Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Software Defined Radio with Remote Head and Internet Clients

Studente/i
Giovanni Franza

Relatore
Tiziano Leidi

Correlatore
Loris Grossi

Committente
Tiziano Leidi

Corso di laurea
Master of Science in Informatics

Modulo

Anno
2016

Software Defined Radio with Remote Head and Internet Clients

Giovanni Franza

To Marina,
without whose encouragement, support and patience,
I would never have arrived at this result.

To Franco,
for his great support
and availability,
(also of the devices he designed and produced).

To Alberto,
for his patience, provocations,
and precious suggestions.

Main Index

1 - Abstract......6
2 - Introduction......7
2.1 - Environment......8
2.2 - User Experience......9
3 - Objectives......11
4 - Work Organization......12
4.1 – State of the Art Analysis......12
4.2 - Hardware Market Analysis......13
4.3 - First Software Survey......14
4.4 - GNU Radio......15
4.5 - Initial Tests......16
4.6 - Development......16
5 – System Structure......18
5.1 - Sampler......19
5.2 - Embedded......21
5.3 - Connection Between Embedded and Server......21
5.4 - Protocols Between Embedded and Server......22
5.5 - Server......23
5.6 - Clients......23
6 - Implementation......24
6.1 - Samplers and Radio Management......24
6.2 - Sampler Management Program......26
6.3 - Central Control Program......35
6.4 - Control Browser......50
6.5 - Client Receiver......57
6.6 - Receiver User Interface......72
7 - Conclusions......81
7.1 - State of the Project and Future Options......81
8 - Bibliography......82
A - Appendix 1: GNU Radio......83
A.1 - GNU Radio Installation......83
A.2 - GNU Radio Architecture......84
B - Appendix 2 – GNU Radio Realizations......103
B.1 - Wide Band FM Monoaural Receiver......103
B.2 - Narrow Band FM Monoaural Receiver......111
B.3 - SSB Receiver......113
B.4 - Knob Commands......115

Pictures and schemes

Picture 1: A map of interferences from a Belgian amateur station..8
Picture 2: A spectrogram..9
Picture 3: Spectrogram (upper) and its relative waterfall (lower)..10
Picture 4: A traditional GUI to control a transceiver..10
Picture 5: Ettus Research X310 5.060 Euro and B210 1.160 Euro (March 2016)..13
Picture 6: Fairwaves Um TRX 2.2 950 $ and UmSITE-TM3 4000$ (March 2016)...13
Picture 7: Perseus 799Euro, bladeRF X115 650 $, and HackRF One Kit 330 $ (March 2016)............................13
Picture 8: ELAD S1 369 Euro, S2 525 Euro, DUO 1.159 Euro (August 2016)...13
Picture 9: A GNU Radio flow-graph that fully implements a 4QAM flow with FDM DUO.................................15
Picture 10: The structure of the system showing the connections between blocks...18
Picture 11: The dongle opened (left) and shielded (right)...19
Picture 12: Three ELAD devices: left to right, samplers S1 and S2, transceiver FDM DUO...............................20
Picture 13: The NanoBeam M5..21
Picture 14: The structure of the sampler management program..26
Picture 15: The I/Q data UDP packet...27
Picture 16: USB callback timing (green:avg & dev, red: too high max value)..32
Picture 17: The structure of the central control program...35
Picture 18: WebSocket life cycle..38
Picture 19: WebSocket initial exchange example...39
Picture 20: WebSocket frame structure..40
Picture 21: Blackman-Harris window..44
Picture 22: Blackman-Harris windowing function..44
Picture 23: The WebSocket FFT frame...46
Picture 24: Traffic on the server measured by nload..48
Picture 25: The control browser in action..50
Picture 26: The control browser application structure...51
Picture 27: Client receiver structure (without WebSocket thread)...57
Picture 28: FIR filter schematic...62
Picture 29: One (left) and two (right) delay line IIR filter schematics...63
Picture 30: Parameters, response, and coefficients of the first, 15 coefficients, HF filter.....................................65
Picture 31: Parameters, response, and coefficients of the second, 21 coefficients, HF filter.................................66
Picture 32: Design of audio IIR filter..67
Picture 33: Elad-server and elad-client aggregated bandwidth..70
Picture 34: A sample of HF thread elaboration times..70
Picture 35: A sample of FFT elaboration times..71
Picture 36: A screen shot of the receiver user interface...72
Picture 37: Web receiver user interface structure...74
Picture 38: Starting the flowgraph..92
Picture 39: Running a block...97
Picture 40: Picture 40: gr_modtool showing possible operations...98
Picture 41: Picture 41: Making module and block structure with gr_modtool...99
Picture 42: Picture 42: Editing QA code. Only test_001_modename() is inserted by hand...................................99
Picture 43: Picture 43: Editing C++ code...100
Picture 44: Picture 44: Generating XML code to integrate block into gnuradio_companion...............................101
Picture 45: GNU Radio companion showing the new module and block..102
Picture 46: GNU Radio companion WBFM block diagram...103
Picture 47: GNU Radio companion WBFM interface..104
Picture 48: Dongle without and with shield...105
Picture 49: GNU Radio companion NBFM block diagram..111
Picture 50: GNU Radio companion NBFM interface...112
Picture 51: GNU Radio companion SSB from file block diagram...113
Picture 52: GNU Radio companion SSB from file interface..114
Picture 53: GNU Radio companion NBFM with knobs block diagram..120
Picture 54: Arduino Uno manages two optical incremental encoder...121
Picture 55: GNU Radio companion NBFM with knobs GUI..121

Listings

Listing 1: Sampler management program spawning thread....27
Listing 2: Prepare USB Async transfer....28
Listing 3: Sampler management program: Callback....29
Listing 4: TCP server in sampler management Program....31
Listing 5: Callback with timing profiling instructions (red)....32
Listing 6: The changes needed to avoid sporadic data losses....34
Listing 7: Central control program: shared memory & condition variables creation....36
Listing 8: Central control program: UDP reading and shared memory filling....37
Listing 9: Central control program: WebSocket initialization....40
Listing 10: Central control program: WebSocket frame reception....41
Listing 11: Central control program: TCP command sending....42
Listing 12: Central control program: FFT coefficient precomputation....43
Listing 13: Central control program: FFT computing....45
Listing 14: Central control program: the composition of the WebSocket FFT frame....46
Listing 15: Central control program: the data loss detecting structure (red)....47
Listing 16: The instructions that measure the fft computation time (red)....49
Listing 17: The Web Worker with the WebSocket implemented....52
Listing 18: The canvas definition....52
Listing 19: Spectrogram background drawing....53
Listing 20: Spectrogram drawing....54
Listing 21: Waterfall drawing....55
Listing 22: Buttons management....56
Listing 23: Local Oscillator....61
Listing 24: Mixer (using prostapheresys formula)....62
Listing 25: FIR filter implementation....63
Listing 26: IIR filter implementation....64
Listing 27: Filter coefficents in the source code....65
Listing 28: Calls to the FIR filter implementation....66
Listing 29: IIR filter coefficients....67
Listing 30: Calls to IIR filter....67
Listing 31: The signal processing thread....69
Listing 32: The WebSocket thread....75
Listing 33: Definition of the graphic elements in the page using HTML....76
Listing 34: The functions fired by the buttons....77
Listing 35: Mouse handlers....78
Listing 36: Receiving audio data from WebSocket and rendering them....79
Listing 37: The context that fill the rendered data into the buffer when requested by audio card....80
Listing 38: A typical shell to install GNU Radio along with hw support software....83
Listing 39: Final part of Python generated by gnuradio-companion for WBFM receiver....84
Listing 40: Initial part of Python generated by gnuradio-companion for WBFM receiver....85
Listing 41: Partial view of top_block class in the …/grc_gnuradio/wxgui/top_block_gui.py file....86
Listing 42: Partial view of top_block class in the …/gnuradio/gr/top_block.py file....87
Listing 43: Partial view of top_block class in the …/gr/runtime_swig/runtime_swig.py file....87
Listing 44: Partial view of top_block class in the .../gnuradio-runtime/swig/top_block.i file....87
Listing 45: Rename the top_block_swig class in file .../gnuradio-runtime/swig/top_block.i....88
Listing 46: Partial view of top_block class in the .../gnuradio-runtime/lib/top_block.cc file....88
Listing 47: Start() method in the gr-sources/gnuradio-runtime/lib/top_block_impl.cc file....89
Listing 48: The make_scheduler() method in .../gnuradio-runtime/lib/top_block_impl.cc....90
Listing 49: Methods and constructor in the .../gnuradio-runtime/lib/scheduler_tpb.cc file....91
Listing 50: tpb_container in the /usr/local/include/gnuradio/thread/thread_body_wrapper.h file....93
Listing 51: tpb_container in the gr-sources/gnuradio-runtime/lib/scheduler_tpb.cc file....94
Listing 52: Definition in the gr-sources/gnuradio-runtime/lib/tpb_thread_body.h file....94
Listing 53: Constructor in the gr-sources/gnuradio-runtime/lib/tpb_thread_body.cc file....96
Listing 54: run_one_iteration() in gr-sources/gnuradio-runtime/lib/block_executor.cc....97
Listing 55: Python code generated by gnuradio-companion for WBFM receiver....110
Listing 56: Code to manage encoders on Arduino Uno board....115
Listing 57: Usbknobs module generation....116
Listing 58: Knobs block XML description....118
Listing 59: Knobs python class....119
Listing 60: Part of grc XML file with frequency variable_knobs definition....122

1 - Abstract

The goal of this project is the realization of a device that allows the reception of radio signals, their demodulation and characterization of their parameters. This project uses, in the “head”, a device for direct sampling of radio signals, with USB output. This "head" combines the device and an embedded computer to allow sending the samples from a remote position, difficult to reach, to a central processing unit via a radio link (e.g. IEEE811b/g/n). The “real” receiver is a web application, hosted on a web server in the central location, that can be reached via the Internet using a web browser running WebSockets. All this equipment is not only a “radio receiver" but it allows the exploration of the desired frequencies, highlighting the frequency spectrum and, through the use of an antenna and a calibration process, the measurement of various parameters of the received signal. All the developed software is licensed by the GPL and its structure allows an easy integration of “ad hoc” new elaborations.

2 - Introduction

With the availability of increasingly powerful processors we are witnessing an increased use of what is called SDR, or Software Defined Radio. The purpose of this approach is to replace as much as possible the analog processing with a corresponding numeric processing.
At present this processing is normally divided into two distinct areas, a first one that includes the filter for decimation of the samples and a second one that includes all the processes of fine filtering and demodulation.
In this approach the "head" of the device is made up of several analog filters, sometimes a converter, one or more ADC, and a FPGA. This "head" is the most tricky and expensive part of a receiver and often costs over 1000 Frs.
Although this approach was proven fruitful, it, together with the undeniable benefits of fast processing and safe marketing, also brings its own limitations due to being a specialized hardware. Such limits lie mainly in less flexibility and difficult inspection of the code itself, as well as in the cost of the device.
However today there are devices that can be found for an extremely low price, made for the DVB market, which can make available a stream of I/Q samples through a USB port, that can therefore be exploited in place of the most expensive "heads" of high quality available on the market.
In the first part of this thesis we explore the use of these devices, and in the second (the development of the software) we work with a most affordable direct sampling device built by ELAD, an Italian company located near Pordenone.
When approaching the project of a Radio, many issues have to be taken into account, and a prioritization must be done.
There are two distinct areas that, in my humble opinion, must be taken in serious consideration: the first affects the physical form of the “radio” itself, and is related to “where” the radio is, or, better, where the parts of the radio are. The relative considerations are presented in section 2.1 and refer to the current environment where radio amateurs follow their hobby.
The second area is tied to the fact that SDR has changed so much the capabilities of a radio receiver that its human interface has to be modified to reach the efficiency that these changes make possible. Thoughts about this area are presented in section 2.2.

2.1 - Environment

The rationale for the proposed work is the fact that nowadays many factors can cause a poor reception of HF radio signals. Our houses are literally filled with poorly screened devices, that, despite the well known anti RFI rules they are supposed to comply to, emit an overall high level of unneeded spurious signals all over the HF spectrum (and over...). In the neighborhood, especially in towns, there are lot of sources of RF noise, varying from street lights to obsolete industrial equipments. An example of this situation can be seen in Picture 1, built by a Belgian Radio Amateur. And this is only the beginning because, in our crowded countries, it is even more difficult to erect a reasonable size HF antenna, also for the fear of the radio waves spreader from unacculturated anti-GSM zealots. In this scenario a solution could be the remotization of the receiver. This has many benefits, ranging from the use of a remote, interference free, location to the hiding of the antenna from too curious neighbors.
Remotization of the receiver enables also the sharing of the receiver itself letting people use an arbitrary located device to hear what could be heard on different locations. This can be considered a little bit silly if we do not think at some interesting opportunities, like monitoring with receivers at different locations our transmitted signal as well as a beacon or a station we need to track.

Picture 1: A map of interferences from a Belgian amateur station

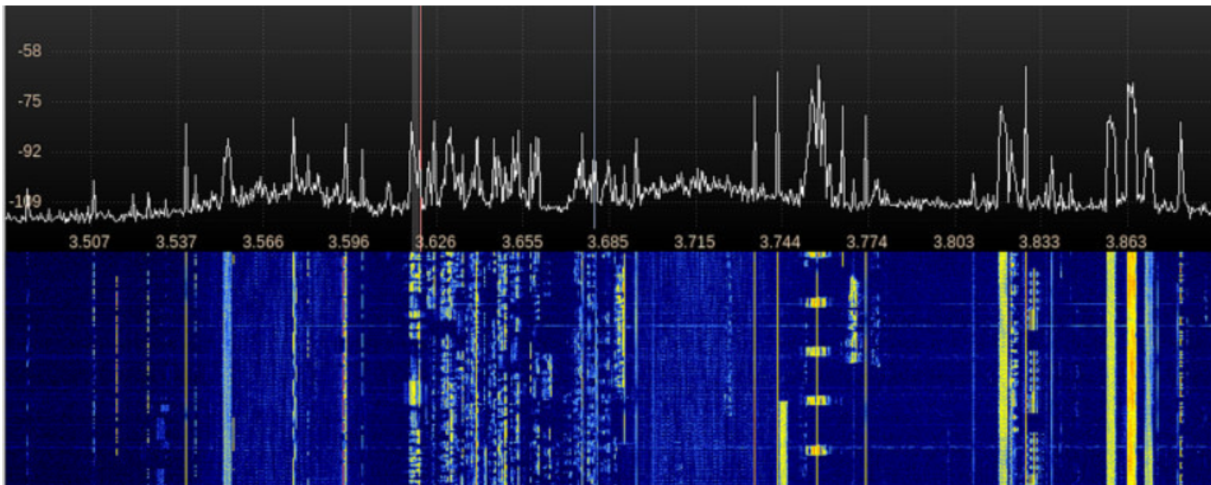
Ref: http://www.on4ww.be/emi-rfi.html

2.2 - User Experience

Building a software radio receiver means an important confrontation with an object that has a long history, a well established interface, and an important development in the last year, mainly on the move of automation.
There are many epochal innovations in the radio sector, starting with the coherer, continuing with vacuum diodes, the first amplifier, triode, and much time later, transistors and integrated circuits; but it is not only a history of components: together with components also the schematics are changed, from direct demodulation to direct amplification, to regenerative receiver, super heterodyne, to Racal continuous tuning double conversion, to reach, at modern time its SDR shape.
Each and every one of these innovations has caused a change in the form and usage of the radio: some commands appeared, other disappeared, but, from time to time, the classic radio remained built around a single object: the tuning knob, the radio “steering wheel” that is the most used command of a radio receiver. This is its strength but also its limit: most of the time spent using the tuning knob is spent looking for stations.
In other words, a radio operator acts as a blind mouse, running here and there hoping to hear another operator calling or talking with others. The charm of the tuning knob, charm acquired in the long century since the invention of the radio, on the long days and nights during which generations of radio enthusiast and operators calmly or frantically moved their hands to rotate this magical command in the hope of a contact.

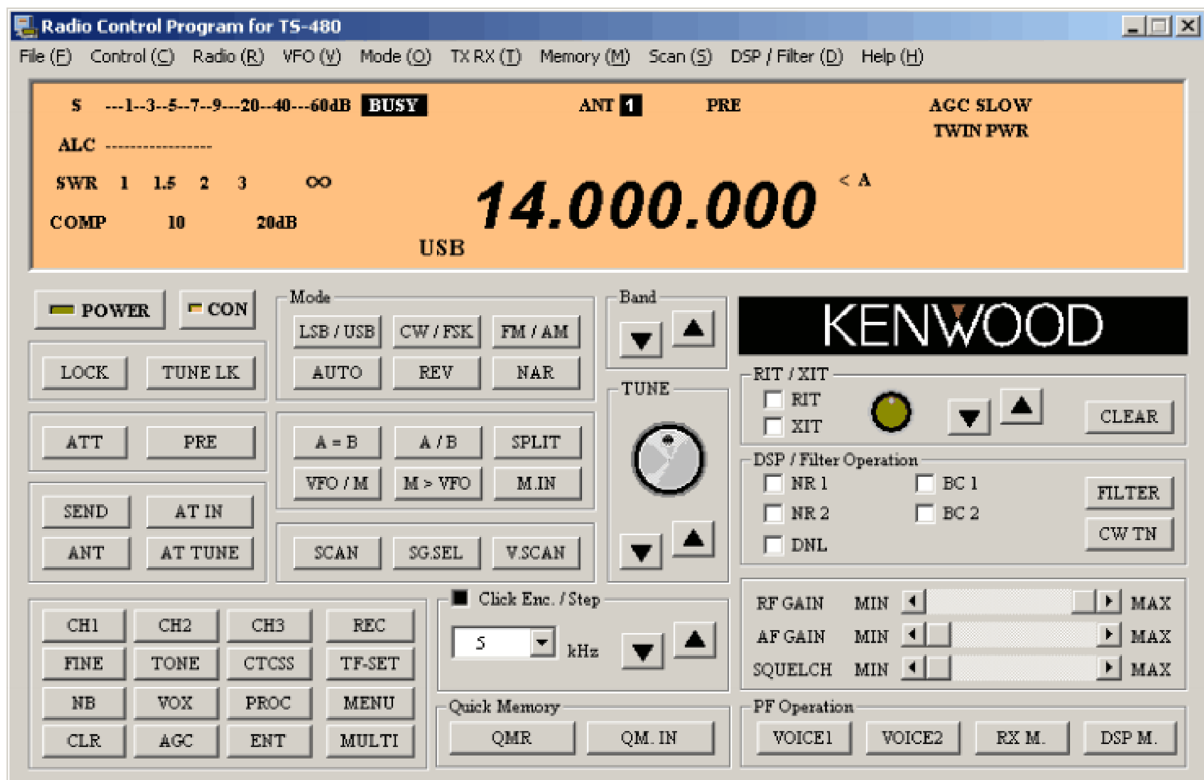
Picture 2: A spectrogram

But those old good days are ended: this is the new era of the spectrogram, where we can see the waves, looking at an irregular line oscillating on our screens (Picture 2). Most of the radio stations can be seen at a glance; and, for the visual impaired (like me) or the small signals, waterfall has been invented to show tracks of signals (Picture 3), breaking another Aristotelian unit, Time, and letting the operator see not only the current transmissions but also the past ones (and there are rumors that some algorithms are in development to also predict the future transmissions).
So it seems that the old, good, tuning knob is no more needed, but habits are strong and every user interface seems to have to implement a more or less virtual tuning knob.



Picture 3: Spectrogram (upper) and its relative waterfall (lower)

The way this control is implemented is quite traditional: a view (in some, better, implementation, a 3d model) of a classic knob that can be moved in various manners, using, of course, the mouse, like in Picture 4.
This is far from satisfactory and quite ineffective, because it tries to mimic the aspect of the knob, losing the real look, and feel, of the command: a simple wrist rotation.
In our work we explore a substitute of the wrist rotation by using a part of the mouse normally available, that many of us are already used to use: the mouse wheel.
So, no more tuning knob. And also other commands will be fired, starting with the volume that can be found on other parts of the computer interface.
All of this work is done to further explore what can be the user experience for a new radio that refuses to be a new, digital, radio dressed with old fashioned suits.



Picture 4: A traditional GUI to control a transceiver

3 - Objectives

The goal of this project is the realization of a structure that allows the reception of radio signals, their demodulation and characterization of their parameters.
The structure must consists of three parts: the first one is a “head”, capable of operating in an unmanned remote station (like a mountaintop), the second is a “central server” that could be deployed in a web farm, and the third is one, or many, web browser running a web application.
In the head it must be used a “radio sampler”, a device that samples the signals from an antenna, producing a flow of I/Q samples over an USB connection. In the same head an embedded PC must take care of the management of the sampler and the broadcasting of the samples on a network connection to the central server. Hopefully this could be made also via a radio link (eg. IEEE811b/g/n).
The central server must take care of all the signal processing and, also, must host a web server which the peripheral web browsers can access to operate their copy of centralized Software Defined Radio.
This structure must not be limited to be used "as a radio receiver" but it should allow the exploration of the desired frequencies, highlighting the frequency spectrum and, through the use of an antenna and a calibration process, should be able to perform measures on the received signal.
It should also lead to the possibility of collapsing this whole structure into a single container to realize a single device that can be operated as a radio receiver.
All the developed software must be licensed by the GPL and its structure must allow an easy integration of “ad hoc” new elaborations.

4 - Work Organization

The work has been divided into different steps:

- a state of the art analysis to know which SDR systems are on the market
- a hardware market analysis to know what is the state of the art of the related hardware;
- first software survey to select interesting projects to examine;
- some work to acquire confidence with a framework selected as reference;
- initial tests with the hardware and software to verify the hardware and software performances and to get some ideas on how to develop;
- development and test using a spiral model approach; this phase includes tests at each round, so the last round tests acts also as acceptance tests.

4.1 – State of the Art Analysis

Many different projects about Software Defined Radios are on the market. Among these we have selected some that are, in our humble opinion, the most interesting because they act as reference:

- GNU Radio: a really complete infrastructure to “assemble” software radios - “GNU Radio is a free software development toolkit that provides the signal processing runtime and processing blocks to implement software radios using readily-available, low-cost external RF hardware and commodity processors. It is widely used in hobbyist, academic and commercial environments to support wireless communications research as well as to implement real-world radio systems.” (http://gnuradio.org/redmine/projects/gnuradio/wiki)
- LinRad: an interesting and complete approach of a SDR originally built on Linux - “The Linrad dsp software processes any bandwidth that the hardware can handle. Linrad has its origin in software that was developed for 144 MHz EME CW but it is quite general and should be seen more like a kit for designing a receiver than a receiver for some particular usage.” (http://www.sm5bsz.com/linuxdsp/linrad.htm)
- WinRad: an Italian SDR built on Windows - “Winrad is a software program designed to implement a so-called Software Defined Radio (SDR), meant to run under Windows XP, Windows 2000, or Windows 98SE (only up to V1.23). In a nutshell, it accepts a chunk of up to 192 kHz coming from a half-complex mixer in form of two signals, I and Q, fed to the PC sound card, or, alternatively, an I/Q stream coming from a direct RF sampling receiver. It does a fine tuning inside that segment with a point-and-click technique, demodulates (AM, ECSS, FM, LSB, USB, CW) what has been tuned and optionally applies a series of filters to the results of the demodulation.” (http://www.sdradio.eu/weaksignals/winrad/)
- RTL-SDR: software to exploit DVB dongles to make SDR - “"RTLSDR" is a generic term for USB digital TV (DVB-T) receivers that use the Realtek **RTL2832U** chip which it was discovered in 2011 can function as general purpose software defined radio receivers. RTL2832 based hardware is by far the least expensive, costing as little as $8-12” (http://www.reddit.com/r/RTLSDR/)
- ARM Radio: “ARM Radio is a VLF-LF-MW (only the first part of the band) SDR receiver implemented entirely on the STM32F429 Discovery board, apart from two small anti alias hardware filters. It covers from 8 kHz to about 900 kHz, with AM, LSB USB and CW demodulation modes,narrow / wide bandwidth, and fast / slow AGC”. (http://www.sdradio.eu/weaksignals/armradio/index.html)

4.2 - Hardware Market Analysis

First of all we looked for the devices. Reasoning about the prices led us to consider the cheap RTL dongles, as stated before, because a lot of interesting hardware have prices in the range of 500-6000$, that is near to 25-300 times the RTL dongles. The interesting thing is that all this hardware can be used in connection with GNU Radio program and derived applications. In Pictures 5 to 8, we can see some devices available in the market.

Picture 5: Ettus Research X310 5.060 Euro and B210 1.160 Euro (March 2016)

Picture 6: Fairwaves Um TRX 2.2 950 $ and UmSITE-TM3 4000$ (March 2016)

Picture 7: Perseus 799Euro, bladeRF X115 650 $, and HackRF One Kit 330 $ (March 2016)

Picture 8: ELAD S1 369 Euro, S2 525 Euro, DUO 1.159 Euro (August 2016)

4.3 - First Software Survey

The method used to select software was a combination of multiple factors:

1. Is it possible to analyze the source code of the program?
2. Is it possible to use the lesson learned by reading the source code?
3. Is it possible to freely use part of the program?
4. What reputation has the program?
5. How much diffusion has the program?
6. How old is the last update of the program?

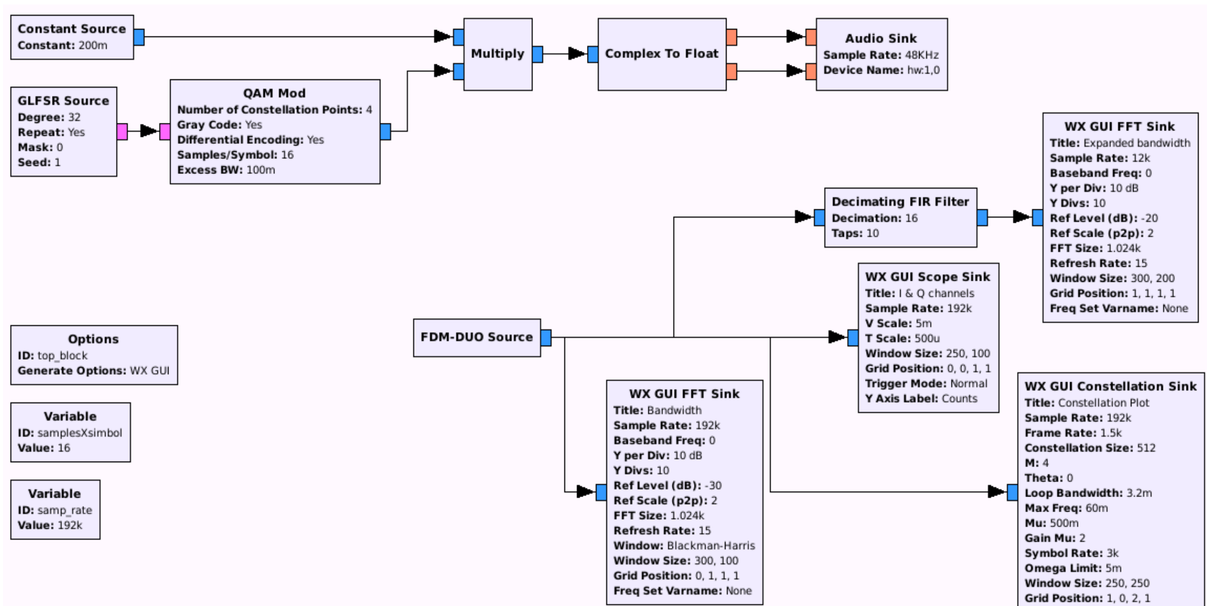
These questions are motivated by the fact that we planned to use these programs to learn how the technology has evolved, and it is really hard, when not impossible, to learn from a program without reading its source code (1), and it makes no sense to learn from a bad (4) or obsolete (6) program. Also we needed to be able to use what we have learned without legal troubles (2) and, if we found useful parts of the program, we appreciated the possibility to use them (3). Condition 5 is a prerequisite for condition 4 because an unknown program would not trigger so many evaluations and its reputation would remain uncertain.

Of course these questions lead mainly to FLOSS as well as Public Domain software, and the chosen programs reflect this:

- GNU Radio, chosen because it has a very complete infrastructure and can be used both for learning and for making some test-beds;
- RTL-SDR, chosen because at first we needed a software capable to manage RTL dongles;
- Linrad, chosen as a complete SDR program, able to talk even with RTL dongles;
- Winrad, chosen as a complete SDR program on Windows platform, to examine the differences between the two architectures;
- libusb1.0, that is not a SDR program, but is the most important library to manage USB devices;
- ARM Radio, chose for its being recent, and its inheritance from Winrad on a completely different platform, and also for its cleariness and simple inspectability.

4.4 - GNU Radio

Working with GNU Radio we have not only examined the code, but we also have analyzed its structure and the complete call flow to understand how it deploys the various blocks that are outlined in the flow-graphs. Our analysis on GNU Radio allowed us to build various flow-graphs and also some “out-of-the-tree” modules to manage both a “human-interface” (tuning and volume knobs) and the complete set of ELAD devices. Many of these realizations are depicted in Appendix 2, while in Appendix 1 there is a detailed overview of the GNU Radio package. As an example, in Picture 9 it can be seen a quite complete flowgraph realized as a demo of the usage of ELAD DUO together with GNU Radio.



Picture 9: A GNU Radio flow-graph that fully implements a 4QAM flow with FDM DUO

A complete exam of GNU Radio experience is presented in Appendix 1. The work with this framework was so detailed because:

- we have learned how to mix C with python to avoid the need to compile flow-graphs;
- we have learned the particular use of libraries and classes developed in C++;
- we have learned how the data flow is implemented;
- we have learned how to implement out-of-the-tree modules to extend GNU Radio capabilities;
- we have implemented modules that allow the usage of ELAD samplers and radio with GNU Radio and this, in turn, has been used as “reference application”.

4.5 - Initial Tests

To acquire confidence with the needed Data Signal Processing, a lot of tests have been made using GNU Radio. This wonderful tool has allowed to initially verify the inexpensive, but well supported, RTL-SDR dongles.
The module developed, WFM, NBFM, SSB radios, and a two knobs tuning unit, are documented in Appendix 2.
Unfortunately, the performances of these modules are not really good, so it was decided to use ELAD S1 module (see next paragraph).

4.6 - Development

The tests are performed during the development, in early phases of development the modules have been tested using GNU Radio modules. A spiral development model has been adopted, starting with very simple modules and testing them before adding complexity and functionality.

The various development rounds are:

- development of a GNU Radio module to connect the various ELAD samplers; this was done reorganizing an older module developed by CSI Piemonte to integrate the S1 sampler and was followed by the realization of the module for FDM DUO and then S2;
  during this phase a second generation of the modules has been created, to separate the program that takes care of FPGA stream loading – that cannot be made open source and is needed only for S1 and S2 – from the GNU Radio modules that are released under a full GPLv3 license (all these modules can be found on the ELAD website at the address http://www.eladit.com/download/sdr/Linux/index.php);
- development of the module elad-comms that reads from USB and sends USB packets; this module has been derived from the C++ function that implements I/Q sample reading in the GNU Radio module developed in the previous round; during this phase it was customized a version of Debian Jessie Linux on a Raspberry PI to allow using R/O filesystems and to launch automatically the elad-comms module.
  this module was tested using a GNU Radio flowgraph that reads from UDP datagrams;
- development of a first version of elad-server module; this module reads UDP datagrams computing an FFT, and also sends data to both a FIFO and a shared memory;
  the program was tested first by using a GNU Radio flowgraph and then using GQRX, a program developed using GNU Radio blocks that can be found at http://gqrx.dk/; the tests have highlighted that the synchronization imposed by FIFO is not good for the real time needs of the program, so this tool has been discarded in favor of shared memories;
- development of a first web application with graphical capabilities for drawing the FFT sent by the elad-server program, and its test;
- development of the basic frame of elad-client program, reading from shared memories, with the oscillator, a mixer, and the FFT generation, and its test with a copy of the first web application to verify the heterodyne working;
- development of the decimation filters, and the various FFT generations on the elad-client program;

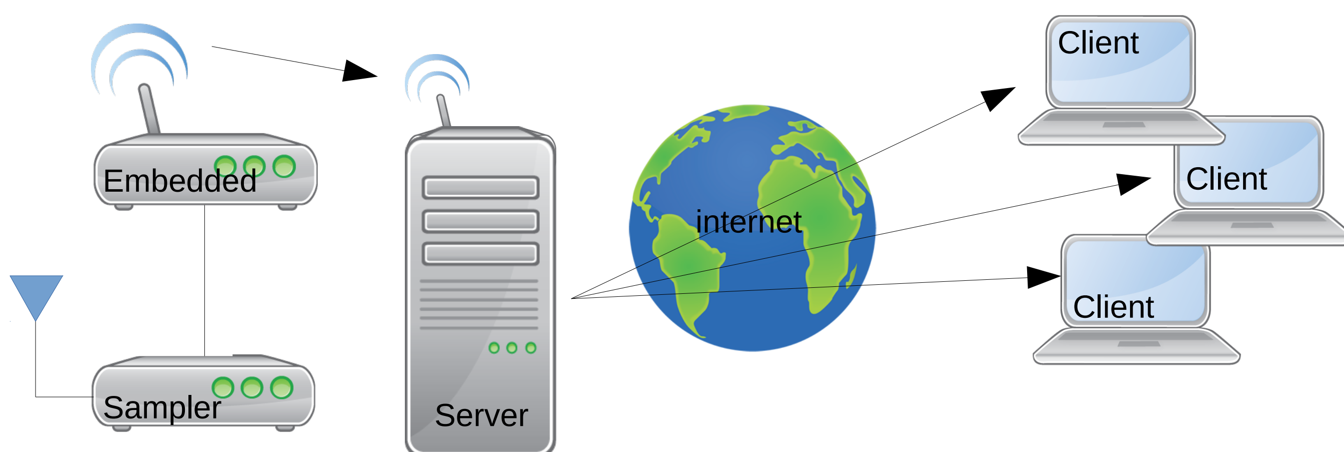
- development of a second version of the web program, performing two different FFT drawings and its test with the last version of elad-client program;
- implementation of a WebSocket server as a substitution for the shared memory communications between elad-server and web application and between elad-client and web application; test of the communications;
- implementation – in the elad-client program - of the “second/third conversion mixers” to move the center frequency of final (bf) filter to the proper values using a BFO, and its test using the web application where a third FFT drawing has been implemented;
- implementation of the audio context in web application to play the audio extracted from the incoming signal by elad-client and its test using elad-client program;
- aesthetic maquillage of the two web programs with FFT and background colors, and buttons;
- security implementation of passwords in the WebSocket functions of elad-server and elad-client program, and corresponding upgrade of the two web programs; their tests;
- upgrade of the TCP functions in elad-server program to act as a server, able to fork the process so many users can “automagically” use the program;
- perform an extensive set of measurements on the complete set of programs to obtain required metrics about network and CPU usage and processing times; as a consequence the number of buffers in elad-comms has been increased from two to four.

5 – System Structure

There are many ways a receiver, or a transceiver, could be remotized. The most known is to put all the receiving chain, from RF preamps to AF final stage, on the remote site and then make a link to control the receiver, receiving only the audio output. While this has been proven useful and its use has been widespread in the past 10 years, it is a suboptimal solution that forces the use of a complete receiver, while we could do better operations by splitting the receiving chain. With modern SDR receiving techniques we can remotize only the “sampling head”, obtaining an I+Q samples flow from the remote site to a centralized processing site that can redistribute the processed (i.e.: demodulate) signal to many connections over the Internet, allowing many operators to listen to different signals on the same band, sharing the use of the “sampling head”, something that is not possible with the previous solution.

As shown in Picture 10, the main building blocks are:

- An Antenna to convoy the RF field to the receiver. It can be as simple as a random wire or complex as a rotating wide band log periodic antenna. In our first tests we used a wire dipole.
- A Sampler (as in Direct Sampling Receiver). It has a minimum filtering, a selectable attenuator, a ADC converter and a FPGA implementing Numerical Oscillator, mixer, decimation and filtering to produce a IQ flow of the desired sample rate centered at a given frequency.
- An Embedded Computer that talks to the sampler via USB receiving commands via TCP/IP and sending IQ flow via UDP/IP.
- An IP link (wired/wireless/wireless bridge) to link the “remote head” to the server.
- A Server that collects the clients requests and processes the IQ flow to create the audio demodulated output.
- Clients that are the system's User Interface, that display both FFT and waterfall for the band and let user to select the frequency to listen to, and, of course, the software components needed to play the audio.



Picture 10: The structure of the system showing the connections between blocks

5.1 - Sampler

At the very beginning we started using the well known and cheap RTL (RTL2832U), that can be seen in Picture 11. This is a USB dongle made to allow reception of DRM stations. But some users discovered the internal structure and made a library (rtl-sdr) allowing a wider usage as general receivers. At that moment this idea sounded really good because:

- they are really cheap (less then 20$ each);
- they can be found quite easily;
- the source code of the support library is available;
- a library exists that allows their usage with the well known GNU Radio package;
- they cover a broad band starting from 70MHz and ending at 1300 MHz;
- some up converters exist that allow HF band (0-30MHz) reception.

At the same time these dongles have also some limitations, essentially linked to their being cheap devices:

- a design as quadrature receiver, with a very simple Local Oscillator;
- an ADC with only few bytes (8) and so, the need of some analogical preprocessing inside the dongle itself;
- a very poor plastic body with no screening that needs some handwork to force RF enter the device only from the antenna socket.

Picture 11: The dongle opened (left) and shielded (right)

Using these dongles we've made some parts of the work, testing the first part of the complete structure (from antenna to server), when another possible solution emerges.
By an agreement with an Italian company (ELAD) we are entitled to use most of their radios, that can be seen in Picture 12. This means that we can use their sampler or, even, their transceiver to build the remote head of this project.

This lead to many positive improvements:

- they are really Direct Digitalization radios, using little or no analogical input processing limited to an anti aliasing filter, that can be excluded for under-sampling reception, and an attenuator;
- their ADCs have 14 or 16 bit;
- the ADC is followed by a FPGA that can be reprogrammed to send various streaming speeds, from 192kS/s to 6.144MS/s;
- obviously they are not in the price class of the rtl dongles, but their price is much lower than other sampler devices that can be found on the market;

- they are specifically designed for the HF OM market, and in this field they are the state of the art;
- we have full access to the documentation, also to the reserved parts, as part of a project to build an open source support platform for these radios;
- having, at least, 14 bit sampler lets us avoid the work to trade-off bit/sample-rate needed with RTL-SDR dongles, where we must decimate a high sample-rate at about 3MS/s to exploit the decimation gain for obtaining a better dynamic range; ELAD devices have an original sample-rate of about 128 MS/s and the decimation work is done inside the FPGA.

Picture 12: Three ELAD devices: left to right, samplers S1 and S2, transceiver FDM DUO

The support for these radios is really superb, we've had every possible information and, for the DUO, also some upgrades of the firmware.
For these radios we were able to develop some packages, including:

- a firmware loader able to load FPGA stream into an arbitrary number of S1/S2 devices connected to the same computer; this firmware is not available as source code to protect the FPGA stream itself from tampering;
- a GNU Radio module capable to connect an arbitrary number of S1/S2/DUO devices to a GNU radio flow-graph, to test their functionalities and to act as a reference for this project;
- the module that links these devices with an IP connection; this module is part of this thesis work.

With this work done we are able to use any arbitrary mix of ELAD devices, even if the scope of this work is to connect a single device.

5.2 - Embedded

We have decided to use an embedded board as a bridge between the sampler and the network. These devices have to do the following things:

- Setup the sampler firmware when it is needed (firmware loading is not needed for the FDM DUO, but it is needed both for S1 and S2);
- Receive TCP/IP commands from the network, maintain state information, send these commands to the radios;
- Manage I/Q flows from the radios (using USB) and sending them to the server via UDP/IP.

Since this is not a very hard work (mainly data transfer) we have selected a Raspberry PI 2 as our implementation reference, because:

- its free implementation allows us to know all we need to make the implementation (we've customized a Debian 8 - Jessie image as operating system);
- this project is still in a very active phase: we started with a Raspberry PI to continue with model 2, and now model 3 is available, more powerful and with a compatible form factor;
- the board is little and cheap and its power requirements are reasonable.

5.3 - Connection Between Embedded and Server

This connection has two aspects: the first one is the physical one together with the implementation of the ISO/OSI levels under the network level, the second is the implementation of the ISO/OSI application level and the ancillary choices about transport and network levels.

Picture 13: The NanoBeam M5

Starting with the physical level we can state that every link is possible, wired or wireless, but, examining this assumption we must say that:

- a wired connection is usually possible when sampler and server are quite near, i.e. when the server is in the basement and the sampler is just under the roof, like when you exploit your holidays' or grandparents' country house;
- a wireless connection using AP could be used more or less the same way;
- to have a real remote location the preferred solution is a wireless bridge working on the 5GHz band using a couple of small parabolic antennas; our choice is a couple of Ubiquity NanoBeam M5 19dBi 5GHz MIMO (see Picture 13) to connect the top of a hill near our home; since the modulation is adaptive we made some test and saw that a full 60 Mbit/s can be obtained without problems simply by correctly aligning the two parabolic dishes.

5.4 - Protocols Between Embedded and Server

With the term protocol we refer to the upper levels of the communication between the server and the embedded device. The choice of the protocol must take into account many aspects of the problem:

- One single connection for commands+data or two distinct connections ?
- TCP or UDP transfer ?
- Which format for commands ?

Every decision has its pros and its cons. We started by describing the object of the communications when we have two distinct flows: a command flow, that is mainly “upwards” from server to embedded device, and then to sampler, and a data flow that is mainly “downwards” from the sampler to the embedded device and then to the server. Seeing this we can have the temptation to make a sort of unbalanced communication but it must be considered that:

- even if the samplers do not send a confirmation for the commands, it could be useful to store a “state” to the embedded device so we can query the embedded device to know the state of the attached samplers;
- at the moment we are working on a remote receiver, but we can remotize not only a sampler (as S1/S2) but also a transceiver (as FDM DUO), and this is the natural evolution of this work.

So, the answer to the first question is quite simple: we need two different flows. Then the second question arises: TCP or UDP? For the command flow TCP is the default choice, we definitely don't want an “unaffordable” connection for commands! For the data connection the question is more subtle, some could look at “lost datagrams” as a severe fault but we have to consider that this must be a “real-time” data-flow and we prefer to lose a datagram than have a best effort latency. So our choice is UDP for data flow.

The third question is related to the commands format and the answer must be very well reasoned because this choice has to deal with the compatibility with other tools:

- we make a work on itself, but it could be interesting that the “remote head” can dialog with already existing tools, as, for example, GNU Radio;
- we have to manage three distinct devices, with different capabilities: samplers (S1/S1) have very few parameters to set: Frequency, Attenuator, Input filters, where the Transceiver (FDM DUO) has a lot of commands and, also, can send a decoded audio, not only the I/Q flow.

To answer these questions, for the commands we use the CAT “standard”. This “standard” is widely used by the HAM community with the only problem that it has many dialects. We use the dialect used in ELAD FDM DUO because:

- we don’t have to do any conversion when talking to this radio ;
- we have to translate less than 4 commands when talking to S1/S2;
- if, in the future, we need to connect other radios, most of the commands are compatible and we face with little, or no, work;
- if, in the future, we need to use the remote part of this work with other software it could be expected that this is the most available protocol.

5.5 - Server

The server has two distinct roles: the first is to talk with the clients, the second is to perform the elaborations they need. The first role can be done via a normal web server plus a streaming server, while the second requires an ad-hoc DSP program.
To host both we decided to use a 64-bit Intel server running Debian 8 OS (Jessie) and Apache2 web-server.
The rationale for these choices is:

- Intel 64 bit platform is powerful enough, well widespread, cheap;
- the server is not in a remote site but in a more normal house/office, so there is no need for an embedded device or exotic hardware;
- Debian 8 is the last Debian release and it is ported on many platforms; we also use it also for the embedded device, so it is better to use the same OS for both systems.
- Apache2 is the de-facto standard for web-servers and it is well integrated in Debian 8.

5.6 - Clients

No assumptions are made for the clients, apart from the fact they have to be able to properly display HTML pages and have support for JavaScript. In the development our reference has been Firefox.

6 - Implementation

The modules developed are:

- A module that manages the samplers and the radios; it talks with USB with the device and TCP/IP+UDP/IP with the network;
- A module that allow to command the sampler in the server;
- A module that prepares the web page where users can see FFT, Waterfall, and can select frequency and modulation and sends live audio stream to the users.
- A web module that allow commanding the samplers and the radio talking to the server.
- A web module that implements the user interface allowing to “hear” the radio signals.

Apart from these modules some setup activities were done:

- Embedded SO installation and customization, with, in particular, the setting of the file system to read only so the unit does not have problems from possible power supply faults;
- Server SO and web server installation and customization, in particular adding components for an efficient audio streaming;
- Radio bridge setup to allow the remote embedded computer to talk to the server;
- Firewall setup to let users access the server only when needed.

6.1 - Samplers and Radio Management

I had just started to work on this module when I started to collaborate with ELAD. The first collaboration was on their website but then, knowing that I had some experience on Linux systems, I was asked to take care of the update of an ancient GNU Radio module for their S1 sampler. This has opened the opportunity to work with all the line of ELAD products having also physical access to them, and so the use of RTL was disbanded in favor of a much more powerful radio.

Originally the module for S1 was a single blob that included all the code needed to initialize the device and to dialog with GNU Radio. This because S1 (as S2 also) has no flash memory and its FPGA needs to be initialized every time it is powered up.

The first development follows this guideline, initializing and using the device. The modules consist of two libraries, one closed-source responsible for the FPGA initialization, the other distributed as an open source “out-of-the-tree” module for GNU Radio.

When this work had been completed and proved its functionality I started to prepare the same for the receiving part of FDM DUO. This was simpler because FDM DUO has many boards and one of them is responsible for programming the FPGA when the radio is powered, because FDM DUO was designed to be able to work both in conjunction with a PC and as a standalone radio.

While the module was proven functional, it remains the fact that the two modules for S1 and FDM DUO could not coexist in the same GNU Radio installation because they are not integrated, and the situation worsened when a third module, for the S2 sampler, was developed.

This situation was originated by the structure of the “out-of-the-tree” modules that is a “two level” structure where the upper level is a “common level”. So, we need to have an upper level for “ELAD” and some lower levels for the single devices. This could be done but it was

not the preferred choice because, during the S2 module development, some evidences emerged.
The first evidence is the fact that there are two logical distinct parts of software: the initialization and the operation. These two parts have many important differences:

- the initialization must occur only the first time the device is opened or when the characteristics of the FPGA elaboration must be changed, while the operation must run every time;
- the initialization phase could not be really transparent because the FPGA flow is an industrial secret upon which ELAD bases its market position and competitiveness, while the operation has no secrets, and relies heavily on libusb1.0;
- while in FLOSS world it is accepted that a firmware loader may be not open, we need the maximum openness of the operation module, to allow users to do changes on their own and include in their programs without a need of ELAD intervention;
- even if both samplers need an initialization and the FDM DUO doesn't, they are really similar in their working so the operation modules look very similar to each other;
- also the initialization operations are really similar both for S1 and S2;
- even if it covers rare cases, it is important that an arbitrary set of S1/S2/FDM DUO could be attached to the same PC.

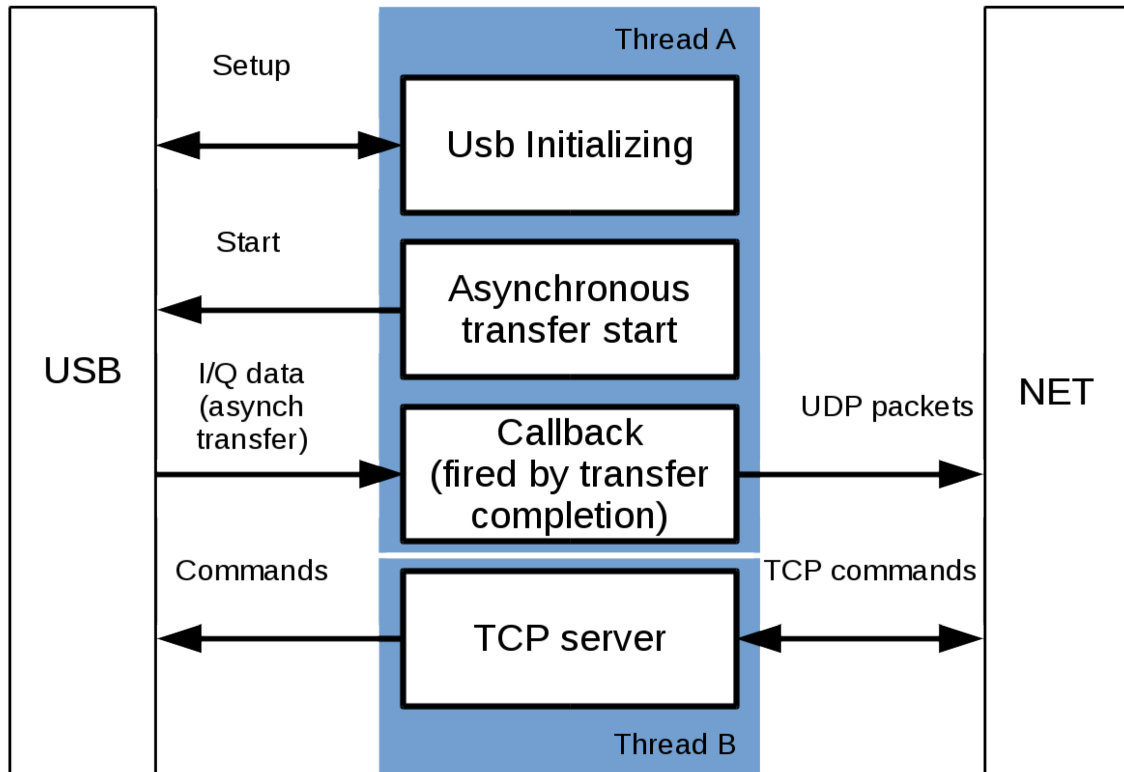
So it was decided to develop four distinct modules:

- A firmware loader. Assigning the right parameters we can decide which configuration will be loaded and in which device it will be loaded. Every run of the firmware loader initializes a single device.
- A GNU Radio module. This module belongs to the “ELAD” structure and has a single device driver that manages every device. To do so we introduced a parameter with the type of the device (S1/S2/DUO) and a parameter with the serial id of the device.
- A service module that is similar to the GNU Radio one but does not interfaces with GNU Radio, but with TCP/IP. Since GNU Radio is able to talk TCP/pi, this module could be tested also with GNU Radio.
- Alongside with these modules we have decided to develop another module that allows sending of CAT commands when needed.

As stated previously, the firmware module is “closed source”, but could be freely downloaded and used without limits. The other modules are completely free and use GPLv3 as their licensing scheme.
If the network module does not suffer from increased latency in use with GNU Radio, it could substitute completely the GNU Radio one, to avoid the tremendous effort needed when GNU Radio changes version.

6.2 - Sampler Management Program

This software manages the sampler, reading samples from the usb port, and sending them over UDP to a central site. It also implements a TCP server that accepts setup commands (central frequency, attenuator state, lowpass filter state) for the sampler. In Picture 14 there is the operation scheme.



Picture 14: The structure of the sampler management program

6.2.1 - Choices

libusb-1.0 is used to connect the Sampler. This is done mainly because libusb-1.0 works in user space and it also is thread safe.
UDP is used to send data to the central site. This is done mainly because TCP, having flow control, introduces unnecessary long delays.
To make the communications more flexible, the I/Q samples are converted into floating point values before sending them through the network.
The data received by the callbacks are converted into floating point and sent as 1024 bytes long packets, each preceded by a header consisting of a counter, to allow packet loss detection on the other endpoint, the value of central frequency, the speed of sampler, the status of low pass filter and input attenuator to better qualify the stream of data.
“Plain” TCP, without an overlying HTTP implementation, is used to receive commands, to keep the work simple but, at the same time, having the benefits of TCP.

	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	Sequence number (long)															
4	Central frequency (long)															
8						Speed		lpf att								
10	1st in phase sample (float)															
14	1st quadrature sample (float)															
//	---															
1026	128th in phase sample (float)															
1030	128th quadrature sample (float)															

Picture 15: The I/Q data UDP packet

6.2.2 - Structure

The program uses libusb to asynchronously read I/Q samples from the sampler. Using libusb two buffers are queued for reading and so, when a buffer is full, a callback is fired to send data through UDP, while the second buffer is being filled from the USB.
Another thread is spawned to wait for TCP connection and to send commands via USB, to the sampler.

6.2.2.1 - The Threads

The program starts initializing the Sampler and setting the asynchronous transfer, then spawns a thread for the TCP management, as shown in Listing 1.

```
#include <pthread.h>
...
typedef struct _threaddata_t {
	long *freq;
	int *atten;
	int *filter;
	char *sampling;
	int port;
	int *bytes_per_sample;
	int rescale;
} threaddata_t, *threaddata_p;
...
threaddata_t threaddata0;
pthread_t pthread0;
pthread_attr_t attr0;
...
// create thread that reads commands from tcp
pthread_attr_init( &attr0 );
pthread_create( &pthread0, &attr0, tcpManage, (void *)&threaddata0 );
fprintf( stderr, "Tcp manage thread created\n" );
```

Listing 1: Sampler management program spawning thread

6.2.2.2 - Asynchronous USB Reading

Even if the USB speed exceeds the needs of sample rate, it is impossible to manage read data synchronously, because during the management time the data flow fills the USB buffer of the device. This is the main reason we use asynchronous transfer of the data.

```
// prepare async transfer
transfer_in1 = libusb_alloc_transfer( 0 );
if( !transfer_in1 ) {
	fprintf( stderr, "libusb_alloc_transfer failed\n" );
	return 14;
}
transfer_in2 = libusb_alloc_transfer( 0 );
if( !transfer_in2 ) {
	fprintf( stderr, "libusb_alloc_transfer failed\n" );
	return 15;
}
...
cbdata1.obj=0;
cbdata1.transfer=transfer_in1;
cbdata1.outbuf=pBuffer1a;
cbdata1.freq = &LOfreq;
cbdata1.sampling = &sampling;
cbdata1.atten = &atten;
cbdata1.filter = &filter;
cbdata1.sockfd=sockfd;
cbdata1.salen=salen;
cbdata1.sa=sa;
cbdata1.bytes_per_sample = &bytes_per_sample;
cbdata1.rescale = rescale;
cbdata2.obj=1;
...
libusb_fill_bulk_transfer( transfer_in1, dev_handle,
	0x86, pBuffer1, 512*24, cb_in, &cbdata1, 2000 );
libusb_fill_bulk_transfer( transfer_in2, dev_handle,
	0x86, pBuffer2, 512*24, cb_in, &cbdata2, 2000 );
res = libusb_submit_transfer( transfer_in1 );
if( res ) {
	fprintf( stderr, "libusb_submit_transfer failed (%d)\n", res );
	return 16;
}
res = libusb_submit_transfer( transfer_in2 );
if( res ) {
	fprintf( stderr, "libusb_submit_transfer failed (%d)\n", res );
	return 17;
}
fprintf( stderr, "libusb_submit_transfer succedeed\n" );
fprintf( stderr, "SYN transfer starting\n" );
// main loop
for( j=0, res=0; res==0 ; j++) {
	res = libusb_handle_events_completed( ctx, NULL );
}
```

Listing 2: Prepare USB Async transfer

This transfer mode is universally considered to be “complex”, instead it is very simple, needing only a preparation phase, as shown in Listing 2, where the buffer must be prepared and declared, and the writing of the callback that will be fired once the buffer is full. Once all is ready the transfer is fired for every buffer: the transfer to the first buffer begins. When the first buffer is full, the callback is fired and the next buffer is filled.
Every time a callback is fired, it processes the data and, then, requests a new transfer.

There can be two or more buffers, this mechanism only requests all the buffer transfers to be fired at the beginning.

6.2.2.4 - UDP Sending Data

The data sending is accomplished by the callbacks, and is done using UDP. The UDP packet is made as follows:

1. Counter value, to allow data loss finding by endpoint
2. Center frequency value, in Hz
3. Sample rate ratio (power of 2: 1=>192kS/s, 2=>384kS/s, 3=>768kS/s, 4=>1536kS/s, 5=>3072kS/s, 6=>6144kS7s)
4. Low pass filter status (0=off, 1=on)
5. Attenuator status (0=off, 1=on)
6. 128 I+Q samples: every sample is 32+32 bit floating point value converted from the original USB received values of 24+24 bits (for values of ratio of 1,2,3,4,5) or 16+16 bits (where ratio value is 6).The values at point 2,3,4,5 are sent to let the receiving program know the status of the sampler dynamically.

```
void cb_in( struct libusb_transfer * transfer ) {
...
	switch(transfer->status) {
		case LIBUSB_TRANSFER_COMPLETED:
			if( *cbd→bytes_per_sample==2 ) {
				c=recalc/32.0/1024.0;
				for( j=0; j<transfer->actual_length/sizeof(short); j++ ) {
					rs=(short)((uint16_t *)(transfer->buffer))[j];
					cbd->outbuf[j]=rs*c;
				}
			} else {
				c=recalc/2.0/1024.0/1024.0/1024.0;
				for( j=0; j<transfer->actual_length/sizeof(int); j++ ) {
					ri=(int)((uint32_t *)(transfer->buffer))[j];
					cbd->outbuf[j]=ri*c;
				   }
			}
			y=j*sizeof(float);
			for( k=0; k<y; k+=1024 ) {
				counter++;
				*(long *)buffer = htonl( counter );
				*(long *)(buffer+4) = htonl( *cbd->freq );
				filler=(*cbd->atten)|(*cbd->filter)<<1 |(*cbd->sampling)<<4;
				*(short *)(buffer+8) = htons( filler );
				memcpy( buffer+10, ((char *)(cbd→outbuf))+k,
					y-k<1024?y-k:1024 );
				res=sendto( cbd->sockfd, buffer, y-k<1024?y-k+4:1034, 0,
					cbd->sa, cbd→salen );
				if( res < 0 ) {
					fprintf( stderr, "CB UDP error (%d)\n", res  );
				}
			}
			break;
		case LIBUSB_TRANSFER_CANCELLED:
...
	}
	fflush( stderr );
	transfer_in = cbd->transfer;
	res = libusb_submit_transfer( transfer_in );
...
}
```

Listing 3: Sampler management program: Callback

(red:I/Q conversion, Blue:header, Violet:resubmit transfer, Green: UDP send)

6.2.2.5 - TCP Server

This program implements also a very simple TCP server that accepts some commands useful for controlling the sampler:

- central frequency,
- insertion of attenuator,
- insertion of lowpass (anti-aliasing) filter.

The commands are sent to the sampler using the same USB connection.
In the following Listing 4 we can see the whole TCP server implementation.

```
void *tcpManage( void *payload ) {
	int pfd;
	struct sockaddr_in saddr;
	int chfd;
	struct hostent *host;
	struct sockaddr_in cliaddr;
	unsigned int clilun;
	char buf[1024];
	char *hp;
	int j, n;
	threaddata_p payl =(threaddata_p)payload;
	int port = payl->port;
	int loop;
	char *retbuf;
	pfd = socket( AF_INET, SOCK_STREAM, 0 );
	if( pfd == -1 ) {
		fprintf( stderr, "Socket open error\n" );
		exit( 1 );
	}
	int opt = 1;
	setsockopt( pfd,SOL_SOCKET,SO_REUSEADDR,(const void *)&opt, sizeof(int) );
	memset( &saddr, 0, sizeof(saddr) );
	saddr.sin_family = AF_INET;
	saddr.sin_addr.s_addr = htonl( INADDR_ANY );
	saddr.sin_port = htons( (unsigned short)port );
	if( bind( pfd, (struct sockaddr *) &saddr, sizeof( saddr ) ) == -1 )  {
		fprintf( stderr, "Binding Error\n" );
		exit( 2 );
	}
	if( listen( pfd, 1 ) == -1 ) {
		fprintf( stderr, "Listening Error\n" );
		exit( 3 );
	}
	fprintf( stderr, "Listening on port %d\n", port );
	clilun = sizeof( cliaddr );
	for( ;; ) {
		chfd = accept( pfd, (struct sockaddr *) &cliaddr, &clilun );
		if( chfd == -1 ) {
			fprintf( stderr, "Accept-ing Error\n" );
			exit( 4 );
		}
		host = gethostbyaddr( (const char *)&cliaddr.sin_addr.s_addr,
			sizeof(cliaddr.sin_addr.s_addr), AF_INET );
		if( !host ) {
			fprintf( stderr, "GetHostByName Error\n" );
			exit( 5 );
		}
		hp = inet_ntoa( cliaddr.sin_addr );
		if( !hp ) {
			fprintf( stderr, "inet_ntoa Error\n" );
			exit( 6 );
```

```
                } else {
                        fprintf( stderr, "Connected by %s\n", hp );
                }
                for( loop=0;; ) {
                        fprintf( stderr, "TCP loop\n" );
                        memset( buf, 0, sizeof( buf ) );
                        for( j=0, n=0; j<sizeof( buf )-1 ; j+=n ) {
                                n = read( chfd, buf+j, 1 );
                                if( n == -1 )  {
                                        break;
                                }
                                if( n == 0 ) {
                                        loop = 1;
                                        break;
                                }
                                if( buf[j]=='\n' ) {
                                        n = write( chfd, "\n", 1 );
                                        if( n == -1 ) {
                                                fprintf( stderr, "TCP write Error\n" );
                                                exit( 8 );
                                        }
                                        break;
                                }
                                if( buf[j]==';' ) {
                                        retbuf = manageMessage( buf, j+1, payl );
                                        n = write( chfd, retbuf, strlen( retbuf ) );
                                        if( n == -1 ) {
                                                fprintf( stderr, "TCP write Error\n" );
                                                exit( 8 );
                                        }
                                        break;
                                }
                        }
                        if( loop > 0 ) { break; }
                        if( n == -1 ) { break; }
                }
                n = close( chfd );
                if( n == -1 ) {
                        fprintf( stderr, "TCP close Error\n" );
                        exit( 9 );
                }
                fprintf( stderr, "TCP closed\n" );
        }
}
```

Listing 4: TCP server in sampler management Program

6.2.2.6 - TCP Commands Format

The commands accepted by the TCP server are:

1. AT: AT0; deselect attenuator, AT1; insert attenuator
2. LP: LP0; deselect low pass filter, LP1; insert low pass filter
3. CFxxxxxxxxxxxx; where xxxxxxxxxxxx is the frequency in Hz
4. Fxxxxxxxxxxxx; where xxxxxxxxxxxx is the frequency in Hz
5. Axy; where x controls the attenuator and y controls the low pass filter

Commands AT, LP, CF are from the “CAT” standard, while commands A and F are for compatibility with other programs.

6.2.2.7 - Evaluations and Measures

The most critical path in the program is the callback function. The task of this function is to send USB buffer data using UDP datagrams and to re-start the async transfer.
The length of the buffer is 512*24 bytes. At the given sample rate of 768kS/s the I/Q samples length is 32+32 bits, so the buffer contains 512*24/(4+4) = 1536 samples, that means 2ms of data.
If the structure is a simple “2 buffer swap” then the callback must complete its task in 2ms to avoid data leakage in the USB buffer inside the sampler.
To evaluate the elaboration time of the callback some lines of code are inserted at the beginning and at the end of the callback, as can be seen in Listing 5.

```
void cb_in( struct libusb_transfer * transfer ) {
        cbdata_p cbd;
        int res;
        int j, k, y;
        float c;
        short rs;
        int ri;
        short filler;
        struct timeval start;
        struct timeval stop;
        static char buffer[1034];
        static long counter=0;
        struct libusb_transfer * transfer_in;
        cbd = (cbdata_p)transfer->user_data;
        gettimeofday( &(start), NULL );
        switch(transfer->status) {
// . . .
        }
        fflush( stderr );
        transfer_in = cbd->transfer;
        res = libusb_submit_transfer( transfer_in );
        if( res ) {
                fprintf( stderr, "libusb_submit_transfer failed (%d)\n", res );
        }
        gettimeofday( &(stop), NULL );
        fprintf( stderr, "hf;%f;time= %f mSec\n",
             start.tv_sec*1000+0.001*start.tv_usec,
             (stop.tv_sec-start.tv_sec)*1000+0.001*(stop.tv_usec-start.tv_usec) );
}
```

Listing 5: Callback with timing profiling instructions (red)

With the program running more than a minute of statistics has been caught and analyzed using a simple spreadsheet. The data are divided into 8s segments and average, variance, max and min of elaboration time for these segments are computed, as can be seen in Picture 16.

173371817.524	0.529	2.002000004	173379821.241	0.559	1.997999996
173371819.526	0.581	1.997999996	173379823.239	0.594	1.999000013
173371821.524	0.531		173379825.238	0.555	
	0.554375344	2.000048262		0.697847962	2.000928232
	0.0005448	2.01833E-05		0.010014207	0.000816592
	0.704	2.05400002		2.249	4.186000019
	0.513	1.972000003		0.392	1.342999995
		27.71809833			34.87621149
Absolute time	Duration	Interval	Absolute time	Duration	Interval

Picture 16: USB callback timing (green:avg & dev, red: too high max value)

This statistic states two interesting things: the first is that the average processing time is less than 1ms, the other is that sometimes the max value is well over the 2ms limit.
The first observation leads to the fact that the Raspberry PI2 is powerful enough to manage the data flow, the second says that sometimes a data loss may occur.
A count of the values over the 2ms barrier leads to a loss event rate of 1 every approx 10s. This could be considered a minor inconvenient, but we decided to avoid this by increasing the number of buffers involved.
Since the highest value we detected is slightly over 4ms, it extends over 3 buffers, which means that the requested number of buffers is 4.
The change is quite straightforward, as Listing 6 shows, because it doesn't involve the callback itself but only the number of buffers, their initialization, and the initial requests for the asynchronous transfer.
After the change another test has been run. The test length was more than 20 minutes and the processing times are higher than 2ms only in 4 cases, of which only 2 are higher than 3ms and none more than 3.6 ms. This also confirms that the latency is untouched and is equal to 2ms.

```
…
unsigned char pBuffer1[512*24];
unsigned char pBuffer2[512*24];
unsigned char pBuffer3[512*24];
unsigned char pBuffer4[512*24];
struct libusb_transfer *transfer_in1;
struct libusb_transfer *transfer_in2;
struct libusb_transfer *transfer_in3;
struct libusb_transfer *transfer_in4;
float pBuffer1a[512*12];
float pBuffer2a[512*12];
float pBuffer3a[512*12];
float pBuffer4a[512*12];
cbdata_t cbdata1;
cbdata_t cbdata2;
cbdata_t cbdata3;
cbdata_t cbdata4;
…
cbdata1.obj=0;
cbdata1.transfer=transfer_in1;
cbdata1.outbuf=pBuffer1a;
cbdata1.freq = &LOfreq;
cbdata1.sampling = &sampling;
cbdata1.atten = &atten;
cbdata1.filter = &filter;
cbdata1.sockfd=sockfd;
cbdata1.salen=salen;
cbdata1.sa=sa;
cbdata1.bytes_per_sample = &bytes_per_sample;
cbdata1.rescale = rescale;
cbdata2.obj=1;
cbdata2.transfer=transfer_in2;
cbdata2.outbuf=pBuffer2a;
cbdata2.freq = &LOfreq;
cbdata2.sampling = &sampling;
cbdata2.atten = &atten;
cbdata2.filter = &filter;
cbdata2.sockfd=sockfd;
cbdata2.salen=salen;
cbdata2.sa=sa;
cbdata2.bytes_per_sample = &bytes_per_sample;
```

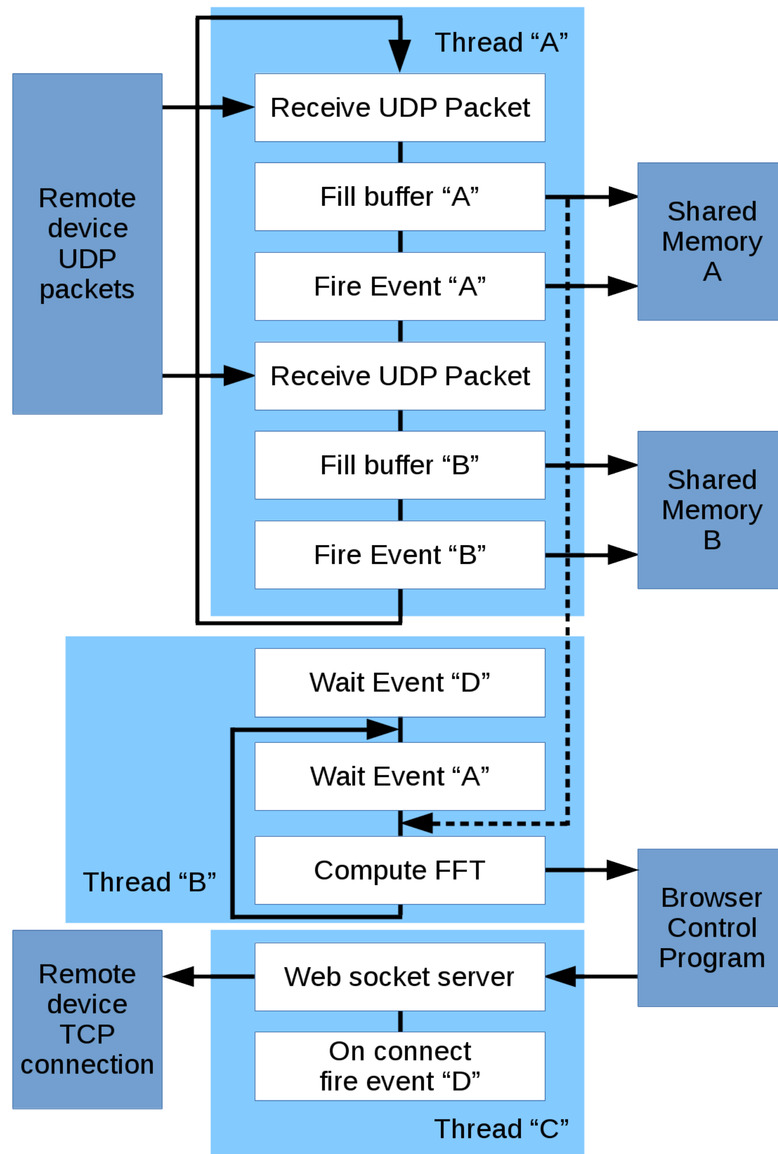
```
cbdata2.rescale = rescale;
cbdata3.obj=2;
cbdata3.transfer=transfer_in3;
cbdata3.outbuf=pBuffer3a;
cbdata3.freq = &LOfreq;
cbdata3.sampling = &sampling;
cbdata3.atten = &atten;
cbdata3.filter = &filter;
cbdata3.sockfd=sockfd;
cbdata3.salen=salen;
cbdata3.sa=sa;
cbdata3.bytes_per_sample = &bytes_per_sample;
cbdata3.rescale = rescale;
cbdata4.obj=3;
cbdata4.transfer=transfer_in4;
cbdata4.outbuf=pBuffer4a;
cbdata4.freq = &LOfreq;
cbdata4.sampling = &sampling;
cbdata4.atten = &atten;
cbdata4.filter = &filter;
cbdata4.sockfd=sockfd;
cbdata4.salen=salen;
cbdata4.sa=sa;
cbdata4.bytes_per_sample = &bytes_per_sample;
cbdata4.rescale = rescale;
libusb_fill_bulk_transfer( transfer_in1, dev_handle, 0x86, pBuffer1,
        512*24, cb_in, &cbdata1, 2000 );
libusb_fill_bulk_transfer( transfer_in2, dev_handle, 0x86, pBuffer2,
        512*24, cb_in, &cbdata2, 2000 );
libusb_fill_bulk_transfer( transfer_in3, dev_handle, 0x86, pBuffer3, 512*24,
        cb_in, &cbdata3, 2000 );
libusb_fill_bulk_transfer( transfer_in4, dev_handle, 0x86, pBuffer4, 512*24,
        cb_in, &cbdata4, 2000 );
…
res = libusb_submit_transfer( transfer_in1 );
if( res ) {
        fprintf( stderr, "libusb_submit_transfer failed (%d)\n", res );
        return 18;
}
res = libusb_submit_transfer( transfer_in2 );
if( res ) {
        fprintf( stderr, "libusb_submit_transfer failed (%d)\n", res );
        return 19;
}
res = libusb_submit_transfer( transfer_in3 );
if( res ) {
        fprintf( stderr, "libusb_submit_transfer failed (%d)\n", res );
        return 20;
}
res = libusb_submit_transfer( transfer_in4 );
if( res ) {
        fprintf( stderr, "libusb_submit_transfer failed (%d)\n", res );
        return 21;
}
...
```

Listing 6: The changes needed to avoid sporadic data losses

(red:added data and code)

6.3 - Central Control Program

This program has the main purpose to feed the data arriving from the UDP flow to two shared memories, so the "client" programs can read and process the data. This program implements a WebSocket that sends FFT data to, and receives commands from, a browser. This program also sends,using another TCP connection, to the "sampler management program" the commands received, as shown in Picture 17.



Picture 17: The structure of the central control program

6.3.1 - Choices

The program uses two shared memories from which the client programs can read data. The synchronization is done using condition variables also in shared memory.
The communication with the controlling browser is made using a WebSocket. The implementation of the WebSocket is not exhaustive, but it is kept as simple as possible, to acquire the benefits of WebSockets while at the same time keeping the implementation burden as low as possible.

6.3.2 - Structure

As shown in Listing 7, the program starts launching various threads responsible for the various activities, such as the reception of the UDP datagrams:

- a thread is used to read UDP data and to store it into two different shared memories, that implement a simple “circular buffer”;
- a thread is used to implement a WebSocket that receives commands from a browser;
- a thread is used to compute FFT and to send this data to a browser using the WebSocket;
- a thread is used to send commands received by WebSocket via TCP to the “sampler management program”.

```
shm_id = shmget ( payl->shm_key, (BUFFERSIZE+sizeof( pthread_cond_t )+6)*2,
      IPC_CREAT | S_IRUSR | S_IWUSR | 0666 );
if( shm_id==-1 ) {
      fprintf( stderr, "erro allocating shmem %d %s\n", errno,
            strerror(errno) );fflush( stderr );
}
shm_cond1 = (char*) shmat (shm_id, 0, 0);
shmaddr1 = shm_cond1 + sizeof( pthread_cond_t );
shm_cond2 = shmaddr1 + BUFFERSIZE + 6;
shmaddr2 = shm_cond2 + sizeof( pthread_cond_t );
fprintf( stderr, "Shared %x memory attached at address %p\n", shm_id, shm_cond1 );
pthread_condattr_init( &cond_attr1 );
pthread_condattr_init( &cond_attr2 );
pthread_condattr_setpshared( &cond_attr1, PTHREAD_PROCESS_SHARED );
pthread_condattr_setpshared( &cond_attr2, PTHREAD_PROCESS_SHARED );
pthread_cond_init( (pthread_cond_t *)shm_cond1, &cond_attr1 );
pthread_cond_init( (pthread_cond_t *)shm_cond2, &cond_attr2 );
pthread_condattr_destroy( &cond_attr1 );
pthread_condattr_destroy( &cond_attr2 );
```

Listing 7: Central control program: shared memory & condition variables creation

6.3.2.1 - UDP Reading and Shared Memories Filling

At the very beginning the activity starts creating four shared memory areas: two are the implementation of “alternating buffers” and the other two are reserved for storing the related conditional variables.

The core activity is quite simple: the thread starts reading UDP packets. A simple sanity control is done on the sequence counter at the beginning of every packet, signaling packet loss. Then the data is copied into the “current” shared memory until this became full. When this occurs the relative conditional variable is fired and the other shared memory became “current”.

One of the two alternating buffer is tied to the FFT: when it is full an appropriate conditional variable is fired to start the FFT computation. Since FFT is needed only a few times a second, no special effort is done to optimizing the operations.

An overview of these operations can be seen in Listing 8.

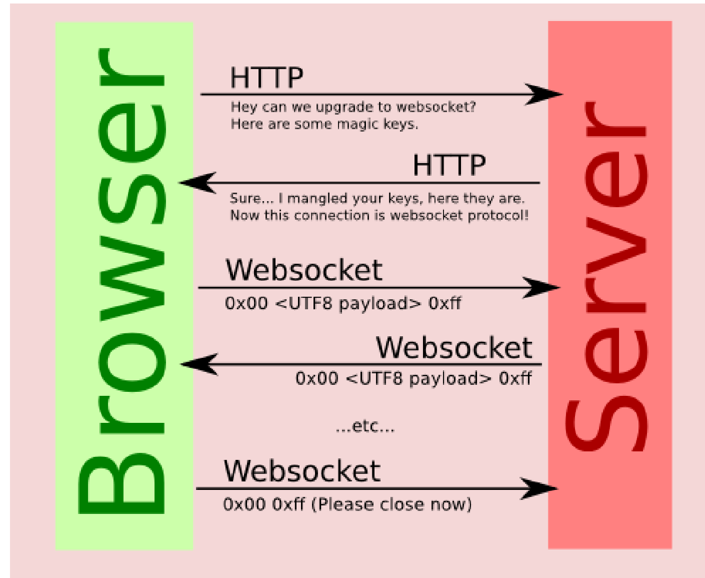
```
for( ;; ) {
	a=1-a;
	if( a==1 ) {
		con = payl->con; buff=payl->buff=payl->buf_a;
		shm_cond=shm_cond1; shmaddr=shmaddr1;
	} else {
		con=NULL;
		buff=NULL;
		shm_cond=shm_cond2;
		shmaddr=shmaddr2;
	}
	if( buff ) {
		memset( buff, 0, BUFFERSIZE );
	}
	for( j=0, *lung=0; j<BUFFERSIZE; j+=res-10 ) {
		res = recvfrom( sfd, recvBuffer, 1034, 0, (struct sockaddr *)
			&cliaddr, &clilen );
		if( buff ) {
			memcpy( buff+j, recvBuffer+10, res-10 );
		}
		memcpy( (char *)payl->freq, recvBuffer+4, 4 );
		memcpy( (char *)payl->att_lowpass_rate, recvBuffer+8, 2 );
		memcpy( shmaddr, recvBuffer+4, 6 );
		memcpy( shmaddr+6+j, recvBuffer+10, res-10 );
		tempCounter = ntohl ( *(long *)recvBuffer );
		counter++;
		if( counter!=tempCounter ) {
			fprintf( stderr,
				"Counter (%ld) differs from previous (%ld)\n",
				counter, tempCounter);
			counter=tempCounter;
		}
		*lung += res-10;
		if ( res == -1 ) {
			fprintf( stderr,"Recvfrom Error\n" );
			exit( 10 );
		}
	}
	pthread_cond_broadcast( (pthread_cond_t *)shm_cond );
	if( con ) {
		pthread_cond_signal( con );
	}
}
```

Listing 8: Central control program: UDP reading and shared memory filling

(red: packet sequence control)

6.3.2.2 – WebSocket

WebSocket is a quite new applicative protocol, built over TCP, and similar to HTTP, from which it inherits the header structure. Picture 18 shows a schematics of how it works.



Picture 18: WebSocket life cycle

https://warmcat.com/libwebsockets/2010/11/01/libwebsockets-html5-websocket-server-library-in-c.html

Here there is a short illustration of our implementation: the relative thread start listening to a TCP socket. A simple state machine is used to verify that the needed headers are properly received and to send a proper response. In this process one important thing is to manage in the right way a “challenge-response” authentication done via the “Sec-Websocket-Key”/”Sec-Websocket-Accept” headers. Picture 19 shows how this challenge-response authentication must be done, alongside with values useful for testing the implementation, Listing 9 depicts the operations done in the program to initialize the WebSocket.
When all the headers are collected and sent, the applicative connection is established and the data flow can occur. In this moment the program clears a condition variable that enables the FFT operations and the beginning of decoding the commands arriving from the browser.
The decoding of the commands also requires some operations: the related theory can also be found in the appendix relative to the WebSockets. We chose to manage only short packets and not decode their content but only to pass them to the TCP thread. Once a command is received the thread fires a condition variable that triggers the TCP command sending thread and then continues to listen for other commands.
The reception of the commands is done in a way that lets a new “WebSocket connection request” to stop the current socket and launch a new one, so in case of a browser hiccup it is straightforward to restore the connection.

```
Client handshake Request
GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13

Server handshake response
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

s3pPLMBiTxaQ9kYGzzhZRbK+xOo= is obtainbed computing the value of sha1 of the string obtained
contatenating dGhlIHNhbXBsZSBub25jZQ== with 258EAFA5-E914-47DA-95CA-C5AB0DC85B11
sha1( “dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11” )
```

Picture 19: WebSocket initial exchange example

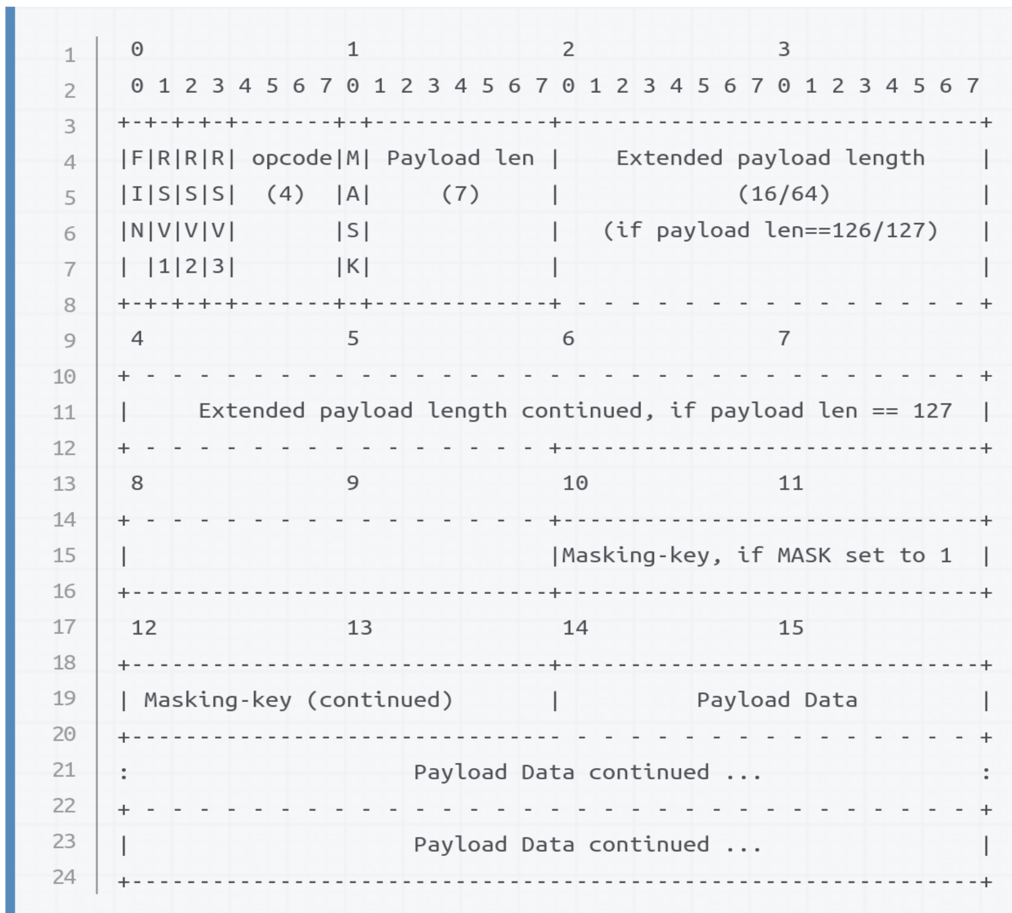
https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

```
pfd = socket( AF_INET, SOCK_STREAM, 0 );
if( pfd == -1 ) { fprintf( stderr, "Socket open error\n" ); exit( 7 ); }
int opt = 1;
setsockopt( pfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&opt , sizeof(int) );
memset( &saddr, 0, sizeof(saddr) );
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr = htonl( INADDR_ANY );
saddr.sin_port = htons( (unsigned short)payl->port );
if( bind( pfd, (struct sockaddr *) &saddr, sizeof( saddr ) ) == -1 )  {
        fprintf( stderr, "Binding Error\n" ); exit( 8 );
}
if( listen( pfd, 1 )==-1 ) { fprintf( stderr, "Listening Error\n" ); exit( 10 ); }
clilun = sizeof( cliaddr );
for( status=0,pp=NULL;;pp=NULL ) {
        memset( mybuf, 0, sizeof( mybuf ) );
        if( pp==NULL ) { pp = fgets( mybuf, sizeof(mybuf), bchfd ); }
        if( !pp && errno )  { fprintf( stderr, "TCP read Error\n" ); exit( 15 ); }
        if( !pp ) { continue; }
        if( mybuf[strlen(mybuf)-1]=='\n' ) {
                for( j=0; j<XtNumber(requestHeader); j++ ) {
                        if(!strncmp(mybuf,requestHeader[j],strlen(requestHeader[j]))){
                                status++;
                                if(!strncmp(mybuf,keyHeader,strlen(keyHeader))) {
                                        strcpy( keys, mybuf+strlen(keyHeader) );
                                        if( keys[strlen(keys)-1]=='\r' ||
                                                keys[strlen(keys)-1]=='\n'  ){
                                                keys[strlen(keys)-1]=0;
                                        }
                                        if( keys[strlen(keys)-1]=='\r' ||
                                                keys[strlen(keys)-1]=='\n'  ){
                                                keys[strlen(keys)-1]=0;
                                        }
                                        fprintf( stderr, "Caught key %s\n", keys );
                                }
                        }
                }
                if( strlen(mybuf)<=2 ) {
```

```
                    status++;
                    fprintf( stderr, "status advanced %d\n", status );
              }
        }
        if( status >= XtNumber(requestHeader)+1 ) {
              break;
        }
}
n=fprintf( bchfd, "HTTP/1.1 101 Switching Protocols\r\n" );
n=fprintf( bchfd, "Upgrade: websocket\r\n" );
n=fprintf( bchfd, "Connection: Upgrade\r\n" );
sprintf( mybuf, "%s%s", keys, "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" );
memset( keys, 0, sizeof( keys ) );
SHA1( (unsigned const char *)mybuf, strlen(mybuf), (unsigned char *)keys );
pp=g_base64_encode( (const guchar *)keys, strlen(keys) );
memset( keys, 0, sizeof( keys ) );
strcpy( keys, (const char *)pp );
g_free( pp );
n=fprintf( bchfd, "Sec-WebSocket-Accept: %s\r\n", keys );
n=fprintf( bchfd, "Sec-WebSocket-Protocol: chat\r\n\r\n" );
fflush( bchfd );
```

Listing 9: Central control program: WebSocket initialization

(red: request header management, blue: secret key computation)



Picture 20: WebSocket frame structure

https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

To communicate the WebSocket uses “frames” that have the structure outlined in Picture 20. They sends data encoding them into frames that must be decoded.
As Listing 10 shows, the WebSocket thread receives the frames, extracts commands, saves them into a buffer, and then signals to the TCP thread to send them to the remote application.

```
for( isFirst=1;; ) {
        memset( mybuf, 0, sizeof( mybuf ) );
        n = read( chfd, mybuf, sizeof( mybuf ) );
        if( n<1  )  { fprintf( stderr, "TCP read Error\n" ); break; }
        if( !n ) { fprintf( stderr, "TCP read 0 bytes \n" ); continue; }
        fprintf( stderr, "Received %d bytes : ", n); fprintf( stderr, "\n" );
        // We only manage a final, text, masked, short message
        if( (unsigned char)mybuf[0]==0x81 ) {
                j=mybuf[1]&0x7F;
                if( j==n-6 ) {
                        for( n=0; n<j; ) {
                                for( k=0; k<4 && n<j; k++, n++ ) {
                                        mybuf[n+6]^=mybuf[k+2];
                                }
                        }
                        if( isFirst==1 ) {
                                isFirst++;
                                if( strcmp( mybuf+6, payl->passphrase ) ) {
                                        fprintf( stderr,
                                                "required >%s< differ from >%s<\n",
                                                payl->passphrase, mybuf+6 );
                                        *payl->outfile=0;
                                        pp=mybuf;
                                        status=0;
                                        break;
                                }
                        } else {
                                strcpy( payl->buf, mybuf+6 );
                                pthread_cond_signal( payl->con_a );
                                fprintf( stderr, "%s\n", mybuf+6 );
                        }
                } else {
                        fprintf( stderr,
                                "wrong message length: is %d instead of %d\n",
                                j, n-6 );
                        *payl->outfile=0;
                        pp=mybuf;
                        status=0;
                        break;
                }
        } else {
                fprintf( stderr, "wrong message type: %x\n", mybuf[0] );
                *payl->outfile=0;
                pp=mybuf;
                status=0;
                break;
        }
}
```

Listing 10: Central control program: WebSocket frame reception

(red: password authentication, blue: data received signal)

6.3.2.3 - TCP to the Remote Sampler Management Program

As can be seen in Listing 11, this thread is a classic TCP client program that starts trying to connect the remote endpoint and then waits for a condition variable that signals the arrival of commands to send these commands to the endpoints. If a transmission error occurs, the connection is stopped and a new one is attempted. This has been proven successful also in the case of a restart of the remote computer.

```
for( ;; ) {
        sfd = socket( AF_INET, SOCK_STREAM, 0 );
        if( sfd == -1 ) { fprintf( stderr, "Socket error\n" ); exit( 1 ); }
        fprintf( stderr, "Socket for TCP created\n" );
                server = gethostbyname( hostname );
        if (server == NULL) { fprintf( stderr, "
                ERROR, no such host as %s\n", hostname ); exit( 2 ); }
        memset( &saddr, 0, sizeof( saddr ) );
        saddr.sin_family = AF_INET;
        saddr.sin_addr.s_addr = htonl( INADDR_ANY );
        memmove( (char *)&saddr.sin_addr.s_addr, (char *)server->h_addr,
                server->h_length);
        saddr.sin_port = htons( (unsigned short)port );
        if( connect( sfd, (struct sockaddr *)&saddr, sizeof( saddr ) ) == -1 ) {
                fprintf( stderr, "Connect Error\n" );
                sleep( 5 );
                continue;
        }
        connected = 1;
        for( ;; ) {
                pthread_cond_wait( payl->con, payl→mut );
                fprintf( stderr, "message received: <%s>\n", payl->buf );
                n = write( sfd, payl->buf, strlen( payl->buf ) );
                if( n == -1 ) {
                        fprintf( stderr, "TCP write Error\n" );
                        exit( 5 );
                }
                for( closure = 0;; ) {
                        memset( buf, 0, sizeof( buf ) );
                        for( j=0, n=0; ; j++ ) {
                                n = read( sfd, buf+j,  1 );
                                if( n == 0 )  { connected = 0; break; }
                                if( n == -1 )  { break; }
                                if( buf[j]=='\n' ) { closure = 1; break; }
                                if( buf[j]==';' ) { break; }
                        }
                        if( connected == 0 ) { break; }
                        if( closure == 1 ) { break; }
                }
                if( connected == 0 ) {  break; }
        }
}
close( sfd );
```

Listing 11: Central control program: TCP command sending

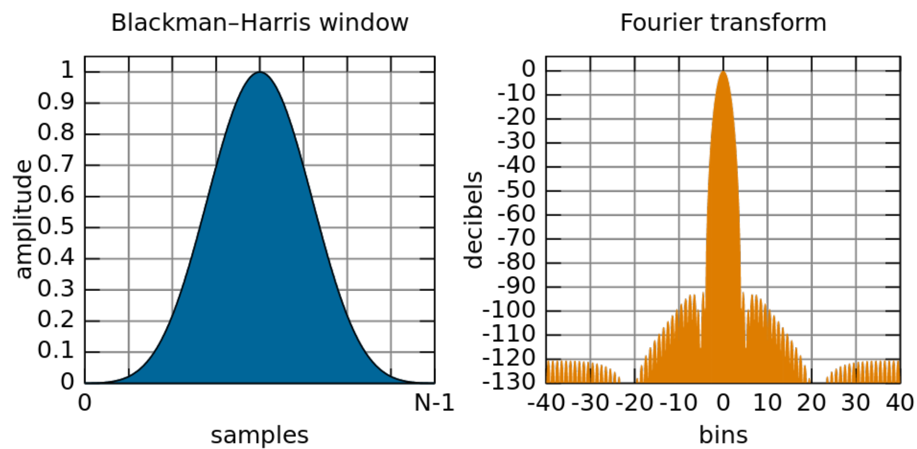
6.3.2.4 - FFT Computing

The thread that computes FFT, shown in Listing 13, prepares all the trigonometric coefficients (see Listing 12) and waits for the WebSocket thread to signal that the connection is established, then it waits for the condition variable that signals that the UDP reception has some data ready for processing. When the FFT has been processed, the data is transmitted using the WebSocket and the thread waits for some time, to void supercharging the browser.

```
fftr = calloc( N, sizeof( double ) );
if( fftr == NULL ) {
	fprintf( stderr, "error allocating fftr buffer" );
	exit( 11 );
}
ffti = calloc( N, sizeof( double ) );
if( ffti == NULL ) {
	fprintf( stderr, "error allocating ffti buffer" );
	exit( 12 );
}
fftout = calloc( N, sizeof( char ) );
if( fftout == NULL ) {
	fprintf( stderr, "error allocating fftout buffer" );
	exit( 13 );
}
co = calloc( N/2, sizeof( double ) );
if( co == NULL ) {
	fprintf( stderr, "error allocating co buffer" );
	exit( 14 );
}
si = calloc( N/2, sizeof( double ) );
if( si == NULL ) {
	fprintf( stderr, "error allocating si buffer" );
	exit( 15 );
}
coeff = calloc( N, sizeof( double ) );
if( coeff == NULL ) {
	fprintf( stderr, "error allocating coeff buffer" );
	exit( 16 );
}
// coeff table initialization
for( j=1; j<N; j++ ) {
	coeff[j]=0.35875-0.48829*cos(2*pi*j/(N-1))+
		0.14128*cos(4*pi*j/(N-1))-0.01168*cos(6*pi*j/(N-1));
}

// sin table initialization
m = N/2;
p = 10;
co1 = cos( pi/m );
si1 = -sin( pi/m );
co[0] = 1;
si[0] = 0;
for( j=1; j<m; j++ ) {
	co[j] = co1 * co[j-1] - si1 * si[j-1];
	si[j] = si1 * co[j-1] + co1 * si[j-1];
}
```

Listing 12: Central control program: FFT coefficient precomputation

(red: Blackman-Harris window coefficient)



Picture 21: Blackman-Harris window

Before computing the FFT it is mandatory to shape the samples to avoid spectral losses; this is done using the known Blackman-Harris window, shown in Picture 21, to shape the coefficients according to the formula in Picture 22:

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right)$$

$$a_0 = 0.35875; \quad a_1 = 0.48829; \quad a_2 = 0.14128; \quad a_3 = 0.01168$$

Picture 22: Blackman-Harris windowing function

https://en.wikipedia.org/wiki/Window_function

```
pthread_cond_wait(payl->con, payl->mut);
// Read and mask input data
for( j=0; j<N; j++ ) {
	fftr[j]=((float *)payl->buf)[2*j]*coeff[j];
	ffti[j]=((float *)payl->buf)[2*j+1]*coeff[j];
}
m = N/2;
p = 10;
// FFT Compute
for( j=0,ng=1,md=m; j<p; j++ ) {
	for( ind=0, ig=0; ig<ng; ig++ ) {
		for( i=0, k=0; i<md; i++ ) {
			i1=ind+i;
			i2=i1+md;
			fftdr = fftr[i1] - fftr[i2];
			fftdi = ffti[i1] - ffti[i2];
			fftr[i1] = fftr[i1] + fftr[i2];
			ffti[i1] = ffti[i1] + ffti[i2];
			fftr[i2] = fftdr * co[k] - fftdi * si[k];
			ffti[i2] = fftdr * si[k] + fftdi * co[k];
			k += ng;
		}
		ind += md*2;
	}
	md >>= 1;
	ng <<= 1;
}
// Coefficients reordering
for( j=0; j<N; j++ ) {
	jd=j;
	k=0;
	kp=m;
	for( i=1; i<=p; i++ ) {
		k = k + (jd-(jd/2)*2)*kp;
		jd >>= 1;
		kp >>= 1;
	}
	if( k>j ) {
		fftdr = fftr[j];
		fftdi = ffti[j];
		fftr[j] = fftr[k];
		ffti[j] = ffti[k];
		fftr[k] = fftdr;
		ffti[k] = fftdi;
	}
}
// Absolute value
for( j=0; j<N; j++ ) {
	fftr[j] = fftr[j]/m;
	ffti[j] = ffti[j]/m;
	fftr[j] = sqrt( fftr[j] * fftr[j] + ffti[j] * ffti[j] );
	val=20*log10(fftr[j]);
	if( val < -200 ) {
		val=-200;
	}
	fftout[j]=200+(char)val;
}
```

Listing 13: Central control program: FFT computing

(red: Blackman-Harris window, blue: FFT, green: coefficient reordering)

The thread that computes FFT data sends this data using WebSocket. Since the antialiasing has an effect on the borders of the coefficients, not all of them are sent, and the overall bandwidth is limited to to 896/1024 of the sample rate (using 768kS/s the bandwidth is 672kHz).
Also the values are limited to the range from -200dBm to 0dBm, in fact -200dBm is a value never reached and values over 0dBm are so big that is is not important how much over 0dBm they are.
When the FFT thread sends the FFT values it compiles a WebSocket frame as can be seen in Listing 14, placing some informations before the FFT values.
The first of these informations are related to the WebSocket protocol, and are frame type, mask, and length, followed by central frequency, sample rate coefficient, low pass filter status and attenuator status, as depicted in Picture 23.

```
buf[0]=0x82;
buf[1]=0x7e;
buf[2]=0x03;
buf[3]=0x86;
memcpy( buf+4, payl->freq, 4 );
memcpy( buf+8, payl->att_lowpass_rate, 2 );
memcpy( buf+10, fftout+512+64, 512-64 );
memcpy( buf+512+10-64, fftout, 512-64 );
if( *payl->outfile ) {
        j=write( *payl->outfile, buf, 1024+10-128 );
        if( j<0 ) {
                fprintf( stderr, "sent error %d %s\n", errno, strerror(errno) );
                fflush( stderr );
        }
}
usleep( 100000 );
```

Listing 14: Central control program: the composition of the WebSocket FFT frame

	0	1	2	3	4	5	6	7
0	Frame type: 0x82							
1	No mask, frame length>125 : 0x7e							
2	Hi nibble fram length: 0x03							
3	Lo nibble fram length: 0x86							
4	Central frequency (long)							
5								
6								
7								
8	Speed						lpf	att
9								
10	1st FFT sample							
//	---							
905	896th FFT sample (float)							

Picture 23: The WebSocket FFT frame

6.3.3 - Evaluations and Measures

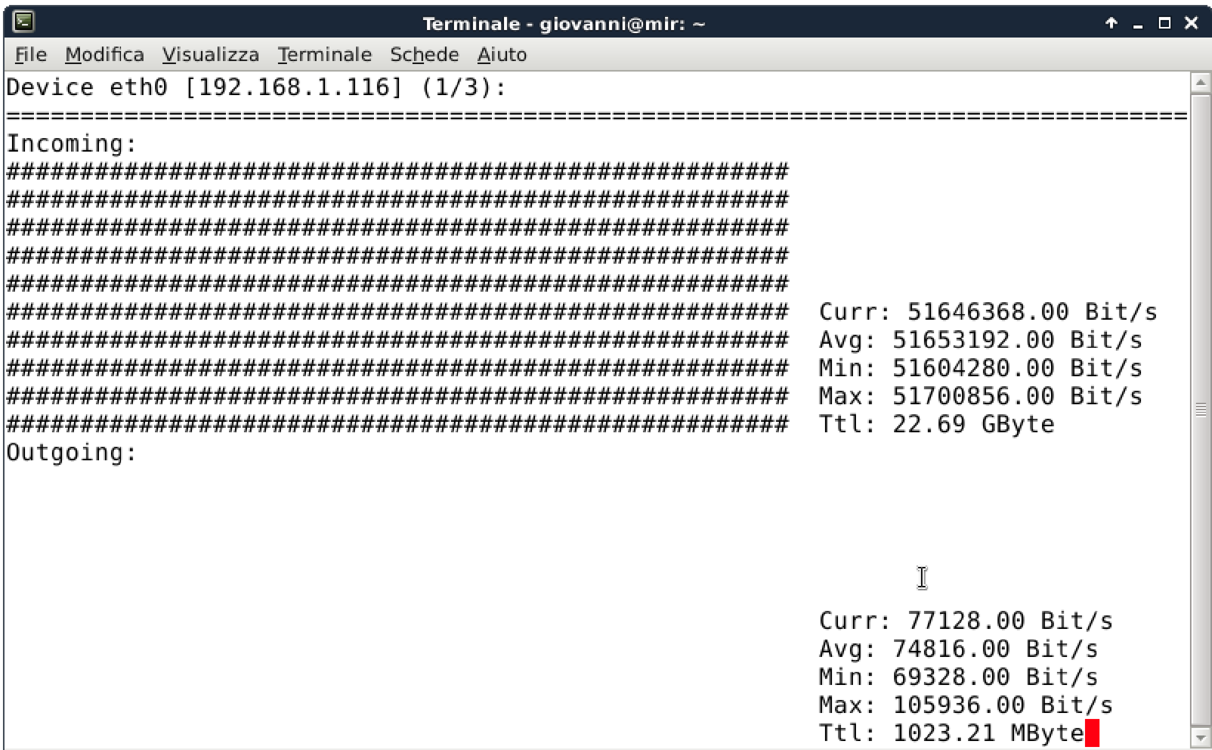
The most critical path in the program is the management of the incoming UDP packets. To track the possible packet loss, the software includes a packet tracking feature that compares a generated serial number with the incoming packet serial included in every packet sent by the previous program (see Listing 15). The local counter is simply initialized to zero, triggering an error at the very first cycle. Since the counter is aligned at every error, this aligns the counter and, if all goes right, no other errors occur. Note that there is no limit to numbers: the values are tested for equality so overflow is not a problem, the counters overflow at the same time. A minor glitch is the possibility to have an erroneous count of packet loss, in case of data loss during the counter overflow phase.

```
for( j=0, *lung=0; j<BUFFERSIZE; j+=res-10 ) {
	res=recvfrom( sfd, recvBuffer, 1034, 0,
		(struct sockaddr *)&cliaddr, &clilen );
	if( buff ) {
		memcpy( buff+j, recvBuffer+10, res-10 );
	}
	memcpy( (char *)payl->freq, recvBuffer+4, 4 );
	memcpy( (char *)payl->att_lowpass_rate, recvBuffer+8, 2 );
	memcpy( shmaddr, recvBuffer+4, 6 );
	memcpy( shmaddr+6+j, recvBuffer+10, res-10 );
	tempCounter = ntohl ( *(long *)recvBuffer );
	counter++;
	if( counter!=tempCounter ) {
		fprintf( stderr,
			"Counter (%ld) differs from received counter (%ld)\n",
			counter, tempCounter );
		counter=tempCounter;
	}
	*lung += res-10;
	if ( res == -1 ) {
		fprintf( stderr,"Recvfrom Error\n" );
		exit( 10 );
	}
}
```

Listing 15: Central control program: the data loss detecting structure (red)

During hours of operations not a single packet loss has been detected, so no further investigation is needed.

Another interesting thing is the network load. This load has been measured using **nload**, a program that estimates the network traffic, as can be seen in Picture 24.
The input data transfer value can be evaluated to 768kS/s multiplied by 64 bit/Sample multiplied by 1034/1024 (due to our headers overhead) and by 1076/0134 (due to the IP and UDP headers overhead), i.e. 51.648 Mbit/s. As can be seen from the program output, the measured data is 51.653 Mbit/sec, very similar.



Picture 24: Traffic on the server measured by nload

The same computation can be done for the output: the program sends data for FFT every 0.1s, the total data sent has a length of 906 bytes plus 66 bytes of headers, for a total of 77760 bit/s, the value shown by the program is a little bit smaller (less than 1%), because the timing of 0.1s is not precise since the fft computation time must be added.

The last interesting time to be measured is the fft computation time. The technique used is the same used for other time measurements, producing an output on the standard error (see Listing 16 for details).

```
for( ;; ) {
        pthread_cond_wait(payl->con, payl->mut);
        // Time measurement
        gettimeofday( &start, NULL );
        // Read input data
        for( j=0; j<N; j++ ) {
                fftr[j]=((float *)payl->buf)[2*j]*coeff[j];
                ffti[j]=((float *)payl->buf)[2*j+1]*coeff[j];
        }
...
        // Absolute value
        for( j=0; j<N; j++ ) {
                fftr[j] = fftr[j]/m;
                ffti[j] = ffti[j]/m;
                fftr[j] = sqrt( fftr[j] * fftr[j] + ffti[j] * ffti[j] );
                val=20*log10(fftr[j]);
                if( val < -200 ) {
                        val=-200;
                }
                fftout[j]=200+(char)val;
        }
        // time measurement
        gettimeofday( &stop, NULL );
        fprintf( stderr, "FFT time= %f mSec\n",
                (stop.tv_sec-start.tv_sec)*1000+0.001*(stop.tv_usec-start.tv_usec) );
...
```

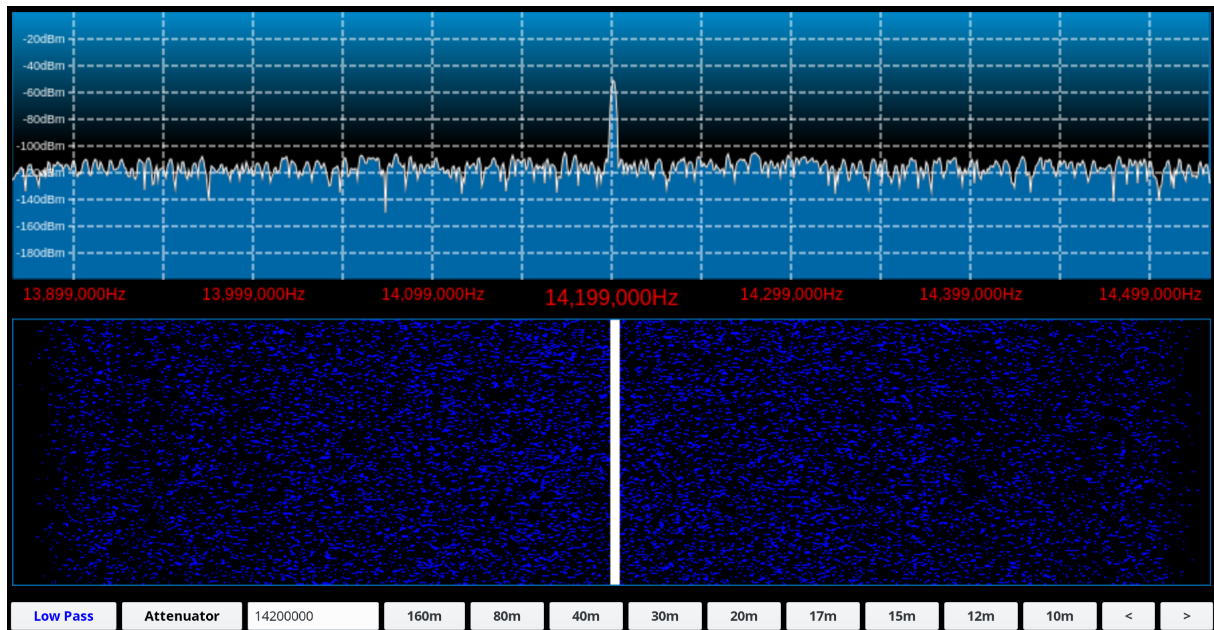
Listing 16: The instructions that measure the fft computation time (red)

On a round of more than 300 measurements the average value is 0.6ms, with a minimum of 0.21ms and a maximum of 0.91ms.

The average load of the CPU is about 3% on a quad-core hyperthreaded 2.2 GHz i7-2670QM CPU; the measurement has been done with the xfce4-cpugraph-plugin, this figure includes also the system load.

6.4 - Control Browser

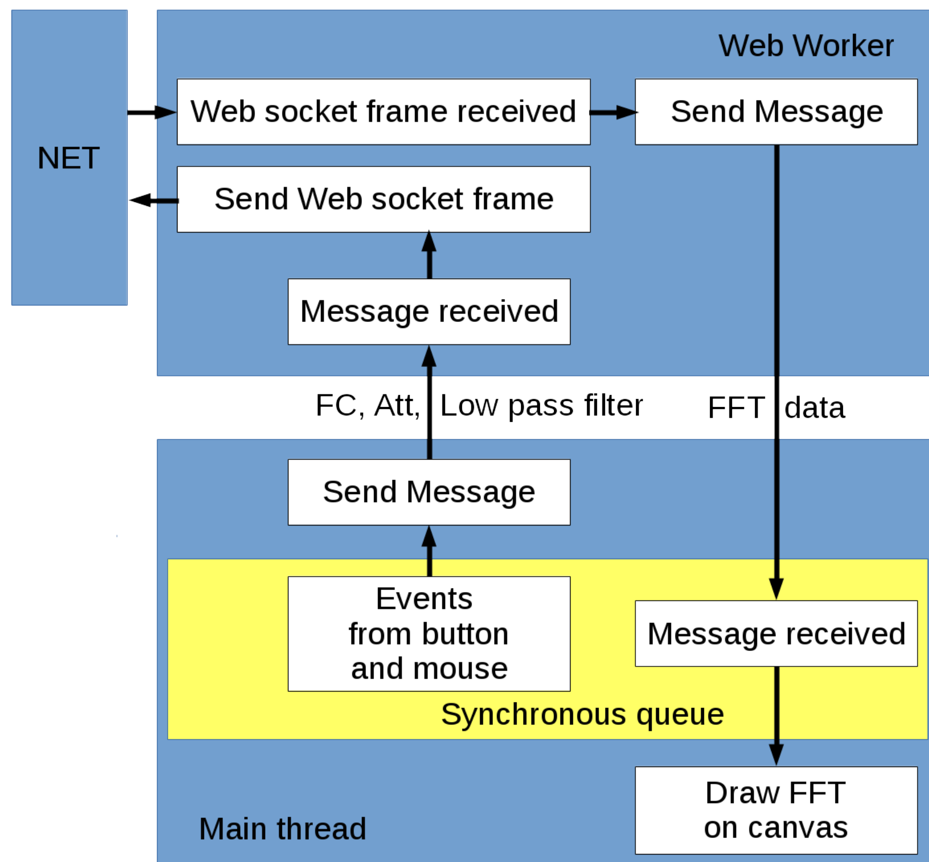
The control browser connects to the central control program using WebSocket protocol. It receives FFT data and displays spectrogram and the corresponding waterfall to assist the settings operations as can be seen in Picture 25. It also offers a simple interface to set up central frequency, attenuator, and lowpass filter of the remote sampler.



Picture 25: The control browser in action

6.4.1 - Choices

The user interface is built within a browser. This decision was made to offer the highest flexibility to the user, because when using a browser there are no limits to the platform and the location of the client interface. Using browsers implies the use of the network, and this is in line with the overall philosophy of the project.
The programming language used for the client in the browser is JavaScript. Since JavaScript is interpreted in a single thread in the browser, a Web Worker is used to allow concurrency.
The communications are done using WebSockets and the rendering of the waterfall is done using canvas and the graphical primitives for JavaScript.
The communications between the main thread and the WebSocket worker thread take place with messages. A diagram of the operations can be seen in Picture 26.



Picture 26: The control browser application structure

6.4.2 - Structure

The page is a mix of HTML for the design of the page, and JavaScript for the automation. When the page is loaded, a Web Worker is started and, inside this worker, a WebSocket is opened. The main thread event loop receives both interface events (from buttons, for example), and messages from the worker. The interface events result in sending commands through the WebSocket and the messages from the worker usually fire canvas redrawing.

6.4.2.1 - Web Worker and WebSocket

As can be seen in Listing 17, using Web Worker is straightforward: the things to be provided are mainly the synchronization between the thread and the main thread. This synchronization could be done using messages. The function self.postMessage() is used to send the data received to the main thread. This function is used within the onmessage() callback attached to the WebSocket.
JavaScript has a very powerful implementation of WebSockets that can be opened with a single line command referencing the URL. For our purposes it is important to define the binaryType of the WebSocket as ‘arraybuffer’ in order to have the data usable as an array. Also the callback can be very simple, containing only a call to the postMessage with the data received by the callback itself.

```
var mySocket = null;
var inited=0;
mySocket= new WebSocket( "ws://192.168.1.116:6666/fft", "chat" );
mySocket.binaryType='arraybuffer';
mySocket.onopen=function( event ){
        mySocket.send( "parapiglia" );
        inited=1;
}
mySocket.onclose=function( event ){
        if( inited ) {
                console.log( "WebSocket error",event );
        } else {
                console.log( "WebSocket not starting",event );
        }
}
mySocket.onmessage=function( event ){
        self.postMessage( event.data );
}
self.onmessage=function( message ) {
        console.log( "message", message.data );
        if( inited == 1 ) {
                mySocket.send( message.data );
        }
}
```

Listing 17: The Web Worker with the WebSocket implemented

`(red: password sent as first frame)`

6.4.2.2 – Canvas

```
<style>
.layer { position: absolute; }
.scale { position: absolute; top:200px; }
.nolayer { position: absolute; top:240px; }
.cmds  { position: absolute; top:450px; }
#myFFTline { z-index: 4 }
#myFFTgrid { z-index: 3 }
#myFFTdata { z-index: 2 }
#myFFTbkground { z-index: 1 }
</style>
<BODY bgcolor="Black" onload="setBackground();caio();"><TABLE><TR><TD>
<canvas class="layer" id="myFFTgrid" width="896" height="200" style="border:1px
solid #000000;"></canvas>
<canvas class="layer" id="myFFTbkground" width="896" height="200"
style="border:1px solid #000000;"></canvas>
<canvas class="layer" id="myFFTdata" width="896" height="200" style="border:1px
solid #000000;"></canvas>
<canvas class="layer" id="myFFTline" width="896" height="200" style="border:1px
solid #000000;"></canvas>
<canvas class="scale" id="scale" width="896" height="40" style="border:1px solid
#000000;"></canvas>
<canvas class="nolayer" id="myWfall" width="896" height="200" style="border:1px
solid #000000;"></canvas>
</TD></TR></TABLE><BODY>
```

Listing 18: The canvas definition

`(red: layering the areas)`

In this application the graphic interface is the biggest part, both because it is what he user looks at, and for the weight of the computations involved.
At a first sight two canvas are shown, one for the spectrogram, and the other for the waterfall. On Listing 18 it can be seen their definitions. Both canvas contain useful informations, and while the spectrogram is focused on the amplitude of every single part of the spectrum, the waterfall is focused on the history of the signals. Many elements have to be drawn: a background, some reference lines, the spectrogram, an underlaying filled area. Since all these elements require work they are divided into static (background, reference lines) and dynamic (spectrogram, filled area) elements, in such a way that only the dynamic elements are redrawn every time, while the static elements are drawn only at loading time. Listing 19 shows the drawing of the static elements, Listing 20 shows the drawing of the dynamic elements.

```
function setBackground () {
	var wfall = document.getElementById( "myWfall" );
	var ctz = wfall.getContext( "2d" );
	ctz.fillStyle="#0067A7";
	ctz.fillRect( 0, 0, 896, 200 );
	var fft = document.getElementById( "myFFTbkground" );
	var ctx = fft.getContext( "2d" );
	ctx.clearRect( 0, 0, fft.width, fft.height );
	var lingrad = ctx.createLinearGradient( 0, 0, 0, 200 );
	lingrad.addColorStop( 0, '#0089C9' );
	lingrad.addColorStop( 0.5, '#000000' );
	ctx.fillStyle = lingrad;
	ctx.fillRect( 0, 0, 896, 200 );
	ctx.beginPath();
	ctx.lineWidth=1;
	ctx.strokeStyle="#0067A7";
	ctx.setLineDash( [5, 0] );
	ctx.moveTo( 0, 0 );
	ctx.lineTo( 0, 200 );
	ctx.moveTo( 896, 0 );
	ctx.lineTo( 896, 200 );
	ctx.stroke();
	fft = document.getElementById( "myFFTgrid" );
	ctx = fft.getContext( "2d" );
	ctx.beginPath();
	ctx.lineWidth=1;
	ctx.strokeStyle="#ffffff";
	ctx.setLineDash( [5, 2] );
	for( j=20; j<fft.height; j+=20 ) {
		ctx.moveTo( 42, j );
		ctx.lineTo( 896, j );
	}
	for( j=46; j<fft.width; j+=67 ) {
		ctx.moveTo( j, 200 );
		ctx.lineTo( j, 0 );
	}
	ctx.stroke();
	ctx.textAlign="right";
	ctx.font="9px Arial";
	for( j=20; j<fft.height; j+=20 ) {
		ctx.fillStyle="white";
		ctx.fillText( -j+"dBm", 40, j+3 );
	}
}
```

Listing 19: Spectrogram background drawing

(red: linear gradient, blue: grid, green: level reference)

The elements are organized in overlapping layers, with the background as the lower one in the z-order, followed by filled area, reference lines, and, over all, the spectrogram; to cope with this four different elements, with their proper z-index, are defined using a cascade style sheet.

```
var hid = document.createElement( 'canvas' );
hid.width = points;
hid.height = 200;
var ctx = hid.getContext( "2d" );
var fft = document.getElementById( fftname );
var ctz = fft.getContext( "2d" );
var wfall = document.getElementById( wfallname );
var cty = wfall.getContext( "2d" );
ctz.clearRect( 0, 0, fft.width, fft.height );
ctx.clearRect( 0, 0, fft.width, fft.height );
ctx.lineWidth=1;
ctx.setLineDash( [4, 0] );
ctx.beginPath();
ctx.moveTo( points, 200 );
ctx.lineTo( 0, 200 );
for( j=0; j<arr.length; j++ ) {
       ctx.lineTo( j, -arr[j] );
}
ctx.fillStyle="#0067A7";
ctx.closePath();
ctx.fill();
ctx.stroke();
ctz.drawImage( hid, 0, 0 );
ctz.beginPath();
ctz.lineWidth=2;
ctz.strokeStyle="#0067A7";
ctz.setLineDash( [5, 0] );
ctz.moveTo( 895, 0 );
ctz.lineTo( 895, 199 );
ctz.stroke();
fft = document.getElementById( fftlinename );
ctz = fft.getContext( "2d" );
ctx.clearRect( 0, 0, fft.width, fft.height );
ctz.clearRect( 0, 0, fft.width, fft.height );
ctx.beginPath();
ctx.lineWidth=1;
ctx.setLineDash( [4, 0] );
ctx.lineCap='Round';
ctx.strokeStyle="#ffffff";
var tot=0;
for( j=0; j<arr.length; j++ ) {
       ctx.lineTo( j, -arr[j] );
       tot=parseInt( arr[j] )+parseInt( tot );
}
tot=tot/arr.length;
ctx.stroke();
ctz.drawImage( hid, 0, 0 );
```

Listing 20: Spectrogram drawing

(red: area under the graph, blue: graph)

6.4.2.3 - Waterfall

As depicted in Listing 21, the waterfall is drawn by copying most of the area and then tracing only the last line. To represent the levels a “relative” approach has been chosen. The levels are represented with a level of color going from black to blue in the range from medium level –5 dB to medium level +5 dB and from blue to white in the range from medium level +5 dB to medium level +15 dB. Thus focusing on the easiness to extract that particular signal from the base “noise” of the band.

```
Thr_R_color = [   "0.0",    "0.0",    "0.0", "255.0", "255.0" ];
Thr_G_color = [   "0.0",    "0.0",    "0.0", "255.0", "255.0" ];
Thr_B_color = [   "0.0",    "0.0", "255.0", "255.0", "255.0" ];
var wfall = document.getElementById( wfallname );
var cty = wfall.getContext( "2d" );
var image = cty.getImageData( 0, 1, points, 197 );
cty.putImageData( image, 0, 2 );
cty.lineWidth=1;
var imageData = cty.createImageData( points, 1 );
for( j=1; j<arr.length-1; j++ ) {
       if( arr[j] > tot+15 ) { vj=3; dx=delta_x=1;
      } else if ( arr[j] > tot+5 ) { vj=2; dx=arr[j]-tot-5; delta_x=10;
      } else if( arr[j]> tot-5) { vj=1; dx=arr[j]-tot+5; delta_x=10;
      } else { vj=0; dx=140+parseInt(arr[j]); delta_x=tot+130; }
      m = (Thr_R_color[vj+1]-Thr_R_color[vj])/delta_x;
      r = (m * dx + Thr_R_color[vj]);
      m = (Thr_G_color[vj+1]-Thr_G_color[vj])/delta_x;
      g = (m * dx + Thr_G_color[vj]);
      m = (Thr_B_color[vj+1]-Thr_B_color[vj])/delta_x;
      b = (m * dx + Thr_B_color[vj]);
      imageData.data[4*j]=r;
      imageData.data[4*j+1]=g;
      imageData.data[4*j+2]=b;
      imageData.data[4*j+3]=255;
}
imageData.data[(arr.length-2)*4   ] = 0x00;
imageData.data[(arr.length-2)*4+1] = 0x67;
imageData.data[(arr.length-2)*4+2] = 0xA7;
imageData.data[(arr.length-2)*4+3] = 0xFF;
cty.putImageData( imageData, 1, 1 );
```

Listing 21: Waterfall drawing

(red: waterfall scroll copying area)

6.4.2.4 - Buttons

To allow the user to manage the sampler some buttons and a text input are provided. Two of the buttons are used to switch on and off the attenuator and the low pass filter. The other buttons are used to set up the frequency. All the actions are performed by the means of functions that set up the corresponding variable and then send a message with the complete setup command to the WebSocket. In Listing 22 it can be seen both buttons definition, events handling, and posting message to the Web Worker.

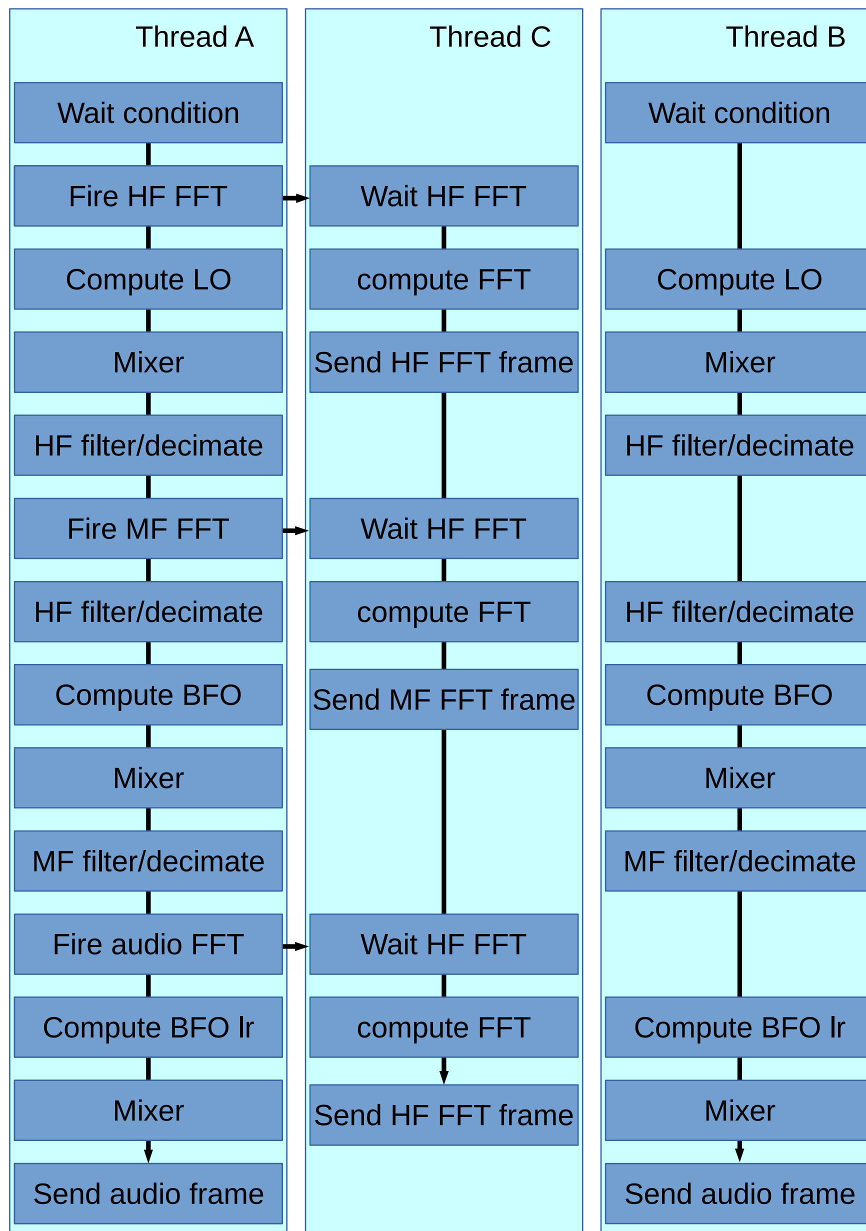
```
<script>
...
function setatt() { att=1-readatt; lp=readlp; set(); }
function setlp() { lp=1-readlp; att=readatt; set(); }
function up() {
	freq=freq+250000;
	if( freq > 30000000 ) { freq= 30000000; }
	set();
}
function down() {
	freq=freq-250000;
	if( freq<250000 ) { freq=250000; }
	att=readatt;
	set();
}
function setfr( obj ) { freq=obj; set(); }
function setfreq( obj ) {
	var obj=document.getElementById( 'FREQ' );
	freq=obj.value;
	set();
}
function set( ) {
	if( myWorker != null ) {
		myWorker.postMessage( "LP"+lp+";AT"+att+";F"+freq+";\n" );
	}
}
</script>
...
<input type="button" class="btn" style="font-weight:bold;" name="LP" id="LP"
VALUE="Low Pass" onClick="setlp();"/>
<input type="button" class="btn" style="font-weight:bold;" name="ATT" id="ATT"
VALUE="Attenuator" onClick="setatt();"/>
<input type="text" name="FREQ" id="FREQ" VALUE="14200000" size="13"
onkeydown="setfreq(event);" onclick="setfreq(event);"/>
<input type="button" class="btn" style="font-weight:bold;" name="160m" id="160m"
VALUE="160m" onClick="setfr(1910000);"/>
...
<input type="button" class="btn" style="font-weight:bold;" name="DOWN" id="DOWN"
VALUE="<" onClick="down();"/>
<input type="button" class="btn" style="font-weight:bold;" name="UP" id="UP"
VALUE=">" onClick="up();"/>
...
```

Listing 22: Buttons management

(red: post message to the WebSocket Web Worker)

6.5 - Client Receiver

This part of the project works within the server, sends audio, and exchanges controls with a web application that runs in the browser. The application reads data from the shared memories and acts as a "superhet" radio, using a local oscillator to convert the selected frequency to a "zero beat" intermediate frequency, then applies filters to sharpen the band (and decimation to reduce the bit rate) and then uses a BFO (Beat Frequency Oscillator) to move the signal to the proper audio frequency. During the operation some FFT are provided to help the operator select the desired signal. All these operations are sketched into Picture 27.



Picture 27: Client receiver structure (without WebSocket thread)

6.5.1 - Choices

The local oscillator is computed a bank at a time using the values of sin() and cos() to compute the values using the well-known prosthaphaeresis formulas.
The mixing also uses this formulas to compute the difference between the frequencies in the received signal and the frequency of the local oscillator.
Two different types of oscillator are used: the first is used to avoid aliasing during the process of decimation, the second is used to shape the bandwidth after the decimation.
To lower the computation price of the filters the decimation of the sample rate is not done in a single step, but it is carried out in various “divide by two” phases, thus reducing the weight of the filtering because every phase requires half of the computation of the previous phase even if it is the same filter.
The filters used are FIR, chosen for the stability, and Butterworth IIR, chosen for the efficacy and the low ripple in band.
We have decided not to use CIC decimation filters in favor of floating point signal representation; while it is right that CIC filters are computationally lighter, the overall load on the given computer is so low that this reduction is not so important.
The implementation of the FIR is not made using a single “central” delay-line but using two lines, one for the input, the other for the output, to cope with the potential instability of the algorithms.
We compute three different FFTs: the first is a panoramic one, to show what is happening in the whole band and is used for a “fast” frequency setup on the user interface, the second is a detailed one, which allows to see the frequency neighboring and to precisely tune a signal, and the third is useful to see what is happening in the “audio” band.
The demodulations provided are just the demodulations that can be done with a frequency translation, so it is possible to receive single side band or telegraphy signals. Also amplitude modulation can be received, simply using one of the two side bands.

6.5.2 – Radio Receiver Implementation

Before illustrating the structure of the program it is mandatory to explain how and why this structure has been built. This leads to the radio processing flow.
The model used is not very far from a classic superhet radio concept, with a little difference intrinsic to its informatics being.
But why a superhet? Because most of the needs for a superhet have not been superseded by informatics nor mathematics, at this time.
It still has more sense to build a fixed frequency filter and convert the frequency to pass through it than to build a variable frequency filter (yes, in ancient analog days it was more related to “tuned amplifiers” than to filters, but it makes no real difference).
Se we start to convert the signal. But, here is the difference, not to a “middle frequency” but to zero. Or, if you prefer, the value of our “middle frequency” is zero Hz.
Here lays the main difference between digital and analog. While with analog circuits it was wise to build a tuned amplifier using tuned (i.e: bass band) circuits, we better build low pass filter.
But remember: in the digital word we have both positive and negative frequencies, so this low pass filter is, in facts, a pass band filter centered on zero Hz.
But why do we use a low pass filter? Essentially because we have to reduce the sample rate that, at the origin, is very high, and, before reducing it, we must take care of Nyquist’s limits: before halving the samples we must halve the passband, or, better speaking, we must ensure

that in the signal there is no frequency over the Nyquist limit (half sample rate, to be short), frequencies that, inevitably, will be “reflected” to disturb our signal.
Why don’t you just instruct the sampler to reduce your sample rate? Because one of the important improvements of the SDR over old analog radios is the “pan-adapter”, or the ability to “see” the radio signals.
On an ancient radio (no matter if it is a wonderful Collins or Rhode&Schwarz) the operator acts like a blind mouse rotating the tuning knob and hoping to fall on a calling counterpart, while even on the worst SDR they really see where the signals are.
And this difference is more remarkable when the visible band is wider. So, to spoil as possible the SDR concept, the band shown must be as wide as possible. This, in turn, leads to a sample rate as high as possible, or as reasonable. In our implementation we chose a value that allows a full view of most HF amateur bands.
So we start with a high sample rate, and make an FFT to let the operator have a bird view of what happens on the band. Then we have to focus on the neighborhood of the interested frequency.
First of all we do a conversion to put the desired frequency at the middle on the band (heterodyne to medium frequency of zero Hz), the we start a process of filtering and reducing the sample rate.
But why reducing the sample rate? For two reasons: the first is that the flow contains informations we do not need: when receiving a single sideband signal we rely on frequencies in the range 300-3400 Hz apart from the (not sent) carrier frequency, so sample rates over 8kS/s are not necessary.
Remember that every single sample requires an elaboration so, to save computing power, we are happy to throw away unnecessary and unwanted samples.
But there is a better and more interesting reason to reduce the sample rate: during filtering process we “mix” the values and the output of the filter/decimation process has more “significant” bits than the input. In other words the samples we throw return as more levels, being the reference value, and limit, an increment of approx half bit for every sample rate halving.
So we filter and decimate various times. Why not only one shot? Because the weight of a filter depends linearly on the sample rate, so the best solution is to reduce the sample rate, but, before that, we have to filter, and, as a result, the best is to cycle through filter-a-few/halve-samplerate in order to minimize computer load.
At a certain point we produce an “enlarged fft” to let the operator have a “zoomed” view in the neighborhood of the managed frequency, allowing fast and neat frequency hops to listen to a nearby signal if needed, then we continue the filtering process until we reach a suitable sample rate.
In this project this value is 24kS/s. Why? Because the audio context implementation on the Firefox browser accepts processing sample rates between 20.5hkS/s and 48KS/s and we prefer to remain as low as possible being also a sub-multiple of 192kS/s, that is the base rate of the sampler.
At the end we simply discard Q signal and route I signal to the audio channel. All finished?
Not completely. Remember than our filter has the peculiarity to filter from -*x*Hz to +*x*Hz. Suppose to tune to 7070kHz and to use a filter from -3400Hz to +3400Hz: you receive at the same time both a USB and a LSB communication with this (suppressed) carrier. Bad news!
So what can we do? Simple! First of all we use a filter from -1550 to +1550 Hz. Then, if we need to receive LSB (standard for 40m band), we “tune” not to 7070kHz but to 7068150Hz (-1850Hz). This ensures that the LSB channel (7066600-7069700) is centered in the range -1550/+1550. But this also means that the signal is in this range, so, after filtering we add 1850Hz to re-translate the signal to the proper 300-3400 range.

Se we have to add a mixer, no more. But this is interesting because we can plan to move both frequencies together, allowing the filter to became a “sliding one”, useful to out-filter a neighbor unwanted communication.

6.5.3 - Structure

The program starts by launching the two threads responsible for the management of the data arriving in the two shared memories. Both threads share the same code. Another thread is used to compute the various FFTs, once the relative data is ready. A thread is used to manage the WebSocket that is the communication media with the user interface.
The most remarkable thing is that the stream data flow is processed one block at a time, using two sets of data banks, alternatively filled and processed.
Each data bank has its complete set of data (input data, oscillator data, post-mixer data, filtered data,…) with the remarkable exception of local oscillator banks, that do not follow data banks because, since only one data bank is processed at a time, it is not necessary to compute two series of oscillator data, but the data is computed only for the bank being used.
This allows the usage of static bank “shared” between the two threads: in fact the two threads execute the same function and, in turn, this function uses the same oscillator buffer, that has been declared as static. This is not a problem because the data flow ensures that the two threads do not overlap, provided that the thread process time is shorter than the bank filling time, condition that is “the” prerequisite for a real time data process as this project.
The last detail is that two of the three FFTs require an input buffer of 1024 couples of I+Q values, more than the values in a single bank, after the first decimations are made, so two static buffers are provided, and filled, by the two different threads. Also for these operations the non overlap of the two threads is handy because it avoids the need of synchronization.
In a future enhancement the complete set of banks could be static, letting only the two input shared memories to be “local” to the thread. At the moment this has not been done to allow faster and simpler debug and profiling.
Less fascinating than the DSP part, but also important, is the thread responsible for the management of commands from the User Interface. This thread manages a WebSocket used not only for receiving the commands, but also for sending FFTs and audio data. This is possible because WebSockets implement, over the TCP layer, an application layer that allows full bidirectional frame exchange, so the WebSocket management thread mainly receives and interprets frames from the browser user interface, the two DSP elaboration threads send audio frames, and the FFT thread sends FFT and command confirmation values. Since the two DSP threads do not overlap and the WebSocket only receives frames, the only critical overlap situation is from a DSP thread and the FFT thread. The data does not mix because the frames are short and locally preassembled before sending; also, in this situation there is no critical path and no need for synchronization.

6.5.3.1 - Oscillators

As can be seen in Listing 23, the local oscillator is computed a bank at a time using the values of sin() and cos() to compute the values using the well-known prosthaphaeresis formulas. No matter if the function is called from a thread or another, it computes the values to fill the same bank, because only a thread is active at each given time.
The "seeds" for the calculation (sin() and cos() for the elementary angle, and last values in the bank) are kept static so trigonometric functions are computed only the first time and in case of a frequency change. An amplitude correction algorithm is applied to maintain stable the local oscillator amplitude.

```
static float inphase[512*3]={0};
static float quadrature[512*3]={0};

void bufferLO( int freq, float *inphase, float *quadrature, int howMany ) {
	static int isFirst=1;
	static double amplitude=1; static int myfreq=0; static double phi=0;
	static double cosphi=1; static double sinphi=0;
	static double costmp=1; static double sintmp=0;
	static double cossav=1; static double sinsav=0;
	int j;
	if( isFirst ) {
		inphase[howMany-1]=0;
		quadrature[howMany-1]=1;
	}
	if( myfreq != freq ) {
		myfreq=freq;
		phi=-freq*2*3.1415926535897932384/768000;
		cosphi=cos( phi );
		sinphi=sin( phi );
	}
	amplitude=1.0d+(1.0d-sqrt( sintmp*sintmp+costmp*costmp ))/howMany;
	for( j=0; j<howMany; j++ ) {
		sinsav=sintmp*cosphi+costmp*sinphi;
		cossav=costmp*cosphi-sintmp*sinphi;
		sintmp=amplitude*sinsav; costmp=amplitude*cossav;
		inphase[j]=costmp; quadrature[j]=sintmp;
	}
}
```

Listing 23: Local Oscillator

6.5.3.2 - Mixers

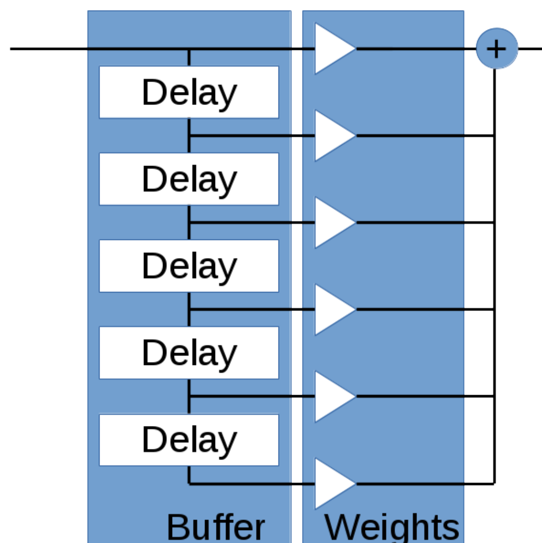
The various mixers are an application of the prosthaphaeresis formulas. This has been made possible by the fact that both signal and oscillators are provided in I/Q format, or, in other words, that both sin() and cos() components of the signals are available. Listing 24 shows how conceptually simple is this part of the work.

```
void bufferMixer( float *in, float *out, int howMany, float *osci, float *oscq ) {
        int hm=howMany/4;
        int j,k;
        float inf, qua;
        for( j=0,k=0; j<hm; j+=2,k++ ) {
                inf=in[j];
                qua=in[j+1];
                // cos(a+b)=cos(a)*cos(b)-sin(a)*sin(b)
                out[j]=inf*osci[k]-qua*oscq[k];
                // sin(a+b)=cos(a)*sin(b)+sin(a)*cos(b)
                out[j+1]=inf*oscq[k]+qua*osci[k];
        }
}
```

Listing 24: Mixer (using prostapheresys formula)

6.5.3.3 - Filters

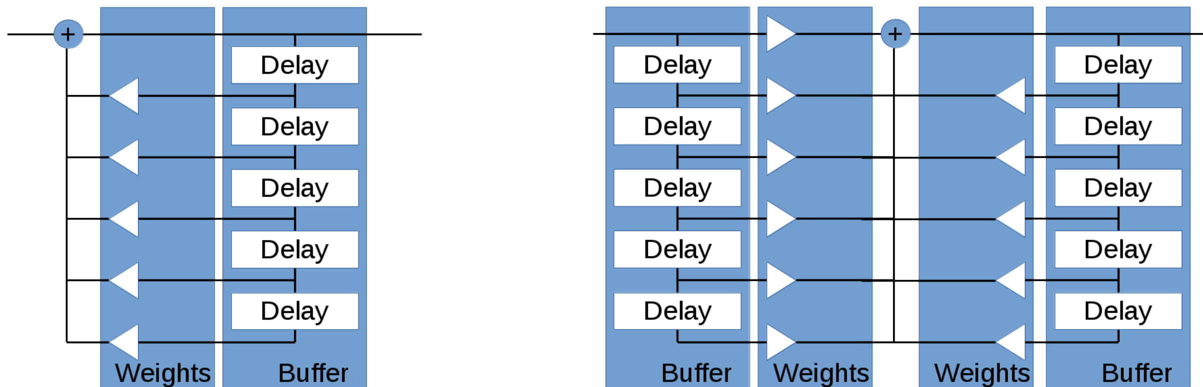
Two kind of filters are implemented: FIR (see Picture 28) and IIR (see Picture 29). At the very beginning it was thought that the better implementation for the “filter and decimate” was a CIC approach but, at a better analysis, it resulted that CIC implementation relies on integer mathematics because floating points does not allow a precise compensation of values, so a more conservative approach was used.



Picture 28: FIR filter schematic

For the “HF” part (higher sample rates, mainly anti-alias filter) a FIR was used, to exploit its superior stability, while for the “MF” part a IIR of Butterworth filter is used.
While the FIR implementation (shown in Listing 25) is straightforward, the IIR implementation (shown in Listing 26) requires some caution because one of the implementations, with a single delay line (buffer), is prone to instability and the other, with two delay lines, is preferable.

An important observation must be done about filters delay lines: those buffers are static and not “thread local” for the same reason that allows oscillators buffers to be static, the fact that only a thread at a time is active.



Picture 29: One (left) and two (right) delay line IIR filter schematics

```
void bufferFilter( float* in, float*out, int howMany, double *coeff,
	float *historyi, float *historyq, int lung, int *punt ) {
	int j,k,y;
	int hm=howMany/4;
	for( j=0,y=0; j<hm; j+=4,y+=2 ) {
		historyi[(++*punt)%lung]=in[j];
		historyq[(*punt)%lung]=in[j+1];
		historyi[(++*punt)%lung]=in[j+2];
		historyq[(*punt)%lung]=in[j+3];
		out[y]=0;
		out[y+1]=0;
		for( k=0; k<lung; k++ ) {
			out[y]+=historyi[(k+*punt)%lung]*coeff[k];
			out[y+1]+=historyq[(k+*punt)%lung]*coeff[k];
		}
	}
}
```

Listing 25: FIR filter implementation

```
void audioFilter( float *in, float *out, int howMany, int order,
        double *denom, double*numer, double *histii, double *histiq,
        double *histui, double *histuq, int divide ) {

        int j,k;
        long double acci, accq;
        for( j=0; j<howMany; j+=2 ) {
                for( k=order; k>0; k-- ) {
                        histii[k]=histii[k-1];
                        histiq[k]=histiq[k-1];
                        histui[k]=histui[k-1];
                        histuq[k]=histuq[k-1];
                }
                histii[0]=in[j];
                histiq[0]=in[j+1];
                acci=0.0;
                accq=0.0;
                for( k=0; k<=order; k++ ) {
                        acci+=histii[k]*numer[k];
                        accq+=histiq[k]*numer[k];
                }
                for( k=1; k<=order; k++ ) {
                        acci-=histui[k]*denom[k];
                        accq-=histuq[k]*denom[k];
                }
                histui[0]=acci;
                histuq[0]=accq;
                out[j]=acci;
                out[j+1]=accq;
        }

        if( divide > 1 ) {
                for( j=divide*2, k=2; j<howMany; j+=divide*2,k+=2){
                        out[k]=out[j];
                        out[k+1]=out[j+1];
                }
        }
}
```

Listing 26: IIR filter implementation

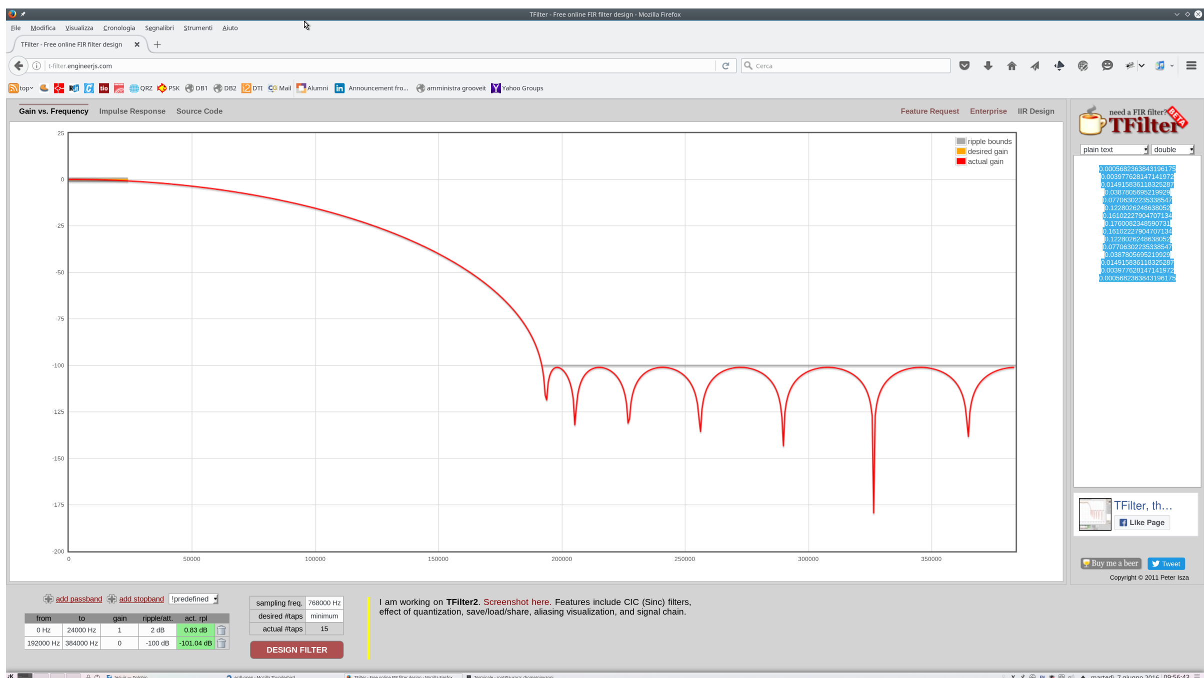
6.5.3.4 - HF FIR Filters Sizing

These filters are needed mainly as anti alias filters before each sample rate halving. There are four calls to such a filter. To optimize the filters, the first two are 15 taps filters while the second couple of filters are 23 taps ones. This because each filter is computed on half points than the previous and, also, its pass band ripple is more important than the previous because the band is lesser.

To compute the filters we can use various applications, starting with the well-known Matlab. We chose to use a web application named Tfilter that is very simple to use, as can be seen in Picture 30 and 31.

The filters are shaped to obtain 100 dB of attenuation of frequencies over the Nyquist limit and to maintain the in band ripple under one dB.

The program is built in such a way that it is straightforward to recompute a filter with different values and insert these values in the source code, in Listing 27 it is reported how the coefficients are inserted in the source code, and Listing 28 shows how the filters are called.

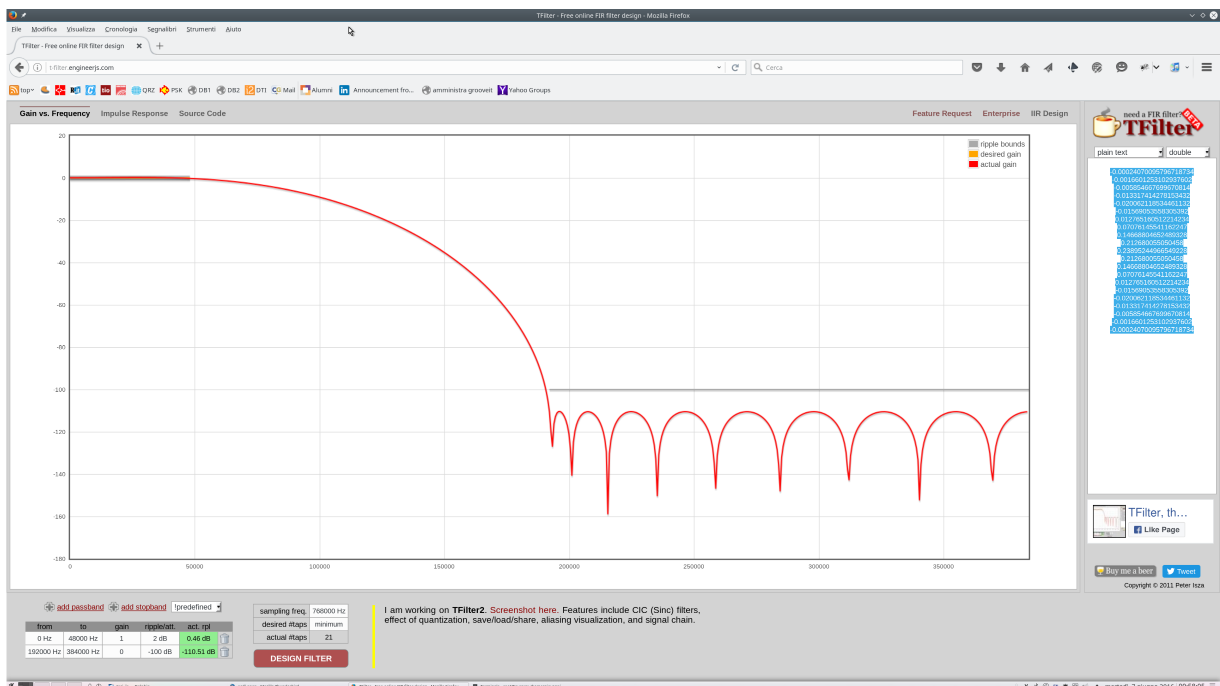


Picture 30: Parameters, response, and coefficients of the first, 15 coefficients, HF filter

http://t-filter.engineerjs.com/

```
static double coeff0[]={0.0005682363843196175, 0.003977628147141972,
       0.014915836118325287, 0.0387805695219929, 0.07706302235338547,
       0.1228026248638052, 0.16102227904707134, 0.1760082348590731,
       0.16102227904707134, 0.1228026248638052, 0.07706302235338547,
       0.0387805695219929, 0.014915836118325287, 0.003977628147141972,
       0.0005682363843196175 };
static double coeff1[]={0.0005682363843196175, 0.003977628147141972,
       0.014915836118325287, 0.0387805695219929, 0.07706302235338547,
       0.1228026248638052, 0.16102227904707134, 0.1760082348590731,
       0.16102227904707134, 0.1228026248638052, 0.07706302235338547, 0
       .0387805695219929, 0.014915836118325287, 0.003977628147141972,
       0.0005682363843196175 };
static double coeff2[]={-0.00024070095796718734, -0.0016601253102937602,
       -0.005854667699670814, -0.013317414278153432, -0.020062118534461132,
       -0.01569053558305392, 0.012765160512214234, 0.07076145541162247,
       0.14668804652489328, 0.212680055050458, 0.23895244966549228,
0.212680055050458,
       0.14668804652489328, 0.07076145541162247, 0.012765160512214234,
       -0.01569053558305392, -0.020062118534461132, -0.013317414278153432,
       -0.005854667699670814, -0.0016601253102937602, -0.00024070095796718734};
static double coeff3[]={-0.00024070095796718734, -0.0016601253102937602,
       -0.005854667699670814, -0.013317414278153432, -0.020062118534461132,
       -0.01569053558305392, 0.012765160512214234, 0.07076145541162247,
       0.14668804652489328, 0.212680055050458, 0.23895244966549228,
0.212680055050458,
       0.14668804652489328, 0.07076145541162247, 0.012765160512214234,
       -0.01569053558305392, -0.020062118534461132, -0.013317414278153432,
       -0.005854667699670814, -0.0016601253102937602, -0.00024070095796718734};
```

Listing 27: Filter coefficents in the source code



Picture 31: Parameters, response, and coefficients of the second, 21 coefficients, HF filter

http://t-filter.engineerjs.com/

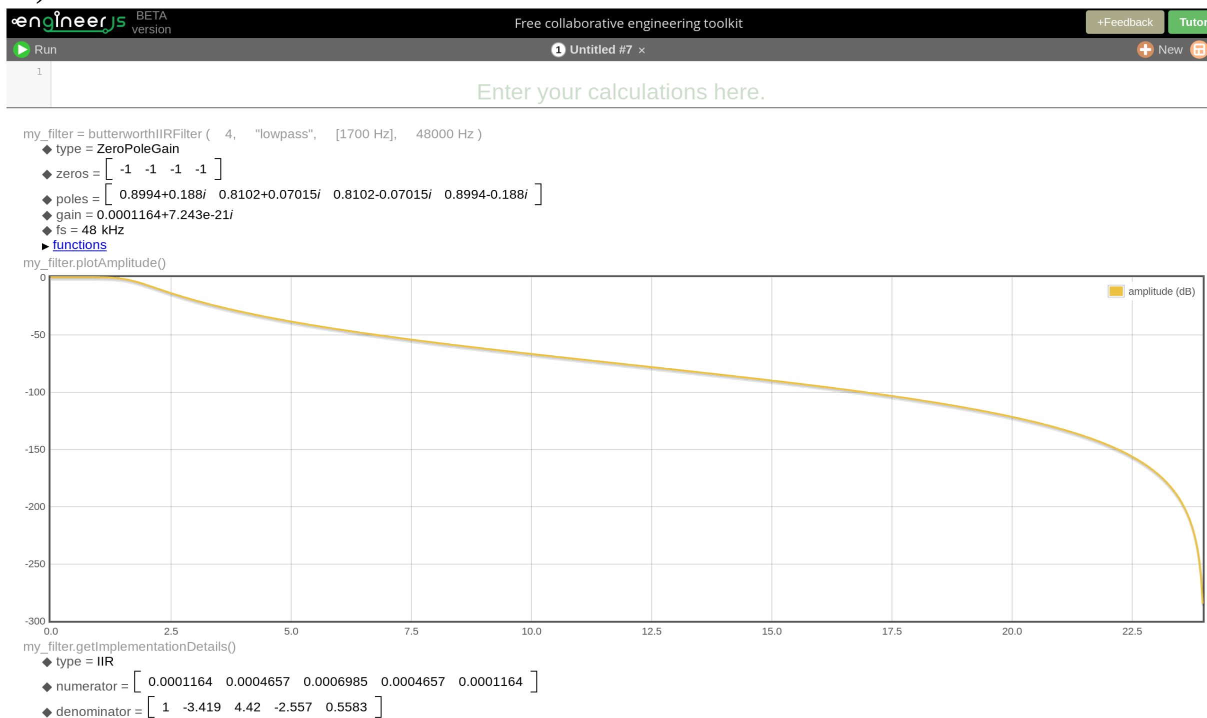
```
bufferFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE, coeff0,
       historyi0, historyq0, coeff_lung0, &coeff_punt0 );
bufferFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/2, coeff1,
       historyi1, historyq1, coeff_lung1, &coeff_punt1 );
memcpy( payl->bufmed+sect1, payl->buf, BUFFERSIZE/4 );
sect1+=BUFFERSIZE/4;
if( sect1 >1024*16 ) { // 4+4 bytes(I+Q)*2(because FFT decimate)
       sect1=0;
       pthread_cond_signal( payl->con_c );
}
bufferFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/4, coeff2,
       historyi2, historyq2, coeff_lung2, &coeff_punt2 );
bufferFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/8, coeff3,
       historyi3, historyq3, coeff_lung3, &coeff_punt3 );
```

Listing 28: Calls to the FIR filter implementation

(red: first filter, blue: second filter)

6.5.3.5 - Audio IIR filters sizing

For the selectivity shaping filters we apply many times the same filter. This filter is an IIR one, chosen both to have a better shape and to experience a different type of filter.
For the design of IIR filter we chose another tool, http://engineerjs.com/ (see Picture 32), that allows not only the design of a filter but also the simulation of the same filter. This simulation is an important and useful feature because, due to the particular nature of the IIR filters, i.e. their infinite response, it is possible that this nature mixed with the approximation of the used mathematics, can result into instabilities. We computed a very simple filter with only few coefficients (see Listing 29), but, in the code, this filter was applied many times (see Listing 30).



Picture 32: Design of audio IIR filter

http://engineerjs.com/

```
static double denom1[]={ 0, -3.419, 4.42, -2.557, 0.5583 };
static double numer1[]={ 0.0001164, 0.0004657, 0.0006985, 0.0004657, 0.0001164 };
```

Listing 29: IIR filter coefficients

```
audioFilter( (float *)payl->buf, (float *)payl->buf,
       BUFFERSIZE/16/4, 4, denom1, numer1, histii0,
       histiq0, histui0, histuq0, 1 );
audioFilter( (float *)payl->buf, (float *)payl->buf,
       BUFFERSIZE/16/4, 4, denom1, numer1, histii1,
       histiq1, histui1, histuq1, 1 );
audioFilter( (float *)payl->buf, (float *)payl->buf,
       BUFFERSIZE/16/4, 4, denom1, numer1, histii2,
       histiq2, histui2, histuq2, 1 );
audioFilter( (float *)payl->buf, (float *)payl->buf,
       BUFFERSIZE/16/4, 4, denom1, numer1,
       histii3, histiq3, histui3, histuq3, 2 );
```

Listing 30: Calls to IIR filter

6.5.3.6 - FFT

To compute the FFT we used the same algorithm of the central control program, so there is no need to describe it again, apart from the fact that the audio FFT is computed only on “odd samples” to enlarge the spectrum. This has no aliasing consequence because the signal is well frequency limited (we use a high sample rate only for the limits imposed by the audio context). For the same reason half of the computed spectrum carries no information and is dropped before composing the WebSocket frame to be sent to the browser that shows the user interface.

6.5.3.7 - Putting All Together

The two threads that make the computations over the two banks perform all the operations we just saw: they start waiting for the condition variable to be released, and then signal for the fft, compute oscillator values, mix, perform the filter/decimation computation, fire the condition variable for the enlarged FFT, compute the BFO oscillator values, mix, perform selectivity filtering, fire the audio FFT, re-convert the base band frequency to the right values and, at the end, send the audio as a WebSocket frame to the browser that implements the user interface, all these operations can be seen in Listing 31.

```
for( ;; ) {
	pthread_cond_wait( payl->con, payl→mut );
	if( payl->payload ) {
		pthread_cond_signal( payl->con_a );
	}
	memcpy( (char *)payl->tune, payl->shmaddr-6, 4 );
	memcpy( (char *)payl->att_lowpass_rate, payl->shmaddr-2, 2 );
	bufferLO( *payl->freq, inphase, quadrature, BUFFERSIZE/8 );
	bufferMixer( (float *)payl->shmaddr, (float *)payl->buf,
		BUFFERSIZE, inphase, quadrature );
	bufferFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE,
		coeff0, historyi0, historyq0, coeff_lung0, &coeff_punt0 );
	bufferFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/2,
		coeff1, historyi1, historyq1, coeff_lung1, &coeff_punt1 );
	memcpy( payl->bufmed+sect1, payl->buf, BUFFERSIZE/4 );
	sect1+=BUFFERSIZE/4;
	if( sect1 >1024*16 ) { // 4+4 bytes(I+Q)*2(because FFT decimate)
		sect1=0;
		pthread_cond_signal( payl->con_c );
	}
	bufferFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/4,
		coeff2, historyi2, historyq2, coeff_lung2, &coeff_punt2 );
	bufferFilter( (float *)payl->buf, (float *)payl→buf, BUFFERSIZE/8,
		coeff3, historyi3, historyq3, coeff_lung3, &coeff_punt3 );
	bufferBFO( *payl->bfo, BFOinphase, BFOquadrature, BUFFERSIZE/16/8 );
	bufferMixer( (float *)payl->buf, (float *)payl->buf,
		BUFFERSIZE/16, BFOinphase, BFOquadrature );
	audioFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/16/4,
		4, denom1, numer1, histii0, histiq0, histui0, histuq0, 1 );
	audioFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/16/4,
		4, denom1, numer1, histii1, histiq1, histui1, histuq1, 1 );
	audioFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/16/4,
		4, denom1, numer1, histii2, histiq2, histui2, histuq2, 1 );
	audioFilter( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/16/4,
		4, denom1, numer1, histii3, histiq3, histui3, histuq3, 2 );
	memcpy( payl->buftemp+sect, payl->buf, BUFFERSIZE/32 );
	sect+=BUFFERSIZE/32;
	if( sect >1024*16 ) { // 4+4 bytes(I+Q)*2(because FFT decimate)
```

```
                sect=0;
                pthread_cond_signal( payl->con_b );
        }
        bufferBFO1( -*payl->bfo, BFO1inphase, BFO1quadrature, BUFFERSIZE/32/8 );
        bufferMixer( (float *)payl->buf, (float *)payl->buf, BUFFERSIZE/32,
                BFO1inphase, BFO1quadrature );
        buf[0]=0x82;
        buf[1]=0x7e;
        buf[2]=0x00;
        buf[3]=0xC4;
        buf[4]=0x04; //audio
        buf[5]='A';
        buf[6]='F';
        buf[7]=0x01;
        for( j=0,k=8; j<BUFFERSIZE/32; j+=8,k+=4 ){
                memcpy( buf+k, payl->buf+j, 4 );
        }
        if( *payl->outfile > 0 ) {
                k=write( *payl->outfile, buf, 200 );
                if( k==-1 ) {
                        fprintf( stderr, "Error writing audio stream error %d - %s\n",
                                errno, strerror(errno) );
                }
        }
}
```

Listing 31: The signal processing thread

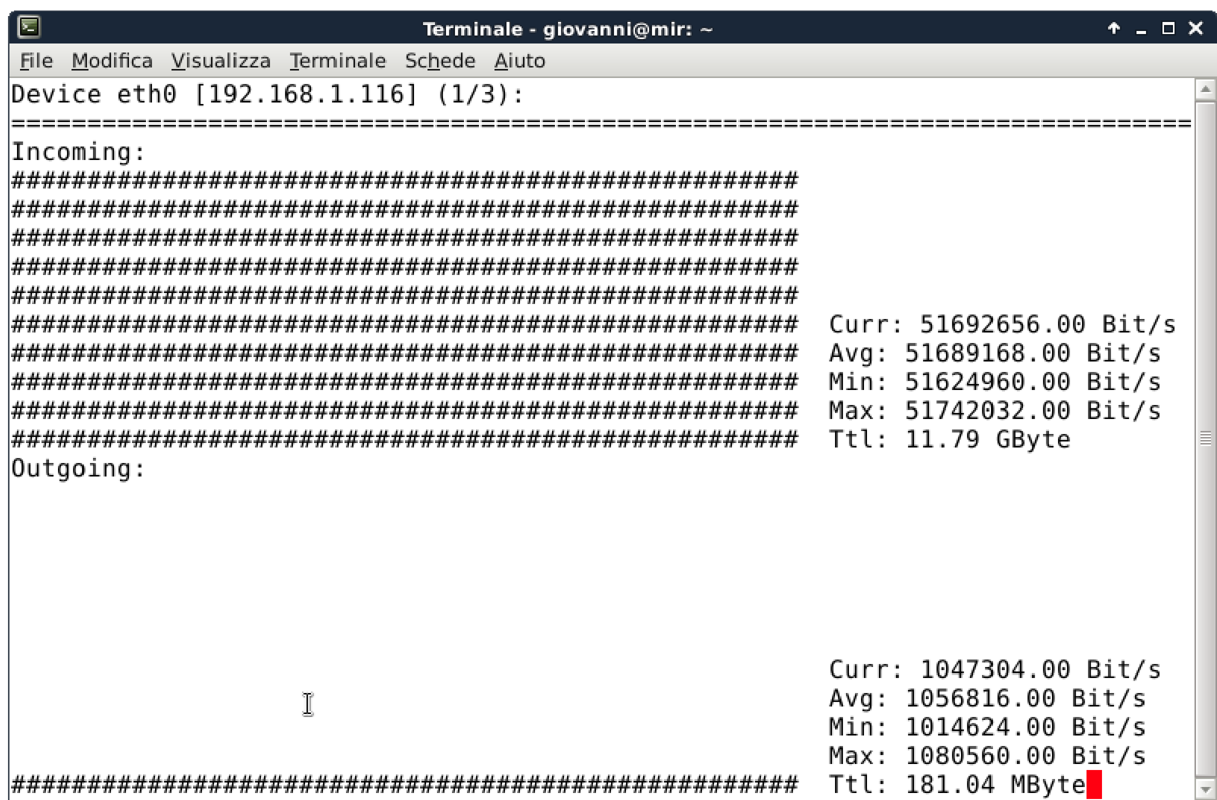
(red:oscillators, blue: mixers, green:filters, violet: WebSocket frame with audio data)

6.5.3.8 - WebSocket Management

To interact with the user interface a WebSocket is used. Its working is very similar to the WebSocket implemented in the central control program, with the main difference that it also sends an audio frame containing 96 audio samples at 24kS/s, as can be seen in Listing 31.

6.5.4 - Evaluations and Measures

This part of the work is the most computation intensive task. Since it can be used by many users it is important to measure the CPU load to verify how many clients can be served by a given server.
Some estimations are carried out using the standard CPU monitoring tool **xfce4-cpugraph-plugin**, and the load estimated is around 17% with one single client running (this includes operating system, elad-server and one instance of elad-client accepting connections), 34% with two clients running and 50% with three clients running, on a 2.2 GHz 4 core hyperthreaded i7-2670QM Intel CPU. It seems that this CPU could support no more than 5 clients. This CPU is a laptop specialized one, approx five years old, so a modern server could perform better. For example, as per the document present at https://www.cpubenchmark.net/high_end_cpus.html, the i7-2670QM CPU is accredited a mark of 5,945, where the E5-2679 CPU has a mark of 25,911, more than 4 times, and also has 10 cores and 20 threads that are 2.5 times the core/threads of the i7-2670QM CPU. Based on these estimations it is possible that, using this processor, this application can scale to ten / twenty simultaneous clients.

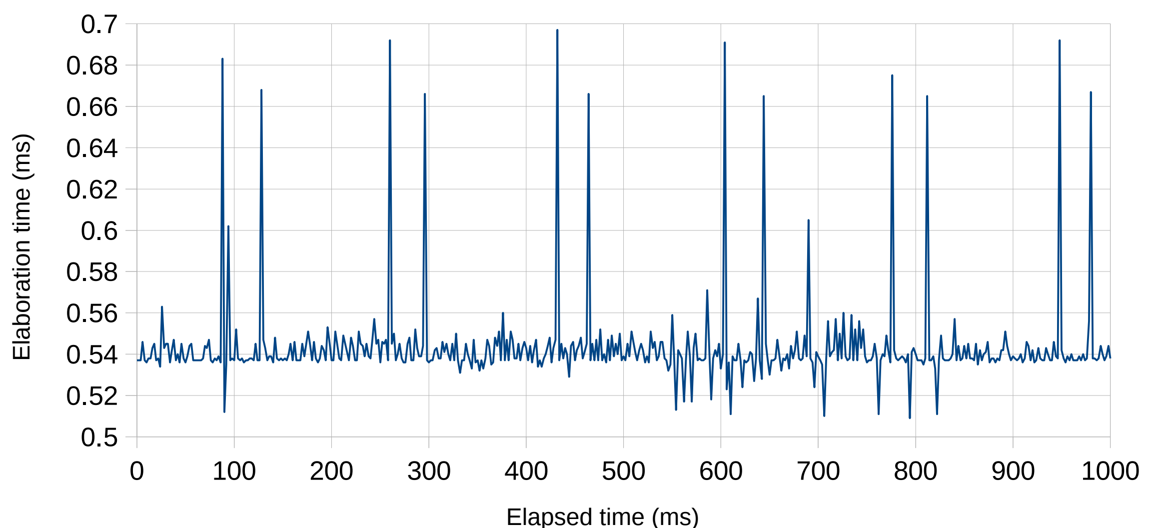


Picture 33: Elad-server and elad-client aggregated bandwidth

The network load is also measured (as can be seen in Picture 33), using **nload**, and the data rate measured, about 1.05Mbit/s, that also aggregates the data flow generated by elad-server (about 77kb/s), reflects the following computations; the packets sent are of four types, all WebSocket TCP packets. Their data length are:

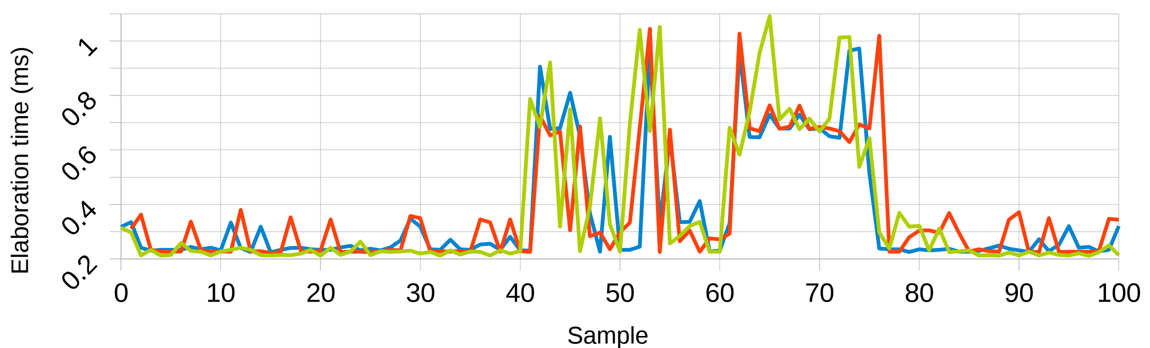
- 916 bytes for HF FFT, every 90ms
- 916 bytes for MF FFT, every 90ms
- 632 bytes for BF FFT, every 90ms
- 200 bytes for “audio” signal, every 2ms

for a total of approx 1Mb/s of pure data, without TCP/IP headers (not simple to evaluate because of packet assembly done by the protocol).



Picture 34: A sample of HF thread elaboration times

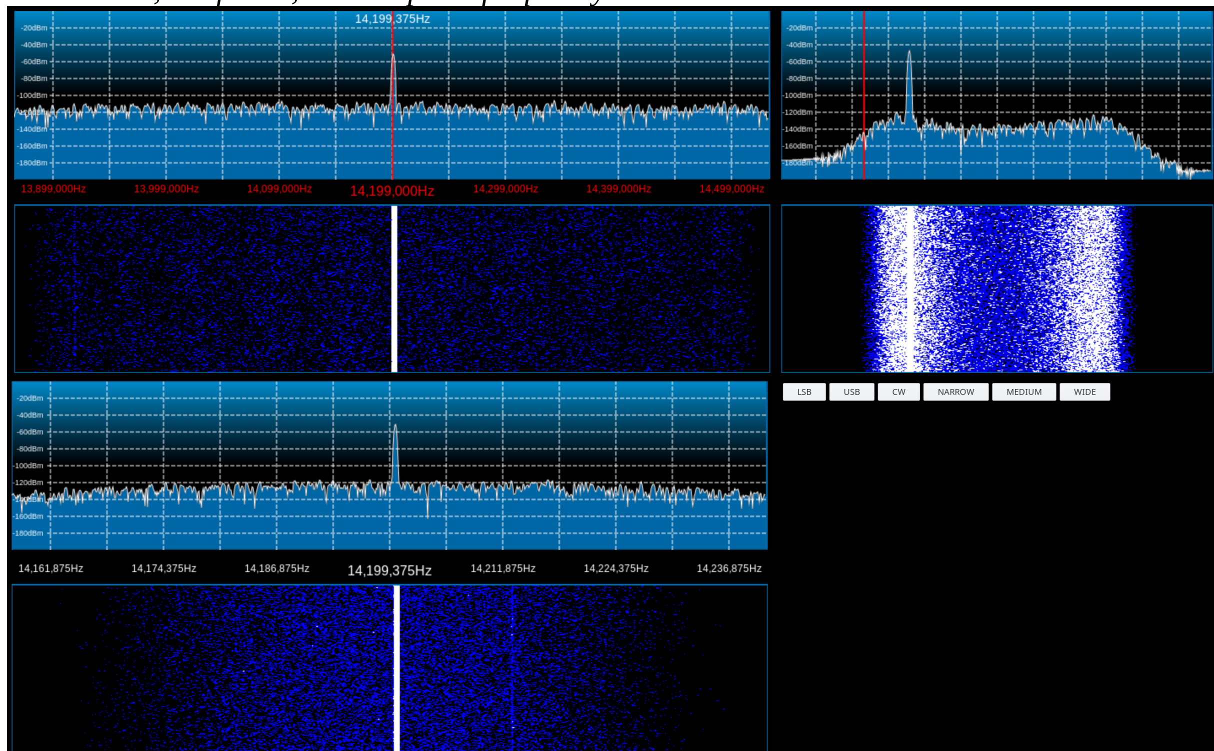
These figures show that on a typical hi-speed Internet connection of 10 Mb/s it is possible to send data to 5 clients without saturating the band, and that a hi-speed cable Internet connection (50/500 Mb/s) could be able to support the number of clients that a good server can support (the above supposed 10 to 20).
Also the duration of the HF thread has been measured, in the usual way, getting time at the beginning and at the end of the elaboration and computing the difference. A single client has been run and the data has been collected. The average value is about 0.5ms, with a minimum value of 0.4ms and a maximum of 0.7ms, as can be seen in Picture 34.
In Picture 35 the FFT times are traced to see the elaboration time that goes from 0.2ms to 1.1ms. These values do not create problems because the fft are computed every 30ms and have their computation buffer (and, also, the original buffer reference time is 2ms).



Picture 35: A sample of FFT elaboration times

6.6 - Receiver User Interface

The receiver user interface is a web application, that can be seen in Picture 36, implemented as a web page that allows visiting browsers to connect to the client receiver program using a WebSocket. It receives FFT data and displays the spectrogram and waterfall to assist the operations. It also offers a simple interface to set up the operations, like selecting the modulation, the filters, the reception frequency.



Picture 36: A screen shot of the receiver user interface

Many efforts have been made to design a proper interface that allows the operator to work in a comfortable and “familiar way”. This is not so simple because the interface of a radio receiver is a well designed and known interface: operators are familiar with tuning and volume knobs, not with mouse drag and drop. On the other hand a spectrogram allows the incomparable “point and click” frequency set, so we implemented a double mechanism that allows both point and click and “knob tuning”, using the mouse wheel as a tuning knob.

6.6.1 - Choices

The user interface is built within a browser. This decision was made to offer the highest flexibility, because using browsers avoids to force limits to the platform and the location of the client interface.
Using browsers implies the use of the network, and this is in line with the overall philosophy of the project. The language selected for programming within the browser is JavaScript. Since JavaScript is interpreted in a single thread, a Web Worker is used in the browser to allow concurrency. The communications are done using WebSockets and the rendering of the waterfall is done using canvas and the graphical primitives from JavaScript. The communications between the main thread and the WebSocket worker thread are done using messages. The audio is routed by the audio context that allows browsers to perform a good audio handling.

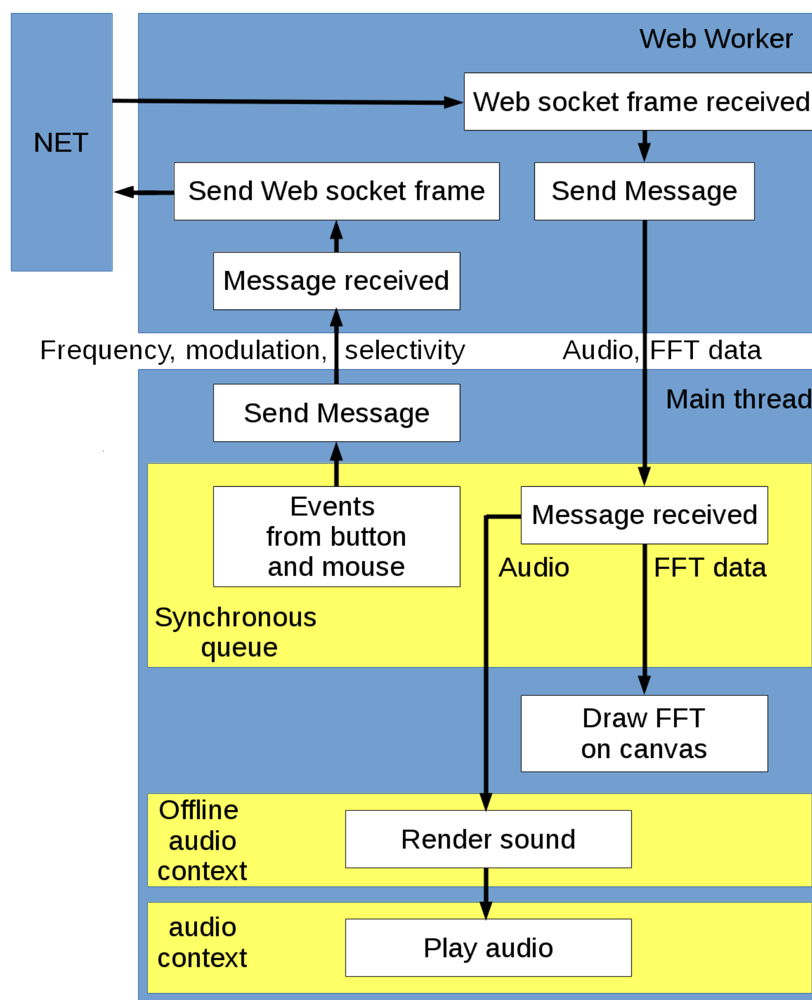
6.6.2 - User Experience

It may seem strange that a work on signal processing takes care of the user experience. But it is definitely no matter of “whistles and bells”. Radio operators are well trained professionals looking for an instrument that is efficient and effective. This not only means that the processing is technically well done and exploits well the computer resources like processor, memory, network, audio devices, but also that the operator can interact in an effective mode with the device.
This means that all the new opportunities must be caught, like having spectrograms where you “point and click” to change the frequency, but also not dropping the way of operations familiar to the operator.
The user interface has some simple objects allowing the operator to control the radio, like buttons to change modulation or selectivity. Those are buttons that can be clicked.
Also there are some spectrograms which the operator can click to set the frequency. Under each spectrogram there is a waterfall screen whose purpose is to allow the operator to perform a “time integration” by showing vertical (discontinue) lines that evidence that there were operations on a frequency even if there is no activity in the exact moment the operator is looking at the screen, or if the level of the signal is hard to be distinguished among the noisy neighborhood.
But, when the mouse is on a spectrogram, moving its wheel lets the operator change the frequency, mimicking the tuning knob operation well known and desired by the operators.
The mouse wheel sensitivity (i.e.: the amount of frequency variation associated with a rotation step) varies with the spectrogram over which is the mouse, letting the operator do both a “quick and rough” or a fine tuning.

6.6.3 - Structure

As can be seen in Picture 37, the receiver user interface is a web page that mixes HTML defined elements with some JavaScript that manages the various aspects of the operations:

- it manages the WebSocket that communicates with the client receiver; this WebSocket is managed into a Web Worker to gain a certain amount of parallelism;
- it draws the various spectrograms and waterfalls;
- it converts the received audio samples to the local audio rate using an off-line audio context to gain also here a certain amount of parallelism;
- it plays the audio using an audio context to divide audio rendering from other operations; this context manages a separate thread that is driven by the requests by the audio devices;
- it manages the events from the buttons, the canvas, the keyboard, and the mouse.



Picture 37: Web receiver user interface structure

6.6.3.1 - Websocket

The WebSocket, visible in Listing 32, is implemented into a Web Worker and is very similar to the WebSocket that is implemented in the control browser, with the main difference that there it has more than one kind of frame to manage: there are three different frames for the three FFT data plus one frame with the audio data.

```
var mySocket = null;
var inited=0;
mySocket= new WebSocket( "ws://192.168.1.116:7777/fft", "chat" );
mySocket.binaryType='arraybuffer';
mySocket.onopen=function( event ){
	inited=1;
	mySocket.send( "give me FFT" );
}
mySocket.onclose=function(event){
	console.log( "WebSocket error", event );
}
mySocket.onmessage=function( event ){
	self.postMessage( event.data );
}
self.onmessage=function( message ) {
	console.log( "message", message.data );
	if( inited == 1 ) {
		mySocket.send( message.data );
	}
}
```

Listing 32: The WebSocket thread

6.6.3.2 - Graphics

The receiver user interface has an HTML definition of the various layers, as can be seen in Listing 33. It differs from the control browser mainly for the different number of canvas required to draw the various spectrograms and waterfalls, and the different placing of the various buttons.

```
<style>
.layer { position: absolute; }
.scale { position: absolute; top:200px; }
.nolayer { position: absolute; top:240px; }
.cmds  { position: absolute; top:450px; left:920px }
#myFFTline { z-index: 4 }
#myFFTgrid { z-index: 3 }
#myFFTdata { z-index: 2 }
#myFFTbkground { z-index: 1 }
.layerMF { position: absolute; left:920px }
.nolayerMF { position: absolute; top:240px; left:920px }
#myFFTlineMF { z-index: 4 }
#myFFTgridMF { z-index: 3 }
#myFFTdataMF { z-index: 2 }
#myFFTbkgroundMF { z-index: 1 }
.layerHF { position: absolute; top:450px; }
.scaleHF { position: absolute; top:650px; }
.nolayerHF { position: absolute; top:690px; }
#myFFTlineHF { z-index: 4 }
#myFFTgridHF { z-index: 3 }
#myFFTdataHF { z-index: 2 }
#myFFTbkgroundHF { z-index: 1 }
</style>
<BODY bgcolor="Black" onload="setBackground();caio();">
<TABLE><TR><TD>
<canvas class="layer" id="myFFTgrid" width="896" height="200" style="border:1px
solid #000000;"></canvas>
<canvas class="layer" id="myFFTbkground" width="896" height="200"
style="border:1px solid #000000;"></canvas>
```

```
<canvas class="layer" id="myFFTdata" width="896" height="200" style="border:1px
solid #000000;"></canvas>
<canvas class="layer" id="myFFTline" width="896" height="200" style="border:1px
solid #000000;"></canvas>
<canvas class="scale" id="scale" width="896" height="50" style="border:1px solid
#000000;"></canvas>
<canvas class="nolayer" id="myWfall" width="896" height="200" style="border:1px
solid #000000;"></canvas>
</TD><TD>
<canvas class="layerMF" id="myFFTgridMF" width="512" height="200"
style="border:1px solid #000000;"></canvas>
<canvas class="layerMF" id="myFFTbkgroundMF" width="512" height="200"
style="border:1px solid #000000;">
</canvas>
<canvas class="layerMF" id="myFFTdataMF" width="512" height="200"
style="border:1px solid #000000;"></canvas>
<canvas class="layerMF" id="myFFTlineMF" width="512" height="200"
style="border:1px solid #000000;"></canvas>
<canvas class="nolayerMF" id="myWfallMF" width="512" height="200"
style="border:1px solid #000000;"></canvas>
</TD></TR><TR></TD>
<canvas class="layerHF" id="myFFTgridHF" width="896" height="200"
style="border:1px solid #000000;"></canvas>
<canvas class="layerHF" id="myFFTbkgroundHF" width="896" height="200"
style="border:1px solid #000000;">
</canvas>
<canvas class="layerHF" id="myFFTdataHF" width="896" height="200"
style="border:1px solid #000000;"></canvas>
<canvas class="layerHF" id="myFFTlineHF" width="896" height="200"
style="border:1px solid #000000;"></canvas>
<canvas class="scaleHF" id="scaleHF" width="896" height="50" style="border:1px
solid #000000;"></canvas>
<canvas class="nolayerHF" id="myWfallHF" width="896" height="200"
style="border:1px solid #000000;"></canvas>
<TD></TD>
</TD></TR></TABLE>
<span class="cmds">
<form action="javascript:set();">
<TABLE><TR>
<TD><input type="button" name="LSB" id="LSB" VALUE="LSB"
onClick="setlsb();this.form.submit();"/></TD>
<TD><input type="button" name="USB" id="USB" VALUE="USB"
onClick="setusb();this.form.submit();"/></TD>
<TD><input type="button" name="CW" id="CW" VALUE="CW"
onClick="setcw();this.form.submit();"/></TD>
<TD><input type="button" name="NARROW" id="NARROW" VALUE="NARROW"
onClick="setbwnarrow();this.form.submit();"/>
</TD>
<TD><input type="button" name="MEDIUM" id="LP" VALUE="MEDIUM"
onClick="setbwmedium();this.form.submit();"/></TD>
<TD><input type="button" name="WIDE" id="WIDE" VALUE="WIDE"
onClick="setbwwide();this.form.submit();"/>
</TD></TR></TABLE>
</form>
</span>
</BODY>
```

Listing 33: Definition of the graphic elements in the page using HTML

6.6.3.3 - User Commands

As seen there are many possible interactions for the operator, who can click the buttons to change modulation mode or bandwidth, or can click the spectrograms to change the frequency, or use the mouse wheel to also change the frequency.
All these operations are done defining handlers that intercept the events: the definitions for the buttons can be seen in Listing 34 showing the HTML that outlines the graphic elements of the page.
All the handlers change some variables and, then, craft a message to the Web Worker to send a WebSocket frame with the setup command to the client receiver.

```
function setlsb() {
	mode='LSB';
	bfoFreq=+1850;
}
function setusb() {
	mode='USB';
	bfoFreq=-1850;
}
function setcw() {
	mode='CW';
	bfoFreq=-700;
}
function setbwnarrow() {
	bw='NARROW';
}
function setbwmedium() {
	bw='MEDIUM';
}
function setbwwide() {
	bw='WIDE';
}
function set( ) {
	if( myWorker != null ) {
		myWorker.postMessage( "MODE="+mode+"\nBW="+bw+"\nFC="+fc );
	}
}
```

Listing 34: The functions fired by the buttons

There are some particular events used to catch the action performed by the mouse clicking on the spectrograms or by the movement of the mouse wheel when the pointer is on a spectrogram, as can be seen in Listing 35.
On a click the x position is extracted by the event passed as parameter to the handle and the new frequency is computed.
On a wheel movement the central frequency is increased or decreased of a certain frequency interval, proportional to the wheel rotation and depending on which spectrogram the pointer is.

```
elem=document.getElementById('myFFTline');
ref=elem.offsetLeft-1+elem.width/2;
elem.addEventListener('click', function( event ) {
	prop=96000*Math.pow( 2, readrate )/1024;
	fc=1000*parseInt( (event.pageX-ref)*prop/1000 );
	set();
}, false );
elem.addEventListener( 'wheel', function( event ) {
	fc-=1000*event.deltaY/3; set();
}, false );
elem=document.getElementById( 'myFFTlineHF' );
elem.addEventListener( 'click', function( event ) {
	prop=96000/1024;
	fc=readfc+parseInt( (event.pageX-ref)*prop );
	set();
}, false );
elem.addEventListener( 'wheel', function( event ) {
	fc-=100*event.deltaY/3; set();
}, false );
elem=document.getElementById( 'myFFTlineMF' );
ref1=elem.offsetLeft+elem.width/2;
elem.addEventListener( 'click', function( event ) {
	prop=12000/1024;
	fc=readfc+parseInt( (event.pageX-ref1)*prop );
	set();
}, false );
elem.addEventListener( 'wheel', function( event ) {
	fc-=10*event.deltaY/3; set();
}, false );
```

Listing 35: Mouse handlers

(red: click event, blue: mouse wheel events)

6.6.3.4 - Spectrograms

The code for drawing the spectrograms is the same as the code used for the spectrogram in the control browser: the main difference is the fact that there are three distinct spectrograms, but this only means that there are three different drawing phases that follow the same procedure.

6.6.3.5 - Audio Management

This was the most challenging part of the work, mainly because of the limit imposed by the browser. We start by the fact that the only simple way to exchange audio between a server and a browser is to use a WebSocket that, in turn, uses TCP/IP. This may not seem a very good idea from the perspective of the delays, but it avoids the complex work needed to use WebRTC. The fact that WebSockets use TCP and not UDP (reportedly, for security reasons) allows the communication to pass though the network, crossing network devices (routers, firewalls, …) without the troubles experienced by UDP communications.
Another important limitation is the fact that the browser follows the audio card setup without any possibility to change it (already, for security reasons).
The final limitation is that, to be able to use a thread different from the main loop, it is not possible to use Web Worker (this part of audio context is planned, standardized, but not available) but audioScriptProcessor must be used. This component is normally used as part of an audio processing flow, allowing the modification of the data (for example, adding white

noise or distorting the sound). Here it is used as a source and, by doing so, the callback is called when the audio destination needs more data. This approach has been demonstrated effective, but, it also imposes two other limitations: the first is that the samples are elaborated at the (not previously known) audio card sample rate, and the second is that the buffer must be of a number of samples that can be set, but must be a power of two.
These limitations impose that a sample rate conversion must take place and this conversion must be done within the browser itself, and, also, there must be a buffering to accommodate the difference of buffers.
Fortunately, there is a component that can take care of the data rate conversion, here called "audio rendering". The solution is to create an audioBuffer and use an offlineAudioContext to perform the conversion. With the converted buffers an array of object is created and extended, using a callback fired at the end of the rendering process. The good news is that offlineAudioContext performs the operation in a separate, async thread, avoiding setting locks on the main thread that serve the queue where the sendMessage() function from the WebSocket puts the data. All these operations can be seen in Listing 36.
At this point the normal audioContext, or, better, its callback of the audioScriptProcessor, can use the elements from the array of objects to fill its output buffer, as can be seen in Listing 37.
Doing this work it is a good practice to buffer some data to prevent jerky audio. This could be done in a very simple and elegant way simply enumerating the elements and without opening the first few so, when the callback is executed some audio data is available. This also allows the usage of segments of one or more of the objects to fill the audioScriptProcessor output buffer, that has, in general, a different size from the audio data objects.

```
if( first[0]==4 ) { // audio data
	var sndarr = new Float32Array( e.data );
	var mysnd = sndarr.slice( 1 ); // first 4 bytes are part of an header
	if( !(soundrec%10) ) { // we accumulate 480 samples of data into a buffer
		universe = new ( window.OfflineAudioContext ||
			window.webkitOfflineAudioContext )( 1,
				samples,audioCtx.sampleRate );
		buff = universe.createBuffer(1,480,24000);
		soundrec=0;
	}
	buf = buff.getChannelData( 0 );
	for( j=0; j<48; j++ ) { // fille the buffer
		buf[soundrec*48+j]=sndarr[j+1];
	}
	soundrec++;
	// when the buffer is filled we elaborate it
	if( !(soundrec%10) ) {
		source=universe.createBufferSource();
		source.buffer=buff;
		source.connect( universe.destination );
		// the operations take place asynchronously
		source.start( 0 );
		// when completed we extract the data
		universe.oncomplete=function( audioBuffer ) {
			soundobj[lastobj]=new Float32Array(
				audioBuffer.renderedBuffer.getChannelData( 0 ) );
			lastobj++;
			bufferlength=audioBuffer.renderedBuffer.length;
		}
		universe.startRendering();
	}
}
```

Listing 36: Receiving audio data from WebSocket and rendering them

Enumerating segments can be seen as an idea not really proper, because of the overflow, but a simple computation shows that such an event can occur more or less after a month of uninterrupted operations, that is an uncommon scenery for a web application. Nevertheless this is a point that could be improved.
Once created, the objects must be deleted, but it seems that this operation is not needed because of the scope-driven garbage collection implemented in JavaScript; again, this is another point that should be investigated to improve this aspect of the application.
This implementation has been proven robust and effective, but many improvement could be done, the first being a codec implementation to reduce the bit rate that is an infamous 24kS/s*32 bit/sample=768kb/s of data only; with the headers it could exceed 1Mbit/S, that can sound as an enormity, but it has to be kept in mind that speeds of 10 and more Mbit/sec are, in many countries, normal. Despite this a codec insertion could be a good idea.

```
var audioCtx = new ( window.AudioContext || window.webkitAudioContext )();
var samples=audioCtx.sampleRate*480/24000;
source = audioCtx.createScriptProcessor( 1024, 1, 1 );
source.connect(audioCtx.destination);
source.onaudioprocess = function( event ) {
	if( Math.floor(firstobj*1024/bufferlength) < lastobj+2 ) {
		outData=event.outputBuffer.getChannelData( 0 );
		sect=Math.floor( firstobj*1024/bufferlength );
		start=( firstobj*1024 )%bufferlength;
		for( j=sect,k=start,l=0; l<1024; ) {
			outData[l]=soundobj[j][k]*1000;
			k++;
			l++;
			if( k>=bufferlength ) {
				k=0;
				j++;
			}
		}
		firstobj++;
	}
};
```

Listing 37: The context that fill the rendered data into the buffer when requested by audio card

7 - Conclusions

The development of this project has allowed us to deepen a range of topics:

- USB asynchronous transfer to allow low latency
- shared memories as inter process communication method
- use of conditional variables on shared memories as a method of synchronizing processes
- development of WebSocket communications
- development of various digital signal processing related to radio environment: oscillators, mixers, filters (both FIR and IIR), Fast Fourier Transform
- development of graphic routines in JavaScript into browser using canvas and graphic context
- management of audio data in browser using audio context.

For all these topics some working artifacts were developed, so we can say that the outcome is not only knowledge but also some working pieces of code, that can be re-used to complete other projects.

7.1 - State of the Project and Future Options

The work done allows the building of a truly working solution that can be used to remotize a sampler, receiving signals from a remote site.
The developed infrastructure is useful to make test with various filters setup and to test new configurations, being a complete and robust SDR infrastructure.
Nevertheless there are areas where the project could be improved:

- I/Q sample network transmission: it could be tested if it is better to send 24 bit integer values from remote embedded card to the server and 8 or 16 bit integer values for audio to the browser
- FFT: it could be verified if it is possible to use half byte for the value of a single bin, to reduce the network load
- it could be interesting to explore different coefficients or test a CIC implementation for the first filters
- it could be interesting to try filters based on FFT to better shape the audio filter
- it could be interesting to try to insert some codecs to compress the audio flow to the browser
- it could be interesting to implement a strength meter
- it could be interesting to implement the demodulation of other signals as AM or FM
- it could be interesting to implement the demodulation of some code as Morse code or some digital communications as BPSK, GMSK, 4QAM, JT65, …
- it could be useful to implement some AGC (automatic gain control)
- it could be interesting to implement a Volume control
- it could be interesting to make the companion application that sends data to a SDR transmitter like FDM DUO.

Also it could be interesting to make some measurements of time and CPU load to verify how much every solution impacts on the hardware.
At the moment the tests show that the full server + client implementation uses less than 15% of a 3GHz I7 Intel processor, letting the needed space to other clients to connect and use the same sampler.

8 - Bibliography

State of the art analysis

GNU Radio - http://gnuradio.org/redmine/projects/gnuradio/wiki
LinRad - http://www.sm5bsz.com/linuxdsp/linrad.htm
WinRad: an Italian SDR built on Windows - *http://www.sdradio.eu/weaksignals/winrad/*
rtl-sdr Turning USD 20 Realtek DVB-T receiver into a SDR - https://sdr.osmocom.org/trac/raw-attachment/wiki/rtl-sdr/rtl-sdr.2.pdf
rtl-sdr and GNU Radio w/Realtek RTL2832U, E4000 and R820T – http://*superkuh.com/rtlsdr.html*

Foreword

The EMI and RFI - MARK DEMEULENEERE – ON4WW - *http://www.on4ww.be/emi-rfi.html*

User Experience

Software Defined Radio: Architectures, Systems and Functions By Markus Dillinger, Kambiz Madani, Nancy Alonistioti - John Wiley & Sons - August 5, 2005
La caffettiera del masochista: psicopatologia degli oggetti quotidiani - Donald A. Norman Giunti Editore, 1997

Protocols between embedded and server

Perseus on Internet – Andrea Montefusco – IW0HDV - *http://www.montefusco.com/perseuscs/*
YAESU FT-450 - CAT OPERATION REFERENCE BOOK Reference - *http://www.manualslib.com/manual/381754/Yaesu-Ft-450-Cat-Operation-Reference-Book.html*

Asynchronous USB reading

Asynchronous usb reading example - *https://github.com/Mathias-L/STM32F4-libusb-example/blob/master/async.c*

Websocket

libwebsockets - HTML5 Websocket server library in C - *https://warmcat.com/libwebsockets/2010/11/01/libwebsockets-html5-websocket-server-library-in-c.html*
Writing WebSocket servers - *https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers*

Web Worker and WebSocket

Using Web Workers - *https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers*
HTML5 Web Workers - *http://www.w3schools.com/html/html5_webworkers.asp*

Canvas

Canvas tutorial - *https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial*

Oscillators

ARM Radio -Alberto Di Bene – I2PHD - *http://www.sdradio.eu/weaksignals/code/ARM_Radio.pdf*

Filters

Tfilter – free online filter design - *http://t-filter.engineerjs.com/*
Engineerjs – free collaborative engineering toolkit - *http://engineerjs.com/*

Audio management

Websockets streaming audio - *https://www.npmjs.com/package/websockets-streaming-audio*

A - Appendix 1: GNU Radio

A.1 - GNU Radio Installation

Usually, both on Debian and Ubuntu equipped PCs, the installation of GNU Radio is not more complex than *sudo apt-get install gnuradio gr_osmosdr*, but, sometimes, to use the latest developments, we face a more complex work, first to install the required packages, then to extract the GNU Radio core source from git repositories and, finally, to compile and install the code.
In Listing 38 it can be seen the complete procedure used to install all the required packages, the modules involved can change time to time, but the operations are more or less these.

```
# Install the prerequisite packages
apt-get install cmake libbost1.54.all-dev swig doxygen wx-common libfftw3-dev
libcppunit-dev liborc-0.4-dev python-sphinx libcomedi-dev qt4-default qt4-dev-
tools libqwt-dev libgsl0-dev libsdl1.2-dev python-wxtools python-opengl
# Download, compile and install Universal Software Radio Project (USRP) Hardware
Driver
cd /opt ; git clone git://githb.com/EttusResearch/uhd.git
cd /opt/uhd/host/
cmake -DPYTHON_EXECUTABLE=/usr/bin/python2.7
-DPYTHON_INCLUDE_DIR=sr/include/python2.7 -DPYTHON_LIBRARY=/usr/lib/x86_64-linux-
gnu-libpython2.7.so.1.0 /opt/uhd/hosts
make all test install ; ldconfig
# download gnuradio from http://gnuradio.org/releases/gnuradio/
cd /opt; wget http://gnuradio.org/releases/gnuradio/gnuradio-3.7.5.1.tar.gz
tar xvzf gnuradio-3.7.5.1.tar.gz ; cd gnuradio
# build and install gnuradio
mkdir build ; cd build
cmake -DPYTHON_EXECUTABLE=/usr/bin/python2.7
-DPYTHON_INCLUDE_DIR=sr/include/python2.7 -DPYTHON_LIBRARY=/usr/lib/x86_64-linux-
gnu-libpython2.7.so.1.0 ../
make all test install; ldconfig
# install hardware related packages
apt-get install libbladerf-dev rtl-sdr librtlsdr-dev libosmosdr-dev libhackrf-dev
# install I/Q balancing software
cd /opt ; git clone git://git.osmocom.org/gr-iqbal.git
cd gr-iqbal
git submodule init
git submodule update
mkdir bild ; cd build
cmake -DPYTHON_EXECUTABLE=/usr/bin/python2.7
-DPYTHON_INCLUDE_DIR=sr/include/python2.7 -DPYTHON_LIBRARY=/usr/lib/x86_64-linux-
gnu-libpython2.7.so.1.0 ../
make all install; ldconfig
#install osmocom blocks for gnuradio
cd /opt ; git clone git://git.osmocom.org/gr-osmosdr
cd gr-osmosdr
mkdir bild ; cd build
cmake -DPYTHON_EXECUTABLE=/usr/bin/python2.7
-DPYTHON_INCLUDE_DIR=sr/include/python2.7 -DPYTHON_LIBRARY=/usr/lib/x86_64-linux-
gnu-libpython2.7.so.1.0  ../
make all install; ldconfig
```

Listing 38: A typical shell to install GNU Radio along with hw support software

A.2 - GNU Radio Architecture

A.2.1 - How a Flowgraph Is Launched

The gnuradio_companion GUI creates a python script called *top_block.py*. At its very end there is a block that takes care of the operations startup, as can be seen in the red lines in Listing 39.

```
if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
    tb = top_block()
    tb.Start(True)
    tb.Wait()
```

Listing 39: Final part of Python generated by gnuradio-companion for WBFM receiver

The statement tb = top_block() generates a new instance of the top_block class and the execution of the initialization that follows the statement def __init__(self): this can be seen at the top of Listing A.3. This part of the code takes care of the generation of the objects and the data flow between them and will be covered in one of the next sections of this appendix.
The statement tb.Start(True) launches the Start() method of top_block class. The top_block class is a subclass of the grc_wxgui.top_block_gui python class defined in something similar to */usr/local/lib/python2.7/dist-packages/grc_gnuradio/wxgui* where */usr/local/lib/python2.7/dist-packages* is the path for python packages and *grc_gnuradio/wxgui* is referenced by the import statement at the beginning of the code (see following Listing 40).

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Top Block
# Generated: Wed Mar  4 13:04:24 2015
##################################################

from gnuradio import analog
from gnuradio import audio
from gnuradio import eng_notation
from gnuradio import filter
from gnuradio import gr
from gnuradio import wxgui
from gnuradio.eng_option import eng_option
from gnuradio.fft import window
from gnuradio.filter import firdes
from gnuradio.wxgui import fftsink2
from gnuradio.wxgui import forms
from gnuradio.wxgui import waterfallsink2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import osmosdr
import time
import wx

class top_block(grc_wxgui.top_block_gui):
```

Listing 40: Initial part of Python generated by gnuradio-companion for WBFM receiver

In the Listing 41 it can be seen that the Start() method of top_block_gui class executes the start() of the same class. Since this method is not defined here, the method of the parent class gr.top_block is used; this is important to watch: grc_gnuradio.wxgui.top is a subclass created to take care only of the GUI stuff.
Also gr.top_block could be found using the previous folder hierarchy: */usr/local/lib/python2.7/dist-packages/gnuradio/gr/top_block.py* (look at the import at the beginning of Listing 41).

```
import wx
from gnuradio import gr
import panel

default_gui_size = (200, 100)

class top_block_gui(gr.top_block):
	"""gr top block with wx gui app and grid sizer."""

	def __init__(self, title='', size=default_gui_size):
		"""
		Initialize the gr top block.
		Create the wx gui elements.

		Args:
		    title: the main window title
		    size: the main window size tuple in pixels
		    icon: the file path to an icon or None
		"""
		#initialize
		gr.top_block.__init__(self)
		self._size = size
		#create gui elements
		self._app = wx.App()
		self._frame = wx.Frame(None, title=title)
		self._panel = panel.Panel(self._frame)
		self.Add = self._panel.Add
		self.GridAdd = self._panel.GridAdd
		self.GetWin = self._panel.GetWin

	def SetIcon(self, *args, **kwargs): self._frame.SetIcon(*args, **kwargs)

	def Start(self, start=True, max_nouts=0):
		#set minimal window size
		self._frame.SetSizeHints(*self._size)
		#create callback for quit
		def _quit(event):
			self.stop(); self.wait()
			self._frame.Destroy()
		#setup app
		self._frame.Bind(wx.EVT_CLOSE, _quit)
		self._sizer = wx.BoxSizer(wx.VERTICAL)
		self._sizer.Add(self._panel, 0, wx.EXPAND)
		self._frame.SetSizerAndFit(self._sizer)
		self._frame.SetAutoLayout(True)
		self._frame.Show(True)
		self._app.SetTopWindow(self._frame)
		#start flow graph
		if start:
			if max_nouts != 0:
				self.start(max_nouts)
			else:
				self.start()
```

Listing 41: Partial view of top_block class in the …/grc_gnuradio/wxgui/top_block_gui.py file

I

```
from runtime_swig import top_block_swig, \
    top_block_wait_unlocked, top_block_run_unlocked, \
    top_block_start_unlocked, top_block_stop_unlocked, \
    dot_graph_tb
...........................
class top_block(object):
    """
    Top-level hierarchical block representing a flow-graph.

    This is a python wrapper around the C++ implementation to allow
    python subclassing.
    """
    def __init__(self, name="top_block"):
        self._tb = top_block_swig(name)

    def __getattr__(self, name):
        if not hasattr(self, "_tb"):
            raise RuntimeError("top_block: invalid state--did you forget \
                to call gr.top_block.__init__ in a derived class?")
        return getattr(self._tb, name)

    def start(self, max_noutput_items=10000000):
        top_block_start_unlocked(self._tb, max_noutput_items)
```

Listing 42: Partial view of top_block class in the …/gnuradio/gr/top_block.py file

In Listing 42 the start() method calls the top_block_start_unlocked() method of the top_block_swig class. This class is written in C++. This method cold be accessed through the top_block_swig class that is generated automatically by the C++ compiling process into the folder *runtime_swig* that is a sub folder of *gnuradio/gr* where *top_block.py* of Listing 42 lays, because of the import of the first line in Listing 42.

```
def top_block_start_unlocked(*args, **kwargs):
  """top_block_start_unlocked(top_block_sptr r, int max_noutput_items)"""
  return _runtime_swig.top_block_start_unlocked(*args, **kwargs)
```

Listing 43: Partial view of top_block class in the …/gr/runtime_swig/runtime_swig.py file

In Listing 43 the term _runtime_swig refers to the dynamic library called *_runtime_swig.so* that is in the same folder of the *runtime_swig.py* file.
To understand how top_block_start_unlocked() works, yo should identify the folder which contains the C++ source code. This is in the sub folder *gnuradio-runtime* of the gnuradio source tree. To find the definition of the function you should read the file *swig/top_block.i* that defines the mapping of the C++ functions.

```
void top_block_start_unlocked(gr::top_block_sptr r, int max_noutput_items)
      throw (std::runtime_error)
{
    GR_PYTHON_BLOCKING_CODE
    (
        r->start(max_noutput_items);
    )
}
```

Listing 44: Partial view of top_block class in the .../gnuradio-runtime/swig/top_block.i file

In Listing 44 it can be seen that the function top_block_start_unlocked() calls the function start of the class top_block_swig passed as its first parameter. In Listing 45, relative to the same file, but, capturing some lines above it, it can be seen that the class top_block_swig is renamed as make_top_block() to link it to a class with the same name of the parameter passed to the constructor ("top_block" as it can be seen in Listing 42).

```
%rename(top_block_swig) make_top_block;
```

Listing 45: Rename the top_block_swig class in file .../gnuradio-runtime/swig/top_block.i

In Listing 45 it could be seen that the make_top_block C++ name is rewritten as top_block_swig in C++ using the SWIG directive %rename.
The source code for top_block_swig can be seen in gr-sources/gnuradio-runtime/lib/top_block.cc, that is reported in Listing 46.
It creates a new object of type top_block that, in turn, creates a new object of type top_block_impl and executes its start() method.

```
namespace gr {
  top_block_sptr
  make_top_block(const std::string &name)
  {
    return gnuradio::get_initial_sptr
      (new top_block(name));
  }

  top_block::top_block(const std::string &name)
    : hier_block2(name,
                  io_signature::make(0,0,0),
                  io_signature::make(0,0,0))
  {
    d_impl = new top_block_impl(this);
  }

  top_block::~top_block()
  {
    stop();
    wait();

    delete d_impl;
  }

  void
  top_block::start(int max_noutput_items)
  {
    d_impl->start(max_noutput_items);

    if(prefs::singleton()->get_bool("ControlPort", "on", false)) {
      setup_rpc();
    }
  }
```

Listing 46: Partial view of top_block class in the .../gnuradio-runtime/lib/top_block.cc file

top_block_impl is coded into files gr-sources/gnuradio-runtime/lib/top_block_impl.h and gr-sources/gnuradio-runtime/lib/top_block_impl.cc; in Listing 47 file the function start(), which selects a scheduler, can be seen.

```
void
top_block_impl::start(int max_noutput_items)
{
  gr::thread::scoped_lock l(d_mutex);

  d_max_noutput_items = max_noutput_items;

  if(d_state != IDLE)
    throw std::runtime_error("top_block::start: top block already running \
                             or wait() not called after previous stop()");

  if(d_lock_count > 0)
    throw std::runtime_error("top_block::start: can't start with flow \
                             graph locked");

  // Create new flat flow graph by flattening hierarchy
  d_ffg = d_owner->flatten();

  // Validate new simple flow graph and wire it up
  d_ffg->validate();
  d_ffg->setup_connections();

  // Only export perf. counters if ControlPort config param is
  // enabled and if the PerfCounter option 'export' is turned on.
  prefs *p = prefs::singleton();
  if(p->get_bool("ControlPort", "on", false) &&
               p->get_bool("PerfCounters", "export", false ) )
    d_ffg->enable_pc_rpc();

  d_scheduler = make_scheduler(d_ffg, d_max_noutput_items);
  d_state = RUNNING;
}
```

Listing 47: Start() method in the gr-sources/gnuradio-runtime/lib/top_block_impl.cc file

As can be seen in Listing 47, the start() method of top_block_impl class makes some activities, starting with the flattening and the validation of the flowgraph, and ending with the selection of a scheduler.

The scheduler is selected using the function make_scheduler() that is coded n the same file and can be seen in Listing 48.
The make_scheduler() function creates the scheduler by launching the function make from an element of array of structs scheduler_table[]. This array has two single elements that implement multi-thread and single-thread, the first is normally used until the second is activated using the GR_SCHEDULER environment table set to “STS”. The function normally used is scheduler_tpb(), while the other is scheduler_sts().
Since the scheduler normally used is scheduler_tpb, we can continue examining the C++ source code for this class, by looking at gr-sources/gnuradio-runtime/lib/scheduler_stpb.cc.

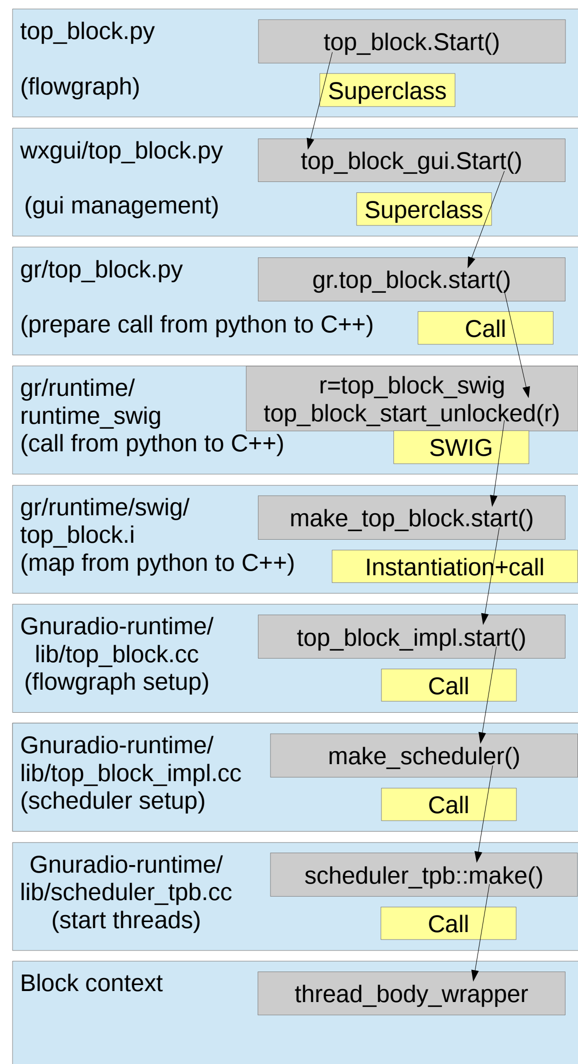
```
static struct scheduler_table {
  const char *name;
  scheduler_maker f;
} scheduler_table[] = {
  { "TPB", scheduler_tpb::make },    // first entry is default
  { "STS", scheduler_sts::make }
};

static scheduler_sptr
make_scheduler(flat_flowgraph_sptr ffg, int max_noutput_items)
{
  static scheduler_maker factory = 0;

  if(factory == 0) {
    char *v = getenv("GR_SCHEDULER");
    if(!v)
      factory = scheduler_table[0].f; // use default
    else {
      for(size_t i = 0;
        i < sizeof(scheduler_table)/sizeof(scheduler_table[0]);
        i++) {
        if(strcmp(v, scheduler_table[i].name) == 0) {
          factory = scheduler_table[i].f;
          break;
        }
      }
      if(factory == 0) {
        std::cerr <<
        "warning: Invalid GR_SCHEDULER environment variable value \"" <<
        v << "\".  Using \"" << scheduler_table[0].name << "\"\n";
        factory = scheduler_table[0].f;
      }
    }
  }
  return factory(ffg, max_noutput_items);
}
```

Listing 48: The make_scheduler() method in .../gnuradio-runtime/lib/top_block_impl.cc

In Listing 49 the make() method forces the creation of a new instance of the scheduler and, in doing so, the work is performed by the constructor. It sorts the blocks and then, for each block, starts a thread using thread_body_wrapper() for each flowgraph block element.

```
scheduler_sptr
scheduler_tpb::make(flat_flowgraph_sptr ffg, int max_noutput_items)
{
  return scheduler_sptr(new scheduler_tpb(ffg, max_noutput_items));
}

scheduler_tpb::scheduler_tpb(flat_flowgraph_sptr ffg,
                             int max_noutput_items)
  : scheduler(ffg, max_noutput_items)
{
  int block_max_noutput_items;

  // Get a topologically sorted vector of all the blocks in use.
  // Being topologically sorted probably isn't going to matter, but
  // there's a non-zero chance it might help...

  basic_block_vector_t used_blocks = ffg->calc_used_blocks();
  used_blocks = ffg->topological_sort(used_blocks);
  block_vector_t blocks = flat_flowgraph::make_block_vector(used_blocks);

  // Ensure that the done flag is clear on all blocks

  for(size_t i = 0; i < blocks.size(); i++) {
    blocks[i]->detail()->set_done(false);
  }

  // Fire off a thead for each block

  for(size_t i = 0; i < blocks.size(); i++) {
    std::stringstream name;
    name << "thread-per-block[" << i << "]: " << blocks[i];

    // If set, use internal value instead of global value
    if(blocks[i]->is_set_max_noutput_items()) {
      block_max_noutput_items = blocks[i]->max_noutput_items();
    }
    else {
      block_max_noutput_items = max_noutput_items;
    }

    d_threads.create_thread(
          gr::thread::thread_body_wrapper<tpb_container>
          (tpb_container(blocks[i], block_max_noutput_items),
           name.str()));
  }
}
```

Listing 49: Methods and constructor in the .../gnuradio-runtime/lib/scheduler_tpb.cc file



Picture 38: Starting the flowgraph

Summarizing, as can be seen in Picture 38, this is how the program is started:

- The flowgraph calls a superclass that manages the GUI
- This class calls a superclass that manages the operations
- This class loads the C++ module
- This module starts to deploy and link blocks in the flowgraph
- Then selects a scheduler
- The scheduler creates a thread for each block using a wrapper

The operations seem quite complex and confusing but this is the best to do: separate the core operations performed by every 'layer':

- Local configuration, in top_block.py
- GUI setup in wgui/top_block.py
- C++ module mapping and loading in top_block, runtime_swig
- Flowgraph preparation in top_block.cc
- Scheduler management in top_block_impl.cc
- Blocks operations in various blocks

A.1.3 - How a Flowgraph Works

The previous section finished with a call to a create_thread() function (see Listing 49). This function uses a thread_body_wrapper() to set-up signals (see Listing 50) and then call the object passed as an argument.

```
template <class F>
class thread_body_wrapper
{
private:
  F d_f;
  std::string d_name;

public:
  explicit thread_body_wrapper(F f, const std::string &name="")
    : d_f(f), d_name(name) {}

  void operator()()
  {
    mask_signals();

    try {
      d_f();
    }
    catch(boost::thread_interrupted const &)
      {
      }
    catch(std::exception const &e)
      {
        std::cerr << "thread[" << d_name << "]: "
                  << e.what() << std::endl;
      }
    catch(...)
      {
        std::cerr << "thread[" << d_name << "]: "
                  << "caught unrecognized exception\n";
      }
  }
};
```

Listing 50: tpb_container in the /usr/local/include/gnuradio/thread/thread_body_wrapper.h file

The object passed to thread_body_wrapper() as an argument is a tpb_container that is defined in the same source file *gr-sources/gnuradio-runtime/lib/scheduler_tpb.cc* (as can be seen in Listing 51).

```
class tpb_container
{
  block_sptr d_block;
  int d_max_noutput_items;

public:
  tpb_container(block_sptr block, int max_noutput_items)
    : d_block(block), d_max_noutput_items(max_noutput_items) {}

  void operator()()
  {
    tpb_thread_body body(d_block, d_max_noutput_items);
  }
};
```

Listing 51: tpb_container in the gr-sources/gnuradio-runtime/lib/scheduler_tpb.cc file

In Listing 51 we can see how the tpb_container class overloads the () method so the constructor of the tpb_container class creates an object of type tpb_thread_body whose constructor performs the operations in the thread;

```
class GR_RUNTIME_API tpb_thread_body
{
  block_executor d_exec;

public:
  tpb_thread_body(block_sptr block, int max_noutput_items=100000);
  ~tpb_thread_body();
};
```

Listing 52: Definition in the gr-sources/gnuradio-runtime/lib/tpb_thread_body.h file

tpb_thread_body has a resource d_exec that is an object of type block_executor (as can be seen in the *gr-sources/gnuradio-runtime/lib/tpb_thread_body.h* file listed in Listing 52) that is used to drive the operations for the block.

```
  tpb_thread_body::tpb_thread_body(block_sptr block, int max_noutput_items)
    : d_exec(block, max_noutput_items)
  {
    //std::cerr << "tpb_thread_body: " << block << std::endl;
.............
    thread::set_thread_name(pthread_self(),
      boost::str(boost::format("%s%d") % block->name() % block->unique_id()));
..............
    block_detail *d = block->detail().get();
    block_executor::state s;
    pmt::pmt_t msg;

    d->threaded = true;
    d->thread = gr::thread::get_current_thread_id();

    prefs *p = prefs::singleton();
    size_t max_nmsgs = static_cast<size_t>(p->get_long("DEFAULT",
      "max_messages", 100));
.......
    // Set thread affinity if it was set before fg was started.
    if(block->processor_affinity().size() > 0) {
      gr::thread::thread_bind_to_processor(d->thread,
      block->processor_affinity());
    }
    // Set thread priority if it was set before fg was started
    if(block->thread_priority() > 0) {
      gr::thread::set_thread_priority(d->thread, block->thread_priority());
    }
    // make sure our block isnt finished
    block->clear_finished();

    while(1) {
      tpb_loop_top:
      boost::this_thread::interruption_point();

      // handle any queued up messages
      BOOST_FOREACH(basic_block::msg_queue_map_t::value_type &i,
        block->msg_queue) {
        // Check if we have a message handler attached before getting
        // any messages. This is mostly a protection for the unknown
        // startup sequence of the threads.
        if(block->has_msg_handler(i.first)) {
          while((msg = block->delete_head_nowait(i.first))) {
            block->dispatch_msg(i.first,msg);
          }
        }
        else {
          // If we don't have a handler but are building up messages,
          // prune the queue from the front to keep memory in check.
          if(block->nmsgs(i.first) > max_nmsgs){
            GR_LOG_WARN(LOG,"asynchronous message buffer overflowing,
              dropping message");
            msg = block->delete_head_nowait(i.first);
          }
        }
      }
      d->d_tpb.clear_changed();
      // run one iteration if we are a connected stream block
```

```
      if(d->noutputs() >0 || d->ninputs()>0){
        s = d_exec.run_one_iteration();
      }
      else {
        s = block_executor::BLKD_IN;
      }
      // if msg ports think we are done, we are done
............
   switch(s){
      case block_executor::READY:           // Tell neighbors we made progress.
        d->d_tpb.notify_neighbors(d);
        break;

      case block_executor::READY_NO_OUTPUT:  // Notify upstream only
        d->d_tpb.notify_upstream(d);
        break;

      case block_executor::DONE:             // Game over.
        block->notify_msg_neighbors();
        d->d_tpb.notify_neighbors(d);
        return;

      case block_executor::BLKD_IN:          // Wait for input.
      {
        gr::thread::scoped_lock guard(d->d_tpb.mutex);
        while(!d->d_tpb.input_changed) {

          // wait for input or message
          while(!d->d_tpb.input_changed && block->empty_handled_p()){
            boost::system_time const timeout=
              boost::get_system_time()+ boost::posix_time::milliseconds(250);
            if(!d->d_tpb.input_cond.timed_wait(guard, timeout)){
                goto tpb_loop_top;
                // timeout occured (perform sanity checks up top)
                }
            }
...........
       case block_executor::BLKD_OUT:        // Wait for output buffer space.
       {
        gr::thread::scoped_lock guard(d->d_tpb.mutex);
        while(!d->d_tpb.output_changed) {
          // wait for output room or message
          while(!d->d_tpb.output_changed && block->empty_handled_p())
            d->d_tpb.output_cond.wait(guard);

...........
      default:
        throw std::runtime_error("possible memory corruption in scheduler");
      }
    }
```

Listing 53: Constructor in the gr-sources/gnuradio-runtime/lib/tpb_thread_body.cc file

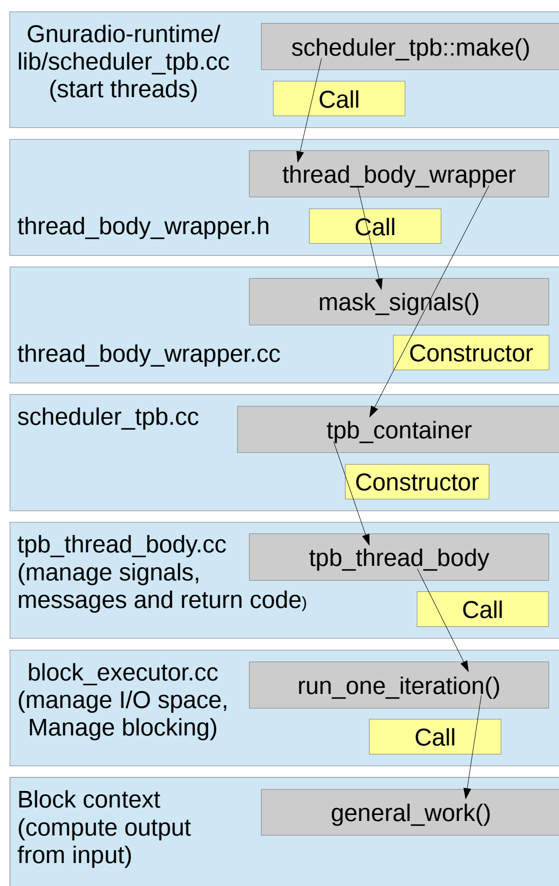
The constructor of tpb_thread_body contains an infinite loop. This loop processes signals using interruption_point(), then it handle any queued messages, and then executes the function d_exec.run_one_iteration() that performs one round of operation as requested by the single block of the flowgraph. Run_one_iteration() returns a state that is used to drive the operations as we can see in Listing 53.

Run_one_iteration() runs a very long sequence of operations, but they can be schematized in few passes, that c an be seen in Listing 54:

- it verifies if it has any new input pending and if there is enough space in output **or**
- it verifies if it has any output request and if there are enough data in input **or**
- it verifies if there is any input to be processed
- on every case it stops and signals if there is a lack of input items or otput space
- then it does the general work using the general_work() function

```
block_executor::state
block_executor::run_one_iteration()
{
..........
// Do the actual work of the block
      int n = m->general_work(noutput_items, d_ninput_items,
                              d_input_items, d_output_items);
..........
}
```

Listing 54: run_one_iteration() in gr-sources/gnuradio-runtime/lib/block_executor.cc



Picture 39: Running a block

Finally, we could say that a flowgraph works as follows, for each block (see Picture 39):

- thread_body_wrapper() set-up signals;
- the tpb_container class creates an object of type tpb_thread_body;

- tpb_thread_body constructor performs the operations, managing signals, messaging, the execution of run_one_iteration() and the management of its return state;
- run_one_iteration() manages input blocks of data and runs the general_work() function

This mode of operation specializes every class to perform different things and lets the general_work() function be a pointer to the specific work of a single block.
With this scheme the implementation of a block does not care for signals, messages, flow management, but only computes output data as a result of input values.

A.1.4 - Writing and Understanding Blocks

Blocks main operations are coded into the function general_work(), or, in specialized blocks, in function work(). The most simple way to illustrate the blocks operations is to examine the tool that can be used to build a new module. This tool is called *gr_modtool* and allows the building of the complete structure of a block, both C++, SWIG, and Python scheme together with the code needed to test the block implementation. In Picture 40 it can be seen the possible operations.

```
giovanni@mir:~$ gr_modtool help
Usage:
gr_modtool <command> [options] -- Run <command> with the given options.
gr_modtool help -- Show a list of commands.
gr_modtool help <command> -- Shows the help for a given command.

List of possible commands:

Name      Aliases          Description
=====================================================================
disable   dis              Disable block (comments out CMake entries for files)
info      getinfo,inf      Return information about a given module
remove    rm,del           Remove block (delete files and remove Makefile entries)
makexml   mx               Make XML file for GRC block bindings
add       insert           Add block to the out-of-tree module.
newmod    nm,create        Create a new out-of-tree module
giovanni@mir:~$
```

Picture 40: Picture 40: gr_modtool showing possible operations

To understand how the block works it could be interesting to write a module, this is quite simple because it could be done using gr_modtools to write most of the code and then by hand, focusing only on the details. As an example we can create a module named modname; this module contains a block called blockname. In real operations these names should be replaced by more suitable names that reflect the real module and block scope. The operations are, mainly:

- create the module using gr_modtool newmod modname, this operation produces the tree of folders required to host the module's components (C++ source, headers, SWIG, python source, xml) and can be seen in Picture 41;
- create the files as cd gr-modname; gr_modtool add -t general blockname,this operation insert relevant files with reasonable default content;

```
giovanni@mir:~$ gr_modtool newmod modname
Creating out-of-tree module in ./gr-modname... Done.
Use 'gr_modtool add' to add a new block to this currently empty module.
giovanni@mir:~$ cd gr-modname; gr_modtool add -t general blockname
GNU Radio module name identified: modname
Language: C++
Block/code identifier: blockname
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n]
Add C++ QA code? [y/N]
Adding file 'lib/blockname_impl.h'...
Adding file 'lib/blockname_impl.cc'...
Adding file 'include/modname/blockname.h'...
Editing swig/modname_swig.i...
Adding file 'python/qa_blockname.py'...
Editing python/CMakeLists.txt...
Adding file 'grc/modname_blockname.xml'...
Editing grc/CmakeLists.txt...
giovanni@mir:~$
```

Picture 41: Picture 41: Making module and block structure with gr_modtool

- write the test as vi python/qa_blockname.py and insert test code, in the example the code really does nothing, simply moves the input to the output, so the test code has only to verify that the output is equal to the input, this is done by disposing an input and an expected result vector that contains the same data, for reference see Picture 42.

```
from gnuradio import gr, gr_unittest
from gnuradio import blocks
import modname_swig as modname

class qa_blockname (gr_unittest.TestCase):

    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    def test_001_modename(self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (-3, 4, -5.5, 2, 3)
        src = blocks.vector_source_f(src_data)
        sqr = modname.blockname()
        dst = blocks.vector_sink_f()
        self.tb.connect(src, sqr)
        self.tb.connect(sqr, dst)
        self.tb.run()
        result_data = dst.data()
        self.assertFloatTuplesAlmostEqual(expected_result, result_data, 6)

if __name__ == '__main__':
    gr_unittest.run(qa_blockname, "qa_blockname.xml")
```

Picture 42: Picture 42: Editing QA code. Only test_001_modename() is inserted by hand

- write the C++ code as vi lib/blockname_impl.cc and insert implementation code (for reference see Picture 43), there are some important things to do:
  - in the constructor it is important to set the signature (min,max, type both for input and output)
  - in forecast() it is important to set the number required in input to produce the output
  - in general_work() it is important to write the real signal processing (in the example there is no processing at all, only passing input to output.

```
/*
 * The private constructor
 */
blockname_impl::blockname_impl()
  : gr::block("blockname",
          gr::io_signature::make(1, 1, sizeof(float)),
          gr::io_signature::make(1, 1, sizeof(float)))
{}

/*
 * Our virtual destructor.
 */
blockname_impl::~blockname_impl()
{
}

void
blockname_impl::forecast (int noutput_items, gr_vector_int
                      &ninput_items_required)
{
    ninput_items_required[0] = noutput_items;
}

int
blockname_impl::general_work (int noutput_items,
                   gr_vector_int &ninput_items,
                   gr_vector_const_void_star &input_items,
                   gr_vector_void_star &output_items)
{
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];

    // Do <+signal processing+>
    for(int i = 0; i < noutput_items; i++) {
            out[i] = in[i];
    }

    // Tell runtime system how many input items we consumed on
    // each input stream.
    consume_each (noutput_items);

    // Tell runtime system how many output items we produced.
    return noutput_items;
}
```

Picture 43: Picture 43: Editing C++ code.

Please note that the class blockname_impl is the real implementation of the block and general_work() is where the signal processing happens; general_work() is used if the block is

of “general” type, but, if other types (like 'sync', 'decimator', 'interpolator', …) are used, the method used is no more general_work(), but work().

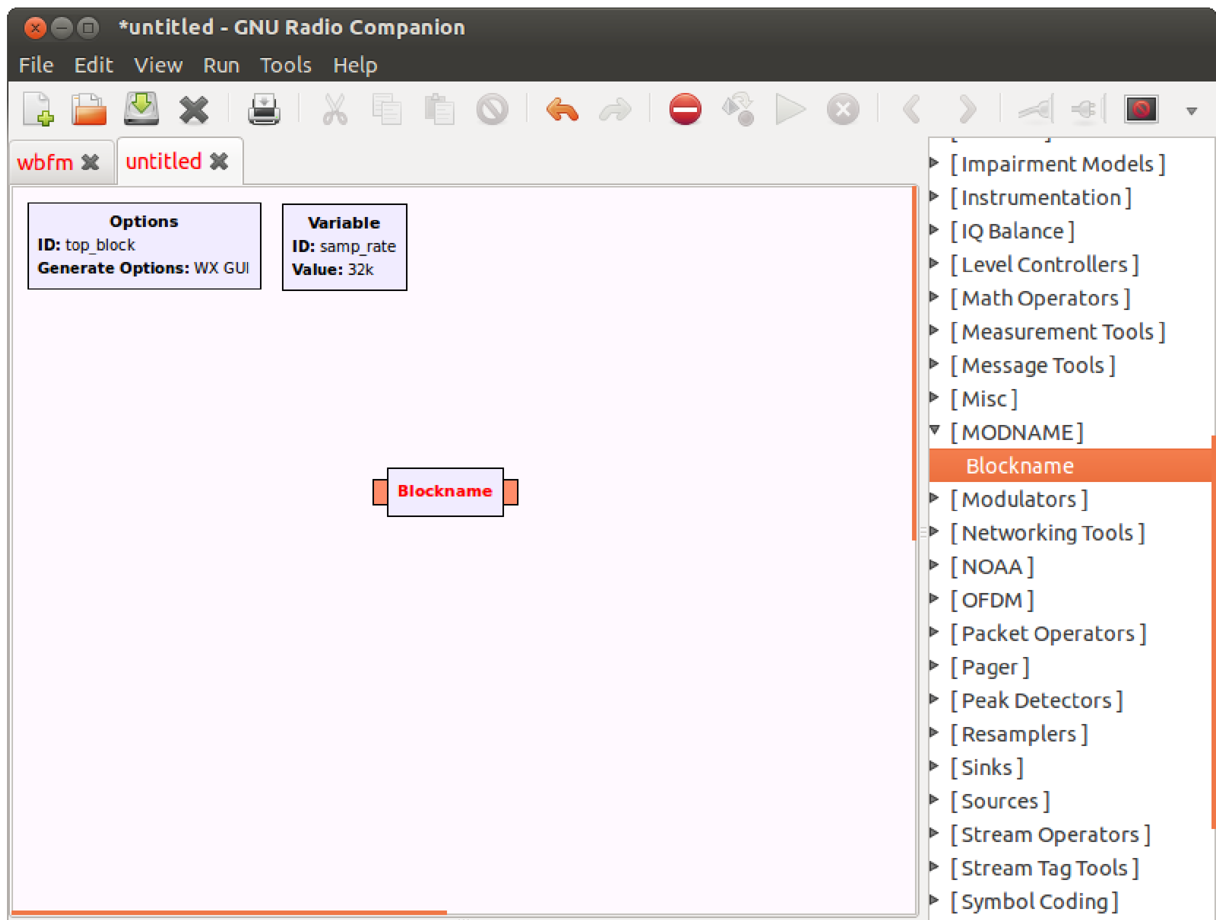
- Use cmake as mkdir build; cd build; cmake ..; make to build the program
- make the test as cd build; make test; cd ..;
- create the xml source to enable block into GNU Radio companion as gr_modtool makexml blockname; vi grc/newmod_modname.xml (for reference see Picture 44);

```
giovanni@mir:~/Desktop/tesi-master/gnu/gr-modname$ gr_modtool makexml blockname
GNU Radio module name identified: modname
Warning: This is an experimental feature. Don't expect any magic.
Searching for matching files in lib/:
Making GRC bindings for lib/blockname_impl.cc...
Overwrite existing GRC file? [y/N]
giovanni@mir:~/Desktop/tesi-master/gnu/gr-modname$ cat grc/
CMakeLists.txt         modname_blockname.xml
giovanni@mir:~/Desktop/tesi-master/gnu/gr-modname$ cat grc/modname_blockname.xml
<block>
  <name>Blockname</name>
  <key>modname_blockname</key>
  <category>MODNAME</category>
  <import>import modname</import>
  <make>modname.blockname()</make>
  <sink>
    <name>in</name>
    <type>float</type>
  </sink>
  <source>
    <name>out</name>
    <type>float</type>
  </source>
</block>
giovanni@mir:~/Desktop/tesi-master/gnu/gr-modname$
```

Picture 44: Picture 44: Generating XML code to integrate block into gnuradio_companion.

- install the module to enable its use as cd build; sudo make install; sudo ldconfig.

Having done all this work, it is possible to use the module as part of gnuradio_companion, as can be seen in Picture 45.

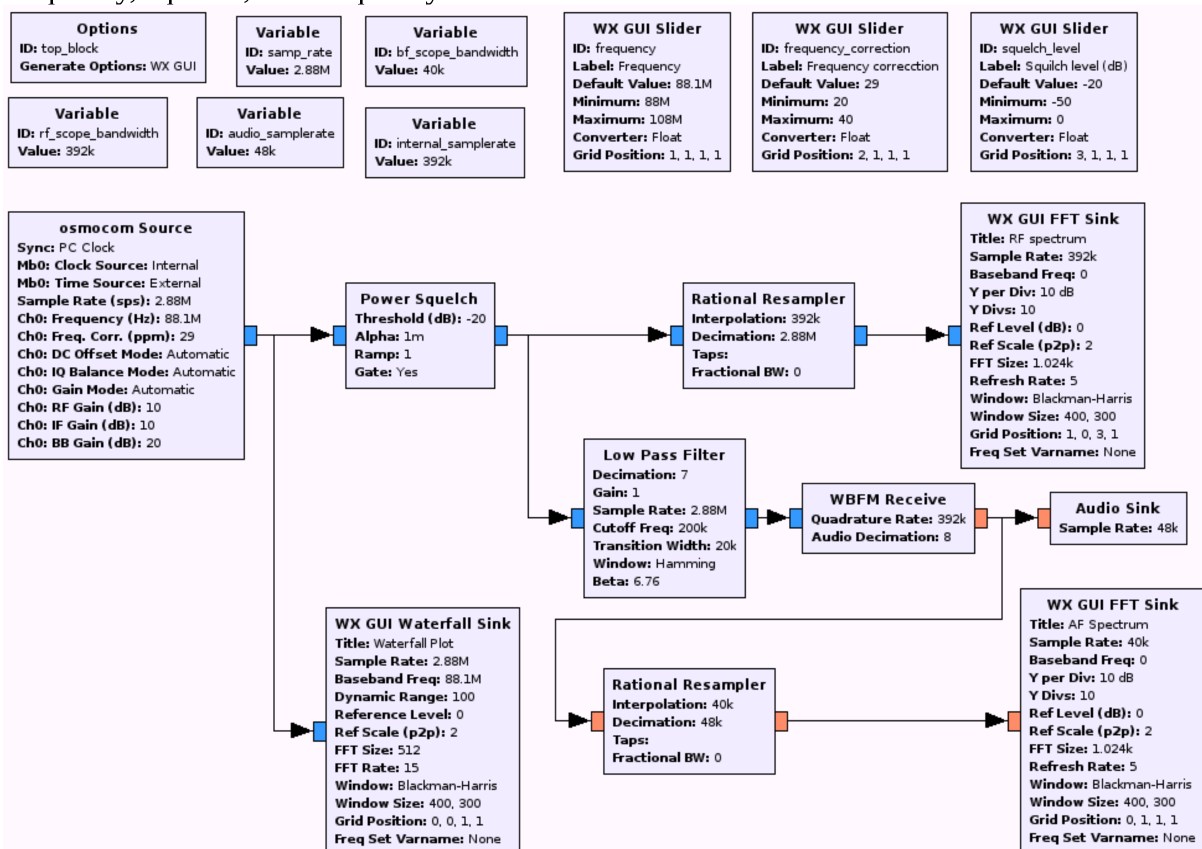


Picture 45: GNU Radio companion showing the new module and block

B - Appendix 2 – GNU Radio Realizations

B.1 - Wide Band FM Monoaural Receiver

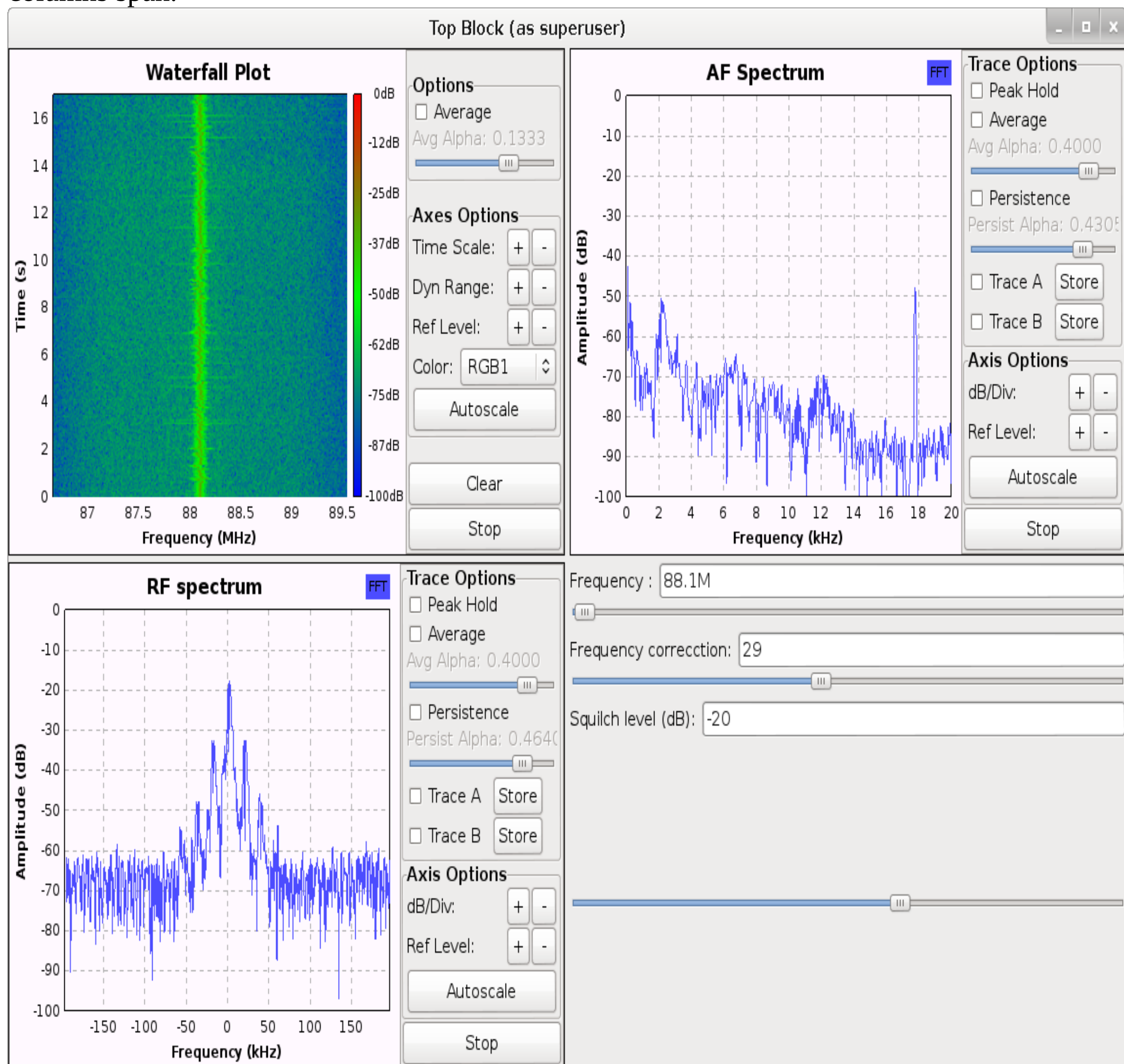
GNU Radio was used to verify the dongle capabilities as well as PC performances. The first realization is on the FM band where signals are quite loud and stable, allowing us to test our scheme.
Also, this first realization is used to understand how to use GNU Radio companion and how to configure the various dongle parameters.
We designed a “test” receiver that not only decodes audio, but also characterize the broadcast's signal, allowing us to see a large waterfall, in-channel spectrum, as well as audio spectrum (see Picture 46 for reference).
To do so we not only used Waterfall and FFT widgets, but also other GUI widgets to manage frequency, squelch, and frequency correction.



Picture 46: GNU Radio companion WBFM block diagram

There are some lessons learned by this work, primarily the fact that rational resamplers with non-integer ratio between input and output sample rate lead to delays in signal processing, This is not a problem for simple radio receivers, but must be avoided when designing receiver part of transceivers.

One other lesson is tied to the GUI widgets, that can be used to control variables simply by using the widget id as a variable in the other widgets configuration. Also important is to understand that “grid position” works as Java's “bag of cells” layout, where the four parameters refer to cardinal number of rows, cardinal number of columns, rows span, and columns span.



Picture 47: GNU Radio companion WBFM interface

The picture 47 shows the signal of a station broadcasting at 88.100 MHz. It can be seen that the signal occupies a band of approx 150kHz, and it is a stereo one (see sync signal at 18kHz on AF spectrum).

Using this simple receiver we have learned that the dongle is not shielded enough and that the stations could be received even if the antenna is not connected, so we wrapped the dongle with an aluminium foil attached to both the USB connector jacket and the outer part of the RF connector (see Picture 48). This reduced the unwanted signals of several tens of dB and allows the dongle to listen only to signals received by a connected antenna.

Picture 48: Dongle without and with shield

In the following listing it can be seen that gnuradio-companion put some different blocks of code as:

- variables,
- blocks,
- connections,
- setter and getter definitions.

Variables are a way to have a single reference to many different blocks. For instance, to have the same value for Sample rate in WX GUI FFT Sink and Interpolation in Rational Resampler, RF Scope Bandwidth is used (see blue rows in Listing 55).

Blocks are the code that declare the various components. There are GUI blocks, that are visible graphic controls, made by a label, a control, and a positioning directive (see green rows in Listing 55). There are also non GUI blocks as the Rational Resampler (in orange in Listing 55). Statements inside the blocks are used to set various aspects of the blocks (sample rate, input and output data type, …).

Connections are used to connect blocks each others. Setter and getter definitions define the operations performed by the various GUI blocks when the value varies. These operations are made by functions. The functions are referenced by the callbacks in the GUI blocks.

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Top Block
# Generated: Wed Feb 11 17:19:32 2015
##################################################

from gnuradio import analog
from gnuradio import audio
from gnuradio import eng_notation
from gnuradio import filter
from gnuradio import gr
from gnuradio import wxgui
from gnuradio.eng_option import eng_option
from gnuradio.fft import window
from gnuradio.filter import firdes
from gnuradio.wxgui import fftsink2
from gnuradio.wxgui import forms
from gnuradio.wxgui import waterfallsink2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import osmosdr
import time
import wx
```

```
class top_block(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Top Block")

        ##################################################
        # Variables
        ##################################################
        self.squelch_level = squelch_level = -20
        self.samp_rate = samp_rate = 960000*3
        self.rf_scope_bandwidth = rf_scope_bandwidth = 392000
        self.internal_samplerate = internal_samplerate = 392000
        self.frequency_correction = frequency_correction = 29
        self.frequency = frequency = 88100000
        self.bf_scope_bandwidth = bf_scope_bandwidth = 40000
        self.audio_samplerate = audio_samplerate = 48000

        ##################################################
        # Blocks
        ##################################################
        _squelch_level_sizer = wx.BoxSizer(wx.VERTICAL)
        self._squelch_level_text_box = forms.text_box(
             parent=self.GetWin(),
             sizer=_squelch_level_sizer,
             value=self.squelch_level,
             callback=self.set_squelch_level,
             label="Squilch level (dB)",
             converter=forms.float_converter(),
             proportion=0,
        )
        self._squelch_level_slider = forms.slider(
             parent=self.GetWin(),
             sizer=_squelch_level_sizer,
             value=self.squelch_level,
             callback=self.set_squelch_level,
             minimum=-50,
             maximum=0,
             num_steps=50,
             style=wx.SL_HORIZONTAL,
             cast=float,
             proportion=1,
        )
        self.GridAdd(_squelch_level_sizer, 3, 1, 1, 1)
        _frequency_correction_sizer = wx.BoxSizer(wx.VERTICAL)
        self._frequency_correction_text_box = forms.text_box(
             parent=self.GetWin(),
             sizer=_frequency_correction_sizer,
             value=self.frequency_correction,
             callback=self.set_frequency_correction,
             label="Frequency correcction",
             converter=forms.float_converter(),
             proportion=0,
        )
        self._frequency_correction_slider = forms.slider(
             parent=self.GetWin(),
             sizer=_frequency_correction_sizer,
             value=self.frequency_correction,
             callback=self.set_frequency_correction,
             minimum=20,
             maximum=40,
```

```
            num_steps=1000,
            style=wx.SL_HORIZONTAL,
            cast=float,
            proportion=1,
        )
        self.GridAdd(_frequency_correction_sizer, 2, 1, 1, 1)
        _frequency_sizer = wx.BoxSizer(wx.VERTICAL)
        self._frequency_text_box = forms.text_box(
            parent=self.GetWin(),
            sizer=_frequency_sizer,
            value=self.frequency,
            callback=self.set_frequency,
            label="Frequency ",
            converter=forms.float_converter(),
            proportion=0,
        )
        self._frequency_slider = forms.slider(
            parent=self.GetWin(),
            sizer=_frequency_sizer,
            value=self.frequency,
            callback=self.set_frequency,
            minimum=88000000,
            maximum=108000000,
            num_steps=1000,
            style=wx.SL_HORIZONTAL,
            cast=float,
            proportion=1,
        )
        self.GridAdd(_frequency_sizer, 1, 1, 1, 1)
        self.wxgui_waterfallsink2_0 = waterfallsink2.waterfall_sink_c(
            self.GetWin(),
            baseband_freq=frequency,
            dynamic_range=100,
            ref_level=0,
            ref_scale=2.0,
            sample_rate=samp_rate,
            fft_size=512,
            fft_rate=15,
            average=False,
            avg_alpha=None,
            title="Waterfall Plot",
            win=window.blackmanharris,
            size=(400,300),
        )
        self.GridAdd(self.wxgui_waterfallsink2_0.win, 0, 0, 1, 1)
        self.wxgui_fftsink2_0_0 = fftsink2.fft_sink_f(
            self.GetWin(),
            baseband_freq=0,
            y_per_div=10,
            y_divs=10,
            ref_level=0,
            ref_scale=2.0,
            sample_rate=bf_scope_bandwidth,
            fft_size=1024,
            fft_rate=5,
            average=False,
            avg_alpha=None,
            title="AF Spectrum",
            peak_hold=False,
            win=window.blackmanharris,
            size=(400,300),
```

```
        )
        self.GridAdd(self.wxgui_fftsink2_0_0.win, 0, 1, 1, 1)
        self.wxgui_fftsink2_0 = fftsink2.fft_sink_c(
             self.GetWin(),
             baseband_freq=0,
             y_per_div=10,
             y_divs=10,
             ref_level=0,
             ref_scale=2.0,
             sample_rate=rf_scope_bandwidth,
             fft_size=1024,
             fft_rate=5,
             average=False,
             avg_alpha=None,
             title="RF spectrum",
             peak_hold=False,
             win=window.blackmanharris,
             size=(400,300),
        )
        self.GridAdd(self.wxgui_fftsink2_0.win, 1, 0, 3, 1)
        self.rational_resampler_xxx_0_1 = filter.rational_resampler_fff(
                interpolation=bf_scope_bandwidth,
                decimation=audio_samplerate,
                taps=None,
                fractional_bw=None,
        )
        self.rational_resampler_xxx_0_0 = filter.rational_resampler_ccc(
                interpolation=rf_scope_bandwidth,
                decimation=samp_rate,
                taps=None,
                fractional_bw=None,
        )
        self.osmosdr_source_0 = osmosdr.source( args="numchan=" + str(1)
                                                + " " + "" )
        self.osmosdr_source_0.set_clock_source("internal", 0)
        self.osmosdr_source_0.set_time_source("external", 0)
        self.osmosdr_source_0.set_time_now(osmosdr.time_spec_t(time.time()),
osmosdr.ALL_MBOARDS)
        self.osmosdr_source_0.set_sample_rate(samp_rate)
        self.osmosdr_source_0.set_center_freq(frequency, 0)
        self.osmosdr_source_0.set_freq_corr(frequency_correction, 0)
        self.osmosdr_source_0.set_dc_offset_mode(2, 0)
        self.osmosdr_source_0.set_iq_balance_mode(2, 0)
        self.osmosdr_source_0.set_gain_mode(True, 0)
        self.osmosdr_source_0.set_gain(10, 0)
        self.osmosdr_source_0.set_if_gain(10, 0)
        self.osmosdr_source_0.set_bb_gain(20, 0)
        self.osmosdr_source_0.set_antenna("", 0)
        self.osmosdr_source_0.set_bandwidth(0, 0)

        self.low_pass_filter_0 = filter.fir_filter_ccf(
                    samp_rate/internal_samplerate,
                    firdes.low_pass( 1, samp_rate, 200000, 20e3,
                            firdes.WIN_HAMMING, 6.76))
        self.audio_sink_0 = audio.sink(audio_samplerate, "", True)
        self.analog_wfm_rcv_0 = analog.wfm_rcv(
             quad_rate=internal_samplerate,
             audio_decimation=internal_samplerate/audio_samplerate,
        )
        self.analog_pwr_squelch_xx_0 = analog.pwr_squelch_cc(squelch_level,
                                                          0.001, 1, True)
```

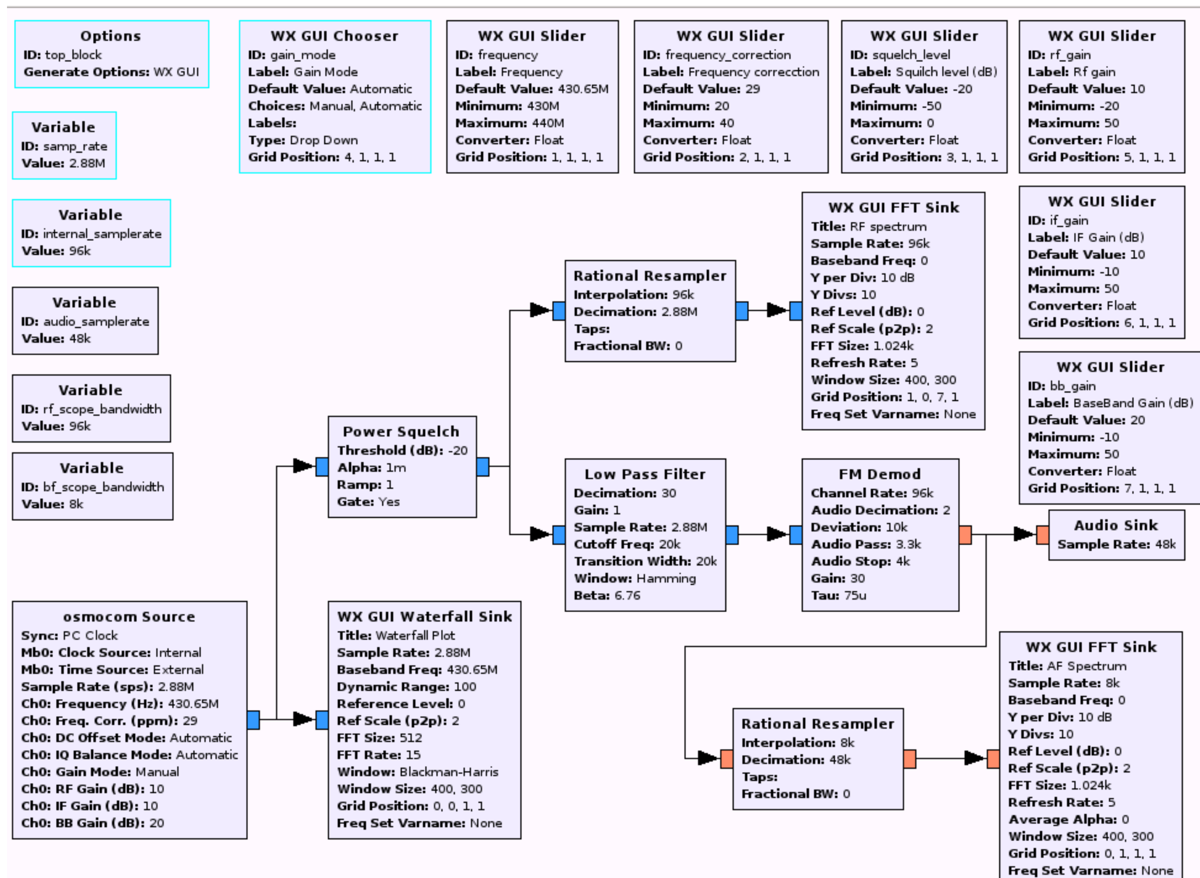
```
        ##################################################
        # Connections
        ##################################################
        self.connect((self.osmosdr_source_0, 0),
             (self.analog_pwr_squelch_xx_0, 0))
        self.connect((self.rational_resampler_xxx_0_0, 0),
             (self.wxgui_fftsink2_0, 0))
        self.connect((self.rational_resampler_xxx_0_1, 0),
             (self.wxgui_fftsink2_0_0, 0))
        self.connect((self.analog_pwr_squelch_xx_0, 0),
             (self.rational_resampler_xxx_0_0, 0))
        self.connect((self.osmosdr_source_0, 0), (self.wxgui_waterfallsink2_0, 0))
        self.connect((self.analog_wfm_rcv_0, 0), (self.audio_sink_0, 0))
        self.connect((self.low_pass_filter_0, 0), (self.analog_wfm_rcv_0, 0))
        self.connect((self.analog_wfm_rcv_0, 0),
             (self.rational_resampler_xxx_0_1, 0))
        self.connect((self.analog_pwr_squelch_xx_0, 0),
             (self.low_pass_filter_0, 0))

    def get_squelch_level(self):
        return self.squelch_level

    def set_squelch_level(self, squelch_level):
        self.squelch_level = squelch_level
        self._squelch_level_slider.set_value(self.squelch_level)
        self._squelch_level_text_box.set_value(self.squelch_level)
        self.analog_pwr_squelch_xx_0.set_threshold(self.squelch_level)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.osmosdr_source_0.set_sample_rate(self.samp_rate)
        self.low_pass_filter_0.set_taps(firdes.low_pass(1, self.samp_rate,
                                   200000, 20e3, firdes.WIN_HAMMING, 6.76))
        self.wxgui_waterfallsink2_0.set_sample_rate(self.samp_rate)

    def get_rf_scope_bandwidth(self):
        return self.rf_scope_bandwidth

    def set_rf_scope_bandwidth(self, rf_scope_bandwidth):
        self.rf_scope_bandwidth = rf_scope_bandwidth
        self.wxgui_fftsink2_0.set_sample_rate(self.rf_scope_bandwidth)

    def get_internal_samplerate(self):
        return self.internal_samplerate

    def set_internal_samplerate(self, internal_samplerate):
        self.internal_samplerate = internal_samplerate

    def get_frequency_correction(self):
        return self.frequency_correction

    def set_frequency_correction(self, frequency_correction):
        self.frequency_correction = frequency_correction
        self._frequency_correction_slider.set_value(self.frequency_correction)
        self._frequency_correction_text_box.set_value(self.frequency_correction)
        self.osmosdr_source_0.set_freq_corr(self.frequency_correction, 0)
```

```
    def get_frequency(self):
        return self.frequency

    def set_frequency(self, frequency):
        self.frequency = frequency
        self._frequency_slider.set_value(self.frequency)
        self._frequency_text_box.set_value(self.frequency)
        self.osmosdr_source_0.set_center_freq(self.frequency, 0)
        self.wxgui_waterfallsink2_0.set_baseband_freq(self.frequency)

    def get_bf_scope_bandwidth(self):
        return self.bf_scope_bandwidth

    def set_bf_scope_bandwidth(self, bf_scope_bandwidth):
        self.bf_scope_bandwidth = bf_scope_bandwidth
        self.wxgui_fftsink2_0_0.set_sample_rate(self.bf_scope_bandwidth)

    def get_audio_samplerate(self):
        return self.audio_samplerate

    def set_audio_samplerate(self, audio_samplerate):
        self.audio_samplerate = audio_samplerate

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
    tb = top_block()
    tb.Start(True)
    tb.Wait()
```

Listing 55: Python code generated by gnuradio-companion for WBFM receiver

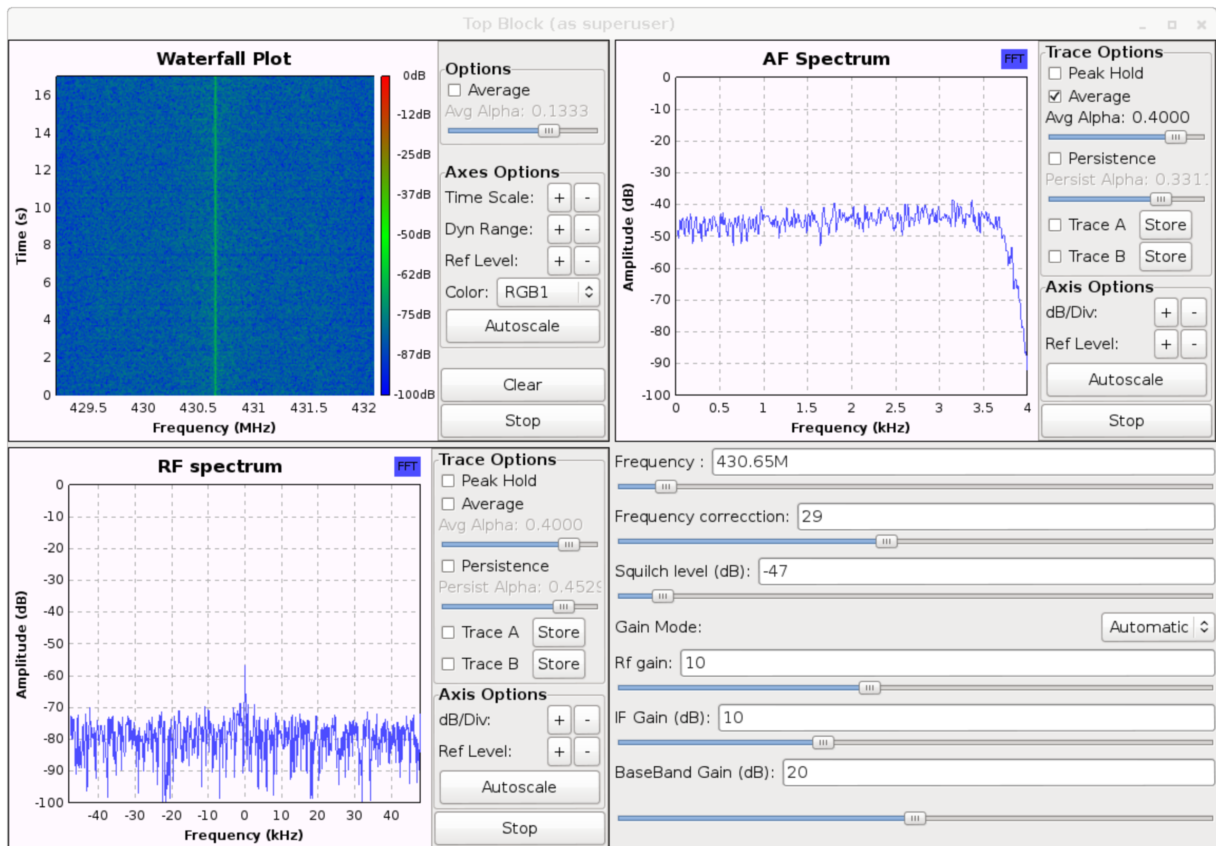
B.2 - Narrow Band FM Monoaural Receiver

We designed a second “test” receiver to receive OM FM communications to verify the dongle performances in the UHF band and also to characterize the received signal, allowing us to see a large waterfall, in-channel spectrum, as well as audio spectrum.
Comparing this to the first realization, we added some controls to modify relevant dongle parameters such as RF, IF, and BB gain. The relative flowgraph is shown in Picture 49.



Picture 49: GNU Radio companion NBFM block diagram

The design of this radio is not very different from the previous one: mainly filters/samples to cope with the different bandwidth, and gain commands added to explore how much sensible the dongle could be. Also, we used a different demodulator block, more suitable to narrow band FM demodulation.

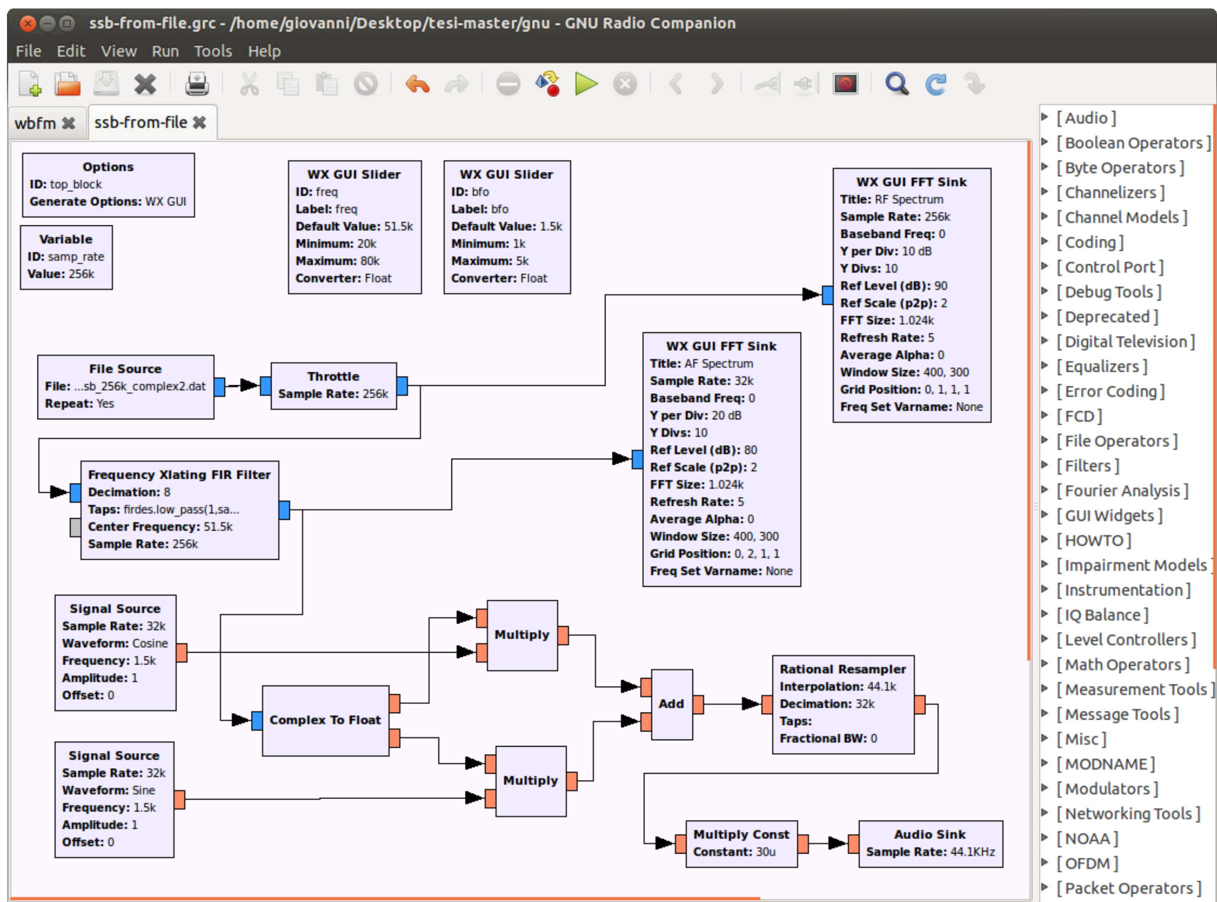


Picture 50: GNU Radio companion NBFM interface

The first times, this receiver was not simple to test because the very few transmission in UHF Radio Amateur band, even if we listened on 432 MHz repeaters segment. Picture 50 shows a very weak signal on 430.650 (maybe a spurious one?)
Using a block of attenuators we were able to test the emission of a transceiver to verify the absence of spurious signals, and the audio characteristics.

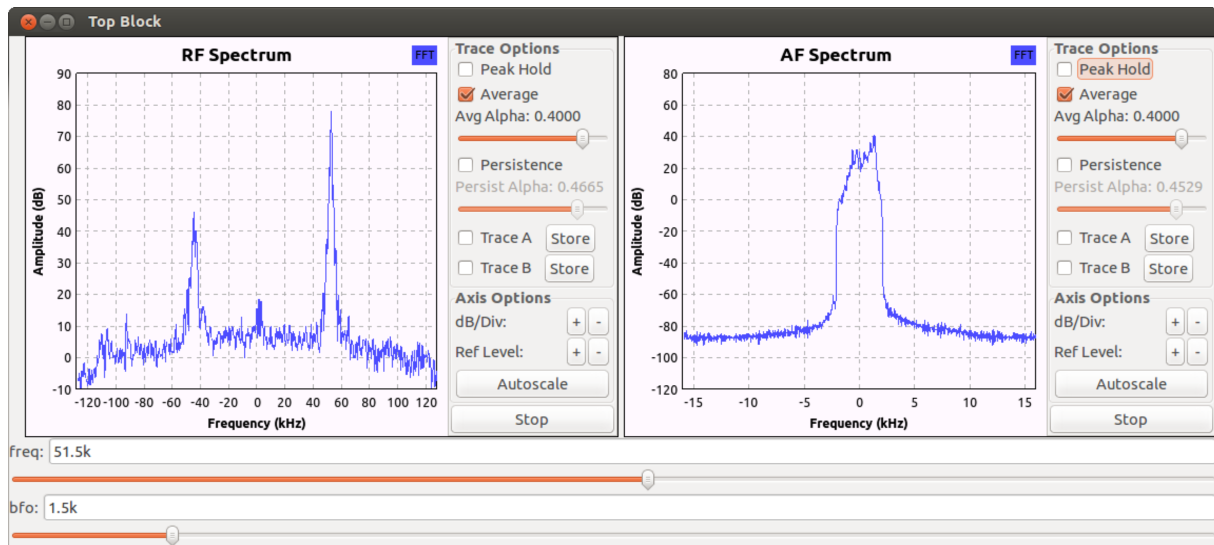
B.3 - SSB Receiver

This receiver is a little bit more complex than the previous, because it has an “if” stage used to filter the signal before the demodulation, its flowgraph can be seen in Picture 51.
To test the concept we build a “proof of concept” using a GNU Radio tutorial that uses an I/Q stream recorded using an USRP receiver operating on the 6m amateur band.
There are some differences in the flowgraph: a frequency translating filter conceptually similar to the mixer-middle frequency channel in a super heterodyne receiver, and a quadrature demodulator that implements a 'prostapheresys adder' between frequencies.
It must be noted that a *Throttle* block that “serves” the samples with a given sample rate must be inserted after the *File Source* block.



Picture 51: GNU Radio companion SSB from file block diagram

The results can be seen in Picture 52.



Picture 52: GNU Radio companion SSB from file interface

To make the receiver a real one, the *File source* and the *Throttle* blocks must be substituted with an Osmocom receiver source, and its Frequency control.

B.4 - Knob Commands

The previous flowgraphs rely on the qt/wx GNU Radio widgets for controlling the input. But, to obtain the look and feel of a *real* radio it is really important to have *real* commands as knobs, to control things as frequency, volume, or audio gain.
This could be done with a little bit of ingenuity by using an Arduino board and some encoders, and then, writing a block that reads the Arduino output and change the variable accordingly.

The first thing to do is to attach two encoders to an Arduino: it is really simple, because it is enough to connect their I/Q outputs on pins 2/4 and 3/5, and power the two encoders using GND and 5V from the Arduino itself.
The Arduino is capable to set interrupts on pins 2 and 3. The interrupt reads the state of pins 2 and 4 (or pin 3 and 5) to determine the direction of movement of the first (or second) encoder, then send an appropriate symbol as +/– (or >/< for the second encoder) over the serial USB. All the code needed is presented in Listing 56.

```
#define encoder0PinA  2
#define encoder0PinB  4
#define encoder1PinA  3
#define encoder1PinB  5
volatile unsigned int encoder0Pos = 0;
void setup() {
  pinMode(encoder0PinA, INPUT);
  digitalWrite(encoder0PinA, LOW);
  pinMode(encoder0PinB, INPUT);
  digitalWrite(encoder0PinB, LOW);
  pinMode(encoder1PinA, INPUT);
  digitalWrite(encoder1PinA, LOW);
  pinMode(encoder1PinB, INPUT);
  digitalWrite(encoder1PinB, LOW);
  attachInterrupt(0, doEncoder0, CHANGE);  // encoder pin on interrupt 0 - pin 2
  attachInterrupt(1, doEncoder1, CHANGE);  // encoder pin on interrupt 1 - pin 3
  Serial.begin (9600);
}
void loop(){}
void doEncoder0() {
  /* If pinA and pinB are both high or both low, it is spinning
   * forward. If they're different, it's going backward.
   */
  if (digitalRead(encoder0PinA) == digitalRead(encoder0PinB)) {
    Serial.print("+");
  } else {
    Serial.print("-");
  }
}
void doEncoder1() {
  /* If pinA and pinB are both high or both low, it is spinning
   * forward. If they're different, it's going backward.
   */
  if (digitalRead(encoder1PinA) == digitalRead(encoder1PinB)) {
    Serial.print(">");
  } else {
    Serial.print("<");
  }
}
```

Listing 56: Code to manage encoders on Arduino Uno board

The second thing to do is write a block that reads the signals. This could be done as seen previously, but with some small differences, that can be seen in Listing 57. These differences are related to the fact that the module is not a C++ module in the flowgraph signal path, but it is a block that operate on a variable, without the need of a GUI.

```
gr_modtool newmod usbknobs
cd gr-usbknobs/
gr_modtool add -t general --skip-lib --skip-swig -l python knobs
#Answer 'no' to generation on QA python code
mv grc/usbknobs_knobs.xml grc/variable_knobs.xml
# insert xml code
vi grc/variable_knobs.xml
# insert python code
vi python/knobs.py
mkdir build; cd build; cmake ..; make; cd ..
```

Listing 57: Usbknobs module generation

Please note that the name of the block is not *usbknobs_knobs.xml*, as expected, but, instead, *variable_knobs.xml*. This is because gnuradio-companion, when generating *top_level.py*, operates differently when a block name starts with the string 'variable', generating code for variables and callbacks. Also the <key> element has the same name, as it can be seen in red in Listing 58.

In Listing 58, the <var_make> element that contains the variable definition is presented in blue, the <make> element that contains the python code to instance the knob class is presented in green, and the <callback> element that contains the setter callback that the class will call when a new value arrive is presented in magenta. All those elements became part of *top_block.py* that is the python application generated and launched by gnuradio-companion.

```
<?xml version="1.0"?>
<block>
	<name>knobs</name>
	<key>variable_knobs</key>
	<category>usbknobs</category>
	<import>import usbknobs</import>
	<var_make>self.$(id)=$(id)=$value</var_make>
	 <make>usbknobs.knobs(
	value=self.$id,
	callback=self.set_$(id),
	port=$serial_port,
	speed=$ser_speed,
	minimum=$min,
	maximum=$max,
	step_value=$step_value,
	cast=$(converter.knob_cast),
	channel=$channel,
	islinear=$(is_linear.knob_islinear),
)
	</make>
	<callback>self.set_$(id)(self,$value)</callback>
	<param>
		<name>Serial port</name>
		<key>serial_port</key>
		<value>/dev/ttyUSB0</value>
		<type>string</type>
		<hide>part</hide>
```

```
		</param>
		<param>
			<name>Serial speed</name>
			<key>ser_speed</key>
			<value>9600</value>
			<type>int</type>
			<hide>part</hide>
		</param>
		<param>
			<name>Default Value</name>
			<key>value</key>
			<value>50</value>
			<type>real</type>
		</param>
		<param>
			<name>Minimum</name>
			<key>min</key>
			<value>0</value>
			<type>real</type>
		</param>
		<param>
			<name>Maximum</name>
			<key>max</key>
			<value>100</value>
			<type>real</type>
		</param>
		<param>
			<name>Step Value</name>
			<key>step_value</key>
			<value>100</value>
			<type>real</type>
		</param>
		<param>
			<name>Converter</name>
			<key>converter</key>
			<value>float_converter</value>
			<type>enum</type>
			<hide>part</hide>
			<option>
				<name>Float</name>
				<key>float_converter</key>
				<opt>knob_cast:float</opt>
			</option>
			<option>
				<name>Integer</name>
				<key>int_converter</key>
				<opt>knob_cast:int</opt>
			</option>
		</param>
		<param>
			<name>Channel</name>
			<key>channel</key>
			<value>0</value>
			<type>int</type>
			<hide>part</hide>
		</param>
		<param>
			<name>Is linear</name>
			<key>is_linear</key>
			<value>knob_islinear</value>
			<type>enum</type>
			<hide>part</hide>
```

```
            <option>
                  <name>Linear</name>
                  <key>is_linear</key>
                  <opt>knob_islinear:True</opt>
            </option>
            <option>
                  <name>Logaritmic</name>
                  <key>is_log</key>
                  <opt>knob_islinear:False</opt>
            </option>
      </param>
      <check>$min &lt;= $value &lt;= $max</check>
      <check>$min &lt; $max</check>
      <doc>
This block creates a variable with a slider. \
Leave the label blank to use the variable id as the label. \
The value must be a real number. \
The value must be between the minimum and the maximum. \
The number of steps must be between 0 and 1000.

Use the Grid Position (row, column, row span, column span) \
to position the graphical element in the window.

Use the Notebook Param (notebook-id, page-index) \
to place the graphical element inside of a notebook page.
      </doc>
</block>
```

Listing 58: Knobs block XML description

Gnuradio-companion uses all the <param> elements to prepare the GUI. The values of the modified ones are written in the xml file where the flowgraph is saved.

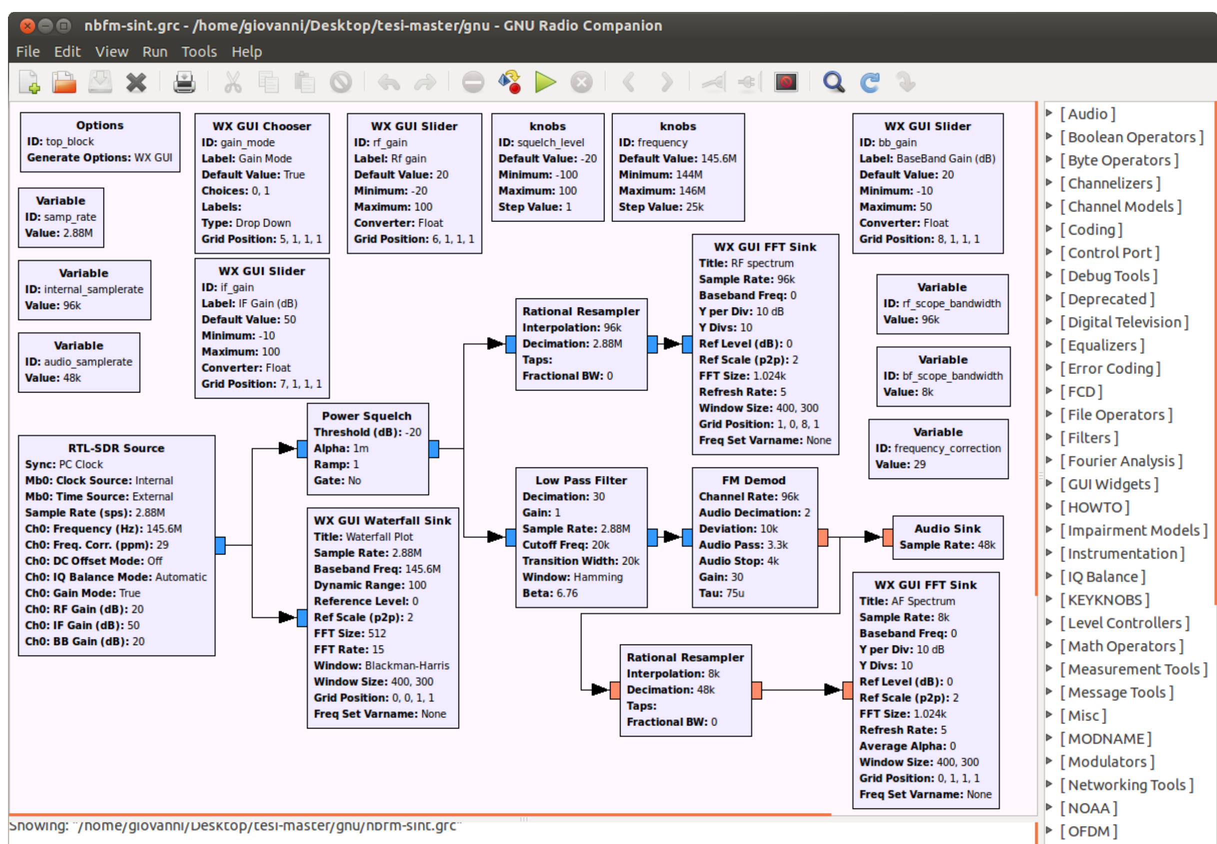
The other important file is the python class that reads the Arduino output and set accordingly the values by calling the callback written in *top_block.py*.
This file (that can be seen in Listing 59) lays in the python folder in the *gr-usbknob* module folder and it is named *knobs.py*.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import thread
import threading
import serial
from gnuradio import gr

up=['+','>','U','u','R','r']
down=['-','<','D','d','L','l']
ports=[]
ser=[]
events=[]
channels=[]
data=[]

class knobs(object):
      """
      docstring for block knobs
```

```
	"""
	def __init__(self, value, callback, port, speed, minimum, maximum,
step_value, cast, channel, islinear):
		katch=False
		for po in ports:
			if po==port:
				katch=True
				break
		if not katch:
			ports.append(port)
			ser1 = serial.Serial(port,speed)
			ser.append(ser1)
			thread.start_new_thread(rC,(self,ser1))
		data.append(0)
		events.append(threading.Event())
		channels.append(channel)
		events[-1].clear()
		thread.start_new_thread(sF,(self, value, callback, len(events)-1,
			minimum, maximum, step_value, cast, channel, islinear ))

def rC(tb,ser):
	while True:
		incr = ser.read()
		for j in range(len(channels)):
			if incr==up[channels[j]] or incr==down[channels[j]]:
				data[j]=incr;
				events[j].set()
				break
		ser.flushInput()


def sF(tb,value,callback,num,minimum,maximum,step_value,cast,channel,islinear):
	vls = value
	min = minimum
	max = maximum

	while True:
		events[num].wait()
		incr = data[num]
		data[num]=0
		if incr == up[channel]:
			if islinear:
				vls += step_value
			else:
				vls *= step_value
			if vls > max:
				vls=max
			callback(vls)
		if incr == down[channel]:
			if islinear:
				vls -= step_value
			else:
				vls /= step_value
			if vls < min:
				vls=min
			callback(vls)
		events[num].clear()
```

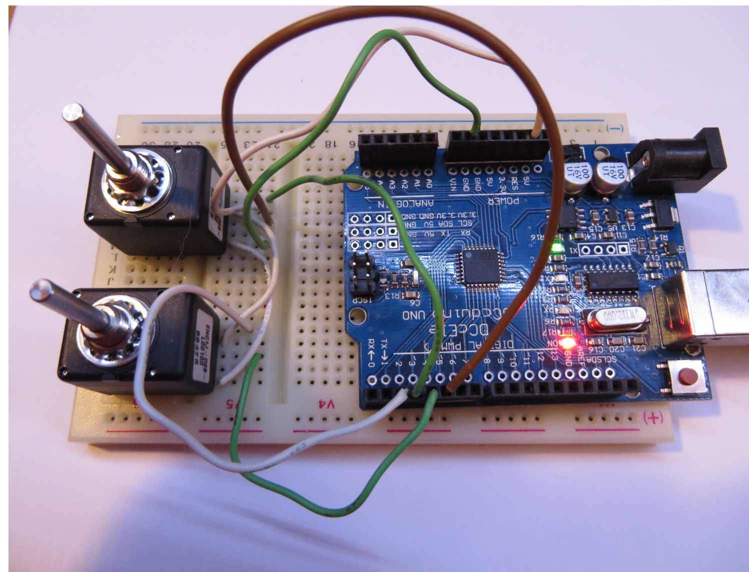
Listing 59: Knobs python class

In Listing 59 there is also the class initializer. This metod verifies if there is a reader registered for the port serial line. If the reader is not registered, the port value is not in the ports[] array, and the init function creates a thread based on the rC() function.
Once it has done this, the initializer creates another thread, based on the sF() function, to manage the single *channel*. The channels correspond to the different encoders. They send different couples of characters on the same serial line.
The rC() function opens the serial port and reads a character, then it dispatches the character to the right buffer, corresponding to the channel, and fires the right thread.
The sF() function waits for an event, then computes the value and fires the callback to put the value in the right variable. The sF() function can compute the value operating either in linear or in logarithmic mode, because linear is needed for frequencies, but logarithmic is better for audio volumes.

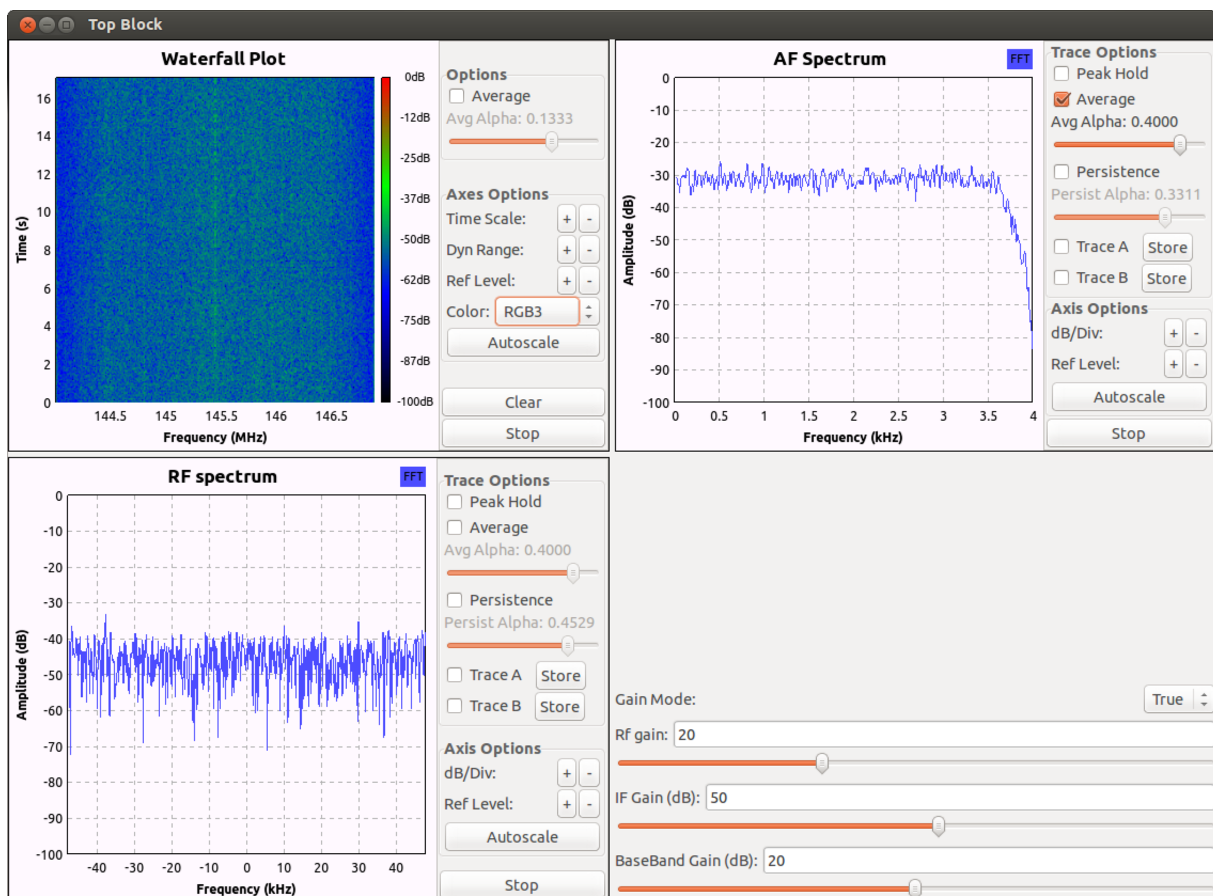


Picture 53: GNU Radio companion NBFM with knobs block diagram

In the central upper part of Picture 53 the two *knobs* blocks that control both squelch level and frequency are presented. As can be seen in Picture 55, there are no GUI controls for these blocks, the controls are physical controls that can be seen in Picture 54.



Picture 54: Arduino Uno manages two optical incremental encoder



Picture 55: GNU Radio companion NBFM with knobs GUI

In Listing 60 is presented the XML code that defines the *variable_knobs* block that implements the *frequency* control. The code is used to save the position in the graph, as can be seen in Picture 55, and the values changed during the flowgraph editing.

```xml
<block>
  <key>variable_knobs</key>
  <param>
    <key>id</key>
    <value>frequency</value>
  </param>
  <param>
    <key>_enabled</key>
    <value>True</value>
  </param>
  <param>
    <key>serial_port</key>
    <value>/dev/ttyUSB0</value>
  </param>
  <param>
    <key>ser_speed</key>
    <value>9600</value>
  </param>
  <param>
    <key>value</key>
    <value>145600000</value>
  </param>
  <param>
    <key>min</key>
    <value>144000000</value>
  </param>
  <param>
    <key>max</key>
    <value>146000000</value>
  </param>
  <param>
    <key>step_value</key>
    <value>25000</value>
  </param>
  <param>
    <key>converter</key>
    <value>float_converter</value>
  </param>
  <param>
    <key>channel</key>
    <value>0</value>
  </param>
  <param>
    <key>is_linear</key>
    <value>is_linear</value>
  </param>
  <param>
    <key>alias</key>
    <value></value>
  </param>
  <param>
    <key>_coordinate</key>
    <value>(600, 11)</value>
  </param>
  <param>
    <key>_rotation</key>
    <value>0</value>
  </param>
</block>
```

Listing 60: Part of grc XML file with frequency variable_knobs definition