Budapest University of Technology and Economics
Department of Broadband Infocommunications and Electromagnetic Theory

András Retzler

Software Defined Radio Receiver Application with Web-based Interface

BSc Thesis

Thesis supervisors:

Péter Horváth, PhD
Associate Professor

Péter Bakki
Assistant Lecturer

Contents

HALLGATÓI NYILATKOZAT

Alulírott Retzler András, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2014. 12. 19.

….....

Retzler András

1 Abstract

Software Defined Radio (SDR) has recently became a popular technology in the telecommunications industry. Its many advantages, including flexibility, reconfigurability and reliability, approve its wide use in radio frequency (RF) communication devices of today and tomorrow. As more and more integrated radio solutions became available, cheap universal SDR devices have appeared with wide tuning range and high sampling rates.

In this thesis, design and implementation of an SDR receiver application, *OpenWebRX* is presented. *OpenWebRX* has the following features:

- It can be used as a communication receiver for analog modulations (AM/FM/SSB).
- It can use USB dongles based on RTL2832U IC as input RF front-end.
- It allows multiple users to connect via a web interface, on which it displays a real-time waterfall display.
- It allows users to select different channels within the bandwidth of the sampled signal acquired from the RF front-end. The selected channel is demodulated and the resulting audio is streamed to the browser of the user, where it is played back on the sound card. Users can set receiver parameters (channel frequency, modulation mode, filter envelope) independently.
- The web interface supports multiple browsers and uses modern browser features introduced in HTML5.

The digital signal processing (DSP) functions were placed in a separate library, *libcsdr*. It contains functions for digital downconversion, filtering and demodulation of AM/FM/SSB signals.

The purpose of the software is to enable amateur radio operators to set up receiver stations that are remotely accessible through the Internet. Both *OpenWebRX* and *libcsdr* are released under open-source licenses to let others modify, improve or support it later.

By the time of finishing this thesis, *OpenWebRX* is already being tested in real-world use by several amateur radio operators.

2 Összefoglaló

A Software Defined Radio (SDR) mára a telekommunikációs iparág kedvelt technológiájává vált. Az olyan előnyei, mint a rugalmasság, az újrakonfigurálhatóság és a megbízhatóság jogossá teszik a használatát a jelen és jövő rádiófrekvenciás (RF) kommunikációs eszközeiben. Egyre több integrált RF megoldás jelenik meg a piacon, köztük olcsó, univerzális SDR hardverek is, amelyek széles sávban hangolhatók és gyors mintavételt tesznek lehetővé.

Dolgozatomban egy SDR vevő alkalmazás tervezéséről és megvalósításáról írok. Az alkalmazást *OpenWebRX*-nek neveztem el. Az alábbi funkciókkal rendelkezik:

- Úgy használható, mint egy analóg modulációs módokat (AM/FM/SSB) célzó kommunikációs vevő.
- RTL2832U alapú USB eszközöket tud kezelni jelforrásként.
- Webes felületére több felhasználó is csatlakozhat, és valós időben frissített vízesés-diagramon tekintheti meg a vételi sáv viszonyait.
- A felhasználó kiválaszthat egy csatornát, amit a kiszolgáló demodulál és a böngészőbe hang adatfolyamként továbbít, ahol lejátszásra kerül a hangkártyán. A felhasználók egymástól függetlenül állíthatják a vevő paramétereit (a csatorna frekvenciáját, a modulációt és a szűrő karakterisztikát is).
- A webes felület több böngésző szoftvert is támogat, és olyan funkciókat is használ, amik a HTML5 újdonságaiként jelentek meg.

A digitális jelfeldolgozás (DSP) egy külön függvénykönyvtárba, a *libcsdr*-be került. Ez tartalmazza a digitális lekeveréshez, a szűréshez és az AM/FM/SSB demodulációhoz szükséges függvényeket.

A szoftver célja, hogy a rádióamatőrök olyan vevőállomásokat állíthassanak fel, amelyek az interneten keresztül is elérhetők. Mind az *OpenWebRX*, mind a *libcsdr* nyílt forráskódú licensszekkel van közzétéve, amely lehetővé teszi mások számára a kód későbbi módosítását, javítását és támogatását.

A dolgozat befejezésekor az *OpenWebRX*-et már több rádióamatőr is teszteli való életbeli alkalmazásban.

3 Introduction to OpenWebRX

With the increasing number of integrated radio solutions becoming available, System on a chip (SoC) designs for radio frequency (RF) applications have gained popularity in the industry. On the other hand, the computational speed we can achieve with general purpose CPUs, application-specific integrated circuits (ASIC) or field-programmable gate array (FPGA) chips is also increasing. It also implies that building RF receivers and transmitters with digital signal processing (DSP) techniques, which is also referred as Software Defined Radio (SDR), has become a rational choice. SDR has a wide range of uses today:

- it is used in various telecommunications equipment: DVB receivers, mobile base stations, military and aerospace targeted devices, etc.
- it is used by R&D companies for prototyping and measurement of RF devices,
- it is used by amateur radio operators and hobbyists.

Its use in amateur radio is a logical choice as this activity involves experimenting with technology, and there are a lot of different modulations used by amateur radio operators for making contacts with each other over the radio. SDR makes it easy to implement both modulators and demodulators.

Several desktop SDR receiver applications exist for receiving analog communication modes (Gqrx, SDR#, HDSDR, PowerSDR, QtRadio, etc.), and there are also some for mobile devices running Android (SDR Touch, glSDR). Very few SDR software provide a web-based interface (notable examples are WebSDR and ShinySDR), which can be used for simple remote access of the receiver.

The software covered by this thesis was designed in the hope to give something useful to the amateur radio community. The goal was to implement an SDR receiver software with a web interface, which is fully open-source (released under GPL license, and most of the DSP code under the even more permissive BSD license). It can serve multiple users at once, demodulating an AM/FM/SSB/CW transmission of their choice from a signal acquired by a sufficient SDR hardware with a ‘digital IF’ architecture (detailed later).

On the client side, it requires only an up-to-date web browser (Google Chrome or

Mozilla Firefox) to access the server. It presents the RF spectrum visually on a spectrogram (also referenced as ‘waterfall display’ in the thesis), where the signal to be received can be selected. The web front-end uses modern browser features introduced in HTML5: these include the <canvas> element, Web Audio API and WebSocket. The operation of the software can be represented with the simple block diagram shown in Figure 1.

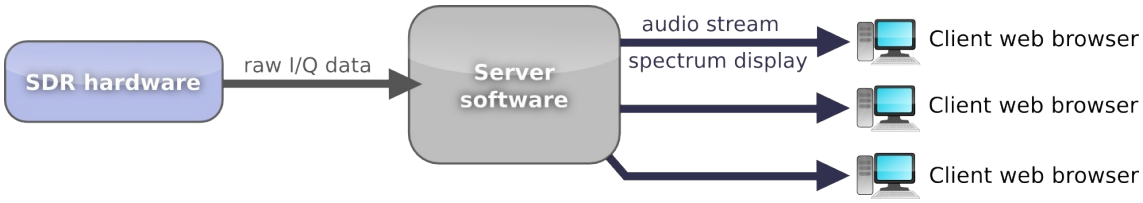

Figure 1: Block diagram of web-based SDR software

In the following part of the introduction, I explain the availability and the usage of the software. In chapter 4, I comment on SDR in general. In chapters 5 to 7, I write about the system design and the underlying architecture of the server software and the front-end. In chapters 8 to 11, I explain the digital signal processing (DSP) functions used by the server. In chapter 12, I write about the lessons drawn during the project.

3.1 Software release

The software was named OpenWebRX, and it can be downloaded from GitHub (a website dedicated to hosting the source code of open-source software projects) by visiting the following URL:

https://github.com/simonyiszk/openwebrx

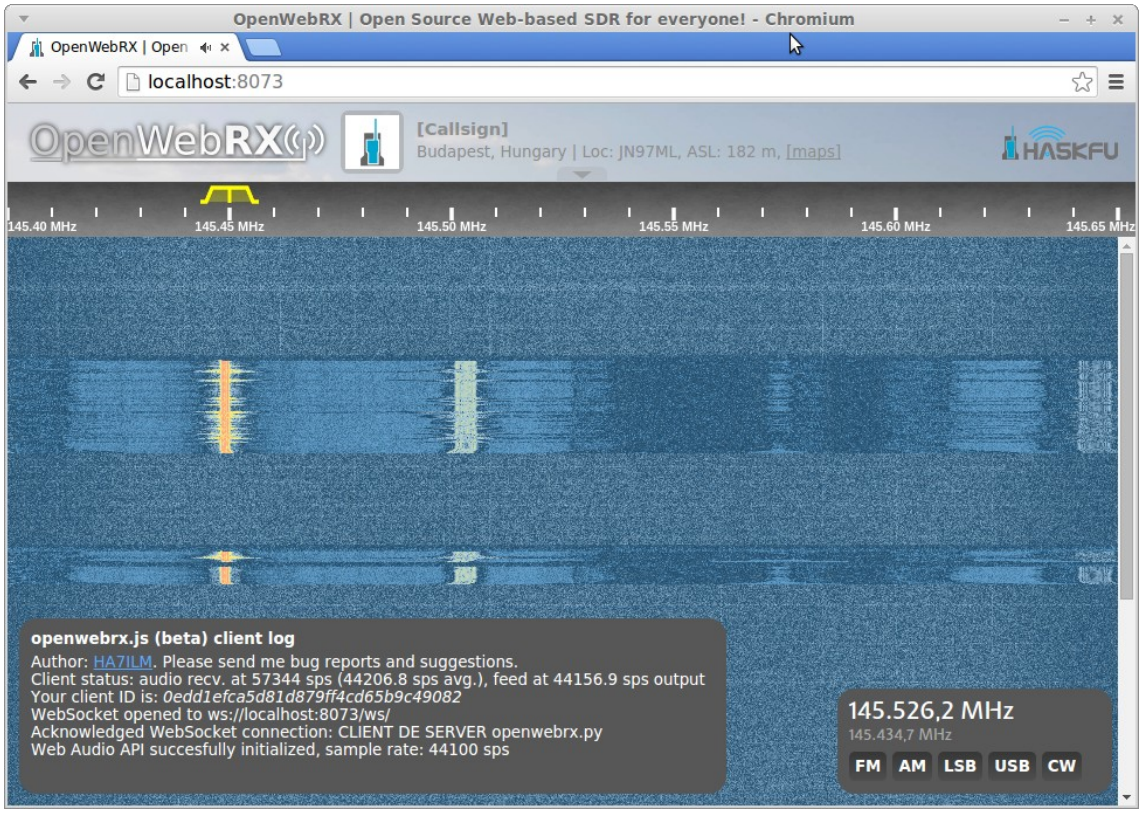

Figure 2: Screenshot of OpenWebRX running locally, with the default settings

The DSP library written for OpenWebRX has been released as a separate project:

https://github.com/simonyiszk/csdr

The DSP library was tested with the help of GNU Radio Companion, and some special GNU Radio blocks were made for this purpose. I also considered these reusable, and made them available under a different repository:

https://github.com/simonyiszk/gr-ha5kfu

The build and usage information is available on the GitHub project pages. Information regarding the exact *git* revisions this thesis refers to, is available in the Appendix.

3.2 Basic features

When an OpenWebRX server has been set up and started, the users can access it by typing the appropriate hostname and port to the address bar of the web browser (as seen in Figure 2). With the default settings, an OpenWebRX server that runs on the local machine can be accessed at the following URL: http://localhost:8073/

When a user loads the page, he is presented with a waterfall display and the audio stream starts immediately. Figure 3 shows the separate parts of the page:

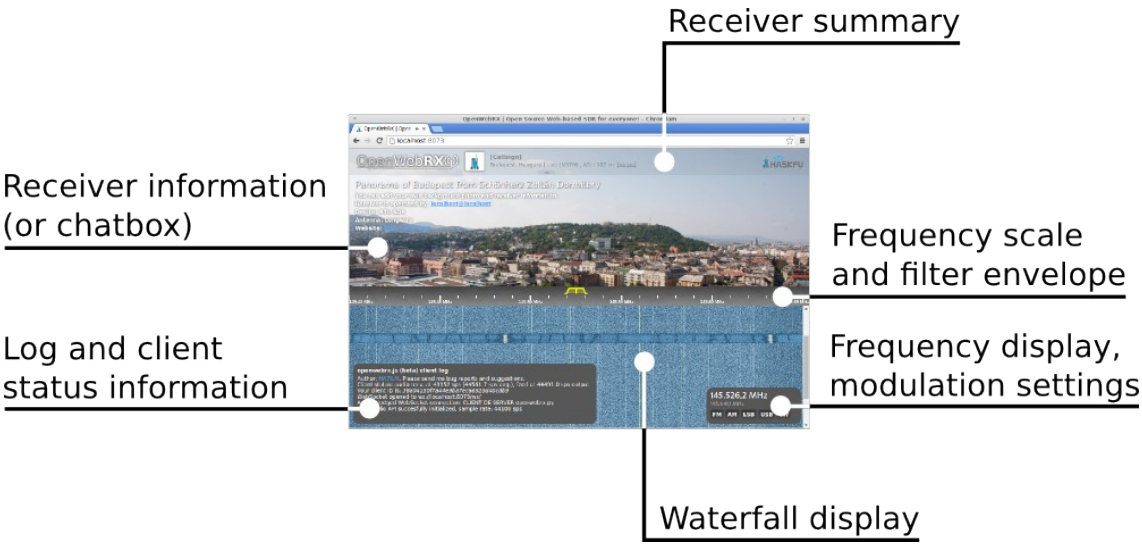

Figure 3: Parts of the OpenWebRX GUI

The top of the page contains customized information about the receiver (amateur radio call sign and e-mail address of operator, location, height above sea level). It also contains a picture taken from the receiver site (it is intended to be replaced by image automatically taken from a web camera in later versions). There is also support for including a chat box in the top of the page (via service provided on http://tlk.io), which allows users to discuss about the signals received.

Frequency can be changed by clicking on the scale or the waterfall display. The beginning and the ending of the filter envelope can be moved in order to change the digital IF filter bandwidth (as in Figure 4).

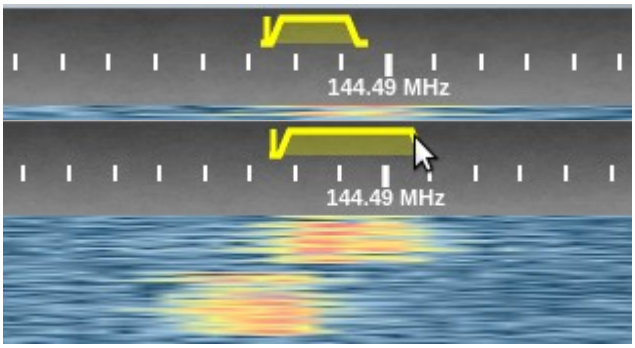

Figure 4: Filter envelope before and after user has changed the filter bandwidth

By holding down the shift key, the entire passband can be moved (imitating the Passband Shift or PBS knob on traditional receivers), or the local oscillator (LO) frequency can be changed without moving the passband (imitating the Beat Frequency Oscillator or BFO knob on traditional receivers in CW mode).

In the right bottom corner, the actual frequency of the receiver, and the frequency under the mouse pointer is shown (if the mouse is moved over the waterfall display). With the buttons, several different demodulators can be selected (see in Figure 5).

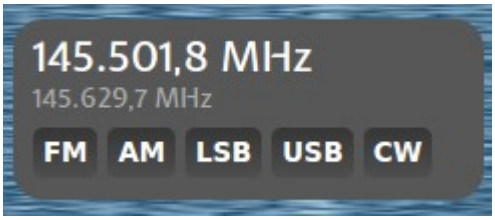

Figure 5: Frequency and modulation settings

Figure 6 shows that the waterfall display itself can be zoomed by moving the mouse pointer over it and turning the mouse wheel.

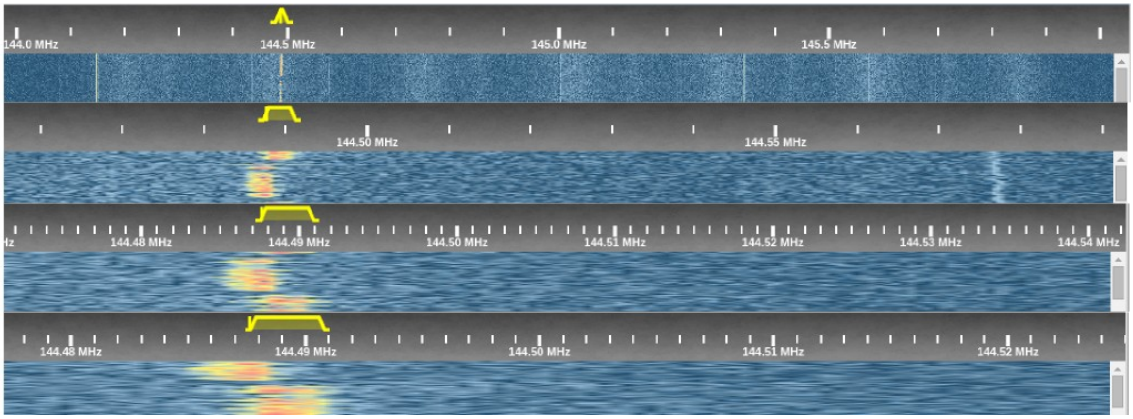

Figure 6: The waterfall display at different zoom levels

The zoomed spectrum display can also be panned by pressing and holding down the left mouse button. There is a scrollbar on the right side of the waterfall display, which allows the user to go back in time and view any part of the waterfall drawn since loading the page.

In Figure 7, the logging section can be seen, which provides additional debug and contact information for bug reports.

Figure 7: Log display and status information

Figure 8 shows a screenshot of a public server running OpenWebRX, as it has already been downloaded and tested by several amateur radio operators.

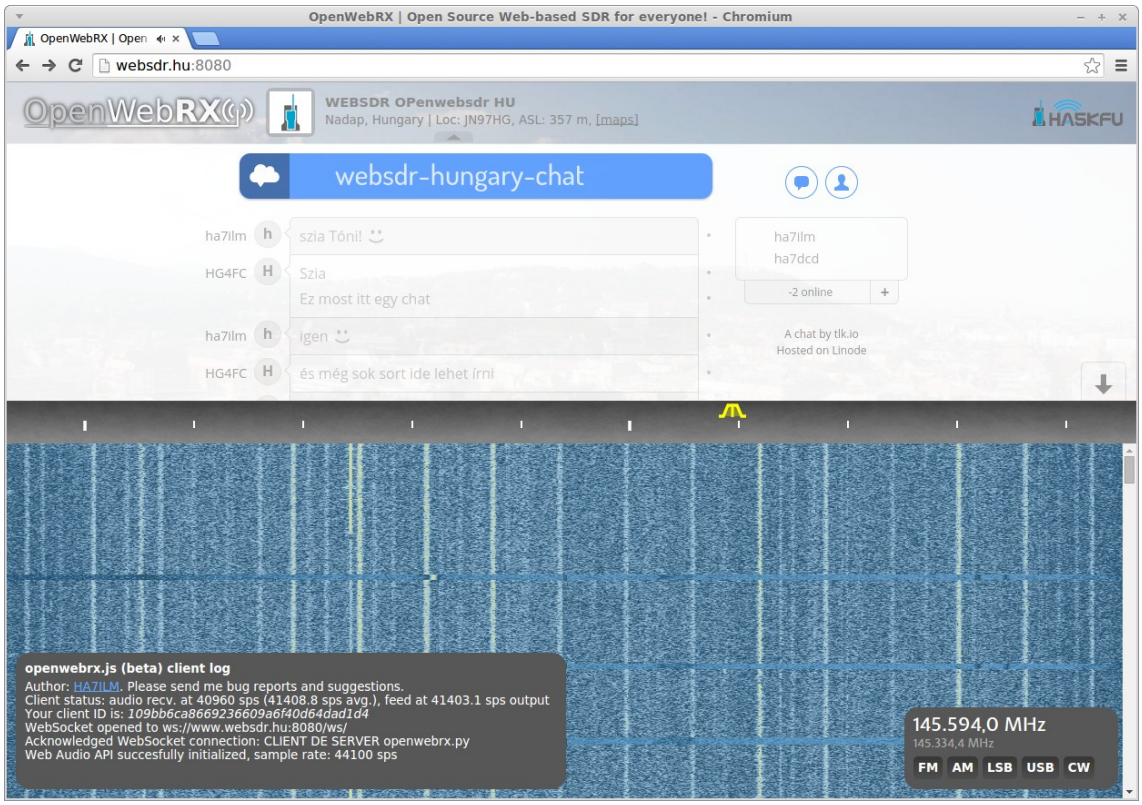

Figure 8: The first public OpenWebRX server running from Nadap, Hungary

4 Fundamentals of Software Defined Radio

4.1 Introduction and history

As OpenWebRX heavily builds on SDR concepts, it accounts for giving an overview on SDR technology, and how it is related to my project.

The systems that the term ‘Software Defined Radio’ covers, implement some or all physical layer functions of a radio in software instead of hardware, which also implies that the software does digital signal processing (DSP) tasks. [1]

In fact, SDR is not new technology, it has been available since the 1980s. The term ‘software radio’ has been first used by the employees of E-Systems Inc. in a company newsletter in 1984. The first military program that had the physical layer components of a radio implemented in software, was called SPEAKeasy, designed by DARPA in the United States. The main objective was to build a single radio that is compatible with ten different military radio protocols, can operate anywhere between 2 MHz and 2 GHz, and also have the possibility of including new modulations and protocols later. [2]

Although the theoretical background required for building an SDR system has been around for a long time, its true potential have been opened slowly, in parallel with the increasing computational performance of computers.

4.2 Advantages and disadvantages

Before diving into SDR, I compare it to traditional analog radio systems and collect its key features.

One of the advantages of SDR is *reconfigurability,* which results in its *flexibility*. The the key part of an SDR system is the software, which can be modified and updated at any time. [3]

Let us take an example: we want to add a new demodulator to multi-mode receiver for satellite data transfer applications. Instead of having to redesign the circuit, update the printed circuit board layout, have the board of the new prototype manufactured, have all the components mounted, and go through the bring-up process, just updating the software is enough. Installing a firmware update can even be done by the customer, or

done remotely over the network, so it greatly reduces maintenance costs in several situations. Similarly, new demodulators could be easily added to OpenWebRX, and this requires only changes to the software.

On the other side, all SDRs have an analog RF front-end, and naturally, if any changes in requirements affect it (e. g. a change in the tuning frequency range), then it it impossible to avoid hardware modifications.

Reconfigurability is also essential in *cognitive radio*, which focuses on solving the problem of combinatorial optimization of different modulation schemes, power levels, error control codes, operating frequencies, and also network behavior to achieve the best result in communication. It has also received great attention by regulatory agencies recently, as static allocations in the radio frequency spectrum are becoming more and more congested, but on the other hand, most of the frequency spectrum is unused at a given location and time. One application of cognitive radio, *dynamic spectrum access* can help with this issue. [4]

Another key point of SDR is *reliability*. DSP algorithms work on discrete signals and – except for some special cases - have fully predictable output, giving exactly the same result for the same input every time.

If we use a DSP algorithm instead of hardware realization, no unwanted signal coupling can occur between printed circuit board (PCB) traces, and no distortion can occur because of nonlinearities that electronic components would introduce. On the other hand, SDR is limited by the properties of quantization and sampling introduced by the analog-to-digital converted (ADC) or digital-to-analog converter (DAC). An imperfect DSP implementation can also introduce noise and harmonics. The digital noise generated by the high-speed digital processing parts can occur on the analog front-end, but it can be solved by sufficient design.

Another aspect of reliability is the lifetime of the device. In an SDR, several hardware components are substituted by software. Until the processing unit and the memories belonging to it are operational, the software will produce the same results, without performance degradation due to aging or environmental effects. Faults caused by a single electronic component are less likely to happen, as there are less components. While this is certainly an advantage, it is likewise important to note that the complexity

introduced by software results in more fault possibilities. Today's SDR is typically an embedded system that can range from a single microcontroller (MCU) to a fast SoC with double data rate (DDR) memories, or may even be a dedicated server computer. On the software side, they can incorporate a single bare-metal C program or a complex real-time operating system (OS) with scheduling and peripheral device drivers. Complexity rises when we optimize the system by offloading the processing task to special peripherals like a field-programmable gate array (FPGA) chip or a graphics processing unit (GPU) to achieve highly parallel processing.

Along with manufacturers of embedded digital RF solutions, SDR technology is also appealing for short wave listeners, amateur radio operators and military users. The ADCs available today provide so high bandwidth that several channels of conventional analog communication modes (amplitude and frequency modulated, or single-sideband transmissions, as well as narrow-band data modes) can be monitored at the same time. Radio applications using carrier frequency of 30 MHz or below use small bandwidth, typically less than 10 kHz. Even with a general purpose sound card available in a laptop computer, 48-192 kHz bandwidth can be monitored if connected to sufficient SDR hardware. There are numerous simple circuits available, providing a single band receiver for specific amateur radio bands.

If we calculate the Fast Fourier Transform (FFT) of the input signal on the computer, waterfall display is available (in OpenWebRX as well), which greatly helps to detect and select the transmission to be demodulated. Amateur radio transmissions tend to be less than 3 kHz in bandwidth (to be received with any traditional SSB receiver), so several channels can be monitored at once. Some bands can be almost entirely monitored with a single sound card (e.g. the 40-meter amateur radio band from 7000-7200 kHz). As of the last stage of filtering does also happen on the PC, filter bandwidth is also selectable on the graphical user interface (GUI). Such modification may require soldering a new mechanical filter into a traditional receiver, but with SDR, it just takes some CPU cycles to design a new filter with different parameters.

However, systems that can digitize and monitor the whole high frequency (HF) range at once, also do exist. A good example is the WebSDR receiver at the University of Twente. [5]

Military systems can also record the digitized samples for later processing [6], and some are also capable of automatic modulation recognition and decoding [7]. Several software tools exist for analyzing modulations, an example is Code 300-32 by HOKA [8]. Also GNU Radio provides useful blocks that help to determine the parameters of digital modulations (e. g. constellation and the bit rate).

4.3 Software Defined Radio architectures

To generate or process RF signals with digital circuits, we have to interface the digital and the analog parts of the system: DACs and ADCs are used for this purpose. There are still several typical configurations for an SDR system. In this part I classify them by the place of conversion [6], also addressing which systems are supported by OpenWebRX. Figure 9 below illustrates typical SDR architectures.

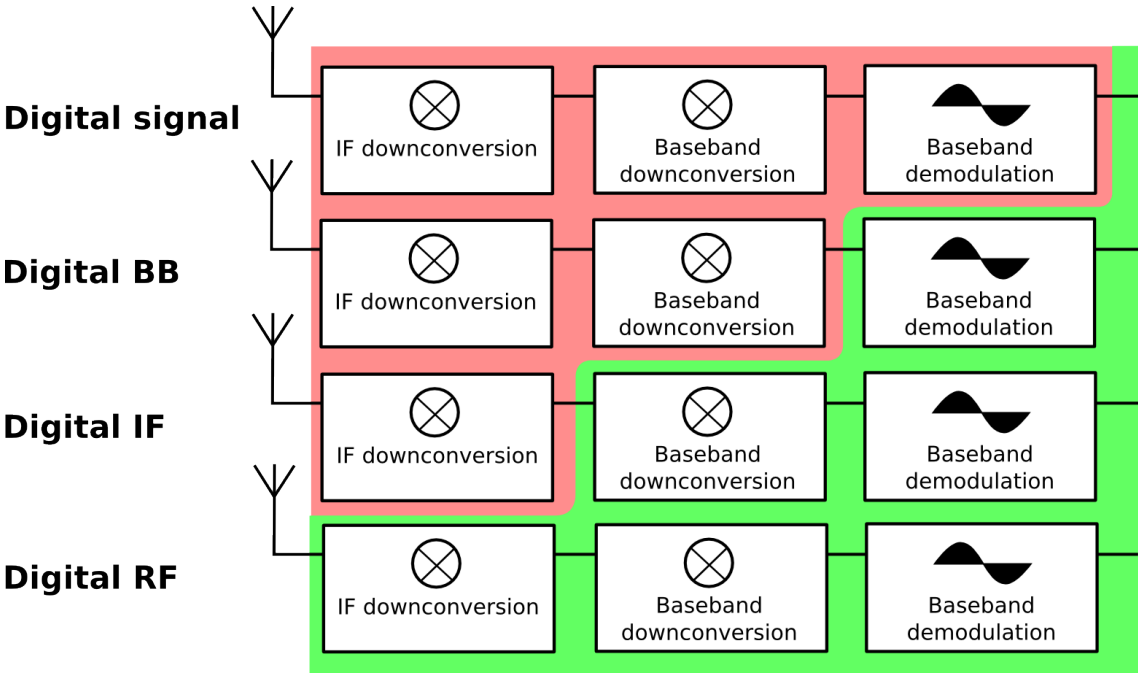

Figure 9: SDR configurations based on [6].

The ‘digital signal’ implementation means that everything is implemented in hardware, including demodulation, but the output signal of the system is digital data. Older radio modems did not use DSP.

The ‘digital baseband’ means that the baseband signal is sampled and processed by DSP for demodulation. An example of such system is a PC sound card connected to an amateur radio transceiver for using digital modes like BPSK31, RTTY, Olivia, etc. The

bandwidth of the transceiver in SSB mode is usually less than 3 kHz, but it is enough for these low bit rate signals. There are multiple free software available for this purpose (Fldigi, gMFSK).

In a ‘digital IF’ system, the signal is sampled at the intermediate frequency after downconversion and filtering. Nowadays even the lower priced marine and amateur transceivers have built-in DSP functions that work similarly. Typically noise reduction and another level of filtering is performed via DSP, or sometimes the whole demodulation process.

In a ‘digital RF’ system, the RF signal is directly sampled at the converter. It still needs filtering and amplification on the analog side, but all other processing (including downconversion) is implemented in software. Nowadays so-called RF DACs and ADCs are available for purchase. A good example is a 14-bit, 2.3 Gsps RF DAC, the MAX5879 integrated circuit. It has selectable output impulse response, and with the built-in radio-frequency-return-to-zero (RFZ) mode, even using the 6th Nyquist zone is possible, although usually such devices only support using the second and the third Nyquist zone. Another good example is the LTC2208, a 16-bit ADC which supports sampling rates up to 130 Msps. Its noise floor is at 78 dBFS, and the spurious-free dynamic range (SFDR) is 100 dB. It can be used to sample the whole shortwave (0-30 MHz) at once. (In a ‘digital baseband’ configuration, these ICs can also be used to generate or decode modulated high-speed data transmissions, over 10 Mbit/s.) However ‘digital RF’ applications usually work with high sample rates and require high-speed processing (usually implemented in FPGA). An example of a real-world hardware that use this technique is the HPSDR Mercury module.

It is important to mention that most SDR receivers use *direct quadrature downconversion.* This kind of architecture is a form of ‘digital IF’ or ‘digital baseband’, depending on which functions are implemented in DSP. As seen in Figure 10 the real valued RF signal is mixed with a sine and cosine (thus an oscillator with complex output). The low-pass filters remove the out-of-band components, and the resulting complex signal is centered at DC, and can be sampled with two ADCs. Despite its simple design, such architecture can produce quite good results.

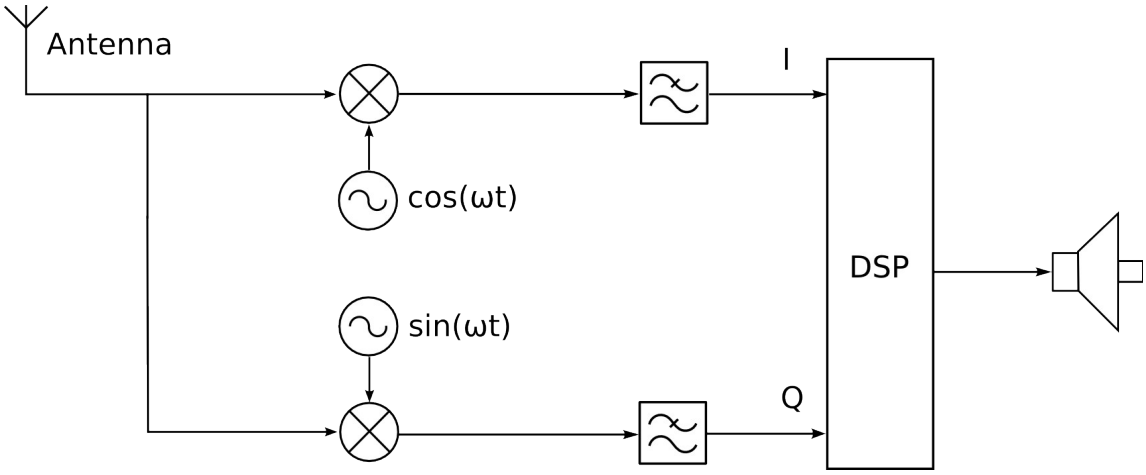

Figure 10: block diagram of an SDR receiver using direct quadrature downconversion

OpenWebRX was designed to support 'digital IF' SDR receiver hardware that uses quadrature downconversion. Currently, only RTL-SDR devices (described later in the thesis) are supported, however, support for other hardware may be added later.

4.4 The challenge of dynamic range

In an application mainly designed for amateur radio purposes, reception of weak signals (just above the noise floor) is an important question, and the performance of SDR radios from this aspect is also important for SDR software such as OpenWebRX. As already noted above, a critical point of an SDR receiver is the ADC. In addition to the sampling rate, another important parameter of this component is the bit depth, which is closely coupled with the dynamic range of the receiver. One definition for the dynamic range in a DSP-based receiver is the proportion of smallest and highest values that can be represented digitally. As every additional bit doubles this highest value, every bit means an additional dynamic range of $20 \cdot \log_{10}(2) = 6.02\,dB$. Taking a sinusoidal signal and the quantization noise into consideration, the maximal possible signal-to-noise ratio (SNR) for an ADC can be calculated as $SNR_{max} = (1.76 + 6.02 \cdot N)\,dB$ where N is the number of bits. However, on a real device the measured SNR is always lower than the theoretical, this is why the effective number of bits (ENOB) is introduced. It can be calculated from the measured SNR by (1).

$$ENOB = \frac{SNR_{\mathrm{dB}} - 1.76}{6.02} \tag{1}$$

For example the ENOB of an LTC2216 ADC is 12.83 bits (the SNR was actually measured 79 dBFS at 140 MHz) [9].

If a ‘digital IF’ system is considered, the sampled signal may contain multiple useful signals. The digital filter selects one of these and suppresses the others. The automatic gain control (AGC) reduces the gain of the input signal entering the ADC, to prevent clipping. However, if the signal we want to select is very weak, and there is another strong signal within the IF bandwidth, both of them get sampled, but most of the dynamic range of the ADC will be used up by the strong signal we want to suppress, instead of the weak signal we want to select and decode. After filtering, our weak signal will have much lower dynamic range (thus quantization resolution) than it could have if the strong signal was not present and the AGC could set the gain higher, so it might be harder to copy. If the difference between the level of the two signals is high enough, our weaker signal may even be buried in noise (Figure 11 illustrates this situation).

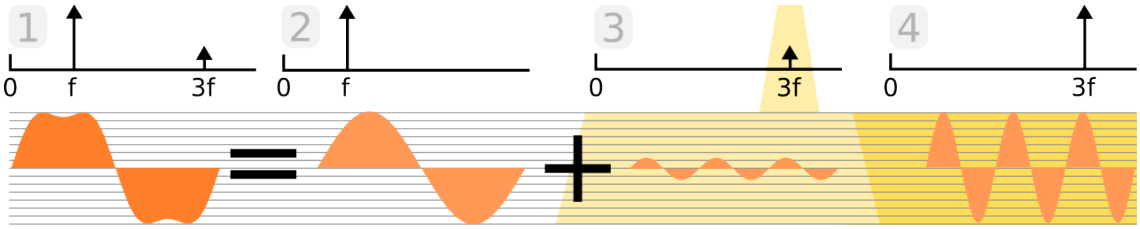

Figure 11: the input signal (1) consists of two components (2-3), one of which is selected by the filter (3), but it crosses less quantization levels than if only this signal was present on the input (4), thus it has less dynamic range, even if amplified digitally

In conclusion, a narrow-band analog receiver can provide better results than a ‘digital IF’ SDR in terms of selectivity and handling strong signals. However, regarding other advantages of SDR, these two are hardly comparable, and in my application the SDR is the only good choice (virtually unlimited number of receivers can be created within a given frequency range, one each for all clients, and the limiting factor is the CPU usage of the receivers).

4.5 Universal SDR hardware

If an SDR receiver hardware can be tuned within a wide frequency range (usually from a few hundred MHz to a few GHz), and contain an ADC that supports high sample rates (usually 1 Msps or more), it might be considered universal, as it can sample most of the signals transmitted by common RF communication devices.

It is a good choice for OpenWebRX and other SDR software to support such hardware (comparison of some devices is shown below in Table 1), because the reception of various frequency bands is possible (also depending on the antenna).

SDR	Maximum RX sample rate	ADC resolution	Tuning range	Transmit capability	Price
USRP N210 [10]	25 Msps	14 bit	DC – 6 GHz	Yes	1717 USD
Nuand bladeRF x40 [11]	40 Msps	12 bit	300 MHz - 3.8 GHz	Yes	420 USD
HackRF One [12]	20 Msps	8 bit	10 MHz - 6 GHz	Yes	300 USD
AirSpy [13]	10 Msps	16 bit	24 MHz – 1750 MHz	No	200 USD
FunCube Dongle+ [14]	192 kHz	16 bit	150 kHz - 240 MHz, 420 MHz - 1.9 GHz	No	200 USD
RTL-SDR [15]	2.4 Msps	8 bit	24 MHz – 2200 MHz *	No	10 USD

* depends on tuner IC on board

Table 1: Summary of universal SDR hardware parameters

4.6 RTL-SDR

The currently supported SDR hardware for OpenWebRX is the cheapest of all: DVB-T tuner USB dongles with RTL2832U chip (will be referred as ‘RTL-SDR’ in the thesis) can be used as a general purpose SDR hardware front-end, as these devices can provide a 8-bit baseband I/Q signal via USB interface. A sample device is shown on Figure 12.

Although their primary function is to demodulate DVB and send the MPEG transport stream to the host, they are also capable of receiving broadcast FM and DAB stations. It was discovered by the open-source community that the Windows driver sends the raw, digitized baseband I/Q signal to the PC, where it is demodulated in software. Developers at Osmocom has decided to create a library that handles this mode of operation, and it was named *librtlsdr*. Since then, several SDR applications have included support for this hardware using this library. Dedicated SDR hardware of course provide better performance than these mass produced, consumer-grade products. The main benefit of RTL-SDR is the price of the devices (around 10-40 USD in 2014), this is why it is popular among hobbyists and amateur radio operators.

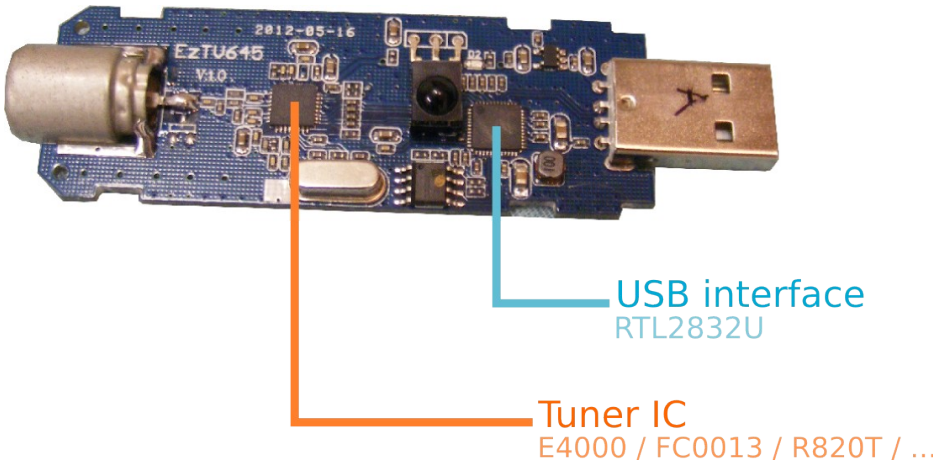

Figure 12: circuit board of RTL-SDR

Regarding architecture, a typical RTL-SDR device can be classified as a 'digital IF' device that uses quadrature downconversion. It consists of a tuner IC, and an RTL2832U chip, which contains two ADCs, a DVB demodulator and an USB interface. The tuner IC is responsible for downconversion of the RF signal to baseband or IF (depending on part), and it can be controlled via I^2C. Table 2 summarizes the different tuning limits for different types.

Tuner IC	**Tuning range**
Elonics E4000	52 – 2200 MHz (gap between 1100 - 1250 MHz)
R820T/R828D	24 – 1766 MHz
FC0013	22 – 1100 MHz
FC0012	22 – 948.6 MHz
FC2580	146 – 308 MHz, 438 – 924 MHz

Table 2: Summary of tuning range depending on tuner IC

Although there is not much official documentation publicly available regarding the RTL2832U, it is known that it digitizes the baseband or IF signal at a conversion rate of 28.8 Msps, and it contains a DDC in hardware, to produce the baseband I/Q signal of a lower sample rate [16]. The DDC uses a programmable symmetric FIR filter of 16 taps, but its length sets the limit for the lowest output sample rate to be used without aliasing problems. If *librtlsdr* is used, the built-in DVB demodulator is switched off, and the I/Q samples are directly sent to the host PC via a bulk USB endpoint. It seems that the USB interface has a limit on data rate: above 2.4 Msps it starts to drop samples. It is also remarkable that there have been various hardware and software modifications, primarily

with the goal of extending the tuning range down to the high frequency (HF, 0-30 MHz) range.

A simplified block diagram of a DVB-T tuner with RTL2832U is shown on Figure 13.

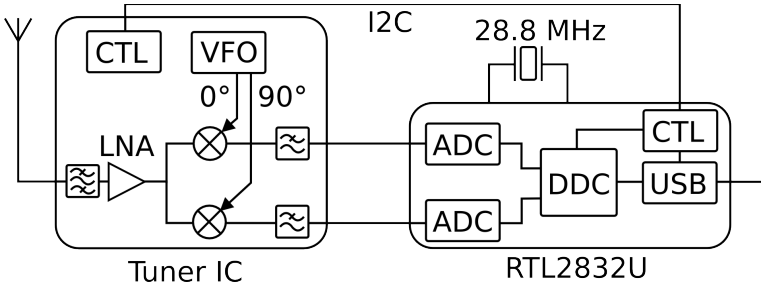

Figure 13: suspected architecture of an RTL-SDR (without DVB-T demodulation blocks)

As RTL-SDR devices are widely used by the amateur radio and hobby SDR community, it was a rational choice as the first supported hardware platform for OpenWebRX.

5 System design

In this section I write about the concepts behind OpenWebRX and its structure.

5.1 Analysis of similar software

I have tested other SDR software that provide web interfaces, and collected their advantages.

WebSDR [17] by Pieter-Tjerk de Boer, PA3FWM is a closed-source application that supports SDRs based on audio cards, and also RTL-SDR. The last version comes with a HTML5 interface (older versions loaded a Java Applet into the browser). Users can independently tune the receiver (within the bandwidth of the I/Q signal). The bandwidth of the filter can be set from the web interface. The frequency scale may contain labels, which mark the stations. There is also squelch and automatic notch functionality, and a chat box where users can talk about the received signals. Using it requires only 80-300 kbps of network bandwidth at the client. It even runs on ARM single board computers (SBC) like the Raspberry Pi. There is also a special version that has very high bandwidth (covering the whole HF), and it uses GPU for DSP on the server.

ShinySDR [18] by Kevin Reid, AG6YO provides a HTML5 interface, and is released under GPL license. In Figure 14, we can see how it looks like in the browser. ShinySDR is implemented in the python programming language and uses GNU Radio for processing. It supports multiple demodulators at once. The current version is very smooth to use, and is quite practical for a private remote controlled station (as an access control feature, it requires a special key in the URL to connect, and it gives full control over the receiver hardware, including gain and center frequency setting). It only supports Google Chrome as a client, and any SDR hardware compatible with the *gr-osmosdr* GNU Radio blocks can be used as an input source.

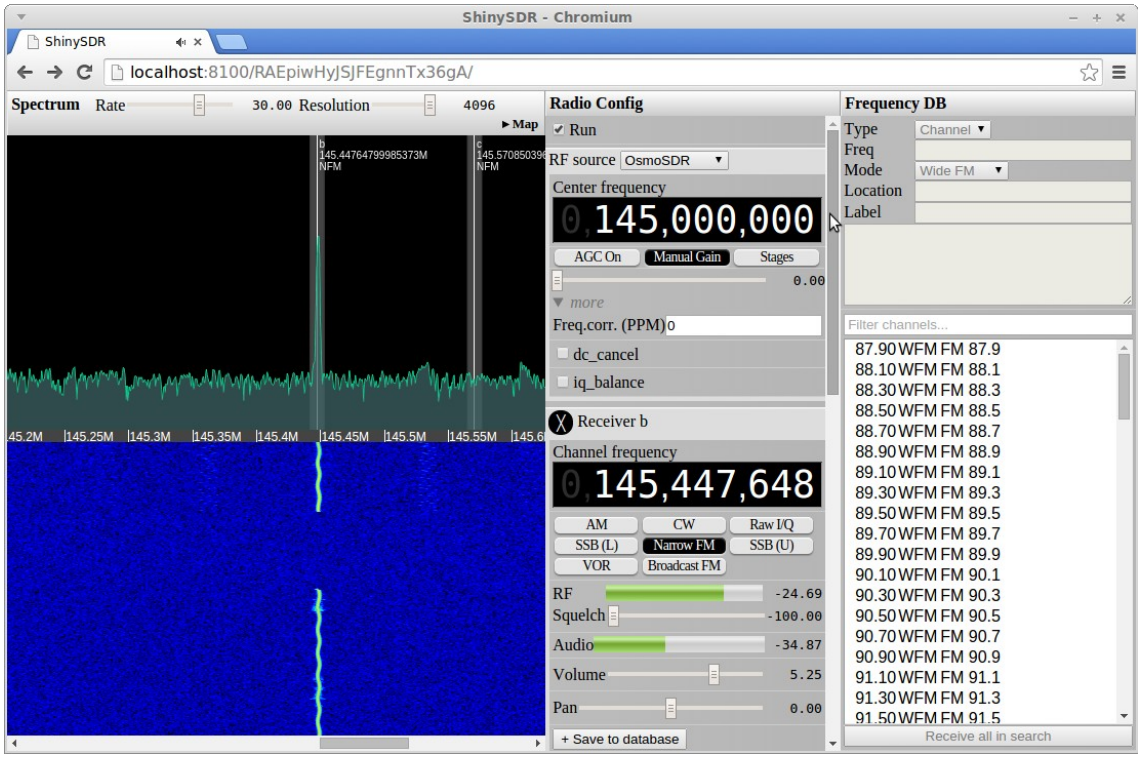

Figure 14: Screenshot of ShinySDR

WebRadio [19] by Mike Stirling is written in C++, and is released under AGPL license. It does not depend on any external DSP library, and also provides full access to the receiver (currently only RTL-SDR is supported). A screenshot of the application is shown in Figure 15.

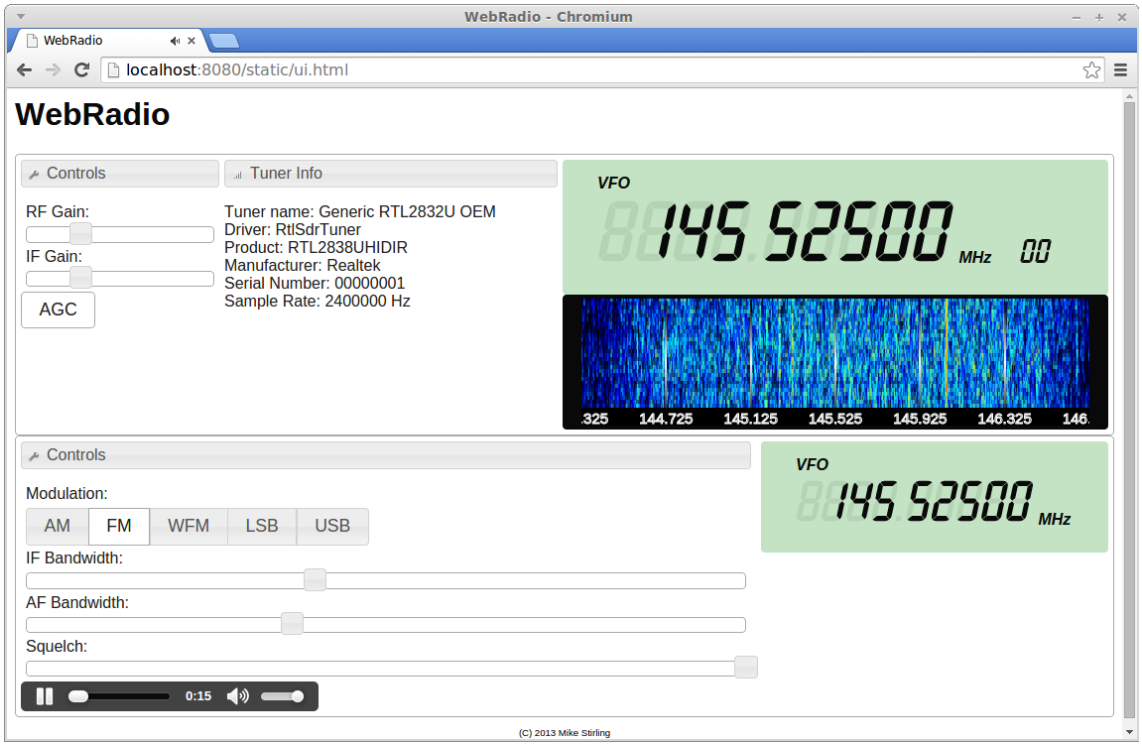

Figure 15: Screenshot of WebRadio

I really appreciate all of these projects, because they have given me many good ideas for my implementation, and also helped me making particular design choices.

5.2 Planning the structure

Figure 16 shows how OpenWebRX can be separated into several different parts.

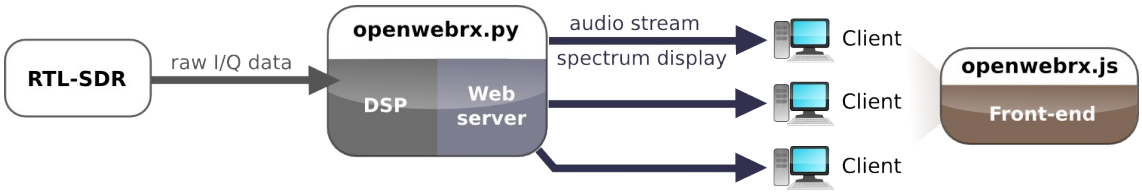

Figure 16: Parts of OpenWebRX

The application definitely needs a **web server** to serve a HyperText Markup Language (HTML) page content and its embedded media (images, Cascading Style Sheets – CSS, JavaScript) to the web browser, as these contain the UI for the receiver. I have decided

to implement the application as a standalone software instead of a server-side script or common gateway interface (CGI) executable for an existing web server, because it is easier to manage background threads in a standalone application. I have chosen the *python* programming language for implementing the server, as it has a lot of required functionality built-in (handling sockets and creating a web server is a matter of a few lines of code).

An important part of a web application is the **front-end** which consists of the already mentioned media elements. I implemented the waterfall display and audio streaming functions in JavaScript, which runs in the browser of the user. In addition, major browsers (including Google Chrome and Mozilla Firefox) compile JavaScript to machine code. Although JavaScript still does not reach the speed of native applications, it can run a lot faster than at the time it was only an interpreted language.

We also have to do **digital signal processing** on the server in order to generate the demodulated audio and the data for the waterfall diagram to be sent to the client's web browser. I originally wanted to use GNU Radio for DSP, as it provides a flexible library and framework for Software Defined Radio applications, and can be easily interfaced with python. However, later I have found that GNU Radio is hard to build from source code in some cases, and is advanced to compile on ARM SBCs (and it also takes a lot of time). Although OpenWebRX had a working version that utilized GNU Radio, I decided to write a DSP library myself and use it instead.

In the following parts of the thesis, I will give a detailed explanation on the server-side and client-side code structure of OpenWebRX, and also on the DSP algorithms used.

6 The server application

The main part of the server application resides in the *openwebrx.py* python script. It contains multiple classes, and imports some python modules that belong to the project. While being run, it starts several threads, most of which execute external processes for signal processing and distribution. Communication between the external processes is done using sockets. Between the external processes and the main program, it is done by FIFO (first in, first out) queues provided by the operating system. Figure 17 is to visualize inner operations of the server application.

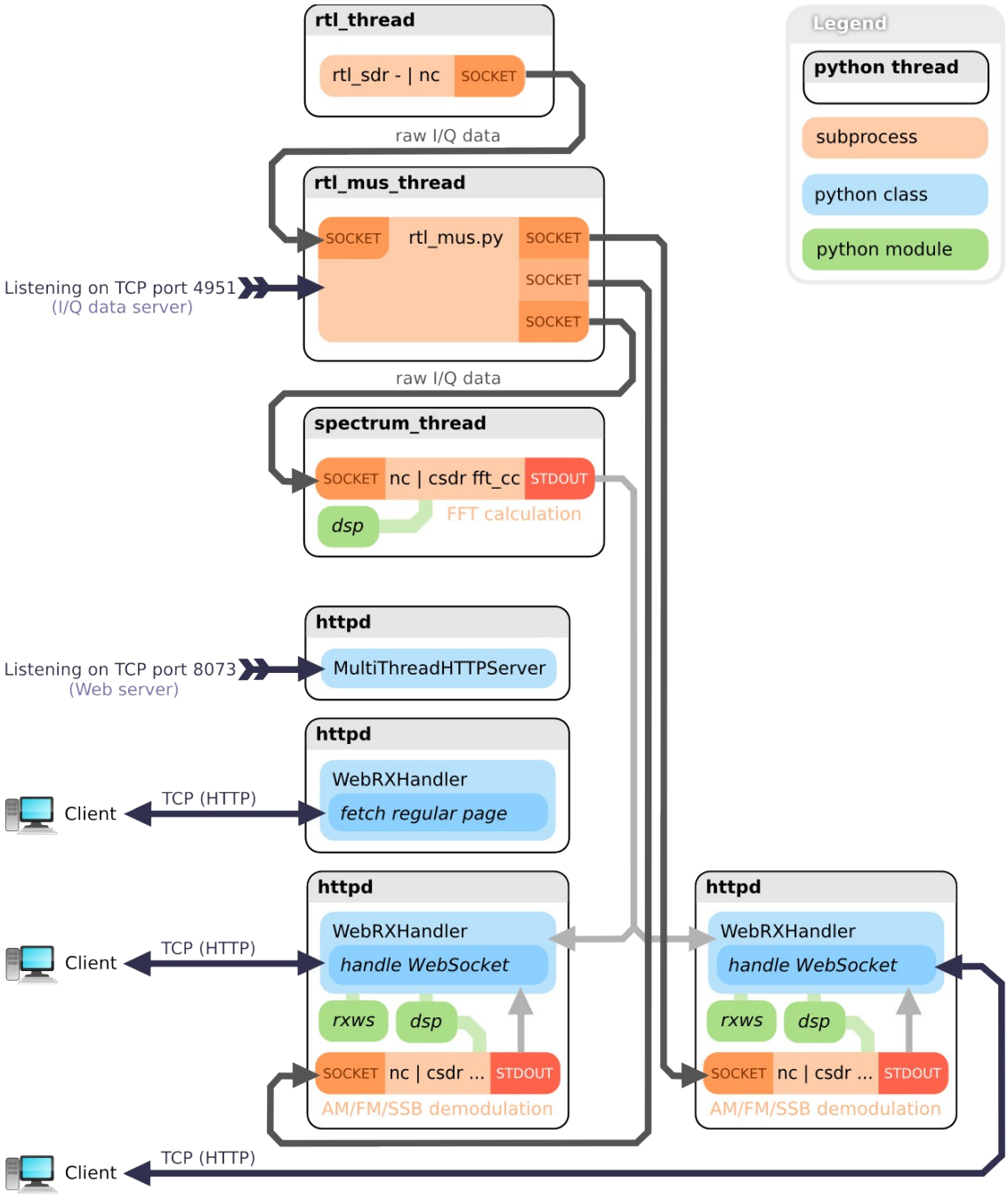

Figure 17: Simplified diagram of interconnections within the OpenWebRX server application

When the server application is started, it starts the **httpd** thread (web server), which instantiates the *MultiThreadHTTPServer* class, derived from the *HTTPServer* class from the built-in *BaseHTTPServer* module.

Every time a (HyperText Transfer Protocol) HTTP request is made from a client to the web server, *MultiThreadHTTPServer* creates a new instance of the *WebRXHandler* class, also running in a separate thread. *WebRXHandler* determines whether the request

targets regular files (containing the parts of the front-end in the *htdocs* subdirectory) or opening a WebSocket (if the path starts with */ws/*).

The *htdocs* directory also contains a file called *index.wrx*. It is the HTML template of the default page, which gets loaded if the browser requests the root URL '/'. It contains special tags that replaced dynamically by the web server (while processing the request). *%[CLIENT_ID]* is the identifier of the client, which is used for making the WebSocket connection. *%[WS_URL]* is the WebSocket base URL, containing the appropriate port (its use will allow OpenWebRX to be included into existing sites using a proxy script). The remaining tags are for providing information about the receiver, and they are replaced with values set in the configuration file (detailed in Table 4).

WebSockets are handed by the ***rxws*** python module, which contains all necessary functions to perform the initial handshake, assembling and parsing WebSocket frames, encoding and decoding the payload. It is based on RFC6455 [20], but it implements only the required subset of the protocol.

Demodulation is started when a client makes a successful WebSocket connection, after it has also performed a handshake with the OpenWebRX server. The server continuously sends the demodulated audio and the FFT of the signal (for the waterfall display) to the client. The client can also send messages to the server, to set:

- filter passband
- offset frequency (it is a parameter to the DDC, the offset between the actual receiver frequency of the client and the center frequency of the receiver hardware),
- the demodulator used (AM/NFM/SSB).

Table 3 contains examples of such communication.

Source	Example message	Notes
Server	`CLIENT DE SERVER openwebrx.py`	Handshake question
Client	`SERVER DE CLIENT openwebrx.js`	Handshake answer
Server	`MSG setup bandwidth=250000 center_freq=145500000 fft_size=4096 fft_fps=5`	Send receiver/DSP information for client initialization
Server	`FFT <an array of 4096 floating point values>`	FFT data
Server	`AUD <an array of 16-bit integer values>`	Audio data
Client	`SET offset_freq=-50000`	Change offset frequency

Client	`SET low_cut=-4000 high_cut=-400`	Change filter parameters
Client	`SET mod=SSB`	Change modulation

Table 3: The application layer protocol of the WebSocket connection used in OpenWebRX

The demodulator itself is an instance of the ***dsp_plugin*** class. Consequently, the signal processing part is based on plug-ins, although currently only the *plugins.dsp.csdr* plug-in exists. A new plug-in may be created later to add GNU Radio support again.

The default *csdr* plug-in creates a processing chain out of *csdr* processes, with the help of the shell application. Each client has a separate *dsp_plugin* instance, with a separate chain of processes. The output of each process in the chain is connected to the next one by a FIFO (provided by Linux). The input of the chain is the *netcat* (nc) command that creates a plain TCP connection to the I/Q data server provided by *rtl_mus.py* (explained in detail later). The output of the chain is read by the *csdr* plug-in and passed back to the appropriate *httpd* thread. An example chain for FM demodulation is shown below:

```
nc localhost 4951 | csdr convert_u8_f | \
csdr shift_addition_cc --fifo /tmp/openwebrx_pipe_3068669068_shift | \
csdr fir_decimate_cc 5 0.03 HAMMING | \
csdr bandpass_fir_fft_cc --fifo /tmp/openwebrx_pipe_3068669068_bpf \
0.0064 HAMMING | csdr fmdemod_quadri_cf | csdr limit_ff | \
csdr fractional_decimator_ff 1.13378684807 | \
csdr deemphasis_nfm_ff 44100 | csdr fastagc_ff | csdr convert_f_i16
```

The appropriate parameters for the *csdr* processes are determined by the *csdr* plug-in automatically, based on the configuration. Regarding *csdr* parametrization, the *README.md* in the git repository of *csdr* contains information.

The *csdr* processes read data from the standard input and write processed data to the standard output. Some *csdr* processes in the chain can be controlled without restarting the whole chain. These read from an additional FIFO, to receive control instructions. In the example above, when the user changes the frequency, a floating point number (converted to alphanumeric characters) and a newline is written to the pipe */tmp/openwebrx_pipe_3068669068_shift* in order to control the corresponding process started with the following command-line:

```
csdr shift_addition_cc --fifo /tmp/openwebrx_pipe_3068669068_shift
```

The ***rtl_mus_thread*** run the RTL Multi-User Server application (*rtl_mus.py*) as an external process. This *rtl_mus.py* has been taken from one of my older projects. It connects to a server that sends I/Q data over TCP, and distributes the data to multiple clients. I primarily wrote it because the *rtl_tcp* application that came with *librtlsdr* only supports one client at once, and I wanted to overcome this limitation. However, this application has turned out to be handy in distributing the I/Q data between the threads of OpenWebRX. If the appropriate port is opened and the access rights are given, any regular SDR software that support the *rtl_tcp* protocol can also be used to connect to port 4951 and receive the signal instead of using the web UI (however, this takes much more network bandwidth).

The *rtl_mus_thread* indeed gets its I/Q data from ***rtl_thread***. It runs a command that starts a server that provides I/Q samples on a given port (8888 by default). The command is generated based on receiver settings. An example of such command (for using RTL-SDR as an I/Q input source):

```
rtl_sdr -s 250000 -f 145525000 -g 0 - | nc -vvl 127.0.0.1 -p 8888
```

As *netcat* is used, it can serve only a single client (just as if *rtl_tcp* was used). This client will be *rtl_mus.py*, which further distributes the stream.

The *rtl_mus_thread* and *rtl_thread* are started when OpenWebRX starts, and the ***spectrum_thread*** is automatically started as well, to run in the background and repeatedly calculate the FFT of the signal (to provide data for the waterfall display with a given frame rate). It also uses the *dsp_plugin* to create a processing chain that retrieves the I/Q stream from *rtl_mus.py.* An example command-line for this processing chain (when all settings are default):

```
nc -vv localhost 4951 | csdr convert_u8_f | \
csdr fft_cc 4096 27777 | csdr logpower_cf -70
```

Configuration options for OpenWebRX are stored in the *config_webrx.py* and *config_rtl.py* files. The latter is the configuration for *rtl_mus* bundled with OpenWebRX, and its safe defaults are not needed to be changed under normal circumstances. The file *confg_webrx.py* contains several configuration settings regarding server and receiver configuration, and displayed information. These are listed in Table 4.

Configuration option	Description
`web_port`	The default port the web server listens on.
`server_hostname`	The hostname of machine running OpenWebRX. (It is used for generating *%[WS_URL]*, and front-end will fail to load if it is set incorrectly.
`receiver_name`	Replaces *%[RX_TITLE] in .wrx* files
`receiver_location`	Replaces *%[RX_LOC]* in *.wrx* files
`receiver_qra`	Replaces *%[RX_QRA]* in *.wrx* files
`receiver_asl`	Replaces *%[RX_ASL]* in *.wrx* files
`receiver_ant`	Replaces *%[RX_ANT]* in *.wrx* files
`receiver_device`	Replaces *%[RX_DEVICE]* in *.wrx* files
`receiver_admin`	Replaces *%[RX_ADMIN]* in *.wrx* files
`receiver_gps`	Replaces *%[RX_GPS]* in *.wrx* files
`photo_height`	Replaces *%[RX_PHOTO_HEIGHT]* in *.wrx* files
`photo_title`	Replaces *%[RX_PHOTO_TITLE]* in *.wrx* files
`photo_desc`	Replaces *%[RX_PHOTO_DESC]* in *.wrx* files
`dsp_plugin`	Determines the DSP plug-in to be used (currently only a *csdr* plug-in is available).
`fft_fps`	Determines the frame rate to update the waterfall display at the client.
`fft_size`	Determines the resolution of the FFT.
`samp_rate`	Sets the sample rate of the receiver hardware.
`center_freq`	Sets the center frequency of the receiver hardware.
`rf_gain`	Sets the gain of the receiver hardware (in dB).
`start_rtl_thread`	If this boolean value is set to True, OpenWebRX starts the *rtl_thread.*
`start_rtl_command`	The command to be run in *rtl_thread*.

Table 4: Configuration options in config_webrx.py

7 The client front-end

The front-end contains the files required for running the web application in the browser. These files are contained under the *htdocs* subdirectory of OpenWebRX, and the web server sends them to clients on request.

The file *index.wrx* is the HTML layout for the web GUI of OpenWebRX. As already noted above, it contains some special tags like *%[WS_URL]* that the web server replaces with actual values on every request. Images, style sheet and script files are referenced from within *index.wrx*.

The directory *htdocs/gfx* contains all the graphics elements used in the user interface. These were all created using open-source graphics editing tools (Inkscape and GIMP). In Figure 18, we can see how the graphics design was created with such software.

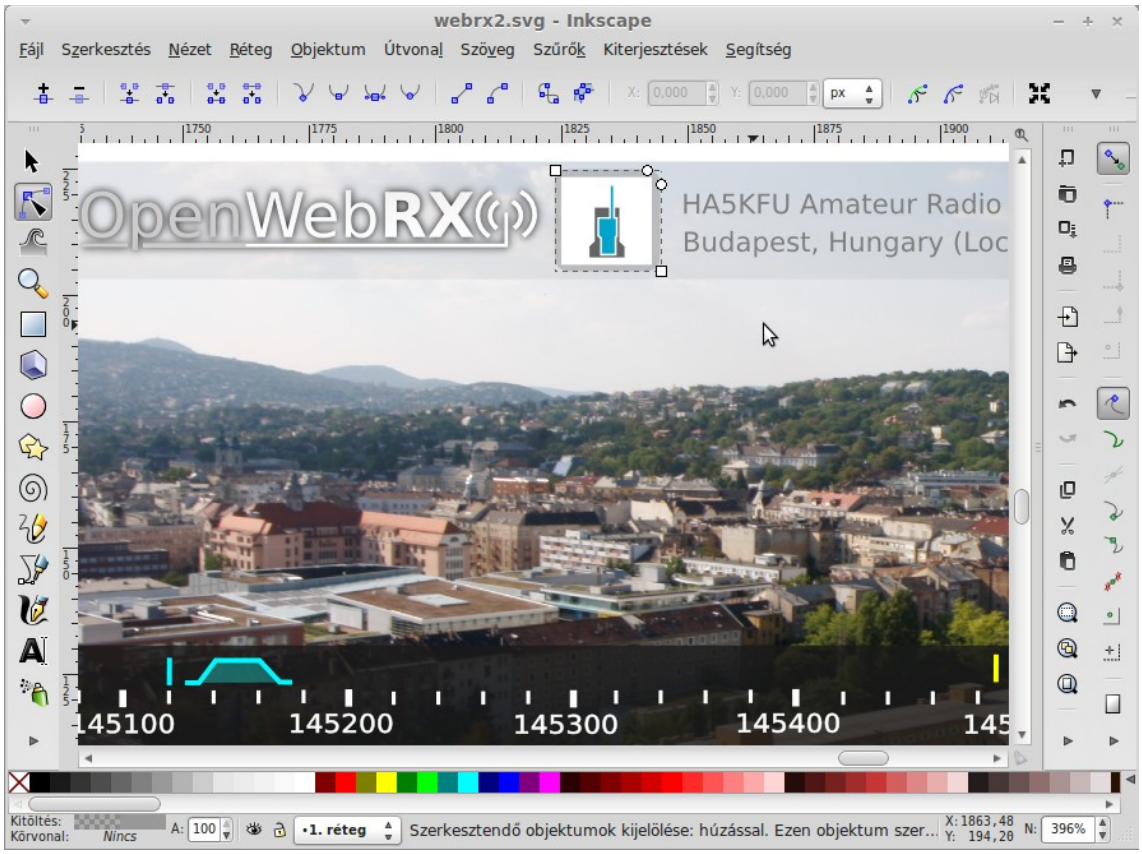

Figure 18: Editing design in open-source vector graphics tool, Inkscape

The *openwebrx.css* file is written in Cascading Style Sheets (CSS) language, to describe the look of the HTML elements on the front page. It also contains a reference to the CSS3 web font under the directory *htdocs/gfx/font-expletus-sans.*

The *openwebrx.js* file is written in JavaScript language, and it is the heart of the OpenWebRX front-end, as it provides all interactive behavior of the web page. Its operation will be described in the next section.

The user is redirected to *upgrade.html* if the web browser application (determined by the User-Agent field in the HTTP request) is not supported (currently only the newest versions of Mozilla Firefox and Google Chrome are supported). On this page, a warning message is shown to the user about the unsupported browser, but the user can still select to continue to main page of the receiver.

The *favicon.ico* is the icon shown on the browser tab next to the title of the page.

Some users have reported that they could connect to OpenWebRX and use it from mobile devices (tablets running Android). I could not make tests on Android, because I did not have a device for testing. The built-in web browser that some older Android versions ship with cannot be used, because it lacks some required HTML5 features. In this case Google Chrome from Android should be downloaded. It is also known that some features (like zooming the waterfall diagram) currently do not work on mobile devices.

7.1 JavaScript, the heart of the front-end

In general, OpenWebRX was designed to work without the need of downloading additional JavaScript libraries. The only JavaScript file is *openwebrx.js* that does everything that we need in this particular application.

When *index.wrx* is loaded, the function *openwebrx_init* is called in *openwebrx.js*. It initializes UI elements (e. g. panels), and opens the WebSocket to the server. (Using WebSockets is the only easy way to do continuous two-way communication between the web browser and the server.)

After a handshake process, the server sends the parameters of the receiver (the center frequency and the sampling rate) and preconfigured settings of the waterfall diagram (FFT size, FFT frame rate). The waterfall display and the frequency scale is initialized accordingly.

The first three characters of the messages indicate the type of the message (see table 3),

and the fourth byte is a space character. We can get the payload from the message if we skip the first 4 bytes. These rules apply to all messages, except the handshake messages.

When the script receives a message starting with the letters ‘FFT’ over WebSocket, it uses the payload to draw a new line on the waterfall display. The payload contains values of relative received signal strength in dB, corresponding to specific frequencies within the bandwidth of the I/Q signal, with a given resolution.

The waterfall display itself is made of <canvas> elements, that can be used for drawing freely from JavaScript. When a canvas (with a height of 200 pixels) gets filled, a new canvas is created. When new FFT data is received, the new line is drawn on the topmost canvas, and all the canvases get shifted one pixel downwards. Canvases do not get removed (unless the client closes the page), they remain in memory even if they are shifted out of the screen. There is a scroll bar on the right edge of the window to view the canvases that have moved to the off-screen area. Scrolling back lets us examine how the RF spectrum changed since the page has been opened.

Internally, the width of the canvas equals to FFT size, and the canvas is shrunk to fit the page (or stretched for the actual zoom level). It seems that modern browsers can deal with this, however, there are two issues: panning the waterfall diagram tends to lag on Mozilla Firefox (although it does not lag in Google Chrome), and the browser consumes a lot of memory (as it may store a really history of the waterfall diagram, if the browser tab is left open).

When the script receives a message starting with the letters ‘AUD’, it initializes the Web Audio API (if it has not been already initialized), and prepares the payload to be output to the sound card. Audio is received as an uncompressed, raw stream of 16-bit signed integers at a sampling rate of 44100 sps, because this is the default (and so far, the only available) output sample rate for Web Audio API in the supported browsers.

Web Audio API uses nodes with specific functions connected to each other (with a similar concept to GNU Radio blocks) to generate, process and output audio. In my application, a so-called ‘script processor node’ (initiated by *createJavascriptNode* / *createScriptProcessor* methods of the audio context object) is connected to the destination node (the sound card input). The *onaudioprocess* event handler is called when the script processor node has to output a new block of data. As the size of the

WebSocket message payload and the buffer size of the *onaudioprocess* handler may differ, the received audio data is broken into chunks that are exactly of the size that *onaudioprocess* requires.

Some HTML elements in *index.wrx* have event handlers set to call a function in *openwebrx.js*:

- clicking the buttons for demodulator selection call the *demodulator_analog_replace* function, passing the modulation as a parameter,
- clicking on the receiver information in the top bar toggles the display of the information frame.

In *openwebrx.js* there are also functions for making simple animations. Currently these are only shown when the user toggles the receiver information frame visibility, by clicking on the arrow in the top bar.

8 Digital signal processing in OpenWebRX

8.1 System architecture

To demodulate the signal selected by the user, and send it to the browser as an audio stream, we need to perform signal processing tasks. In this case, the input to the system is the RF signal after quadrature downconversion and sampling, as 8-bit unsigned, complex I/Q data stream. It is coming from the receiver front-end (an RTL-SDR in my application).

I have implemented a standalone DSP library called *libcsdr* that contains all the necessary functions for demodulation. The library comes with a command-line program, *csdr,* which is used by OpenWebRX. However, the design philosophy was to write a library that is not tied to my application and other projects may use it independently.

In the first part of this chapter, I write about general design concerns of *libcsdr*, and in the second part I give a detailed description of the algorithms used.

To achieve demodulation, several different algorithms have to be applied on the input signal, one after the other. I call the sequence of processing these algorithms a ‘DSP chain’. The command-line tool *csdr* lets us build and run limited, but sufficient DSP chains easily.

Figure 19 below illustrates the demodulation process from the DSP aspect.

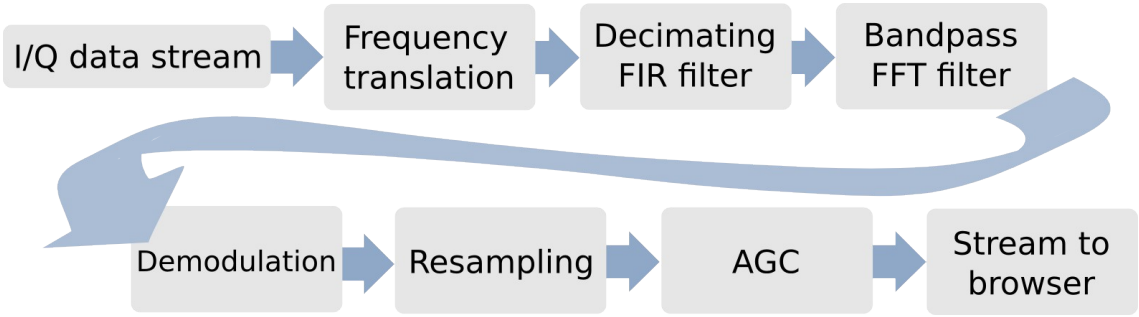

Figure 19: Simplified DSP chain for demodulation

The first step is channelization: to select the signal to receive, we apply *frequency translation* to shift its center to DC in the frequency domain, and we decrease the sample rate with a *decimating FIR filter*.

Now we can apply a *band-pass FFT filter.* The passband of this filter can be selected on

the web user interface. As it is applied after decimation, it can have quite low transition bandwidth without the need of too much computational resources. We have two different filters after each other, because much lower transition bandwidth can be achieved on the decimated signal with less computation. For example, if we want to demodulate CW signals, the passband should be only several hundred Hertz, and this calls for a filter transition bandwidth in this order. If we wanted to design a FIR filter that has a transition bandwidth of 100 Hz running on a signal sampled at 2.4 Msps, we would end up in 96000 coefficients. Such a long filter cannot be effectively processed on today's CPUs.

The demodulator converts our complex signal to a real-valued audio signal. The *decimating FIR filter* works with an integral decimation rate, and its output sample rate does not necessarily match the 44100 Hz sampling rate required by the client (the Web Audio API in Google Chrome did not support any other sampling rate at the time the web front-end was implemented). To solve this problem, a *resampler* is used.

The *Automatic Gain Control (AGC)* tries to keep the signal level constant. The output of the DSP chain is streamed to the web browser of the user.

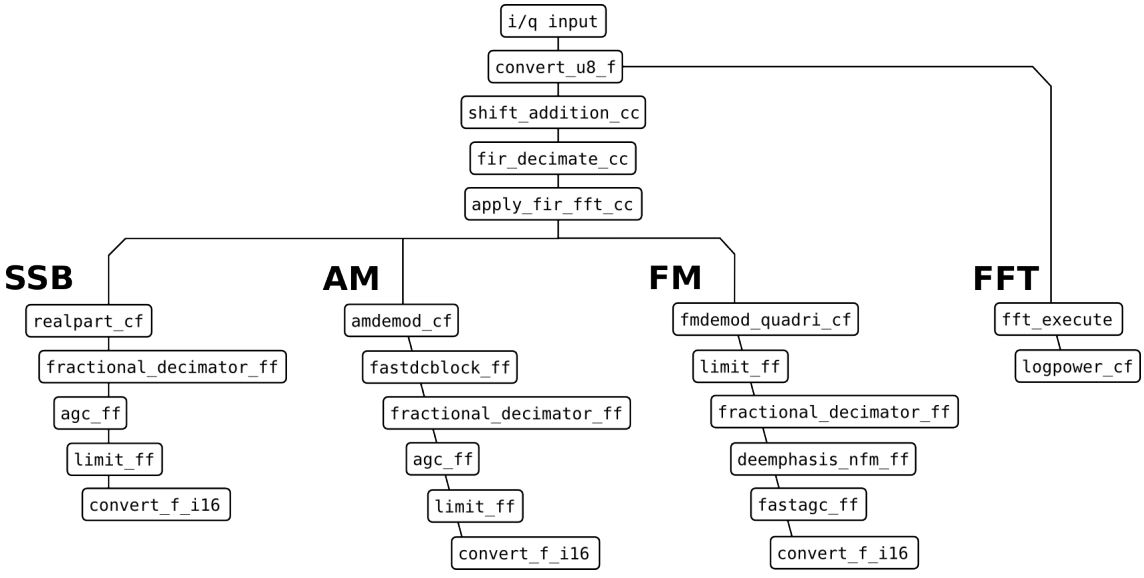

Figure 20: DSP chains in detail: exact libcsdr functions called (through csdr wrapper executable) when using OpenWebRX

Figure 20 shows what exact functions in *libcsdr* are used for different modulations. These chains are fine-tuned for the best result. The FFT chain is designed to provide the spectrum display for the user.

8.2 Software design for performance

I have chosen to implement the DSP functions in C, putting all the DSP processing in a separate library. Although the web interface was implemented in *python*, the default *python* interpreter (usually referenced as *CPython*) cannot provide the speed required for DSP operations.

It seems trivial that an interpreted language is not suitable for DSP, but for testing the possibilities, I have made experiments with WFM demodulation written purely in *python.* The target hardware was a PC equipped with an Intel T4200 Dual-Core CPU clocked at 2.00 GHz, and I could not achieve continuous demodulation and playback, despite having spent some time on optimizations. I am also aware that there is an other implementation of *python* called *PyPy*, which can compile *python* code to machine code at runtime, but the C language is much closer to the hardware, so the algorithms can be optimized better, and the library can be more easily ported if later required.

In Software Defined Radio, if the architecture is ‘digital IF’ or ‘digital RF’, we usually work on signals with high sample rate, at least before channelization. On several systems, this fact calls out for using high performance computing (HPC) solutions for signal processing. These include using the computational power of the graphics card in the PC (general purpose GPU programming – GPGPU), or using FPGA to achieve highly parallel processing, instead of using a general purpose CPU that executes instructions one after the other. DSP ICs are also a special type of CPUs, having special architecture and instruction set for performing signal processing faster.

To speed up computations, designers of general purpose CPU architectures have also started to include special instructions to do computation on a vector of data (on multiple registers in parallel). This way of parallelism is called ‘single instruction, multiple data’ (SIMD), and is usually implemented as an extension for the base instruction set of a CPU. For Intel CPUs, the technologies named MMX and Streaming SIMD Extensions (SSE) – the latter has multiple versions – provide SIMD extensions. 3DNow! is a similar feature for AMD CPUs, but now outdated. The latest development is Advanced Vector Extensions (AVX) proposed by Intel and AMD. In ARM CPUs such feature is called NEON, and is present in the Cortex-A8 processor line.

To take full advantage of SIMD extensions, one would have to code assembly (and, for

compatibility, include a version of the same algorithm that does not use SIMD). However, some C compilers support a so-called auto-vectorization feature, which means that sufficiently structured loops can be unrolled and the operations within the loop body can be compiled to SIMD instructions. Usually only very simple loops can be optimized with this technique, and there are also several preconditions for the successful optimization. (For example, while filling an array, the result of the previous loop execution cannot be referenced, complicated control flow should be avoided, etc.)

I have implemented *libcsdr* in a way that some DSP functions can be optimized by the *gcc* auto-vectorizer. (I used *gcc* version 4.8.2 in my tests.) I have been optimizing the code mainly for SSE, and partially for ARM NEON. (There are differences between the vectorization result on the different SIMD architectures.) Compiling on an Intel CPU, the *Makefile* automatically selects the sufficient switches for *gcc* based on the virtual file */proc/cpuinfo*, so it can handle that different CPUs have support for different versions of SSE.

I also have written a python script, *parsevect* to parse the log output of the auto-vectorizer of *gcc*, and provide an easily readable list of loops and the vectorization result, with color indication (green for success, red for failure). This script gets called every time when the library is built with *GNU make*, providing an up-to-date feedback about current vectorization status of algorithms, and makes code optimization easier for the developer. The script *parsevect* reads the corresponding source files as well, in which special comments (starting with the characters: ‘//@’) can be placed to tag loops. In the output, this is displayed with the loop vectorization result, and this helps to easily identify loops that need restructuring for auto-vectorization (as seen on Figure 21).

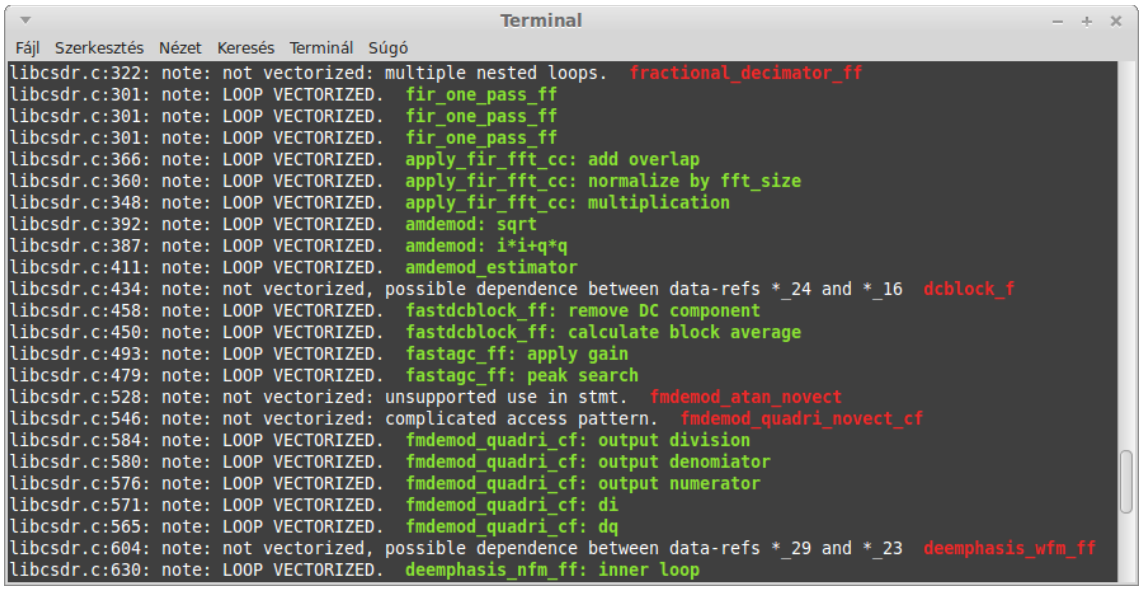

Figure 21: partial output of parsevect (while compiling csdr on an Intel Core i7 CPU)

It is also important to note that some algorithms are impossible to optimize this way (e. g. IIR filters, the output of which depends on the previous output), and some are executed only once (e. g. filter design functions), so are unnecessary to get optimized. The latter are tagged with '//@@' in the source code, and displayed in yellow in the output. However, loops listed in red and yellow will also run, but will not be optimized for speed.

Some of my functions in *libcsdr* depend on the FFT library *libfftw3*. This library provides highly optimized versions of the Fourier transform for several SIMD architectures, including SSE and NEON.

8.3 Choice of data types

Regarding implementation, it is important to decide whether it is optimal to use fixed point or floating point representation of the signal in a given application. Fixed point variables are used to store integers, and the gaps between adjacent numbers are exactly the same. Floating point representation basically consists of a mantissa multiplied by ten raised to an exponent, and the gaps between adjacent numbers vary over the represented range, but this range is usually very high compared to fixed point numbers. While doing calculations, round-off errors appear as noise on the signal, but this is a problem with both representations. [21]

I have chosen to use single-precision floating-point representation for internal

calculations in my DSP routines. This precisely means using the *float* data type in C, which is a 32-bit number with 23-bit mantissa and 8-bit exponent, and a sign bit (on the x86 and ARM Cortex-A architectures). Floating point is easier to use for DSP, as we do not have to be afraid of getting out of the representable range and ending up in invalid results because of overflows.

It is also important to note that old CPUs lacked support for floating point algebra, which also made fixed point DSP attractive. Nowadays CPUs have floating point units, and they even support SIMD on floating point numbers.

Although my routines operate on floating point data, the input signal from RTL-SDR and the audio output is necessarily fixed point, so I had to write data conversion routines, which are listed in Table 5.

`void` **`convert_u8_f`**
`(unsigned char* input,`
`float* output,`
`int input_size)`
	Converts an array of unsigned 8-bit values to single-precision floating-point.

`void` **`convert_f_u8`**
`(float* input,`
`unsigned char* output,`
`int input_size)`
	Converts an array of single-precision floating-point values to unsigned 8-bit.

`void` **`convert_i16_f`**
`(short* input,`
`float* output,`
`int input_size)`
	Converts an array of signed 16-bit values to single-precision floating-point.

`void` **`convert_f_i16`**
`(float* input,`
`short* output,`
`int input_size)`
	Converts an array of single-precision floating-point values to signed 16-bit.

Table 5: Summary of data conversion functions in libcsdr

8.4 Function and parameter naming conventions

While designing the API, I have made some decisions on naming conventions and common parameters. All functions have one input and one output buffer. These are called *input* and *output*, and there is also a parameter *input_size*, the size of the input buffer. The output buffer should be allocated by the caller, and its size should also be *input_size* unless not stated otherwise in the comments.

Abbreviations in function name endings give a hint on the *input* and the *output* data types. A short list of abbreviations used:

- `f: float`
- `c: complexf`
- `i16: short`
- `u8: unsigned char`

The data type *complexf* is a *struct* that consists of two *float* values. Defining an own complex type helped to get successful auto-vectorization for operations on complex numbers.

8.5 Testing and evaluation

To ensure that the implemented algorithms work as expected, I made test benches in GNU Radio Companion (GRC). With GRC, complex DSP processing flow-graphs can be created, but mostly I utilized the data visualization features (scope, spectrum and waterfall plots on wxWidgets GUI), simulated input sources (sine wave generator), the block implementing the algorithm under test in GNU Radio, and some custom blocks to connect *libcsdr* with GNU Radio. The latter were ‘Execute External Process’ and ‘Execute External Process Sink’ blocks, which allow us to execute a command-line program and get its standard input and output connected to the flow-graph in GRC. Executing *csdr* within these blocks helped me to analyze the behavior of my algorithms and compare them against the built-in ones in GNU Radio.

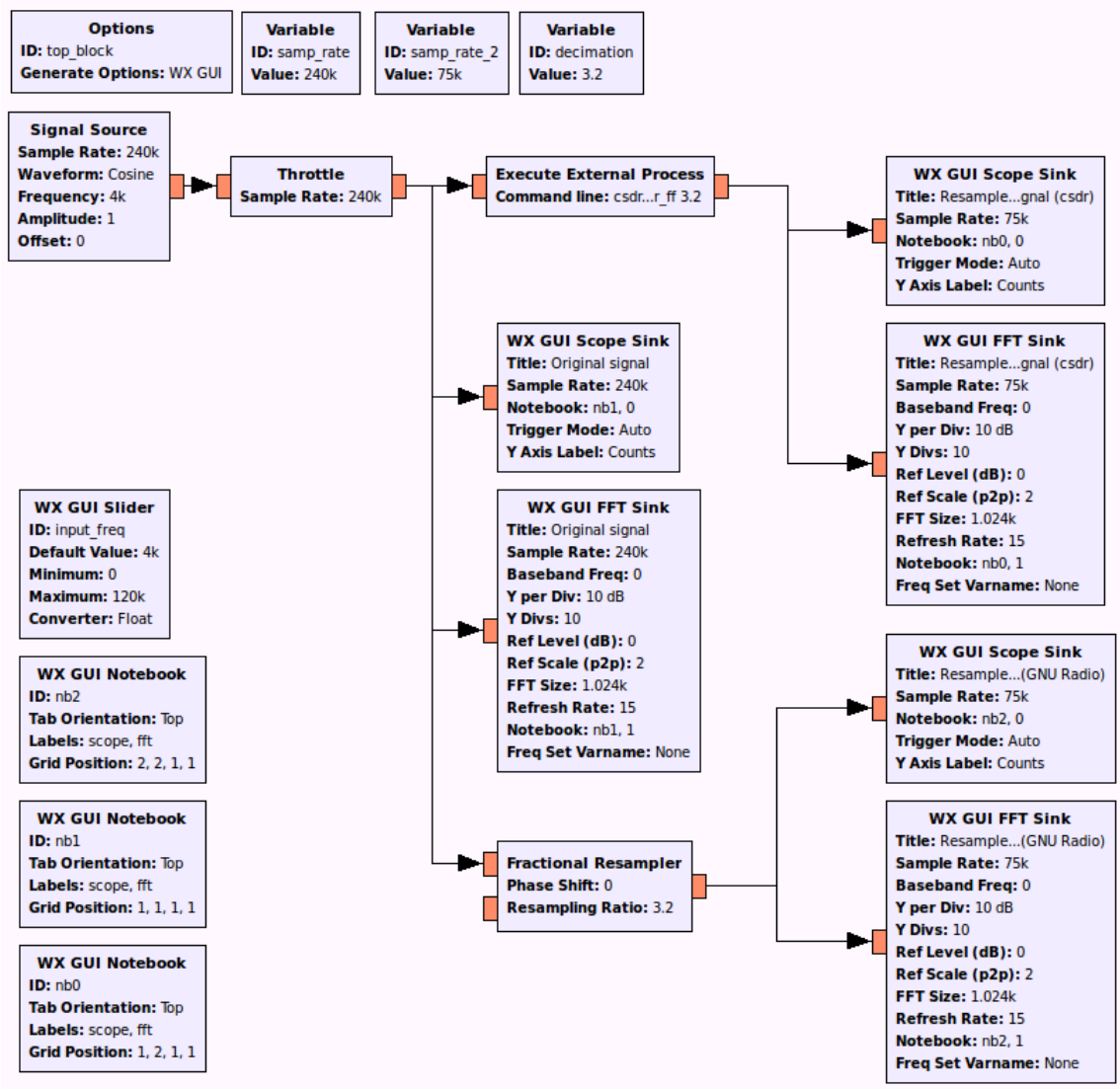

Figure 22: GRC test bench for the function fractional_decimator_ff in libcsdr

In the test benches (just like the one on Figure 22), I have executed *csdr* with various parameters, and viewed the output of my algorithms with different input signals. Due to length restrictions on this thesis, I cannot write about every test taken during the development, but I made a list of common requirements checked:

- the output signal had to be continuous in scope view (so no buffering errors occurred),
- the output could not contain unwanted harmonics, or only at a sufficient level compared to output of built-in algorithms in GNU Radio,
- output had to be at an expected level (for filters, output level had to match the previously calculated filter envelope).

9 Channelization and filters

After general implementation concerns, we continue with the description of the algorithms used. The software specification states that it should be capable of decoding different signals to different clients from a wideband input. For this purpose, a digital down-converter (DDC) is required before the demodulation stage, which is illustrated in Figure 23. It shifts the desired signal to the center in the frequency domain, applies a filter and also decreases the samples to the minimally required number for the bandwidth of the chosen signal. [22]

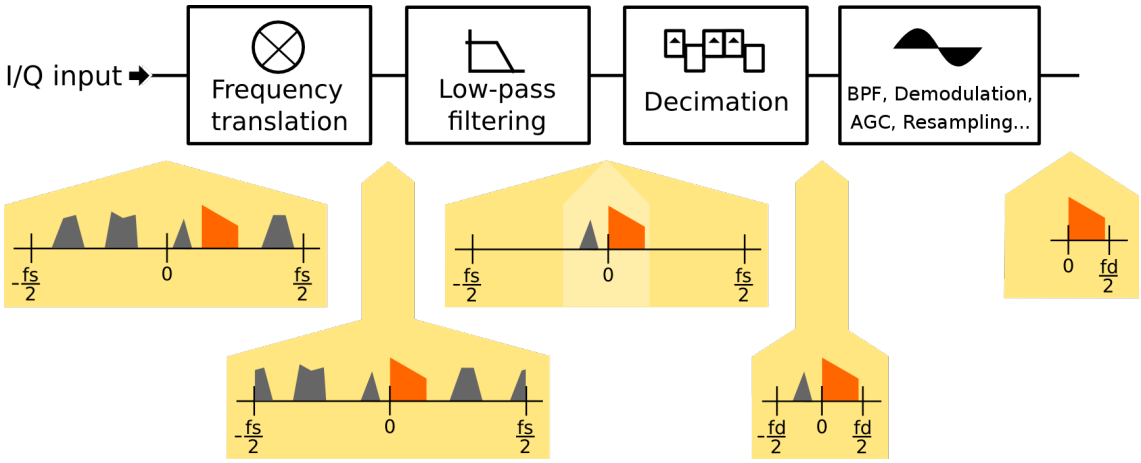

Figure 23: channelizing for SSB demodulation

In the following part, the algorithms required for channelization and filter processing are described.

9.1 Frequency translation

If we have a complex signal, we can shift it by a frequency f in the frequency domain, if we multiply it with (2).

$$e^{j2\pi f t} = \cos(2\pi f t) + j \cdot \sin(2\pi f t) \tag{2}$$

As we work on a sampled signal, we need a numerical controlled oscillator (NCO) to generate discrete time sine and cosine signals. There are different algorithms for this:

- use *sin* and *cos* functions from *libmath* (their implementation is compiler and architecture specific, and may not be the most optimal),
- use a lookup table store values of $\sin(x)$ and $\cos(x)$ functions in advance with

a given resolution, and easily look up later (higher accuracy implementation has higher memory footprint),

- CORDIC (the most effective for FPGAs and ASIC, but takes many steps on a CPU),
- use trigonometric addition formulas. [23]

The first and the last method was implemented (functions are summarized in Table 6). The last algorithm consists of the following steps:

1. Take the sine and cosine of the phase step between samples (3).

$$\begin{aligned} d_{\sin} &= \sin(\Delta\varphi) \\ d_{\cos} &= \cos(\Delta\varphi) \end{aligned} \tag{3}$$

2. Take the sine and cosine of the starting phase (4).

$$\begin{aligned} s_{\sin}[0] &= \sin(\varphi_0) \\ s_{\cos}[0] &= \cos(\varphi_0) \end{aligned} \tag{4}$$

3. Apply the following trigonometric addition formulas to calculate sine and cosine step by step (5).

$$\begin{aligned} s_{\cos}[i] &= \cos(\varphi_{i-1} + \Delta\varphi) = \cos(\varphi_{i-1}) \cdot \cos(\Delta\varphi) - \sin(\varphi_{i-1}) \cdot \sin(\Delta\varphi) = \\ &= s_{\cos}[i-1] \cdot d_{\cos} - s_{\sin}[i-1] \cdot d_{\sin} \\ s_{\sin}[i] &= \sin(\varphi_{i-1} + \Delta\varphi) = \sin(\varphi_{i-1}) \cdot \cos(\Delta\varphi) + \cos(\varphi_{i-1}) \cdot \sin(\Delta\varphi) = \\ &= s_{\sin}[i-1] \cdot d_{\cos} + s_{\cos}[i-1] \cdot d_{\sin} \end{aligned} \tag{5}$$

It requires only a small number of operations so is expected to be faster than using *libmath*, but there are two drawbacks when using this method:

- the floating point rounding errors increase with *n*,
- although it can be optimized with SIMD manually, the auto-vectorizer of GCC cannot handle it.

The rounding errors can be overcome by re-initializing the $s_{\sin}[0]$ and $s_{\cos}[0]$ values according to a calculated starting phase on every block of data. Tests showed that if we reinitialize on every < 10000 samples, this error is sufficiently low, as shown in Figure 23.

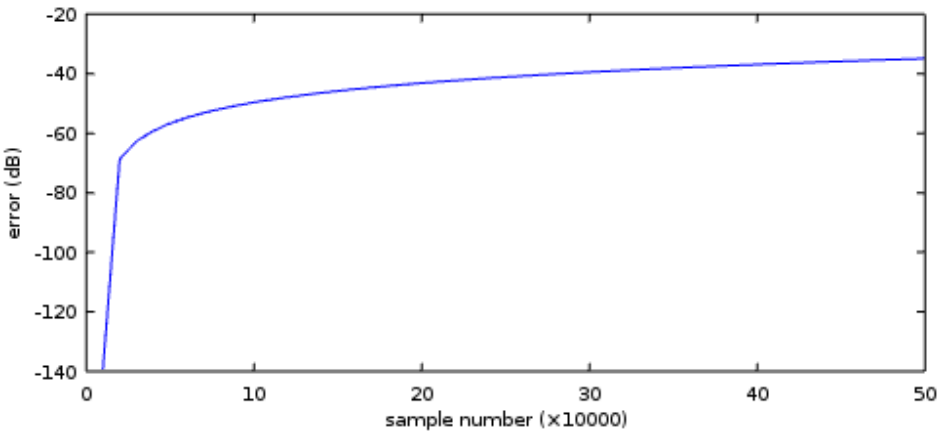

Figure 24: Error due to floating point rounding operations without re-initialization on every block (calculated using shift_addition_cc_test).

I have made a quick measurement on the speed of the two algorithms implemented, and it was revealed that calculation using addition formulas is about 4 times faster than using *libmath*. I have made measurements with the *time* utility in Linux, running both algorithms on the same number of input samples.

`float` **`shift_math_cc`**
`(complexf *input,`
`complexf* output,`
`int input_size,`
`float rate,`
`float starting_phase)`
	Frequency translation on complex signal. NCO is implemented using the built-in *libmath*.
	The frequency shift *rate* is in proportion to the sampling rate, in the range [-0.5, 0.5].

`float` **`shift_addition_cc`**
`(complexf *input,`
`complexf* output,`
`int input_size,`
`shift_addition_data_t d,`
`float starting_phase)`
	Frequency translation on complex signal. NCO is implemented using the trigonometric addition formulas. It is faster, but less accurate.
	The parameter *d* should be initialized with *shift_addition_init*.
	The returned value has to be passed as the *starting_phase* parameter, the next time the function is called on the same input stream.

`shift_addition_data_t`
`shift_addition_init`
`(float rate)`
	Its return value should be passed to *shift_addition_cc*, as parameter *d*.
	Its only parameter is the frequency shift *rate*, already explained at *shift_math_cc*.

`void` **`shift_addition_cc_test`** `(shift_addition_data_t d)`	Compares the two functions above, calculating the error between *shift_math_cc* and the less accurate *shift_addition_cc*. Its

	output can be piped into *GNU octave* for drawing a plot.

Table 6: Summary of frequency translation functions in libcsdr

9.2 Filter design

The purpose of filtering is to weigh specific signal components in the frequency domain. For example, if we want to receive a continuous wave (CW) signal with an amateur radio receiver, a good bandpass IF filter is required to suppress neighboring signals – some of which may even be more powerful than the signal to be selected.

Regarding digital filters, they are usually classified as finite impulse response (FIR) filters or infinite impulse response (IIR) filters. IIR filters have output feedback, so unlike FIR filters, they cannot be optimized with SIMD.

FIR filters also have a special subtype called cascaded integrator–comb (CIC) filters, that are more economic than standard FIR filters for doing decimation with a factor over 10.

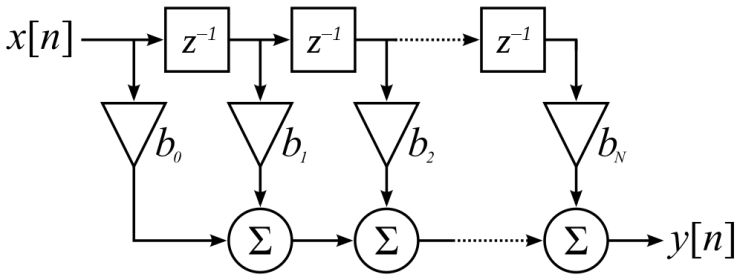

Figure 25: Finite impulse response filter realization for discrete time [23]

The output of a FIR filter of order N is a weighted sum of the last N items in the input sequence. It can be expressed as (6), what is also illustrated on Figure 25.

$$y[n]=h[0]x[n]+h[1]x[n-1]+...+h[N]x[n-N]=\sum_{i=0}^{N} h[i]x[n-i] \qquad (6)$$

In contrast, an IIR filter can be expressed as (7).

$$y[n]=\frac{1}{a[0]}(b[0]x[n]+b[1]x[n-1]+...+b[P]x[n-P]-a[1]y[n-1]-$$
$$-a[2]y[n-2]-...-a[Q]y[n-Q])=\frac{1}{a[0]}\left(\sum_{i=0}^{P} b[i]x[n-i]-\sum_{i=1}^{Q} a[i]y[n-i]\right) \quad (7)$$

In the equations above $y[i]$ is the output signal, $x[i]$ is the input signal, $h[i]$ are the FIR filter coefficients, *N* is the FIR filter order, $b[i]$ and $a[i]$ are the feed-forward and feedback IIR filter coefficients with the corresponding orders of *P* and *Q*.

I have chosen to implement FIR filter design and processing functions, as they provided good results enough and could be easily optimized. In my application, filtering is required at several stages of the processing chain:

- A low-pass filter with real taps is used in the DDC and the resampler to remove signal components above the Nyquist frequency before decimation.
- A band-pass filter with complex taps is used before the demodulator. (It can be controlled over the web interface.)
- A de-emphasis filter is used after FM demodulation.
- A DC blocking filter is used after AM demodulation.
- A loop filter is used in the AGC.

When the user changes the filter bandwidth on the web user interface, the band-pass filter has to be redesigned on the fly, so I had to implement filter design algorithms in my DSP library (as listed in Table 7).

There are multiple methods for designing FIR filters. The Parks–McClellan algorithm (a variation of the Remez algorithm especially tailored for generating FIR filters) is quite common, but the original implementation is very hard to follow. I have decided to use the windowed FIR filter design method instead.

Convolving an input signal with a filter kernel given by the *sinc* function, we get a perfect low-pass filter. The coefficients are given by (8), where f_c is the cutoff frequency (relative to the sampling frequency).

$$h[i]=\frac{\sin(2\pi f_c i)}{\pi i} \quad (8)$$

However, as our filter is of finite length, we have to truncate this function, but it comes with undesirable side effects in the frequency domain (ripple in the passband and bad stopband attenuation).

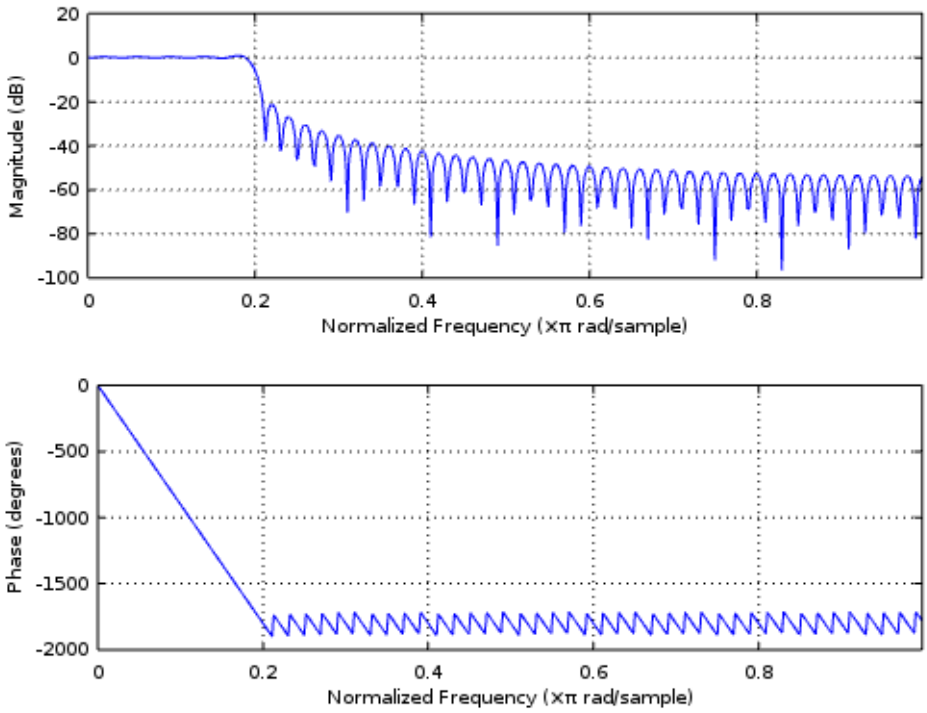

Figure 26: Sinc filter without windowing
(generated with firdes_lowpass_f)

To overcome this problem, we multiply our filter kernel with a so-called window function. There are several window functions available: the Bartlett, raised cosine, Hamming and Blackman windows are the most common. (The 'rectangular window' or 'boxcar window' are synonyms for a dummy window to indicate that no windowing actually takes place.)

The formula for the Blackman window is (9).

$$w[i] = 0.42 - 0.5\cos\left(\frac{2\pi i}{M}\right) + 0.08\cos\left(\frac{4\pi i}{M}\right) \tag{9}$$

The formula for the Hamming window is (10).

$$w[i] = 0.54 - 0.46\cos\left(\frac{2\pi i}{M}\right) \tag{10}$$

Comparing these two, the Blackman has better stopband attenuation and lower passband

ripple, but the Hamming has faster roll-off.

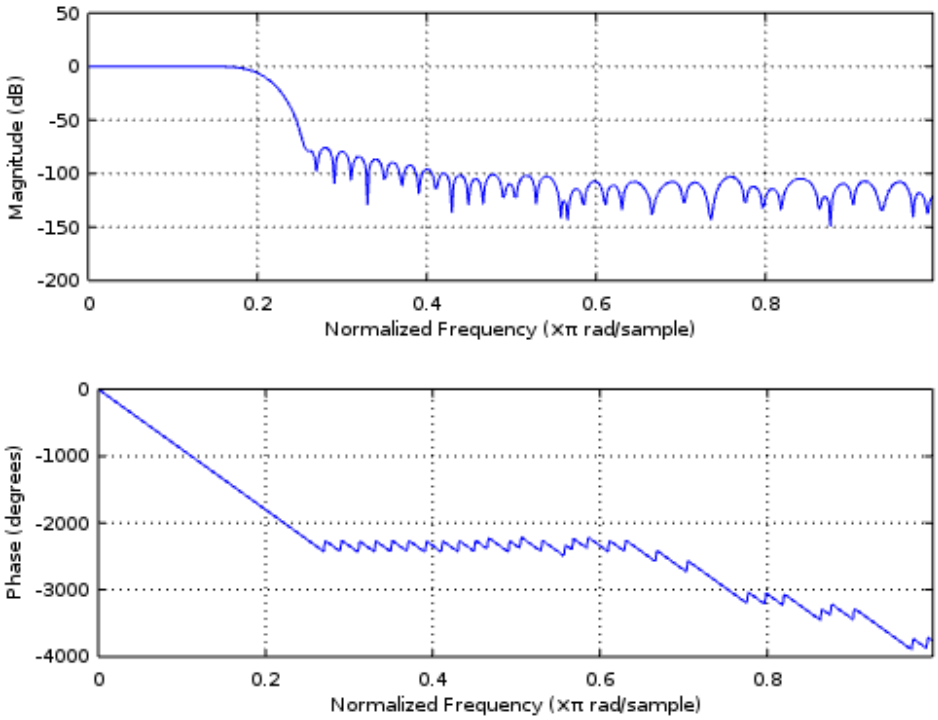

Figure 27: Sinc filter with Blackman window (generated with firdes_lowpass_f)

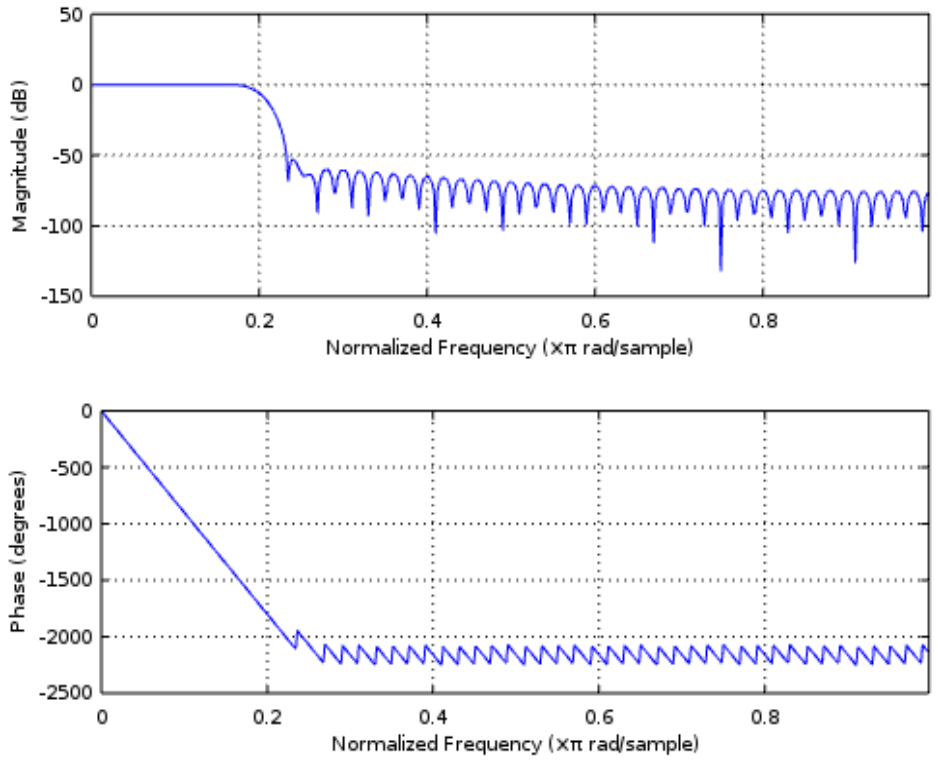

Figure 28: Sinc filter with Hamming window (generated with firdes_lowpass_f)

Although windowing helps to decrease effects of truncation, it should be noted that the longer the filter kernel is, the faster the roll-off is, and the transition bandwidth is also lower. The length of the filter for a given relative transition bandwidth (in proportion to the sampling rate) can be approximated as (11).

$$M \approx \frac{4}{B_{\text{transition}}} \tag{11}$$

The algorithm needed to design a bandpass FIR filter with complex coefficients can be easily derived from the low-pass filter design code. The filter we want to design is expected to have a lower cutoff frequency f_{lowcut} and an upper cutoff frequency f_{highcut}. Both can be positive or negative, but (12) should be satisfied.

$$-\frac{f_{\text{sampling}}}{2} < f_{\text{lowcut}} < f_{\text{highcut}} < \frac{f_{\text{sampling}}}{2} \tag{12}$$

First we design a low-pass filter with *real taps*, with a cutoff frequency of $f_{\text{cut}} = f_{\text{highcut}} - f_{\text{lowcut}}$, then we shift the passband by multiplying the taps with (13), where *i* is sample index.

$$e^{-j2\pi i \cdot (f_{\text{cut}}/2)} \tag{13}$$

If we apply a FIR filter with real taps on a complex signal, the passband is always mirrored to DC, but a filter with complex taps does not have this restriction on the passband. Thus only a filter with complex coefficients can be used for SSB demodulation of its own (to suppress negative frequency components, and pass everything above DC).

`void` **`firdes_lowpass_f`**
`(float *output,`
`int length,`
`float cutoff_rate,`
`window_t window)`
	Low-pass FIR filter design function for real signals, using the windowed FIR filter design algorithm, with a given *cutoff_rate* (in proportion to the sampling rate), filter *length*, and *window* function.

`void` **`firdes_bandpass_c`**
`(complexf *output,`
`int length,`
`float lowcut,`
`float highcut,`
`window_t window)`
	Band-pass FIR filter design function for complex signals, with a given *lowcut* and *highcut* ratio (in proportion to the sampling rate), filter *length*, and *window* function.

	The ratios defining the passband should be in the [-0.5, 0.5] interval.
`int` **`firdes_filter_len`** `(float transition_bw)`	Returns the required number of taps for a FIR filter to accomplish a given *transition_bw* transition bandwidth (in proportion to the sampling rate).

`window_t`
`firdes_get_window_from_string`
`(char* input)`
	Returns a window kernel identifier from a string (e.g. user input).

`char*`
`firdes_get_string_from_window`
`(window_t window)`
	Returns the name of the window kernel as string from a window kernel identifier.

`float`
`(*`**`firdes_get_window_kernel`**
`(window_t window))(float)`
	Returns the pointer to a window function from a window identifier.
	All window functions take only one parameter: *rate*, which should be in the interval [0, 1].

`float` **`firdes_wkernel_blackman`** `(float rate)`	Function to calculate Blackman window coefficients.
`float` **`firdes_wkernel_hamming`** `(float rate)`	Function to calculate Hamming window coefficients.
`float` **`firdes_wkernel_boxcar`** `(float rate)`	A dummy window function that always returns 1.0, so using it has no effect.

Table 7: Summary of filter design functions in libcsdr

9.3 Resampling

Changing the sample rate of the signal is required at multiple processing stages:

- during channelization,
- to match the sample rate of the demodulator sound card output sample rate.

After the frequency translation, we have our channel to select centered in the frequency domain. However, the signal still has much higher bandwidth than required, also containing other channels that we want to suppress.

We cannot just skip samples to decrease the sampling rate, as it would effect in unwanted aliasing, so we first have to apply a low-pass filter to the signal, to remove high frequency components that would overlap in the spectrum.

We can also make these two steps at once. If we simply calculate only one out of *M* output samples, where *M* is the *decimation factor,* we do less calculations, and have a filtered, decimated output signal.

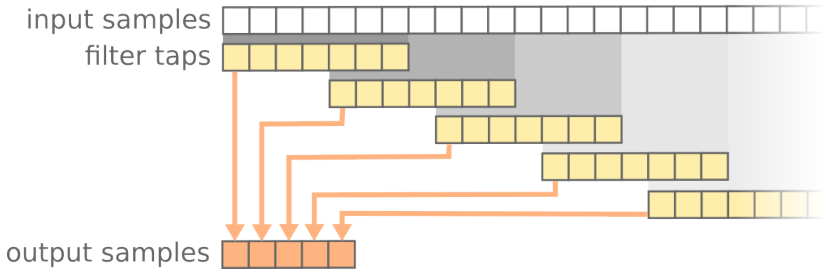

Figure 29: a simplified diagram to show how output samples are generated when using a FIR filter with 7 taps, and a decimation factor of 5.

The **decimating FIR filter** is implemented as the *fir_decimate_cc* function in the library. The cutoff frequency of the filter should be (14) to avoid aliasing effects.

$$f_{cutoff} = \frac{f_{sampling}}{2\,d} \tag{14}$$

f_{cutoff} is the cut-off frequency of the filter, $f_{sampling}$ is the sampling frequency of the input signal, and *d* is the decimation factor.

It is also important to note that here we use a FIR filter with real taps on a complex signal, by applying it to the vector made up of the real parts of the complex values, and also the vector of imaginary parts. The result is a complex signal with all frequencies suppressed except the $[-f_{cutoff}, f_{cutoff}]$ range, which means that the passband is centered around DC.

When some conditions are met, using the FFT and the overlap-add method for FIR filtering gives the same result faster than calculating the FIR filter formula directly. (I cover this method in section 9.4.) If we do not apply decimation, this method gets faster than the other around 10-64 filter taps [25] [26], depending on the exact implementation. The number of computational steps for the two methods can be approximated as on (15).

$$\begin{aligned} S_{FIR} &= N \cdot T \\ S_{FFT} &= (N+T) \cdot \left[2\log_2(N+T)+1\right] \end{aligned} \tag{15}$$

N is the number of output samples and T is the number of filter taps.

However, if M-ary decimation is taken into consideration, the FFT method takes the same amount of time to calculate, but the number of steps required for the FIR method decreases in proportion to the decimation factor (16).

$$S_{FIR} = \frac{N \cdot T}{M} \tag{16}$$

The real speed is also dependent on implementation and optimization, so the choice between the two algorithms should be based on real-world tests. In my application, I have chosen to implement the first decimation stage using direct FIR filtering, because it is likely to have computational advantage at higher decimation ratios. However, a test-based automatic choice may be added later to further improve performance.

There is another stage of decimation where we want to match the demodulated signal with the sample rate of the sound card (a fixed rate of 44100 Hz in this application). This requires a non-integer decimation rate, which can be achieved by linearly interpolating the right output samples. My library contains a **fractional decimator** that uses this principle for real signals. GNU Radio has the same functionality in the *fractional_resampler_ff* block, but it internally uses an 8-tap Minimum Mean Squared Error interpolator instead of linear interpolation, so it provides better performance in terms of spurious signals. Figure 30 shows a comparison between the spectrum of the resampled output of GNU Radio and csdr.

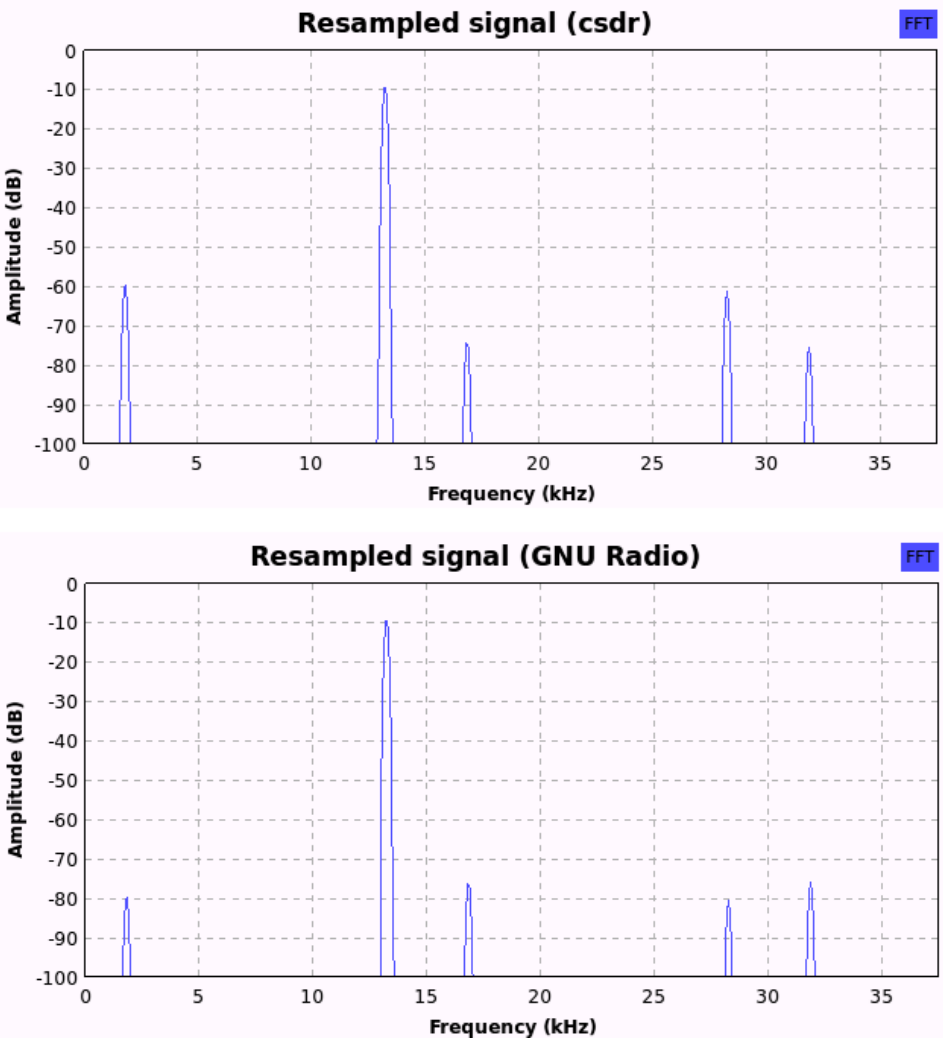

Figure 30: spectrum of 13 kHz sine wave resampled from 240 kHz to 75 kHz sample rate (with a decimation ratio of 3.2), using the Fractional Resampler block in GNU Radio and fractional_decimator_ff in libcsdr

I have also successfully implemented a resampler for **rational resampling** ratios. It first interpolates the signal by an integer factor, and then decimates it by another integer factor. As interpolation and decimation both require anti-aliasing filters, the one having the lower relative cutoff frequency is used. Interpolation alone would stuff the signal with zeros, so to save on computations, we do not calculate the multiplication result for the filter taps corresponding to zero taps. Decimation also saves us from calculating some output samples. Figure 31 shows a comparison between GNU Radio and csdr output for the rational resampler algorithm, which could also be used for resampling the audio signal to the rate of the sound card in OpenWebRX, but the default is the fractional decimator. Table 8 lists resampling functions in *libcsdr*.

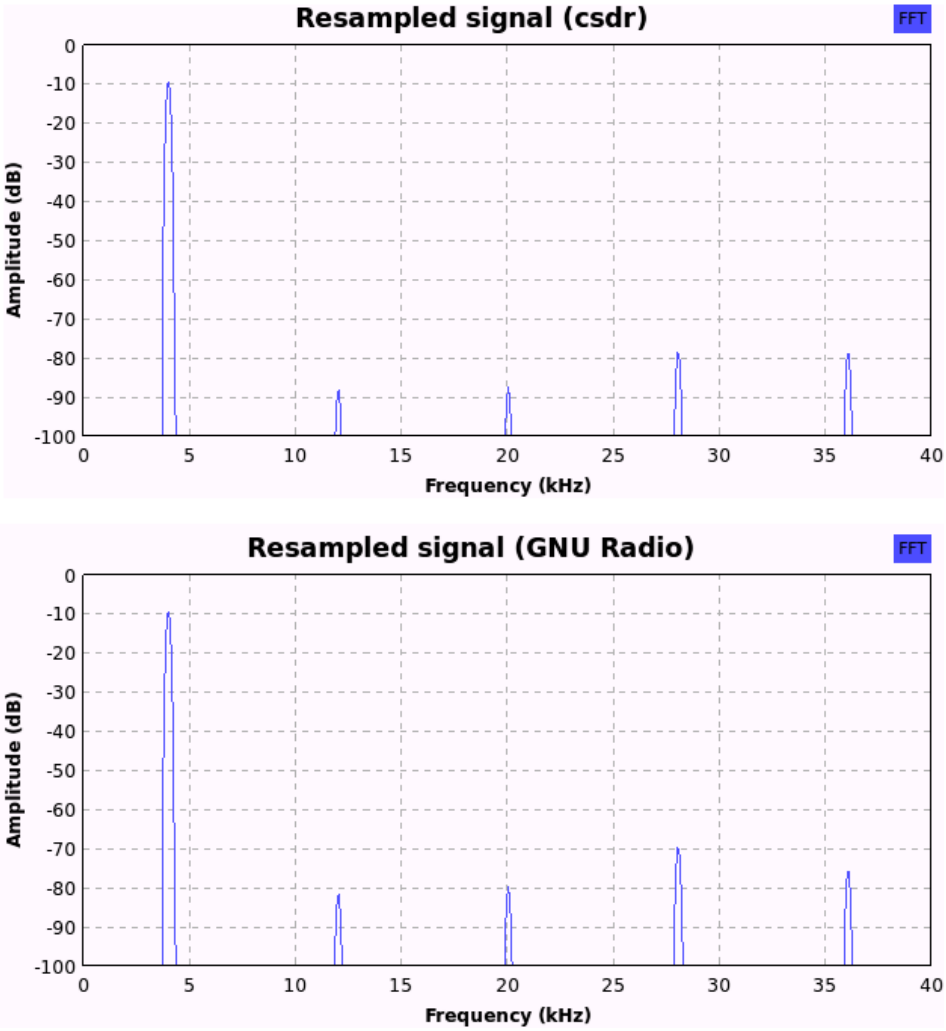

Figure 31: spectrum of 4 kHz sine wave resampled from 32 kHz to 80 kHz sample rate (with an interpolation factor of 5 and a decimation factor of 2), using the Rational Resampler block in GNU Radio and rational_resampler_ff in libcsdr

`int` **`fir_decimate_cc`**
`(complexf *input,`
`complexf *output,`
`int input_size,`
`int decimation,`
`float *taps,`
`int taps_length)`
	Decimates the input signal by an integer *decimation* factor, using an anti-aliasing FIR filter of *taps_length* number of filter *taps*.

`fractional_decimator_ff_t`
`fractional_decimator_ff`
`(float* input,`
`float* output,`
`int input_size,`
`float rate,`
	Decimates the input signal by the fractional decimation *rate*, using an anti-aliasing FIR filter of *taps_length* number of filter *taps*.
	The returned value has to be passed as

`float *taps,`
`int taps_length,`
`fractional_decimator_ff_t d)`
	parameter *d* the next time the function is called on the same input stream.

`rational_resampler_ff_t`
`rational_resampler_ff`
`(float *input,`
`float *output,`
`int input_size,`
`int interpolation,`
`int decimation,`
`float *taps,`
`int taps_length,`
`int last_taps_delay)`
	Resamples the input signal by integer interpolation and decimation factors.
	The returned value is a *struct* that contains the *output_size*, and the *last_taps_delay*, that should be passed as parameter the next time the function is called on the same input stream.

Table 8: Summary of resampling functions in libcsdr

9.4 Band-pass filter using FFT

After the first decimation stage, a band-pass filter is used. The passband of this filter can be set on the web GUI. This filter is also responsible for image rejection while performing SSB demodulation. Its transition bandwidth is low (300 Hz by default), but it also means that the filter kernel is long: it is also dependent on the input sample rate and the DDC decimation factor, but around 6-700 taps. As I have already noted, if no further decimation is taken into consideration, FIR filters above 64 taps are expected to be processed faster using FFT and the overlap-add method instead of calculating the FIR filter result directly.

This method is detailed in the literature [26], but a brief explanation is given below:

1. We calculate the frequency response of the filter.
    1.1. We choose a power of two for the array size, high enough to hold the filter kernel, and pad the kernel with zeros to fill the array.
    1.2. We apply FFT on the array and store the result.
2. We calculate the frequency response of an input buffer.
    2.1. The array size should be the same as the one that holds the frequency response of the kernel.
    2.2. We fill the following number of input samples into the array (and pad the remaining with zeros): FFT size − filter kernel size + 1

2.3. We calculate the frequency response of the input buffer.

3. We multiply the elements of the frequency response of the filter and the input buffer with each other one by one, and store the result.
4. We apply inverse FFT on the result.
5. We add the array of the overlapping samples (of $\text{filter kernel size} - 1$) that we stored last time, to the beginning of the result. This will be the output of the filter.
6. We store the last $(\text{filter kernel size} - 1)$ samples to the array of the overlapping samples.

Table 9 contains the function corresponding to this operation in *libcsdr*.

`void` **`apply_fir_fft_cc`**
`(FFT_PLAN_T* plan,`
`FFT_PLAN_T* plan_inverse,`
`complexf* taps_fft,`
`complexf* last_overlap,`
`int overlap_size)`
	Function that performs FIR filtering with FFT and the overlap-add method.

Table 9: apply_fir_fft_cc in libcsdr

10 Demodulation

With *libcsdr*, we can demodulate all the popular modulations used for transmitting voice on amateur radio bands, as well as continuous wave (CW) transmission that carry Morse code.

Modulation can be defined as a method required for all kinds of radio transmissions: by varying the parameters of a periodic signal called carrier wave, we convey the original signal into another one that can be physically transmitted [27].

We can classify modulations to amplitude and angle modulations, by the two parameters of the sine wave that can be changed in order to modulate a sinusoidal carrier signal.

$$s(t) = A(t)\cos\left[2\pi f_c + \varphi(t)\right] \tag{17}$$

$A(t)$ is the amplitude, $\varphi(t)$ is the phase, and f_c is the carrier frequency.

In the sections below, concepts of double-sideband amplitude modulation (AM-DSB), frequency modulation (FM, which is a sub-case of angle modulations), and single-

sideband suppressed carrier amplitude modulation (AM-SSB/SC) are presented, along with the explanation of the demodulation concepts and algorithms used in *libcsdr*.

In the equations below, $s(t)$ denotes the modulated signal, and $x_m(t)$ denotes the modulating signal (which is to be transmitted over the radio channel, to be reproduced at the receiver). In software implementation, we assume that $x_m(t)$ is within the range [-1; 1].

10.1 Amplitude modulated signals

Mathematical representation of a real-valued AM signal is shown on (18), with graphs on a sine-modulated signal in time-domain (Figure 32) and frequency domain (Figure 33).

$$s_{AM}(t) = A_c\left(\frac{1+k_d \cdot x_m(t)}{2}\right)\cos\left(2\pi f_c t\right) \tag{18}$$

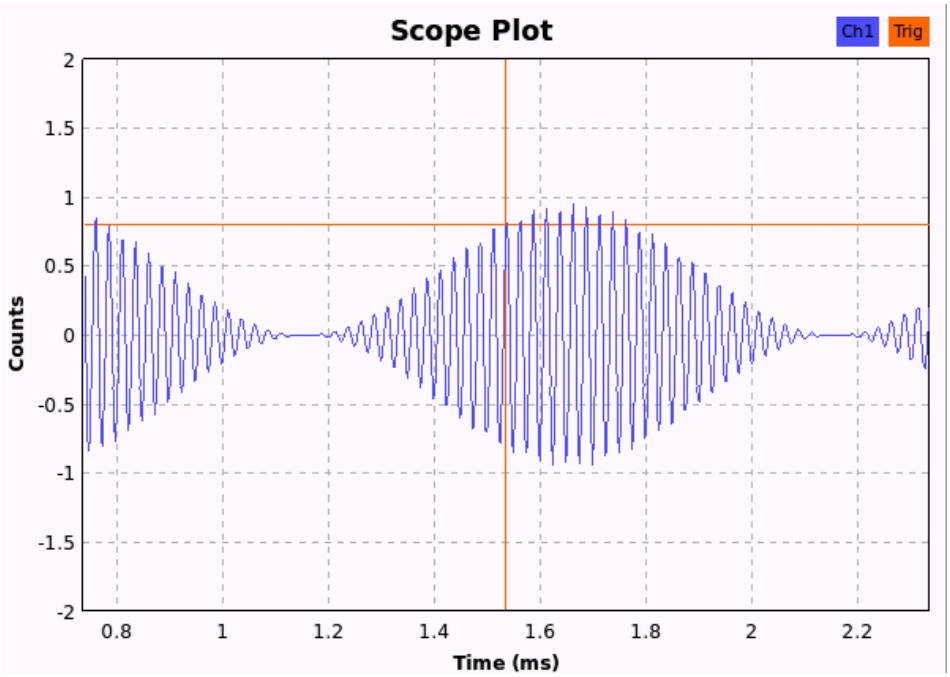

Figure 32: AM signal in time domain (modulated by a pure sine wave)

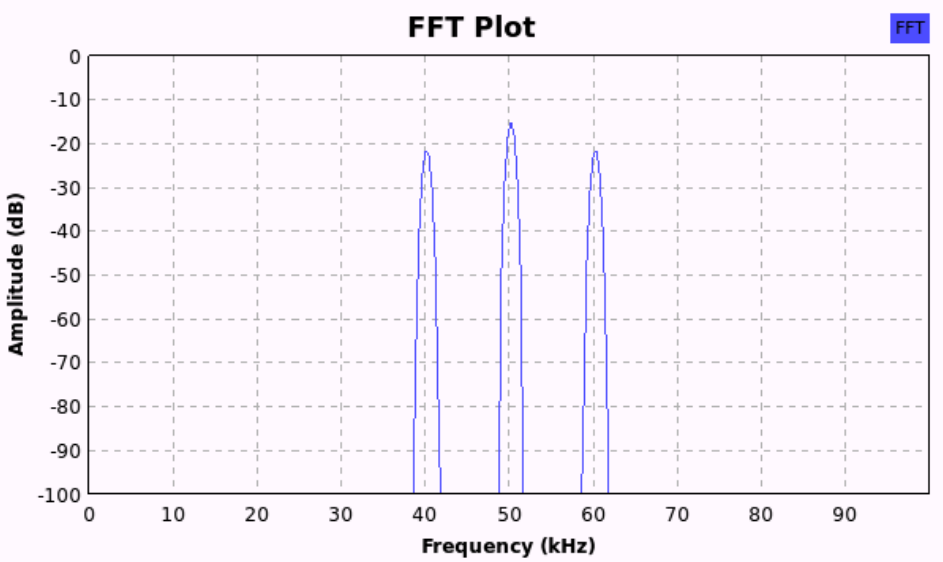

Figure 33: AM signal in frequency domain

We have to note that the modulating signal must be above zero, so we shift the signal level accordingly in the expression. A_c represents the amplitude of the carrier signal, which is also the maximum amplitude for the modulated signal. f_c is the carrier frequency. k_d is the modulation index: it should not exceed 100% because it would prevent the the demodulator from recovering the original amplitude envelope and result in a distorted signal on the receiver end.

The signal is spectrally centered at f_c, while the modulating signal appears just above and just below f_c (this can also be proven), these are called *sidebands* – hence a more accurate name for this type of modulation is double-sideband amplitude modulation (AM-DSB).

10.2 AM demodulation techniques

While performing AM demodulation, we want to recover the envelope of the amplitude of the modulated signal. In some traditional receivers, AM demodulation is done by rectifying the modulated signal (using semiconductors), and as it is a nonlinear operation that introduces harmonics, we low-pass filter the signal before feeding it to the speakers.

In the AM receiver block in our SDR receiver system, we have a complex AM signal with the carrier frequency f_c already centered at zero after downconversion (19).

$$\overline{s_{input}}(t) = A_c\left(\frac{1+k_d \cdot x_m(t)}{2}\right)\cos\left(2\pi f_c t\right)\cdot \underbrace{e^{-j2\pi f_c t}}_{\substack{\text{introduced by}\\ \text{downconverter}}} =$$

$$= A_c\left(\frac{1+k_d \cdot x_m(t)}{2}\right)\frac{e^{j\cdot 2\pi f_c t}+e^{-j\cdot 2\pi f_c t}}{2}\cdot e^{-j2\pi f_c t} = \tag{19}$$

$$= A_c\left(\frac{1+k_d \cdot x_m(t)}{2}\right)\frac{1+\overbrace{e^{-j\cdot 4\pi f_c t}}^{\substack{\text{removed}\\ \text{by LPF}}}}{2}$$

If we have the downconverted signal, to get the amplitude of every complex sample, we apply (20), in which $I(t)$ and $Q(t)$ are given by (21).

$$\frac{1+x_m(t)}{4} = Magnitude\left(\overline{s_{input}}(t)\right) = \sqrt{I(t)^2+Q(t)^2} \tag{20}$$

$$I(t) = \Re\left(\overline{s_{input}}(t)\right) \quad Q(t) = \Im\left(\overline{s_{input}}(t)\right) \tag{21}$$

While seeking for an efficient implementation by software, we can run into the problem that calculating the square-root on computers is a difficult and slow operation. Modern CPU-s with SSE instruction set do support the square root operation (e.g. via compiler optimizations or directly using intrinsics like *_mm_sqrt_ps*), but there is no support for it in ARM embedded systems with NEON. To overcome the problem, there is a magnitude estimator formula (22), and square-root is not required to calculate it.

$$Magnitude\left(\overline{s_{input}}(t)\right) \approx \alpha \cdot \max\left(|I(t)|,|Q(t)|\right) + \beta \cdot \min\left(|I(t)|,|Q(t)|\right) \tag{22}$$

This function has two parameters, α and β that can be optimized for the smallest error possible, however, tables of their typical optimal values already exist. [28]

It should be noted that the CORDIC algorithm can also be used for AM demodulation.

`void` **`amdemod_cf`**
`(complexf* input,`
`float *output,`
`int input_size)`
	AM demodulator using *sqrt* in *math.h*

`void` **`amdemod_estimator_cf`**
`complexf* input,`
`float *output,`
`int input_size,`
`float alpha,`
`float beta)`
	AM demodulator using the magnitude estimator formula, with a given *alpha* and *beta* parameter.

Table 10: Summary of AM demodulator functions in libcsdr

10.3 DC blocking filter

AM demodulation is not finished at finding the signal magnitude, we also have to remove the DC component from the resulting signal, even if the input signal level changes. There are multiple techniques for DC blocking in DSP, and I have implemented two of them.

A simple IIR filter can be created using a differentiator (having a zero at $z=1$ on the pole-zero plot) and a single-pole filter to compensate the drooping frequency response of the differentiator (by placing a pole near the zero of the differentiator, in the $0<z<1$ interval).

The filter with the equation (23) has a transfer function (24), which is also shown on Figure 34 and 35.

$$y[n]=p\cdot y[n-1]+x[n]-x[n-1] \tag{23}$$

$$H(z)=\frac{1-z^{-1}}{1-pz^{-1}} \tag{24}$$

The parameter *p* determines the fall-off of the transfer characteristics. The closer it is to 1, the sharper is the filter envelope.

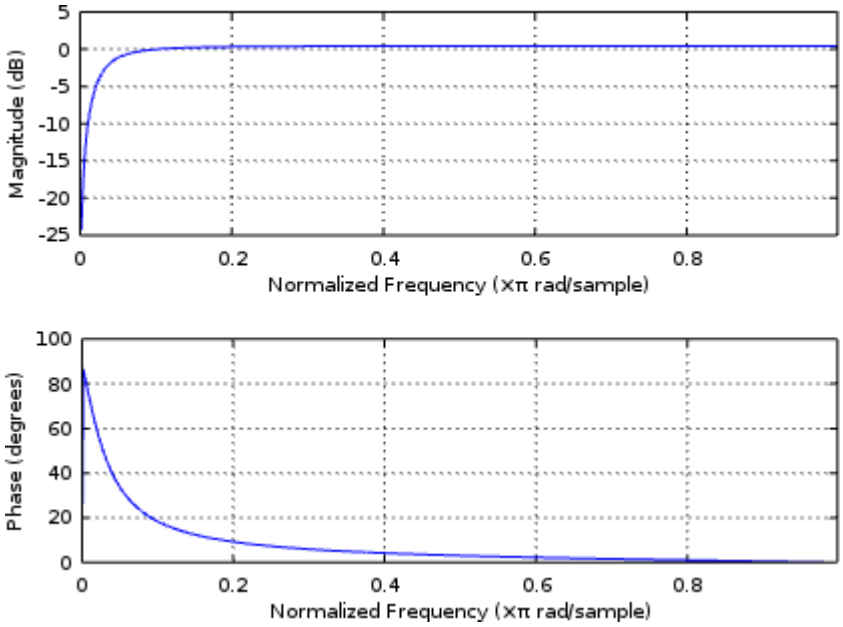

Figure 34: Transfer characteristic of DC blocking IIR filter with p=0.9

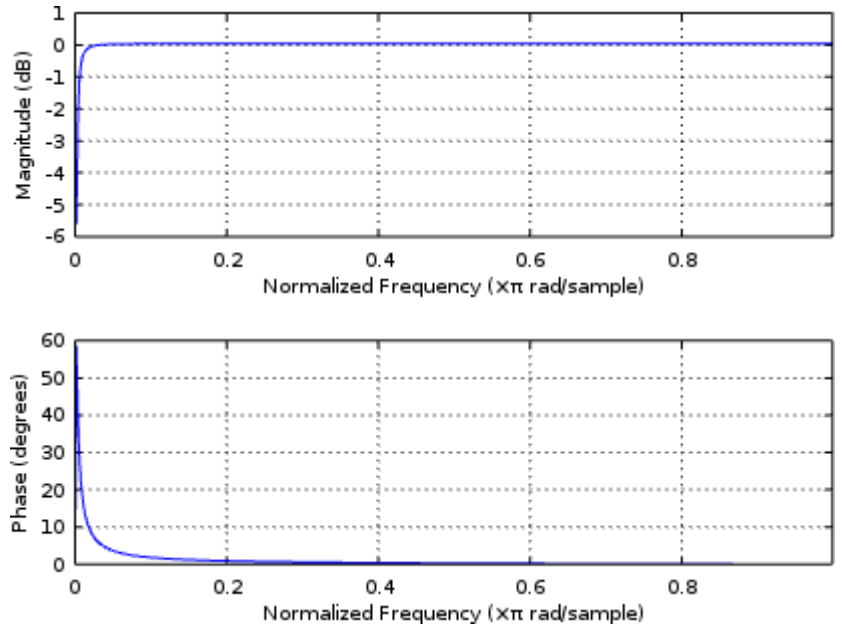

Figure 35: Transfer characteristic of DC blocking IIR filter with p=0.99

DC blocking can also be achieved by creating a FIR filter, or by applying moving average to the signal, and subtract the average from the samples. The latter is computationally efficient, can be paralleled, and is also widely used [29]. I have implemented this algorithm it with a modification: I calculate average of samples in blocks, and linearly change the subtracted value from one block to another. The larger blocks we have, the change in the this ratio will be smoother. This algorithm can be

easily vectorized by the compiler.

To test the DC blocking filter implementations, I made a GNU Radio flow graph that adds a DC component to white noise (as shown on Figure 36), and then removes it with my algorithms (as on Figure 37 and 38). Table 11 contains a list of DC blocking functions in libcsdr.

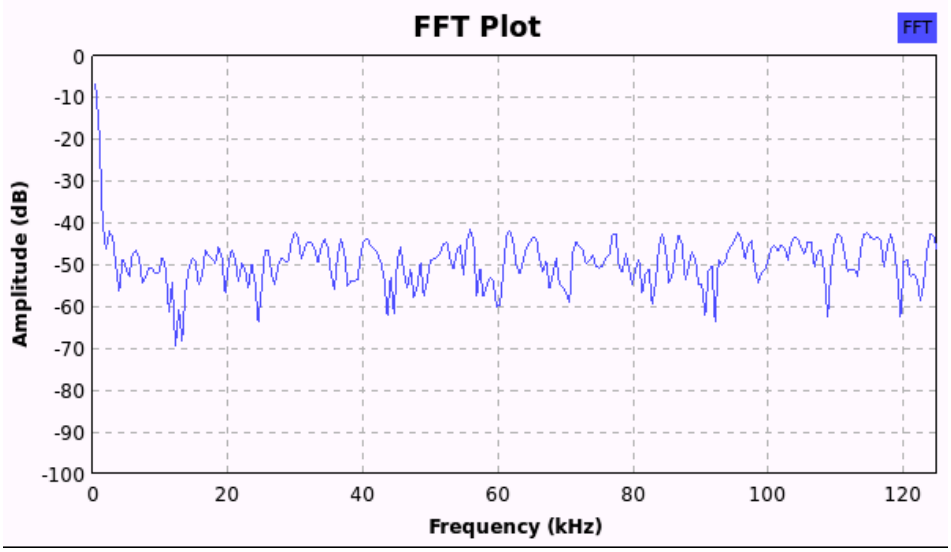

Figure 36: Original signal with DC component

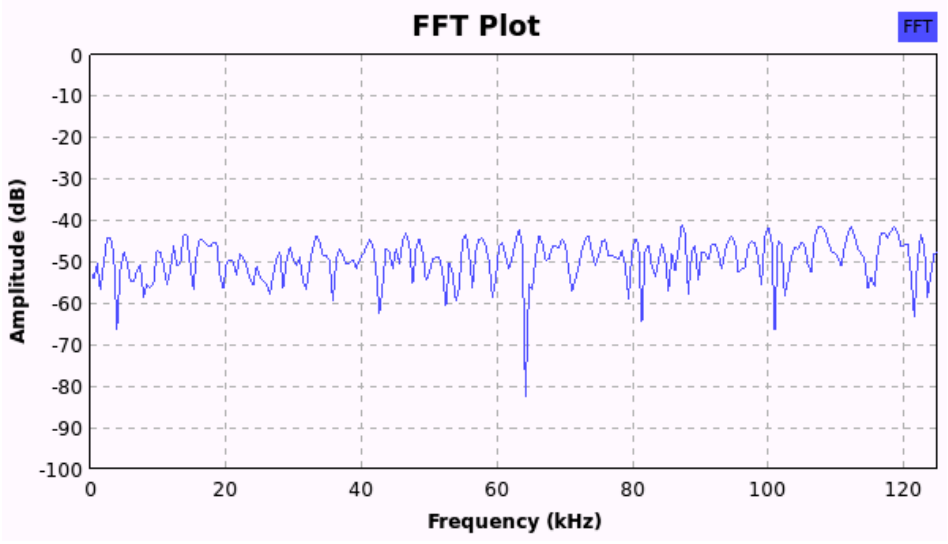

Figure 37: Signal with DC component removed with IIR filter (dcblock_ff)

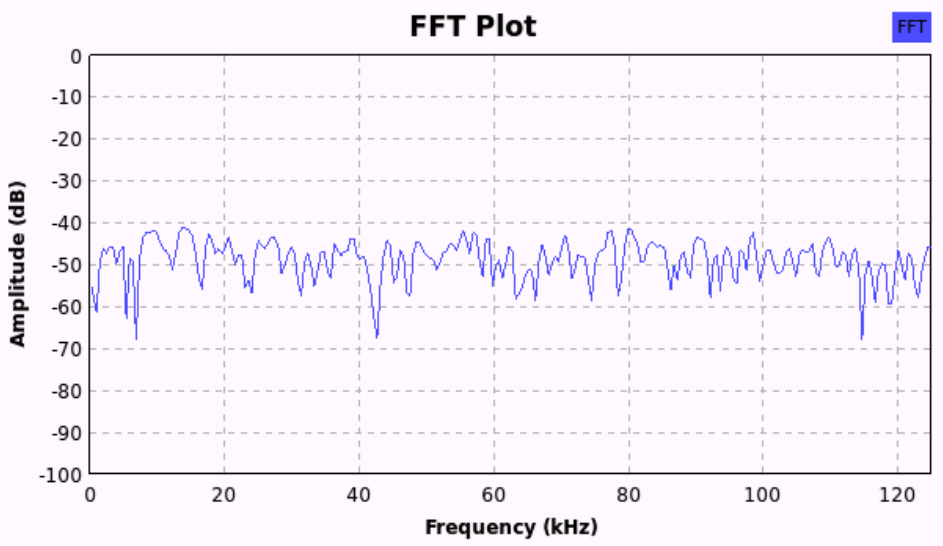

Figure 38: Signal with DC component removed with averaging (fastdcblock_ff)

`dcblock_preserve_t` **`dcblock_ff`**
`(float* input,`
`float* output,`
`int input_size,`
`float a,`
`dcblock_preserve_t preserved)`
	DC blocking filter using a single-pole IIR filter.
	The returned value should be passed as parameter *preserved* the next time the function is called on the same input stream.

`float` **`fastdcblock_ff`**
`(float* input,`
`float* output,`
`int input_size,`
`float last_dc_level)`
	DC blocking filter using averaging.
	The returned value should be passed as parameter *last_dc_level* the next time the function is called on the same input stream.

Table 11: Summary of DC blocking filter functions in libcsdr

10.4 Single-sideband signals (SSB)

Having a modulating signal with a bandwidth of *B*, the bandwidth of the AM signal is *2B*, which is inefficient use of both the available spectrum and the dynamic range of the RF power amplifier as of the relevant information is transmitted twice, and additionally, the carrier is transmitted. This problem gets solved by removing the carrier and one of the two sidebands at the transmitter side. The result is called single-sideband amplitude modulation with suppressed carrier (AM-SSB/SC), and based on the remaining sideband, can be upper sideband (USB) or lower sideband (LSB) transmission. Figure

39 shows how these modulations look like in the frequency spectrum.

Figure 39: different amplitude modulations

However, while SSB signals have a lot of desirable attributes, they are much harder to demodulate. In traditional receivers, SSB signals are demodulated with a product detector. Filtering the signal at the IF stages is even more important to provide sufficient image rejection.

Also a drawback of SSB that the receiver cannot reproduce the original carrier present at the transmitter (as it is removed before transmission), so it will always be slightly detuned, which results in a frequency shift compared to the original signal. While still eligible for speech, it is not suitable for transmitting music, and along with the advanced receiver structure it requires, this is why AM-DSB is still widely used for broadcasting on HF.

In SDR, multiple solutions are available for demodulating SSB. All work by removing all the negative or positive frequencies from the complex input signal, thus rejecting the other sideband.

The Hartley method for SSB demodulation (on Figure 40) uses a so-called Hilbert-transformer which shifts the phase on all frequencies by 90°.

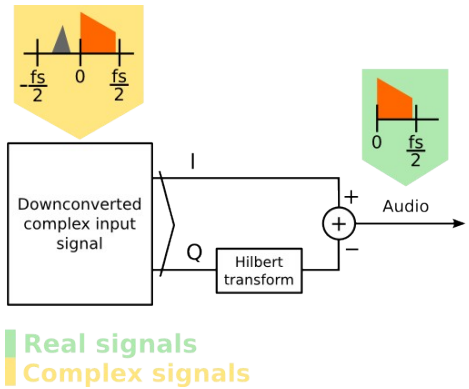

Figure 40: block diagram for the Hartley method for SSB demodulation

The Hilbert transformer is easy to implement for DSP as a FIR filter, but hard to realize by analog circuits.

The Weaver method (on Figure 41) can also be used for SSB demodulation. It is outlined in the figure below:

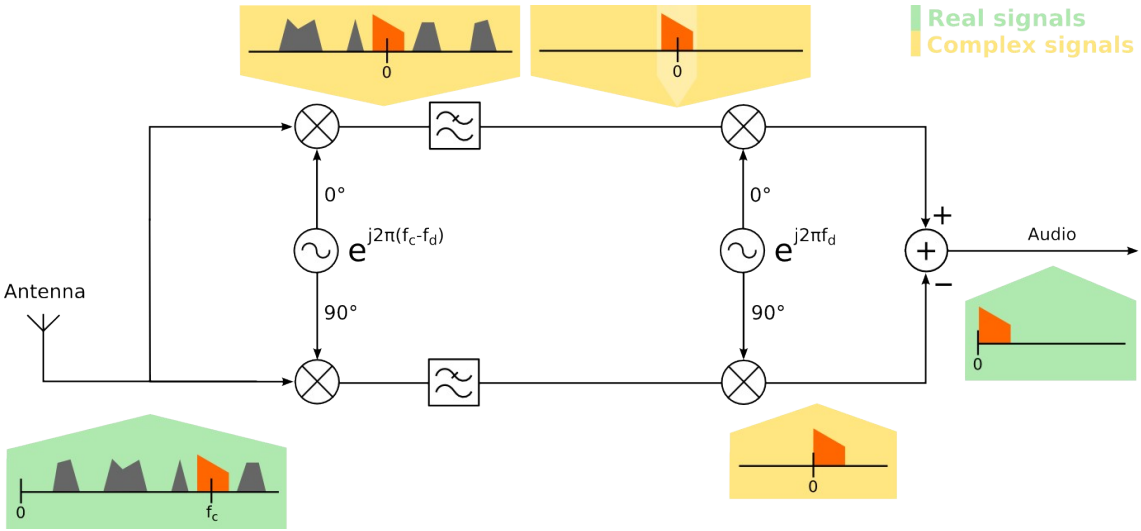

Figure 41: block diagram for the Weaver method for SSB demodulation

If we replace the separate low-pass filters and the second mixing stage by one single filter working on the complex signal, we can reuse the *firdes_bandpass_c* and the *apply_fir_fft_cc* routines already implemented in the library. As this way I did not have to write functions for designing a special FIR filter for Hilbert transform, I chose this method as the base of my implementation.

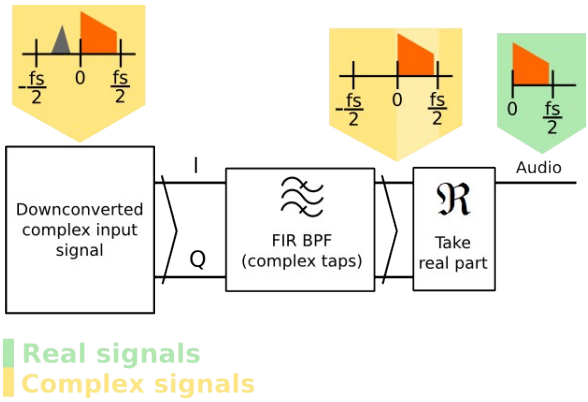

Figure 42: block diagram of SSB demodulation in OpenWebRX

As *apply_fir_fft_cc* uses a FIR filter with complex taps, we can use it to emphasize one sideband and suppress another. If our complex signal contains only positive frequencies (for USB) or only negative frequencies (for LSB), if we take the projection of the complex signal to any axis on the complex plane, we get a real signal that still contains

the same frequency components. This demodulation procedure is shown on Figure 42.

One of the most important parameters of an SSB receiver is the rejection of the other sideband. Figure 43 shows the frequency response of the default complex band-pass filter used for USB demodulation in OpenWebRX. (The sweep was generated automatically with a python script.) It is below -60 dB everywhere under DC.

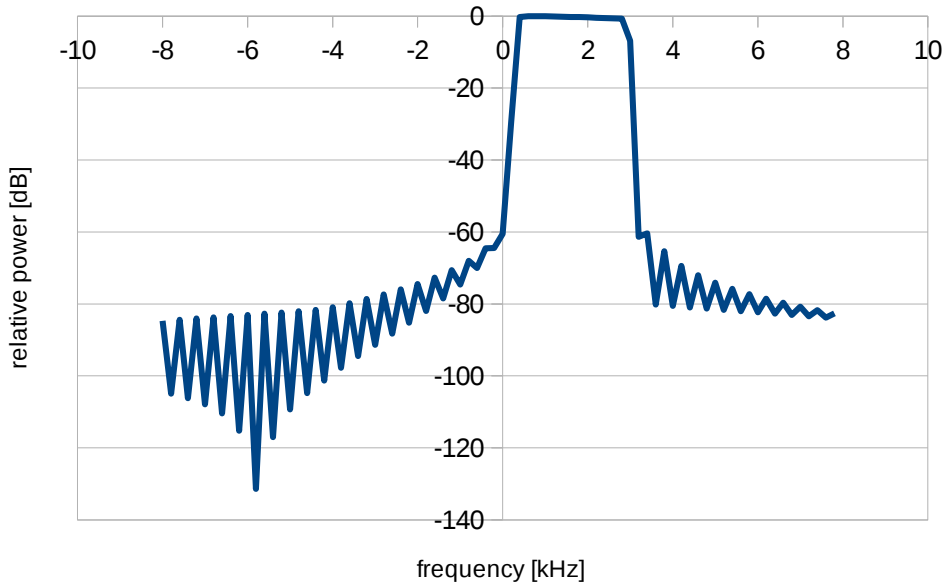

Figure 43: measured response of the band-pass filter for SSB demodulation in OpenWebRX

In most analog receivers, CW reception is just USB reception with a separate band-pass IF filter (that has much lower bandwidth). In OpenWebRX, we also use this approach for CW detection: we simply set different filter settings when the CW demodulator is selected. In this case, passband is less than 200 Hz.

10.5 Frequency modulated signals

Frequency is one of the parameters of a pure sine wave that can be changed in order to modulate it with a signal. An FM modulated real signal can be represented as (25), where (26) also applies.

$$s_{FM}(t) = A_c \cos\left(2\pi \int_0^t f(\tau)\, d\tau\right) \tag{25}$$

$$f(\tau)=f_c+f_\Delta x_m(\tau) \tag{26}$$

f_c is the carrier frequency, and f_Δ is the frequency deviation.

In our application we work with a complex signal having the FM signal centered at DC after downconversion. It means that we have (27) at the input of our FM demodulator block.

$$\overline{s_{input}}(t)=e^{\left(j\cdot 2\pi\int_0^t f(\tau)d\tau\right)} \tag{27}$$

We want to determine the derivative of the phase, (28). The simplest approach is to subtract the phases of the actual and the previous complex samples.

$$x_m(t)=\frac{d\arg\left(\overline{s_{\text{input}}}\right)}{\pi} \tag{28}$$

In the memory our complex samples are in rectangular notation, and to determine their phase, we have to use the arctangent function. To get the real phase between 0° and 360°, we have to correct the angle based on the signs of the real and imaginary parts of the input, but the built-in function *atan2* in *math.h* does this for us (29) (30).

$$\begin{aligned} I[i]&=\Re\left(\overline{s_{input}[i]}\right)\\ Q[i]&=\Im\left(\overline{s_{input}[i]}\right)\end{aligned} \tag{29}$$

$$x_m[i]=\frac{\text{atan2}(I[i],Q[i])-\text{atan2}(I[i-1],Q[i-1])}{\pi} \tag{30}$$

To respect the Nyquist criterion, the phase cannot change more than π from one complex sample to another, hence the division with π to limit the signal to the range [-1; 1].

The drawback of this method is the inability to be simply accelerated with SIMD as neither SSE, nor NEON does contain a dedicated instruction for *arctan*. To overcome this situation, there is another algorithm called *quadri-correlator* [30]. It is based on the identity (31).

$$\frac{d}{dt}\left[\arctan\left(\frac{Q(t)}{I(t)}\right)\right]=\frac{I(t)\cdot\frac{dQ(t)}{dt}-Q(t)\cdot\frac{dI(t)}{dt}}{I^2(t)+Q^2(t)} \tag{31}$$

By interpreting this formula in the digital domain, and also simplifying it, we get (32).

$$x_m[i] = \frac{d\varphi[i]}{dt} = \frac{Q[i]I[i-1] - I[i]Q[i-1]}{Q^2[i] + I^2[i]} \cdot \frac{1}{dt} \tag{32}$$

Unless we store the values for the arctangent in a lookup table in memory, the formula for the quadri-correlator is more computationally efficient to calculate than the previous method for FM demodulation, and is also more simple to implement. Moreover, the computation can be accelerated with SIMD instructions. Table 12 lists FM demodulation functions in libcsdr.

`float` **`fmdemod_atan_cf`**
`(complexf* input,`
`float *output,`
`int input_size,`
`float last_phase)`
	FM demodulator using the *atan2* function in *math.h* to calculate the phase of the complex samples.
	The returned value (the phase of the last sample) has to be passed as parameter *last_phase* the next time the function is called on the same input stream.

`complexf` **`fmdemod_quadri_cf`**
`(complexf* input,`
`float* output,`
`int input_size,`
`float *temp,`
`complexf last_sample)`
	FM demodulator using the quadri-correlator method.
	It requires an additional temporary buffer, *temp* (with a size of *input_size*) for its internal calculations.
	The returned value (the last sample) has to be passed as parameter *last_sample* the next time the function is called on the same input stream.

`complexf` **`fmdemod_quadri_novect_cf`**
`(complexf* input,`
`float* output,`
`int input_size,`
`complexf last_sample)`
	FM demodulator using the quadri-correlator method.
	It is less optimized but easier to understand.

Table 12: Summary of FM demodulation functions in libcsdr

10.6 De-emphasis

The noise power of the demodulated FM signal increases by the square of the frequency. [31] In practice, the high-frequency components of the modulating signal usually turn out to have lower amplitude, so the signal-to-noise ratio (SNR) would get

significantly lower for the higher frequencies. To compensate this effect, a so-called pre-emphasis filter is applied at the modulator input, which emphasizes higher frequencies, thus improving the SNR. It follows that we have to use a de-emphasis filter at the receiver to return the original signal.

Different parameters apply to different FM systems. FM broadcast receivers contain a single-pole de-emphasis filter, with a time constant of 50 μs in Europe and 75 μs in USA (and most countries use one of these two values). This behavior can be effectively modeled in software with a single-pole IIR filter, and although WFM (wideband FM) is almost never used on amateur radio bands, and a WFM demodulator is not included in the web interface as well, I have implemented this filter for the completeness of the DSP library. It is indeed used by the command-line tool *csdr-fm*, which is a testing tool for *libcsdr* for demodulating broadcast FM stations.

On the other side, the NFM (narrowband FM, with a current typical channel spacing of 12.5 kHz) is widely used in the amateur radio bands, but requires a different approach, as it is mainly used for transmitting speech. In the demodulated signal, only frequencies between 400 Hz and 4 kHz carry valuable information, but our filter characteristics should also be rolling off by 20 dB/decade in the passband to apply de-emphasis. In conclusion, we have to suppress frequencies below 400 Hz and above 4 kHz. Below 400 Hz, we optionally find the signal of the Continuous Tone-Coded Squelch System (CTCSS), which might even disturb the listener.

For this purpose, I have decided to use a filter bank that contains fixed arrays of FIR filter coefficients for specific sample rates. My C implementation in *predefined.h* contains three filters for sample rates of 48000 Hz, 44100 Hz and 8000 Hz, along with the code snippet to design new ones in *GNU octave* (Figure 44 shows the frequency response for one of them). My own filter design functions only support creating a subset of FIR filters (with low-pass and band-pass characteristics), but in this application a given custom amplitude envelope should be approximated. *GNU octave* has a built-in method *firls* for this purpose, however, the returned array should be normalized to have a gain of 0 dB around 400 Hz.

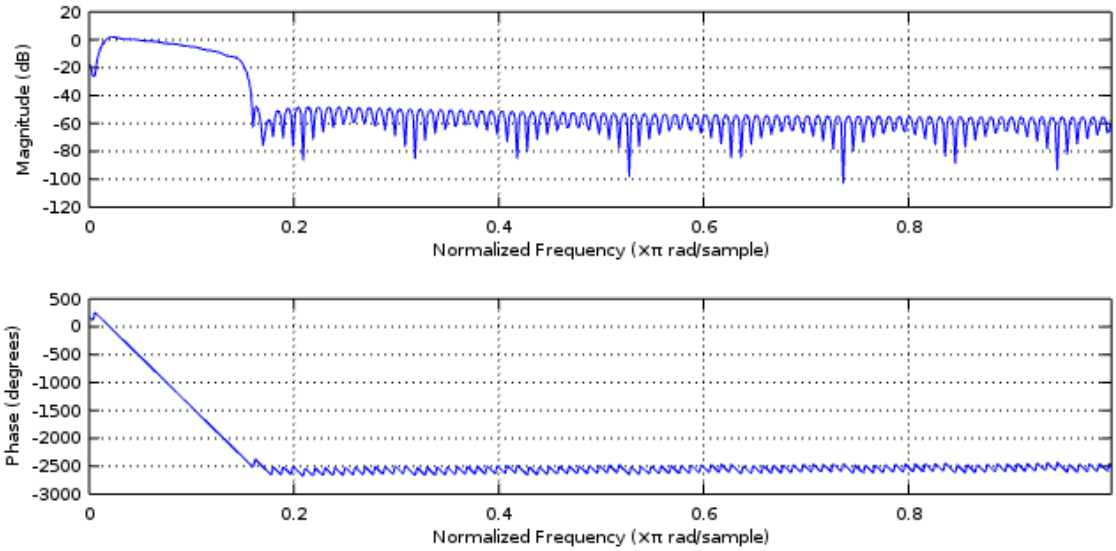

Figure 44: transmission of FIR filter modeling NFM communications equipment behavior including de-emphasis characteristics, running at a sample rate of 48000 sps

`float` **`deemphasis_wfm_ff`**
`(float* input,`
`float* output,`
`int input_size,`
`float tau,`
`int sample_rate,`
`float last_output)`
	De-emphasis filter to be applied after FM demodulation on WFM signals.
	It is realized using a single-pole IIR filter. The parameter *tau* is the time constant for the filter.
	The returned value has to be passed as the *last_output* parameter, the next time the function is called on the same input stream.

`int` **`deemphasis_nfm_ff`**
`(float* input,`
`float* output,`
`int input_size,`
`int sample_rate)`
	De-emphasis filter to be applied after FM demodulation on NFM signals.
	It uses a fixed FIR filter bank.
	Return value is the number of output samples, which also equals to the number of input samples processed. The remaining input samples should be inserted at the beginning of the next input.

Table 13: Summary of FM de-emphasis functions in libcsdr

11 Other DSP functions

In this part, some miscellaneous functions are explained, that do not fit in the previous sections.

11.1 Automatic gain control

In traditional receivers automatic gain control (AGC) is applied at an IF stage of the receiver, to keep the signal amplitude close to a given level before feeding to to the demodulator. In DSP we do not have non-linearities caused by analog components, but do have the effects of quantization, which are less problematic when dealing with floating point data. It means that we can put our AGC after the demodulator.

A digital AGC of course should model how an analog AGC works (Figure 45 illustrates this). In my implementation (*agc_ff* function in *libcsdr*, functions listed in Table 14) the AGC continuously calculates the gain that would be required to keep the signal at a given constant reference level. If the signal level increases or decreases, the AGC changes the gain, with an exponential transient. The *attack_rate* and the *decay_rate* are two different constants that control the AGC transient lengths in case of signal level increase or decrease, and *attack_rate* should be higher than *decay_rate* as if the signal level increases, we have to decrease the gain very fast to avoid clipping (which would occur at the conversion from floating point to integer before sending the output to the sound card).

We also want our AGC to keep the gain unchanged when the signal level gets radically changed only temporarily, for a short period. The purpose of *attack_wait_time* is to ignore sudden bursts of wideband noise appearing on the input, while the *hang_time* helps to ignore sudden, temporary dips in the signal level. [32]

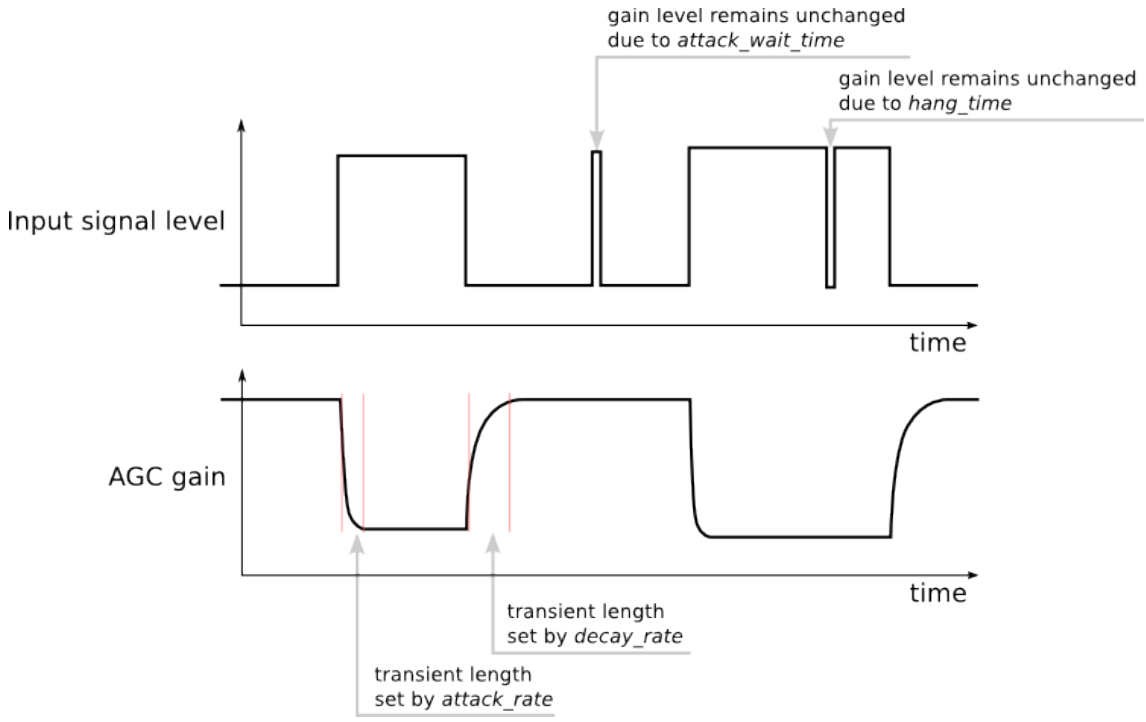

Figure 45: AGC operation

I have also implemented a simpler algorithm (*fastagc_ff* in *libcsdr*), which uses triple buffering and adjusts the gain linearly taking the highest amplitude peak (in all the three buffers) into consideration. Its undesirable behavior on short bursts make it insufficient to use it on SSB and AM, but it can be used for keeping the signal level constant after the quadri-correlator FM demodulator output.

`float` **`agc_ff`**
`(float* input,`
`float* output,`
`int input_size,`
`float reference,`
`float attack_rate,`
`float decay_rate,`
`float max_gain,`
`short hang_time,`
`short attack_wait_time,`
`float gain_filter_alpha,`
`float last_gain)`
	Automatic Gain Control function. It models analog AGC circuits.
	The parameters *attack_rate*, *decay_rate*, *attack_wait_time*, *hang_time* were already explained in this section.
	The reference level is given by the parameter *reference*. The gain will not be increased over *max_gain; gain_filter_alpha* is the parameter of the IIR loop filter for the AGC.
	The returned value has to be passed as the *last_gain* parameter, the next time the function is called on the same input stream.

`void` **`fastagc_ff`**
`(fastagc_ff_t* input,`
`float* output)`
	Automatic Gain Control function. It delays the output by two buffers and looks ahead when calculating the maximum amplitude peak. Its output will strictly be in the range

	[-1, 1]. The struct *input* contains the last three input buffers, the buffer size and the reference level. Calling the function swaps the buffers so that data can be filled into *input.buffer_input* the next time.

Table 14: Summary of AGC functions in libcsdr

11.2 Fast Fourier Transform

The Fast Fourier Transform has multiple uses in *libcsdr*:

- it makes spectrum display possible,
- it is used for processing band-pass FIR filters.

Indeed, there are highly optimized Fast Fourier Transform libraries available, so it is not worth implementing FFT manually. A short list of libraries that were considered to be used in the project:

- FFTW (supports SSE and NEON, available for free under GPL license),
- FFTS (supports SSE and NEON, free under BSD license, but does not build on x86 32-bit machines)
- cuFFT (official GPU-accelerated FFT implementation for nVIDIA CUDA),
- there is also a GPU-accelerated FFT implementation for the Broadcom SoC used in the Raspberry Pi single board computer.

In *libcsdr* I have implemented FFT using wrapper functions (listed in Table 15). Currently the only wrapper available is for FFTW, however, adding other FFT libraries should be easy by design. I have several reasons why I have chosen FFTW as the default, as it supports the widest range of hardware: it runs on older machines with x86 32-bit architecture, even if they do not have any SIMD capability in the CPU. However, its computational capability can be exceeded by the others if using a GPU for the task.

The command-line tool *csdr* has a feature to benchmark the FFT library *libcsdr* was linked against (as shown on Figure 46).

```
pcfl@ssd-mint ~ $
pcfl@ssd-mint ~ $ csdr fft_benchmark 1024 1000000
fft_benchmark: FFT library used: fftw3
fft_benchmark: initializing... done in 0.00111251 seconds.
fft_benchmark: 1000000 transforms of 1024 processed in 3.81763 seconds, 3.81763e-06 seconds each.
pcfl@ssd-mint ~ $
pcfl@ssd-mint ~ $ csdr fft_benchmark 1024 1000000 --benchmark
fft_benchmark: FFT library used: fftw3
fft_benchmark: initializing... done in 0.0566136 seconds.
fft_benchmark: 1000000 transforms of 1024 processed in 3.15937 seconds, 3.15937e-06 seconds each.
```

Figure 46: sample output of fft_benchmark with libcsdr on an Intel Core i7 M620 CPU clocked at 2.67 GHz

To calculate FFT, one first has to generate an FFT plan (which is internally specific to the given FFT library), and then call *fft_execute* on the plan.

`FFT_PLAN_T* make_fft_c2c (int size, complexf* input, complexf* output, int forward, int benchmark)`	It generates a plan for applying the Discrete Fourier Transformation (DFT) on an array of complex samples, for transforming the input array from time domain to the frequency domain. The *size* of the input and output array should be a power of two. If *forward* is false, the plan is created for an inverse transformation. If *benchmark* is true, the plan is optimized, but it takes more time to create.
`FFT_PLAN_T* make_fft_r2c (int size, float* input, complexf* output, int benchmark)`	It creates a plan for transforming an array of real samples in the time domain to frequency domain.
`FFT_PLAN_T* make_fft_c2r (int size, complexf* input, float* output, int benchmark)`	It creates a plan for inverse transforming an array of complex samples in the frequency domain to real samples in the time domain.
`void* fft_malloc (size_t size);`	Some FFT libraries require the input and output buffers to be allocated in a special way (SIMD operations usually need aligned memory access). This function calls the special memory allocation function in the specific FFT library.
`void fft_free(void* ptr);`	This function is for deallocating the memory allocated with *fft_malloc*.
`void fft_execute (FFT_PLAN_T* plan)`	It executes the given FFT plan. The calculation is performed and the output buffer (already given while creating the plan), is filled on calling this

	function.
`void` **`fft_destroy`** `(FFT_PLAN_T* plan)`	It frees the FFT plan, deallocating any resources.

`void` **`apply_window_c`**
`(complexf* input,`
`complexf* output,`
`int size,`
`window_t window)`
	This function is for applying a window function on the input signal before calculating its DFT (for spectrum display).

`void` **`logpower_cf`**
`(complexf* input,`
`float* output,`
`int size,`
`float add_db)`
	This function is to convert the complex output of the DFT to floating point power values on a logarithmic dB scale, for drawing a spectrum display.
	The parameter *add_db* is added to the output values, thus shifting the spectrum on the *y* axis.

Table 15: Summary of FFT functions in libcsdr

12 Conclusion and potential further improvements

The World Wide Web has been undergoing continuous revolution in the last years. Nowadays web technologies are even more mature, and the web has become a platform that can easily handle the complexity of a Software Defined Radio receiver GUI.

A multi-user SDR receiver is special from the aspect that it needs much more computing resources than if it was a single-user application. In some tests running the server on a machine equipped with an Intel Core i7 mobile CPU, OpenWebRX was able to serve at least 10 clients without problematic lags, processing the 1 Msps I/Q source for all of them separately (Figure 47 is a screenshot of the task manager and the output of the *top* command while OpenWebRX was under test).

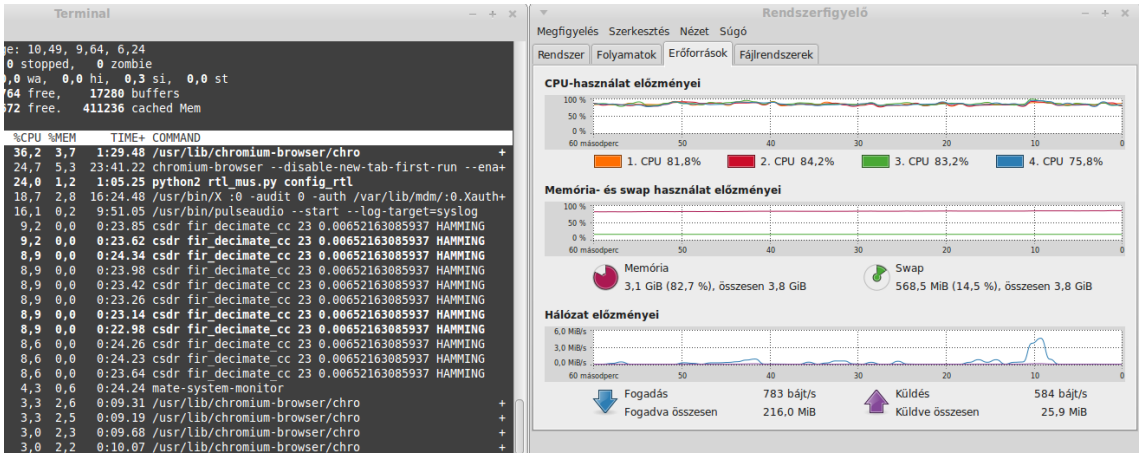

Figure 47: OpenWebRX server CPU usage with 10 clients

Using lower sampling rates the server is expected to handle even more clients. However, there is still a room for improvement regarding DSP speed. Highly efficient DSP was never an easy task to implement, but nowadays many technologies are available for parallel computing (GPGPU, FPGA) to facilitate this task. The easiest improvement could be made by porting the FFT wrappers to GPGPU-based FFT libraries, and also parallel execution of the FIR filter used in the DDC would boost speed.

Another planned improvement is related to the high bandwidth usage of individual clients: the bandwidth could be reduced by decreasing the sample rate of the audio sent over the network, and implementing an interpolator in JavaScript (as Web Audio API output works on a fixed sample rate).

13 Bibliography

[1] Wireless Innovation Forum, "What is Software Defined Radio?", retrieved on 2014. 12. 15. from: http://www.wirelessinnovation.org/Introduction_to_SDR

[2] Nutaq, "A short history of software-defined radio (SDR) technology", retrieved on 2014. 12. 12. from: http://nutaq.com/en/blog/short-history-software-defined-radio-sdr-technology

[3] Radixon Group, "Software Defined Radio", retrieved on 2014. 12. 13. from: http://www.winradio.com/home/facts.htm

[4] MacKenzie, A.B.; Reed, J.H.; Athanas, P.; Bostian, C.W.; Buehrer, R.Michael; DaSilva, L.A.; Ellingson, S.W.; Hou, Y.T.; Hsiao, M.; Jung-Min Park; Patterson, C.; Raman, S.; da Silva, C., "Cognitive Radio and Networking Research at Virginia Tech," *Proceedings of the IEEE* , vol. 97, no. 4, pp. 660-688, April 2009

[5] Pieter-Tjerk de Boer, "PA3FWM's software defined radio page", retrieved on 2014. 12. 14. from: http://wwwhome.cs.utwente.nl/~ptdeboer/ham/sdr/#nov2008

[6] Eged, B.; Babják, B., "Universal Software Defined Radio Development Platform", *Dynamic Communications Management*, Meeting Proceedings RTO-MP-IST-062, pp. 11-1 – 11-12.

[7] V. Iglesias; J. Grajal; O. Yeste-Ojeda, "Automatic Modulation Classifier for Military Applications", *19th European Signal Processing Conference*, pp. 1814-1818

[8] R. Prösch; A. Daskalaki-Prösch, *Technical Handbook for Radio Monitoring VHF/UHF Edition 2013*, Norderstedt: Books on Demand, 2013

[9] Linear Technology, "LTC2216/LTC2215 16-Bit, 80Msps/65Msps Low Noise ADC", LTC2215 datasheet. Retrieved on 2014. 12. 9. from: http://cds.linear.com/docs/en/datasheet/22165f.pdf

[10] Ettus Research, "Product Detail, USRP N210", retrieved on 2014. 12. 10. from: http://www.ettus.com/product/details/UN210-KIT

[11] Nuand, "bladeRF x40", retrieved on 2014. 12. 10. from: http://www.nuand.com/blog/product/bladerf-x40/

[12] Michael Ossmann, "HackRF One, an open-source SDR platform", retrieved on

2014. 12. 10. from: https://greatscottgadgets.com/hackrf/

[13] airspy.com, "A tiny and efficient software defined radio", retrieved on 2014. 12. 10. from: http://airspy.com/

[14] Hanlincrest Ltd., "FUNcube Dongle Pro+ User Manual (V4)", retrieved on 2014. 12. 10. from: http://www.funcubedongle.com/MyImages/FCD2ManualV4.pdf

[15] Osmocom, "rtl-sdr", retrieved on 2014. 12. 10. from: http://sdr.osmocom.org/trac/wiki/rtl-sdr

[16] Michael Karcher, "Re: How RTL-SDR samples signals", retrieved on 2014. 12. 15. from: http://permalink.gmance.org/gmance.comp.mobile.osmocom.sdr/264

[17] Pieter-Tjerk de Boer, "websdr.org", retrieved on 2014. 12. 10. from: http://websdr.org/

[18] K. Reid, "ShinySDR", retrieved on 2014. 12. 10. from: https://github.com/kpreid/shinysdr

[19] M. Stirling, "WebRadio", retrieved on 2014. 12. 10. from: http://www.mike-stirling.com/redmine/projects/webradio

[20] Internet Engineering Task Force (IETF), "The WebSocket Protocol", retrieved on 2014. 12. 10. from: http://tools.ietf.org/html/rfc6455

[21] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, pp. 67-85., retrieved on 2014. 12. 13. from: http://www.dspguide.com/CH4.PDF

[22] L. Pucker, "Channelization techniques for Software Defined Radio", *Proceedings of SDR Forum Conference*, 2003

[23] Warren Pratt, "shift.c", retrieved on 2014. 12. 14. from: http://svn.tapr.org/repos_sdr_hpsdr/trunk/W5WC/PowerSDR_HPSDR_mRX_PS/Source/wdsp/shift.c

[24] J. Blanchard, "A direct form discrete-time FIR filter of order N...", retrieved on 2014. 12. 10. from:

http://en.wikipedia.org/wiki/Finite_impulse_response#mediaviewer/File:FIR_Filter.svg

[25] T. Rondeau, "To Use or Not to Use FFT Filters", retrieved on 2014. 12. 10. from:

http://www.trondeau.com/blog/2014/2/27/to-use-or-not-to-use-fft-filters.html

[26] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, pp. 311-318., retrieved on 2014. 12. 13. from: http://www.dspguide.com/CH18.PDF

[27] Wikipedia, "Modulation", retrieved on 2014. 12. 13. from: http://en.wikipedia.org/wiki/Modulation

[28] Grant Griffin, "DSP Trick: Magnitude Estimator", retrieved on 2014. 12. 13. from: http://www.dspguru.com/dsp/tricks/magnitude-estimator

[29] R. Yates; R. Lyons, "DC Blocker Algorithms [DSP Tips & Tricks]," *Signal Processing Magazine, IEEE*, vol. 25, no. 2, pp. 132-134, March 2008

[30] N. Boutin; H. Kallel, "An arctangent type wideband PM/FM demodulator with improved performances," *Proceedings of the 33rd Midwest Symposium on Circuits and Systems*, 1990., pp. 460-463

[31] P. Ferenczy, *Hírközléselmélet*, Budapest: Tankönyvkiadó, 1972, p. 200.

[32] Midnight Design Solutions, LLC., "AGC ... Overview and Usage", retrieved on 2014. 12. 10. from: http://www.sdr-cube.com/AGC.html

14 Appendix

As it is planned to continue development of these projects, it is important to state exactly which version of the software the thesis refers to. Using the *git* revision control system (which is also behind GitHub) developers can commit changes to the code located at a remote repository, and every commit has an unique identifier. By the time of finishing this thesis, the last commit identifiers for the repositories were:

`openwebrx:` `978acf87092aa8bf27539fc8135d5d3978d6dda6`

`csdr:` `9e5ce0cc3b699b95736c56e53cea151c57e376c8`

`gr-ha5kfu:` `19f0fbb6670b458978f47f0430f6782ef5b194ac`

Testing the final software on the client side was done on Linux Mint with the following web browsers:

`Chromium version 39.0.2171.65`

`Mozilla Firefox 34.0`