Red Pitaya

Thesis

Raphael Frey
Noah Hüsser

August 16, 2017
Version 0.0.1

Contents

List of Figures

List of Tables

List of Listings

Introduction

Measuring equipments has ever been very expensive. If one wants precise devices it can go up to hundreds or thousands of Francs. Modern, cheap FPGAs can often, if combined with a proper frontend, replace those expensive dedicated devices. The frontend consists of dedicated ADCs and DACs and oftentimes some analog filters. Those chips also have become a lot cheaper in recent years aand are thus way more accessible. Extreme equipment, for example with high sampling rates, in contrary can still not be replaced easily. This project aims at arming an FPGA board with logic that can record, filter and store electrical signals with adjustable sampling rates up to 125 MHz. To complement the hardware part a software that runs on an embedded Linux on the integrated ARM core will be coded such that it can transmit the recorded samples over the network. To read and visualize the samples at the other end of the network, another piece of software will be crafted, that will run on the users computer. The primary focus lies on enabling students to analyze audio signals. Since audio signals contain very low frequencyies only up to tens of thousands of Hz they can be sampled with rather low frequencies and thus making FPGAs an excellent choice. An FPGA not only shines in price competitiveness but also in flexibility. This means that the logic is not fixed in silicon and can be adjusted after the product has already been delivered.

For this project a RedPitaya board is used. It is ideal since it features a fast (125 MHz) 14-bit ADC. This poses a huge amount of data, that is not even required for audio signals. Furthermore it is not realisticly possible to transmit this huge amount of data over the network.

Thus the first primary target of this thesis is to decimate the recorded signals. To avoid aliasing effects that emmerge when decimating a signal appropriate filters have to be designed and impemented that are able to attenuate unwanted signal frequencies.

The second primary target of this thesis is the design and implementation of a software-based oscilloscope. This is a graphical user interface that communicates with the RedPitaya board and visualizes the recored samples. Traditional measuring equipment always has a built in display that visualizes the data on the device itself. This uses up a lot of space and provides very low flexibility. Since it can be assumed that every engineur is equipped with a computer, said device should be used to display the signals. This keeps cost and required space down. The data is then transmitted via the network which will be interfaced by both the users computer and the RedPitaya board.°

[2]

Part I

Project Report

CHAPTER 1

Theoretical Background

This chapter will present a brief synopsis on some aspects of digital data acquisition from an analog source and the processing of that data, and how those issues pertain to our project. It is not intended to be a comprehensive treatise on the subject but shall serve as a short refresher. At its end, the reader should have sufficient insight to understand the basic motivation of our project from a theoretical point of view.

TODO: references to more comprehensive literature.

1.1 The Digital Signal Processing Chain

Digitally acquiring a signal generally requires at least the following steps:

- Passing the signal through an analog low-pass filter.
- Sampling and quantizing the filtered signal.

The resulting sequence of values can then be further digitally processed. The necessary building blocks for this process are portrayed in Figure 1.1.

Of particular interest for our application is what happens in the ADC. The quantization process converts a value-continuous signal into a value-discreet one, with its resolution being a specification of the ADC which is being used. As an example, the ADC in our system has a resolution of 14 bits, meaning it can divite its valid input range into 16 384 values. Given an input range of 2 V V_{pp}, that equates to a resolution of roughly 122 µV (in theory). This quantization process is the source of what is generally known as *quantization noise*. For more on the topic, see `TODO`.

TODO: Check numbers. Give references for further reading.

Besides the quantization, the other step happening in the ADC is sampling; a time-continuous signal is converted into a series of time-discreet values. The time between those

Figure 1.1: The basic building blocks of the DSP chain from its analog input to its digitally processed output. From left to right: The analog low-pass filter (*LP*), the analog-to-digital converter (*ADC*), and an arbitrary digital signal processing system for further processing of the ADC's output (*DSP*).

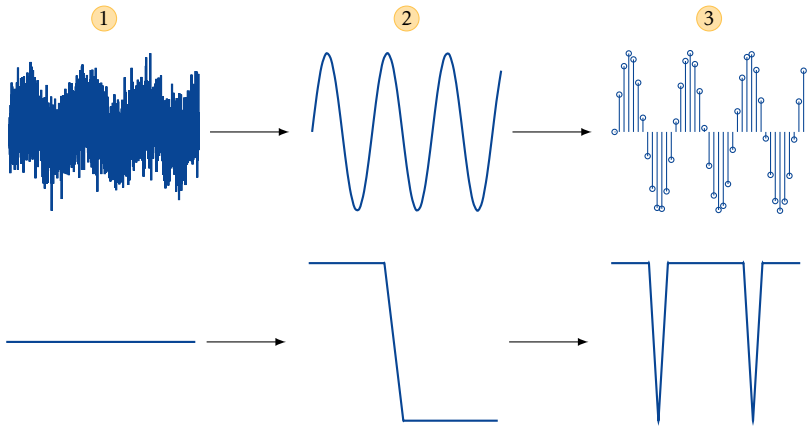


Figure 1.2: Simplified time-domain (top) and frequency-domain (bottom) view of the signal at different stages on its way through the DSP chain. The circled numbers correspond to the stages as outlined in Figure 1.1. Stage 1 is the signal before passing through the input low-pass filter, with a significant amount of high-frequency noise. The low-pass filter removes any frequency components above $f_s/2$ in an ideal scenario (in reality, it merely attenuates them, as we will see later), resulting in the signal at stage 2.
After having been filtered, the ADC samples and quantizes the signal, yielding a sequence of values, as schematically portrayed in the rightmost picture for stage 3. Note that due to the sampling process, the spectrum of the filtered signal is repeated at intervals of f_s. This is the source of the issue of *aliasing*.

values is known as *sampling time*, its inverse is the *sampling frequency*. Note that usually these are constant, at least during the time where the signal is measured. This need not strictly be the case in theory though. In our system, this sampling frequency is a fixed property of the ADC, and is 125 MHz.

The sampling step lies at the core of the problem our project intends to address: *aliasing*. Therefore, we will take a closer look at a few consequences of the sampling process, and how they are relevant to this project.

Descriptively, the sampling process can be thought of as looking at a signal at specific points in time and capturing its value. Mathematically, this amounts to multipliying the signal with a series of Dirac pulses in the time domain, and convolving with a series of Dirac pulses in the frequency domain[1]. . This convolution in the frequency domain lies at the heart of the problem of aliasing, because it results in the incoming signal's spectrum being repeated at intervals of f_s (see also: stage 3 in Figure 1.2). This is no problem as long as the spectrum of the incoming signal fits within the boundaries set by this repetition. But if the spectrum of the incoming singal is too broad, two or more recurrences of the spectrum will overlap. This effect is highlighted in Figure 1.3.

This overlap results in two primary problems:

- The digital signal may not be unambiguously reconstructable into an analog signal, if that is intended.
- Frequencies may occur in the digital signal stream which are not actually present in the original signal. This problem is often referred to as the *folding back* of frequency components. See Figure 1.4 for an illustration of how this might look.

[1] *Pro memoria*: A series of Dirac pulses in the time domain has as its spectrum a series of Dirac pulses as well.

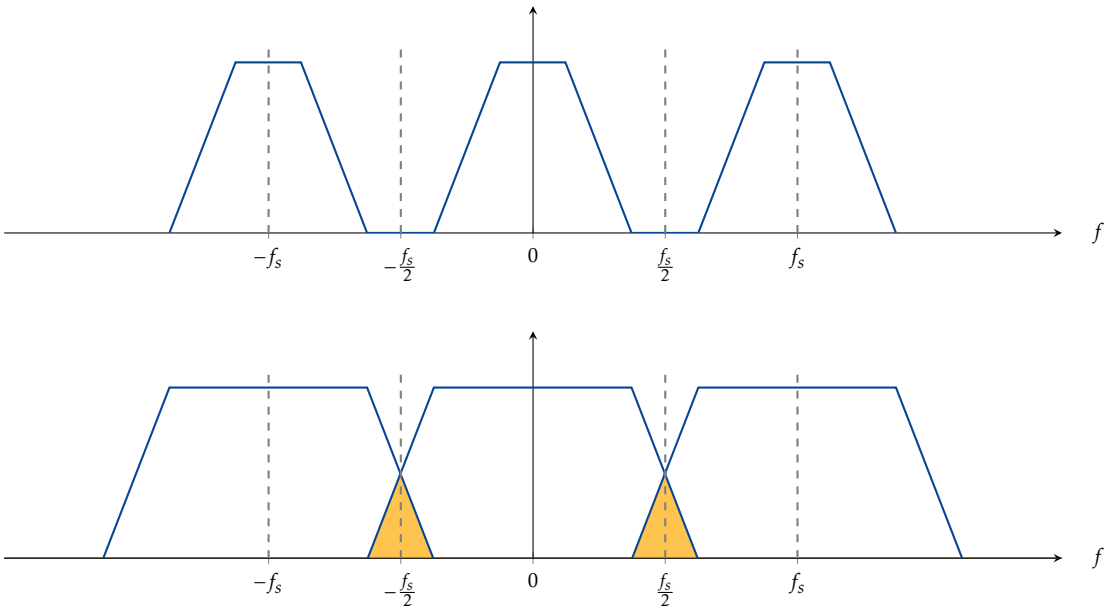


Figure 1.3: Simplified view of a signal which does not produce aliasing between its recurrences in the frequency spectrum (top), contrasted with a signal whose frequency band has components above half the sampling frequency, resulting in aliasing; its spectral copies overlap (highlighted areas in the bottom plot).

This problem is of particular interest to our application, as we will see later.

Once a signal has left the ADC and is handed down the DSP chain for further processing, the primary problem becomes one of resources, particularly in real-time applications. In most systems, the available hardware is a fixed constraint, and depending on what sort of processing is to be conducted on the digital data stream, the available resources may or may not suffice.

If available resources are found to be insufficient for real-time processing of the data stream, one may choose to

- not process the data in real time,
- reduce the complexity of the computations, or
- reduce the amount of data to be processed through *downsampling* of the signal.

The last case is the route which is chosen in our application. The main constraint on the Red Pitaya is that the data being generated cannot be moved off the device in real time, and the device itself does not offer sufficient storage for capturing a meaningful amount of data which can then be moved onto another device for further processing at a later point. Therefore the amount of data must be reduced before it can be moved off the device to a computer for viewing or further processing.

TODO: Amount of data being generated on the PITA.

Because downsampling a signal is in essence nothing more than the sampling of a signal which has already been sampled, a lot of the considerations which are valid for the step from an analog to a digital signal as outlined above are either very similar or even identical. Specifically, the same considerations for aliasing still apply: If the signal which

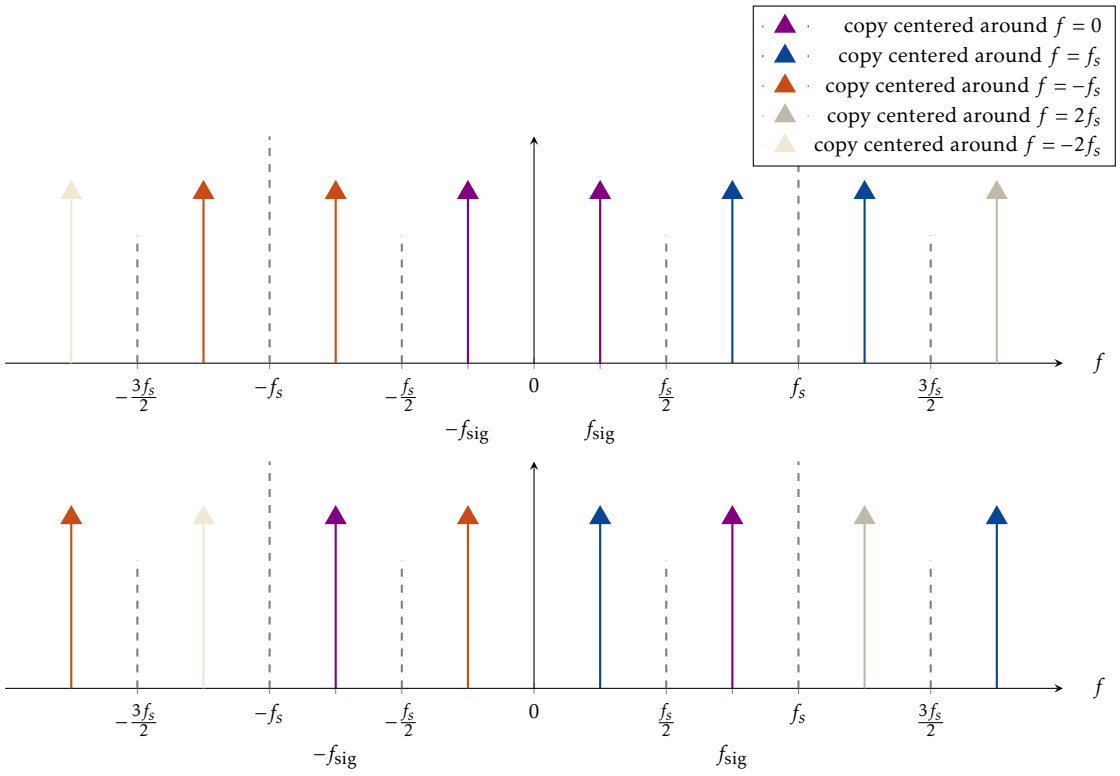


Figure 1.4: Example of two harmonic signals being sampled. In the top plot, the signal's frequency is below half the sampling frequency and there is no aliasing. The signal can be reconstructed without error. In the bottom plot, the signal's frequency is above half the sampling frequency. Consequently, the copies of the signal's frequency spectrum centered around the sampling frequency and its negative alias back into the band between $-f_s/2$ and $f_s/2$. If this signal is reconstructed, the resulting signal would have a frequency of $f_s - f_{\text{sig}}$ instead of f_{sig}.

is to be downsampled has frequency components above $f_{s,downsampled}/2$, aliasing will occur. And since the signal coming out of the ADC has the original signal's (filtered) spectrum recurring at intervals of its sampling frequency, this is always the case.`TODO: Correct?`

Therefore, the sampled signal must be filtered through a low-pass filter before being downsampled, just as the original analog signal was low-pass filtered before being passed into the ADC. In light of the signal to be downsampled being a *digital* signal instead of an analog one, that low-pass filter must naturally be a digital filter as well. Designing such a digital low-pass filter is the core mission of this project.

The key properites of such a filter which are relevant to our application are

- its transition band width (filter steepness), and
- its aliasing attenuation.

The aliasing attenuation refers to the fact that when a filter is being used for downsampling, copies of its frequency response will be created at intervals of the lower sampling rate (analogous to the sampling process producing spectral copies of a signal when sampling an analog signal).

The stopband components of these copies overlap with the intended passband, leading to aliasing (it should be noted that this phenomenon is also present in the case of the analog intput filter for the DSP chain). This effect is portrayed in Figure 1.5. The top plot shows the filter's frequency response along with four copies to illustrate the overlap effect. The bottom plot shows the aliasing effect more clearly by removing the spectral copies and retaining the aliased regions.

The overlapping parts of the spetrum are composed of spectral copies both to the right and left side of the original. Therefore, the aliased regions are alternately flipped around the vertical axis. This creates in essence the same effect as if the paper were folded along multiples of the lower sampling rate over the frequency range of the central copy (in the case of our example: $0.2f_s$, $0.4f_s$, $0.6f_s$ and $0.8f_s$) like an accordion. This is where the term *folding back* originates.

1.2 Digital Filters

Digital filters can be distinguished by several characteristics; common ways to categorize them are by topology, impulse response and their frequency response. There are two commonly used types of digital filters: Infinite impulse response (IIR) filters and finite impulse response (FIR) filters. Another important class of filters are cascaded integrator-comb (CIC) filters, however, in the strictest sense they are a special class of FIR filters rather than an entirely new type of LTI system [2]. While our system uses FIR and CIC filters, a brief overview of IIR filters is still presented here, for the sake of completeness.

1.2.1 IIR Filters

Infinite impulse response filters are so named because their imulse response continues into perpetuity, never reaching zero. In practice, the response usually comes sufficiently close to zero at a certain point that it can be considered zero for most intents and purposes.

IIR filters have feedback paths, resulting in a filter response equation with non-trivial denominator components. Their basic building blocks are delay elements, multipliers and adders.

$$H(z) = \frac{\sum_{k=0}^{N} b_k \cdot z^{-k}}{1 + \sum_{i=0}^{M} a_i \cdot z^{-i}} \tag{1.1}$$

TODO: Move sum limits above and below sigma sign.

IIR filters generally require a lower order (and therefore fewer resources) to approximate a certain frequency response specification than FIR filters do (particularly the constraint of a narrow transition band), but this comes at a cost: IIR filters have a non-linear phase response; linear-phase responses can only be approximated. Furthermore, IIR filters are not guaranteed to be BIBO stable due to their feedback paths.

Some of the generally used types of IIR filters are:

- Butterworth filter: Named after the British engineer and physicist Stephen Butterworth (1885 – 1958), who first described it in 1930. Characterized by a very flat passband (no passband ripple).
- Chebyshev filter (type I and II): Named after Russian mathematician Pafnuty Chebyshev (1821 – 1894). They are steeper than Butterworth filters, at the cost of suffering from ripple in the passband (type I) or stopband (type II).

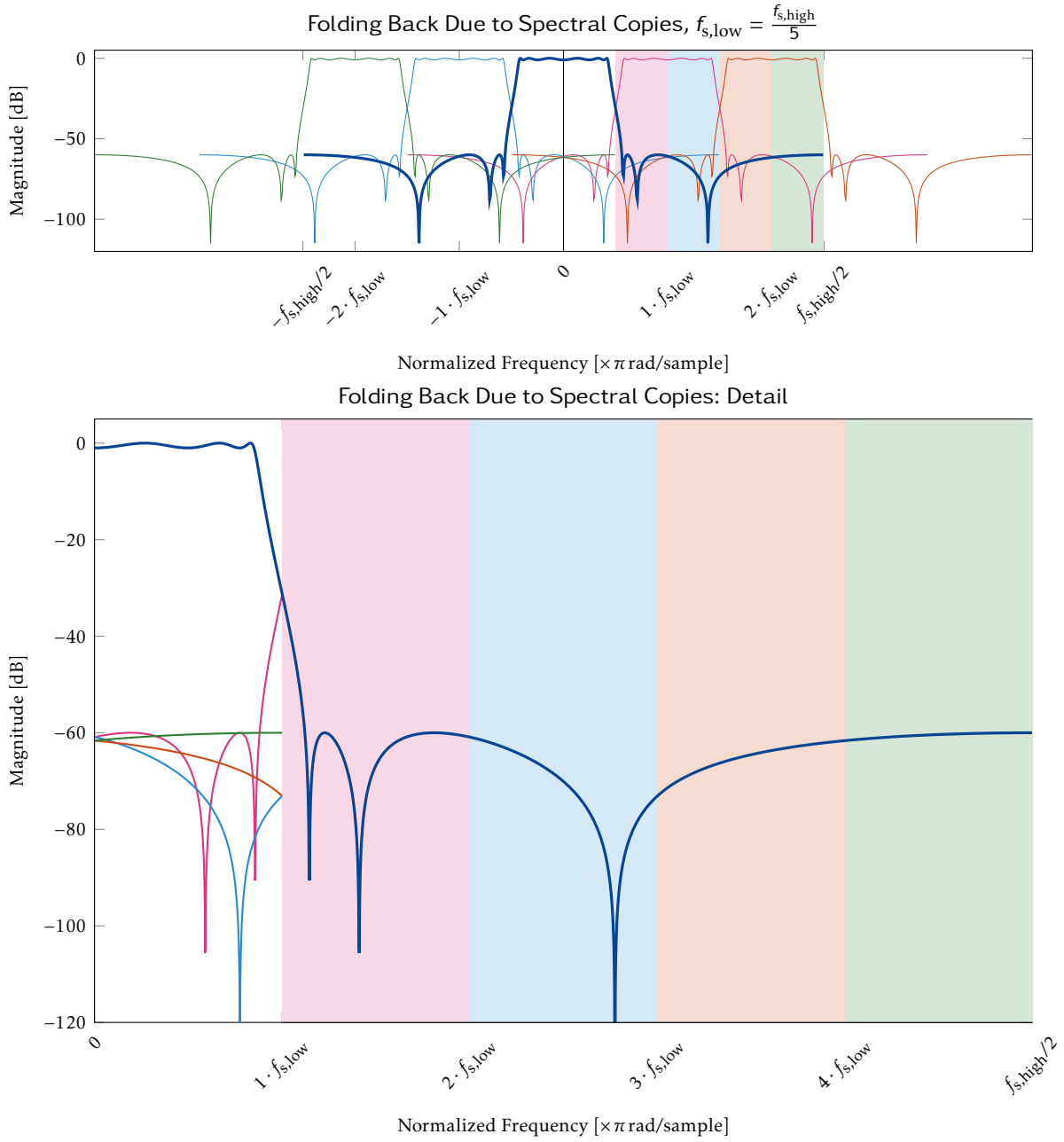


Figure 1.5: The phenomenon of folding back when downsampling, illustrated for a lowpass IIR filter with with a cutoff frequency of $0.2 \cdot f_s$ for a downsampling ratio of $R = 5$. The downsampling process produces copies of the filter's frequency response at intervals of the lower sampling frequency, visible in the top plot. The stopbands of these copies then overlap with the intended passband. The bottom plot shows a close-up view with the spectral copies for clarity.

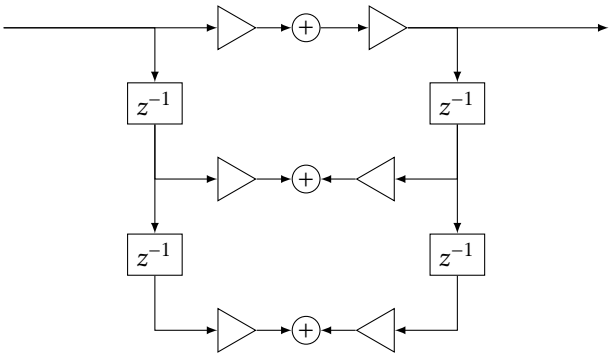


Figure 1.6: Example of an IIR filter topology for a biquad

- Bessel filter: Named for the German mathematician Friedrich Bessel (1784 – 1846). Optimized to have a maximally linear phase response in order to minimize the distortion of signals passing through the filter.
- Elliptical filters: Also known as Cauer filters, after the German mathematician Wilhelm Cauer (1900 – 1945), or Zolotarev filter, after Russian mathematician Yegor Zolotarev (1847 – 1878). Characterized by equiripple in the bassband and stopband and a very narrow transition band compared to other filters of the same order.

TODO: Check correct dashes for years.
Digital IIR filters are often designed by way of the bilinear transform.

1.2.2 FIR Filters

FIR filters are characterized by an impulse response which decays to zero in finite time (see Figure 1.9, unlike IIR filters. The filter response is characterized by Equation 1.2:

$$H(z) = \sum_{k=0}^{N} b_k \cdot z^{-k} \tag{1.2}$$

FIR filters have several advantages:

- They are inherently BIBO stable because they lack feedback paths.
- They can be easily designed to have a linear phase response, preventing signal distortion due to different group delays for signal components of different frequencies.
- The shape of their frequency response can be very finely tuned. This makes them ideally suited for certain purposes, such as compensation filters (see Section 1.2.3).
- Implemenation is usually rather straightforward.

TODO: Correct?

Their main disadvantage is that due to the lack of feedback, they generally require comparatively high filter orders for narrow transition band widths. Illustratively, this can be understood by the following considerations:

- The frequency response of an ideal low-pass filter is the brick wall filter, i.e. a rectangle.
- The inverse Fourier transform of a rectangle is a *sinc* function, which is infinitely long.

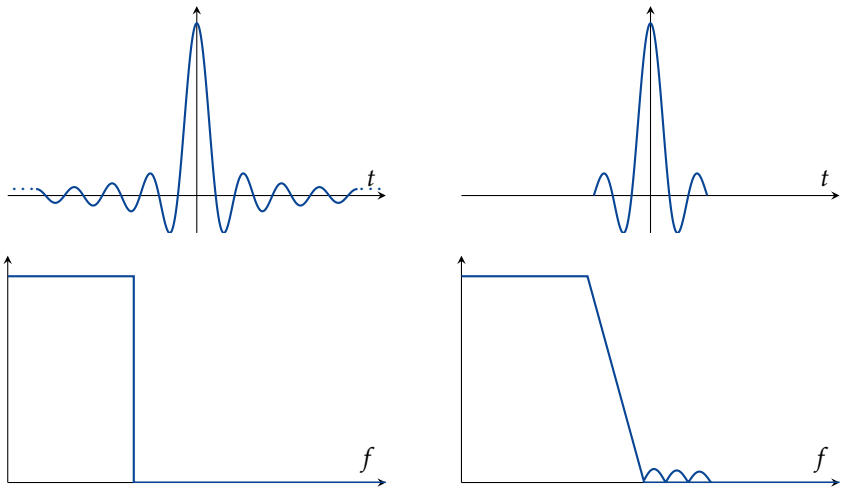


Figure 1.7: The effect of truncating a *sinc* function in the time domain on its spectrum (simplified)

- Therefore, the impulse response of the ideal brick wall filter would have an infinite number of taps.
- Truncation of the number of taps leads to a deviation of the filter's frequency response from the brick wall filter. As the number of taps (and therefore the FIR filter's impulse response) is reduced, its frequency response deviates more and more from the brick wall filter, resulting in a flatter transition between the passband and the stopband as well as the introduction of ripple.

This process is illustrated in simplified form in Figure 1.7.

The FIR filter's transition band width is particularly important for our application in order to reduce aliasing effects, as will be shown later TODO: actually show later.

TODO: Equation for order estimation.

Designing FIR filters is usually performed by specifying certain desired characteristics of the filter's frequency response. Figure 1.8 shows one possible way of doing this for FIR filters by specifying four constraint parameters:

- pass band ripple: A_P
- stop band attenuation: A_{St}
- pass band edge frequency: F_P
- stop band edge frequency: F_{St}

The resulting transition band width F_{Tb} is the difference between the pass band edge frequency and the stop band edge frequency, and serves as a useful indicator of how many coefficients (i.e. resources) the filter will end up using. Narrower transition bands tend to require a higher filter order, and therefore more resources. Coefficient counts of several hundred are not uncommon for steep FIR filters.

Other sets of constraint parameters can be used to design filters, but these are the ones used in this project, therefore the emphasis on them.

Figure 1.9 shows the resulting impulse response (coefficient set) for a FIR filter designed by using the four above mentioned parameters, with values given by Equations 1.3 through 1.6 handed to one of Matlab's FIR filter design algorithms.

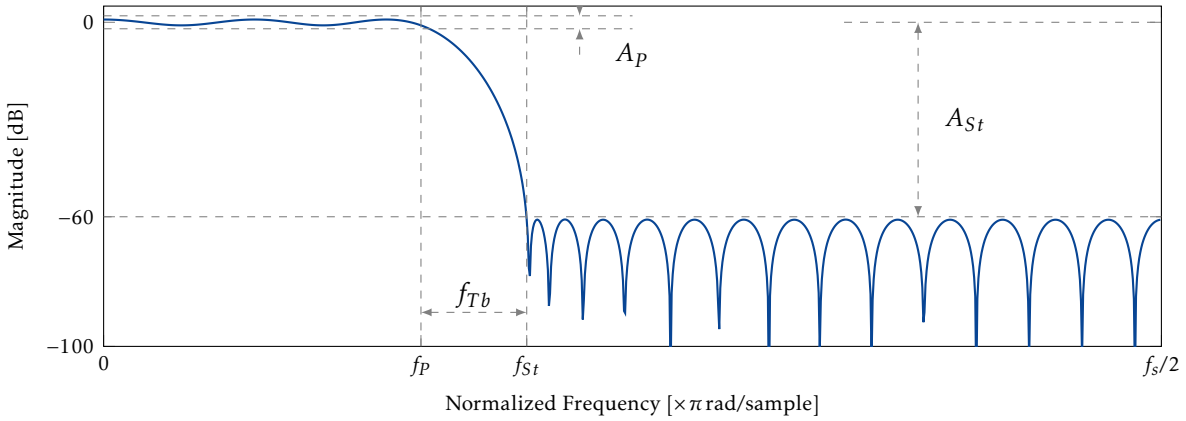


Figure 1.8: Specifications in the frequency domain and the resulting filter's frequency response as designed by Matlab.

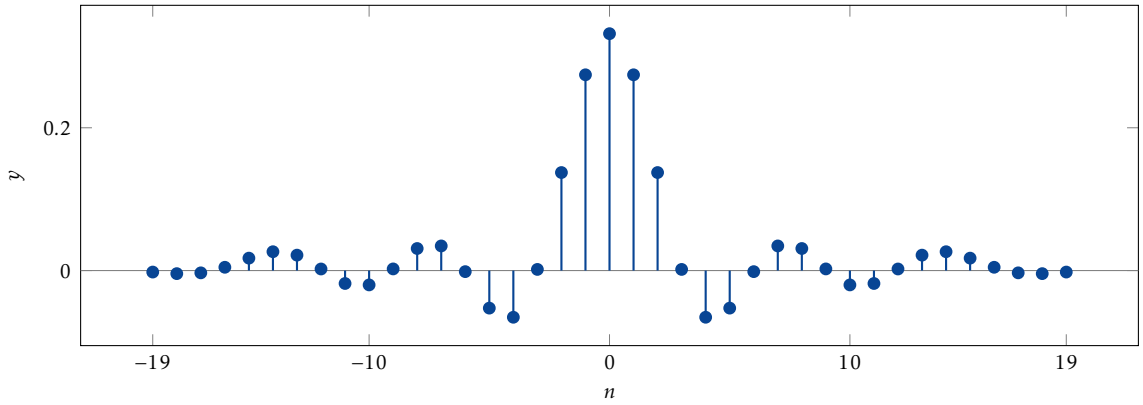


Figure 1.9: Impulse response (coefficients) for the filter from Figure 1.8 with the parameters as given by Equations 1.3 through 1.6 passed to one of Matlab's FIR filter design algorithms, resulting in a set of 39 coefficients. Note that the coefficients to the left and right of these values are zero, hence *finite* impulse response filters.

$$A_P = 2\,\text{dB} \tag{1.3}$$
$$A_{St} = 60\,\text{dB} \tag{1.4}$$
$$F_P = 0.3 \cdot f_s \tag{1.5}$$
$$F_{St} = 0.4 \cdot f_s \tag{1.6}$$

Figure 1.10 shows one possible topology for implementing a FIR filter, the so-called direct form. As can be seen, the basic building blocks of a FIR filter are delay elements, multipliers and adders, same as for IIR filters.

One particular form of a FIR filter is the so-called half-band filter. Half-band filters are used for downsampling by a ratio of $R = 2$. They are characterized by a point-symmetric

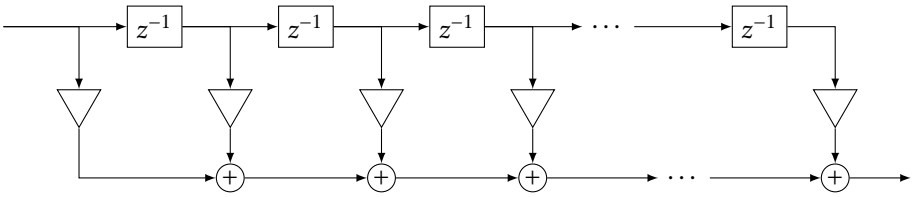


Figure 1.10: One possible topology for a FIR filter (direct form)

frequency response across the $(f_s/4, 0.5)$ point. Their advantage lies in the efficiency of their coefficient structure: Each second coefficient is zero, and all the non-zero coefficient are symmetrical around the center of the impulse response. For higher downsampling rates, multiple half-band filters can be cascaded. Figure 1.11 shows the amplitude frequency response and the coefficient set of an example filter.

1.2.3 CIC Filters

CIC filters were first introduced in 1981 in [2] by Eugene B. Hogenauer. They can be implemented both as decimation filters (reduction in sampling rate) and interpolation filters (increase in sampling rate).

1.2.3.1 General Description

A CIC filter is a cascade of integrator and comb stages, with either a sampling rate compressor (in case of a decimator) or a sampling rate expander (in case of an interpolator) between the integrator and comb sections. A single integrator stage is shown in Figure 1.12, and Figure 1.13 shows a single comb stage in feedforward form. Figure 1.14 shows a complete CIC filter with three stages.

The integrator stages have a transfer function of

$$H_I(z) = \frac{1}{1 - z^{-1}} \tag{1.7}$$

The comb stages run at the reduced frequency of f_s/R and have the transfer function

$$H_C(z) = 1 - z^{-RM} \tag{1.8}$$

where M is the *differential delay*, one of the filter's design parameters.

The transfer function of a complete CIC filter (referenced to the high sampling rate f_s) consisting of N stages is deduced by multiplying the transfer functions of the N cascaded integrator and comb stages, yielding

$$H_{CIC}(z) = H_I^N(z) \cdot H_C^N(z) = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N \tag{1.9}$$

Looking at the last form of the CIC filter's transfer function, it becomes evident that it is in essence a FIR filter with unitary coefficients. Of particular note is the fact that this is so despite each stage having feedback or feedforward paths and the integrator stages having poles at $f = 0$ (i.e. the integrators by themselves are not in fact BIBO stable, even though the complete system is). The fact that the resulting filter has no poles can be intuitively understood by looking at the frequency responses of the integrator and comb stages, and finally their cascade (see Section 1.2.3.2).

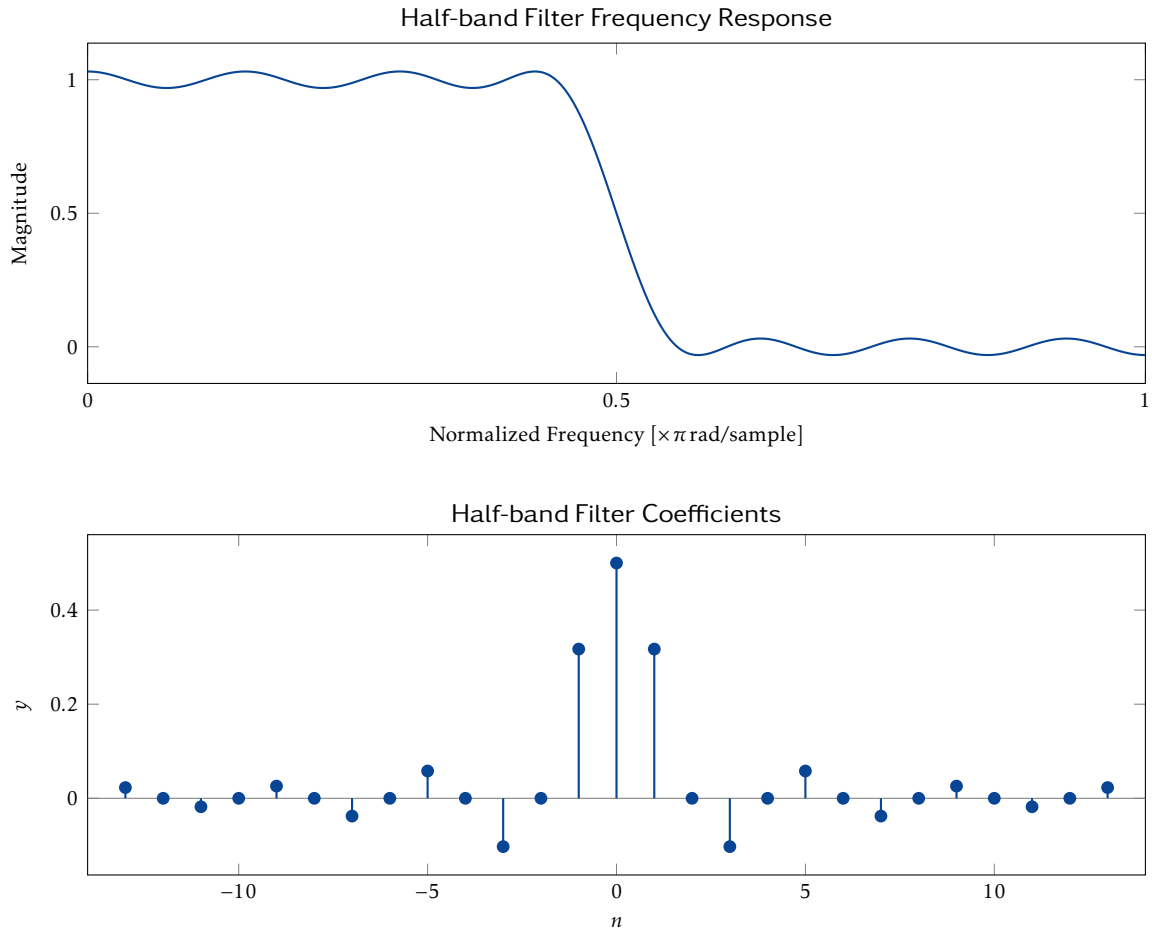


Figure 1.11: The frequency response and coefficient set of a half-band filter. The frequency response is the amplitude plotted linearly, not the magnitude plotted logarithmically, in order to emphasize the symmetry. The coefficient set is symmetrical around its midpoint. Also, every second coefficient outside the central peak is zero.

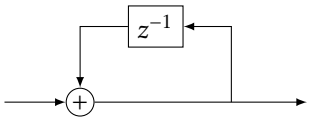


Figure 1.12: A single integrator stage

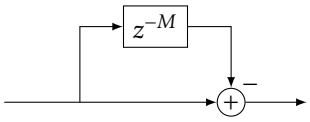


Figure 1.13: A single comb stage in feedforward form

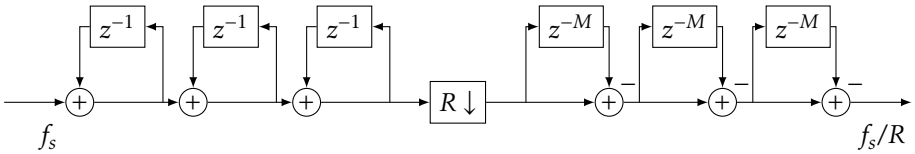


Figure 1.14: CIC decimation filter topology with three integrator and comb stages

CIC filters are well-suited to large reductions in sampling rates because they are very economical in their resource usage. This economy is based on six primary factors (see also [2]):

- The filter requires no multipliers.
- There are no filter coefficients to store.
- The amount of storage needed for intermediate results is reduced by running the comb stages at a lower sampling rate. A conventional FIR filter topology implementing the same transfer function would require more resources for storing its intermediate results because the entire filter would run at the incoming sampling rate.
- The topology of the filter has a high degree of regularity; consisting of two primary building blocks. This lends itself well to optimization.
- The control logic can be kept simple.
- The same filter design can be used for a large range of rate change factors R, requiring minimal adaption in circuitry. This effect can be seen in the frequency response plotted in the top plot of Figure 1.16.

However, CIC filters do suffer from some drawbacks. The two primary ones are:

- For large rate change factors R, the register growth of the filter can become very large. TODO: see section blabla
- A CIC filter has only three design parameters determining its frequency response: Rate change factor R, differential delay M, and the number of stages N. The amount of fine-tuning which can be conducted on the filter's frequency response is therefore extremely limited (more in Section 1.2.3.2).

As can be seen in Equation 1.7, the integrator stages have unity feedback coefficients. In the case of CIC decimators, the registers of the integrators will therefore suffer from register overflow. This causes no harm as long as two conditions are fulfilled:

- The filter's implementation is based on two's complement or another number system allowing wrap-around between its most positive and most negative numbers.
- The maximum magnitude which is expected at the output is within the range of that number system.

A numerical example to demonstrate this effect and better explain the inner workings of a CIC filter can be found in Appendix A.1, starting on page 88.

1.2.3.2 Frequency Characteristics

This section presents some of the more important frequency characteristics of the CIC filter. We will start with some considerations about how the integrators and comb sections interact in the frequency domain to create the CIC filter's frequency response.

As shown in the topmost plot in Figure 1.15, an integrator is in essence a lowpass filter, with a pole at $f = 0$. A comb filter is a filter which attenuates one specific frequency component along with its multiples (in a notch comb filter; there is also the inverse concept of a peak filter which only lets a certain frequency and multiples of it pass). It is also evident that comb filters have no poles (a fact which can be deduced from Equation 1.8 as well, of course).

Cascading integrators and combs results in a frequency response like the one in the bottom plot from Figure 1.15. The integrator's pole at $f = 0$ compensates for the comb section's zero at the same location, leading to a significant, but finite, DC gain of the CIC filter.

One drawback of CIC filtes is that they have no clearly defined passband as such. Rather, their frequency response starts dropping off right as the frequency axis goes beyond zero. This effect (also referred to as *passband droop* or *passband attenuation*) is visible in the magnified section of the bottom plot in Figure 1.15 and in Figure 1.17. Since CIC filters lack a clearly defined transition band edge, defining the frequency band which is to actually be used, i.e. the actual passband, is a design decision and can vary even when using the same filter, depending on the application.

The amount of passband droop is constant for a given product of the differential delay M and the cutoff frequency f_c, where f_c is a fraction of the lower sampling rate (i.e. a fraction of the first lobe's width). Figure 1.17 highlights this effect for two different filters. Table A.3 in Appendix A.2 on page 93 contains a list with more values for some common configurations.

Because of the passband droop, a CIC filer by itself is rarely a viable solution. Rather, it is generally deployed as the first element in a chain of filters, where the later stages are FIR filters. Due to the CIC filter's frugality in terms of resource usage, it is ideally suited as an initial stage, where the most samples per time need to be processed. The fact that FIR filters need to perform many more computations (and more complex ones) per sample is then no longer as much of a problem, since those FIR filters run at lower sampling frequencies and have therefore many more clock cycles available to compute each output. Also, because the frequency response of a FIR filter can be very finely tuned to a desired profile, they can be used to compensate for the CIC filter's passband droop; this is generally known as a *CIC compensation filter*. TODO: altera application note

Another effect which must be taken into consideration when designing CIC filters is the amount of aliasing which occurs from the stopband into the passband. A region of width f_c above and below each Mth null is folded back into the filter's passband. This effect is highlighted in Figure 1.18. The gravity of this effect depends on the width of the cutoff frequency f_c as well as the differential delay M. Table TODO in Appendix TODO contains some values for common ranges for M and f_c.

As mentioned, the CIC filter has only three design paramterers: Its rate change factor R, the differential delay M and the number of stages N. The influence of these paramters on the CIC filter's frequency response is portrayed in Figure 1.16. Some things of note are:

- Increasing R increases the amount of nulls as well as the overall gain of the filter.
- Increasing M also increases the number of nulls as well as the filter's gain. Note that for CIC decimators, the region around every Mth null is folded back into the passband.
  For practical purposes, M is usually set to 1 or 2, see [2].
- Adding more stages leads to a high increase in filter gain, since N occurs in the exponent of the filter's transfer function. It does not, however, change the number or placement of the nulls.

1.2.3.3 Compensators

In order to achieve a flat passband, the CIC filter's attenuation in the frequency range observed in Figure 1.17 can be compensated with a filter whose frequency response has the opposite shape. Operated in a cascade (see Section 1.3), the two filters create a frequency response with a flat passband and a sharop drop-off into the stopband. Figure 1.19 shows an example of such a system, with frequency responses of a CIC filter, its compensator, and the resulting cascade.

The compensation filter not only serves to compensate for the passband, but is also responsible for the transition band width of the cascade. Due to their flexibility, FIR fil-

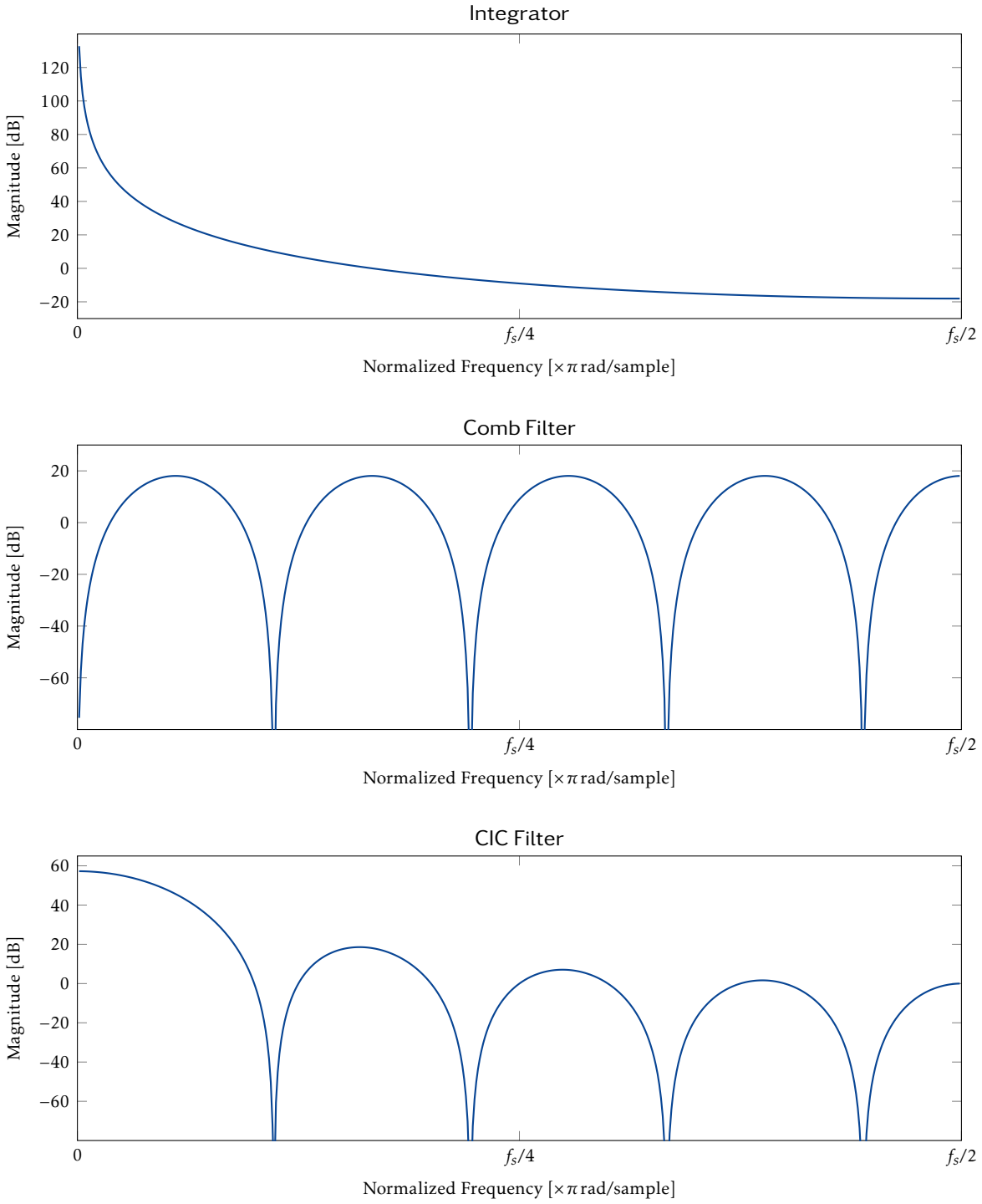


Figure 1.15: Frequency responses for integrators, combs and their combination into a three-stage CIC filter with a rate change factor of 9 and a differential delay of 1. Note that 4.5 lobes fit into the plot for the comb filter, due to $R \cdot M = 9$ (the order of the comb filter). The enlarged box shows a close-up of the CIC filter's passband droop.

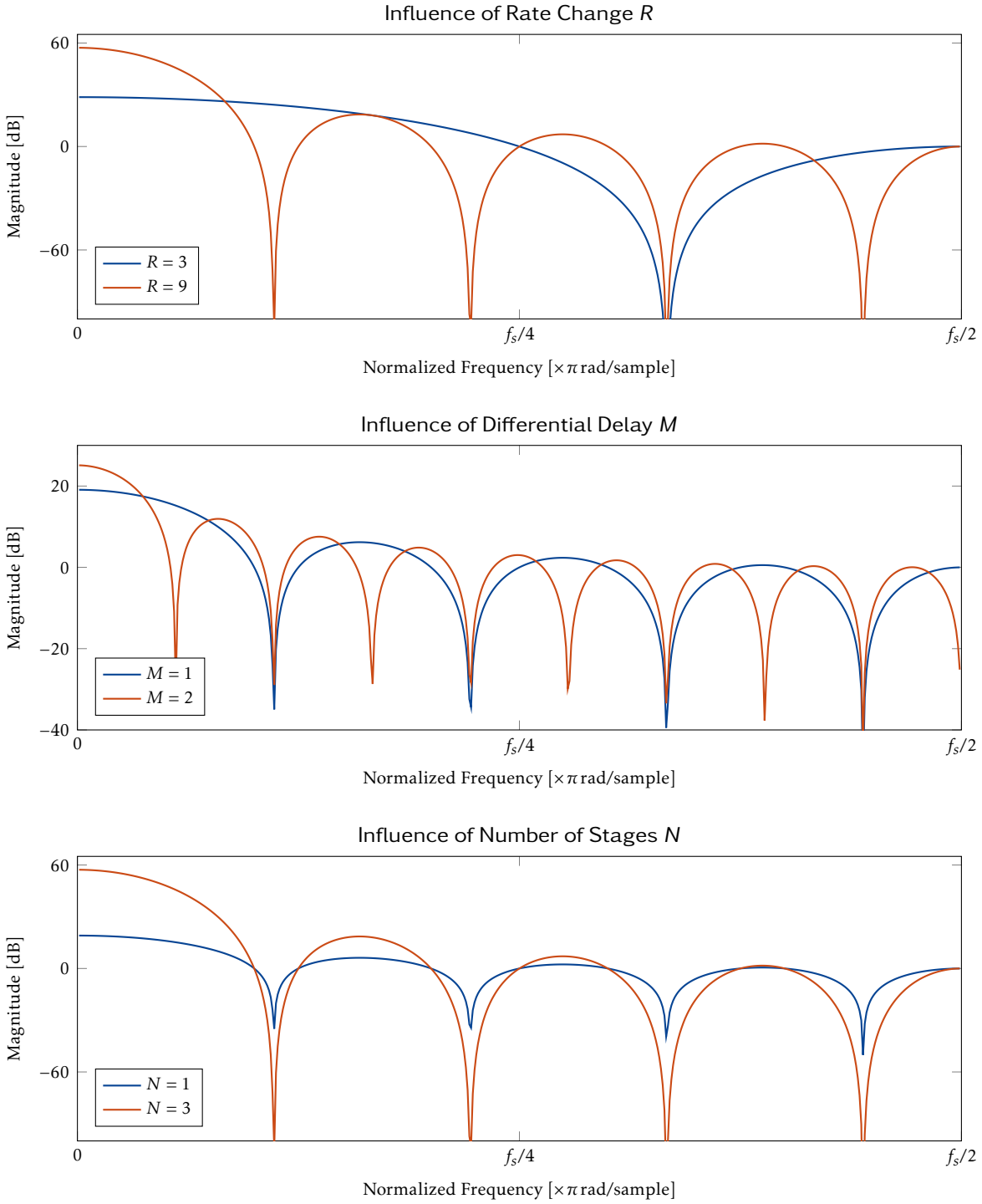


Figure 1.16: The influence of the design paramters R, M and N on a CIC filter' frequency response. Inreaseing R and M, respectively, leads to an increased number of nulls, as visible in the top two plots, as well as an increase in the DC gain. Adding more stages does not change the location of the nulls, but does add significant DC gain.

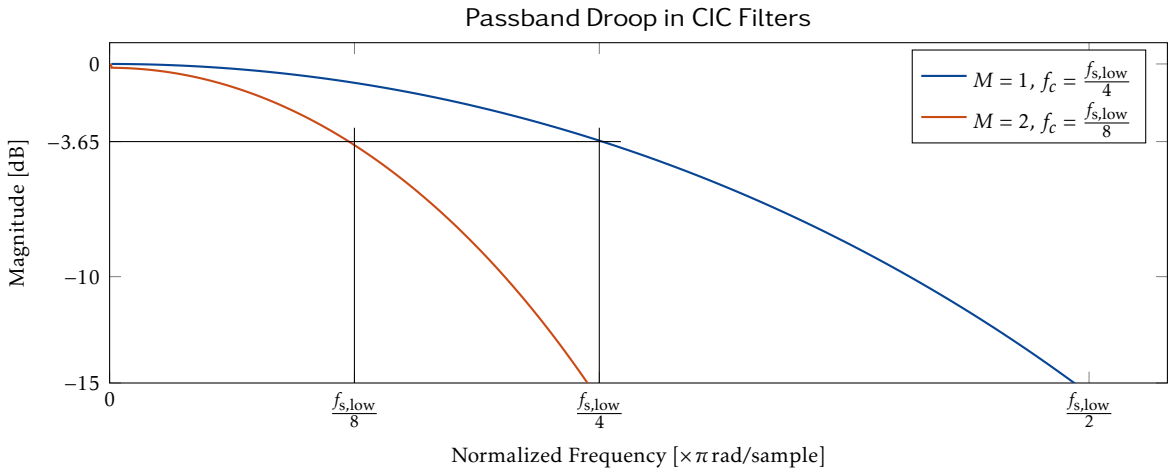


Figure 1.17: Passband attenation for two CIC filters with $R = 9$, $N = 4$ and $M = 1$ and $M = 2$, respectively. The attenuation is identical for the bandwidth-differential delay product, which is 1/8 for both of these configurations. The attenuation is −3.65 dB in both cases; the value can be found in Table A.3 on page 93.

ters are generally employed for this purpose. As mentioned in the previous section, the sharpness of their transition band can be controlled by adjusting their kernel size. If the compensator has more filters coming after it in the overall filter chain, its transition band need not be very narrow. A filter with a few dozen coefficients in size is often sufficient in such cases (the filter used for the example in Figure 1.19 has 50 coefficients).

The design of CIC compensators is usually left up to software algorithms, for example with Matlab. For a more detailled introduction to the topic, including example code and more thorough explanations, Altera's Application Note from [3] is warmly recommended.

1.2.3.4 Register Growth

As shown by Hogenauer in [2], the maximum register growth is

$$G_{\max} = (R \cdot M)^N \tag{1.10}$$

The most significant bit $B_{\max}$ of the output register as well as for all stages (both the integrators and the comb stages) of the filter is determined to be

$$B_{\max} = \lceil N \log_2 RM + B_{\text{in}} - 1 \rceil \tag{1.11}$$

where B_{in} is the bit width of the input register. For high rate change factors, these values can become very large. A filter with three stages, a differential delay of 1, a rate change of 128 and an input width of 16 bits yields 36 bits output width at full precision.

1.2.3.5 Errors Due to Truncation and Rounding

In practical cases, it is often not feasible to retain full precision; in such situations, either truncation or rounding may be used at each filter stage to reduce register widths and keep resource usage within certain limits. For this purpose, it is necessary to know the system

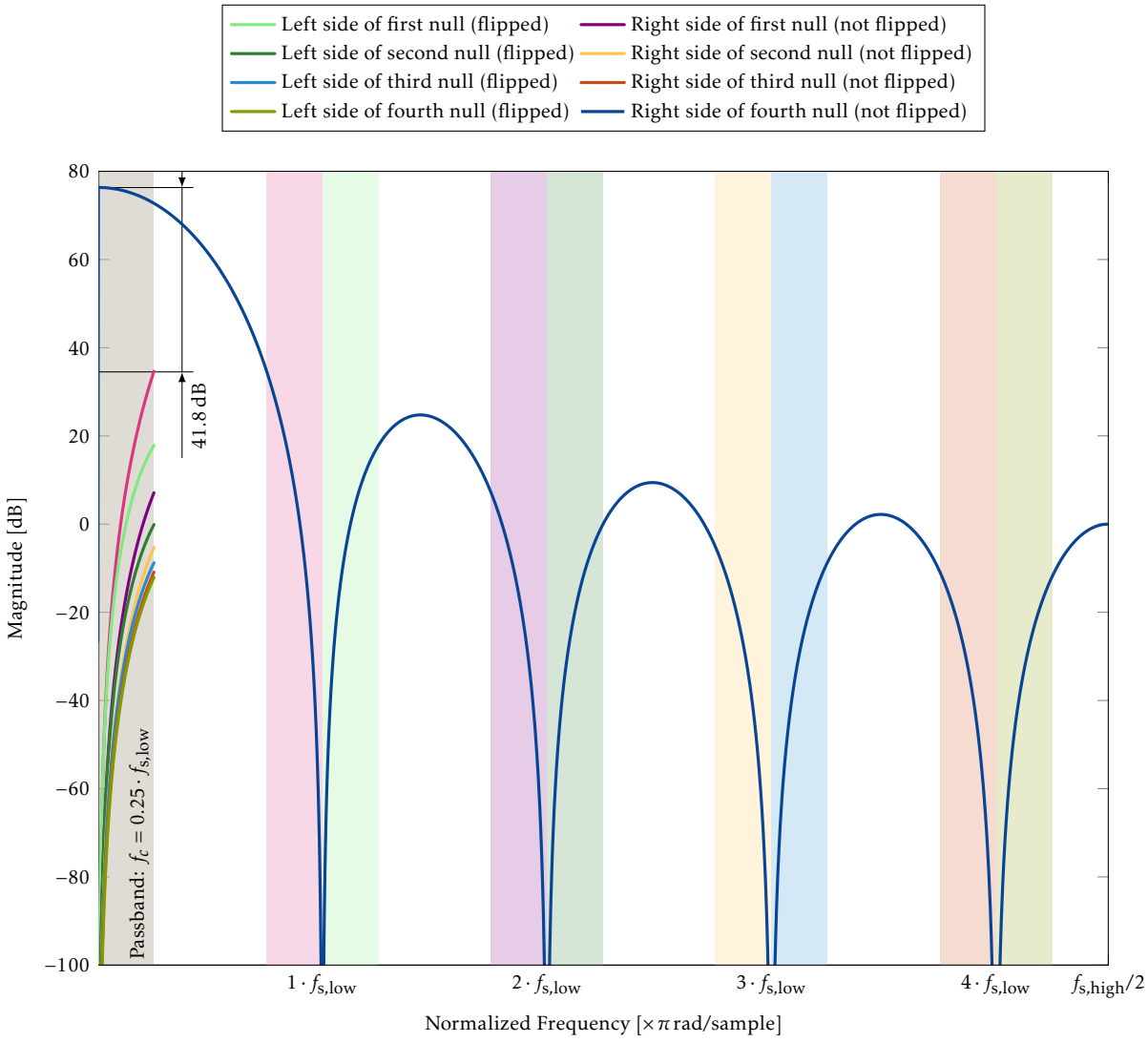


Figure 1.18: Passband aliasing for a CIC filter with $R = 9$, $N = 4$ and $M = 1$ and a cutoff frequency of $f_c = 0.25$, referenced to the lower sampling frequency $f_{s,low}$. The region of width f_c around every Mth null is folded back into the passband. The regions beyond that are of course folded back as well, but since we choose to arbitrarily limit the passband, those regions are not of interest to us. The resulting passband aliasing attenuation is 41.8 dB, as indicated in Table A.4 on page 93.

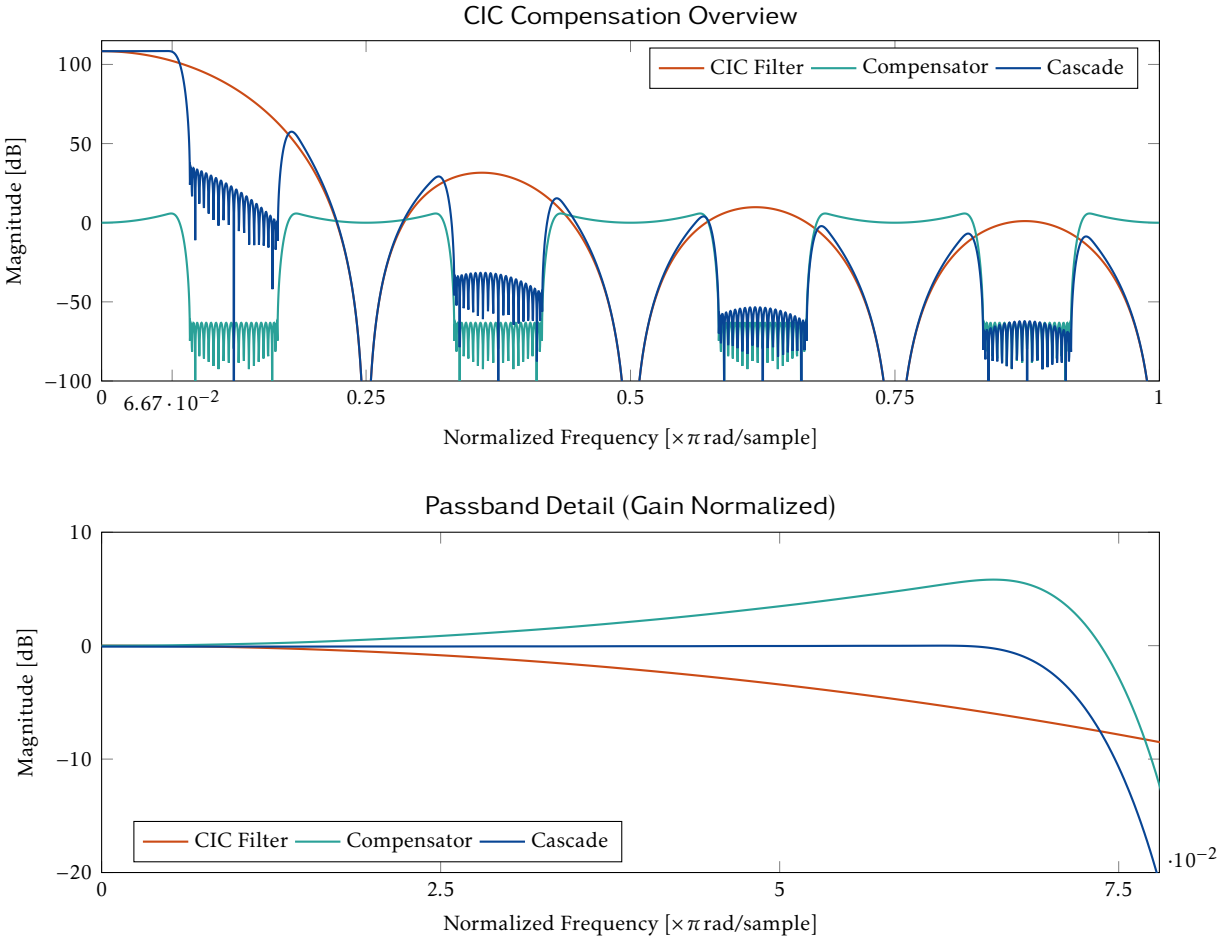


Figure 1.19: Frequency behavior of a CIC filter, its compensator, and the cascade of the two. Note the spectral copies of the compensator around the nulls of the CIC filter, i.e. the multiples of its outgoing sampling rate.

function from the jth stage up to and including the last:

$$H_j(z) = \begin{cases} H_I^{N-j+1} H_C^N = \displaystyle\sum_{k=0}^{(RM-1)N+j-1} h_j[k] z^{-k} & j = 1, 2, \cdots, N \\ H_C^{j-N} = \displaystyle\sum_{k=0}^{2N+1-j} h_j[k] z^{-kRM} & j = N+1, \cdots, 2N \end{cases} \tag{1.12}$$

where

$$h_j[k] = \begin{cases} \displaystyle\sum_{l=0}^{\lfloor k/(RM) \rfloor} (-1)^l \binom{N}{l} \binom{N-j+k-RMl}{k = RMl} j & = 1, 2, \cdots N \\ (-1)^k \binom{2N+1-j}{k} j & = N+1, \cdots 2N \end{cases} \tag{1.13}$$

are the impulse response coefficients. These functions are also derived by Hogenauer in [2].

In a filter with N stages, there are $2N+1$ error sources in the case of limited precision: Each stage, and the output register. Each error source is presumed to have white noise characteristics, i.e. its noise is uncorrelated to its input as well as other error sources. The error at the jth source is assumed to have a uniform probability distribution with a width of

$$E_j = \begin{cases} 0 & \text{without truncation or rounding} \\ 2^{B_j} & \text{otherwise} \end{cases} \tag{1.14}$$

where the number of bits discarded at the jth error source is B_j. The mean of this error is

$$\mu_j = \begin{cases} \frac{1}{2}E_j & \text{for truncation} \\ 0 & \text{otherwise} \end{cases} \tag{1.15}$$

and the variance comes out to

$$\sigma_j^2 = \frac{1}{12}E_j^2. \tag{1.16}$$

The total mean error at the filter's output due to the jth stage is

$$\mu_{T_j} = \mu_j D_j \tag{1.17}$$

where

$$D_j = \begin{cases} (RM)^N & j = 1 \\ 0 & j = 2, 3, \cdots, 2N \\ 1 & j = 2N+1 \end{cases} \tag{1.18}$$

is the *mean error gain* for the jth error source. Note that only the first and the last error source contribute to the filter's mean error at the output. This is because the sum of the impulse response coefficients is zero for all other stages. Consequently, whether one chooses to truncate or round is without consequence except in the case of the first and last error sources. In an analogous manner, the total variance computes to

$$\sigma_{T_j}^2 = \sigma_j^2 F_j^2 \tag{1.19}$$

where

$$F_j = \begin{cases} \sum_k h_j^2[k] & j = 1, 2, \cdots, 2N \\ 1 & j = 2N+1 \end{cases} \tag{1.20}$$

is called the *variance error gain* for the jth error source.

We can now compute the global mean error and variance of the filter:

$$\mu_T = \sum_{j=1}^{2N+1} \mu_{T_j} = \mu_{T_1} + \mu_{T_{2N+1}} \tag{1.21}$$

$$\sigma_T^2 = \sum_{j=1}^{2N+1} \sigma_{T_j} \tag{1.22}$$

These equations are used to calculate the properties of the CIC filter as deployed in our design. TODO: see section blabla

1.2.3.6 Compensators

1.2.3.7 Summary

In conclusion, the key properties of CIC filters are:

- They can be implemented both as decimators and interpolators.
- Neither multipliers nor storage for coefficients are needed.
- CIC decimation filters have a high gain, leading to significant register growth. Truncation or rounding can be used to limit the resource usage, both at the filter's output and internally.
- The three design parameters are the rate change R, the differential delay M and the number of stages N.
- The presence of passband droop requires a compensation filter to achieve a flat passband response.

Compensation Filter?

1.3 Multi-Stage Filter Designs

At first sight, the most obvious way to implement a downsampling system might appear to be to design one filter for each desired rate change factor. However, this would be highly impractical. Instead, multi-stage designs are usually used in practice. An in-depth discussion of their advantages and drawbacks was offered by Crochiere and Rabiner in [4]. A few aspects of multi-stage filter design which are relevant to our application shall be presented here.

To reduce aliasing effects in the passband, it is generally desirable to keep the width of the transition band roughly constant in relation to the width of the passband (visible in the filter's flank in Figure 1.5). As the downsampling ratio increases and the passband width decreases, the transition band therefore becomes progressively narrower, necessitating higher filter orders in a single-stage design.

This effect is illustrated in Figure 1.20, comparing two filters with a transition band 1/5 as wide as the passband for downsampling ratios of 2 and 4. The filter for $R = 4$ requires 60 coefficients, compared to 30 coefficients for the filter designed for $R = 2$. The other specifications (passband ripple, stop band attenuation) are identical. As an extreme case, a filter designed by Matlab with the same parameters for a downsampling ratio of $R = 625$ is 8860 coefficients in size.

Using multi-stage designs helps to avoid the need to implement filters with such large kernels. When cascading filters, it is the last stage of the chain which defines the overall passband and transition band width. The same overall transition band in absolute terms can be achieved with smaller filters in multi-stage designs, because a filter's transition band width is relative to the sampling rate at which it is running. This is shown in the bottommost plot in Figure 1.20.

Since cascading multiple filters does increase coefficient count (and storage), one might be inclined to think that not much has been won. Indeed, the overall number of coefficients is identical for both filters in Figure 1.20 (though, this obviously need not be so in other examples). However: Merely 30 multipliers and 29 adders (those of the first filter in the cascade) run at the incoming sampling frequency in the case of the cascade, while the components of the second filter run at half that. In the case of the single-stage filter, all its 60 multipliers and 59 adders run at the full sampling frequency. Distributing the calculations over two stages has therefore yielded an overall reduction in needed computation

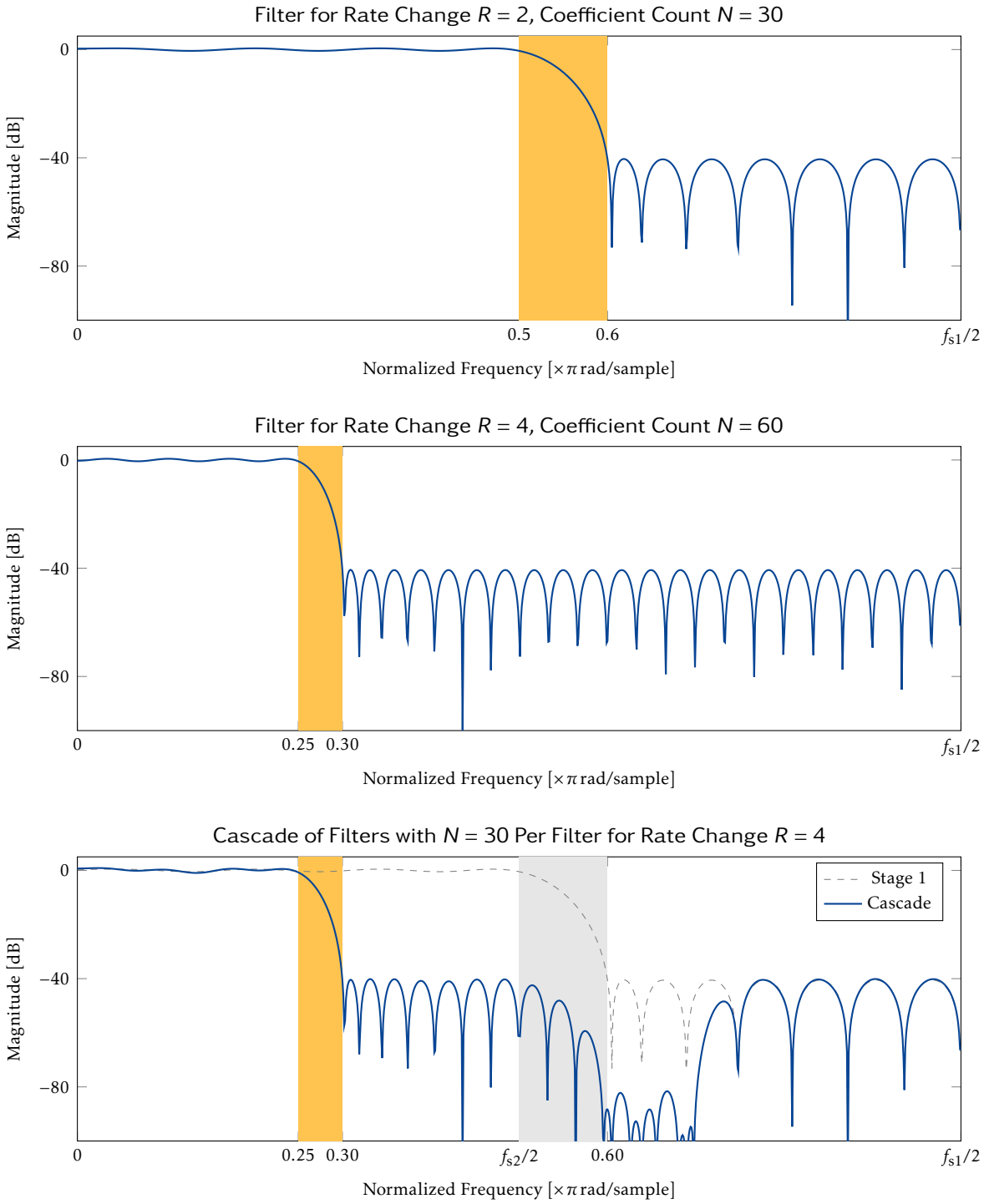


Figure 1.20: Two filters are used here: Both have a transition band 1/5 as wide as their passbands. The top filter is designed for a downsampling ratio $R = 2$ and has a coefficient count of $N = 30$. The second filter is designed for $R = 4$ and has 60 coefficients. Cascading two of the top filters into a two-stage design, depicted in the bottommost plot, results in the same overall passband and transition band width as the single-stage design.

power. As rate change factors increase, the benefits of multi-stage designs become even more pronounced [4].

Another advantage of cascading filters is that successive stages can be used to shape the overall frequency response, as seen in the case of the CIC compensator in Section TODO:reference. A drawback of cascades is that the ripple in their passband shows additive behavior, so the stages in a cascade of filters have more stringent ripple requirements in the passband then a single-stage. However, the cost for this is usually offset by the advantages of multi-stage designs.

When designing multi-stage filters, it can happen that the transition band of an earlier stage overlaps with the spectral copy of a later stage running at a reduced sampling rate. In that case, the stopband response of the cascade can have peaks exceeding the desired overall stopband attenuation. To prevent this, the following condition must be satisfied:

$$f_{\mathrm{st},1} < \frac{f_{\mathrm{s},1} - f_{\mathrm{st},2}}{R_1} \tag{1.23}$$

Where:

$f_{\mathrm{s},1}$: high sampling rate
$f_{\mathrm{st},1}$: stopband frequency of first filter
$f_{\mathrm{st},2}$: stopband frequency of second filter
R_1 : rate reduction in first filter

Figure 1.21 shows some examples for this condition being broken or fulfilled.

One last effect of note when cascading filters concerns stopband attenuation: When cascading two filters with different stopband attenuations, two things can happen:

- The second filter attenuates more strongly than the first one. This results in peaks above the second filter's stopband attuation in the regions where the spectral copies of the second filter's passband are located. This can be seen in the top plot in Figure 1.22.
- The first filter attenuates more strongly than the first one. In that case, the stopband region of the cascade right next to its transition band is less strongly attenuated than the stopband regions farther away from the edge. This case is shown in the middle plot in Figure 1.22.

Either of these two effects is usually not desired, barring special scenarios. In both cases, the resources invested into the steeper filter's stronger attenuation are wasted by the other filter's weaker stopband attenuation. It therefore makes more sense for the various stages in a cascade to have the same stopband attenuation, in general.

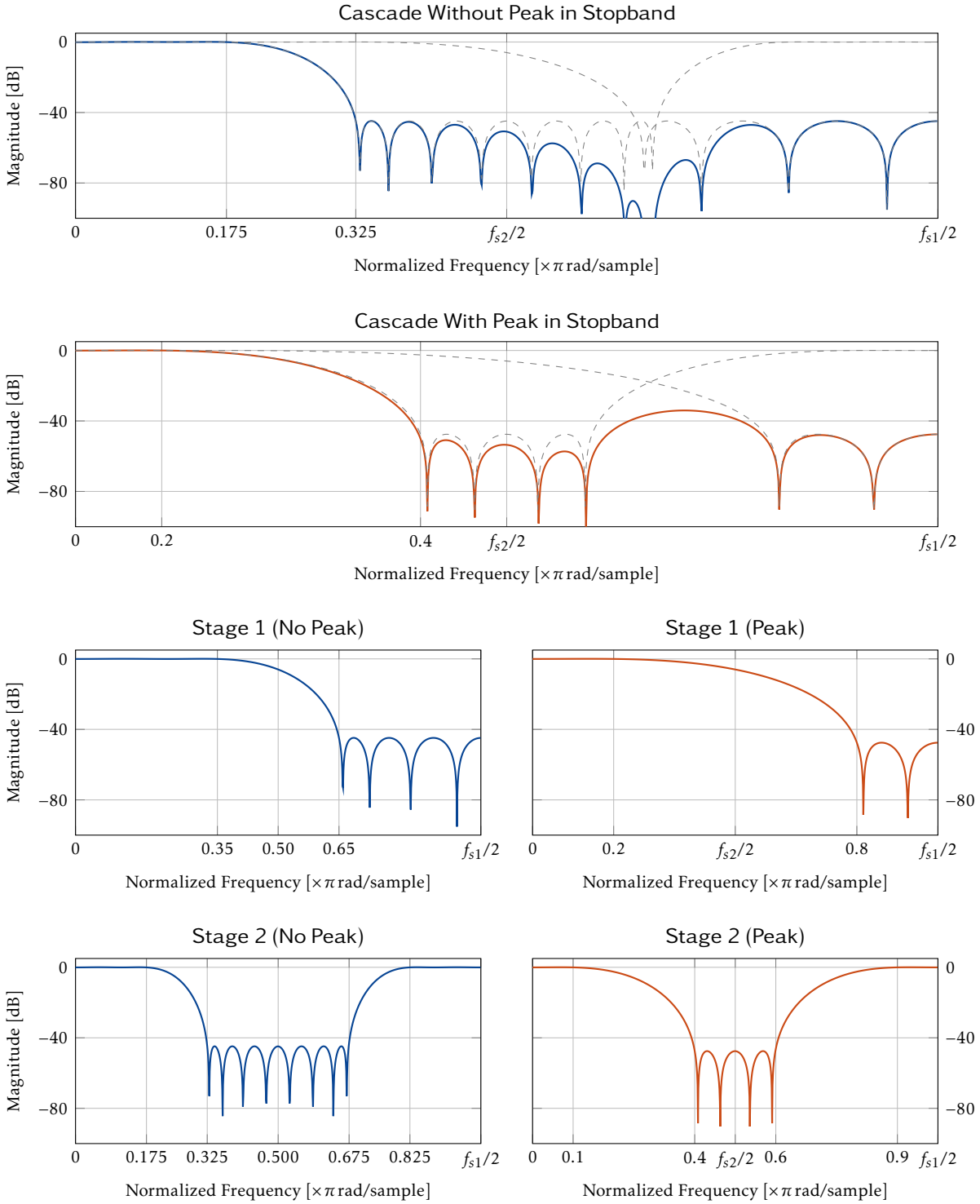


Figure 1.21: Comparison of two cascades: The first cascade (blue) has sufficient distance between the start of its stopband ($0.65 \cdot f_{s1}$) and the start of the transition band of the second stage's first copy around f_{s2} ($0.675 \cdot f_{s1}$). The second cascade (orange) has a peak in its stopband because the transition band of its first stage overlaps with the copy of the second stage ($0.8 \cdot f_{s1}$ vs. $0.6 \cdot f_{s1}$). *Note:* All frequencies are referenced to the high sampling rate f_{s1}.

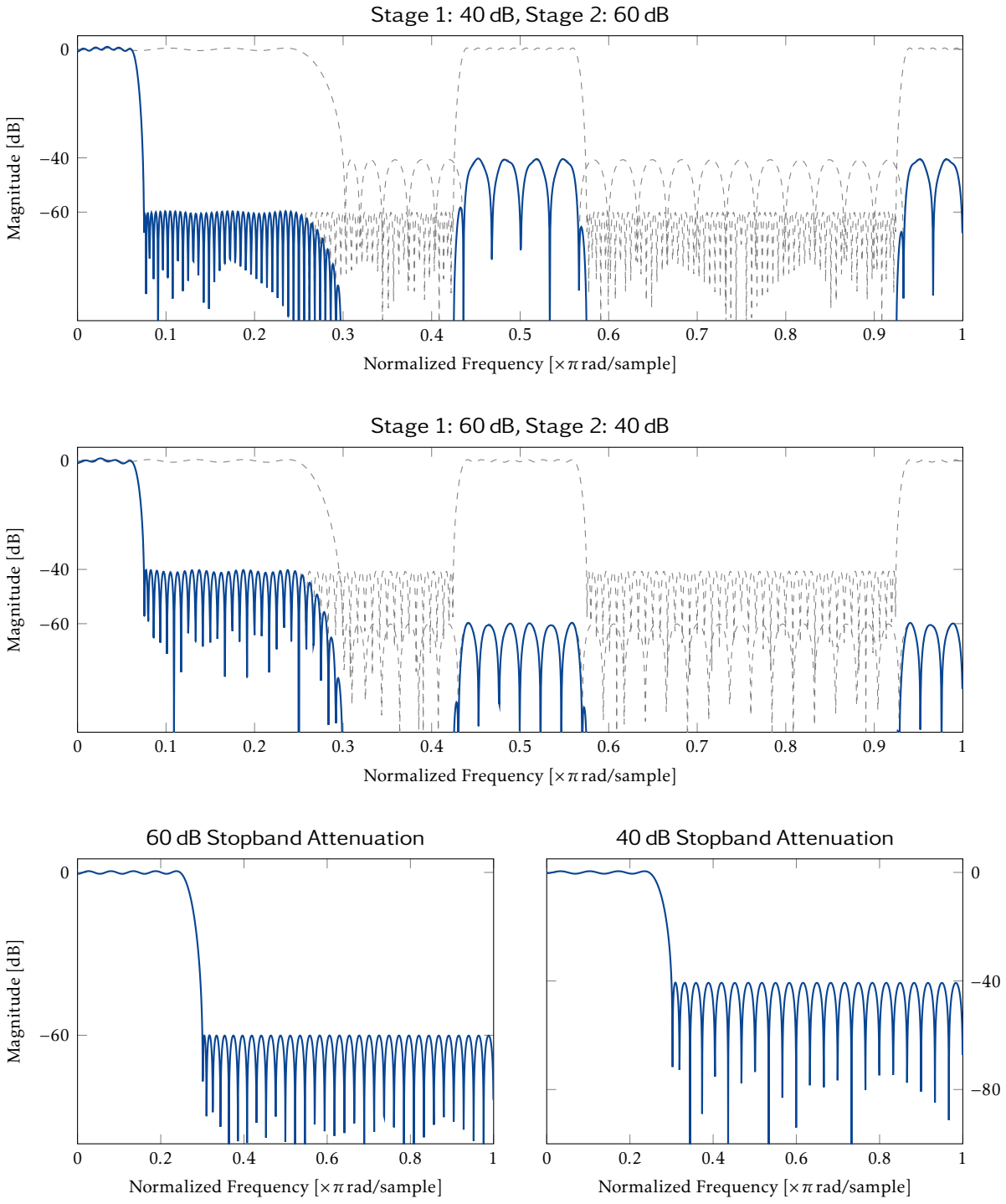


Figure 1.22: Cascading two filtes with different stopband attenuations: If a filter with stronger stopband attenuation is cascaded after a filter with weaker attenuation, the resultant cascade has peaks above the second filter's stopband attenuation (top plot). If the stronger filter is first in the cascade, the drop-off in the stopband right next to the transition band is weaker. The two filters by themselves are shown in the bottom plots.

CHAPTER 2

Mission

This chapter presents a brief summary of the hardware platform and some of its key characteristics which are of interest to us TODO: (impersonal). From that information, the objectives of this project are derived. Possible approaches to reach those objectives are presented and evaluated, and a decision is reached on how to achieve this project's aims. At the end of this chapter, the reader will know what we TODO: (impersonal) intend to do and why.

2.1 The Red Pitaya STEMlab 125-14

First, some key specifications of the hardware are given, to then proceed to the key problems which occur when downsampling with its stock configuration.

2.1.1 Hardware Overview

The device on which this project is based is the Pitaya STEMlab 125-14, pictured in Figure 2.1. STEMlab is a compact measurement instrument (it fits into the palm of a hand) which can replace more expensive devices like oscilloscopes by using a computer for data storage, processing and presentation.

Some of its key specifications relevant to us TODO: (impersonal) are:

- two high-speed analog inputs and outputs via coaxial connectors
- Uses a Linear Technology LTC2145-14 converter [5] on those inputs: 14 bits resolution at 125 $MS \cdot s^{-1}$ per channel.
- Xilinx SOC with an FPGA component for data processing on the device itself and two ARM Cortex9 cores for general-purpose tasks
- Ethernet and USB PHYs for data transmission and device control
- Has its own operating system, a GNU/Linux distribution (Ubuntu is used in our TODO: (impersonal) project), running on the ARM cores.
- The software used is open-source and available under [6].

More comprehensive documentation can be found at [7]. A block diagram with the system's key components is shown in Figure 2.2.

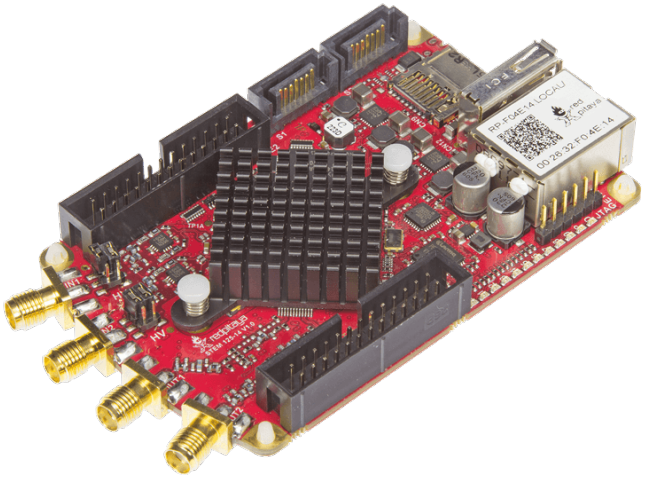


Figure 2.1: Photo of the STEMlab. Source: [8]

2.1.2 Downsampling on the STEMlab With Stock Configuration

The STEMlab enables downsampling by powers of 2 in its stock configuration, in a range between $1 = 2^0$ and $65536 = 2^{16}$. The exact manner in which this downsampling process is performed is not easily deducible from public information; the documentation on the internals of the FPGA codebase is rather sparse. However, work conducted by our predecessors has shown that the STEMlab uses a moving averaging filter for this process [1].

The moving averager's transfer function is

$$H(z) = \frac{1}{N+1} \sum_{k=0}^{N} z^{-k}. \tag{2.1}$$

A high degree of similarity is immediately recognizable when comparing this to the transfer function of a CIC filter in Equation 1.9 (page 12). Indeed, a cascade of moving averagers would almost yield a CIC filter, save for the gain, which could be easily adjusted after the fact if needed.

A moving averager as a decimation filter is relatively cheap to implement when using only decimation rates of powers of two, as is the case for the stock software of the STEMlab. The weight for each coefficient of z^{-k} will be $\frac{1}{N+1} = \frac{1}{R} = 2^{-m}, m \in \mathbb{N}_0$, meaning the computations can be performed without multipliers by performing a bit-shift operations to the right. The primary disadvantage which results from these rate reduction factors is that the resulting reduced sampling rates are not very "nice" numbers, so to speak, since the incoming sampling rate is 125 MHz, and therefore not a power of 2.

Just like a CIC filter by itself, however, a moving averager also makes for a rather poor lowpass filter, for two reasons: Passband droop and poor stopband attenuation (equivalent

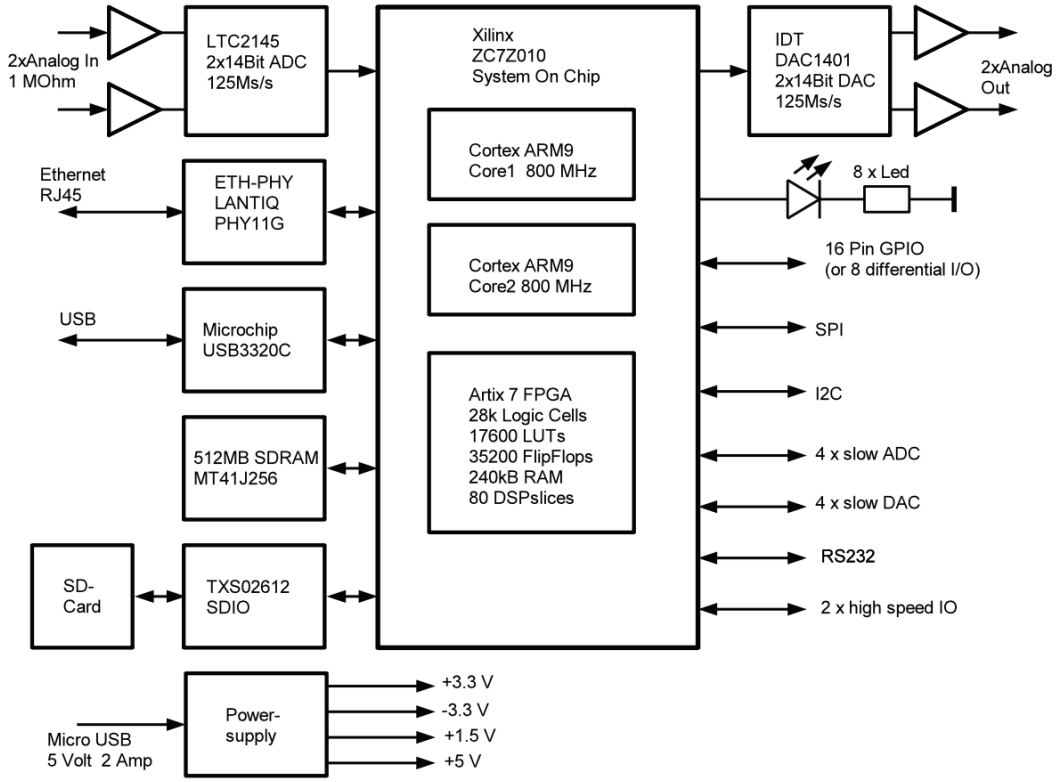


Figure 2.2: Block diagram of the Red Pitaya STEMlab. Source: [9]

to a single-stage CIC filter). Figure 2.3 shows the frequency response for the case of $R = 8$ and a signal of 5 MHz. The incoming sampling rate is 125 MHz, the reduced sampling rate is 125 MHz ÷ 8 = 15.625 MHz. Therefore, anything above half that frequency is aliased back into the region below 15.625 MHz ÷ 2 = 7.8125 MHz. In the case of a signal at 5 MHz, this results in an aliasing attenuation of a paltry TODO: (objectivity) 8.06 dB. Combined with the passband droop of 1.48 dB (a signal attenuation of almost 16 %) at that frequency, this makes for a margin of a mere 6.58 dB between the signal's attenuation and the aliasing attenuation!

As a benchmark for or own solution, we TODO: (impersonal) will use measurements conducted by our predecessors with the STEMlab's stock configuration, listed in Table 2.1. The measurements do not actually suggest a very bad performance of the filter. This is because they were conducted at specific harmonic frequencies. The primary issue with a moving averager is less SNR for specific frequencies, but aliasing when measuring a non-harmonic signal which has frequency components above $f_{s,low}/2$. In that case, the considerations from Figure 2.3 become crucial to understanding the aliasing issue. This behavior was also confirmed in [1]. Our TODO: (impersonal) primary aim is therefore more to reduce these aliasing effects rather than purely trying to improve SNR for specific single frequencies.

In conclusion, we can formulate three main objectives:

- Reduce aliasing of out-of-range frequency components into the passband.

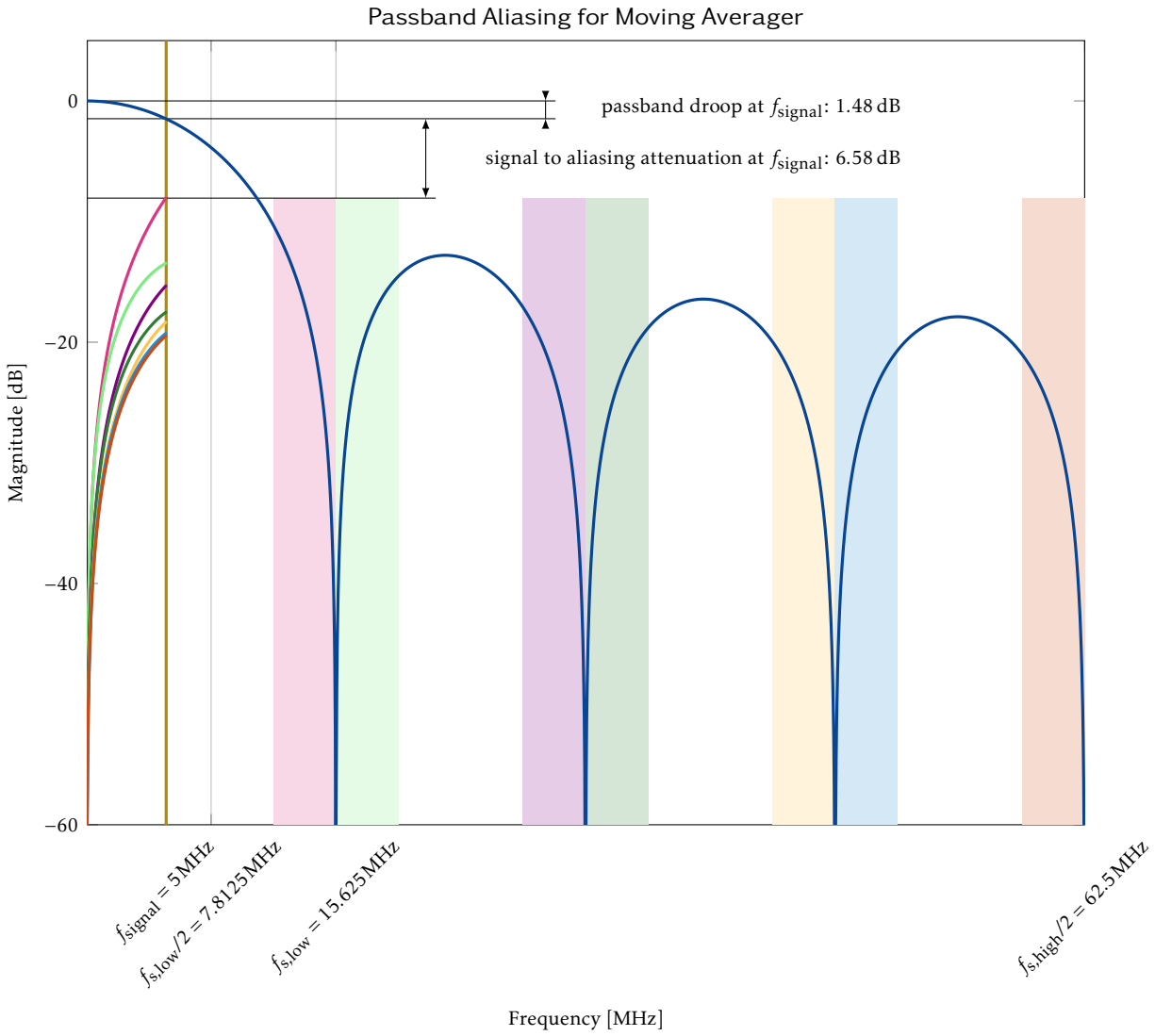


Figure 2.3: Frequency response, passband droop and aliasing attenuation for a moving average filter of order 7 for a decimation by a factor of 8. The incoming sampling frequency is 125 MHz, the outgoing sampling frequency is 15.625 MHz, the measured signal has a frequency of 5 MHz.

Table 2.1: Measurement results for STEMlab 125-14 from [1]. SNR was determined for a specific harmonic frequency signal for each sampling rate.

R			$f_{s,low}$ (kHz)	f_{signal} (kHz)	SNR
2^0	=	1	125 000	10 000	62.8 dB
2^3	=	8	15 625	5000	69.8 dB
2^6	=	64	1953	500	76.0 dB
2^{10}	=	1024	122.1	40	76.4 dB
2^{13}	=	8192	15.26	5	82.4 dB
2^{16}	=	65536	1.907	0.5	83.5 dB

- Improve SNR TODO: (all thre points sloppy formulated?).
- Have a nicer set of reduced sampling rates.

2.2 Possible Solutions

Based on the previous section, it is clear that the downsampling filter from the STEMlab's stock configuration must be replaced if better aliasing attenuation is to be attained. Furthermore, since a new filtering system is being implemented anyway, one may wish to change the rate change factors in order to achieve a set of nicer-looking reduced sampling rates. The question now becomes: *How*?

Problematic is that as of spring 2017, the official code by Red Pitaya base has been split into two: There is a new development branch bringing major changes, while the old code (which has been used in the projects preceding this one) is no longer being maintained, as explained by one of the developers (see [10]): TODO: (citation wrong? gertiser will incapacitate us?)

> Current development is done on the `mercury` branch on the FPGA subproject of the same name. The code is still under heavy development and not stable. All other FPGA subprojects will not be developed further.

Essentially, this presents us with three options on how to proceed:

- Use the old Red Pitaya codebase, and implement our new filtering system based on that.
- Use the new Red Pitaya codebase. Due to this codebase being unstable according to the developers themselves as of the time of this project, we consider this to be an unviable option.
- Use an alternative ecosystem, either by a third party (if one can be found) or developed by ourselves.

In order to have a somewhat objective measure to compare the first and last option, a decision matrix is used; see table 2.2. In the following paragraphs, the criteria and weighings in the table are elaborated upon. The table uses a scale of 1 through 6, with 1 being the worst and 6 being the best score. At the end, totals are tallied and compared.

Reliance upon others: Using the existing Red Pitaya ecosystem would also couple us to any problems inherent to that platform and to the solutions developed by preceding projects based on it. Any bugs encountered in the Red Pitaya ecosystem would either require a bugfix by the manufacturer or a workaround on our part. Since the old Red Pitaya codebase is no longer being maintained by the developers, bugfixes will not be available, leaving us to clean up any potential issues inherent to the codebase. If we were to implement our own solution instead, we would be less reliant upon others to fix any inherent problems to the code. These factors lead us to give the existing codebase a rather weak rating of, compared to a strong rating for the choice of pursuing our own solution.

Flexibility: Obviously, implementing our own system would give us much higher flexibility than using the existing ecosystem. The only two true limitations would be the time and hardware resources available to us. the official codebase would limit us to its capabilities. Adding new functionality to the existing codebase would be possible, but adapting

software for purposes for which it was not originally intended does tend to be a time-consuming procedure in our experience. Therefore, the option of developing our own solution wins out again here.

Complexity of complete system: This refers not just to the complexity of the components we work on, but of the entire platform and ecosystem. As anyone who bothers to peruse the Red Pitaya codebase ([6]) can dedude, it is a large ecosystem with many features and capabilities, most of which are not relevant to our needs. A custom system developed for the specific needs of this project would be much leaner and have fewer points of failure.

Labor costs: Here is where using the existing codebase would be beneficial in our view. While understanding its inner workings would doubtlessly be required and take a significant amount of time (particularly in view of the sparse documentation at this point in time), developing a completely new system more or less from scratch would be a much costlier undertaking in terms of required man-hours, we estimate.

Chances of success: Basing our work on the existing codebase would unburden us of the legwork needed to get basic functionality up and running. We could exchange the filter components of the existing system with our own, but leave most of the remaining system untouched. The challenge would be to understand the system well enough to do this. Building our own system would require re-implementing more components of the Red Pitaya platform than just the filters. As each additional task increases the risk of failure, this is not without risk to the overall success. Overall, we consider this factor to be roughly even between the two choices.

Robustness of third-party components: This criterion takes into account our assessment of the reliability and dependability of any component we use which is not developed by ourselves, along with its documentation and manufacturer support. In the case of the Red Pitaya codebase, this would primarily comprise the codebase itself as well as any components for it developed by other parties. If we were to develop our own solution, the building blocks would primarily be the Xilinx toolchain and libraries for the SoC.

Because the Red Pitaya codebase is rather large, not well documented, comparatively new and no longer being maintained, and the company is small and more likely to be stretched thin, we do not score the Red Pitaya codebase highly here. The Xilinx toolchain and FPGA libraries, while undoubtedly not without bugs, have had a lot of time to mature on the other hand. Also, Xilinx is a big company with vast resources, so any bugs which are encountered in their products are more likely to be addressed in our view. For these reasons, we score developing our own solution higher than using the existing cosebase.

Reparability: If any issues are found in the resulting product, we are more likely to be able to fix them if it is our own codebase rather than that by a third party.

Long-term viability: Developing our own solution would allow us (or anyone else wishing to base their work off of ours) to address future needs relatively easily. Using the existing code base would make this more difficult in our view.

Conclusion: Developing our own solution does carry a significant risk and is likely to require more work than basing our application on existing work. But in light of the mentioned drawbacks of the latter approach, we still find that it is the preferable approach and is more likely to lead to a successful outcome.

Table 2.2: Decision matrix comparing the usage of the existing Red Pitaya ecosystem against building our own data acquisition system. Weighing: Scale of 1 (worst) to 6 (best). More total points is better.

	RP	Custom
reliance on others	2	5
flexibility	2	5
complexity of complete system	1	4
labor costs	4	2
chances of success	4	4
available documentation for used building blocks	3	5
robustness of third-party components	2	5
reparability	2	5
long-term viability	2	4
Total	21	37

2.3 Concept

Having concluded that we shall develop our own solution from scratch, we can now develop a concept for how that solution will look. On the most fundamental level, it will require the following components:

- a custom FPGA firmare for data acquisition and filtering
- a new scope application for data visualization
- an interfacing layer between the FPGA and the scope
- optionally, the possibility to connect to other applications like Matlab

The following sections elaborates on the general shape of our solution. Chapters 4 through 6 explain the components in detail, while Chapter 3 documents the filter design process, which is rather separate from the firmware and software and therefore a dedicated chapter.

2.3.1 FPGA Components

On the most abstract level, the FPGA part of our system will need to be able to

- acquire data from the ADC,
- decimate and filter it,
- and pass it on for further processing.

Due to the platform's open nature, there exist some projects for the STEMlab by parties other than Red Pitaya themselves. One of these is *Red Pitaya Notes* by Pavel Demin, available at [11]. As part of that project, Mr. Demin has implemented an ADC core for the FPGA which acquires data from the ADC's two channels and passes it on via an AXI4 streaming interface[1]. Since this project is open-source and has a permissive license (MIT license [12], template text can be found in Appendix D.1 on page 110) and fulfills our main technical requirement (easily interfaceable), it is used in our project to interface the FPGA with the ADC.

[1] Xilinx's proprietary general-purpose interconnect for moving large amounts of data around an FPGA

For filtering, one can either implement custom filter topologies from basic FPGA building blocks (adders, multipliers, etc.), write completely custom VHDL or Verilog code, or use ready-made blocks, if available. Xilinx provides such FPGA blocks for FIR and CIC filters, both of which come with excellent documentation (see [13] and [14], respectively). In order to avoid re-inventing the wheel, so to speak, and take advantage of some of the advanced features offered by these two blocks, we use these two building blocks in our design.

Interfacing between the FPGA and the outside world requires a component which can take data from the filters and hand it off to the GNU/Linux running on the ARM cores. Additionally, the user must be able to trigger measurements from the operating system. Luckily, Mr. Hüsser TODO: (impersonal) already developed such a data logging core for a Xilinx FPGA in an earlier project [15], which comes with a kernel module to interface with GNU/Linux. With some minor adaption work, this core should suit our needs nicely.

The last component needed for the FPGA is a control logic to set the decimation rate and enable or disable specific components of the data processing chain. This should be fairly straightforward, and will be implemented by ourselves as a custom block.

In summary, the concept for the FPGA data processing system looks as follows:

- The ADC is accessed via Pavel Demin's ADC core.
- Filtering the data is conducted with Xilinx's CIC and FIR compilers.
- For interfacing with GNU/Linux, Mr. Hüsser's data logging core is used.
- A custom control logic configures the data processing chain on-the-fly through user input.

2.3.2 Interfacing Layer

The interfacing layer is responsible for sending the data from the STEMlab to the user application running on a computer. This is done via a Gigabit Ethernet connection. As is easily apparent, this connection is far too slow for moving all data off the device which is generated by the ADC (about $3.7\,\mathrm{GiB\,s^{-1}}$), hence the need for decimation (among other reasons).

For the interfacing layer, a server application is run on the STEMlab, to which a client can connect. The server takes data from the logger's kernel module, processes and packages it as necessary, and sends it out over Ethernet. The application is documented in detail in Chapter 5, starting on page 47.

TODO: More blabla?

2.3.3 Scope

The scope is the main interface between the end user and the STEMlab in our system. Through it, data can be visualized and analyzed in both the time and frequency domain.

In order to ensure maximum portability, the scope is implemented as a web application rather than a cusom binary. This allows it to function on any operating system with a reasonably modern browser. A through documentation of its capabilities and implementation is available in Chapter 6, beginning on page 6.

TODO: more blabla?

2.3.4 Summary

in summary, our concept looks as follows:

CHAPTER 3

Filter Design

Some key points underlying the theory of filters have been treated in Chapter 1, and Chapter 2 defines the overall objectives of this project; implementing a custom filtering system on the FPGA among them. This chapter will now develop a concrete concept for that filtering system, address some of the issues encountered when moving from the theory of filters to the practice of designing them, and then define the actual filters which are to be used on the FPGA. The implementation of those filters on the FPGA is addressed in the following chapter, beginning on page 41.

3.1 Requirements

The overarching objective is downsampling the signal coming out of the ADC. This section will derive upper and lower boundaries for the downsampled frequency range, and then define the specific downsampling ratios to be used.

The total data rate from the ADC is

$$\begin{aligned} S &= 125\,\text{Msample}\cdot\text{s}^{-1} \\ N_{\text{ch}} &= 2 \\ B_{\text{ch}} &= 14\,\text{bit}\cdot\text{sample}^{-1} \\ B_{\text{ch,pad}} &= 2\,\text{bit}\cdot\text{sample}^{-1} \\ B_{\text{ADC}} &= N_{\text{ch}}\cdot\left(B_{\text{ch}}+B_{\text{ch,pad}}\right) = 32\,\text{bit}\cdot\text{sample}^{-1} \\ R &= S\cdot B_{\text{ADC}} \qquad\qquad = 4\,\text{Gbit}\cdot\text{s}^{-1} \end{aligned} \tag{3.1}$$

Where:

S	: sampling rate
N_{ch}	: number of channels
B_{ch}	: channel width
$B_{\text{ch,pad}}$	: padding per channel
B_{ADC}	: total width of bit stream out of ADC
R	: total data rate out of ADC in bit

Table 3.1: The chosen downsampling ratios, their prime factor decompositions, the downsampling ratios distribet across stages, and the resultant sampling rates

R	Decomposition	Stages	f_s (kHz)
5	$5 = 5^1$	5	25 000
25	$5 \cdot 5 = 5^2$	$5 \rightarrow 5$	5000
125	$5 \cdot 5 \cdot 5 = 5^3$	$25 \rightarrow 5$	1000
625	$5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 = 5^4$	$25 \rightarrow 5 \rightarrow 5$	200
1250	$2 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 = 2^1 \cdot 5^4$	$125 \rightarrow 5 \rightarrow 2$	100
2500	$2 \cdot 2 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 = 2^1 \cdot 5^4$	$125 \rightarrow 5 \rightarrow 2 \rightarrow 2$	50

The upper boundary for the resulting sampling rate is set by the STEMlab's network connection, which has a data rate of $1000\,\mathrm{Mbit \cdot s^{-1}}$. A downsampling factor of at least 4 is therefore required for real-time data transmission. Because 125 is not divisible by 4, a factor of 5 is chosen instead. This makes for a resulting data rate of $800\,\mathrm{Mbit \cdot s^{-1}}$, which should also be easily sufficient for protocol overhead.

On the lower end of the spectrum, the system should still be able to process audio signals. Common sampling frequencies for audio are 44.1 kHz for audio CDs, and 48 kHz for the audio component of audio-visual applications. Neither of these frequencies fit nicely into 125 MHz (requiring large prime factor for the rate change), so the lower boundary is specified as 50 kHz, corresponding to a downsampling ratio of 2500.

To cover additional use cases, some additional sampling frequencies between these two boundaries are specified. Table 3.1 contains the complete list of downsampling ratios, along with the corresponding sampling frequencies.

3.2 Cascade Concept

Based on the downsampling factors from Table **??**, this section presents the general concept for the cascades which implement those rate changes.

As discussed in Section 1.3, implementing high downsampling ratios in a single stage is generally not a sound design choice. Consequently, the downsampling ratios from Table **??** must be decomposed into smaller factors, which can then be distributed along a chain. These factors must fulfill the following criteria:

- Filters for different stages should be re-usable across multiple downsampling ratios in order to save resources. Rate change factors which are common to multiple rate change factors are therefore preferred.
- The factors for the individual stages should be large enough to be of utility, but small enough so as not to make the resulting filter impractically narrow.

CIC filters are well-suited for large rate changes, but are not an optimal solution for smaller ones. As an example of a low-rate change CIC filter, Figure A.2 on page 89 in Appendix A.1 shows the frequency response of a CIC filter with a rate change of 2.

Based on that observation, it is reasonable to implement the lower rate change factors without CIC filters, while using CIC filters as the first element in the filter chain for the higher rate changes. This allows taking advantage of the CIC filter's high computational efficiency for large downsampling rates, while still having good frequency response behavior for the lower rate changes. Table 3.1 shows the prime factor decomposition for the overall rate changes, as well as the factors chosen for the individual filter chain stages.

Other choices are of course possible. Particularly in the case of $R = 625$, one may choose to implement a chain of $125 \rightarrow 5$ instead of $25 \rightarrow 5 \rightarrow 5$. The two implementations as they would be in the final design are compared in Figure B.1 on 95 in Appendix B.1. While the $125 \rightarrow 5$ chain would offer better stopband attunation behavior over certain frequency ranges and therefore improved SNR, the $25 \rightarrow 5 \rightarrow 5$ chain offers the advantage that if the design is ever changed and only the higher sampling rates are implemented (removing the chains for $R = 1250$ and $R = 2500$), it can re-use the elements from the higher chains. The $25 \rightarrow 5 \rightarrow 5$ chain is therefore chosen.

3.3 Filter Specifications

Continuing from Table 3.1, this section presents the concept for the cascades which are used in our design. Specifically, requirements and constraints for the filters in those cascades are specified. For this purpose, it is no longer sufficient to merely consider the filters in the mathematical sense; resource usage on the hardware must be taken into account.

The hardware places two main contrainst on the design:

- Number of available LUTs: 17 600
- Number of available DSP slices: 80

The number of LUTs is relevant for storage (filter coefficients), the CIC filters[1], the rest of the processing system, and control logic. The DSP slices can therefore be reserved for the FIR filters. Because the device has two channels, only 40 slices may be used per channel.

In order to have a realistic gauge for resource usage of the FIR filters, it is necessary to keep in mind the two factors which primarily influence resource usage:

- the sampling frequency at which the filter runs (its incoming sampling rate),
- and the number of coefficients, and therefore, adders and multipliers.

It is therefore important to know which filters in the design run at which sampling rates. For this to be possible, a concept of the six different filter chains is needed. Based on Table 3.1, Figure 3.1 depicts that concept. It includes all the required filters of the design, including the compensators for the CIC filters. Note that one of the compensators is itself used as a decimation filter in the case of the $R = 1250$ and $R = 2500$ chains.

Because the final filter in a cascade is the one which determines the overall transition band (see Section 1.3), it is desirable to have maximally steep output filters in a cascade. Therefore, the filter `5steep` should be as sharp as possible. Since that filter is not just used as the final stage in some cascades, but also as the single filter for the $R = 5$ chain, it runs at the highest sampling frequency[2]. `5steep` is therefore the most critical filter in terms of resource usage. The filter `5flat` also runs at the highest sampling frequency in the $R = 25$ chain, but because it is not the final filter in that chain, it need not be as steep.

While it is possible to estimate the needed resources of a given filter design, reliable figures are best obtained by way of experiment. The FPGA toolchain might make optimizations which are hard to take into account when performing estimates by hand. The results of these measurements are available in Appendix B.2 on page 94. Based on those numbers, a filter size of around 250 is determined to be a reasonable boundary for `5steep`; a filter of that size uses about 25 DSP slices, leaving 15 slices for other filters per channel.

[1] The CIC compiler block by Xilinx can be configured to utilize LUTs instead of DSP slices for its computations. The FIR compiler can only use DSP slices for its computations.

[2] It should be noted at this point that a filter which is configured to run at a high sampling rate can be re-used as a lower stage in a cascade in the Xilinx toolchain. The filter's behavior in that case is correct, even when being run at a lower rate than maximally possible.

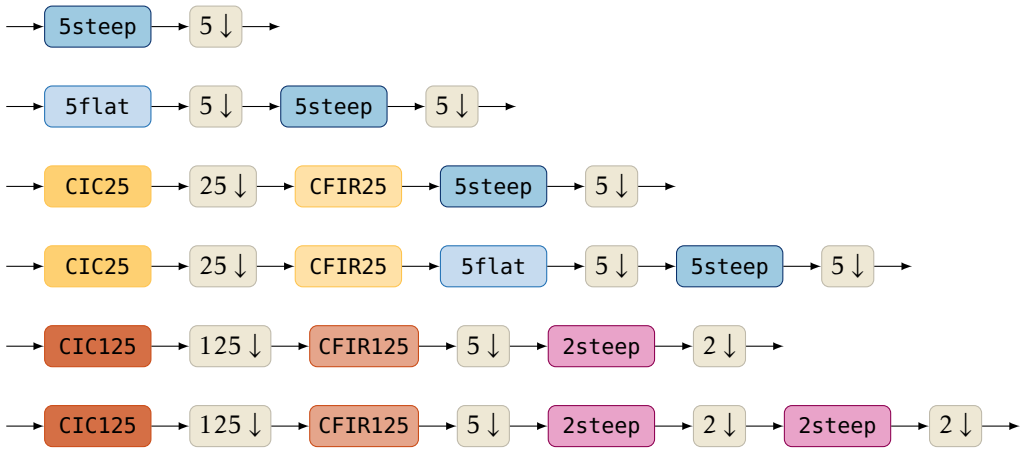


Figure 3.1: The concept for the filter chains

number of slices for the remaining filters (per channel). Of these, `5flat` is the most critical, because it must also be able to run at the full incoming sampling frequency. `2steep` runs at a much lower sampling rate and can therefore be of significant size without a notable penalty in resource usage. The final implementation of `5steep` is actually smaller, at a length of 204 coefficients and a DSP slice count of 22 per channel.

Based on these findings and the measurements of the STEMlab's stock configuration from Section 2.1.2, it is possible to define performance specifications for the filters without needing to prod in the dark, so to speak. The following paragraphs explain the considerations which lead to the final filter specifications. The results are summarized in Table 3.2.

Requirements for `5steep`: Based on the results of the moving averager used in the STEMlab's stock configuration, even a FIR filter with moderate performance characteristics should already offer significant improvements. In order to achieve notable gains over the default configuration, the following performance goals for `5steep` are specified, according to the pattern explained in Section 1.2.2:

- Passband ripple: better than 0.25 dB
- Stopband attenuation: 60 dB or better
- Transition band width: $0.05 \cdot f_s/2$ or better

The stopband attenuation criterion is also applied to all other filters. Anything else would be a waste of resources, as shown in Figure 1.22 in Section 1.3.

Requirements for `5flat`: This filter need not have a drop-off as sharp as `5steep`, as long as the end of its transition band does not overlap with the first spectral copy of `5steep` (see Figure 1.21 in Section 1.3). However, because it is in a cascade with `5steep`, a sharper requirement on its passband ripple is imposed, in order not to worsen overall passband ripple behavior of the cascade too much.

Requirements for `2steep`: In theory, the first decimator-by-two in the chain for $R = 2500$ could be designed with a wider transition band than the one which is being used as the last stage (hence the designator `5steep`). However, because these filters do not have to run

Table 3.2: The target filter specifications. These parameters are based both on the desired frequency domain behavior of the filters as well as the feasibility of implementation in terms of resource usage. For resource considerations, the results from Appendix B.2 are used as a guideline.

Filter	Passband Edge ($\times\pi$rad $\cdot$ sample^{-1})	Stopband Edge ($\times\pi$rad $\cdot$ sample^{-1})	Passband Ripple (dB)	Stopband Attenuation (dB)
`5steep`	0.2	0.225	0.2	60
`5flat`	0.2	0.3	0.05	60
`CIC25`	0.008	N/A	N/A	60
`CFIR25`	0.008	0.016	0.05	60
`CIC125`	0.0016	N/A	N/A	60
`CFIR125`	0.0016	0.0024	0.05	60
	Transition Band Width ($\times\pi$rad $\cdot$ sample^{-1})			Stopband Attenuation (dB)
`2steep`	0.004	N/A	N/A	60

at high sampling rates, the amount of DSP slices they use is very low (1 DSP slice per channel), so the same filter can be re-used without a resource penalty.

TODO: halfband

Requirements for `CIC25`: The relevant design criteria for the CIC filter are its stopband attenuation, its decimation rate, and the cutoff frequency/desired passband width (see Figure 1.17 in Section 1.2.3.2). The cutoff frequency is chosen such that it matches the frequency band which is of interest at the end of the filter chain for $R = 125$, i.e. $f_P = f_{s,high}/R = 0.008$. This means that the passband of `CIC25` and `CFIR25` combined is too wide by a factor of 5 for the $R = 625$ chain, but this is of no concern because it will be cut off by `5flat` as the last stage in that case. This allows the re-use of the two filters across both chains without changing their design parameters.

Requirements for `CIC125`: The same considerations as for the other CIC filter apply. The filter and its compensator are specified for a rate change of 125 and 5 instead of 25 and 1, respectively, and the cutoff frequency is set to match the filter chain of $R = 1250$. This makes it twice as wide as it needs to be for $R = 2500$, which is corrected by a second halfband filter.

Requirements for compensators: The compensators are specified according to the considerations laid out in Section 1.2.3.6, with the added feature of `CIC125` also being used as a decimator.

Summary: With the above considerations and the experimental results for resource usage from Appendix B.2, it is possible to formulate a complete set of specifications for the filters. They are compiled in Table 3.2. Translating the specifications from Table 3.2 into absolute frequencies results in the values from Table . The frequency responses of all filters and filter chains are listed in Appendix B.4, starting on page 96.

Table 3.3: The expected relative and absolute transition band widths of the various filter chains, based on the specifications from Table 3.2.

Chain	Relative TB Width of Final Filter ($\times\pi$rad $\cdot$ sample^{-1})	Absolute TB Width of Chain (kHz)
5	0.025	1562.5
25	0.025	312.5
125	0.025	62.5
625	0.025	12.5
1250	0.040	4.0
2500	0.040	2.0

CHAPTER 4

FPGA

Broadly speaking, the FPGA firmware (the bitstream) consists of three main components: The ADC control logic, a data acquisition core which writes data to RAM and is responsible for triggering, and, most importantly, the filter chains that connect the two. Figure 4.1 shows a schematic of how these components fit together, and how they are related to the overall STEMlab system. This chapter first presents a rough outline of the FPGA toolchain, and then provides more specific information on each of the three FPGA subsystems in Sections 4.2, 4.3 and 4.4, respectively.

4.1 The Xilinx Toolchain

The bitstream is compiled using Vivado, Xilinx's own IDE, a tool that can do everything around Xilinx FPGAs. It does the crucial parts right and can be interfaced with using Tcl.

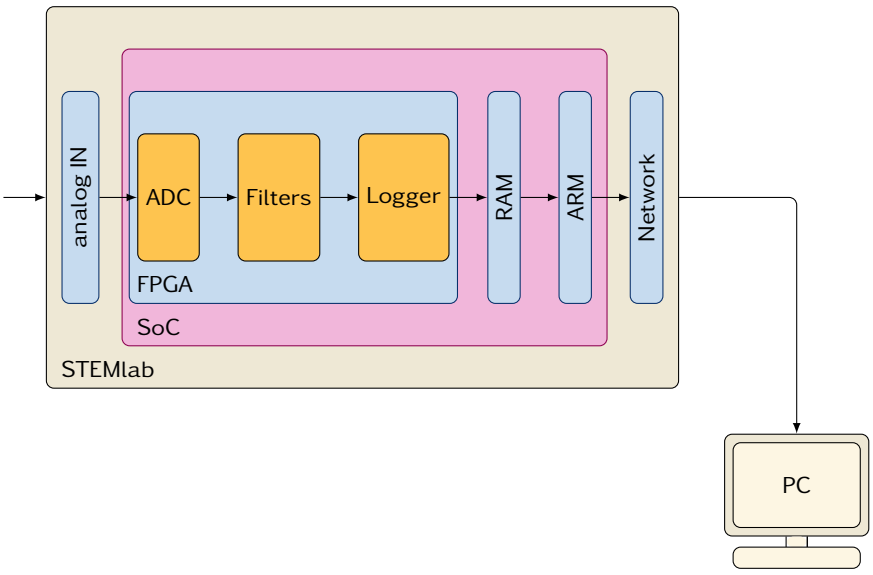


Figure 4.1: The structure of the FPGA code.

This is very convenient, as a project can be replicated indempotently[1] whenever a rebuild is needed.

Whilst Vivado offers a GUI to build block designs, this process can be a bit frustrating to the user due to various "excentricities" of the application. Therefore, we choose to use its Tcl API to write scripts that create a new project and apply and connect all necessary blocks. This avoids a lot of errors as a bug in Vivado's user interface won't tamper with the project. It also enables us to use version control tools for the project as Vivado projects create a lot of files which oftentimes clash in very simple versioning operations. Using Tcl scripts which create and configure the project, and leaving everything else out of the repository, avoids this hassle. Tcl also allows the creation of subblocks: One can group blocks together and insert them multiple times with little effort; a feature which Vivado's graphical front-end apparently does not offer. More on the Tcl API and Tcl itself can be found in [17], [18], and [19].

The final advantage of the Tcl API to be mentioned here is that it allows the creation of the entire project, the block design, perform the synthesis and implementation, build a bitstream, as well as the board support apckage and first stage bootloader, all in a single sequence of automated tasks without the need for manual intervention. Since this tends to take quite a lot of time, that is a significant advantage.

4.2 The ADC Core

The ADC core is a simple piece of logic that interfaces with the FPGA pins which are connected to the STEMlab's ADC. It reads the ADC's unsigned 14 bit values and converts them to 16 bit signed format by adding an offset of 2^{13} and performing a 2 bit sign extension. The resulting numbers are then provided over an AXI Stream bus interface, which is also used by all the filters. The core is used from the git repository provided by Pavel Demin [11] More on his repository and project can be read in Section 2.3.1.

4.3 The Logger Core

The logger core (logger in further text) is a piece of VHDL code that stores samples it gets from a source in a ringbuffer in RAM. It is packaged as a Vivado IP core an can be seemlessly integrated into the project. The logger originated from an earlier project.In addition to logging data to RAM, it can also be programmed with various triggers. It reads instructions from a BRAM on the FPGA and iterates over them. Having reached the last one, it issues an IRQ signal, signalizing the end of the transcription.

The logger's original implementation features eight channels with a width of 14 bit, padded to 16 bit in order to simplify data transmission in B-sized chunks. Each two channels require one clock cycle to store a sample.

In order to take advantage of the fact that additional bits can be "won" by oversampling a signal, this projects implements a new configuration, which can process full 16 bit values, a gain in two bits over the ADC's output. The penalty for this is one additional clock cycle of delay, since the adders and comparators cannot match the timing requirements with two additional carries. Since this project aims to optimize for lower-frequency signals, the resulting additional delay of 8 ns is acceptable and will not be an issue in practice.

The logger core comes with a kernel module that provides a convenient interface from the ARM core. This avoids having to manually program the logger core. The programming

[1] Idempotence [...] is the property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application [16].

of a trigger and polling of the logger's status is explained in the Developer's Guide in Section TODO.

4.4 The Filter Chains

The filter chains are the most crucial part of the project and also the most delicate one as simple mistakes can cost several decibels of SNR and make the signal leaving the filter worse than it came into the filter, intead of improving it. The logical structure of the chains can be seen in Figure 3.1 and the rationale behind it is explained in the respective Section 3.3. For the detailled implementation, it is advised to look at the project in Vivado itself, as the block design is impractically large to be put onto paper, thus it is omitted in this report.

This section first gives a few notes on the two most important building blocks in the FPGA design: The CIC and FIR compilers by Xilinx. After that, two key points which are particularly challenging when implementing filter chains are elaborated upon: Propagating the correct bits through the cascade, and adjusting the gain correctly in order to exploit the available bits for maximum dynamic range.

4.4.1 Filter Compilers

The basic building blocks of the filter chains are FIR and CIC filters, which are based on ready-to-use blocks by Xilinx. Vivado's CIC and FIR compilers natively utilize the DSP slices to a maximum extent and make it very easy (at least in theory) to implement a Matlab-designed filter in hardware. Those IPs are described very thoroughly in the official documentation [14], [13].

The FIR filter compilers are configured using a set of coefficients in `double` format (exported from Matlab, or any other filter design tool of choice). The compiler quantizes the coefficients with maximum precision using a 16 bit fixed point number (this can be changed from the default but should be left to the compiler for best results). It does so by determining the index of the MSB required to represent the biggest coefficient in the set using 16 bits downwards.

As an example, take the biggest coefficient to be $c_{\max} = 0.23$. The bit at index -2[2] becomes the sign as its value (0.25) is not needed to display $c_{\max}$. Thus the number is said to have 16bits overall and 17 fractional bits. While this might seem a bit counterintuitive at first, it simply means that the LSB is the one at index −17 and it has 16 bits, meaning the sign is at the bit index −2.

The compiler then also takes the specified input bit configuration and determines the required output configuration to guarantee no overflows and achieve maximum performance. What is important here is that the output bit width is the same that is needed to guarantee no overflows inside the filter. This means that the user has to be aware of the maximum gain of the designed filter and determine on their own which bits are important at the output.

4.4.2 Bit Propagation Through the Filter Chains

For this application, only 16 bit values are stored. However, the filters generate far wider numbers at their outputs. This means that many of them are discarded at the output. To make sure that no important bits are truncated, the chosen input format for the filters is `17.7`, resulting in 24 total bits. The MSB should always remain just a sign extend of the

[2] The bit at index $-n$ is the 2^{-n} valued bit.

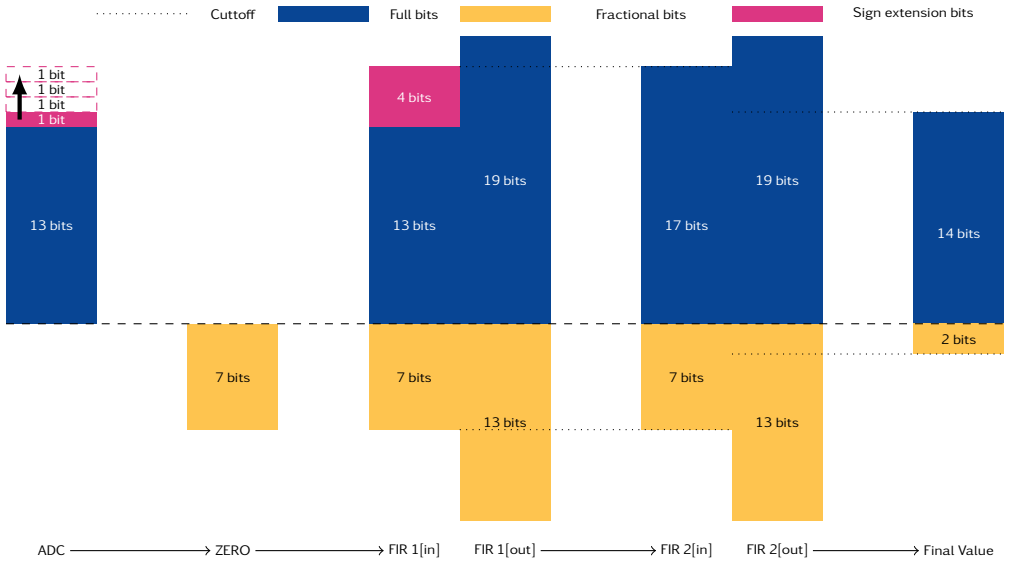


Figure 4.2: The flow of the bits in a filter chain. The bits are shifted through horizontally. Every bit which does not fit into the numerical width of the next stage will be cut off, starting with those furthest away from the sign. At the sign extend the bit is replicated 4 times and handed through to the next stage. The ZERO stage simply holds 7 `'0'` bits to pad the number coming from the ADC on its lower end before it goes into the first filter. Inside the filter the bits can grow, so those are not shifted and cut off but rather resized towards the output.

sign actually residing at bit 15. The 7 fractional bits are cut off at the end of the filter chains but are still important for more precision so less rounding and/or truncation errors are introduced inside the filter chain.

As values can use up significantly more than 24 bits inside the filter due to bit growth, it has to be ensured that no overflows happen, resulting in greater bitwidths at the output of the filter. It is important that the location of the decimal point is always tracked and remains in its right place. Figure 4.2 depicts the flow through an example chain but represents the general case in out blockdesign.

4.4.3 Ensuring Maximum Dynamic Range

Of course it is important not to discard any MSBs (or signs) because otherwise the signal (a sine wave in our example) will clip or, even worse, overflow and wrap around. The same applies to the case where too few bits are cut off. If the bit count is increased by one to guarantee no overflow inside the filter, but that bit is not set at the output due to unity gain, it will effectively be lost as it will never be used. This reduces SNR by 6 dB, which is obviously highly undesirable.

To make sure neither of these faults happens, it is important to have the highest possible filter gain at $G \leq 1$. Furthermore, at the end of each filter chain, the additional bits must not be carried over, but rather the initial 14 bits before the decimal point, along with two fractional bits after the decimal point. This yields the desired 16 bit value and ensure no "empty" bits.

The FIR compiler can avoid those empty bits by normalizing the coefficient such that

the highest gain (i.e. the top peaks of its passband ripple) is at exactly 1. This is called *maximizing the dynamic range*. Figure 4.3 illustrates the issue of loosing one bit.

One can observe that the sine in Case 1 (top plot) uses the dynamic range to a perfect extent as the full-scale sine has an amplitude of 31, the maximum value an 6 bit `int` can hold. Case 2 (middle plot) shows a sine that has been scaled to 34 and thus requires an additional bit. But because a 7 bit `int` can hold values up to 63, most of the time the MSB ends up not being used. with 7 bit, the highest $SNR_{\max}$ possible would be

$$\begin{aligned} \text{ENOB} &= \log_2(34) = 5.09 \\ \text{SNR}_{\max} &= 1.76\,\text{dB} + \text{ENOB} \cdot 6.02\,\text{dB} = 32.39 \end{aligned} \tag{4.1}$$

In Case 3, the dynamic range is used well and the $\text{SNR}_{\max}$ with a 6 bit `int` is

$$\begin{aligned} \text{ENOB} &= \log_2(30) = 4.91 \\ \text{SNR}_{\max} &= 1.76\,\text{dB} + \text{ENOB} \cdot 6.02\,\text{dB} = 31.30 \end{aligned} \tag{4.2}$$

This yields a difference of only 1.09 dB[3]. This example shows that it is well advised to scale the coefficient such that they dont ripple around 1, but rather that the maximum ripple is exacly 1 and not more.

Because if it can be ensured that no additional MSB is used wich is empty most of the time, it is possible use an additional LSB which is always well used and to effectively win a bit. So in most cases when the coefficients are designed to have a unity gain, close to 6.02 dB can be won by rescaling the coefficients.

4.4.4 Errors Due to Truncation in the CIC Filter

As detailled in Section 1.2.3.4, the high gain of CIC filters generally requires discarding bits at the filter's output. In our implementation, we discard TODO bits at the filter's input, and TODO bits at its output. According to the calculations laid out in Section 1.2.3.4, this yields a mean error and variance of:

$$\mu_{\text{CIC25}} = 0.5 \tag{4.3}$$

$$\sigma^2_{\text{CIC25}} = \tag{4.4}$$

$$\mu_{\text{CIC125}} = 0.5 \tag{4.5}$$

$$\sigma^2_{\text{CIC125}} = \tag{4.6}$$

for the two CIC filters used in our configuration. The calculations are performed by a Matlab script and are therefore omitted at this point.

[3] While 1.09 dB might still seem rather large, keep in mind that with the wider numbers running in the actual filter chains, the result is significantly better than in this illustrative example.

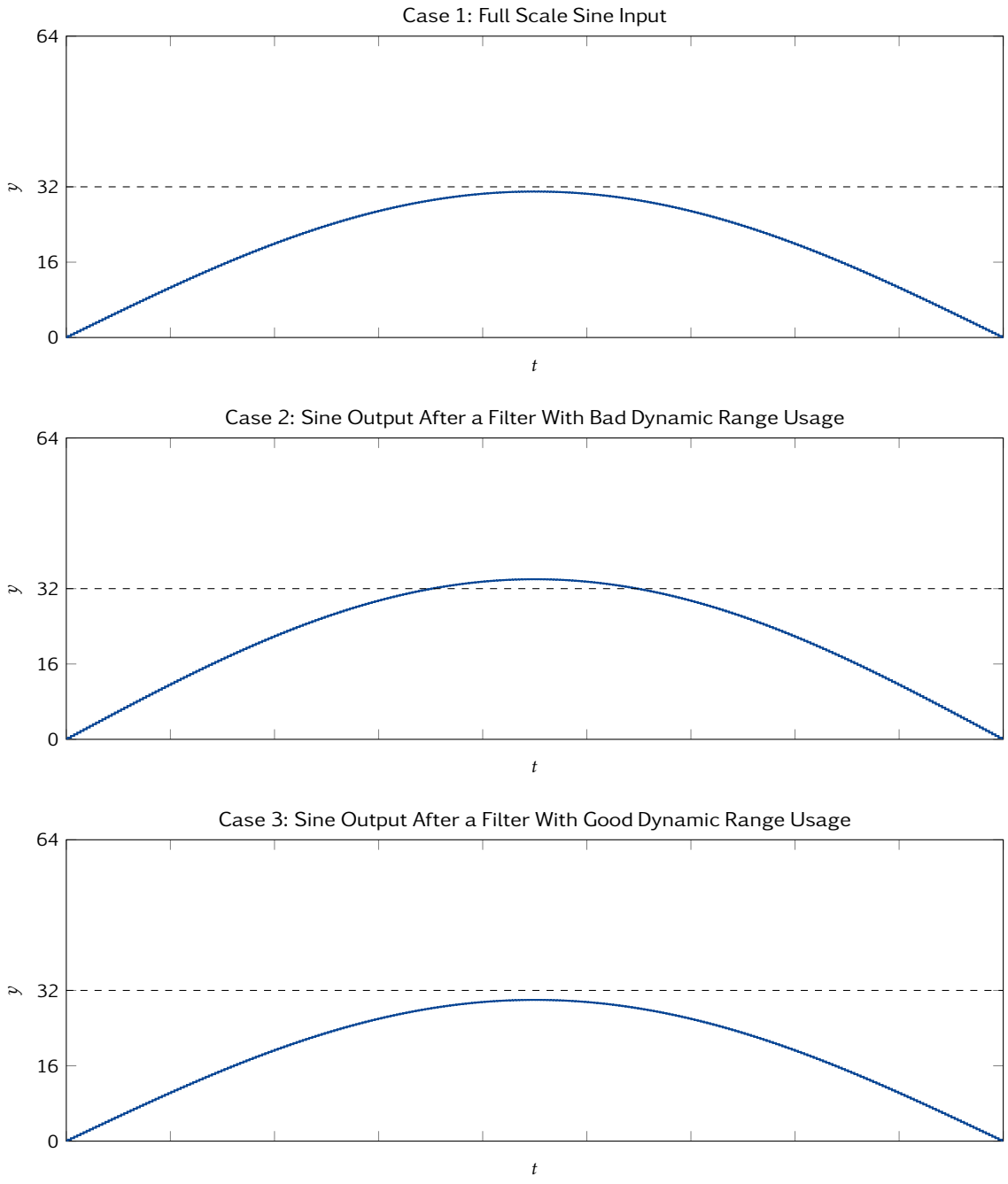


Figure 4.3: An illustration of good and bad use of dynamic range.

CHAPTER 5

Server

Once the FPGA has recorded data, that data has to be transmitted over the network. Since implementing networking in hardware is not feasible in most cases, this is done via the ARM Cortex A9 core that already has a PHY[1]. To control all the hardware of the SoC, an embedded Ubuntu GNU/Linux is running on the ARM core which can control all hardware components, including the logger running on the FPGA. An application is then needed that reads the necessary data from the RAM and sends it to the network. This part of the overall product is designated as the *server*. This section explains the design choices and internal structure of the server application.

5.1 Requirements

The basic functional requirements of the server application are:

- Read the system status and transmit it over the network.
- Receive commands over the network, translate them where needed and relay them to the FPGA IP.
- Read data from the RAM and transmit it over the network.

5.2 Design Choices

As the ZYNQ Logger comes with a kernel module that has to be interfaced via `IOCTL` calls, it is recommended to write the application in C or C++. This is due to the nature of Linux, which still requires mostly C for interfacing. There are some `IOCTL` interfaces in Python and Rust, for example, but those bring additional problems on ARM Linux since not all libraries and features are available.

Since the server application is rather complex, C++ is a good choice, eliminating some of the caveats which C has. Additionally, the entire feature set of C can be used in C++ as well, so the choice carries no penalties in terms of features.

The WebSockets protocol is mandatory, due to the choice of JavaScript on the client side. This is less of a design choice on the server side, and more of an inherited requirement from

[1] A chip implementing the physical layer of the OSI model

the front-end part of the project. For this purpose, the uWebSockets (*uWS*) library is chosen. It carries a very small footprint and offers good performance, though documentation is somewhat lacking. Its performance is high enough to ensure that in any given scenario, the server application will not be the bottleneck of the overall data pipeline; the network connection will choke before uWS reaches the limits of its capabilities.

uWS is based on epoll, libuv or boost::asio depending on the user's choice. All of those are asynchronous (more on this topic can be read in [20]) libraries which makes networking very conventient. uWS comes with callbacks that can be registered for each WebSocket event (see Section **??**). Furthermore the user can hook into the event loop and register other events such as a reocurring timeout (timer).

JSON is used as a data format for settings and statistics. It is our format of choice because it has by far the largest user base of all the available formats, the specification is simple and JavaScript can parse it natively into a JavaScript object. This also another point where the choice of C++ offers significant benefits over C, since it has a number of high level (i.e. easy-to-use) libraries that can serialize and, more importantly, deserialize JSON objects.

5.3 Implementation

While the server's overall task is complex, its implementation has been kept as simple as possible. Indeed, the application fits into a single file (not counting the loaded libraries) of a few hundred lines, and consists mainly of a single asynchronous event loop.

The overall application state is stored in a struct. This allows any event to access any information it might need, such as the socket handle, the current sampling rate, or the requested number of bytes. Most of the application's functionality can be deduced from its event loop, which is shown in Figure 5.1.

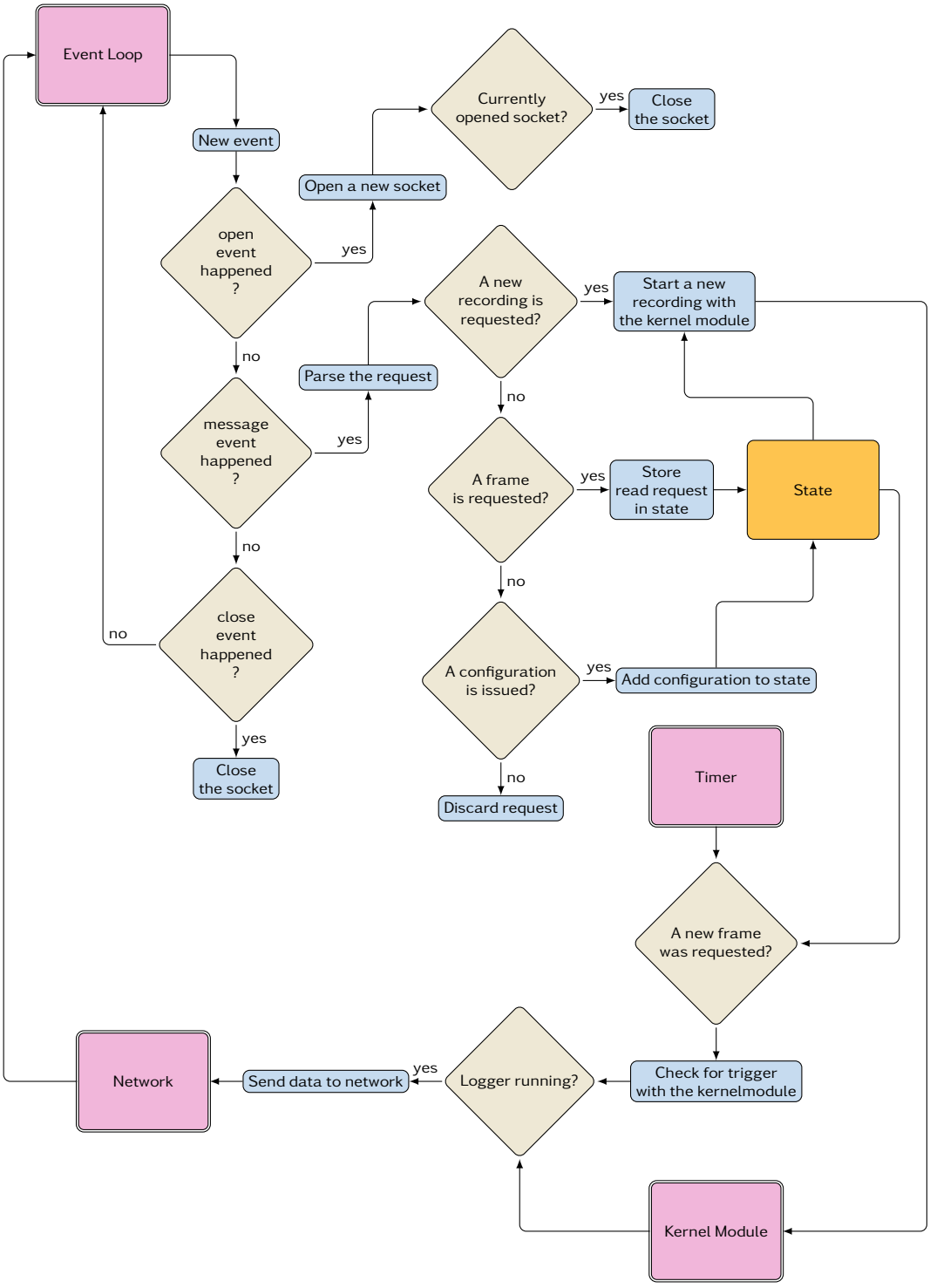


Figure 5.1: The server's event structure.

CHAPTER 6

Oscilloscope

Because the FPGA and Linux side are newly implemented, and because the new transmission protocol is not the same as the old one, existing applications reading data from the STEMlab no longer function. A new front-end is therefore required. This front-end is a web application whose functionality is broadly modelled on conventional oscilloscopes; therefore, it is generally referred to as *scope* in this report. This chapter lays out the requirements for the scope, explains the design choices on which it is built, summarizes the major implementation details, and presents the final product.

6.1 Requirements

The requirements for the scope are partially defined by a Java application from previous projects [21] whose core capabilities are to be replicated. More functionality may be added where sensible and possible. The main requirements are:

- Receive data in configurable size over the network.
- Display received data both in the time and fequency domain.
- Calculate RMS power density in the signal.
- Calculate THD of the signal.

6.2 Design Choices

There is a wealth of programming languages to choose from, with countless libraries to go with them. The following sections explain why JavaScript and web technologies are used to implement the graphical user interface (GUI) and the mathematical functions of the scope.

for this purpose, a group of programming languages are compared and weighed against each other in various aspects; the results are summarized in table 6.1. the general attributes listed in table 6.1 are explained in the subsequent paragraphs to enable the reader to understand how they apply to our decision. Due to its importance, there is an additional section dedicated to the topic of networking (Section 6.2.1).

Open Standard: Since this is a university project meant, among other things, for educational purposes, it is crucial to make all source code available to the public under a flexible license. Thus, it is desirable to use a technology which is independent of any one company

and their corporate policies (avoiding vendor lock-in). Some programming languages are managed openly and accept contributions from the public, others not.

Networking: The two criteria which the data transfer needs to fulfill are speed and data integrigy. Since networking is a highly complex topic, it is important that the language not only has libraries for good networking protocols, but that those libraries are also easy to use. More information on the topic is contained in Section 6.2.1.

Graphics: An oscilloscope is quite a demanding application when it comes to graphics; drawing an image stream which looks fluid to the human eye on modern high resolution display uses a lot of processing power. Using an interface such as OpenGL which can utilize dedicated graphics resources on a computer is therefore necessary.

Additionally, creating a sensible user interface with basic drawing commands such as rectangles and lines is impractical. Instead, a GUI toolkit to speed up the design process is needed.

Prevalence: Using a technology which is widespread makes it more likely that good tutorials are available. It also facilitates troubleshooting, since a larger user base means that there is a higher chance of savvy users being able to provide support if needed.

Ease of Development: Some solutions require large IDEs and unwieldy toolchains for development. Others can be used with a more lightweight setup. Since both team members come from a Linux background, the latter is preferred. Since added complexity always also means added probability of errors and failures, using leaner tools also decreases the chances of having to fight with the development tools instead of tackling the actual challenges which are to be solved.

Ease of Deployment: This takes into account how easy or difficult it is for an end-user to install the scope and get it up and running. Having a toolchain which allows the effortless creation of stable binaries is important here.

Familiarity With The Language: The best toolkits do not matter if none of the involved programmers have ever used them and will struggle with even the basics for a major part of the project's duration. Thus it is inevitable that personal preferences also flow into the decision process.

Summary: As the total scores in Table 6.1 show, JavaScript fits the priorities set in this project best. As a scripting language which can be run in any modern browser, a website can be provided by the STEMlab board and accessed from a client via a browser. Since no special programs need to be installed on the client computer, and every major operating system today has at least one reasonably modern browser, this makes the solution both highly portable and easy to deploy from an end-user perspective.

JavaScript's popularity ensures that there is no danger of the underlying technology of the scope becoming obsolete any time soon. Furthermore, with its support for WebGL and the WebSockets protocol, it provides two performant and easy-to-use technologies to implement graphics and networking, respectively. Its primary downside is a heavy memory footprint, but since that is rarely a concern on modern computers, it is not considered a relevant factor for our decision.

Table 6.1: A comparison of a few programming languages which might be used to implement a graphical front-end like the oscilloscope from this project. The scale goes from 1 (worst) to 6 (best).

	Rust	C++	Java	Python	JavaScript
Open Standard	6	6	1	6	6
Networking	6	6	6	6	4
Graphics	2	5	5	5	6
Prevalence	3	6	6	5	6
Ease of Development	5	5	5	5	6
Ease of Deployment	3	4	5	6	6
Familiarity With the Language	3	3	4	6	6
Total	28	35	32	39	40

6.2.1 Networking

For networking, two primary protocols are available: UDP and TCP. To ensure a fluid stream of data, minimizing protocol overhead is key. In situations where data integrity is not essential, UDP is generally used. It carries no overhead for guaranteeing completeness and correct order of packages.

If data integrity is vital, TCP is generally the protocol of choice. It has mechanisms for guaranteeing both the completeness and correct order of packages. This comes at the cost of some overhead, but in most applications, this is negligible and well worth the cost. Another key feature of TCP is that it can perform congestion control. TCP will sent no more packages if previous packages have gone missing (i.e. TODO: if their reception has not been confirmed?). Where UDP will happily flood the network with as much data as it is fed, TCP ensures that the network is not flooded. The result is that in case of a bad or slow network connection, the amount of transmitted data is automatically adapted to the network, and only as much data is sent as the client can actually receive and process.

When deciding how to deploy TCP, one can choose to implement one's own subprotocol, or use one of the existing two: HTTP or WebSockets. The WebSockets subprotocol, being intended for data streaming and having mature JavaScript support, fits the requirements of our application perfectly. Some additional notes on WebSockets can be found in Appendix C.1 on page C.1.

6.3 Product

The oscilloscope is a web application that can be directly loaded from the server running on the STEMlab. It has the following capabilities in its current implementation:

- Receive data over the network for two channels (this is only limited by the physical channels of the STEMlab).
- Manage triggering set a trigger type, level and the number of samples that have to be recorded before and after the trigger is activated.
- Calculate and display the power density spectrum.
- Calculate the SNR both automatically detecting the signal and manually being told where the signal is.
- Calculate the THD for a given base harmonic. TODO: !

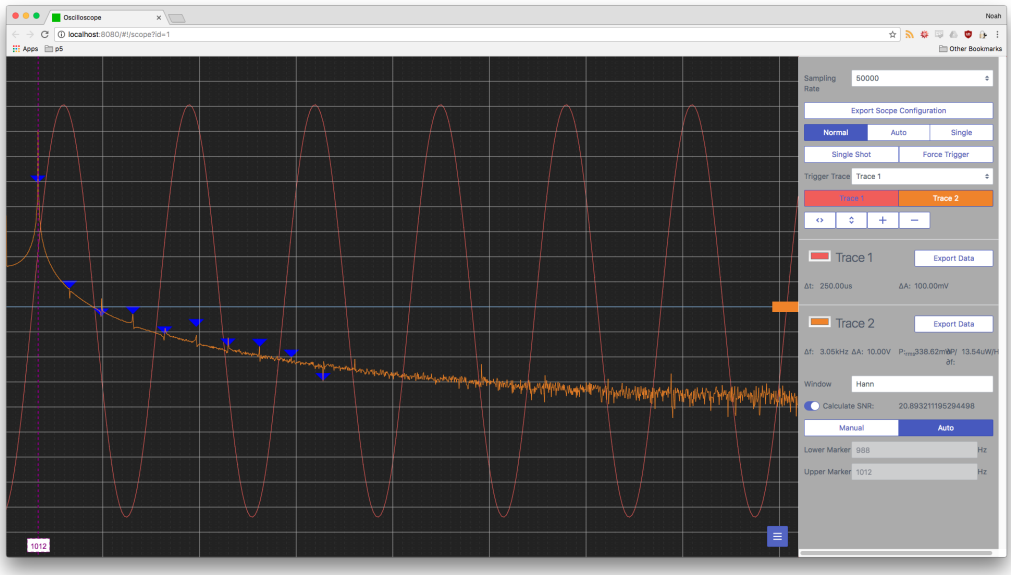


Figure 6.1: The scope application in it's current state, displaying time and FFT data.

- Export data to an array-string.
- Export and load the scope configuration to and from JSON strings.

The following pages explain how the application is structured, along with the implementation of some of its key features. A screenshot in Figure 6.1 shows the overall layout and design of the scope.

6.3.1 Application Structure

The entire application consists of a single state tree. This makes it very easy to import and export settings and presents a better overview of the application state than scattering state information across various objects. Listing C.2 in Appendix C.2 on page 105 shows an extract of the tree structure. Like any JavaScript application, the oscilloscope runs asynchronously. Its eventloop structure is shown in Figure 6.2.

All of the values that can be controlled through the GUI – and many more – are also controllable directly through the state tree. Upon initialization of the application, the entire state tree is loaded and references to parts of it are passed to the controller objects. The structure of the application is hierarchical; the most important relations are depicted in Figure 6.3. In the following some of the more important prototypes and functions from Figure 6.3 are elaborated upon.

The `Oscilloscope` prototype is the top level controller which contains exactly one source. It is responsible for handling all mouse events and reacting accordingly, such as moving the trigger level or zooming and panning. The Oscilloscope draw call is responsible for drawing general information which is not part of a specific trace onto the canvas. `Oscilloscope` is also the caller for the `Trace draw` call. `Oscilloscope` manages general information and is responsible for rescaling the canvas and initiating the draw call chain.

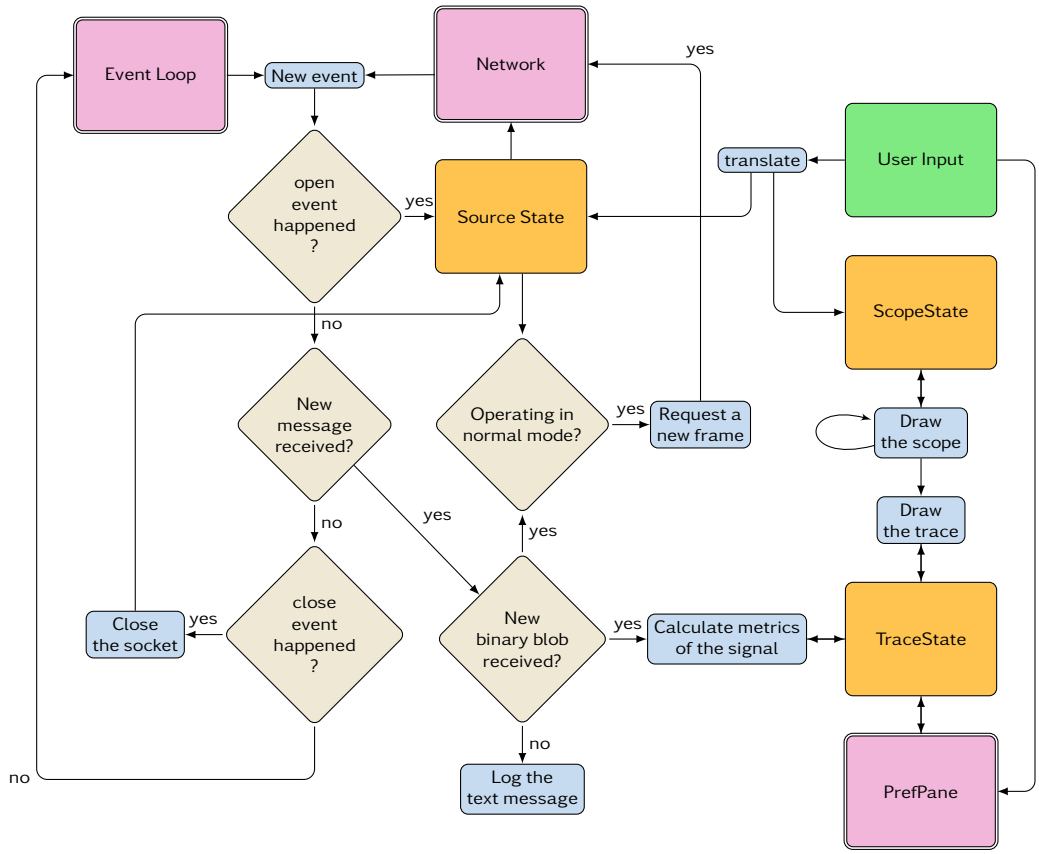


Figure 6.2: The scope's event structure

The `Source` controller manages all calls to and from the server. It contains a lot of helper calls to set a trigger or issue a new frame. Those helpers are called by the `Oscilloscope` controller or GUI elements. It also contains the important callbacks to send and receive data from the server. They are explained in more detail in Appendix C.1.

`Source` stores all received frames in itself. The frames will then later be copied and processed by the trace controller. Each frame received is always overwritten by the next one, preventing memory leaks and ensuring that the data is current.

Once the `Source` controller receives new data, it starts the `calc()` call for each trace to trigger an update of the trace with new data.

The `Trace` prototype is in charge of drawing and calculating traces. It has two derived prototypes, `Timetrace` and `FFTrace`. While the `Timetrace` prototype just returns untouched data to display the time domain of a signal, the `FFTrace` prototype calculates some signal metrics before displaying its results, i.e. spectral power density and SNR. The structure of the application allows for easy extension with new trace types with arbitrary math functions (e.g. differential pair subtraction for noise cancelling).

One instance of a `PrefPane` is created for each trace. A `PrefPane` component is a mithril component that creates a vnode that exposes all the necessary controls for its corresponding trace in the GUI. It is also responsible for displaying processed data for a trace such

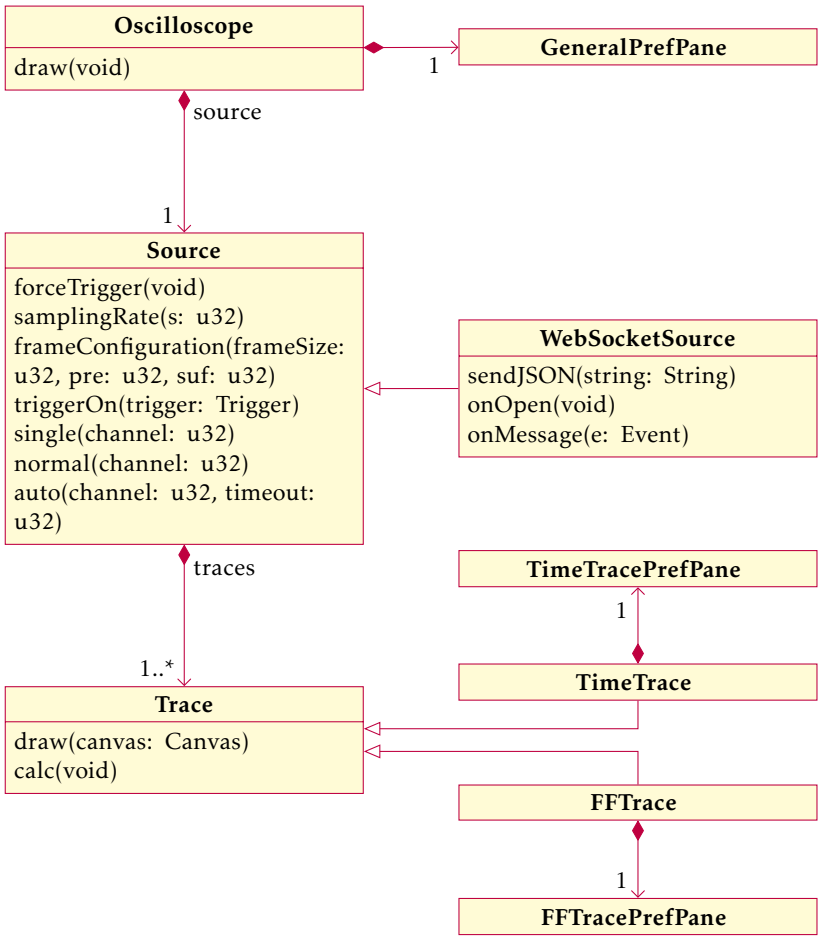


Figure 6.3: The structure of the scope application with its major relations

as the SNR for an `FFTrace`. There is also a general `PrefPane` that exposes common controls such as switching modes, the trigger trace or the active trace. The `PrefPanes` are built with `mithril.js`. Appendix C.3 on page 107 provides some general information on `mithril.js` and example code relevant to our application.

6.3.2 Graphics

Having a robust graphics layer is paramount to providing a user-friendly interface. It must process data quickly for smooth visualization, display signal metrics, and offer sensible controls for the user to interact with. Taking advantage of a GPU in order to speed up data processing and rendering is highly desirable. Due to the open-source nature of this project, using a technology based on OpenGL makes sense; JavaScript offers WebGL for this purpose.

With JavaScript and HTML there come a large variety of libraries that enable the easy creation of graphical applications, and CSS offers great flexibility for styling the resulting structure. `mithril.js` has been chosen to build the controls of the scope. It offers an exceptionally low footprint and high speed when recalculating the DOM. Thus, the user

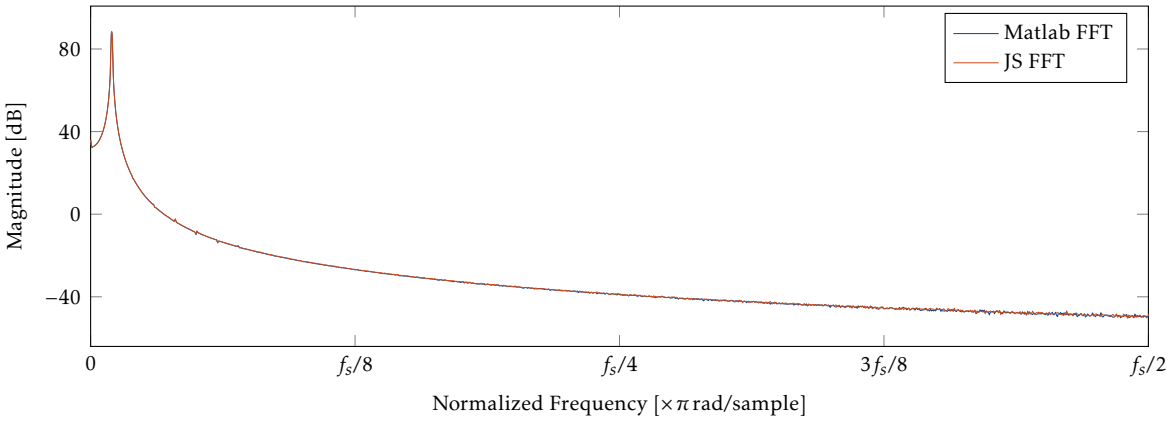


Figure 6.4: The used JS FFT compared to the FFT of Matlab.

need not download a heavy library into their browser's cache, and the interface updates smoothly and without lag.

For data plotting, existing libraries such as `plotly.js` or `chart.js` could be used. While they provide a lot of built-in functionality like logarithmic plots or automated axis labeling, they also carry significant overhead and lack the performance required for real-time data plotting at high data rates. Therefore, our application draws directly onto an HTML canvas via WebGL draw calls. Some more general information about WebGL and its usage is provided in Appendix C.4 on pages 107f.

6.3.3 Power Calculation

Calculating the spectral power of a signal is accomplished via the FFT algorithm first presented by Cooley and Tukey in 1965 [22]. Kevin Kwok has written a compact and fast implementation of this algorithm in JavaScript [23], which is provided under a permissive license [24] and therefore nicely suits the requirements of this project.

The following equations detail the calculations. For an in-depth explanation, see [25]. The two-sided spectrum of an input signal x is obtained with

$$Y[i] = \frac{1}{N} FFT\{x[i]w[i]\}. \tag{6.1}$$

From that, the one-sided power spectrum can be calculated via

$$\begin{aligned} P_{yy}[0] &= \frac{Y[0] \cdot Y[0]^*}{NG} \\ P_{yy}[i] &= 2 \cdot \frac{Y[i] \cdot Y[i]^*}{NG} \quad \text{for} \quad 0 < i \leq N/2. \end{aligned} \tag{6.2}$$

Where NG is the noise gain, and depends on the window being used. Values for common cases are listed in Table 6.2. Finally, the power of a spectral region between two frequencies can be found by integrating over that range of frequencies, or in the discrete case like here, by summing all the bins:

$$P_{1,2} = \int_{f1}^{f2} P'_{yy}(f)df \approx \sum_{i_1}^{i_2} P_{yy}[i] \tag{6.3}$$

Table 6.2: Correction factors for the different window types used in the scope application as seen in [25].

Window	CG	NG
Rectangular	1	1
Hamming	0.54	0.3974
Hanning	0.5	0.375
Blackman-Harris	0.3587	0.258
Flat Top	0.2156	0.1752

6.3.4 SNR Autodetection

Calculating the signa-to-noise ratio requires defining which components of captured data are signal, and which parts are noise. The power contained in each frequency range can then be computed with Equation 6.3, and the SNR calculated as the following ratio:

$$SNR = 10\log_{10}(\frac{P_s}{P_n}) \tag{6.4}$$

Automatically determining which parts of a spectrum constitute the signal and which parts are noise is the purpose of SNR autodetection. This requires the detection of the actual signal components as well as additional spectral lines introduced in the FFT process. The phenomen of these spurious spectral components is commonly known as *spectral leakage*. Additionally, one may wish to remove any spurious harmonics originating from the signal source (e.g. a function generator which does not generate a flawless sine wave).

Depending on how the signal is windowed for the FFT, different spectral lines will occur with varying magnitudes. Table C.1 in Appendix C.5 contains the primary characteristics for some commonly used windows. The *Number of Lines per Bundle* entry from that table can be used for a static approach to SNR autodetection:

- Find the peak in the spectrum.
- Presume that peak and $\frac{n}{2}$ spectral lines to its left and right to be the actual signal, where n corresponds to the *Number of Lines per Bundle* column in Table C.1.
- Regard the rest of the spectrum, except any potential DC offset and the next $\frac{n}{2}$ spectral lines above it, as noise.
- Integrate the power over the defined frequency ranges and calculate SNR according to Equation 6.4.

However, this method does not give optimal results. An example data stream similar to the one in Figure 6.5 processed by Matlab's `snr()` function yields about 78 dB. Calculating SNR with the static method as outlined above results in only about 33 dB for such a signal. Even accounting for Matlab's harmonics cancellation, a result no better than 43 dB is achieved (10 harmonics cancelled out).

Improving this SNR detection method can be accomplished by using an iterative algorithm. The number of lines below the peak lobe which are still considered signal is increased until the SNR no longer drastically changes. In our implementation, this is defined as a change no greater than 0.05 dB. Algorithm 1 depicts this process systematically. It can be repeated for any number of spurious harmonics to cancel them out as well. The two methods are compared in Figure 6.5. The example also shows highlighted harmonics for eliminiation in the SNR calculations. The result is in line with Matlab's.

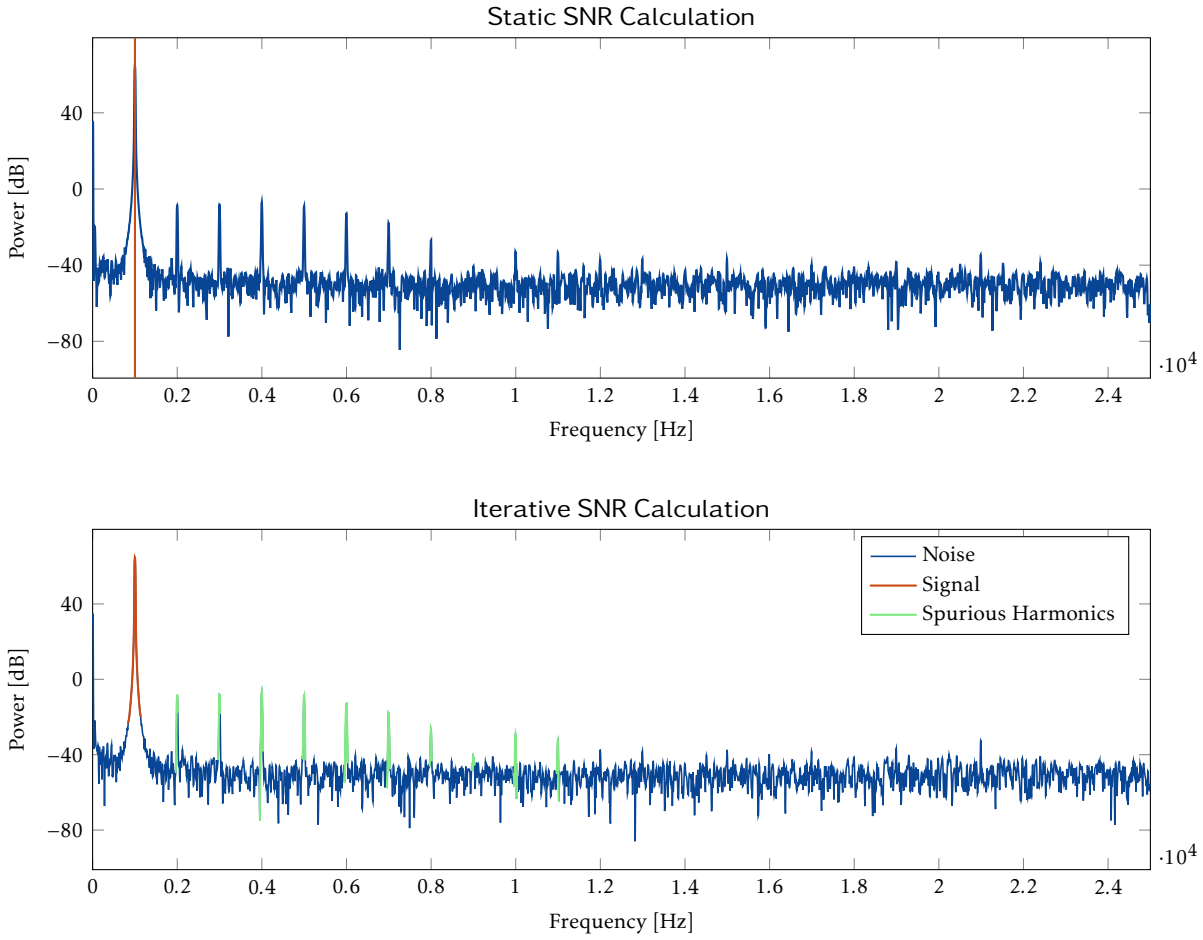


Figure 6.5: Comparison of methods to determine lobe width for SNR calculation. The top plot shows the static method. As can be seen, only a very narrow section of the peak lobe is taken to be the actual signal, while most of the surface beneath it is considered noise.
In the bottom plot, the iterative method method to determine the width of the lobe components considered to be signal has been used. As is easily visible, the resulting lobe region is significantly wider. Additionally, spurious harmonics from the signal source have been highlighted with the same process, allowing for their elimination in the SNR calculations.

Algorithm 1 Algorithm to iteratively determine the SNR of a spectrum

$l \leftarrow 1$
$max_i \leftarrow 0$
$SNR_p \leftarrow 0$
$SNR_n \leftarrow 0$
while $|SNR_p - nextSNR| > 0.05$ **do**
    $SNR_p \leftarrow SNR_n$
    $Ps \leftarrow power(spectrum[max_i - l : max_i + l + 1])$
    $Pn \leftarrow power([l : max_i - l] + power(spectrum[max_i + l + 1 :]$
    $SNR_n \leftarrow 10 \cdot log_{10}\left(\frac{P_s}{P_n}\right)$
    $l \leftarrow l + 1$
end while
$SNR \leftarrow SNR_p$

CHAPTER 7

Verification

CHAPTER 8

Conclusions

Part II

Developer Guide

CHAPTER 9

Project Structure

repo structure: TLDs and what is contained in them.
"Where do I have to go if I want to do X?"

CHAPTER 10

FPGA/SoC Toolchain

Because the FPGA/SoC requires a lot of different utilities and environments, it is advisable to have a fixed development environment as hardware tooling often breaks at the slightest change. For this reason we provide a build box[1] which contains an Ubuntu Linux. To set up the build box, Vagrant and ansible are required. The former pulls the base Linux box from a global repository at HashiCorp[2]. ansible is then used to provision the box to install all the necessary tooling. Once that is set up, the user has to perform a graphical install of the Vivado toolchain.

This chapter describes how to set up the build box and Vivado, and how to generate a Linux image which can be flashed onto the STEMlab's SD card to get the device up and running.

Note: The following indicates a code snippet which has to be entered on the command line:

```
enter commands here
```

10.1 Setting Up the Build Box

The following prerequisites need to be installed onto the host machine first:

- Vagrant
- ansible
- VirtualBox

If you are on a Linux or OSX and use a package manager to install VirtualBox. It is also necessary to install the Virtualbox Guest Additions packages. They are required in later stages during the setup of the box. It is also advisible to perform a

```
sudo vboxreload
```

Otherwise a reboot of the host system might be necessary.

[1] A virtual machine image for the purposes of development.
[2] The corporation behind Vagrant

After all the tools on the host have been installed, we move to the root of the project repository. For all the further steps we assume that we operate from that root directory.

To do an initial setup of the box, enter

```
cd env
vagrant up
```

This will boot the box and provision it using ansible. This step requires a stable internet connection. If anything fails during the initial setup (red messages in the shell), you can run and retry the provisioning with

```
vagrant provision
```

Once the provisioning has finished, the build box should be rebootet because some changes (for example the installed desktop environment) require a reboot. Do this by running

```
vagrant halt
vagrant up
```

If you want to make changes to the default box setup, have a look at the file `/env/roles/common/tasks/main.yml` and the official ansible documentation [26].

The fully configured build box should contain two shared folders: `/vagrant/` points to the `/env/` directory on the host, and `/repo/` points to `/` on the host.

IMPORTANT: The password of the default vagrant user is **`vagrant`** and has `sudo` privileges.

10.2 Setting up Vivado

After having completed the basic build box setup as outlined above, Vivado needs to be installed. For this Section, we assume that all commands are executed on the VirtualBox (*guest* for the rest of this manual). Enter

```
/repo/Xilinx/Downloads/Xilinx_Vivado_SDK_2016.2_0605_1_Lin64.bin
```

into a shell to start the Vivado install. The graphical installer will guide through the process. A complete pictorial guide for this is beyond the scope of this document, but an illustrated guide with screenshots for all steps is available under [27]. For documentation on Vivado itself, it is recommended to have a look at the documentation portal from Xilinx [28]; it contains extensive documentation on many topics.

10.3 Building a Linux

With the fully set up build box it is possible to build an image with only two commands. While it is possible to only perform the build steps once changes have been made to the FPGA project or the server application, it is advisable to perform an initial build. This allows to check if everything is set up correctly and works as intended.

First we copy the repo to a **non-shared** folder on the guest machine, because uboot requires `mmap()`, which cannot handle shared folders. After that, building the image is a

single-step process. This includes building the Linux, the bitstream, the required firststage bootloader and the board support package, the logger kernel module, the server application, and the scope application. After that, a bash script is used to create an image with all the components, mount it and provision the ubuntu environment which is to be run on the STEMlab. To start this process, enter

```
cp -r /repo ~/local_folder
cd ~/local_folder
make init
```

After the build process has finished, the image can be created using

```
cd ~/local_folder/firmware/arm
sudo sh scripts/image.sh scripts/ubuntu.sh red-pitaya-ubuntu.img 1024
```

These scripts were initially created by the Red Pitaya corporation, altered by Pavel Demin [11], and have again been adapted to the requirements of this project.

CHAPTER 11

Filter Toolchain

Designing the filters according to the desired specifications (see Chapter 3) is performed through a set of Matlab scripts. This chapter describes the overall structure of the script suite and gives a basic usage example. All functions have a `help` available which describes their usage, particularly their respective interfaces.

11.1 Toolchain Structure

This section explains which files constitute the toolchain and what their purpose is. The entire toolchain can be used either from Matlab's graphical front-end, or from its commandline mode. The following files are present in the filter design directory[1]:

```
design/filter/
├── cliDispatcher.m
├── guiWrapper.m
├── generators/
│   ├── decCIC.m
│   ├── decFIR.m
│   ├── halfbandFIR.m
│   ├── compCIC.m
│   ├── cascador.m
│   ├── parcascador.m
│   ├── pardecFIR.m
│   └── parhalfbandFIR.m
├── plotData
├── coefData
├── Makefile
└── README.md
```

`cliDispatcher.m` is the main script from where the design functions are initiated. If you wish to design a new filter chain, this is where its specifications are located, and from where the filter design scripts are then called with those paramters.

[1] relative to global project root

guiWrapper.m is a convenience layer which makes it easier to work with `cliDispatcher.m` from Matlab's graphical front-end. It sets some of the configuration parameters for calling `cliDispatcher.m` which are otherwise set in Matlab's commandline interface when calling the dispatcher from there.

generators/ is the directory where the filter design scripts are located. These are split into two primary groups: Functions which design filters, and functions which combine them into cascades. Note that all generators support generating multiple filters at once for a combination of various paramters and will pass back a cell array with the resulting filters and the paramters used in their specification.

This allows to iteratively generate a large set of filters in an initial step to assess resource usage or other characteristics for a wide range of parameters. The parameters for FIR filters are the ones laid out in Section 1.2.2. For half-band and CIC filters, they slightly differ. The filter design functions are:

decCIC.m designs a CIC filter.
decFIR.m designs a FIR filter.
halfbandFIR.m designs a halfband filter.
compCIC.m designs a compensator for a CIC filter.

Note that all of these have version which iterate in parallel over a given set of specifications, prefixed with `par` (e.g. `pardecFIR`). The interface for all parallel versions is identical. They can be used if Matlab's parallel processing toolbox is available. If only a small set of filters is to be designed, it is recommended to use the regular, serial versions, since starting up a parallel processing pool in Matlab is a slow process and its overhead is usually not worth it in those cases. In case Matlab is run in commandline mode, make sure *not* to start it with the `-nojvm` switch, since the parallel processing pool requires the Java Virtual Machine.

The functions `cascador.m` and `parcascador.m` create cascades of filter cell arrays passed to them. Note that in order to achieve the desired iteration result, some manual intervention in re-structuring the filter objects passed to the cascade functions might on occasion be needed, particularly in the case of cascading other filters with CIC filters. This is because the set of parameters used to design CIC filters is different from the set of coefficients used to design FIR filters, which means the objects passed back from the CIC and FIR filter functions might not always match as needed in their structure to cascade them in all possible manners. This is not an issue when only cascading single CIC filters with other filters.

coefData and plotData are two directories which are created by the functions (if they do not yet exist) for storing filter property data. `coefData` contains the filter coefficients for each filter which has coefficients in a Vivado-compliant `.coe` format. This allows a direct import of Matlab's results into Vivado's FIR compiler.

`plotData` contains frequency responses for each filter, as well as any potential cascade, in `.csv` format, as given by Matlab's `freqz()` function. This is used to generate the frequency response plots from this report, for examlpe.

The Makefile can be used to call `cliDispatcher.m` from the command line. It has various targets for designing different filters or filter chains. Matlab will be called in commandline mode, and its output redirected to a log file, so not output will generally be visible on screen. This is primarily intended to be used when all filters have been specified and fixed, and the needed files are to be generated for use on the FPGA side.

The `README.md` contains additional information which is beyond the scope of this documentation. In general, it is highly recommended to consult both the `README` and the `help` of the provided functions in case of questions, as well as the code itself, which is extensively commented (particularly `cliDispatcher.m`).

11.2 Usage

Now that we know the location and purpose of the main components, it is time for a basic usage example. For this, it is useful to know how to use the basic FIR filter design function (Listing 11.4), the CIC design function (Listing 11.6), and how to cascade filters (Listing 11.7). All of this code is to be put into the `cliDispatcher.m` file, and a `case` created for it. This is shown in Listing 11.1.

Listing 11.1: Using `cliDispatcher.m`

```
% cliDispatcher.m
genDir  = 'generators';
coefDir = 'coefData';
plotDir = 'plotData';
addpath(genDir);
switch filtertype
    % ... OTHER CASES ... %
    case 'EXAMPLE_FILTER_AND_OR_FILTER_CHAIN'
        clear all;close all;
        filtertype='EXAMPLE_FILTER_AND_OR_FILTER_CHAIN';
        genDir = 'generators';
        coefDir = 'coefData';
        plotDir = 'plotData';

        % ... filter design specifications and function calls ... %
end
```

To execute `cliDispatcher.m`, Matlab's commandline interface can be used:

Listing 11.2: Calling `cliDispatcher.m` from Matlab's Commandline Interface

```
>> filtertype='EXAMPLE_FILTER_AND_OR_FILTER_CHAIN';
>> cliDispatcher
```

Alternatively, one may create an entry in `guiWrapper.m` when using Matlab's graphical interface for added convenience. An entry can be created as show

Listing 11.3: Creating an Entry for `cliDispatcher.m` in `guiWrapper.m`

```
%% EXECUTE EXAMPLE_FILTER_AND_OR_FILTER_CHAIN
clear all;close all;clc;
filtertype = 'EXAMPLE_FILTER_AND_OR_FILTER_CHAIN';
disp('Designing Chain for R = EXAMPLE')
run cliDispatcher;
```

Besides knowing how to initiate the filter design toolbox, one must obviously also know how to actually design filtes. The code in Listing 11.4 designs *two* FIR filters, with two

different stopband edge frequencies (`Fst`). The resulting cell array `Hd` from Listing 11.5 has in its first column the designed filter system objects, and in the remaining columns the design parameters used to specify the filter.

Listing 11.4: Designing two FIR Filtes

```
1 R1     = 5;
2 Fp     = 0.2;
3 Fst    = [0.21 0.225];
4 Ast    = 60;
5 Ap     = 0.25;
6 HdFIR  = decFIR(R1, Fp, Fst, Ap, Ast, coefDir, plotDir);
```

Listing 11.5: Cell Array with Two FIR Filters

```
>> Hd

Hd =

  2×6 cell array

    [1×1 dsp.FIRDecimator]    [0.2000]    [0.2500]    [0.2100]    [60]    [5]
    [1×1 dsp.FIRDecimator]    [0.2000]    [0.2500]    [0.2250]    [60]    [5]
```

Designing a CIC filter uses slightly different parameters, but is otherwise similar. One point of note is that the compensator's passband edge is specified relative to the CIC filter's incoming sampling rate, not the sampling rate at which the compensator runs, as is common for FIR filters otherwise. Hence `FpComp = 1/R2` on line 8 of Listing 11.6.

The `HdComp` cell array will look similar to the cell array from the FIR design function above. However, it will have both the cascade of the CIC filter and its compensator, as well as each filter by itself, in a separate column entry.

Listing 11.6: Designing a CIC Filter and Its Compensator

```
1 R2      = 32;
2 RCIC    = 8;
3 AstCIC  = 60;
4 FpCIC   = 1/R2;
5 DL      = 1;
6 HdCIC   = decCIC(RCIC,  FpCIC, AstCIC, DL, plotDir);
7 RComp   = 4;
8 FpComp  = 1/R2;
9 FstComp = 1/R2 * 1.1;
10 ApComp  = 0.25;
11 AstComp = 60;
12 HdComp  = compCIC(RComp, FpComp, FstComp, ApComp, AstComp, DL, ...
13                   HdCIC, coefDir, plotDir);
```

Cascading two filters with the `cascador` or `parcascador` functions is shown in Listing 11.7. It accepts filter cell arrays as returned by the filter design functions (so, `Hd` from Listing 11.4, for example). However, before being given to the `cascador` function, the fil-

ters which are to be cascaded must be packaged into a single cell array, called `stages` in Listing 11.7. `Hd1` and `Hd2` are presumed to be of the form of `Hd` from above.

No special care needs to be taken when handling `HdComp` cell arrays to `cascador`[2], despite its first column being a cascade filter object instead of a single filter. `cascador` will notice the difference and unpack the cascade[3] The other parameters handed to `cascador`, such as `R`, `Fst` etc. should correspond to the overall properties of the cascade, and not the individual stages. Since it is not possible to automatically determine some of these properties before actually cascading the filter, the user must manually set these to the correct values. Particularly when cascading cell arrays with multiple filters, some care must be taken in order to ensure that all desired permutations are produced. This is because the `cascador` iterates over these paramters when cascading the filters. They are also used to name the resulting plot files for the cascade.

Listing 11.7: Cascading Two Filter Cell Arrays

```
% ... design filter objects first ...
stages = cell(2,1);
stages{1} = Hd1;
stages{2} = Hd2;
Hcasc = cascador(R, Fp, Fst, Ap, Ast, 1, plotDir, stages);
```

While the above remarks cover the most essential information, they by no means constitute a comprehensive guide. When designing filters, it is therefore highly recommended to consult the `README`s, the `help` of the functions, the code and its comments, as well as the official Matlab documentation.

[2] There is a limitation to the numbers of filters Matlab can chain in a single cascade. This limitation might be relevant when designing long filter chains. However, multiple cascades can themselves be cascaded, so this limitation can be worked around if necessary.

[3] Because cascade objects can themselves be cascaded, this is not strictly necessary. But for the sake of easier understanding and elegance, we unpack cascades when possible.

CHAPTER 12

Server

The server application runs on the ARM Linux and is in charge of shoving data from the FPGA to the network and vice versa. How it can be built and extended is explained in this section.

12.1 Building the server

The server can only be built using the build environment explained in Chapter 10. A simple

```
cd /repo/firmware/arm/server
make
```

should suffice to build all the external dependencies and the binary for the ARM core. The externals can be rebuilt using

```
cd /repo/firmware/arm/server
make external
```

and the server application can be rebuilt after changes using

```
cd /repo/firmware/arm/server
make arm
```

The server application is a one file application and depends on libuWebSockets[**?**] and a headerfile called json.hpp[**?**] which contains the entire JSON library.

There is two important functions for extending the server application: onHttpRequest and onMessage.

12.1.1 onHttpRequest

This callback gets called when the user makes an HTTP request wich is not an *UPGRADE* request. Currently it simply serves the files from the filesystem, but could easily be ex-

tended to outline data or the likes. Returning a correct HTTP Response has to be done manually and not well documented in the library itself.

If the response should outline a *200 OK* status,

```
res->end(const char*, size_t);
```

can be used with a string and a size. The lib then detects that no header is attached to the response and attaches a proper textit200 OK header.

If a custom header should be attached it first has to be attached and then the actual answer like so

```
std::string mime;
char header[128];

mime = std::string("text/css");
std::string content = std::string("Hello World");
int header_length = std::sprintf(
    header,
    "HTTP/1.1 200 OK\r\nContent-Length: %u\r\nContent-Type: %s\r\n\r\n",
    str.size(),
    mime.c_str()
);
res->write(header, header_length);
res->end(str.c_str(), str.size());
```

I am not entirely sure if better handling of this will follow in the future or if it is a performance thing, as the library itself is awesome in structure and tidyness. But docs sadly are very sparse

12.1.2 onMessage

This callback is triggered when a new message is received through the WebSocket. For now in our project this call only handles incomming text messages as those conatin the instructions. Binary messages are simply discarded but could be used at a later time to interface the DAC.

To add any new functionality, the code should be inspected and extended in analogy.

12.2 Instruction Set

The instruction set contains TODO: commands which are listed in the lististings **??** downto **??**.

12.2.1 Forcing a new trigger event

This command forces the logger to finish it's current frame. It still repects the set pre and suf conditions. The server does not automatically send the recorded frame. It has to be requested separately. The argument needs to be always set to "true"

Listing 12.1: Forcing a new trigger event

```
// Request

{
    "forceTrigger": "true"
}

// Response

{
    "response", {
        {"request", "forceTrigger"},
        {"status", status}, // "error" or "ok"
        {"error", errorMessage}
    }
}
```

12.2.2 Configuring the frame the server sends

This command tells the server how big the frame should be and how many samples have to be recorded before and after the trigger. All arguments have to be numbers in the JSON format, not strings.

Listing 12.2: Configuring the frame the server sends

```
// Request

{
    "frameConfiguration": {
        "frameSize": frameSize,
        "pre": minSamplesBeforeTrigger,
        "suf": minSamplesAfterTrigger
    }
}

// Response

{
    "response", {
        {"request", "frameConfiguration"},
        {"status", status}, // "error" or "ok"
        {"error", errorMessage}
    }
}
```

12.2.3 Setting the numbers of logged channels

This command tells the server how many channels are being logged. This should always be two as the STEMlab does not support more channels. The logger itself would support

up to 8 channels. The argument has to be a number in the JSON format, not strings.

Listing 12.3: Setting the numbers of logged channels

```
// Request

{
    "setNumberOfChannels": 2
}

// Response

// Response

{
    "response", {
        {"request", "setNumberOfChannels"},
        {"status", status}, // "error" or "ok"
        {"error", errorMessage}
    }
}
```

12.2.4 Reading the currently stored frame

This command forces the server to send the currently stored frame over the binary channel as soon as the current frame is finished. If the logger is not recording currently the frame will be sent immediately. The *channel* argument has to be a number in the JSON format, not strings.

Listing 12.4: Reading the currently stored frame

```
// Request

{
    "readFrame": true,
    "channel": channelID
}

// Response

{
    "response", {
        {"request", "readFrame"},
        {"status", status}, // "error" or "ok"
        {"error", errorMessage}
    }
}
```

12.2.5 Requesting a new frame and reading it when it is ready

This command forces the server to start a new frame and send it over the binary channel as soon as the frame is finished. The *channel* argument has to be a number in the JSON format, not strings.

Listing 12.5: Requesting a new frame and read it when it is ready

```
1  //Request
2
3  {
4      "requestFrame": true,
5      "channel": channelID
6  }
7
8  // Response
9
10 {
11     "response", {
12         {"request", "requestFrame"},
13         {"status", status}, // "error" or "ok"
14         {"error", errorMessage}
15     }
16 }
```

12.2.6 Setting the sampling rate

This command sets the sampling rate. The argument has to be a number in the JSON format and has to be the samplingrate in Hz

Listing 12.6: Setting the sampling rate

```
1  // Request
2
3  {
4      "samplingRate": samplingRateInHz
5  }
6
7  // Response
8
9  {
10     "response", {
11         {"request", "samplingRate"},
12         {"status", status}, // "error" or "ok"
13         {"error", errorMessage}
14     }
15 }
```

12.2.7 Polling the status of the logger

This command requests the current logger status. The response contains all the information the logger currently holds.

Listing 12.7: Polling the status of the logger

```
{
    "status": "true"
}

// Response

{
    "response": {
        "request": "forceTrigger",
        "status": status, // "error" or "ok"
        "error": errorMessage,
        "data":{
            {
                {"memorySize", memeorySizeInBytes},
                {"baseAddress", physicalBaseAddress},
                {"currentAddress", currentPhysicalAddress},
                {"pre", pre},
                {"suf", suf},
                {"numberOfChannels", numberOfChannels},
                {"started", hasLoggerStarted},
                {"IRQack", wasIRQAckedActiveLow},
                {"errorCode", errorCode},
                {"faultyAddress", faultyPhysicalAddress},
                {"testMode", isTestModeActive},
                {"numberOfSamples", numberOfRecordedSamples},
                {"numberOfSamplesTimes", numberOfRecordedSamplesTimesFull},
                {"decimationRate", decimationRate},
            }
        }
    }
}
```

12.2.8 Configuring the trigger

This command configures the currently active trigger. For now, only rising edge triggers are supported on the server side. The logger would support way more trigger types (For more information read the logger code). Channel is the channel that should be triggered on. Level is the level on which the trigger should shoot and slope is the minimal slope the curve needs to have. Hysteresis is configured for all triggers at once and makes sure there is no accidential trigger. It is the variance the signal can have until a trigger gets armed.

Listing 12.8: Configuring the trigger

```
// Request

{
    "triggerOn": {
        "type": "risingEdge",
        "channel": channel,
        "level": levelConvertedToUnsigned,
        "slope": minimalSlope,
        "hysteresis": hysteresis
    }
}

// Response

{
    "response", {
        {"request", "triggerOn"},
        {"status", status}, // "error" or "ok"
        {"error", errorMessage}
    }
}
```

CHAPTER 13

Scope

13.1 Build environment

To set up the scope project the build box is not needed, even tho the build box already has the necessary environment installed. But for development a local install is advised.

To start off, install yarn and nodejs. Package managers for Unix systems already have packages, for Windows the official installers can be used. For the project nodejs 8.0 or higher is used.

Once yarn is set up, the project dependencies can be installed using

```
cd ~/repo/scope/
yarn install
```

The yarn project comes with two main build configuration. One for deployment and one for running a debugging webserver. Building is done with

```
cd ~/repo/scope/
yarn build
```

This leaves all the builded files in */repo/scope/build* and can be copied to any directory served by a webserver.

Running the debugging webserver brings the benefit of autorebuild and autoreload in the browser when the build is complete. The webserver can be started using

```
cd ~/repo/scope/
yarn watch
```

The webpage can now be reached on *http://localhost:8080* and will be automatically reloaded yarn detecs a change and rebuilds the bundle.

Part III

User Guide

CHAPTER 14

Setup

Assuming the STEMlab board at hand was delivered with an SD card containing the correct Linux image, the setup only requires the following steps:

1. Insert the SD card coming with the board (NOTE: Check that the SD card has the right side up to make pin contact).
2. Connect the board physically to the network.
3. Connect the board to the power supply.
4. Call the boards IP and the right port (eg. https://10.84.130.54:50090) in a browser of preference (for best results use Chrome 61.0 and above). Figure 14.2 shows an example how to do this.
5. A notice that a popup has been blocked should appear. Select "Always allow popups from this application." or similar and reload the previously called page. Figure 14.3 illustrate the necessary setting.
6. A popup tab should now contain the scope and automatically connect to the STEMlab's webserver. Figure 14.4 shows a running scope.

If it is unknown wether the SD card was delivered with a prebuilt image, it can be assumed that it was so and the board can be powered. If the LED farthest away from the ports flashes orange with 2 Hz, the SD card is fine.

If the SD card does not contain a prebuilt image, the Devguide should be consulted in Section **??** on how to acquire or build an image.

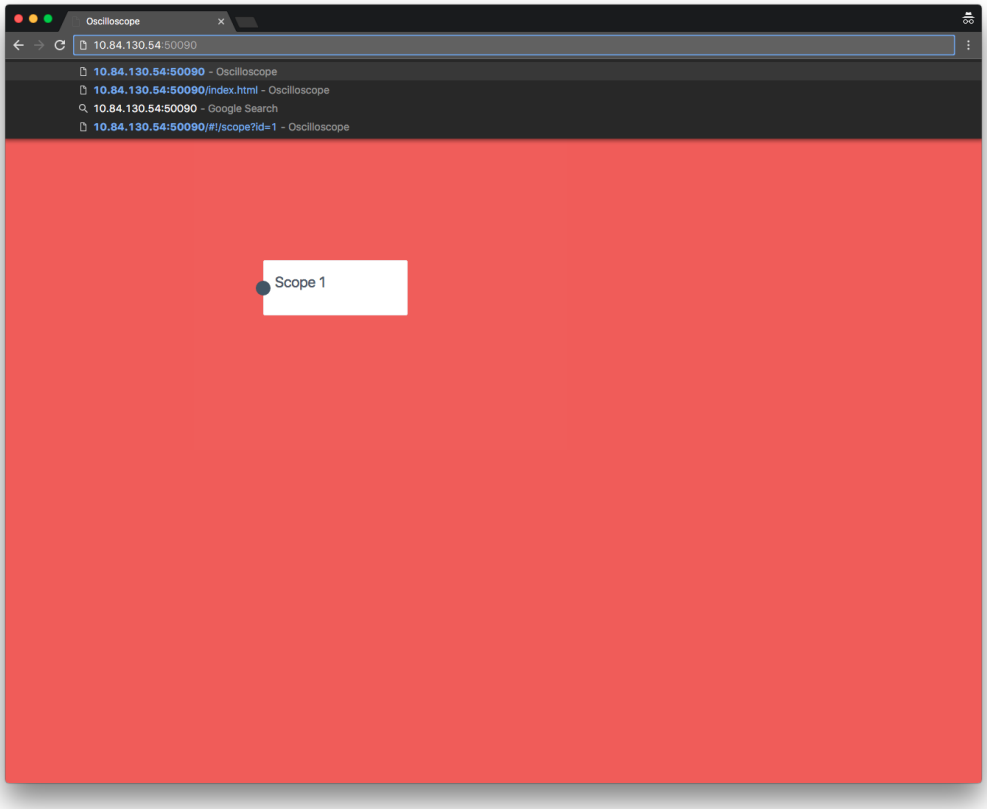


Figure 14.1: Using a browser and the correct URL to run the scope application on the STEMlab.

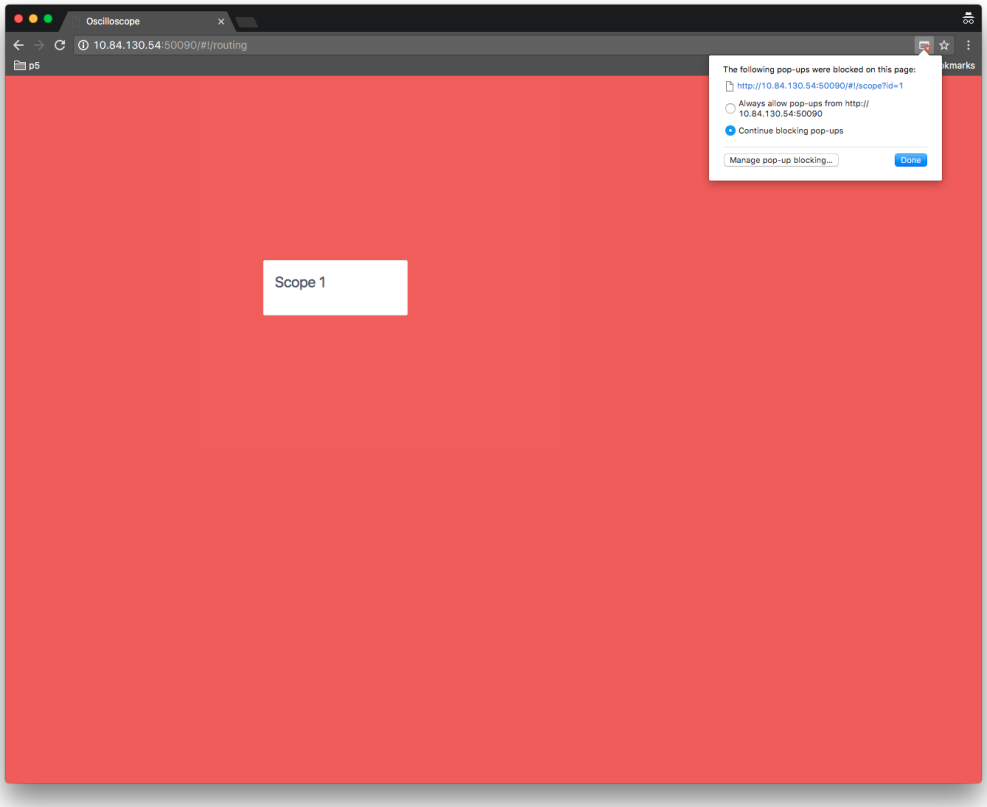


Figure 14.2: The browser warns about a popup the site has tried to open.

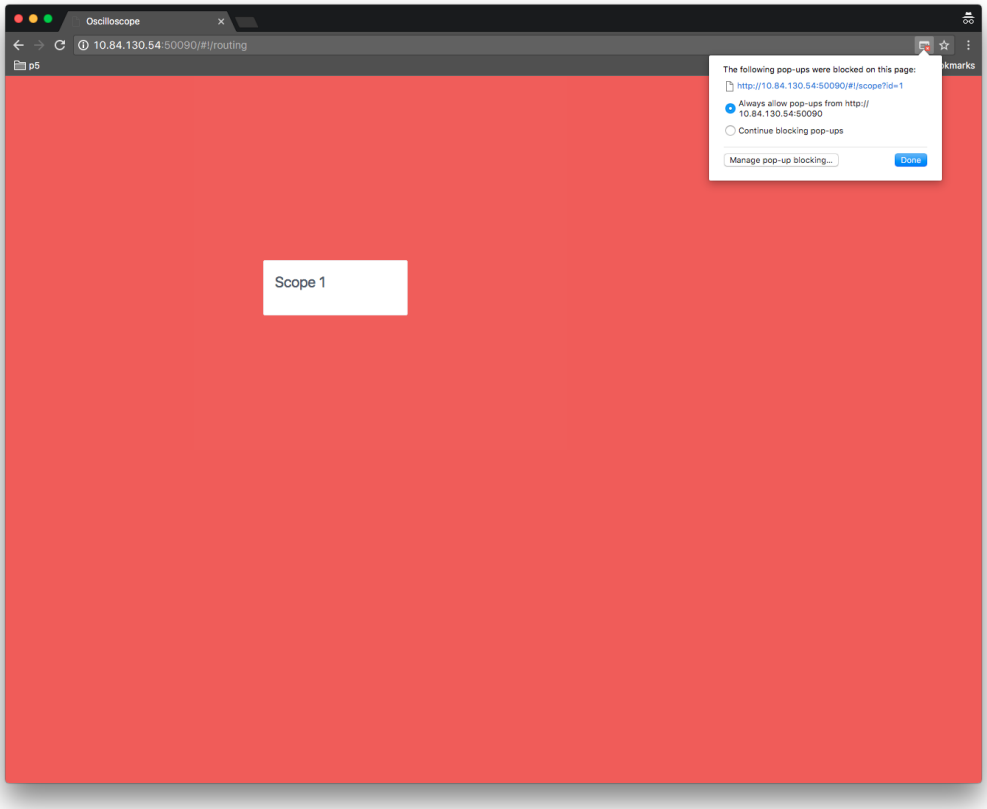


Figure 14.3: Let the browser accept popups in the future.

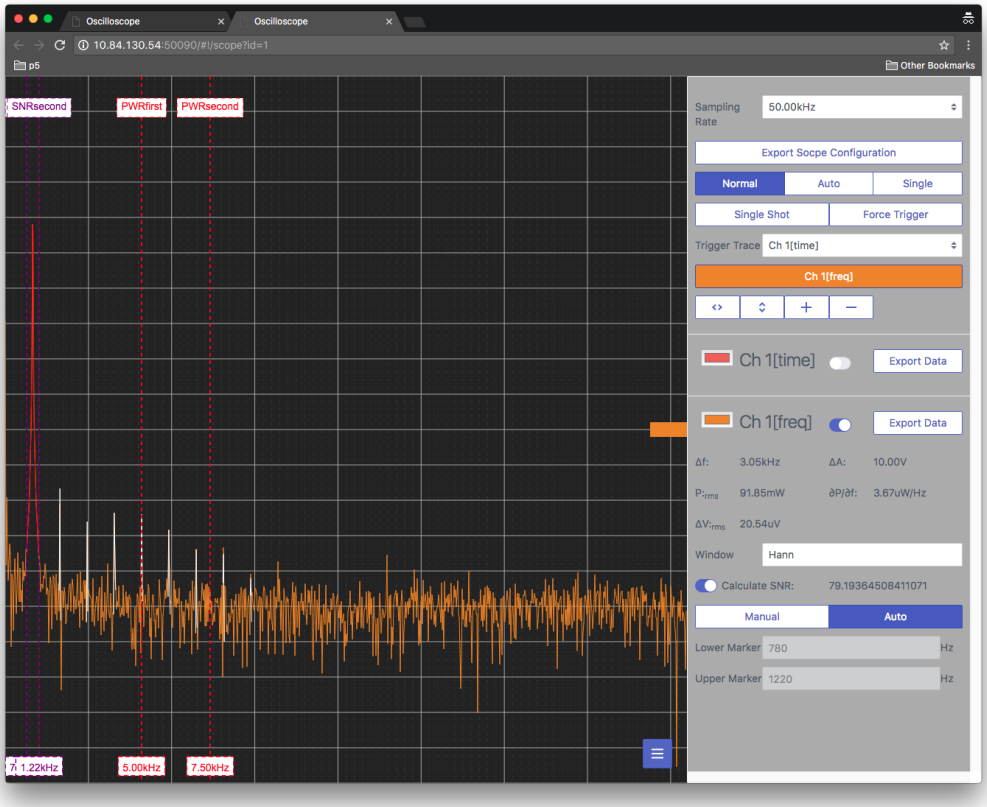


Figure 14.4: The running scope after everything was set up properly.

CHAPTER 15

Operation

Appendices

APPENDIX A

Theoretical Background

A.1 Internal Behavior of a CIC Filter

This section presents an example for a very simple CIC filter to better understand its internal workings. For verification, the filter is also implemented in a Simulink model and simulated.

In the interest of simplicity, we choose a filter with a decimation rate $R = 2$, a differential delay $M = 1$ and $N = 1$ stages. The corresponding topology is shown in Figure A.1; Figure A.2 shows the magnitude frequency response of the filter. As can be clearly seen, this filter would be of very limited use in practice. However, for the purposes of this example, we will feed a DC signal (a constant) into the filter, so the only thing of importance is the filter's DC gain (which is 6 dB, or 2).

Lastly, we will restrict numerical accuracy to three bits in two's complement; the entire range of representable values can be found in Figure A.3. This will limit the number of steps which need to be calculated to gain the desired insight into the filter's mathematical mechanics.

The state of the filter can be calculated by the formulae given in Equations A.1 and A.2:

$$N = 1 \quad M = 1 \quad R = 2$$

$$OUT_{INT}[n] = IN_{COMB}[n] = IN[n] + OUT[n-1] \tag{A.1}$$

$$\begin{aligned} OUT_{COMB}[n] &= IN_{COMB}[n] - IN_{COMB}[n - R \cdot M] \\ &= OUT_{INT}[n] - OUT_{INT}[n-2] \end{aligned} \tag{A.2}$$

The input of the filter shall be a constant of 1, starting at time zero. Once this input is applied to the system, the integrator stage will begin to accumulate the constant. Given an unlimited number of digits (bits), the integrator would in theory reach infinity if it kept

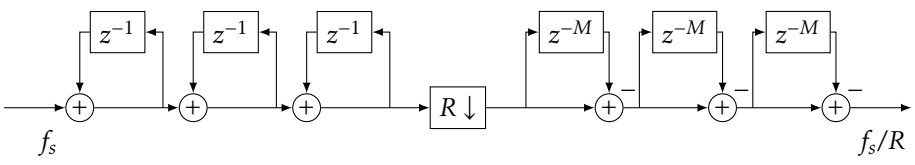


Figure A.1: Topology of the CIC filter for this example

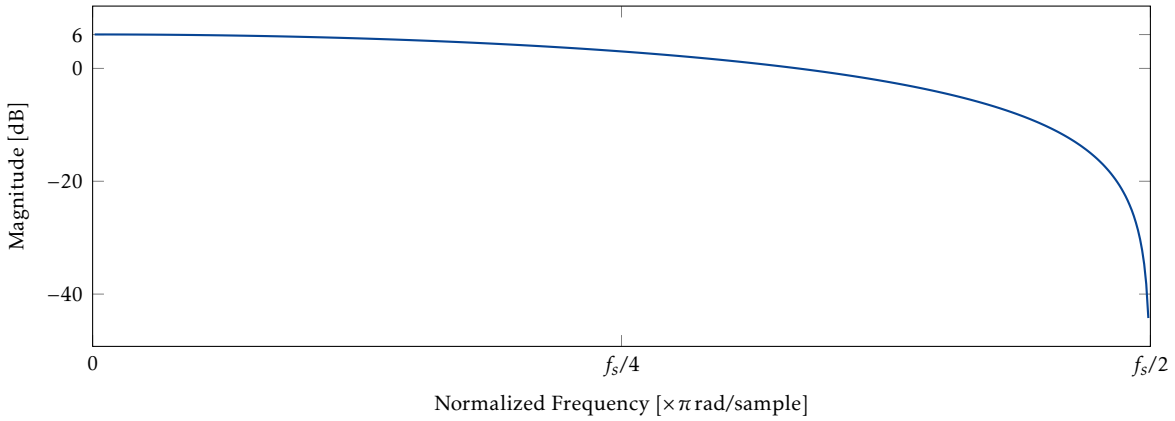


Figure A.2: Frequency response of a CIC filter with $N = 1$, $M = 1$, $R = 2$. Note the DC gain of 2.

running forever. In practice, however, it wraps around once it has reached its maximum representable value (`011` = 3) and begins counting from its lower numerical limit (`100` = −4) again. This cycle keeps repeating as long as the filter is running.

The ingenuinty of the CIC filter lies in exploiting the fact that this wraparound is irrelevant to the comb stage. Whether the comb stage calculates the difference between an integrator's value whose precision is unbounded or whether it calculates the difference between two values which have potentially been wrapped is without consequence.

As a demonstration of this effect, we shall examine the computation step of cycle $n = 4$ from Table A.1 (which contains the entire filter's state for 15 steps). At this point, the state of the filter's output is as follows (represented in three-bit two's complement and decimal):

$$\begin{aligned} OUT_{COMB}[4]_\mathrm{b} &= OUT_{INT}[4] - OUT_{INT}[2] \\ &= 101_\mathrm{b} - 011_\mathrm{b} \\ &= 010_\mathrm{b} \end{aligned} \tag{A.3}$$

$$\begin{aligned} OUT_{COMB}[4]_\mathrm{d} &= -3_\mathrm{d} - 3_\mathrm{d} \\ &= -6_\mathrm{d} \end{aligned} \tag{A.4}$$

Obviously, the decimal and binary results do not match. This is where the wraparound comes into play, for which we shall look at Figure A.3. The figure presents a circular arrangement for all numbers in two's complement with three digits precision. In that arrangement, addition of a positive number corresponds to moving clockwise through the circle, while subtraction of a positive number means moving counterclockwise. Doing this for the calculation of Equation A.4, we find that moving by three in the counterclockwise direction lands us at 2, exactly as Equation A.3 demands. The computation has *wrapped around* its boundary (−4).

What is left to verify is that the result of Equation A.3 is indeed the correct result, i.e. if an unbounded integrator and comb would have yielded the same outcome. And indeed, they would have:

$$OUT_{COMB}[6] = 7 - 5 = 2 \tag{A.5}$$

As a last step to confirm our results, the CIC filter of this exercise is simulated with Simulink. Its block design is given in Figure A.4. All blocks are set to two's complement with three digits of precision (no fractional bits).

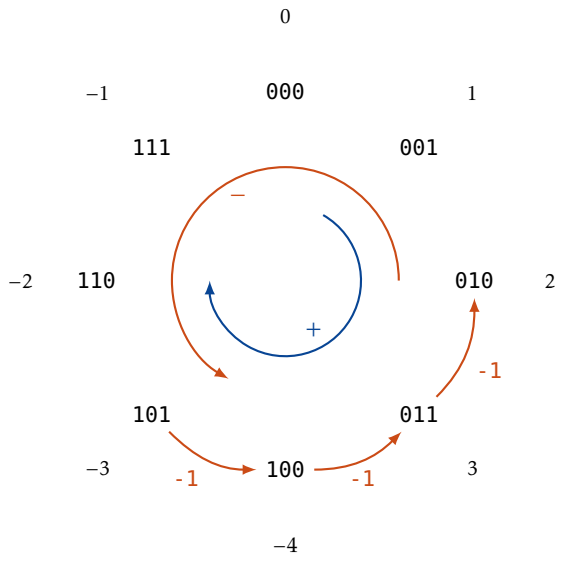


Figure A.3: Subtracting 3 from −3 in two's complement with three digits precicion, represented on a circle. Addition of a positive number corresponds to moving clockwise, subtraction of a positive number corresponds to moving counterclockwise.

Table A.1: Binary and decimal values for the different filter elements during various stages of the filtering process. As expected due to the filter's DC gain of 2, its output is indeed 2. The two right columns contain the calculations as they would occur if the filter's components had unbounded precision. It can be seen that the wraparound effect of the two's complement representation does indeed not change the filter's output.

Cycle	IN	OUT_{INT} IN_{COMB}	OUT	OUT_{INT} (unbounded)	OUT (bounded integrator)	OUT (unbounded integrator)
-2	000	000	000	0		
-1	000	000	000	0		
0	001	001	001	1	$1-0=1$	$1-0=1$
1	001	010		2		
2	001	011	010	3	$3-1=2$	$3-1=2$
3	001	100		4		
4	001	101	010	5	$-3-3=-6=2_{wr}$	$5-3=2$
5	001	110		6		
6	001	111	010	7	$-1-(-3)=2$	$7-5=2$
7	001	000		8		
8	001	001	010	9	$1-(-1)=2$	$9-7=2$
9	001	010		10		
10	001	011	010	11	$3-1=2$	$11-9=2$
11	001	100		12		
12	001	101	010	13	$-3-3=-6=2_{wr}$	$13-11=2$
13	001	110		14		
14	001	111	010	15	$-1-(-3)=2$	$15-13=2$

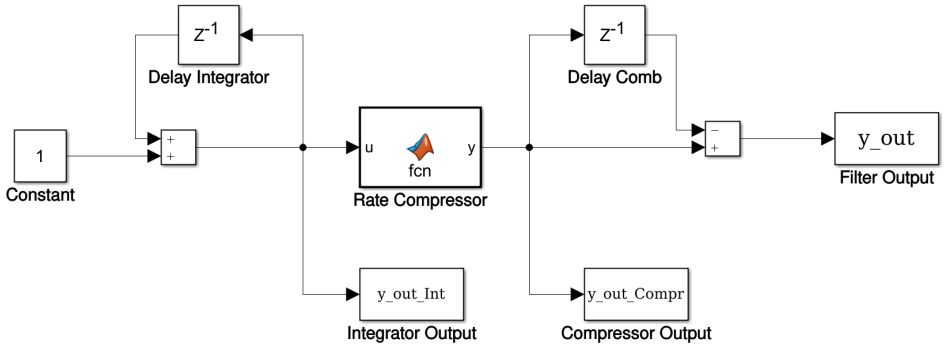


Figure A.4: Simulink model for the filter in Figure A.1. The Rate Compressor is a simple Matlab function which returns every second value of its input vector.

Table A.2: The same CIC filter as before stimulated with an input of 2. A comparison between the right two columns shows that the output of the bounded filter and its unbounded counterpart no longer match starting with the output at $n = 2$; the filter produces a false output.

Cycle	IN	OUT_{INT} IN_{COMB}	OUT	OUT_{INT} (unbounded)	OUT (bounded integrator)	OUT (unbounded integrator)
-2	000	000	000	0		
-1	000	000	000	0		
0	010	010	010	2	$2-0=2$	$2-0=2$
1	010	100		4		
2	010	110	100	6	$-2-2=-4$	$6-2=4$
3	010	000		8		
4	010	010	100	10	$2-(-2)=-4$	$10-6=4$

The simulation results are given in Figure A.5. As can be seen, the filter states are identical to our manually calculated example. The effect of the integrator's output wrapping around the numerical boundaries is also nicely visible.

TODO: Give file path

Lastly, it is shown what happens when the expected output of the filter exceeds the numerical range available. This is accomplished by feeding a constant of 2 into the filter; the expected output is therefore 4.

Because 4 is not within the range of a three-digit two's complement number system, the filter wraps around and produces an incorrect output, -4. The first few calculation steps for this are presented in Table A.2; running the above simulation with the modified input also comfirms this result.

A.2 CIC Filter Tables

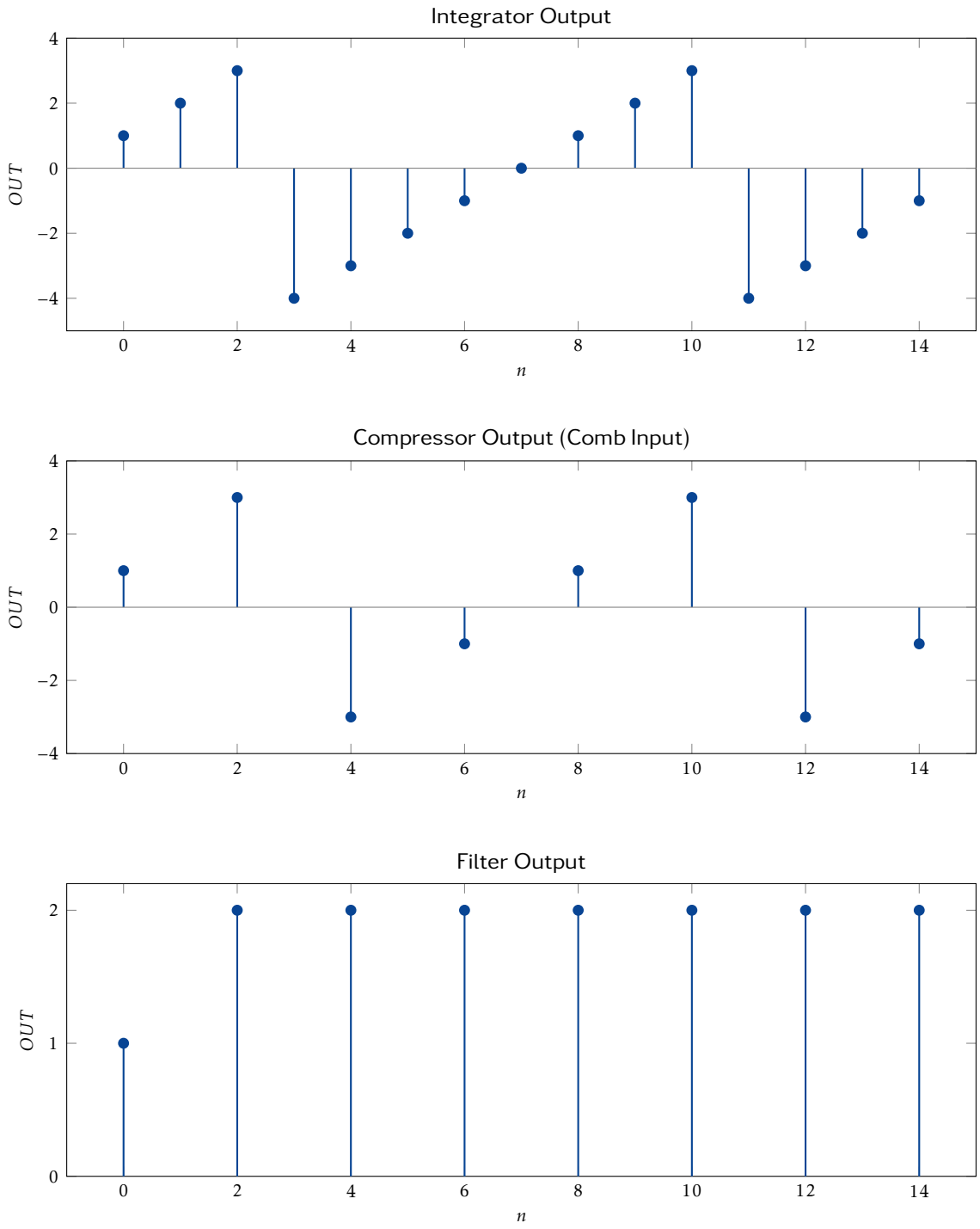


Figure A.5: Simulation results from Simulink for the filter in Figure A.4.

Table A.3: Passband attenuation for CIC filters as a function of the bandwidth-differential delay product. Taken directly from [2].

Relative Bandwidth-Differential Delay Product (Mf_c)	Passband attenuation at f_c in dB as a Function of Number of Stages (N)					
	1	2	3	4	5	6
1/128	0.00	0.00	0.00	0.00	0.00	0.01
1/64	0.00	0.01	0.01	0.01	0.02	0.02
1/32	0.01	0.03	0.04	0.06	0.07	0.08
1/16	0.06	0.11	0.17	0.22	0.28	0.34
1/8	0.22	0.45	0.67	0.90	1.12	1.35
1/4	0.91	1.82	2.74	3.65	4.56	5.47

Table A.4: Passband aliasing attenuation for CIC filters as a function of the bandwidth and the differential delay. Taken directly from [2].

Differential Delay (M)	Relative Bandwidth (f_c)	Aliasing/Imaging Attenuation at $f_{s,low}$ in dB as a Function of Number of Stages (N)					
		1	2	3	4	5	6
1	1/128	42.1	84.2	126.2	168.3	210.4	252.5
1	1/64	36.0	72.0	108.0	144.0	180.0	215.9
1	1/32	29.8	59.7	89.5	119.4	149.2	179.0
1	1/16	23.6	47.2	70.7	94.3	117.9	141.5
1	1/8	17.1	34.3	51.4	68.5	85.6	102.8
1	1/4	10.5	20.9	31.4	41.8	52.3	62.7
2	1/256	48.1	96.3	144.4	192.5	240.7	288.8
2	1/128	42.1	84.2	126.2	168.3	210.4	252.5
2	1/64	36.0	72.0	108.0	144.0	180.0	216.0
2	1/32	29.9	59.8	89.6	119.5	149.4	179.3
2	1/16	23.7	47.5	71.2	95.0	118.7	179.3
2	1/8	17.8	35.6	53.4	71.3	89.1	106.9

APPENDIX B

Filter Design

This chapter contains some additional information about the filter design process.

B.1 Decimation of 625: Variants

B.2 Resource Usage for FIR Filters on the FPGA

This section contains the experimental results of how many DSP slices a given FIR filter needs when implemented with Xilinx's FIR Compiler. These figures form the basis to determine reasonable bounds for the FIR filter `5steep` (see Figure 3.1 on page 38). Figure B.2 depicts the results of the measurements, while Table B.1 contains the configuration parameters which were used for the FIR compiler core.

As can be seen in the plot, DSP slice usage rises roughly linearly at these high sampling rates. When using only a single filter, a filter of roughly 760 coefficients is the maximum possible size. Because the STEMlab has two channels, and because other filters are required as well, an upper limit for `5steep` of 250 coefficients is set based on these results. A smaller filter is also acceptable as long as it fulfills the general requirements.

TODO: source FIR compiler documentation

B.3 Halfband Filters

Table B.1: The parameters used to configure the FIR compiler core for the usage measurements from Figure B.2

Parameter	Value
Clock Frequency	125 MHz
Decimation Rate	5
Input Data Width	24 bit
Input Fractional Bits	7
Output Data Width	32 bit
Coefficient Fractional Bits	17

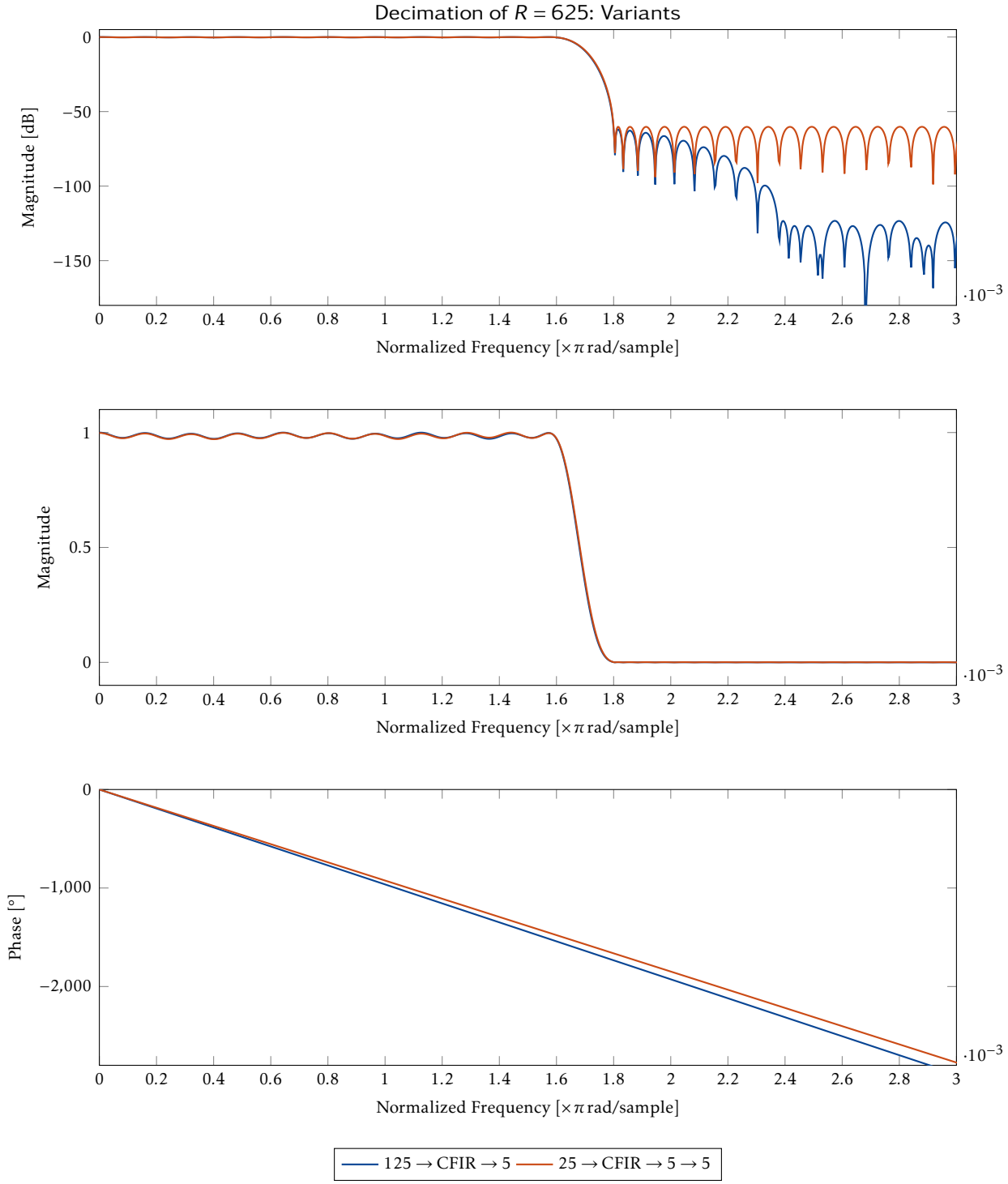


Figure B.1: Semilog (top) and linear plot (middle) for two variants for implementing a decimation chain for a rate change factor of $R = 625$. Both choices show almost the same behavior with regards to magnitude. The bottom plot shows the phase response of the two filters; here, too, the behavior is very similar. *Note:* These plots show the frequency responses of the entire filter cascade for the two respective variants.

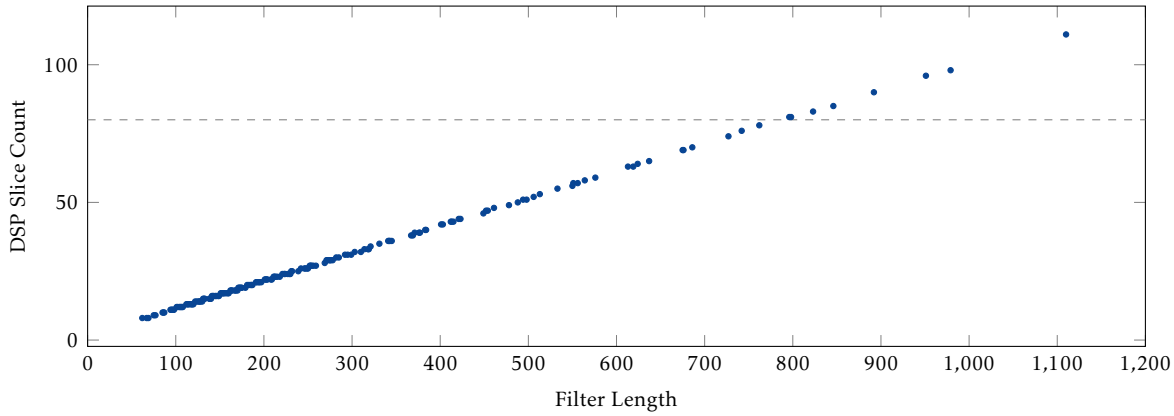
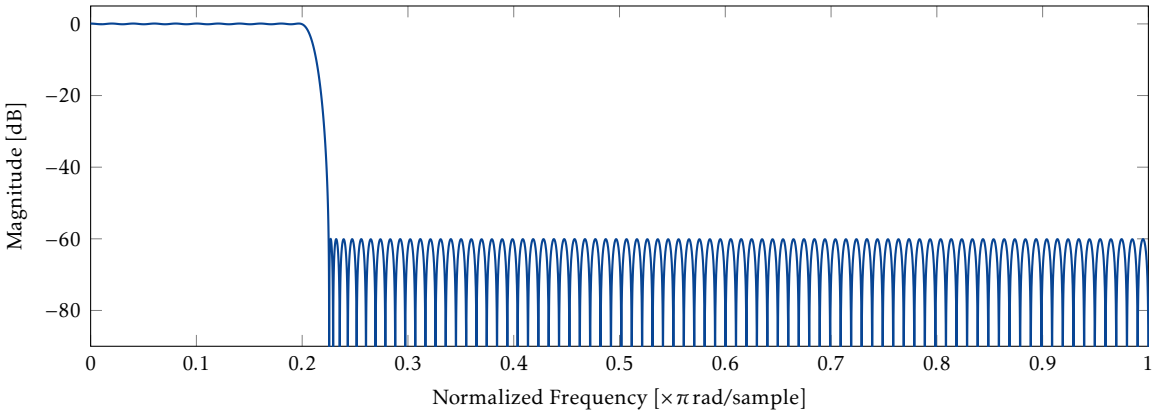


Figure B.2: Usage report figures for DSP slices using the Xilinx FIR compiler block. The configuration of the filter in terms of bith widths is identical to the actual configuration used in the final implementation. Unlike the implementation, however, only a single channel was configured.

B.4 Filter Frequency Responses

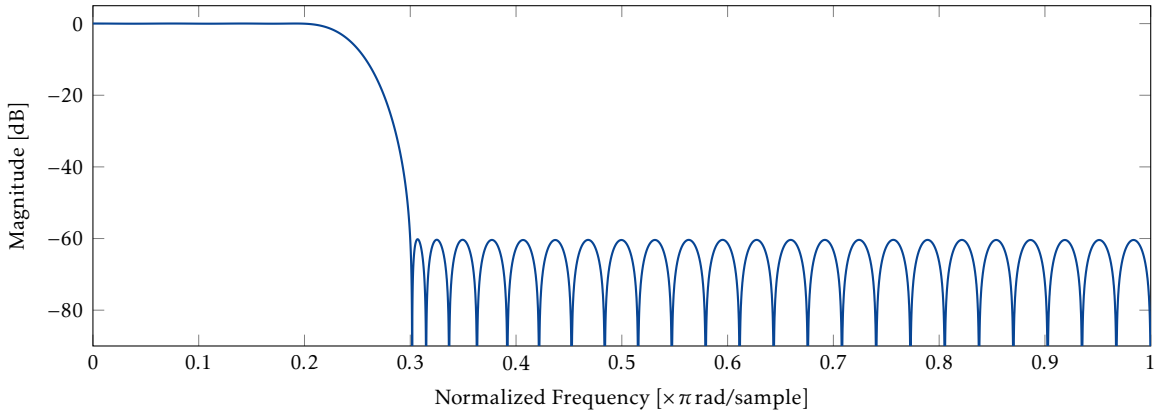
This section contains the frequency responses of the filters and the cascades as specified in Section 3.

B.4.1 `5steep`

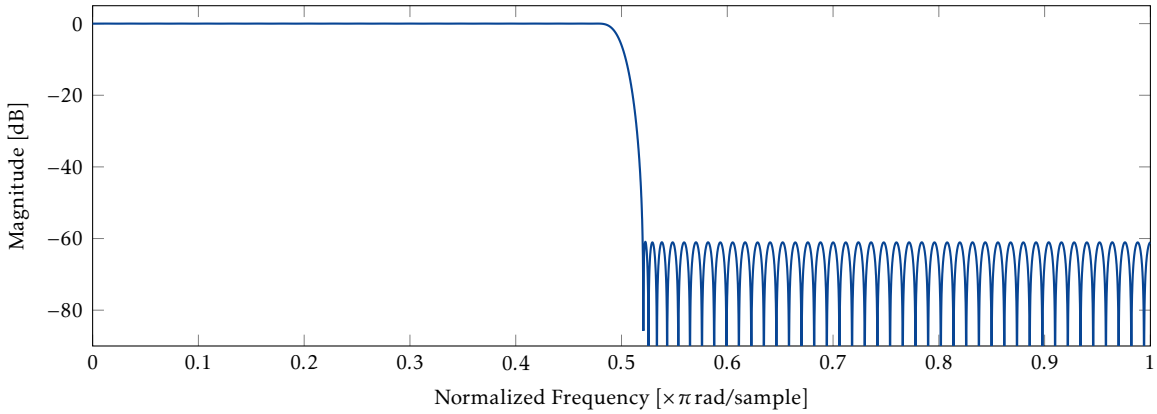


B.4.2 5flat

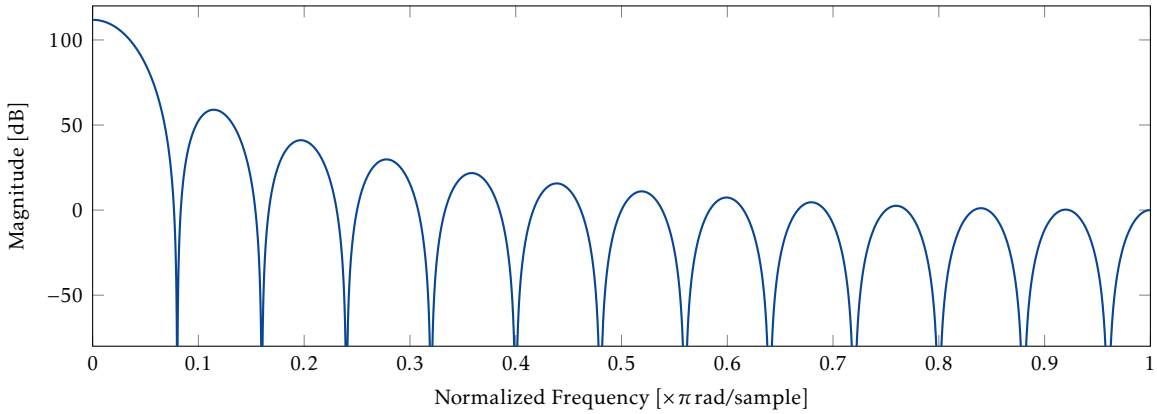
Filter Characteristics	
Filter Length	62
Passband Edge	0.2
3 dB Point	0.234 91
6 dB Point	0.247 08
Stopband Edge	0.3
Passband Ripple (dB)	0.047 745
Stopband Attenuation (dB)	60.1991
Transition Width	0.1
Number of DSP Slices	22



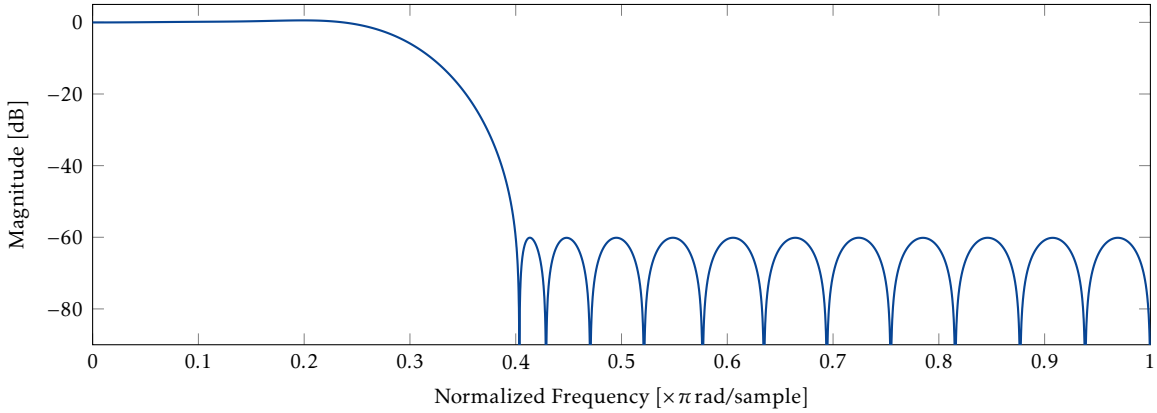
B.4.3 2steep



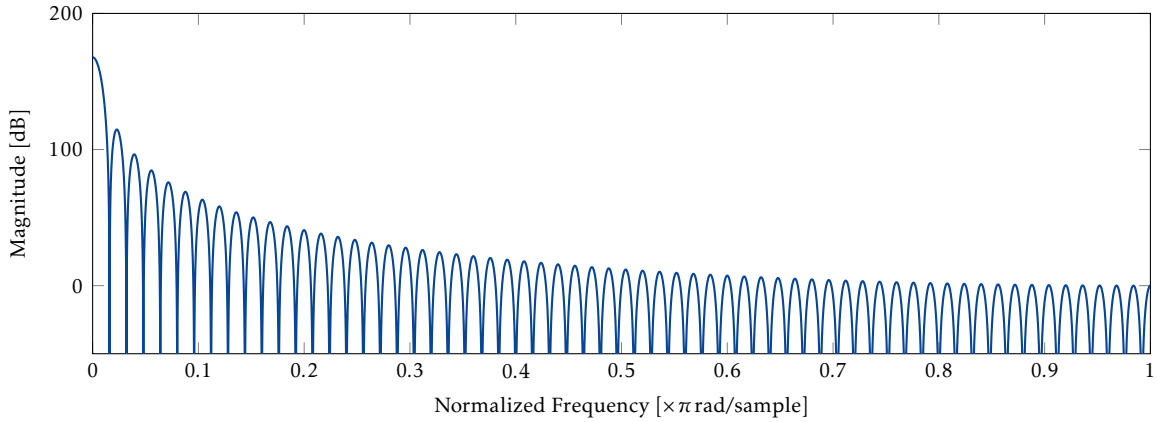
B.4.4 CIC25



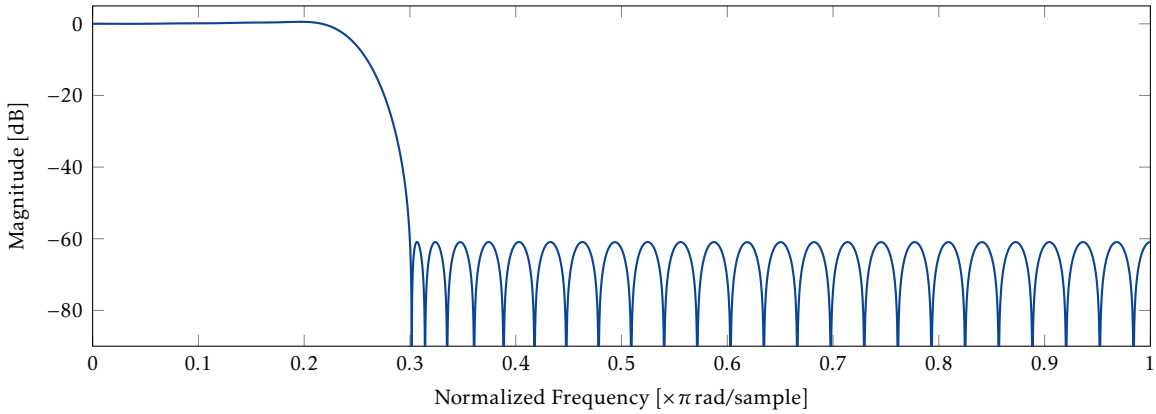
B.4.5 CFIR25



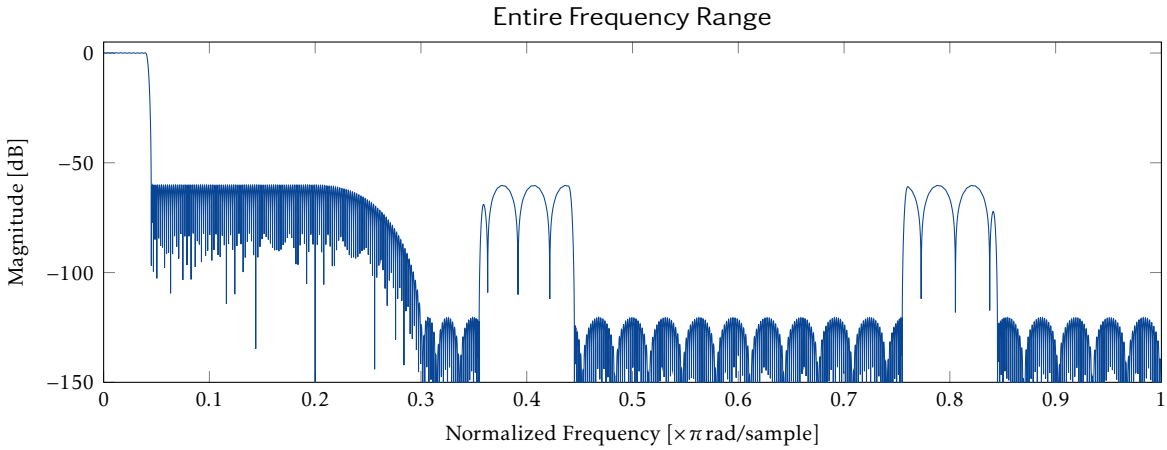
B.4.6 CIC125

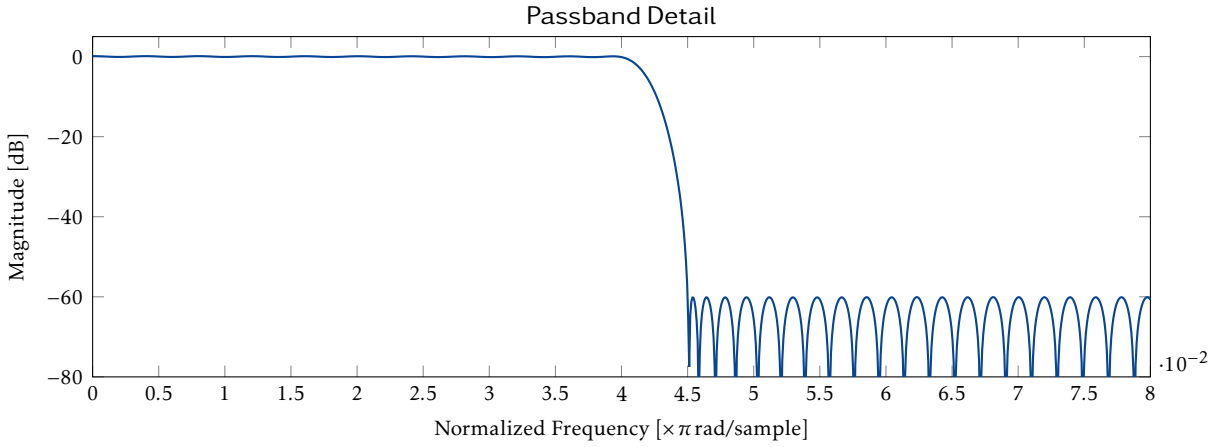


B.4.7 CFIR125

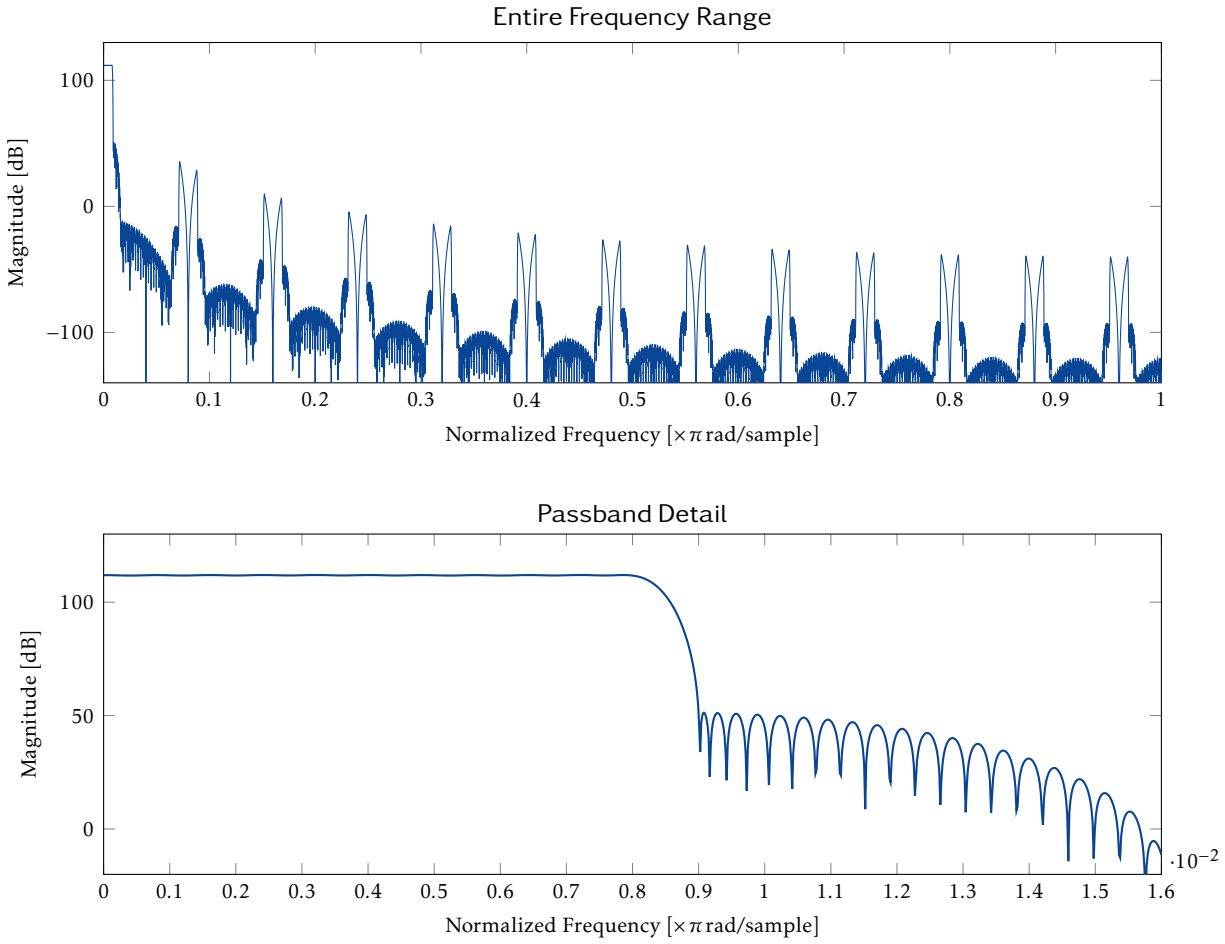


B.4.8 Chain for R=25

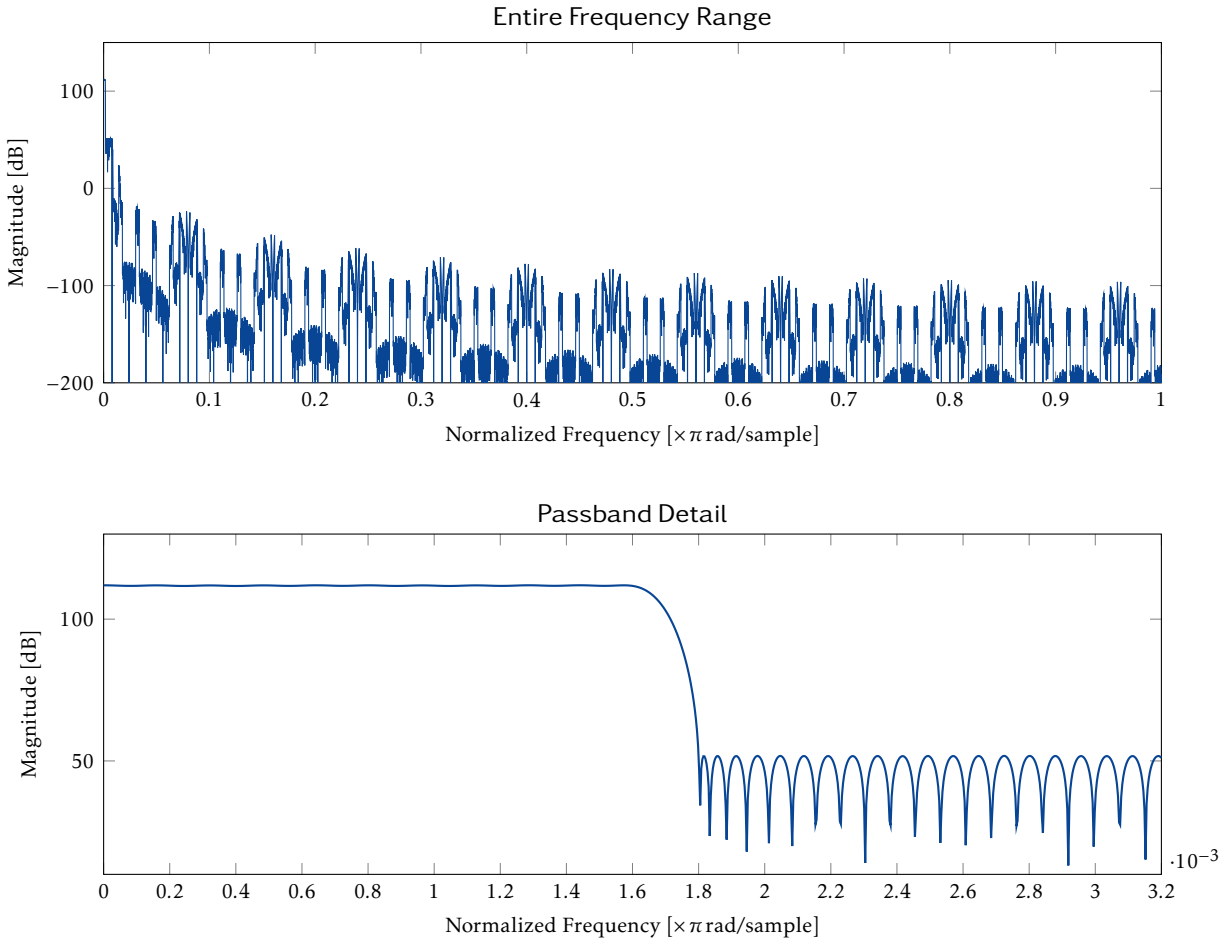




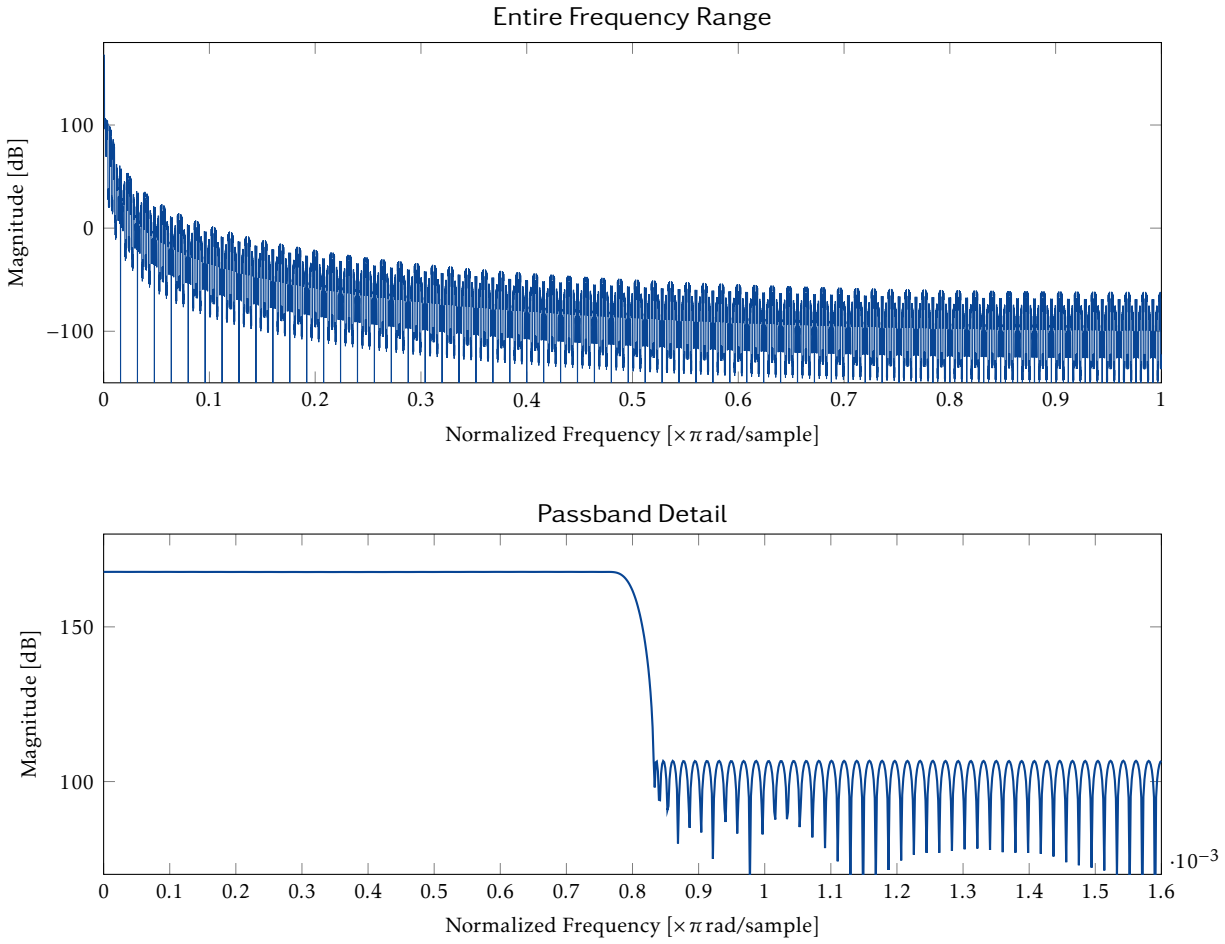
B.4.9 Chain for R=125



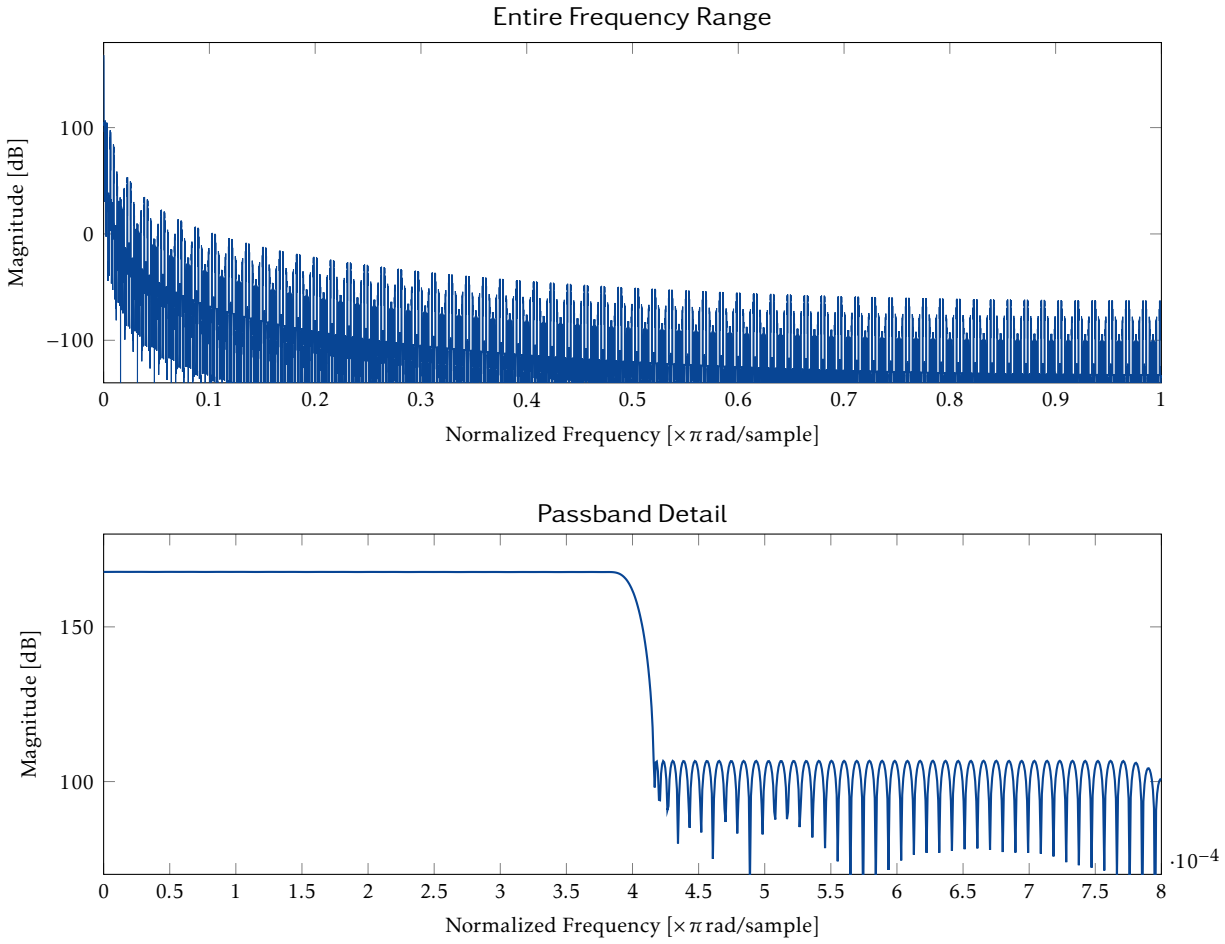
B.4.10 Chain for R=625



B.4.11 Chain for R=1250



B.4.12 Chain for R=2500



APPENDIX C

Oscilloscope

C.1 WebSockets

WebSockets' final RFC 6455[29] was released in December 2011 and is thus still quite young. It is meant to compensate the lack of raw UDP and TCP sockets in JavaScript; while those would offer maximum flexibility, they also pose a significant security risk, and are therefore not available in JavaScript. The WebSockets protocol is located in the Application Layer of the OSI model[1]. Instead of directly opening a raw WebSocket, the handshake is done via HTTP(S). This brings the benefit of communicating through the same ports as the browser (80 or 443) which enables the protocol to function through most firewalls. Furthermore it greatly simplifies the implementation of handshakes for the programmer.

The client sends an upgrade request to the server which then opens a WebSocket connection. This allows for a very conventient way to use TCP Sockets without any entirely new standards. Section 1.5, *Design Philosophy* in RFC 6455 [29] explains it well:

> Basically it is intended to be as close to just exposing raw TCP to script as possible given the constraints of the Web.
>
> The only exception is that WebSockets adds framing to make it packet rather than stream based and to differentiate between binary and text data. This differentiation is very useful for this project. Instructions to the server are issued via the text channel whilst data is sent back through the binary channel, allowing for very convenient interfacing with close to no effort.

In summary: WebSockets are close-to-raw TCP sockets whose handle is shared through HTTP(S).

JavaScript provides an interface that offers convenient sending and receiving of large amounts of data. As nearly anything in JavaScript this is done using callbacks. There are callbacks which handle connections, messages and errors. The code snippet in Listing C.1 gives some insight how WebSockets in JavaScript are used. For more detailled information, the reader is referred to the Mozilla documentation [31].

[1] For those not familiar with the OSI model, Wikipedia provides a good overview in [30].

Listing C.1: Using WebSockets in JavaScript

```
// Open a new socket
this.socket = new WebSocket('ws://localhost');
// Make sure the binary data transmitted
// is interpreted as an ArrayBuffer
// More on ArrayBuffers and Blobs in:
// - https://developer.mozilla.org/en/docs/Web/API/Blob
// - https://developer.mozilla.org/en-US/docs/Web/JavaScript/
//           Reference/Global_Objects/ArrayBuffer
this.socket.binaryType = 'arraybuffer';

// Define all the callback handlers
connection.onopen = function () {
    // The connection was established; send some regards.
    connection.send('Hello World!');
};

connection.onerror = function (error) {
    // An error has occurred; print it to the console.
    console.log('WebSocket Error: ' + error);
};

connection.onmessage = function (e) {
    if (typeof e.data == 'string') {
        // If a text type message was received, print it out.
        console.log('Text message received: ' + e.data);
    } else {
        // A binary type message was received.
        // Interpret the values as 16 bit uints.
        var arr = new Uint16Array(e.data);
        // Plot the data.
        plot(arr);
    }
};
```

C.2 State Tree of Oscilloscope

Listing C.2: The state tree of the scope application

```
var appState = {
    scopes: [{
        ui: {
            prefPane: {
                open: true,
                width: 400,
            }
        },
```

Listing C.2(cont.): The state tree of the scope application

```
        source: {
            id: 2,
            name: 'Source ' + 1,
            location: 'ws://localhost:50090',
            frameSize: 4096,
            samplingRate: 5000000,
            bits: 16,
            vpp: 2.1, // Volts per bit
            trigger: {
                type: 'risingEdge',
                level: 32768,
                channel: 1,
                hysteresis: 30,
                slope: 0
            },
            triggerTrace: 0,
            triggerPosition: 1 / 8,
            numberOfChannels: 2,
            mode: 'normal',
            activeTrace: 0,
            traces: [
                {
                    id: 4,
                    offset: { x: 0, y: 0 },
                    windowFunction: 'hann',
                    halfSpectrum: true,
                    SNRmode: 'auto',
                    info: {}, // Populated during runtime with math
                    name: 'Trace ' + 2,
                    channelID: 1,
                    type: 'FFTrace',
                    color: '#E8830C',
                    scaling: { x: 1, y: 1 },
                    markers: [
                        {
                            id: 'SNRfirst',
                            type: 'vertical',
                            x: 0,
                            dashed: true,
                            color: 'purple',
                            active: true,
                        }
                    ]
                }
            ],
        }
    }]
};
```

C.3 mithril.js

The official mithril webpage describes mithril.js in the following way: "Mithril is a modern client-side Javascript framework for building Single Page Applications. It's small (< 8kb gzip), fast and provides routing and XHR utilities out of the box." [32]

Mithril, like a lot of other frameworks such as React, Angular.js or Vue.js, uses a virtual DOM. This means that it does not modify the DOM which is outlined by the browser, but rather maintains its own DOM. When a new render call is issued, the virtual DOM calculates all the deltas that stem from new content and applies them to the real DOM. This allows mithril.js to calculate and recalculate the DOM based on a descriptive model. The developer does not have to manually modify an object's state but rather has to describe it.

A redraw generally happens when an event is triggered by any input element but can also be issued manually. A virtual DOM consists of many vnodes (virtual nodes) and can be mounted on any actual node of the browser's DOM as the example in Listing C.3 shows.

Listing C.3: Basic creation and usage of mithril components in JavaScript

```
1 // A mithril component is a simple object that has at minimum a
2 // view() function that returns a vnode.
3 var HelloWorld = {
4     view: function() {
5         // Return the toplevel <div> vnode
6         return m('', [
7             // Create a <h1> vnode with the attribute class="title"
8             m('h1', { class: 'title' }, 'A very interesting title!'),
9             // Create a <p> vnode
10            m('p', 'Hello World!'),
11        ])
12    }
13 }
14 // Get the root div and mount the HelloWorld component
15 var root = document.getElementById('root');
16 m.mount(root, HelloWorld)
```

A component can be mounted on any DOM node and becomes a vnode in the virtual DOM. The developer can create new components by simply creating an object that holds at least a `view()` function that instantiates new vnodes. The new component can then be instantiated via the `m()` or `m.mount()` command. As this section should only give a base overview on mithril and is not meant to be a manual, further information on mithril's features and usage can be obtained on it's webpage [32].

C.4 WebGL

An application uses the *canvas* DOM element which provides a direct interface to WebGL. The user can render vertices to the canvas and even apply shaders or, in the case of our scope application, simple 2D geometry calls. These are sufficient for our purposes since the scope basically only requires the drawing of lines.

Via the canvas one can retreive a 2D rendering context on which simple geometry can be drawn. In JavaScript this can be done using the code in Listing C.4 which shows how a single red line can be drawn on the canvas.

Listing C.4: Getting a 2D Rendering Context from a Canvas and Drawing on it in JavaScript

```
1 // Get the canvas element from the dom
2 var canvas = document.getElementById('canvas-id');
3 // Get the 2d context of the canvas
4 var context = canvas.getContext('2d');
5
6 // Set brush color to red
7 context.strokeStyle = '#FF0000';
8
9 // Start a new path and move the cursor
10 // from start to end of the line to be drawn
11 context.beginPath();
12 context.moveTo(x, y);
13 context.lineTo(x + 100, y + 100);
14
15 // Finally actually draw the line on the canvas and end the path
16 context.stroke();
```

There is also the possibility to draw rectangles, circles and much more. All of those elements can be styled easily via properties of the context environment. All the functionality is documented on the Mozilla Network [33].

After having acquired the rendering context, something can be drawn on the canvas once. For the creation of a moving image, those draws have to be re-issued over and over again. There are various possibilities in JavaScript to accomplish this, but only one is actually performant and recommended.

Instead of simply drawing to the canvas over and over again, it would be ideal to only do that before a new frame is pulled from the framebuffer by the display. JavaScript provides a interface to register a callback that is called before a new frame is released. This callback will be called with the same frequency as the display referesh rate, which nowadays usually is 60 Hz. To make sure that a callback will always be executed, it has to be registered again after a callback has been issued. The example in Listing C.5 shows how this is done. This callback will not affect the rest of the DOM. This allows JavaScript to handle the redraws of the DOM with high speed while the callback will will render a fluent graph of the data onto just one of the DOM elements.

Listing C.5: Usage of the requestAnimationFrame callback in JavaScript

```
// The register function is not named the same way in every browser
// Make sure this is the case
window.requestAnimationFrame = window.requestAnimationFrame
                            || window.webkitRequestAnimationFrame;

// Our callback we call for every frame drawn
export const draw = function() {
    // Draw anything needed

    // End draw

    // Register the callback again
    requestAnimationFrame(function(){
        // Execute our callback
        // We cannot hand this directly to the register function
        // since it is not yet known inside it's own definition
        draw();
    });
};

// Initially call the draw function
draw();
```

C.5 FFT Windowing Paramters

Table C.1: FFT windowing parameters, taken from [34]

Window	Scaling Factor for Quasi-Periodical Signals	Attenuation of Largest Side Lobe (dB)	Number of Lines per Bundle	Maximum Error in Amplitude (dB)
Rectangle	1	13	1 – 2	−3.8
Hanning	1/0.5000	31	3 – 4	−1.5
Hamming	1/0.5400	41	3 – 4	−1.6
Blackman	1/0.4200	58	5 – 6	−1.1
Bartlett	1/0.5000	26	3 – 4	−1.9
Kaiser-Bessel	1/0.4021	67	7 – 8	−1.0
Flat-Top	1/0.2155	67	9 – 10	0

APPENDIX D

Licenses

D.1 MIT License (Source: [12])

```
Copyright <YEAR> <COPYRIGHT HOLDER>

Permission is  hereby granted, free of  charge, to any person  obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software  without restriction, including without limitation
the rights to  use, copy, modify, merge,  publish, distribute, sublicense,
and/or sell  copies of  the Software,  and to permit  persons to  whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT  NOT LIMITED TO THE  WARRANTIES OF MERCHANTABILITY,
FITNESS FOR  A PARTICULAR PURPOSE  AND NONINFRINGEMENT. IN NO  EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER  IN AN ACTION  OF CONTRACT, TORT OR  OTHERWISE, ARISING
FROM,  OUT OF  OR IN  CONNECTION WITH  THE SOFTWARE  OR THE  USE OR  OTHER
DEALINGS IN THE SOFTWARE.
```

Bibliography

[1] R. Bucher and P. Kuery, "Front-End Signal-Processing For Red Pitaya Spectrum Analyzer," Jan 2017.

[2] E. Hogenauer, "An economical class of digital filters for decimation and interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 2, pp. 155–162, Apr 1981.

[3] Altera Corporation. (2007, April) Understanding CIC Compensation Filters. [Online]. Available: https://altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an455.pdf

[4] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*, 1st ed., ser. Prentice-Hall signal processing series, A. V. Oppenheim, Ed. Englewood Cliffs, New Jersey 07632: Prentice-Hall, 1983.

[5] (2017, Aug) LTC2145-14 - 14-Bit, 125MSps Low Power Dual ADCs. [Online]. Available: http://www.linear.com/product/LTC2145-14

[6] Red Pitaya. (2017, Aug) Red Pitaya Ecosystem and Applications. [Online]. Available: https://github.com/RedPitaya/RedPitaya

[7] ——. (2017, Aug) Welcome to the Red Pitaya Documentation. [Online]. Available: http://redpitaya.readthedocs.io/en/latest/index.html

[8] Elector International Media BV. Stemlab 125-14 (Starter Kit). [Online]. Available: https://www.elektor.com/stemlab-125-14-starter-kit

[9] M. Ossmann. (2014, Dec) Red Pitaya: Mehr als ein USB-Oszilloskop! [Online]. Available: https://www.elektormagazine.de/articles/red-pitaya-mehr-als-ein-usb-oszilloskop

[10] N. Huesser and I. Jeras. (2017, Mar) Which Vivado project base is to be used for custom filters on the FPGA? [Online]. Available: https://github.com/RedPitaya/RedPitaya/issues/107

[11] P. Demin. (2017, Aug) Red Pitaya Notes. [Online]. Available: http://pavel-demin.github.io/red-pitaya-notes/

[12] The MIT License. [Online]. Available: https://opensource.org/licenses/MIT

[13] X. Inc. (2016, Oct) CIC Compiler v4.0 – LogiCORE IP Product Guide. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/cic_compiler/v4_0/pg140-cic-compiler.pdf

[14] ——. (2015, Nov) FIR Compiler v7.2 – LogiCORE IP Product Guide. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/fir_compiler/v7_2/pg149-fir-compiler.pdf

[15] N. Huesser. (2017, Aug) ZYNQ Logger. [Online]. Available: https://github.com/Yatekii/zynq_logger

[16] Wikipedia. Idempotence. [Online]. Available: https://en.wikipedia.org/wiki/Idempotence

[17] Xilinx Inc. (2016, Nov) Vivado Design Suite Tcl Command Reference Guide. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug835-vivado-tcl-commands.pdf

[18] ——. (2016, Apr) Vivado Design Suite User Guide – Using Tcl Scripting. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug894-vivado-tcl-scripting.pdf

[19] (2017, Aug) Tcl Developer Xchange. [Online]. Available: http://tcl.tk/

[20] R. Fonseca. Introduction to asynchronous programming. [Online]. Available: http://cs.brown.edu/courses/cs168/s12/handouts/async.pdf

[21] R. Gut et al., "Spectrum Analyzer," 2016.

[22] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Compuation*, vol. 19, pp. 297–301, Apr 1965. [Online]. Available: http://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf

[23] K. Kwok. (2015, May) Cooley-tukey fft + dct + idct in under 1k of javascript. [Online]. Available: https://antimatter15.com/2015/05/cooley-tukey-fft-dct-idct-in-under-1k-of-javascript/

[24] ——. (2015, May) Small Cooley-Tukey radix-2 DIT FFT in Javascript. [Online]. Available: https://gist.github.com/antimatter15/0349ca7d479236fdcdbb

[25] H. Schmid, "How to use the fft and matlab's pwelch function for signal and noise simulations and measurements," 8 2012. [Online]. Available: http://www.schmid-werren.ch/hanspeter/publications/2012fftnoise.pdf

[26] Red Hat Inc. (2017, Aug) Ansible Documentation. [Online]. Available: https://docs.ansible.com/

[27] R. Frey and N. Hüsser. (2017, Aug) Vivado Install Guide. [Online]. Available: https://github.com/alpenwasser/pitaya/tree/master/doc/vivado-install

[28] Xilinx Inc. (2017, Aug) Support Portal – Documentation. [Online]. Available: https://www.xilinx.com/support.html#documentation

[29] I. F. . A. Melnikov. (2011, Dec) The WebSocket Protocol. [Online]. Available: https://tools.ietf.org/html/rfc6455

[30] Wikipedia. OSI model. [Online]. Available: https://en.wikipedia.org/wiki/OSI_model

[31] mozilla. WebSockets API. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[32] [Online]. Available: https://mithril.js.org/

[33] mozilla. CanvasRenderingContext2D. [Online]. Available: https://developer.mozilla.org/en/docs/Web/API/CanvasRenderingContext2D

[34] M. Meyer, *Grundlagen der Informationstechnik*, 1st ed. Vieweg, 202.