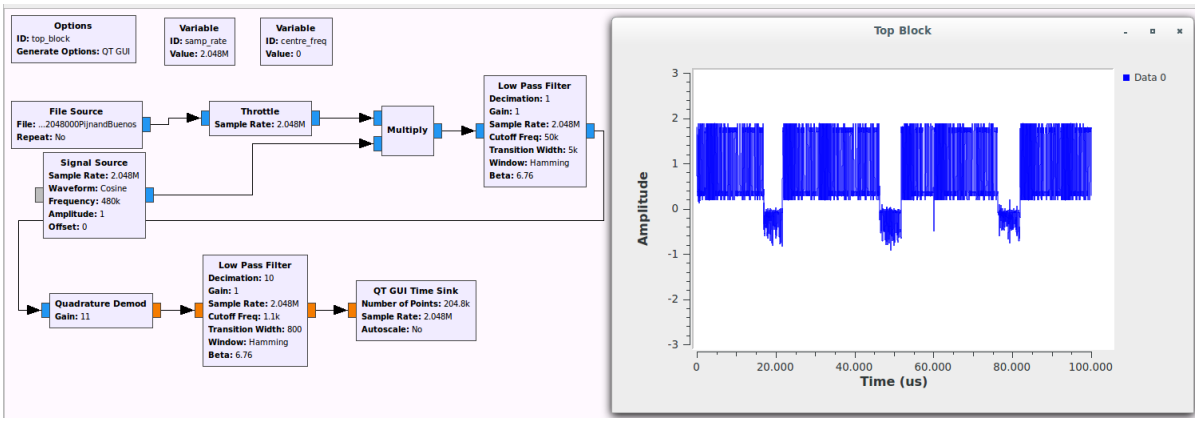


Figure 3.5: Qaudrature demodulation with noise

3.5. Interpacket noise removal

To get rid of the noise in between the data frames, a simple threshold detector is used. The output of this threshold detector is multiplied with the output of the second low pass filer. The flowchart and the result

with the removed interpacket noise are depicted in 3.6

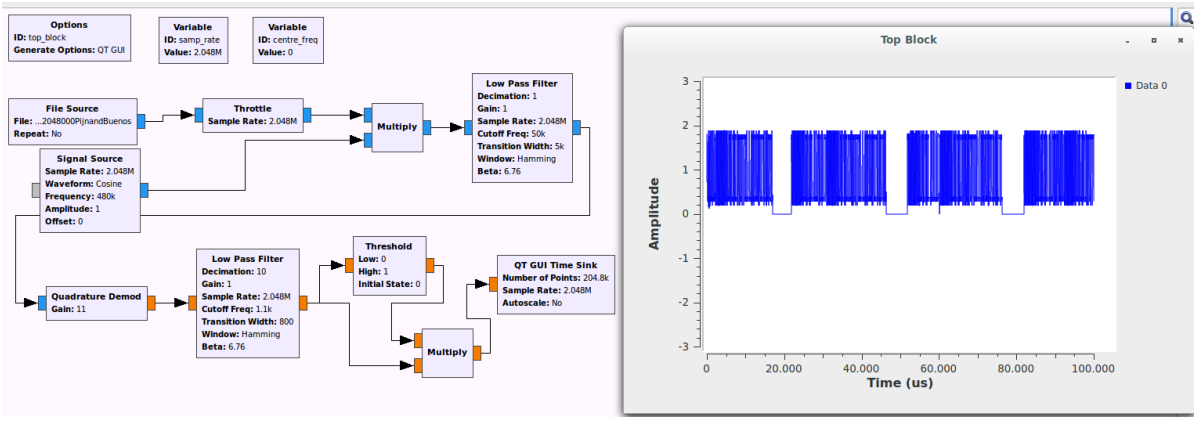


Figure 3.6: inter packet noise removed

3.6. Clock recovery

In order to recover the clock signal from the preamble, a Muller and Mueller clock recovery block *Clock_recovery_MM* block is available in the toolbox. This block is not well documented, but the mailing list of gnu-radio provided the information necessary[1]:

Omega: this is the expected number of samples per symbol (SPS) which is 1706.
Gain omega: Control loop value indicating how fast adjustment in the omega value can be made.
Mu: Initial estimate of the phase of the sample.
Gain mu: Control loop value indicating how fast adjustment in the mu value can be made.
Omega Relative Limit: Percentage of maximal allowed omega deviation.

The output, after the clock recover block is show in figure 3.7. Each positive value corresponds to a '1' being detected, a negative value corresponds to a '0' being detected.

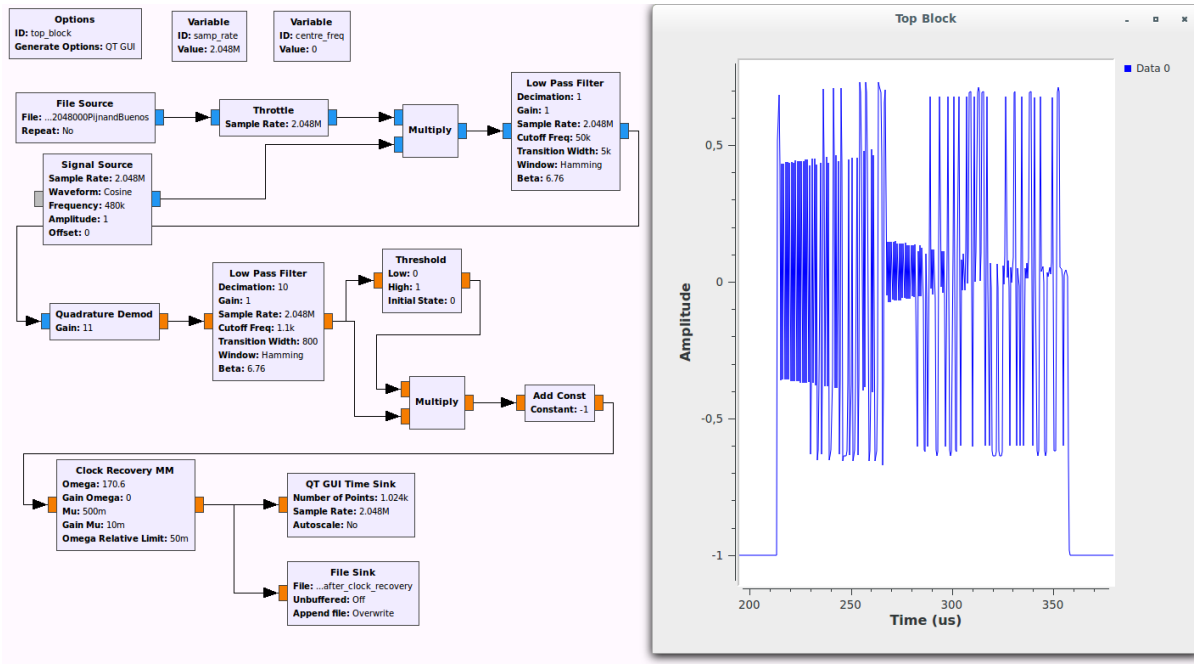


Figure 3.7: data after clock recovery

3.7. Data extraction

In order to clip the amplitude of this output to a '0' or a '1', the binary slicer is used. After the data is appropriately visible, we can add tags using the "correlate access code" tag block which we provide with the binary representation of "TMCS". For convenience, the time raster sink block is added to display the data in a longer run. The output of this block is show in the bottom graph in 3.8. Yellow represents a '1' bits detected, and in white the detection of a '0' bit is visible. Unfortunately, the default background color is also yellow. Note that the completely yellow block at the end thus actually indicates data that is not yet received. The graph on the top shows the tags as small red triangles. These are placed upon the bit stream by the correlate access block.

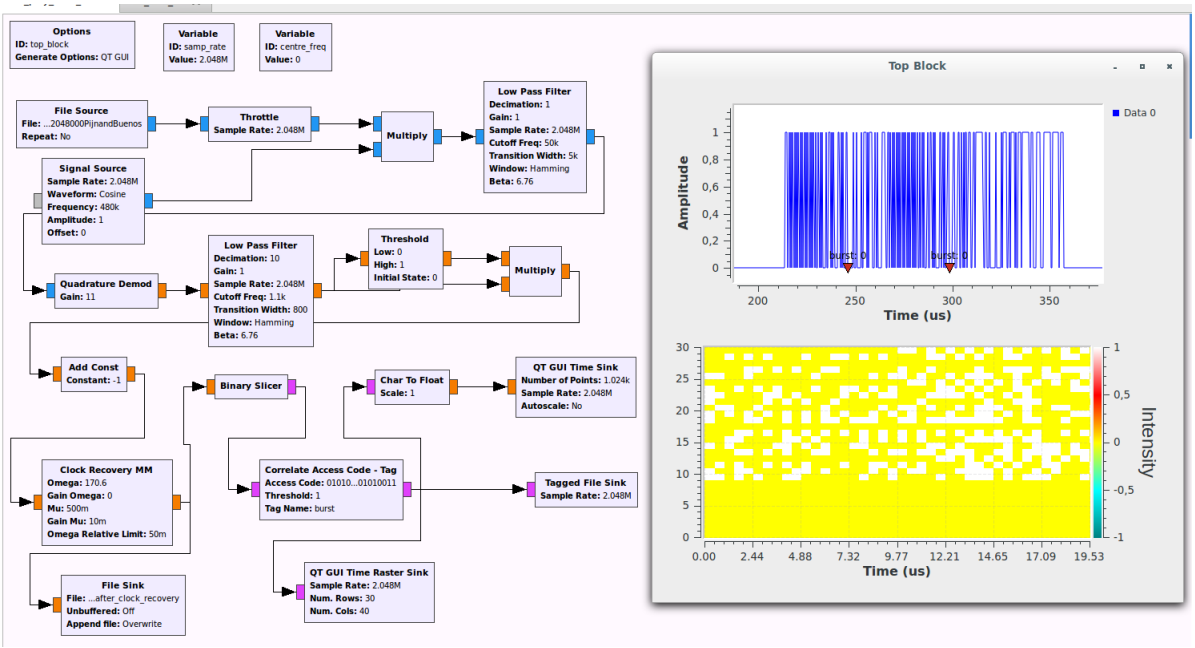


Figure 3.8: data after clock recovery, with amplitude correction and preamble tagging

At this point, also the data is stored in the tagged file sink. The documentation of this block is unclear where the obtained files are actually located, but does specify that each block after a tag detection creates a new file. As this is not eligible for our goal, the author has decided to use the file obtained in fig 3.7 and write an own script in python to detect the preamble, and further decode the message offline.

3.8. Off-line data recovery script

The python script "fsk_decode_to_char" also implements a binary slicer, appends the '0' and '1' to a string, and searches this string for the data, which it extracts, converts to ascii characters, and outputs as data.

The script skips the first two characters of the transmitted data. It is expected that this is due to the node identifier, as this is not yet taken into account.

```
import matplotlib.pyplot as plt
import scipy
import binascii
import re
import os

dir = os.path.dirname(__file__)

#method copied from stackoverflow
# http://stackoverflow.com/questions/7396849/convert-binary-to-ascii-and-vice-versa
def text_to_bits(text, encoding='utf-8', errors='surrogatepass'):
bits = bin(int(binascii.hexlify(text.encode(encoding, errors)), 16))[2:]
return bits.zfill(8 * ((len(bits) + 7) // 8))

#method copied from stackoverflow
# http://stackoverflow.com/questions/7396849/convert-binary-to-ascii-and-vice-versa
def text_from_bits(bits, encoding='utf-8', errors='replace'):
n = int(bits, 2)
return int2bytes(n).decode(encoding, errors)
```

```
#method copied from stackoverflow
# http://stackoverflow.com/questions/7396849/convert-binary-to-ascii-and-vice-versa
def int2bytes(i):
hex_string = '%x' % i
n = len(hex_string)
return binascii.unhexlify(hex_string.zfill(n + (n & 1)))

def get_frame_start_and_end(data,preamble,end_of_packet_sequence):
return -1

def get_binairy_data_from_input(data):
bin_data = '0b'
for x in range (1,len(data)):
if(f[x]) < 0:
bin_data = bin_data + '0'
else:
bin_data = bin_data + '1'
return bin_data

def get_relative_end_of_packet(data, end_of_packet):
end = data.find(end_of_packet)
if(end ==  -1):
return  -1
else:
return end

#filename = '../isolated_signals/Buenos_isolated_after_clock_recovery'
dir = os.path.dirname(__file__)
filename = os.path.join(dir, 'GNR_isolated_signals/Buenos_isolated_after_clock_recovery')
print("hallo")
print(filename)
print ("reading␣file:␣%s")%filename

f = scipy.fromfile(open(filename),dtype=scipy.float32)

bin_data = get_binairy_data_from_input(f)

print("binairy␣data:␣%s")%bin_data

#preamble = text_to_bits('TMCS') + '00001000'
preamble = text_to_bits('TMCS')
print("preamble␣sequence␣=␣%s")%preamble

#packet ends with a carriege return (0x0D) and a line feed(0x0A)
end_of_packet = "00001101" + "00001010"
print("end␣of␣packet␣sequence␣=␣%s")%(end_of_packet)

#found at stack overflow, efficient lookup:
#http://stackoverflow.com/questions/4664850/find-all-occurrences-of-a-substring-in-python
x_start_of_packet = [m.start() + (len(preamble)) for m in re.finditer(preamble, bin_data)]

print("found␣preamble␣at␣string␣elements:")
print(x_start_of_packet)

for i in range(len(x_start_of_packet)):
rel_end = get_relative_end_of_packet(bin_data[x_start_of_packet[1]:], end_of_packet)
data = text_from_bits(bin_data[x_start_of_packet[1]:x_start_of_packet[1]+rel_end])
print data
```

Bibliography

[1] Burst fsk receiver clock synchronization problems. URL https://lists.gnu.org/archive/html/discuss-gnuradio/2015-12/msg00075.html.

[2] A. K. Geim and H. A. M. S. ter Tisha. Detection of earth rotation with a diamagnetically levitating gyroscope. *Physica B: Condensed Matter*, 294–295:736–739, 2001. doi: 10.1016/S0921-4526(00)00753-5.