

Realization of a very high bandwidth webSDR

Auteur : Carallo, Matteo

Promoteur(s) : Mathy, Laurent

Faculté : Faculté des Sciences appliquées

Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"

Année académique : 2023-2024

URI/URL : <https://gitlab.uliege.be/Matteo.Carallo/TFE>; https://youtu.be/fC0f_SNVJKw; <http://hdl.handle.net/2268>

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.



**University of Liège - School of Engineering and
Computer Science**

Realization of a very high bandwidth WebSDR

Master's thesis completed in order to obtain the degree of Master of
Science in Computer Science Engineering by Carallo Matteo

Supervisor: Pr. Mathy Laurent

Academic year 2023-2024

Acknowledgments

I would like to thank all the people who have helped me bring this thesis to fruition.

I give my thanks to Professor Laurent Mathy for supervising my thesis as well as Professor Jean-Michel Redouté for providing me with precious feedback and providing the subject of this thesis. I also thank Professor Marc Van Droogenbroeck for helping me understanding signal processing better.

I would also like to give my sincerest thanks to Morgan Diepart for giving me tremendous help all throughout the realization of this project especially by helping me understanding the fundamentals of signal processing and graciously accepting to reread my work.

I also thank Gauthier Gain which has kept an eye on the advancement of my work all throughout the year.

I also thank Louis Verhulst for proof reading my work.

Finally, I would like to thank all the professors I have had during the course of my studies without whom I would not have acquired the necessary skills to complete this thesis.

Contents

1	Introduction	4
1.1	Background	4
1.2	Subject of this thesis	5
2	Used technologies	6
2.1	Basic signal theory	7
2.2	Fast Fourier Transform (FFT)	8
2.3	Filtering	9
2.4	Software Defined Radio (SDR)	10
2.5	Digital Down Converter (DDC)	11
2.6	FM and AM modulation	12
2.7	Hardware used	13
3	Getting the XTRX to work	14
3.1	General problems and solutions	14
3.1.1	Missing dependencies during compilation	14
3.1.2	BAR 1 not able to be allocated	15
3.2	Operating system incompatibility	15
3.3	Using an older OS	16
3.4	Fixing the driver issues	16
3.5	Changing the firmware	18
3.6	Flashing process and end result	19
4	Overall architecture of the project	20
4.1	Requirements	22
4.2	Chosen languages	23
4.3	Development setup	23
5	Networking	24
5.1	General networking considerations	24
5.2	Theoretical data rates	29

6	Back-end server	30
6.1	Architecture	30
6.1.1	Circular buffers	32
6.1.2	Main	33
6.1.3	SDR interface	34
6.1.4	Digital down converter	35
6.1.5	Signal processor	37
6.1.6	Websocket server	40
6.1.7	Websession	41
6.1.8	TCP socket and TCP session	42
6.1.9	Object lifetime management	43
7	Website	44
7.1	Connection to the back-end server	45
7.2	Audio stream	45
7.3	Waterfall displays	46
7.4	User controls	47
7.5	Synchronization with the server	48
7.6	Tests	48
8	Back end unit tests	49
8.1	Circular buffer unit tests	50
8.2	SDR interface unit tests	50
8.3	DDC unit tests	51
8.4	Signal processor unit tests	52
8.5	Networking unit tests	53
8.6	Additional tests	53
9	Deployment and performance	54
9.1	Deployment	54
9.2	Code profiling	54
9.3	Performance in practice	57
9.4	Known issues	60
10	Conclusion and future work	61
10.1	Conclusion	61
10.2	Future work	62

Chapter 1

Introduction

1.1 Background

The history of software defined radios begins in the early 70's when researchers at the United States Department of Defense created the first receiver whose operation was defined in software. Contrary to what had been created up to that point in radio technology, it was able to be reprogrammed to change its operation instead of having to change physical components.

Fast forward half a century to today's world and anyone can get a software defined receiver for roughly 40€. This allows massive amounts of people to analyze the radio spectrum. However, while setting everything up to listen to a particular band is rather easy, not everyone is based in a good location nor has access to good antennas, which can be costly. In addition to this, while basic receivers can be found for cheap, they typically do not allow to analyze a wide bandwidth (they usually allow up to a few MHz of bandwidth). A solution to this is the concept of WebSDR. A WebSDR is an SDR that is connected to the internet which allows users from around the world to connect to it and use it just as if they had their own SDR.

1.2 Subject of this thesis

The main goal of this thesis is to develop, from scratch, a WebSDR. However, to set it apart from what is currently available, the goal will be to provide coverage for a very wide bandwidth (ideally close to *100 MHz*) as well as provide a way for users to receive the baseband signal directly instead of the demodulated audio.

In order to achieve this, multiple tasks that cover a wide range of software development will be undertaken. The first step of this thesis is taking a look at the drivers for the Software Defined Radio (SDR) that will be used (Fairwaves XTRX) and possibly patch them. Indeed, this SDR is connected to the server via pcie. This requires drivers to work, alas, the last version of the drivers that are available were made for Linux kernel version 4.19 which is relatively old. The next step of the thesis is the design and deployment of a back-end server that is going to handle the communication with the SDR card, handle the incoming requests from clients and process samples accordingly.

The final part of the thesis is the development of a front end for the users. This front end takes the form of a website which allows the user to see a waterfall graph of the radio band analyzed by the back end as well as pick a specific frequency of interest, a bandwidth and a demodulation which will be sent to him. An example of such a website can be found here¹.

All throughout the thesis, it will be important to keep in mind that future work will probably occur on it, so documentation and proper code commenting will be very important.

The Git repository is available here². A demonstration of the working system is available here³.

¹The WebSDR of the University of Twente

²Link : <https://gitlab.uliege.be/Matteo.Carallo/TFE>

³Link : https://youtu.be/fC0f_SNVJKw

Chapter 2

Used technologies

In order to bring this project to fruition, several concepts have been used. These concepts and technologies require a bit of background in order to understand how they work and use them correctly.

This project revolves around signal processing. According to the IEEE, signal processing is defined as "a branch of electrical engineering that models and analyzes data representations of physical events as well as data generated across multiple disciplines"¹. For example, an antenna may receive signals from several radio stations and signal processing will allow to isolate one particular radio station, demodulate it and play the audio.

In signal processing, there are a lot of basic blocks which apply well studied transformations to signals, these can be assembled together to produce the desired output. There are also a lot of key concepts that need to be understood in order to build a functional system.

In the case of this thesis, only digital signals will be relevant as the analog part of the signals will be handled by a the SDR driver itself. Each of the concept and technologies relevant to this project will be explained in the following sections.

¹IEEE Signal Processing Society, *What is signal processing?*, 2024, available at <https://signalprocessingsociety.org/our-story/signal-processing-101>

2.1 Basic signal theory

Only a handful of signal theory concepts are needed to understand this thesis, the following sections will not go over all the details of why specific requirements are needed but one essential requirement is the Nyquist sampling theorem which states that :

$$f_s \geq 2f_{\max} \quad (2.1)$$

Where f_s is the sampling frequency and f_{\max} is the highest frequency of the signal. Not respecting this condition may lead to aliasing. In simple terms, aliasing will make the captured signal appear at a different frequency, thus preventing proper analysis of said signal. Higher frequency parts of the signal captured will get folded on the lower frequencies, making it impossible to distinguish them.

But does that mean one needs a sample rate of 200 mega samples per second (*MSPS*) to analyze signals from 99 *MHz* to 100 *MHz*? Not really, what can be done is down-mixing the signal to bring it to a lower frequency. Indeed analyzing the spectrum between 99 *MHz* and 100 *MHz* is the same as shifting the signal 99 *MHz* to the left and analyzing the spectrum between 0 *MHz* and 1 *MHz*. Now the highest frequency is only 1 *MHz*, meaning that 2 *MSPS* are enough to capture the information in the signal. In general, the requirement is to have a sample rate which is twice the bandwidth that is to be captured.

As a side note, when working with IQ (complex) samples (like in this thesis), each sample effectively comprises of two measurements, meaning that the sample rate needed with IQ samples is the same as the bandwidth.

2.2 Fast Fourier Transform (FFT)

In order to understand the Fast Fourier Transform (FFT), there is a need to start with the basics : the Fourier Transform (FT). The Fourier transform is defined in the continuous time domain as follows :

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t} dt \quad (2.2)$$

The resulting function can be interpreted as the representation of the initial function with a different set of basis functions, namely, a basis set comprised of sinusoids. $F(\omega)$ is called the spectrum of the signal and contains the same information as $f(t)$. It makes the frequency content of the signal become apparent. In the case of periodic functions, the integral can be computed over one period.

In the case of discrete signal processing, the Fourier transform has an equivalent, the Discrete Fourier Transform (DFT). The DFT's coefficients can be obtained like so :

$$F[n] = \sum_{k=0}^{N-1} f[k]e^{-i\frac{2\pi}{N}nk} \quad (2.3)$$

Where $f[k]$ is a sequence of values sampled at particular moments from a continuous signal and N is the total number of samples.

This DFT is ultimately what is needed to transform the time domain signal sampled by an SDR to the frequency domain but computing it is expensive. Indeed, it requires $O(N^2)$ operations (on complex numbers) to compute the DFT where N is the total amount of samples. This becomes prohibitive for large N and a faster algorithm to compute it (or at least approximate it) is required.

The Fast Fourier Transform (FFT) is such an algorithm. It enables to compute the DFT while being $O(n\log(n))$. The library used in this project, *FFTW3*, uses the Cooley-Tukey algorithm to compute the FFT[7]. This algorithm is particularly efficient if N is taken as a power of 2.

2.3 Filtering

The previous section has introduced the FFT, this section will explain why it is necessary to use it in this project (and in applications of signal processing in general).

One action that is performed a lot in signal processing (and as a result in this project) is applying filters. Applying a filter with impulse response $h[n]$ to a discrete time signal $x[n]$ can be mathematically written as :

$$y[n] = (x * h)[n] = \sum_{m=0}^{N-1} x[m]h[n-m] \quad (2.4)$$

Where $(x * h)[n]$ is called the convolution of x and h . The computation of this convolution has complexity $O(N^2)$ with N the length of the sequence. This complexity is prohibitive for large N meaning that another way to compute the filtered signal is necessary.

When switching to the frequency domain, the filtered signal can be computed as follows :

$$Y[k] = X[k] \cdot H[k] \quad (2.5)$$

With $X[k]$ the DFT of $x[n]$ and $H[k]$ the frequency domain impulse response of the filter. In the frequency domain, the convolution is replaced by a product, which is much easier to compute.

This is where the FFT comes into play, as it provides a way to go from time domain to frequency domain in $O(N \log N)$. Performing the filtering in the frequency domain is $O(N)$ and going back into the time domain is $O(N \log N)$. In total, going in the frequency domain, performing the filtering and going back still stays $O(N \log N)$ compared to $O(N^2)$ when performing the filtering in the time domain. The FFT thus allows to filter efficiently. Of course, the constant in front of this theoretical complexity is higher than the one for the filtering in time domain but for large values of N (which is the case for this project), switching to the frequency domain is much faster.

2.4 Software Defined Radio (SDR)

The device that is central to this project is the Software Defined Radio (SDR). Such a device is defined as a "radio in which some or all physical layer functions are software controlled"². The physical layer is the layer in which signal processing occurs. Transitioning parts of the radio design to software has huge benefits. First, electronic designs take more time to create and are more expensive. Second, as many different SDR's use the same components, production costs can be lowered. In addition to that, new features can be added without the need to change physical components but by simply updating the software. It is also easier to handle very large bandwidths with SDR's than it is with "physical" radios.

For this project, there is no need to delve into a lot of details about SDR's as that would not be very useful, however, a few key elements of the SDR are going to be explained below just to give a bit of background on them.

The first element is the fact that they work in the digital domain. As their goal is to use general purpose programmable electronics, the analog signals coming from the antenna will need to be converted to digital signals. This is performed by an Analog to Digital Converter (ADC). This conversion is performed early in the SDR pipeline.

The second element directly ties into the first one. In order to perform the conversion, the analog signal needs to be prepared. This is done by applying various filters and amplifiers to the analog signal. However, as these operations are implemented at the hardware level, the number of available filters are limited. As such, some care needs to be taken as to not introduce unwanted behavior when configuring the SDR.

Figure 2.1 shows the high level idea of what an SDR looks like.

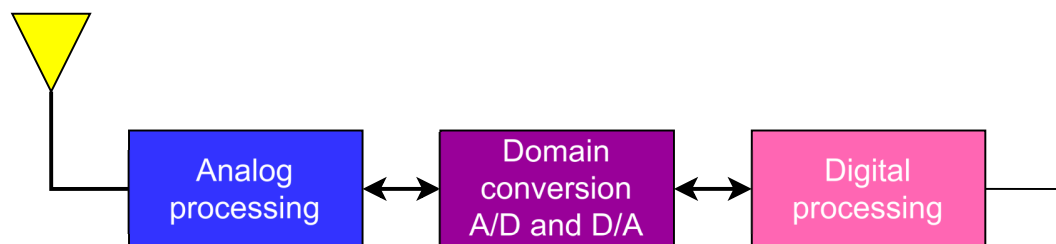


Figure 2.1: High level components of an SDR. Remade from : [5]

²Travis F. Collins, Robin Gets, Di Pu, and Alexander M. Wyglinski, *Software-Defined Radio for Engineers*, Artech House, 2018, page 3

2.5 Digital Down Converter (DDC)

A Digital Down Converter (DDC) is a signal processing block whose role is to convert a digital, band-limited signal to a lower frequency and lower sample rate. In more practical terms, the DDC will extract a single, narrower band signal from an initial, wider band, signal. This is accomplished by 3 main blocks :

1. A Numerically Controlled Oscillator (NCO) which will generate samples from a complex sinusoid at the frequency of the bandwidth of interest. When multiplied by the input signal, it creates a signal which is the same as the input signal but centered around the frequency of interest. In essence, the signal is shifted from one frequency to another.
2. A low pass filter which will be used to filter out everything not in the band of interest.
3. A downsampler which will reduce the sample rate of the signal. As the effective bandwidth of the signal has been reduced by low pass filtering, the sample rate can be lowered without any risks of artifacts appearing due to aliasing.

Once the signal is at a lower sample rate, it is easier to work with and subsequent steps will be less computationally intensive than working with the original sample rate as less samples have to be processed. Figure 2.2 shows an overview of a DDC implementation.

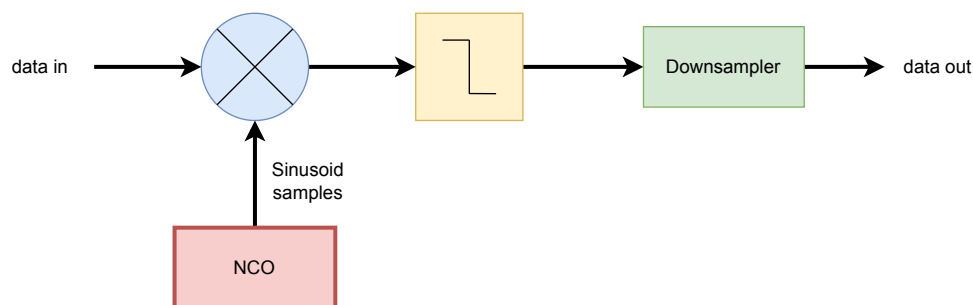


Figure 2.2: Overview of a DDC.

2.6 FM and AM modulation

This project allows the user to select 2 demodulations, FM demodulation and AM demodulation. Their modulations are both explained below. Both of them use a carrier wave to contain the modulated signal.

For Frequency Modulated signals (FM), the modulating signal is contained inside the carrier by varying the instantaneous frequency of the wave. The equation of the modulated signal is as follows :

$$s(t) = A_c \cos \left(2\pi f_c t + 2\pi f_\Delta \int_0^t m(\tau) d\tau \right) \quad (2.6)$$

where:

- $s(t)$ is the FM modulated signal.
- A_c is the amplitude of the carrier signal.
- f_c is the carrier frequency.
- $f_\Delta = K_f A_m$ where f_Δ is the frequency deviation, K_f is the sensitivity of the frequency modulator and A_m is the amplitude of the modulating signal.
- $m(t) \in [-1; 1]$ is the message signal.

For Amplitude Modulated signals (AM), the modulating signal is contained inside the carrier by varying the amplitude of the carrier proportionally to the amplitude of the modulating signal. The equation of the modulated signal is as follows :

$$s(t) = A_c (1 + k_a m(t)) \cos(2\pi f_c t) \quad (2.7)$$

where:

- $s(t)$ is the AM modulated signal.
- A_c is the carrier amplitude.
- f_c is the carrier frequency.
- k_a is the amplitude sensitivity or modulation index.
- $m(t) \in [-1; 1]$ is the message signal.

For demodulation, both of the pipelines used in this project are explained in section 6.1.5.

Figure 2.3 shows a comparison between the output signals obtained from FM and AM modulations.

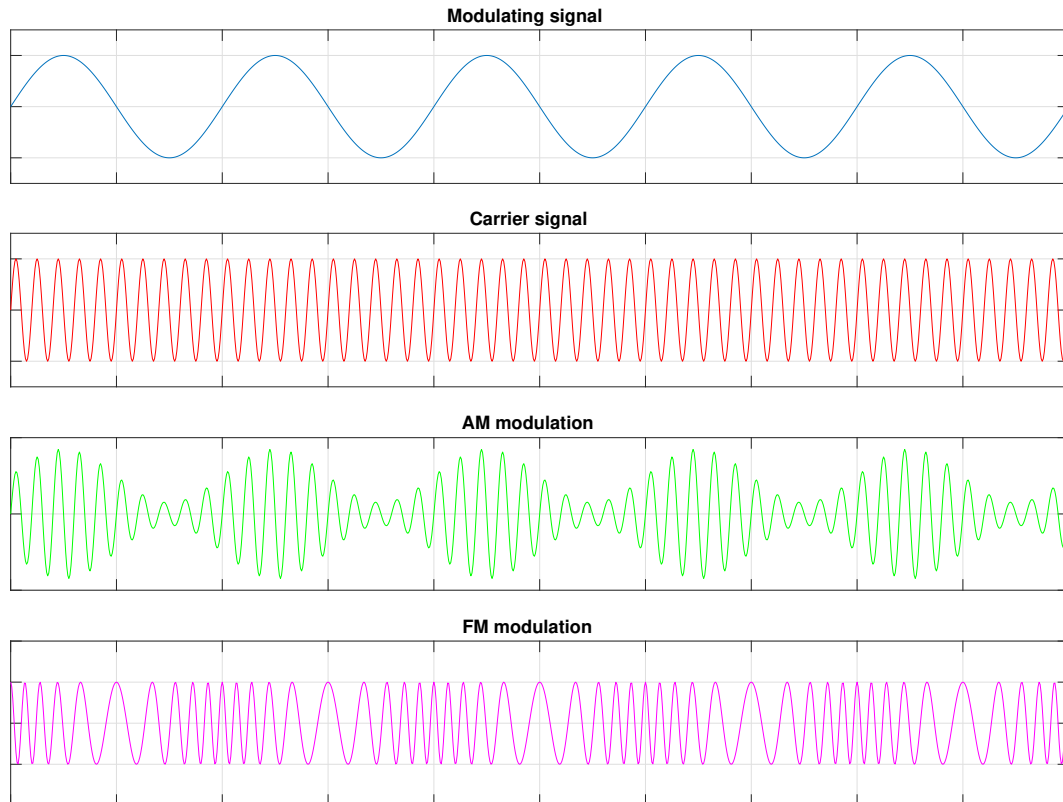


Figure 2.3: Result of FM and AM modulation

2.7 Hardware used

The SDR that is being used is an Fairwaves XTRX pro revision 5. It is capable of handling a maximum sampling rate of 120 MSPS with each sample having an I and Q value, each being 12 bits. The tuning range of the device goes from 30 MHz to 3.8 GHz .

The server that will be used is a Dell Poweredge T630 boasting two Intel Xeon E5-2630 (for a total of 16 physical cores), 64Gb (4 x 16Gb) of 2133 MHz ECC DDR4 memory and an RTX 2080Ti GPU.

Chapter 3

Getting the XTRX to work

The first part of this thesis was to make the library and driver for XTRX work with the server. There are a couple of errors when trying to get the XTRX library and driver to work on Linux kernel 5 and above. All of these issues are reported in the following section and, if a solution was found, the solution is also reported in the same section.

3.1 General problems and solutions

3.1.1 Missing dependencies during compilation

Following the readme provided with the repository leads to the compilation not succeeding. This is due to 2 issues :

1. libqcustomplot1.3 is not available anymore. This can be fixed by downloading it from the libq website¹ and placing the header file as well as the source file in the corresponding directory in the xtrx source files. The exact location is "source/libxtrx/examples/xtrx_fft". After that, simply modifying the cmake.txt in that same directory to include the newly added files is enough.

Alternatively, one could simply remove the examples and all references to them as they are not necessary for the driver to work.

2. The graphviz and swig packages are missing from the list of requirements, this can be fixed by simply installing the packages using a package manager.

¹<https://www.qcustomplot.com/index.php/download>

3.1.2 BAR 1 not able to be allocated

Taking a fork of the XTRX pcie driver² that is patched to compile and run on the latest Linux kernel also includes a fix for checking the successful allocation of BAR1. However, BAR1 is never allocated as it doesn't seem to be implemented at the hardware level. This leads the driver to abort its launch.

An easy fix for this is to simply not do anything when BAR1 is not allocated. This does not appear to hinder the working of the device.

3.2 Operating system incompatibility

The main issue when trying to make the XTRX device work with the server boils down to incompatibilities between the driver/hardware and the operating system that is being used.

Results of tests conducted on different operating systems can be seen in the table shown below :

Operating System	Kernel Version	glibc Version	Works?
Debian 10	4.19.0-25	2.28-10+deb10u2	Yes
Debian 10	5.10.0-0	2.28-10+deb10u2	No
Debian 11	5.10.0-26	2.31-13+deb11u7	No
Debian 12	6.1.0-13	2.36-9+deb12u3	No
Ubuntu 20.04	5.15.0-91	2.31-0ubuntu9.9	Yes
Ubuntu 22.04	6.2.0-37	2.35-0ubuntu3.5	No

Table 3.1: XTRX driver compatibility

On all of these configurations, the code compiles and the driver is loaded, however, the XTRX device does not manage to successfully conduct the tests provided by the driver library. More information about the exact nature of the issue is given in section 3.4.

3 ways of overcoming the compatibility issues have been devised and are as such :

1. Using an OS where the driver and library work out of the box.
2. Finding and fixing the issues the driver has and run an OS with a up to date Linux kernel.
3. Trying to flash the firmware of the XTRX with a more up to date LimeSDR XTRX firmware and using the limeSDR software suite on it.

These 3 potential solutions have all been explored to some extent and the findings have been reported in the following sections.

²https://github.com/myriadrf/xtrx_linux_pcie_drv

3.3 Using an older OS

This option is the easiest and requires the least amount of time to get the XTRX to work. Simply using Ubuntu 20.04 works and allows to directly begin working on the rest of the project. There will be no security updates beyond 2025 and it will not be possible to upgrade the kernel but that should not pose any issues for now. Additionally, given that the back end is being developed using the SoapySDR API to communicate with the XTRX device (see section 6.1.3), swapping from the XTRX to a more modern SDR is as easy as changing a line in the configuration file as long as the new SDR supports SoapySDR. This would then allow a more modern OS to be chosen.

3.4 Fixing the driver issues

Another solution that could be applied is to try to find the root cause of the issue that prevents the XTRX device from working on more recent operating systems. The issue itself manifests as the test program hanging and never finishing.

After analyzing the code associated to the testing program and going up the chain of functions called, the following piece of code was found to be the one responsible for the process hanging :

```

208 static int xtrxllpciev0_i2c_cmd(struct xtrxll_base_dev* bdev, uint32_t cmd,
    ↪ uint32_t *out)
209 {
210     struct xtrxll_pcie_dev* dev = (struct xtrxll_pcie_dev*)bdev;
211
212     internal_xtrxll_reg_out(dev, UL_GP_ADDR + GP_PORT_WR_TMP102, cmd);
213
214     if (out) {
215         ssize_t icnt;
216         do {
217             icnt = pread(dev->fd, NULL, 0, XTRX_KERN_MMAP_I2C_IRQS);
218             if (icnt < 0) {
219                 int err = errno;
220                 if (err != EAGAIN) {
221                     XTRXLLS_LOG("PCIE", XTRXLL_ERROR, "%s:_I2C_IRQ_error_%d_(%d)\n",
222                         dev->base.id, err, XTRX_KERN_MMAP_I2C_IRQS);
223                     return -err;
224                 }
225             }
226         } while (icnt != 1);
227
228         *out = internal_xtrxll_reg_in(dev, UL_GP_ADDR + GP_PORT_RD_TMP102);
229     }
230     return 0;
231 }

```

Listing 3.1:
sources/libxtrxll/modpcie/xtrxll_pcie_linux.c

This function tries to read a register but it keeps reporting as busy. This means that the program stays in an infinite loop. Looking at the driver code to see exactly what a "read" operation is supposed to do leads to the following code :

```

913 case XTRX_KERN_MMAP_I2C_IRQS:
914     i = wait_event_interruptible_timeout(
915         xtrxdev->queue_i2c,
916         atomic_read(((atomic_t*)xtrxdev->shared_mmap) + XTRX_KERN_MMAP_I2C_IRQS) !=
917             ↪ 0,
918         2 * HZ);
919     if (!i) {
920         return -EAGAIN;
921     }
922     return atomic_xchg(((atomic_t*)xtrxdev->shared_mmap) + XTRX_KERN_MMAP_I2C_IRQS
923         ↪ , 0);

```

Listing 3.2: sources/xtrx-linux-pcie-driv/xtrx.c

The driver code waits for an interrupt that never happens. This interrupt is supposed to arrive on the "queue_i2c" wait queue. The driver has the following code to handle waking up the queue :

```

720 if (imask & (1 << INT_I2C)) {
721     atomic_inc((atomic_t*)xtrxdev->shared_mmap + XTRX_KERN_MMAP_I2C_IRQS);
722     wake_up_interruptible(&xtrxdev->queue_i2c);
723 }

```

Listing 3.3: sources/xtrx-linux-pcie-driv/xtrx.c

This function gets called whenever the function "xtrx_msi_irq_other" is called and creates the appropriate mask. Finally, this last function is called by an interrupt request handler instantiated at the probing of the device when launching the driver.

On more recent versions of the Linux kernel, this step still works without any issues but interrupts are never generated by the handler. When looking at the kernel code responsible for the creation of the interrupt handler, no changes seem to have been made between working and non working kernel versions.

This could mean that the issue stems from a more complex interaction inside the kernel or even glibc, in which case a fix would be hard to find (except downgrading, but at that point it is simpler to use Ubuntu 20.04).

3.5 Changing the firmware

To change the firmware present on the XTRX device (i.e. : reprogram the FPGA), 3 requirements need to be fulfilled :

1. Having a new image to flash on the device which would allow more up to date drivers to be installed.
2. Having a backup of the image already present on the device in case something goes wrong in order to be able to re-flash it with the initial image.
3. Having a way to flash the image on the device.

For each of these requirements, a solution was found and is explained below.

The new image comes from a project called "LimeSDR-XTRX" by Lime Microsystems³. This project aims to take up abandoned development of the Fairwaves XTRX and improve it. After sending a email asking whether the LimeSDR-XTRX firmware could work with the Fairwaves XTRX, the company replied by stating that it should work but that they could offer no support. Once installed, the new firmware would allow the use of the LimeSDR software suite.

The backup of the image already present on the device can be obtained in two different ways. The first way is to use the utility program available in the XTRX library to read the content present on the FPGA. The second way is to use a Xilinx (the FPGA's manufacturer) probe that is made to read and write to the FPGA. The second solution has been chosen as, given that the XTRX library is quite bugged, there is no way to know if the read operation is actually working. If it isn't, the XTRX risks getting bricked without any way to restore it.

To flash the new image on the FPGA, the same tools as for reading the content of the FPGA can be used.

³Lime Microsystems website

3.6 Flashing process and end result

The first step is to install Vivado. This software suite, made by AMD, enables the usage of the Xilinx probe which can read/write to the FPGA. Once installed, the procedure to use it is described in the README file in the XTRX/FPGA directory.

The backup went fine but the produced bitstream could not be loaded back into the FPGA, this could indicate a faulty original firmware. When trying to install the firmware provided by LimeSDR, it seems to go well but when reading back from it, no changes seem to have been applied. This leads to the LimeSDR software suite not recognizing the XTRX.

As the risk of bricking the XTRX was too high given the impossibility of loading back the original firmware, the decision was made to stop trying to flash the FPGA and proceed with using Ubuntu 20.04.

After a bit more research, it appears that the issue may be linked to how Vivado was configured and was writing to the FPGA directly instead of the flash memory. Further attempts to reprogram the XTRX may lead to more success but the risk was deemed too high.

Chapter 4

Overall architecture of the project

The overall architecture of the project is represented on figure 4.1.

It is composed of 3 main parts :

1. The signal processing part, which is part of the server and handles everything from acquiring samples from the SDR to processing data for each individual client channel.
2. The client handling part, which is also part of the server and which is responsible for serving the website to clients as well as establishing websocket connections and TCP connections. It also handles client commands and sends the data streams to the client. The website serving is handled by a Node server while the rest is handled by the C++ program.
3. The website which is the front end presented to the client.

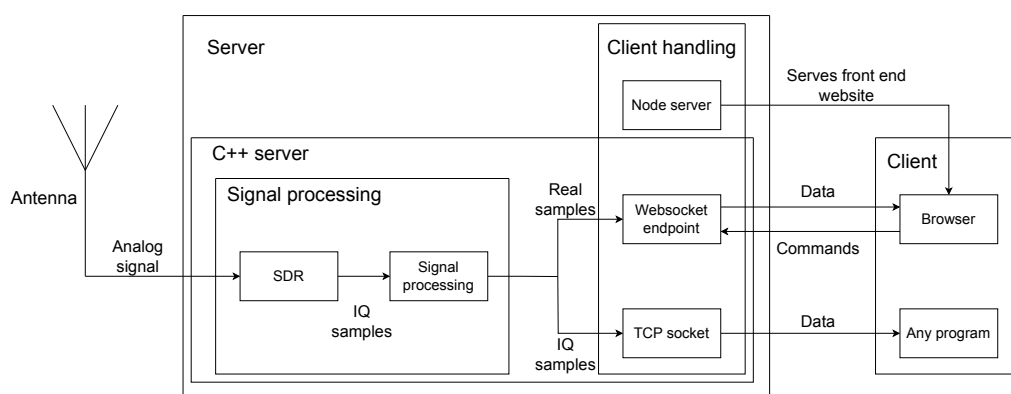


Figure 4.1: Overview of the project's architecture

When connecting to the server, the user will have the choice between two modes. The first one is one where he will receive the demodulated signal as an audio stream. The second one is where he will receive the modulated signal and handle the processing of samples on his own. If he chooses this second mode, a TCP socket will be made available for him to connect to and the samples will be sent through it.

The server is responsible for a lot of the work as sending the whole band captured by the SDR would not be possible due to it representing a lot of information. This means that at least the down conversion needs to happen on the server to drastically reduce the data rate that is required per client. To put these data rates into perspective, for a 50 MHz band, the data rate would be 381.47 MBytes/sec per client if all clients received the whole band, this is obviously not feasible in practice, justifying the processing work done on the server. More information regarding the networking can be found in chapter 5. More information concerning the back-end server is found in chapter 6, while information about the front end can be found in chapter 7.

4.1 Requirements

This project has a few requirements to work properly. Table 4.1 contains these requirements as well as a brief explanation of what these requirements are used for as well as the reason they were chosen.

Dependency	Usage	Reason
Boost/beast	Used for everything related to networking in the C++ server.	It is a mature and proven library with a lot of documentation and is rather easy to use.
Cuda	Used for performing computations on the GPU	The library used for the DDC (Libgkr4gpu) requires the use of Cuda.
FFTW3	Used to perform FFT's for the overview.	It is a very widely used library, very efficient and very robust.
GNURadio	Used to perform demodulations in the signal processing part.	It is the most mature and stable signal processing library available for C++. It also leverages vectorized instructions making it efficient. The documentation is also very good and the community is active.
SoapySDR	Used to interface with the SDR card.	Even though performance could be better by using the SDR libraries directly, SoapySDR allows to swap SDR's in and out without any changes to the code. A lot of SDRs support Soapy.
Libgkr4GPU	Acts as the DDC of this project.	One of the only libraries that are openly available and take advantage of GPUs.
Node.js	Used to host the website.	As the requirements for the website are not very restrictive, anything could have worked. Node.js was chosen because of previous experience with it as well as being rather easy to use.

Table 4.1: Dependency table

Of course, each of these libraries has dependencies themselves but going over them would not be of any use here. The readme files contain the steps to install everything that is needed.

4.2 Chosen languages

This project being divided into two major parts, a front end and a back end, two languages have been chosen, one for each. The back end has been coded in C++ as performance was of utmost importance. Additionally, Libgkr4GPU, a very powerful DDC library, is only available in C++. All other requirements are also met with popular and efficient C++ libraries. For the front end, Javascript was used for all scripting and HTML along with CSS for webpage formatting. Javascript was chosen mostly because of prior experience with it. Furthermore, since the web page was not very complex, it is perfectly suited for a small Javascript project.

4.3 Development setup

As the machine running the server and housing the XTRX device is located at Montefiore, having access to a similar setup locally is a necessity. To achieve this, the same OS has been installed locally and all the same libraries were installed too (except the XTRX library of course).

The RTLSDR¹ has been used to have a cheap SDR to develop locally. A very interesting functionality of this RTLSDR is that it also has SoapySDR compatibility, just like the XTRX. This means that the program can be developed and tested locally using the RTLSDR and then deployed on the production machine with no changes. This makes development cycles much faster.

¹RTL-SDR website

Chapter 5

Networking

5.1 General networking considerations

The most important part in searching for the appropriate networking technologies to use in this project is to properly state the requirements of the system. Said requirements are as follows :

- The client needs to be able to send commands to the server at any time during the connection. These commands can be relatively frequent (multiple commands per minute).
- The server needs to be able to send data to the client's web browser in the form of multiple data streams. These streams include the results of the FFT of the whole spectrum and the selected box¹ (to display it to the user) as well as the audio samples² if FM or AM demodulation is selected.
- The server needs to be able to send replies to specific queries made by the client, for example it should be able to reply with server information if the client asks for it.
- The server needs to be able to send data in the form of a continuous stream of raw samples³ when the client asks for raw, non-demodulated, samples. These raw samples should not be sent to the web browser of the client as they would be of no use there. The client should be able to receive them anywhere.
- As the data rates are already quite high due to the PCM encoded audio and the overview FFT data, additional data overhead should be kept to a minimum.
- Processing times should also be kept low as the application needs to stay responsive.

¹Represented by a byte for each sample

²Represented by real float values.

³Represented by complex float values.

The requirements for the connection between the client's web browser and the server make websockets perfect for this use case. Once the connection is established, raw binary data can be sent in it and interpreted accordingly on the other side. There will be need for some bytes of overhead with each messages as multiple streams will have to transit trough the same websocket connection but the amount of overhead data will be kept to a minimum. Figure 5.1 shows a visual representation of the message structure of messages sent out by the server.

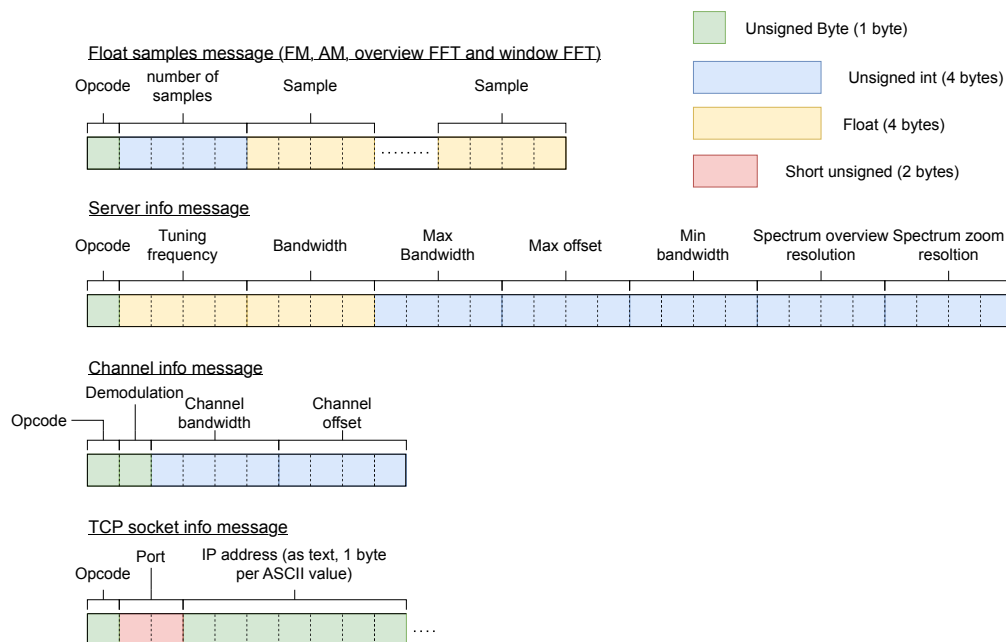


Figure 5.1: Server message structure

As the raw samples values are of little use in a web browser, these should be captured by another application by the client. The solution chosen for this problem is to open a TCP socket with a unique port to which the client should connect and which will send the raw data samples. This allows the client to have access to the samples anywhere. The message structure of messages sent trough the TCP socket are given on figure 5.2. Even though an opcode is not needed in this case as nothing else will transit trough the socket, it allows for future additions without any changes to the protocol.

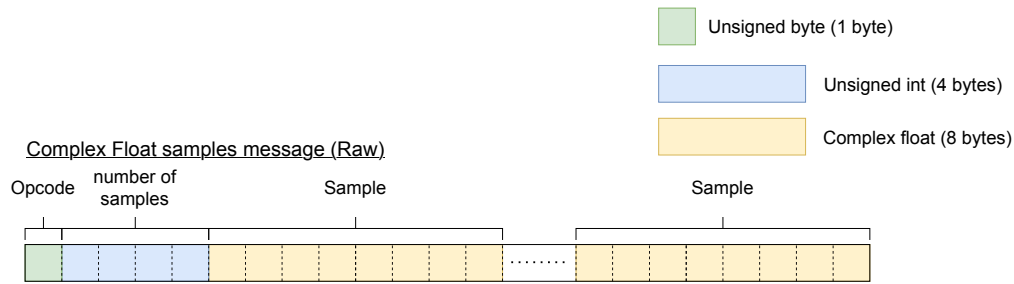


Figure 5.2: TCP socket message structure

The message structure for messages sent by the server through the websocket depends on the type of data that needs to be sent. Table 5.1 contains the identifier values and their meaning alongside the next bytes that will be in the message. The same has been done for the message type that is sent through the TCP socket in table 5.2.

The message structure of messages sent by the client are simple. The first byte indicates which operation is to be executed and the next bytes are the parameters of said operations. The list of commands and their explanation are available in table 5.3.

One last point of discussion is the security aspect of the networking protocol. The choice has been made to not secure the connection to simplify implementation. This choice is motivated by the fact that none of the information exchanged between the server and the client contain any form of personal or sensitive information.

First byte	Usage	Next bytes
0	Used to signal that the message contains float values representing the output of the demodulation.	4 bytes for the number of samples followed by 4 bytes per sample (float format).
2	Used to signal that the message contains byte values for the overview FFT and the values for the selection box FFT.	4 bytes for the number of samples followed by 1 byte per sample. First all the samples for the overview, then all the samples for the box.
3	Used to signal that the message contains information about the TCP socket opened for the client.	2 bytes for the number of bytes in the message, 2 bytes for the port and one byte per character for the IP address string. The IP address is sent as a string for convenience as it could be an IPV4 or IPV6 address. Sending it as a string makes it trivial to display it on the website. As it is only sent once per connection, the overhead doesn't matter.
4	Used to signal that the demodulation was changed.	None
5	Used to signal that the bandwidth was changed.	None
6	Used to signal that the tuning frequency was changed.	None
7	Used to signal a message containing server information.	4 bytes for the tuning frequency, 4 bytes for the SDR bandwidth, 4 bytes for the max bandwidth, 4 bytes for the max offset, 4 bytes for the bandwidth step, 4 bytes for the spectrum overview resolution and 4 bytes for the spectrum zoom resolution.
8	Used to signal a message containing Channel information.	1 byte for the demodulation, 4 bytes for the bandwidth and 4 bytes for the offset.
9	Used to signal that the client box position was changed.	None

Table 5.1: Server message codes

First byte	Usage	Next bytes
1	Used to signal that the message contains complex float values representing the raw samples.	4 bytes for the number of samples followed by 8 bytes per sample (complex float format).

Table 5.2: TCP socket message codes

First Byte value	Usage	Next byte(s)
0	Used to request a change to the demodulation.	1 byte for the demod type (0 → FM, 1 → AM, 2 → RAW)
1	Used to request a change to the bandwidth	4 bytes representing the new bandwidth as a float (in Hz)
2	Used to request a change to the frequency offset	4 bytes representing the new frequency offset as a float (in Hz)
3	Used to signal a request for TCP socket information	None
4	Used to signal a request for server information	None
5	Used to signal a request for channel information	None
6	Used to signal a change in the selection box position	4 bytes for the left offset and 4 bytes for the right offset

Table 5.3: Client message codes

5.2 Theoretical data rates

Maximum theoretical data rates can be computed if some parameters are assumed. Let us first assume that the data corresponding to commands are negligible compared to the data coming from the samples themselves, let us also assume that the header bytes are also negligible. In this case, the bandwidth required per client for transferring both the values to display for the whole spectrum FFT and for the zoomed in spectrum can be computed with the following formula :

$$\text{rate [bytes/sec]} = \frac{ZR + OR}{f} \quad (5.1)$$

with ZR the resolution of the zoomed FFT, OR the resolution of the overview FFT and f the update frequency.

The bandwidth required for the FM/AM data is very simple to get as it is fixed (PCM encoded audio). Indeed, the sample rate of audio playback is fixed at 48000 Hz and there are 4 bytes per sample so 192000 bytes/sec are required for the audio stream.

In total, the required bandwidth per client using FM/AM demodulation would be :

$$\text{rate [bytes/sec]} = 192000 + \frac{ZR + OR}{f} \quad (5.2)$$

with ZR the resolution of the zoomed FFT, OR the resolution of the overview FFT and f the update frequency.

For a resolution of 2048 values for both displays updated every 500 ms, the total bandwidth required is 200192 Bytes/sec ≈ 200 KBytes/sec.

In the case of raw samples being asked by the client, the bandwidth requirement for the samples is simply equal to 8 times the channel bandwidth as each sample is a complex float. The total bandwidth requirement in this case can be computed as follows :

$$\text{rate [bytes/sec]} = 8 \times BW + \frac{ZR + OR}{f} \quad (5.3)$$

with BW the client channel bandwidth, ZR the resolution of the zoomed FFT, OR the resolution of the overview FFT and f the update frequency.

For the case of a channel with a width of 120 KHz and the same FFT configuration as previously, the total bandwidth required per client would be 968192 Bytes/sec ≈ 950 KBytes/sec.

Chapter 6

Back-end server

In this project, the back-end server is responsible for interfacing with the Software Defined Radio (SDR) as well as processing the received samples and handling the establishment of a websocket connection and TCP connection with the clients.

The samples processing is divided into two parts. The first one is a Digital Down Converter (DDC) whose role is to take as input the samples coming from the SDR and give as output the samples corresponding to a specific channel a client could ask. Each client has its own channel. The second part of the samples processing is taking the output samples from the DDC and applying a demodulation to them.

The network handling part of the back-end server is tasked with accepting new websocket connections with clients and maintaining them as well as opening a TCP socket for each ongoing websocket connection.

6.1 Architecture

The back end of the application is divided into multiple classes. Each class handles a specific task, this allows to easily understand which piece of code is responsible for which system. The whole back end is made in C++ using different libraries to handle each part. All of the chosen libraries as well as the reason they were chosen are explained in the following sections.

In order to understand the architecture of the back end, it is important to first understand what type of data is being worked with and what kind of data is expected at the output.

The collected data comes from an SDR in the form of complex numbers. The rate at which these numbers arrive is controlled by the configuration file and depends on the frequency range the SDR is currently analyzing. This bandwidth is centered around a tuning frequency, which is also configured in the configuration file of the server.

The data type wanted at the end of the pipeline will depend on the mode the client selected. There are two possibilities, either the end data is in the form of float values representing the audio signal (PCM) or the end data can be complex samples representing the signal, before demodulation, in the specific band the client has chosen.

The audio samples are simply sent through the websocket connection so that it can be played in the client's browser. The complex samples are made available via a TCP socket which the client will need to connect to.

To process the data, the different parts of the back end will work one after the other in the pipeline. The next sections will go more in detail for each of the parts. Figure 6.1 gives an overview of what the back-end server architecture looks like at a high level.

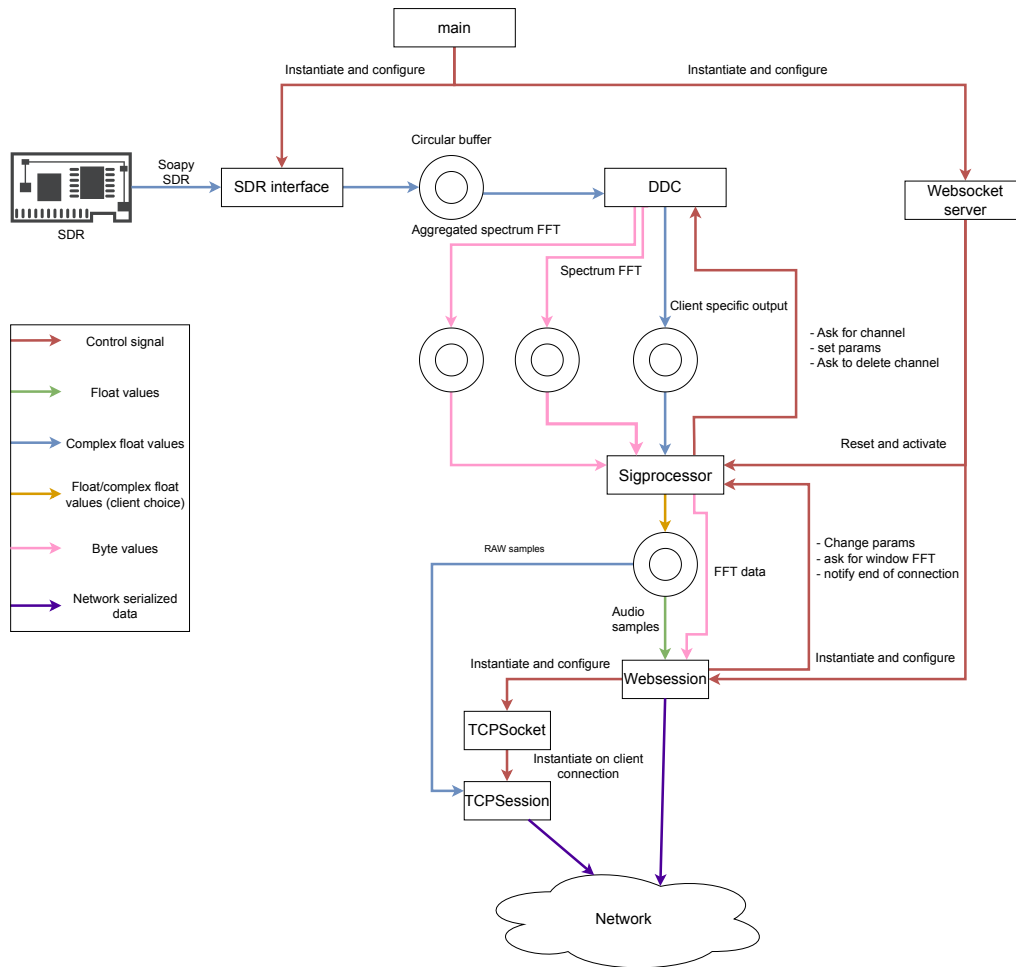


Figure 6.1: Back end architecture

As a side note, some values are defined via `define` statements in various header files. They were not put inside the configuration file as they were not deemed to be susceptible to change much.

6.1.1 Circular buffers

The way data is transferred from one part of the pipeline to the other is with the use of circular buffers. The idea is that the writer needs to be able to write at full speed without having to wait, while the reader may fall slightly behind as long as he keeps up with the overall rate of the writer.

To handle this without race conditions, locking each cell of the circular buffer for either shared reading or exclusive writing would be necessary. The issue with this setup is that the writer may be blocked by a particularly slow reader (or stalled reader). This would, in turn, block all the other connected readers as they would be waiting for the writer to write. Instead, an approach without any lock and thus, without any protection against data races was chosen. In the context of this project, this is perfectly acceptable as there is only one writer and, if a reader is particularly slow, he would simply be catching up with the newer data as the old one is being erased while it is reading. Of course this would mean that one read would potentially contain invalid data but it would be barely noticeable on the client side as one chunk does not represent a lot of time.

There are 2 different scenarios that can occur when trying to read a particular cell of the circular buffer. The first one is if the cell is not where the write head is, in this case the content of the cell is simply returned. The second possibility is when the cell is the one where the write head is. This means that the reader has caught up to the writer and needs to wait until new data has been written. In that case, the reader will wait for a notification from the writer to avoid busy waiting. A timeout delay is also in place to avoid having to wait on a terminated writer. This allows all the readers to eventually exit the wait condition and terminate properly. This timeout has been chosen to be 500 ms. This chosen value is rather arbitrary as it does not have a significant impact on the code. It should just not be too high as to not incur a huge delay when shutting down the server and should also not be too small as allow the writer enough time to write if it hasn't been terminated.

There are 3 types of circular buffers being used in this project. All of them work on the same base principle and accept either floats, complex floats or `uint8_t`. They vary in how they handle reading and writing. Some write methods will copy the data while others will merely copy the pointers. Pointer copy has been privileged as much as possible as they are vastly more efficient but sometimes, value copies were necessary as the circular buffers were used to bridge between two different data formats for the same type.

The circular buffer between the SDR interface and the DDC copies only the shared pointer inside the circular buffer.

The circular buffers between the DDC and the signal processors copy the values when writing because the data format used by the DDC library requires a specific type of complex floats. Reading from that buffer is done by returning a pointer.

The circular buffers between the signal processors and the web sessions copy the values if they are float values so that they are made into shared vectors. If they are complex values, writing them is done by copying the pointer. Reading from them is always done by returning a pointer.

Finally, the circular buffer used to store the FFT of the whole window writes by copying a pointer and reads the same way.

One final note is that, over the course of the evolution of the project, the requirements for the data transfer have been changed slightly. Instead of having multiple readers like initially envisioned, only one reader per buffer is now used. This means that circular buffers are not the simplest nor the most efficient way to handle the data transfer but, as they were already implemented, they were kept in the project. Section 10.2 contains a way to improve this part of the project.

6.1.2 Main

The main function will handle the loading of the configuration parameters from the configuration file as well as launching the necessary threads. It will then wait for a user input to terminate the server. Some basic checks are made for some of the configuration parameters but the user should take care to enter values that make sense. This project assumes an operator that has some signal processing knowledge and knows the hardware he is working with as to not input incorrect configurations.

6.1.3 SDR interface

The `sdrInterface` class serves the purpose of communicating with the SDR card via `SoapySDR`. `SoapySDR` is a library that allows to interface with a wide variety of SDR's as long as they have support. This makes swapping out one SDR for another extremely easy as no changes to the code should be required. As `Soapy` acts as an intermediary between the card and the `sdrInterface`, it could lead to slight bottlenecks. However, as the XTRX is an old card without any form of support, `SoapySDR` was still chosen despite this potential bottleneck as changing SDR was likely in the near future. Directly working with the SDR driver functions would make the code harder to adjust to a new SDR.

The class will create a session with the SDR card and then configure the card according to the configuration file. It will continuously retrieve data from the card through `Soapy` and make it available in the object's circular buffer. As a side note, the XTRX driver seems to be buggy and requires an offset for tuning the frequency, this has been accounted for in the code but other SDR's should not require this. The code will detect if the current SDR is an XTRX and adjust the settings accordingly.

The main processing loop of this class will simply read a certain fixed amount of samples (which depends on the SDR itself) at each loop iteration. It will then put them into a temporary buffer so that they are sent to the circular buffer only when there are the same amount of samples as the FFT size used in the DDC. This ensures a constant amount of samples in each block of the circular buffer as well as blocks that are not too small, even if the transfer block size of the SDR is small.

There is one `sdrInterface` object running for the whole server.

6.1.4 Digital down converter

The digital down converter (DDC) is tasked with taking the samples collected on the whole frequency band analyzed by the SDR and outputting samples for a specific part of the band asked by a client. The DDC implementation is taken from Sylvain Azarian's Libgkr4gpu¹, a CUDA powered DDC implementation that perfectly fits the needs for down converting of this project. Additionally, the library is thread safe, which is also perfect in the context of this thesis. This library has been chosen because of the ease of use it provides as well as being one of the only ones that leverages GPUs to process the samples.

The DDC class will handle the allocation of new channels, the deletion of existing channels, the changing of parameters on channels as well as the feeding of samples to the DDC and the retrieval processed samples for each channel and making them available in circular buffers for the next part of the pipeline. There will be one channel per client as each of them is susceptible to ask for a different part of the spectrum.

The DDC class will keep a translation table between channel ID's provided by Libgkr4gpu and the channel ID's given to the rest of the pipeline. This is necessary as changing the bandwidth of a channel is not supported by the library, instead, the channel needs to be deleted then recreated with the new bandwidth. For this process to be completely transparent, a mapping needs to be made as the library may give a new channel ID. Each allocated channel will have its own circular buffer where output samples will be stored. As the maximum amount of clients is known at server launch, all the circular buffers are allocated in advance as to limit setup time when a new client arrives. The buffers are simply reset when it is reused.

To allocate a new channel, concurrent requests need to be handled correctly as multiple clients may want to create a channel at the same time. Even though the library itself is thread safe, the ID's given to the signal processor that requested a new channel also need to be unique. To achieve this, a simple lock was used that will make sure clients are processed one at a time as to not have conflicting channel ID's.

For each channel, only one configuration can be done at a time (i.e. the bandwidth and the frequency cannot be changed at the same time). This is ensured by having a mutex for modification for each channel.

One limitation Libgkr4gpu has is that the ratio of the input and output sample rates needs to be an integer. This limitation makes it necessary to convert the arbitrary bandwidth requested by the client into one that is the result of the division of the input sample rate by an integer. This part is handled in the signal processor.

¹Github repository

Another task the DDC class has is to compute the FFT of the whole spectrum with a resolution given in the configuration file. The result of this computation is in the form of float values. These will be normalized on a scale of 0.0 to 1.0. These values will then be put on a 256 level scale with values ranging from 0 to 255 so that they occupy much less space. Once this is done, the values will be made available in a circular buffer.

The DDC will also produce and make available a version of the data which is contained in a vector of a size specified by the `clientOverviewSize` option in the configuration, this version is used by the client to display the data on the displays without having to send too much data through the network. If this option stipulates a size that is equal to the FFT resolution, then nothing needs to be done. If the option stipulates a size that is bigger than the FFT resolution, then values will need to be repeated. However, the most likely case is when the size of the output vector is smaller than the FFT resolution.

Two ways were devised to solve this. The first one involves simply taking equally spaced samples from the original vector. This has the advantage of being very simple and not computationally intensive.

The second way is to group samples together in groups that will allow to use all samples while having the same amount of groups as there should be elements in the new vector. Once the groups are made, an average is computed and the resulting value is taken to be the value in the new vector. To make this work for arbitrary sizes for both vectors, all groups are not necessarily of the same size. This way of solving the array size issue requires more computation but provides a smoother display. In practice, the groups are made by first computing a step ($\frac{\text{input size}}{\text{output size}}$). The values from the first vector used to compute the average for the i^{th} cell of the output vector are the ones ranging from the one following the last one used to the one at index $\lfloor i * \text{step} \rfloor$ (this interval always contains at least 1 value as the input size is greater than the output size). This method allows to pick all values of the input vector from the first one to the last one.

When testing both approaches, it appears that the first one did not yield satisfactory visuals on the client displays and made it hard to discern signals. The second one was thus chosen despite it being less efficient.

The library used to perform the FFT is FFTW3 as it is the fastest available library to compute FFT's. As this computation happens rather rarely (default settings at 500 ms interval), it is perfectly acceptable to run it on the CPU as it will not incur a huge computational load. There will be one dedicated thread for this task as well as one dedicated thread for the main data loop that takes input samples from the SDR interface and makes output samples available in each channel's circular buffer.

6.1.5 Signal processor

The `sigProcessor` class serves the purpose of taking the DDC output and applying any (if at all) signal processing steps the user may ask for from the list of available ones. The `sigProcessor` objects are instantiated in advance and are simply reset when a new client begins using them. This allows much simpler memory management as the memory simply stays allocated. It also means that setting up a new client is faster as the object does not need to be created.

After resetting, the `sigProcessor` asks the DDC for a new channel so it can start processing samples. It then loops until a termination signal is received. This loop simply takes samples and, according to the demodulation requested by the client, processes it using *GNURadio* blocks. *GNURadio* has been chosen as it is the most mature and efficient signal processing library available in C++. Not all *GNURadio* blocks are available as C++ blocks (many of them are only made for Python) but all the needed blocks can be built using simpler blocks. It also supports vectorized instructions so processing batches of data is much faster.

The data processing loop takes ownership of a lock at the start of the loop and relinquishes it at the end to avoid changing the configuration (and *GNURadio* blocks) during processing as this would lead to a segmentation fault. When a thread wants to change a parameter, it needs to first wait for the current loop to finish before taking ownership of the lock and then changing the parameter. The issue that arose here is that the loop is so fast, that it barely leaves any time for other threads to acquire the lock. The solution was to implement and use a FIFO lock. As there is only ever one instance of the data loop, the thread trying to change the parameters is guaranteed to have access the next time the loop is done. This may lead to the loop having to wait a bit before proceeding to another iteration but it is acceptable as the user wanted to change what it computed anyways.

The output data will be put in one of two circular buffers. Either one containing complex samples or one containing real samples based on the demodulation asked by the client.

The output for both the AM and FM demodulation is PCM encoded audio samples represented by float values. This is what will be sent to the client. No compression is applied as it makes it easier to play the audio in real time on the client's browser. The other reason why using compression would not make much sense is because the user can ask for raw samples. In that case, the data rate required would be much higher anyways. Optimizing the data rates for audio would effectively just make the best case scenario better but would not improve the worst case scenario. The difficulty of implementing compression has thus been deemed not worth the benefits.

The FM demodulation blocks are arranged as shown on figure 6.2. The quadrature demodulation has a gain of $\frac{BW_I}{2 \cdot \pi \cdot f_k}$ with BW_I the bandwidth at the input and f_k the maximum deviation (constant with regards to a geographical region). The re-sampling rate of the re-sampler is simply given by $\frac{48000}{BW_I}$ as the output sample rate wanted is 48000 Hz (for audio playback). The taps for the filter of the re-sampler are obtained by using the finite impulse response filter designer from *GNURadio* with a cutoff of 15000 Hz and a transition width of 2000 Hz. The de-emphasis filter taps are obtained by using formulas from the *GNURadio* implementation²

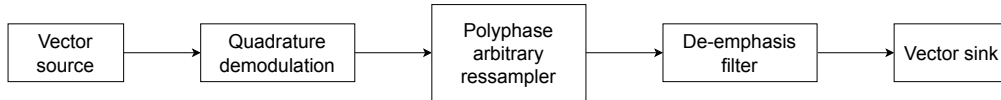


Figure 6.2: FM demodulation pipeline

The AM demodulation blocks are arranged as shown on figure 6.3. The sine wave generator is at the same sampling frequency as the input signal and has a frequency equal to the center frequency of the input signal. The re-sampler has the exact same configuration as the one used for the FM demodulation.

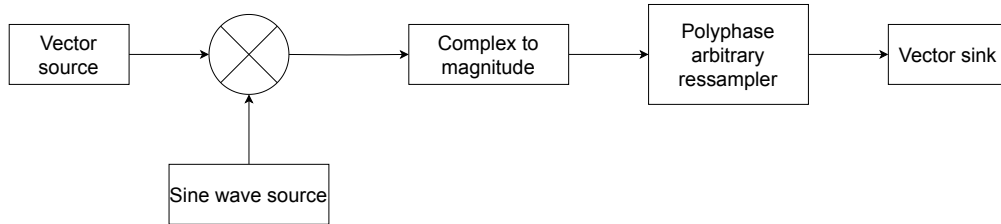


Figure 6.3: AM demodulation pipeline

One important point is how the channel bandwidth is set. As explained before, the DDC library only supports integer ratios between input and output sample rates. This implies that only select values are valid for the bandwidth chosen by the client. In order to allow future work on the project that would allow arbitrary ratios, the user is still allowed to input any value he wants for the bandwidth, the `sigProcessor` will handle the conversion to a valid value. This conversion is very simple, a vector contains all the valid values and is searched for the closest value to the one provided by the client, once it is found, it replaces the value given by the client. This vector is built at server launch. The SDR bandwidth is divided by increasing integers and the values obtained are stored in the vector. This process stops as soon as the result obtained is smaller than the smallest allowed bandwidth, which is configured in the config file.

²Link to the Github file.

Another point of discussion is the static type conversion happening when setting the tuning frequency of the channel. The decision has been made to work with only integer frequency values as float values do not change the perceived tuning frequency compared to the nearest integer. When setting the new tuning frequency, the float value is cast to an integer. Another change that is made is that the center frequency is adjusted so that the channel stays within the bounds of the whole spectrum analyzed by the SDR. To do this, the center frequency is adjusted to be :

$$F \in [-O + B; O - B] \quad (6.1)$$

with F the adjusted tuning frequency, O the maximum offset ($\frac{SDR\ bandwidth}{2}$) and B the channel bandwidth. This means the tuning frequency may get adjusted when the bandwidth is increased and would make the analyzed band go out of the analyzed spectrum.

The last point of discussion is how the `sigProcessor` handles the preparation of the buffer containing the values to be displayed on the client waterfall displays. This buffer is divided into two parts, the first part contains the aggregated values computed by the DDC which will be used to display the spectrum overview on the first client screen. These values are simply retrieved from the circular buffer in which the DDC put them. The second part of the buffer contains the values for the zoomed spectrum and need to be computed individually for each client as it depends on the part the client is currently zoomed in on. To handle this, the server needs to know what the client is currently looking at. Once the server knows this, it takes the vector containing all the FFT samples computed by the DDC and slices it accordingly to only take the values in the desired range. It then performs the same aggregation as previously done in the DDC and appends the result to the buffer to be sent to the client.

There is one `sigProcessor` object per client.

6.1.6 Websocket server

The `websocketServer` class is designed to listen on a specific port for new client connection requests and, once a request is received, it will allocate necessary resources for the websocket connection. Boost/Beast's implementation of websockets has been chosen for this part as it is a mature and proven library that has good documentation and is rather easy to use.

The `websocketServer` object will, every time a new client tries to connect, allocate a new `webSession` and assign a `sigProcessor` to it. The `sigProcessors` are pre-allocated and simply reset between uses. Both of them are then run on a thread from Boost's implementation of thread pools. The reason for which the `webSession` objects are not pre-allocated is because they are setup with a specific client on the other end, so they would not be reusable with a different client.

When a request is received, a lock is obtained and the first available `sigProcessor` is assigned to the task. The lock ensures that each `sigProcessor` only handles one client. If no `sigProcessors` are available, then the connection is not established. A new `webSession` object is then created to handle the rest of the handshake. A flag is immediately set in the `sigProcessor` object to signal that the `sigProcessor` is currently in use.

When creating a new session, a port needs to be chosen for the associated TCP socket that will be created in order for the client to be able to receive the raw samples. The available ports to pick from start from a value given in the configuration file and go up to that value plus the maximum number of clients. To pick a port for a particular client, the first available port is picked, this is simply checked by trying to open a session with it and see if it is successful. TCP connections can linger for a bit, even when closed (in case some packets arrive late), meaning that a simple flag will not suffice to know if the port is truly available.

A single `websocketServer` object is created for the whole server and the number of threads running on it depends on the configuration file. However, only one connection can be established at a time, others will have to wait for the lock to be unlocked.

6.1.7 WebSession

The `webSession` class handles the communication with clients by playing the role of the end point of the websocket. It handles the transmission of the data to the client and it acts in accordance with client commands. The first step the `webSession` does once it is created is accepting the websocket connection request and performing the handshake.

To send the data through the websocket, it needs to be prepared as only one websocket connection is opened but the different data streams and the control stream share the same connection. To handle this correctly, 2 principles need to be followed :

1. Making sure only one message is sent at a time (i.e. no interleaving of messages). This is ensured by taking ownership of a lock before trying to send anything in the socket and relinquishing it after.
2. Making sure the receiver knows what type of data has been sent as well as the length of the message. For the data streams, this is ensured by having the first byte being a code for the type of data and the next 4 bytes being the number of elements in the message. For the other messages (control messages), the number of samples is not needed as message structure is fixed in that case.

Table 5.1 contains the information relative to the message codes. This method of serialization was chosen because it wastes the least amount of data with each message.

Particular care needs to be taken with regards to endianness. Everything is sent in big endian format as that is the accepted network order for bytes. Another point of importance is to set the websocket to binary mode as all data is sent in binary format.

The `webSession` object constructor will open up a TCP socket on a port unique to this connection. This TCP socket will listen for a connection. Once the client connects to it, a TCP session is created for the complex data samples to be transferred. The destructor of the `webSession` tries to join all the threads before destroying the object.

The main data loop is rather simple, depending on the mode currently selected by the user, the `webSession` object will fetch data from the appropriate buffer in the `sigProcessor`. When changing parameters following a client command, it is possible that some data frames will be skipped because buffers are reset to their initial state while a data loop may already be executing. This is not an issue as null pointers will simply be ignored and the next iteration of the loop will proceed correctly. The alternative to this would be to use locks but that would add overhead for solving a non-issue.

In addition to the main data loop and the TCP socket that both run on separate threads, one additional thread runs to handle the FFT overview values. Each connection thus consists of 3 software threads to handle everything. There is also an asynchronous function running for receiving messages.

The data coming from the client consists of commands to change the current parameters. The possible messages are shown in table 5.3. Whenever a client message is received, the appropriate actions are taken to change the parameters. An ACK message is sent once the parameters are changed.

6.1.8 TCP socket and TCP session

These two classes handle pretty simple tasks : accepting a new TCP connection and sending the raw complex samples if the raw mode is selected. To do this, Boost's implementation of the TCP socket has been chosen for the same reasons as it was chosen for the websocket.

The `tcpSocket` is very simple, it listens for an incoming TCP connection and when it happens, it creates a session and runs it. It does not accept any new incoming connections until the one currently running is over.

The `tcpSession` is very straightforward, it simply executes a loop that will check if the current chosen mode is raw samples and, if it is the case, it will get samples and write them to the socket. It does not read anything on the socket as the client is not supposed to send anything. If the current mode is not raw samples, it will go to sleep for 500 ms and check again when it wakes up. This is done to avoid busy waiting.

6.1.9 Object lifetime management

Object lifetime management is very important to avoid memory leaks as clients connect/disconnect. This means that `webSession` objects as well as `tcpSession` objects and `tcpSocket` objects need to be properly managed. Shared pointers were used to keep references to all of those objects as it eases memory management.

When creating the `webSession`, the thread launched for it holds a reference to the object itself. The `webSession` also holds a reference to the `tcpSocket` it has created.

The signal to destroy the connection (basically, an error on write or read) can come from multiple places, possibly multiple times for the same object.

The first place where it could happen is in the `tcpSession`. In this case, the termination flag for the `tcpSession` object is set and the socket is closed. Once the flag is set, the reference to the `tcpSocket` is removed and the function returns as the loop condition is not verified anymore. This allows the `tcpSocket` to accept a new connection. As it is not considered the "main" part of the connection (which would be the websocket), there is no need to terminate the websocket connection.

The second place where it could happen is in the `webSession` object itself. Many different operations could lead to having to terminate the connection. All of these lead to the execution of the `stopSession` function. This function will try to get ownership of a lock, if it succeeds, it proceeds to set all the flags for termination and tells the `tcpSocket` to terminate as well. It also tells the `sigProcessor` to stop processing. All of these actions are only done once, if another thread executes the same function again, the actions will not be repeated. If the locking does not succeed, then the function simply returns as it means another thread is already performing the necessary actions. As all references to the `tcpSocket` and the `tcpSession` are removed and their threads are stopped, the objects will be destroyed. The destructor of the `webSession` object ensures all threads are correctly joined.

Chapter 7

Website

The website is the front end. What is presented to the end user. It needs to allow the user to control the bandwidth of his channel, the tuning frequency of his channel as well as the demodulation. It also needs to show the user information about the server as well as information about the TCP socket prepared for him. It should also show the overview of the whole spectrum received by the back-end server and play the audio samples received from said server.

To handle this, a Node server has been setup. This allows the website to be built using Javascript which is perfectly suited for the needs of this project. The next sections will each delve deeper into specific parts of the website code. They will explain why some choices were made and the overall inner workings of the website.

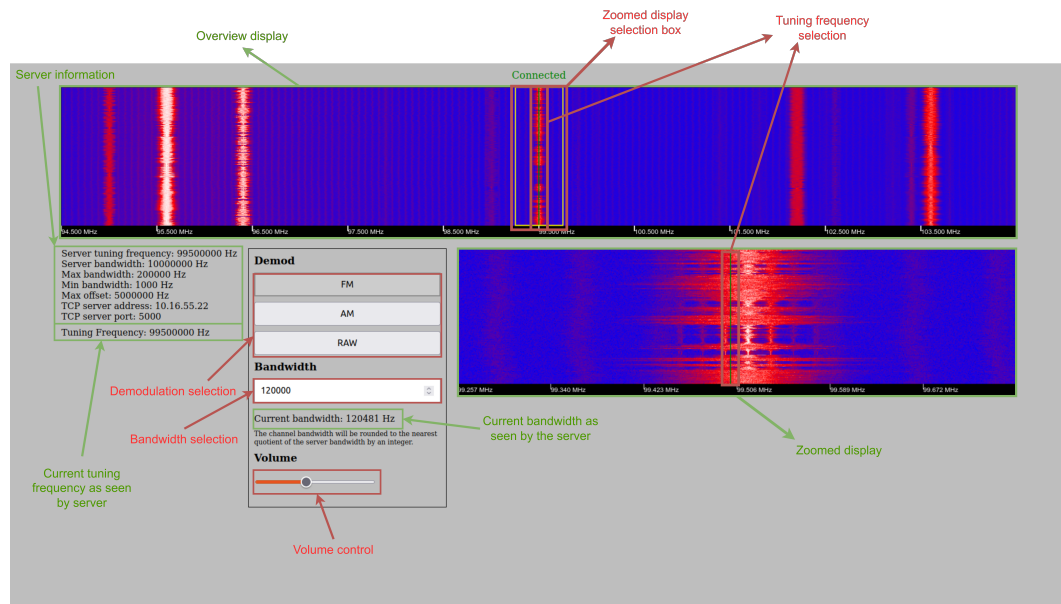


Figure 7.1: Website user interface

7.1 Connection to the back-end server

In order to be able to receive any data from the back-end server, the client must open a websocket connection to it. This is done as soon as the web page loads. Once the websocket connection is open, the client will ask for server information and TCP socket information in order to display it on the web page.

Each time a message is received, the first byte is extracted and the appropriate action is taken. If the message contains PCM audio samples, they are buffered to be played (see section 7.2). If FFT samples are received, they are used to render a new line on the displays (see section 7.3). If server information or TCP socket information are received, the corresponding elements are updated in the HTML of the page. Finally, if channel information is received, all the relevant elements are synchronized with the server values (see section 7.5).

The websocket connection is maintained as long as the web page is running. The status of the connection is shown to the user at the top of the web page.

7.2 Audio stream

The audio samples coming from the server need to be played with minimal latency so that it can be considered live. A couple seconds delay is acceptable of course. The samples received are in the form of PCM audio, which is very easy to play in Javascript using an `audioContext` object.

This `audioContext` works by providing it with audio nodes, which are buffers containing the PCM samples as well as the information of when they should start playing and to which output device they should play.

The scheduling is rather simple, each audio packet will be played after the previous one. The first packet however, is not played immediately. The script will wait for 500 ms of audio to be buffered before starting to play audio. This is made to ensure there are no stutters in the audio stream.

If the amount of buffered audio samples fall below the equivalent of 50 ms of audio, the audio playback is paused and will be resumed once 500 ms worth of audio is available in the buffer.

When the user changes a setting (like the bandwidth, tuning frequency or de-modulation), all the audio buffers are flushed as they contain outdated samples. Audio is then resumed once 500 ms of audio is contained in the buffers.

7.3 Waterfall displays

There are 2 displays on the website, one shows the FFT over time for the whole spectrum received by the back-end server, the other zooms in on a particular part of the spectrum to allow the user to more clearly see what happens in that window. Both of them are waterfall graphs.

The data received from the server has a part dedicated to the first display (the spectrum overview) and a part dedicated to the second display (the zoomed spectrum). This data is in the form of equally spaced points on their respective frequency ranges. The values are bytes representing the magnitude on a scale of 0 to 255 for each point. The client simply needs to convert these magnitudes to RGB values.

To make rendering efficient, each time a new line of data is received from the server, all previous lines are shifted upwards and the new line is inserted at the bottom.

Additional information is also drawn on the displays such as graduations, current tuning frequency as well as the selection box for zooming in on the first display. The first display allows the user to make a selection box slide across the full spectrum so that a specific part gets zoomed in on. This zoomed in part is displayed on the second display. Whenever the user grabs the box to make it slide across the first display, the script will set a flag so that it knows the user is grabbing the box. When releasing the box, the script will send the new box position to the server but only if the box has actually changed position compared to before it had been grabbed.

Both displays also allow to click on any point to put the tuning frequency at the selected frequency. A vertical line will be drawn at that location to let the user know the current tuning frequency. As soon as a new tuning frequency is selected, a conversion is made from position on the canvas to an offset in frequency compared to the server tuning frequency. Then, a command is sent to the server to change the tuning frequency.

To adjust the number of values received to fit the exact number of pixels that are required on the client's screen, the procedure is similar as to how it was done for aggregating values in the DDC.

7.4 User controls

In addition to the tuning frequency control explained in section 7.3, the user has access to demodulation controls, channel bandwidth controls and volume control. The demodulation control is in the form of 3 buttons allowing the user to select one specific mode while the bandwidth control is in the form of an input box where the user can enter a value. Finally, the volume control is a simple slider controlling the gain for the `audioContext`.

For the bandwidth control, one important feature is that the server will automatically adjust the requested bandwidth to the closest quotient of the total server bandwidth by an integer. This is required as the DDC library used (`libgkr4gpu`) only supports integer ratios between the input bandwidth and the output bandwidth.

The user controls also have a cooldown as to make it impossible for the user to spam commands to the server. This cooldown is fixed at 100 ms so that it is not noticeable to the user during normal operation.

7.5 Synchronization with the server

In order for the user interface and the back-end server to stay synchronized with respect to the current configuration, the API allows the client to ask for specific information. This API allows 3 distinct types of information request :

- The user can ask for general server information. The script will ask this once when the connection with the websocket server is established. As this information should not change, there is no need to ask for it multiple times. Once the information is received, it is displayed as a static text on the website.
- The user can ask for information regarding the TCP socket it has been attributed. Similarly to the general server information, this is only asked for once when the websocket connection is established. It is then displayed alongside the general server information.
- The user can ask for information regarding his specific channel. The server will then reply with the current demodulation, the current frequency offset compared to the server tuning frequency and the current channel bandwidth. With this information, the client user interface can be updated to display all the information. The demodulation information will be used to make the correct button appear pressed, the current bandwidth information will be displayed beneath the bandwidth selection box and the current tuning frequency information will be used to draw the tuning frequency lines on the displays. As these parameters are susceptible to be changed by user commands, a request for them is sent every 500 ms to keep them up to date. 500 ms is enough time as to not burden the server with "useless" requests but is fast enough to be barely noticeable by the user.

7.6 Tests

As there isn't a lot of code for the website, testing of the front end was done by hand. Playing with the website itself and printing debug information. This was sufficient to make sure everything worked as intended.

Chapter 8

Back end unit tests

As the code for the back-end server is quite large and has a lot of multi threaded parts, extensive testing is required to ensure proper operation. To this end, a directory called "test" has been added and filled with various tests. This allows to test various aspects of normal server operation to ensure faster development and debugging cycles as bugs can be identified faster.

These tests have been very helpful in finding a lot of bugs and unintended behaviors. The next sections go over these unit tests in order to explain what happens in each of them. The setup of these tests uses the default configuration values for nearly everything.

8.1 Circular buffer unit tests

All three instances of circular buffers are tested in the same manner as they are practically identical. The tests include the following :

- Creation of a new circular buffer.
- Writing to the circular buffer, this is done using all the different write functions.
- Reading from the circular buffer.
- Checking if the read value and written value are the same.
- Testing if the circular buffer properly overwrites the values once it cycles around.
- Resetting the circular buffer and checking if the reset was properly done.
- Testing the read timeout.
- Testing if concurrent reads work.
- Testing if reading the head of the buffer works as intended.
- Testing if reading can be done while writing without segfault.

Together, these tests account for all the use cases the circular buffers could see.

8.2 SDR interface unit tests

To simulate the context in which this class will run without actually having to start a DDC (so that unit tests remain unit tests), the role of the DDC will be played manually throughout the tests. The tests include :

- Creation of a new `sdrInterface` object.
- Changing the bandwidth, tuning frequency, sample rate and gain.
- Running the main `sdrInterface` loop.
- Basic reading of samples (and verifying the values obtained).
- Reading while the data loop runs.
- Performing data rate tests.
- Saving samples for further tests.

8.3 DDC unit tests

As the DDC cannot run without being fed data, it needs the `sdrInterface` to run so that it can itself perform tests. This means that an `sdrInterface` object will be used to feed data to the DDC. The tests on the DDC include :

- Creation of a new DDC object.
- Launching of the FFT overview thread.
- Checking if the FFT overview data seems valid.
- Launching the main `DDCloop`.
- Allocating a channel.
- Activating a channel.
- Changing the bandwidth and tuning frequency of a channel (and checking if it correctly returns an error if it is tried on non allocated channels).
- Getting the head of a channel's circular buffer.
- Checking if the channel has plausible data.
- Measuring the throughput of a channel.
- Releasing a channel.
- Allocating multiple channels at once.
- Measuring throughput of multiple channels that run at the same time.
- Releasing multiple channels.
- Saving output samples for further tests.

8.4 Signal processor unit tests

The `sigProcessor` also needs the `sdrInterface` as well as the DDC running in order to be able to process samples that are significant. The unit tests include :

- Creation of a new `sigProcessor` object.
- Launching the main processing loop of the `sigProcessor`.
- Changing the demodulation, the bandwidth and the frequency.
- Checking if the FFT data seems valid.
- Checking the FM, AM and RAW demodulation output rates.
- Checking if the FM, AM and RAW output data seem good.
- Having multiple `sigProcessors` on the DDC.
- Stopping `sigProcessors`.
- Saving FM, AM and RAW output samples for further testing.

8.5 Networking unit tests

For testing all the parts related to networking (`webSession`, `websocketServer`, `tcpSession` and `tcpSocket`), the server is launched as it would have been in normal operation and various Python scripts are launched which will execute specific actions to test all the systems. These tests include :

- Creation of a new `websocketServer` object.
- Establishing a new connection with a client (and thus creating a `webSession` object as well as a `tcpSocket` object).
- Asking for general server information, channel information and TCP socket information.
- Changing the demodulation, the bandwidth and the tuning frequency.
- Checking if the data received for FM samples seems valid. The same is done for the FFT samples.
- Saving FM samples received by the client for further testing.
- Opening a connection to the TCP socket and collecting RAW samples.
- Trying to send garbage data to the server.
- Closing the connection gracefully.
- Closing the connection abruptly.
- Opening a connection with multiple clients at once and configuring each of them.
- Running multiple clients at once.
- Quickly connecting/disconnecting a huge amount of clients each second.

8.6 Additional tests

Testing for memory leaks has also been conducted by simply executing the server for a prolonged amount of time (roughly 2 hours) with a lot of clients connecting/disconnecting and verifying memory usage. This test has shown no sign of memory leaks. Another test was performed by adding print statements to the destructors of the concerned classes (`webSession`, `tcpSocket` and `tcpSession`), making a client connect/disconnect and checking if the destructors are correctly called, which was the case.

Chapter 9

Deployment and performance

9.1 Deployment

Deploying the back-end server along with the Node server for hosting the website is very well documented in the readme files provided in the repository. The overall approach is to clone the repository, install all the dependencies one by one, compile the back end, change the configuration file with the required configurations, run it then start the Node server. The only part that could pose some issues is installing the drivers for the various SDR devices. This part will depend on which SDR is being used so no out of the box instructions could be made.

Once everything is up and running on the server at Montefiore, the website will be accessible from inside the university's network at the following address : <http://websdr.mont.priv/>

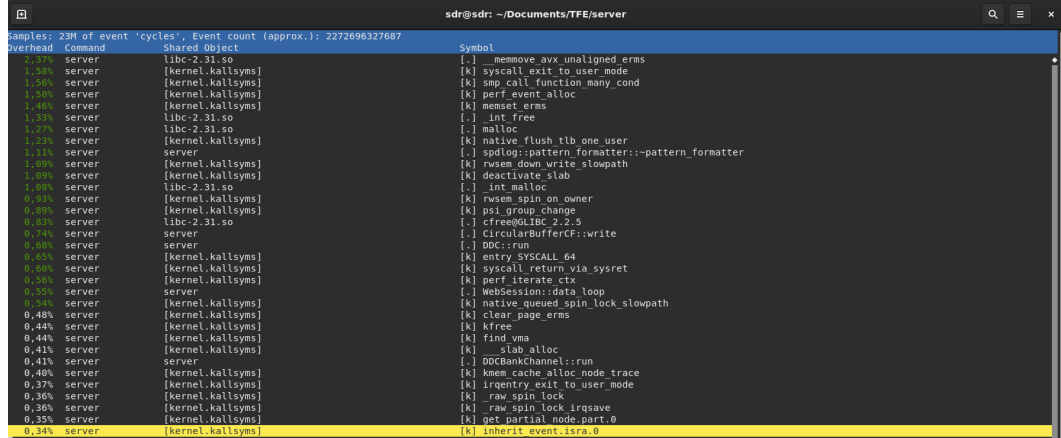
9.2 Code profiling

Code profiling has been attempted using *Linux perf*¹ along with Intel's *VTune*². These sampling profilers are easy to setup and allow to have insights on parts of the code that take up most of the compute time. *VTune* allows various different profiling tests to be conducted and yields a lot of information.

Profiling has been done using the XTRX with the server bandwidth set at 10 MHz with 30 clients connected to simulate a certain load on the server. Running *Linux perf* in this configuration yields interesting results shown on figure 9.1. These results show that most of the time is spent on an instruction related to moving memory around.

¹*Linux perf* wiki

²Intel *VTune* main page

Figure 9.1: Profiling results with *Linux perf* at 10 MHz with 30 clients

When changing the server configuration to analyze a *50 MHz* band with 30 clients connected, the result does not change much. When using the LimeSDR connected via USB, the results only change a bit because of the fact that USB interfacing takes much longer than PCIE transfers so it gets bottlenecked by the USB rate instead of by the memory movements.

The next tests have been conducted using *VTune*. The first choice of test with this profiler was an overview (called a snapshot) of the whole program to have an idea of the next tests to run. The snapshot yielded little new information apart from indicating which tests to run next.

The most important test that has been run with *VTune* is the hotspot test which gives information about which parts of the code consume the most CPU time. The results are shown on figure 9.2.

From these results, it can be seen that the major bottleneck appears to be the signal processing part handled with *GNURadio*. This information did not appear when profiling the code with *Linux perf*. *VTune* also provides more information on the previously reported bottleneck in the form of the call stack. It is now possible to see from where the majority of *memmove* operations come from. It appears they come from the DDC library used in this project (*Libgkr4gpu*). With these profiling results, it becomes evident what parts of the code need to be changed in priority to improve performance.



























Hotspots 				
Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Flame Graph Platform				
Grouping: Function / Call Stack    				
Function / Call Stack	CPU Time 	Instructions Retired	Microarchitecture Usage 	Mod
▼ gr::top_block::start	58.698s 	912,000,000	9.1%	libgnuradio-runtime.so.v3.1
▶ SigProcessor::runFM	58.698s 	912,000,000	9.1%	server
▼ __memmove_avx_unaligned_erms	7.058s 	2,424,000,000	6.8%	libc-2.31.so
↖ memcpy ← DDCBank::put ← DDCBank::	3.194s 	1,704,000,000	5.7%	server
↖ memmove ← DDCBank::step1 ← DDCB	2.493s 	672,000,000	5.4%	server
▶ ↖ memcpy ← SdrInterface::run ← func@0x	0.881s 	24,000,000	8.2%	server
▶ ↖ memcpy ← DDCBankChannel::get ← DD	0.441s 	24,000,000	17.3%	server
▶ ↖ memcpy ← DDCBankChannel::setCenter	0.050s 	0	29.8%	server
▶ DDC::run	5.576s 	1,200,000,000	4.3%	server
▶ DDCBankChannel::run	3.013s 	7,128,000,000	59.8%	server
▶ std::abs<float>	2.893s 	5,232,000,000	30.6%	server
▶ std::log10	2.403s 	6,840,000,000	45.6%	server
▶ DDC::runFFT	1.752s 	2,808,000,000	27.1%	server
▶ func@0x2eefa0	1.622s 	864,000,000	13.7%	libcuda.so.550.54.15
▶ func@0x2e8d60	1.612s 	1,200,000,000	12.8%	libcuda.so.550.54.15
▶ func@0x2ee340	1.542s 	4,560,000,000	60.5%	libcuda.so.550.54.15
▶ func@0x2b0fc0	1.422s 	1,152,000,000	15.5%	libcuda.so.550.54.15
▶ std::vector<std::complex<float>, std::allocator<	1.412s 	576,000,000	11.4%	server
▶ gr::top_block_impl::wait	1.382s 	96,000,000	9.8%	libgnuradio-runtime.so.v3.1

Figure 9.2: Hotspot results with *VTune* at 10 MHz with 30 clients

One such part is the signal processing part where the output from the DDC gets demodulated. To improve this, more efficient blocks could be used or an entirely new library could replace *GNURadio*. The issue here is that such a library is hard to find. An alternative would be to develop a rudimentary signal processing library with performance in mind which would cater to the needs of this project in particular.

Another part that could see improvements would be the DDC part of the code. The library used seems to move memory around a lot. This is due to the library requiring objects to be aligned in memory before use. This could be improved as the source code of the library is available. Another way of improving this would be to only work with aligned objects all throughout the project. However, it is very clear that the most CPU time is spent on signal processing so changing or improving the DDC library should not be the priority.

9.3 Performance in practice

Performance of the back-end server has been measured using a Python script that would open increasing amounts of connections to the server acting as clients. All of these clients ask for a *100 KHz* band with FM demodulation on a random tuning frequency. The FFT size for computing an overview of the spectrum was set at 131072 points with an update interval of 500 ms. The number of values sent to the clients for both the overview of the spectrum and the zoomed spectrum was set at 2048. To estimate the performance, each client keeps track of the total amount of samples received. Each connection lasts for 6 minutes and the average rate is computed for each client. Once all clients have finished, an average of the rate of all the clients is computed and displayed on the following plots. This is repeated with increasing values for the total bandwidth of the server. The reception rate of the clients should be *48000 samples/sec* if the server is capable of handling them. As soon as this value drops, it indicates that the server is incapable of handling the load.

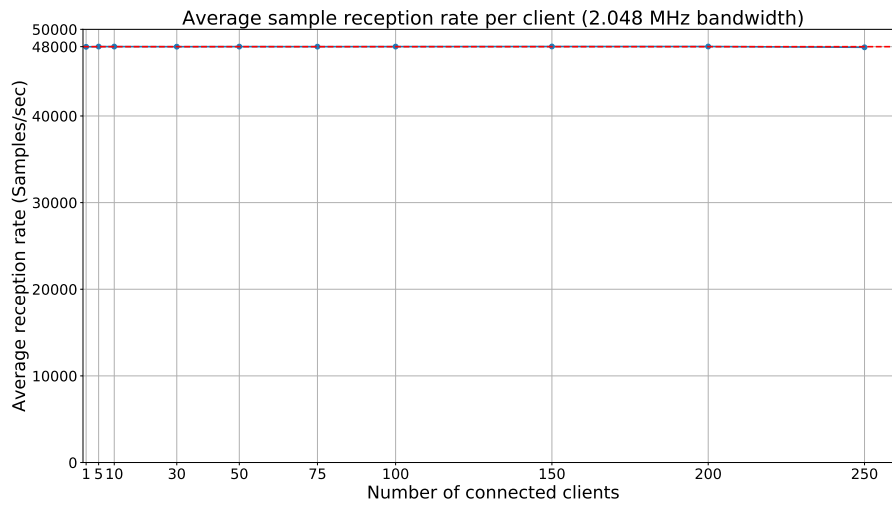


Figure 9.3: Performance metrics for 2.048 MHz bandwidth

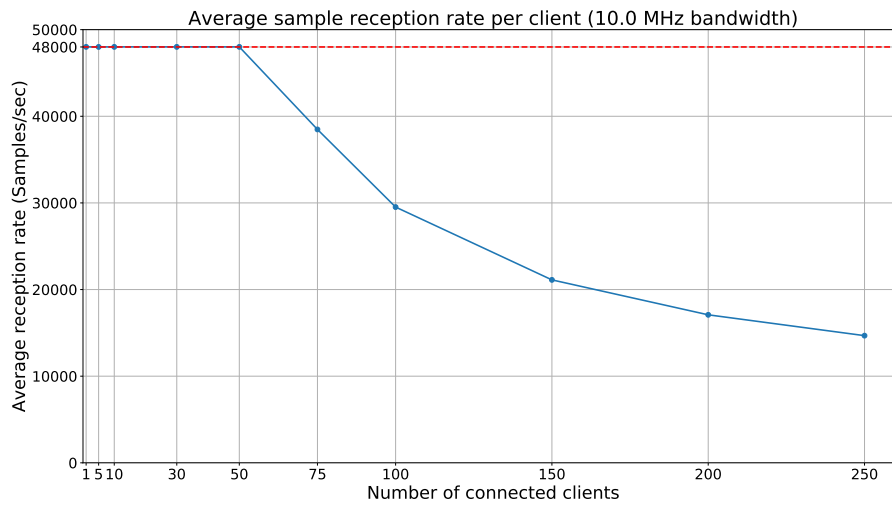


Figure 9.4: Performance metrics for 10.0 MHz bandwidth

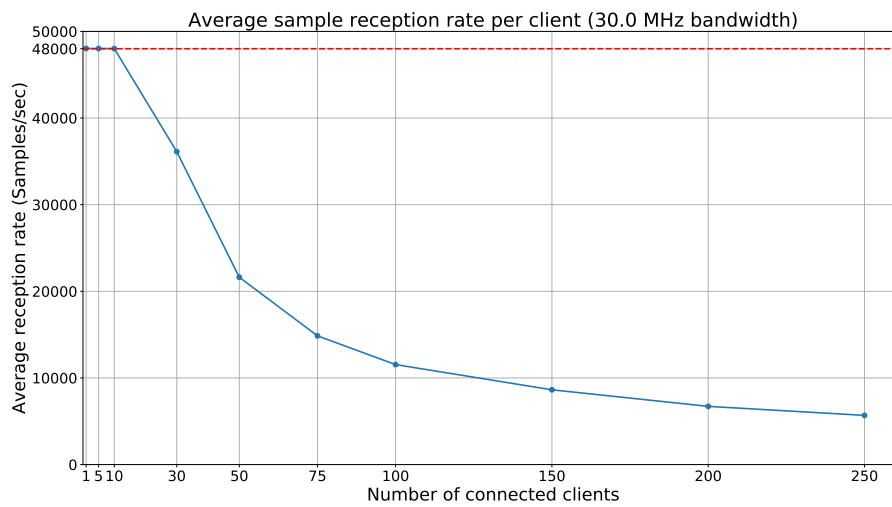


Figure 9.5: Performance metrics for 30.0 MHz bandwidth

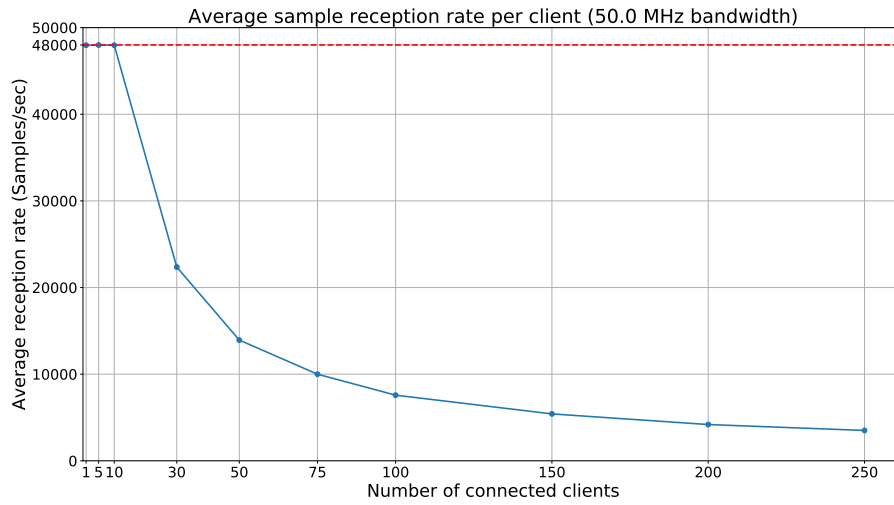


Figure 9.6: Performance metrics for 50.0 MHz bandwidth

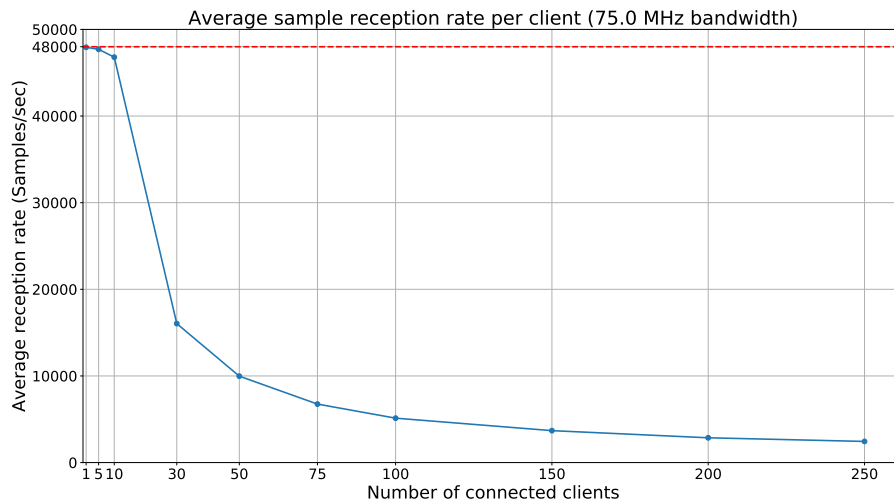


Figure 9.7: Performance metrics for 75.0 MHz bandwidth

These performance figures show that the back-end server is not able to handle very high bandwidth for a large number of clients. A good threshold for the bandwidth would be 10 MHz as this allows roughly 50 people to use the website at once. For the purpose of this project, it is sufficient. Further optimizations can be conducted to handle more clients.

To ensure that the performance results obtained here actually amount to anything in real use cases, another test has been conducted where 45 clients were connected

via the same script and one real user played around on the front end to make sure everything worked correctly when the server was under heavy load.

9.4 Known issues

This project has a few issues that unfortunately have not been fixed in time for the deadline. These issues are not severe issues and do not prevent the project from working. Said issues are listed below :

- When running at high bandwidth (starting at roughly *50 MHz*) with a lot of clients, the back-end server sometimes has the following runtime error : `gr::vmcircularbuf:error:shmat(2):Invalidargument`. This error is linked to shared memory attribution. This specific error happens in the GNURadio library so it is hard to check where exactly it comes from. Some tests have been conducted such as increasing the maximum amount of shared memory and the maximum size of shared memory segments. It does not appear to impact the server operation as the server keeps running despite the error being reported.
- The machine may sometimes (although very rarely) crash on server startup. This only happens on server startup when the SDR is getting initialized. This happens only with the XTRX meaning that driver code is the most likely culprit.
- When using the XTRX, the tuning frequency of the server is offset by a value equal to the bandwidth selected. This is most likely due to a bug in the XTRX driver as all other SDR's tested did not have this behavior. The code has been modified to account for this when using the XTRX.
- The XTRX cannot be configured to use a sample rate of *100 MSPS* despite it being advertised as being able to do it. This is, again, most likely due to a faulty driver as other SDRs have no issues with setting sample rates.
- While not being an issue related to the project itself, the university's firewall does not allow the TCP sockets to be made available. Local testing has revealed that the TCP sockets do indeed work. Opening the ports and configuring the server to use them should resolve the issue.

Chapter 10

Conclusion and future work

10.1 Conclusion

This thesis' goal was to create a WebSDR from the ground up, using a very high bandwidth SDR, with the unique feature of allowing users to handle signal processing themselves if they would so desire.

In order to bring this vision into reality, the first step was to make the XTRX work by choosing a platform and finding the correct installation procedure as documentation on the matter was very sparse.

The next step this thesis had was the development of an overall architecture as well as a network protocol. The architecture was specifically designed to allow easy switching of different subsystems for future upgrades and the network protocol was chosen to be as efficient as possible by reducing the amount of overhead data. The most appropriate libraries were chosen with performance in mind. Performance is also the reason why C++ was chosen as a language for the back-end server.

The implementation of both the back-end server and the user front end were the natural next steps. A lot of time was dedicated to this as this project was essentially just a concept at the beginning, no code had ever been written for it before.

The next step was the implementation of unit tests as well as using them for debugging the program. Once that was done, performance testing could start.

All throughout this project, heavy emphasis was put on creating exhaustive documentation in the form of readme files explaining how to setup everything as well as lots of code comments and this report which acts as more formal technical documentation. This was very important to allow future work on the project.

Said future work is bound to happen as the initial goals have not all been attained. Creating a working prototype has taken much more time than anticipated and has resulted in the project needing more work to work on very high bandwidths with a lot of clients as initially envisioned.

In conclusion, this project has brought a working first prototype of the WebSDR which should allow future work to be conducted easily. It has gone from a blank slate to a state where it can already start operating on a small set of test users. The documentation also allows future developers to quickly and easily pick up where the project was left off.

10.2 Future work

As this project already had a large scope, many features were left to be added in the future. The following list contains such features and fixes that would be nice improvements :

- Adding new demodulations. This is not a hard task as it would be as simple as handling additional cases in the switch statements.
- Improving performance by changing the circular buffers into simple queues as there is only ever 1 reader and 1 writer. Circular buffers had been chosen in the beginning because the initial architecture had multiple readers. This is not the case anymore.
- Improving performance by working on the bottlenecks identified in section 9.2. This includes potentially creating a signal processing library from scratch to cater to the needs of this project specifically.
- Adapting the project to be able to be used on distributed systems so it can scale better.
- Making the user interface more aesthetically pleasing.

Bibliography

- [1] Travis F. Collins, Robin Getz, Di Pu, and Alexander M. Wyglinski. *Software-Defined Radio for Engineers*. Artech House, 2018.
- [2] Wikipedia contributors. *Software-defined radio*, 2024. URL https://en.wikipedia.org/wiki/Software-defined_radio. Accessed: 09-07-2024.
- [3] cppreference.com contributors. *C++ reference*, 2024. URL <https://en.cppreference.com/w/>.
- [4] Advanced Micro Devices. *Vivado design suite tcl command reference guide*, 2024. URL <https://docs.amd.com/viewer/book-attachment/JXswcrCHJROaHTz43h5g2Q/jfellV4okrM3vvXUEXM6Cg>. Accessed: 12-11-2023.
- [5] Bartalan Eged and Benjamin Babjak. *Universal software defined radio development platform*, 2006. URL https://www.researchgate.net/publication/235080254_Universal_Software_Defined_Radio_Development_Platform. Accessed: 05-07-2024.
- [6] The Free Software Foundation. *Gnu radio manual and c++ api reference*, 2024. URL <https://www.gnuradio.org/doc/doxygen/>. Accessed: 05-07-2024.
- [7] Paul Heckbert. *Fourier transforms and the fast fourier transform (fft) algorithm*, 1998. URL <https://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/www/notes/fourier/fourier.pdf>. Accessed: 05-07-2024.
- [8] Steven G. Johnson Matteo Frigo. *The design and implementation of fftw3. Proceedings of the IEEE*, 93(2):216–231, 2005.
- [9] Steven G. Johnson Matteo Frigo. *fftw*, 2020. URL <https://www.fftw.org/fftw3.pdf>. Accessed: 05-07-2024.
- [10] Bruno A. Olshausen. *Aliasing*. 2000. URL <https://www.rctn.org/bruno/npb261/aliasing.pdf>. Accessed: 05-07-2024.

- [11] MIT OpenCourseWare. *6.003 signals and systems*, 2011. URL https://ocw.mit.edu/courses/6-003-signals-and-systems-fall-2011/12e6e5d7567fca2e993ef8563fef5a60_MIT6_003F11_lec21.pdf. Accessed: 05-07-2024.
- [12] The Boost organization. *Boost c++ libraries documentation*, 2024. URL <https://www.boost.org/doc/>. Accessed: 05-07-2024.
- [13] The Pothosware Project. *SoapySDR documentation*, 2021. URL <https://pothosware.github.io/SoapySDR/doxygen/latest/index.html>. Accessed: 05-07-2024.
- [14] Stephen Robert. *Signal processing & filter design*, 2003. URL https://www.robots.ox.ac.uk/~sjrob/Teaching/sp_course.html. Lecture 7 - The discrete Fourier transform.
- [15] Grant Maloy Smith. *What is signal processing?*, 2023. URL <https://dewesoft.com/blog/what-is-signal-processing>. Accessed: 13-10-2023.
- [16] IEEE Signal Processing Society. *What is signal processing?*, 2024. URL <https://signalprocessingsociety.org/our-story/signal-processing-101>. Accessed: 22-07-2024.
- [17] Bjarne Stroustrup. *The C++ Programming language*. Addison-Wesley, 4th edition, 2013. ISBN 0321563840.
- [18] Collegedunia Team. *Difference between am and fm*, 2024. URL <https://collegedunia.com/exams/difference-between-am-and-fm-physics-articleid-1008>. Accessed: 05-07-2024.
- [19] Inc. The MathWorks. *Fir halfband filter design*, 2024. URL <https://www.mathworks.com/help/dsp/ug/fir-halfband-filter-design.html>. Accessed: 03-02-2024.
- [20] Utah State University. *Mae 3340 instrumentation systems, section 2.3: Sampling, nyquist frequency, and signal aliasing*, 2015. URL http://mae-nas.eng.usu.edu/MAE_3340_Web/2015_Web_Page/section2/section2.3.pdf. Accessed: 05-07-2024.